DEFINITION OF
PROGRAMMING LANGUAGES
USING
TRANSFORMATIONAL SEMANTICS

A thesis submitted for the Degree of
Doctor of Philosophy
of the
Australian National University

Stephen John Edwards
June 1982
I hereby state that this thesis contains only my own original work except where explicit reference has been made to the work of others.

S. J. Edwards
I give my thanks to John Hurst, my supervisor, and Mike Robson, my acting supervisor over the last several months of my course, for their guidance. I also thank Brian Molinari for reading and supplying comments on drafts of this thesis and Robin Stanton for doing this and more. Special thanks go to Chris Johnson, a fellow research student for most of the time, for fruitful discussions and for writing the original document processor on which this thesis was prepared. Finally, I thank my examiners for their suggestions regarding the presentation of this thesis.
ABSTRACT

One method of implementing a high-level language is to translate programs written in it to an intermediate language and interpret the object programs in that intermediate language on the target machine. For an efficient implementation the design of the intermediate language will depend on both the high-level language and the target machine. The idea that an intermediate language suited to the high-level language and a particular target machine can be generated automatically from descriptions of that language and machine is introduced in this thesis. This requires an "interpreter generator" to produce, from the two descriptions, the high-level language compiler producing intermediate language code, and an interpreter for that intermediate language on the target machine. Such a system is discussed in an appendix. This thesis develops a method for describing a high-level language designed for use in the context of an interpreter generator.

This method describes a given language by translating it to a target language with known semantics, that is, by transformational semantics. The translation is defined by a high-level programming language designed for the purpose. The target language is not machine oriented and this reduces the possibility of the description method affecting the perceived structure of the high-level language. Four example definitions, one of a subset of Pascal, are given in
appendices. Emphasis is placed on designing a practical, rather than theoretical, high-level language description method. The proposed way in which the intermediate language might be extracted from such a description is outlined by a number of examples.
# TABLE OF CONTENTS

1 INTRODUCTION.
   1.1 Background.
   1.2 Statement of Thesis.
   1.3 Outline of Thesis Presentation.
   1.4 Statement of Originality.

2 LITERATURE ON RELATED TOPICS.
   2.1 High-Level Language Architecture.
   2.2 Intermediate Language Architectures.
   2.3 Complete Syntax Definition.
   2.4 Translator Writing Systems.
   2.5 Compiler or Parser Generators.
   2.6 Formal Semantic Definitions.
      2.6.1 Axiomatic Semantics.
      2.6.2 Denotational Semantics.
      2.6.3 Operational Semantics.
      2.6.4 Transformational Semantics.
   2.7 Machine-Oriented Low and Intermediate Level Languages.
   2.8 Code Generator Generators.
   2.9 Automatic Program Analysis.
   2.10 Process Synchronization Primitives.

3 SYSTEM FOR THE AUTOMATIC GENERATION OF ARCHITECTURES.
   3.1 The Purpose of SAGA.
      3.1.1 Rosin's Notation for Describing System Implementation.
      3.1.2 Purpose of SAGA Described Using Rosin's Notation.
   3.2 Description of SAGA.
      3.2.1 The Definition of L.
      3.2.2 The Definition of M.
      3.2.3 The Interpreter Generator.
         3.2.3.1 General.
         3.2.3.2 Generation of the Compiler.
         3.2.3.3 Generation of the Interpreter.
APPENDIX ONE. 156
APPENDIX TWO. 162
APPENDIX THREE. 192
APPENDIX FOUR. 227
APPENDIX FIVE. 244
APPENDIX SIX. 252
APPENDIX SEVEN. 306
APPENDIX EIGHT. 310
Chapter 1 - Introduction

1 INTRODUCTION.

1.1 Background.

This thesis was motivated by the arrival of a Burroughs' B1700 computer [BurrC72] in the author's place of work. The architecture of this machine is unusual [WilnW72a] (see sections 2.1 and 2.2). The B1700 was designed to interpret intermediate languages, called S-languages (see section 2.7). When a high-level language is compiled it is translated to an equivalent S-language program; to execute the program the S-language code is interpreted. One S-language is associated with each high-level language implemented on the machine. The converse is usually true, though not always, for example, the same S-language is used for both COBOL [ANSI74] and RPG [IBM65]. The architecture of the B1700 can thus be referred to as an Intermediate Language Architecture (ILA).

The advantages of this approach to high-level language implementation include:

1. the compilers are simpler because the target language is closer to the source language in semantics and the importance of local optimization becomes less.
2. the translated S-language object code module is usually much smaller than the equivalent machine language object code module (see section 2.2, [TaneA78] and [WilnW72b]).
3. A number of different interpreters with different
characteristics for each S-machine allows, for example, for a "production" interpreter which is fast, a "de-bugging" interpreter which is slower but could include tracing, trapping and other diagnostics features, and perhaps another to take run-time statistics for algorithm optimization. The different interpreters could be used in different executions of a program and no re-compilation would be required.

The main disadvantage of this approach is the inefficiency of interpreting the S-language. The designers of the B1700 clearly believed that the advantages outweighed the disadvantages when the machine is designed to cater for S-machine interpretation. For further discussion see [DitzD80].

In order to support the interpretation of S-machines efficiently and to gain the most out of the advantages set out above, the B1700 machine instruction set enables very flexible use of data; the data for the machine is the data and the instructions for the S-machines. This flexibility extends to allowing data to be of any length (up to 64 kbits) and to start at any bit location in memory. Up to 24 bits can be transferred in a single operation to or from the memory and the B1700 instruction set has looping primitives which allow larger bit strings to be transferred easily. Thus S-machine instruction and data fields can be whatever length is most efficient. More detail on the design of the B1700 is given in [BurrC72, WilnW72a, WilnW72b, OrgaE78].
Chapter 1 - Introduction

In a broader context the intermediate level S-languages could be implemented in hardware, perhaps producing a computer something like the B5000 series [LoneW61, BurrC64, OrgaE73], or in firmware, like the Pascal Microengine (R) [WestD79].

1.2 Statement of Thesis.

Normally a large amount of effort is expended in designing the S-language and its associated compiler and interpreter for a new programming language that is to be implemented on the B1700. Consider a system that can produce these three objects automatically. This system can be illustrated thus (extract from figure in section 3.1.2):

Given the definitions of L and a real machine, M, this system will create a suitable intermediate language S and produce a compiler for L, translating L to S, and an interpreter for S for the machine M. This system is called SAGA for System for Automatic Generation of Architecture. The architecture referred to here is the architecture of
Chapter 1 - Introduction

the S-machine. The S-machine produced could not only be used in the context of a B1700 or other ILA, but also in the context of hardware or firmware implementation as just mentioned. The aim of this thesis is to develop a method of defining the high-level language (referred to as L throughout this thesis) in the context of SAGA.

1.2 Statement of Originality

We define L by specifying its translation to a target language with known semantics. The specification of the translation can, in theory, be used as the basis for the generated compiler. It is intended that the interpreter be defined by the constructs of the target language used in the translation. The target language is a machine independent language with high-level control and data structures with which it is intended to preserve the structure of L despite the translation.

1.3 Outline of Thesis Presentation.

This thesis is presented in the following five chapters. Chapter two presents some of the topics related to this thesis and gives references to the literature on these topics. Chapter three gives a plan of SAGA. Chapters four and five explain in detail and supply motivation for the method used to define L. Appendices two and three correspond to these two chapters and describe the languages used therein in a "reference manual" style. Finally, chapter six summarises the work presented in this thesis and discusses some of the ways it is
envisaged that the definition of L will be used for the purpose it was designed. Some further research topics indicated by this thesis are also mentioned.

1.4 Statement of Originality.

The basic concept of SAGA and the two languages used in the definition of L are claimed as the original work of the author. The original features of each language deemed to be of significance are listed at the end of the chapters (four and five) discussing these languages.
2 LITERATURE ON RELATED TOPICS.

The thesis presented in the last chapter draws on a broad range of topics in current computing research including computer architecture, automatic programming and formal semantics. The main areas of background are discussed in the following sections with references to the literature. So that the ideas in this dissertation can be presented in a particular order the following sections are related to the rest of the text by references rather than explicit explanations at this stage. The section numbers in parentheses in each sub-heading below indicate as to where that discussion on background is related in the text.

2.1 High-Level Language Architecture (1.1).

Those architectures designed with the implementation of high-level languages in mind are of interest here. Amongst the various methods used for implementing a high-level language, those which use a computer architecture designed to support that high-level language fall into two categories. Those using a Direct Execution architecture [ChuY75] where the high-level language programs are directly executed by the hardware, and those using a Syntax-oriented or Indirect Execution architecture [ChuY75]. Here the high-level language is translated to an intermediate level language (between high-level
programming languages and low-level computer languages) and then executed by the hardware.

Another approach to implementing a high-level language may or may not use a computer designed to support it. Here the high-level language is translated to an intermediate language, usually also intermediate in level (e.g. [HaynL73]), and then interpreted by a program which is, naturally enough, called an interpreter. Where the computer is designed with this implementation method in mind, its architecture is referred to as an Intermediate Language Architecture (ILA).

Two examples using the last implementation method on machines not specifically designed for the purpose are the implementation of EULER [WirtN66] on an IBM 360/30 [WebeH67] and the portable macro implementation of SNOBOL4 [GrisR72, GrisR71]. The former was a micro-programmed machine, normally used for implementing the IBM System 360 architecture [BlaaG64], but in this case is used to implement an intermediate language [WirtN66] that is designed to be easily compiled from EULER and easily interpreted by that machine. The intermediate language is implemented by a micro-program running on the IBM 360/30 computer.

The complete SNOBOL4 system consists of a compiler which translates SNOBOL4 to an internal intermediate language, in Polish
prefix form, and an interpreter for this language. This interpreter is a conventional machine language program.

2.2 Intermediate Language Architectures (1.1).

Machines designed specifically as ILAs are not common. The major examples are the Burroughs' B1700 [BurrC72, WilnW72a, SaliA76, OrgaE78] and the QM-1 [NanoC74, SaliA76]. Both have an enormous amount of flexibility in their designs and are able to interpret efficiently intermediate languages, or S-languages, to use the Burroughs' terminology.

One of the advantages of the ILA is that the number of distinct S-language or S-machine (virtual machine) operations or instructions is much less than the number of operations or instructions in the instruction sets of conventional computers. This means the "op-code" field of an S-machine instruction can be much smaller than that in a conventional machine instruction. Also, the number of program variables in an average program is far less than the number of individually addressable locations in a computer. Thus the addressing of these variables may be encoded more efficiently than having to specify a complete machine address for each variable, as is normally the case. These points are expanded in [TaneA78] and [WilnW72b]. These advantages are important in the new area of micro-computers [ApplC79, CommPET, CromI79, ExidI79, TandC78] where cost is a major factor and
hence memory size, both primary and secondary, is constrained. These constraints are due directly to cost of the memory hardware and also due to the cost of supplying a large logical memory space or a large number of address bits. Using these advantages, amongst others, of the ILA the Pascal Microengine (R) [WestD79] has been produced. With this micro-computer, Pascal [JensK78] is translated to an intermediate language, which is an extended version of the standard Pascal P-code [NoriK76]. The P-code is interpreted by the micro-programmed micro-computer.

Other ILA designs are proposed in [TaneA78] and in [WeliA73].

2.3 Complete Syntax Definition (3.2.1, 4.1).

Methods for the definition of context free syntax have been known and used for some time; the most commonly known one being Backus-Naur Form (BNF) [NaurP63]. Many people have extended BNF to express more conveniently the context free syntax of programming languages [EdwaS80b] (reproduced as appendix one). A number of formal methods of defining the complete syntax, that is, the context sensitive and context free syntax, of programming languages have been invented. These include Production Systems [LedgH77], Attribute Grammars [KnutD68] and W-grammars [WijnA76, CleaJ77]; all of which are described in [MarcM76].
Both Production Systems and Attribute Grammars use the BNF type of generative grammar for defining the context free syntax. Production Systems uses global objects, most importantly the 'environment', and a number of predicates to restrict the generative grammars to legal programs. The objects in Production Systems include lists, tuples, strings and primitive atomic objects. An Attribute Grammar restricts its generative grammar to legal programs by passing "attributes" up and down the parse tree. Various conditions may be placed on the attributes and operations such as "give value to attribute" and "insert variable" (name into a list) can be applied to them. Both methods use mathematical concepts of sets and integers to form the basis of their definitions.

W-grammars, on the other hand, are quite self-contained, and rely on two levels of generative grammar. The extra level of generative grammar allows an infinite number of production rules to define syntax. This allows the "special cases" involved in the context sensitive syntax to be enumerated in the production rules directly. In a single level grammar there has to be a finite number of production rules and this leads to them being too general, thus requiring restrictions to be placed on them as in Production Systems and Attribute Grammars. The basis of W-grammars is simply: 1. character strings as objects; 2. a parameterization mechanism called the Uniform Replacement Rule; and 3. the ability to match patterns involving character strings. The primitiveness of the underlying model of W-grammars is often used as a criticism against it in that it leads to
Chapter 2 - Literature on Related Topics

much complexity. This complexity is avoided by Production Systems and Attribute Grammars because they use well-known mathematical concepts as their basis.

2.4 Translator Writing Systems (3.2.1).

As the name suggests, translator writing systems are notations for specifying the translation of one language to another. They can occur as part of a formal definition of a language as in the definition of EULER [Wirt66] or as part of a system for automatically producing translators or compilers, as for example, PL and FSL in [Feld66], JOSSLE [Whit75], META/REDUCE [Mart78] and a graph transformation language in [Bosc81].

Translator writing systems also occur as formal systems in their own right [Mich76] and as programming languages such as MACRO [Gree79, Sper78]. The latter is of interest since it was used as the starting point for the design of SYNTRAN which is described in chapter four.

MACRO has all the constituents of a normal high-level programming language but has an extra procedure-like construct which has a different invocation mechanism. This construct is the "macro" which can be triggered or invoked by the successful matching of a pattern, associated with each macro, against the input stream. Macros may also
be called by patterns in other macros. Thus execution of the main part of a MACRO program does not start with the first statement in the outermost block, as in normal high-level languages, but with the first statement in the first macro that has its complete pattern matched. The result of a MACRO program is an output stream built up from the unmatched portions of the input stream interspersed with strings generated by the macros.

The META/REDUCE translator writing system is designed to be used as a tool in experimental language design. A translator is specified by BNF-like rules which have extra constructs inserted in them. These extra constructs enable output to be generated and the state of the translator to be changed and are expressed in LISP [McCaJ60, McCaJ62, MartJ77] or REDUCE 2 [HearA73].

The process of carrying out an action or producing some output when a particular production rule on the source language grammar is used is a common mechanism. It also appears in JOSSLE and in the definition of EULER as well as in META/REDUCE. The pattern matching and macro triggering mechanism in MACRO can be viewed in the same light, the macro body is executed when the pattern in that macro has been matched, but it seems more natural and easier to use.

PL (Production Language) is a low-level language for describing the action of a sequential recognizer with a single stack. FSL (Formal Semantic Language) is a simple high-level language with special
operations for accessing the recognizer stack and accessing and building a symbol table to enforce context sensitive syntax. FSL can also specify that, by enclosing operations in "code brackets", they are to be converted to object code and executed at execution time rather than translation time.

JOSSLE combines the production rule type translation seen in the definition of EULER, a high-level programming language to manipulate a symbol table and enforce context sensitive syntax, and the code brackets of FSL. The objects which appear in the code brackets are constructs of a simple intermediate language that is not completely defined in the cited article.

Both PL/FSL and JOSSLE were designed with the automatic production of compilers in mind while MACRO and META/REDUCE were designed to produce text to text translators.

2.5 Compiler or Parser Generators (3.2, 3.2.3.2, 6.3).

True compiler generators have been discussed in the previous section, the PL/FSL and JOSSLE systems were designed as such; this section looks at what are more properly called parser generators, although they are often called compiler generators.
The distinction made here is that a compiler generator specifies the semantics of the language for which the compiler is being generated in a way other than in terms of the target machine code. The two examples given here: the XPL compiler generator of McKeeman et al. [McKeW70] and GSA [TurbT79], require the definition of language semantics by the writing of "semantic routines" which are called at various points during the parsing of a program and produce target machine code.

The XPL compiler generator consists of a proto-compiler, SKELETON, into which is inserted, at the source code level, tables produced by ANALYZER and the semantic routines written in XPL. The result is a complete compiler written in XPL. This is compiled to produce a compiler for the defined language. ANALYZER is a program which accepts a description of syntax in an extended BNF and produces a set of tables used by SKELETON to direct a table driven parser. The semantic routines not only produce the target machine code but are also required to enforce the context sensitive syntax of the defined language. That is, for example, a symbol table has to be programmed in XPL, and combined with the SKELETON.

GSA (General Syntax Analyzer) is similar in many ways to the XPL compiler generator. The differences are: 1. that the combination of parser tables, semantic routines etc. are carried out after any compilation, that is, they are linked together as relocatable object code modules; and 2. the method of defining syntax contains methods of
controlling error recovery, look ahead, the stack of symbols and procedure calls to the semantic routines.

Other, simpler, parser generators, similar in method to GSA are SYNPUT [DunnD77] and LR [WethC81].

2.6 Formal Semantic Definitions (3.2.1, 4, 5.4.1).

The formal methods for the definition of semantics of programming languages can be split into four categories: axiomatic, denotational, operational and transformational. The first two describe semantics by a strict mathematical approach while latter two include many formal, not-so-formal and pragmatic approaches. All definition methods of programming language semantics, either explicitly or implicitly, use the concept of a state. They describe semantics of a programming language construct by describing the changes that it makes to the state, and, if it is expression-like, the value that is also associated with it.
2.6.1 Axiomatic Semantics.

The axiomatic approach to the definition of programming language semantics, introduced in [Floy67, Hoar69], defines the semantics of programming language statements by applying a predicate to the state before execution of the statement, the pre-condition, and a predicate after the execution of the statement, the post-condition. If the pre-condition is true before the statement is executed, then the post-condition will be true after the statement is executed (provided the statement terminates). Expressions are assumed to have no side-effects.

Recent work has extended the axiomatic approach to parallel processes [Hoar75], expressions with side effects [Kowa77, SchwR77, PritP79] and 'goto' statements [ArbiM79, KowaT77]. A proof rule is a programming language construct with its associated pre- and post-conditions. The axiomatic approach is the most abstract one in use and its aim is to allow the proofs of, at least, partial correctness of programs. Some theoretical results regarding axiomatic semantics are given in [MeyeA80] and [ClarE77].
2.6.2 Denotational Semantics.

The denotational approach uses set-like objects called domains to represent the state of a computation and all values held in it. Functions are defined on the domains to transform from one state to another. Each programming language construct is defined by such a function. The mathematical grounding required by these complex functions, which include domains of functions and are often defined recursively, was supplied by Scott [ScotD71]. More recent tutorials on the subject are [TennR76] and [GordM79]. A complete denotational semantic model for statement oriented programming languages is described in [StybJ76] and Mosses has used denotational semantics to describe the semantics of programming languages for the purposes of a compiler generator [MossP78].

2.6.3 Operational Semantics.

Here the semantics of a language is defined by specifying an interpreter for that language.

The first formal operational semantic description method was the SECD machine of Landin [LandP64] which was able to interpret what he called "applicative expressions" or expressions with no side effects. The letters SECD stand for the parts of the state space used in the abstract machine. These parts are: the Stack, for storing temporary
results; the Environment, for identifier/value associations; the Control, for storing the "program" of applicative expressions; and the Dump, for state saving when new levels of applicative expressions are entered. The dump is used as a stack of states. Developments from this were in the direction of denotational semantics, e.g. [LandP65].

The most widely known operational definition method is probably VDL [LucaP68, WegnP72, MarcM76]. VDL uses labelled trees to represent the state space which, in [WegnP72], has a close resemblance to the state space of the SECD machine. VDL has extra components in its state which are required to model programming language "variables". The VDL formalism does not impose any particular structure on the state space tree except that there must be a "control" sub-tree where interpretation can begin. The actual interpretation is specified in a pure LISP-like [McCaJ62] language, which interprets operations on the control sub-tree. Indeterminism and collateral evaluation are introduced by allowing the interpretation of leaf nodes of the control tree in any order. Sequential, deterministic interpretation occurs when there is always only one leaf node in the control tree to be interpreted at any time.

A more recent approach to operational semantics is shown in SEMANOL [AndeE76]. The interpreter in this case is specified by a high-level language program. The state space is built out of data structures in this language and both data and control structures are designed to facilitate such interpreter definition.
2.6.4 Transformational Semantics.

The semantics of a language can be defined by specifying a translation between it and a target language whose semantics are known. This method is often used informally where a particular construct in a language is defined in terms of other constructs in the same or different language. For example, the Pascal "for" statement is described this way, very informally, on page 25 and page 154 in [JensK78] and on page 80 in [RaveB79]. The same statement could also be described using a machine language code segment that would carry out the for statement.

One of the first methods for formally defining semantics using a translation was described by Feldman with PL and FSL [FeldJ66] (see section 2.4). Another was used in the definition of EULER [WirtN66]. Here EULER was translated to an intermediate language which was, in turn, described by an interpreter. The interpreter was described using a simple language referred to by the authors as "an elementary notation for algorithms". The same simple language was used in the translator to apply the context sensitive restrictions on the syntax of EULER.

The two systems JOSSLE and META/REDUCE (see section 2.4) also deserve a mention in this section as they, too, can be considered to define semantics by translation. And, taking an extreme view, all compilers on all computers each define a language by the way they
behave. These definitions, though, are not freely available, accessible nor readable (usually), and hence do not meet the criterion normally required of a formal definition. They are often used, though, as the final arbiter of what the semantics of a particular implementation of a language is.

A formal study of definition by translation has been carried out by Pepper [PeppF79] and it is he who introduced the term "transformational semantics".

2.7 Machine-Oriented Low and Intermediate Level Languages (1.1).

Machine-oriented languages (MOLs) are low- or intermediate-level languages which are machine-independent but have features based directly on common machine properties. Assembly languages are classed as low-level and machine-dependent. MOLs can be split into three classes by their usage. Firstly, they can be used in formal definitions of programming languages such as FSL and the intermediate language used in EULER. Both of these have been described in previous sections in this chapter. Secondly, they can be used as an intermediate stage in the compilation process to perform machine independent optimization. For example, U-code [PerkD79] has been designed to replace P-code [Norik76], the intermediate language produced by many Pascal [JensK78] compilers. U-code is based on P-code but carries more information to allow better optimization. Similarly,
an intermediate language using n-tuples [FraiD79] has been described for the same purpose though it is both machine and source language independent. More recently, an intermediate language EM [TaneA82] has been designed to facilitate the application of peephole optimization to programs expressed in it.

The majority of MOLs were designed to facilitate portability of high-level languages. Those designed to aid the portability of particular languages include: P-code, OCODE [RichM71], and SIL [GrisR72] which were designed to aid the portability of: Pascal, BCPL [RichM69], and SNOBOL4 [GrisR71], respectively. The first two, P-code and OCODE, are used as the target language for a compiler. P-code is usually interpreted while OCODE is usually translated to machine code. SIL (SNOBOL4 Implementation Language) is used to write the whole SNOBOL4 system which is translated to machine code by a macro processor and an assembler or an assembler with macro facilities.

Other MOLs designed to facilitate portability are the machine and language independent ones stemming from the recognition of the desirability of an UNCOL (UNiversal Computer Oriented Language) [StroJ58]. They include: an UNCOL itself [SteeT61], JANUS [ColeS74] and FAML [NessD77]. The basic idea behind these intermediate languages, (except SIL, which is directly translated) is: that given a compiler for a high-level language which translates that language to an intermediate language and is also written in the intermediate language; the work required to transport the compiler, and hence the
high-level language, to a new machine is just that involved in implementing the intermediate language on that new machine. This implementation may take the form of either an interpreter or a translator.

2.8 Code Generator Generators (3.2.2, 3.2.3.3, 6.3).

Compared to the "front-end" of a compiler (the parser), the "back-end" (the code generator) has shown itself to be much less amenable to automatic production. This is because parser generators deal with the relatively simple field of programming language syntax, something which can be described in a few pages of very formal notation, whereas a code generator generator deals with machine instruction semantics. Cattell and Leverett [CattR79] have discussed the problems involved in designing the code generation phase in a generator for a machine independent "production quality" compiler, that is, a compiler which carries out a substantial amount of optimization. A working system derived from this is described in [CattR80].

Lakos has described and implemented a code generator generator [LakoC77] for use with BCPL [RichM69]. BCPL is translated to an intermediate language OCODE [RichM71] during compilation. The code generator generator accepts a description of the target machine written in Machine Description Language (MDL) and produces tables
which are used by the code generator program itself. This code generator then accepts OCODE output from the compiler and produces machine code for the described target machine. MDL is a "source language dependent machine description language" and was designed to describe machines specifically for the purposes of the OCODE code generator. This language dependence extends to requiring that if there is no machine instruction equivalent to a BCPL operation then that operation must be expressed in terms of other operations that do have equivalent machine instructions.

One code generator generator system that is not source language dependent is described in [GlanR77]. Here the target machine instructions are defined in terms of a Polish prefix language (IR) and the code generators produced accept programs written in the same language. The target machine instruction set description is used as the set of production rules for a grammar, and the programs written in IR are "parsed" using this grammar and produce a sequence of target machine instructions. It is envisaged that high-level language programs are translated to IR programs by a process with a machine dependent phase which incorporates implementation decisions. These decisions will affect the efficiency of the generated code differently for different target machines.

Another source language independent system is described in [DoneM79]. Here, a code generator is described by defining the intermediate code operations input to the code generator in terms of
the target machine instructions using a non-procedural language (CGGL) oriented towards writing code generators. Each intermediate code operation is described by a sequence of one or more target machine instructions that carry out that operation provided certain conditions on the machine state hold e.g. an operand is in a certain register. If an operation in the input is encountered when the required conditions do not hold, "transition"s, defined in the CGGL program, are used to supply a sequence of target machine instructions which will bring about the conditions required.

All the above systems that have been implemented seem to give good performance in respect to their aims. It should be noted, though, that the intermediate languages used as input to the generated code generators of these systems (OCODE, IR) are low level; even the use of target machine index registers has been decided upon a priori, and appears in the input to the code generators. Thus, decisions as to the method of carrying out high-level language operations on the target machine such as procedure calls and returns are supplied either in the description of the target machine (as in MDL and CGGL) or in the intermediate code given to the code generators (as with IR).
2.9 Automatic Program Analysis (4.2.3.1, 5.1).

There are a number of reasons why automatic analysis of program text is investigated. The most common is, perhaps, the hope that such analysis will eventually be able to provide the difficult-to-find assertions, especially invariant assertions [DijkE72], required to prove the correctness of programs [FloyR67, HoarC69]. Examples of such work are [KatzS76] and [WateR79]. The latter also suggests the desirability of having this analysis carried out during the construction of the program; such a system is being developed [RichC78].

Another important topic in automatic program analysis is symbolic execution of program text. Apart from simply testing a program this way [BoyeR75, HowdW78], assertions required by proofs can be tested and suggested [KingJ76], and test data may be produced automatically [ClarL76].

Other topics in this area include analysis of program performance by automatic analysis of the program text [WegbB75] and the extraction of control structure from the program text [BakeB77].
2.10 Process Synchronization Primitives (5.5.2, 5.5.4).

The problem of synchronization of independent concurrently executing processes is encountered where two such processes require to access (and modify) a single piece of data or resource. A clear analysis of this problem was given by Dijkstra in [DijkE68] where he proposed the "semaphore" as a solution. A restriction of the "general" semaphore, called the "binary" semaphore, has a direct analogy with hardware synchronization primitives. Dijkstra showed that the general semaphore could be modelled with the binary semaphore. Associated with each resource which is accessible to more than one process is a semaphore. In the case of a binary semaphore, it has two values, zero and one. When the semaphore has the value zero, a process is using the resource; when the semaphore has the value one, the resource is free. When a process requires the use of a resource, it carries out the "P" operation on the associated semaphore. If the semaphore was one, the resource was free and the semaphore becomes zero and the process proceeds. If, on the other hand, the semaphore was zero, the process must wait until it becomes one again before proceeding as above. When the process is finished with the resource, the "V" operation is applied to the semaphore and its value becomes one. This mechanism allows only one process at a time to access the resource and is said to 'serialize' all accesses to the resource.
A problem with semaphores is that it requires two separate operations to achieve mutual exclusion of processes. This is unstructured and can lead to errors in programming. The concept of "critical regions" was introduced by Brinch Hansen [BrinP72, BrinP73] and overcomes this problem. The area of code between the use of the P and V operations on a semaphore protecting a resource is considered a critical region. A single programming construct surrounds this region and supplies implicitly the semaphore and the P and V operations to correctly access the resource. Brinch Hansen also introduced the "conditional critical region" where the execution of a process could be suspended inside the critical region until a specified condition is satisfied.

With critical regions, a single resource can be accessed by a number of critical regions throughout a program. Monitors [HoarC74] are effectively the collection in one place of the critical regions that access a particular resource. Variables and procedures concerned only with the one resource, are also collected in the same monitor. Monitors have appeared in two recently designed programming languages, in Concurrent Pascal [BrinP75] as classes and in MODULA [WirtN77] as interface modules.

Synchronization between two processes can also be achieved by the recognition of a mutually known event such as in CSP [HoarC78]. Here two processes must name each other and one passes data to the other if both agree on the format of the data. This event is seen as an output
operation in one process and as an input operation in the other. Whichever process reaches an input or output operation first must wait for the corresponding output or input operation, respectively, to be reached in the other process. A similar mechanism, called the rendezvous, is used in Ada [IchbJ79] but here a number of processes may output to the same process input operation ("accept"ing an "entry", in Ada terminology), one at a time, and hence may have to be queued (see appendix seven).

Although not relevant to this thesis as a whole, three other synchronization primitives should be mentioned here. They are "eventcounts" and "sequencers" [ReedD79], "path expressions" [CampR74] and "serializers" [HewiC77].
Chapter 3 - SAGA

3 SYSTEM FOR THE AUTOMATIC GENERATION OF ARCHITECTURES.

This chapter provides a description of the proposed System for the Automatic Generation of Architecture (SAGA). To start with, a notation for describing system implementation [RosiR77] and the purpose of SAGA using this notation are described briefly. The various parts of SAGA are then identified and described and the requirements for each outlined. Finally, the requirements of the method used to define the high-level language to SAGA are discussed. A summary of SAGA can be found in [EdwaS80a] which is reproduced as appendix four.

3.1 The Purpose of SAGA.

3.1.1 Rosin's Notation for Describing System Implementation.

This notation has four basic elements:

1. text, $T$, expressed in some notation, convention or language, $L$, and drawn:

```
    T
   / \L
```

2. a transforming or translating program (TP) which transforms a program or other data written in a language or expressed in a notation $L$, to the same program or data written in another language $S$. This transforming program, itself
Chapter 3 - SAGA

written in $N$, is drawn:

$$L \rightarrow S$$

If the input and output languages of the transforming program, $P$, say, are not of interest then it may be drawn:

$$P$$

3. an interpreting program (IP) which is written in $M$, and which interprets the language $S$, is drawn:

$$S$$

4. an interpreting machine (IM) which interprets or executes programs written in $M$, is drawn:

$$M$$

The three elements with curved sides may be juxtaposed so that their curves and languages match to form transforming machines (TMs) which have the overall shape:
This method of composition is referred to as "loading". The requirements for languages to match is satisfied if the language B is identical to language C in the following diagrams.

Note that a particular TM may have none, one or more IPs embedded in it. This notation is now illustrated with a few examples derived from common situations.

A program U, written in Pascal [JensK78] and compiled to machine language M is depicted thus:

Here, the implementation of the compiling TM is not specified. The compiled program may be loaded and run on an 'M' machine. The program U may be defined further by specifying the input and output "languages" as I and O; thus U becomes I → O. This use of the term "language" is not usual but can be established formally. The execution of U is illustrated by:
If, instead, the Pascal compiler produced P-code [NoriK'76], the compilation phase would be drawn:

The P-code may then be interpreted on an M-machine in which case the execution phase would be drawn:

For more detailed explanations and examples see [RosiR77].

This notation has the following advantages over T diagrams [EarlJ'70]:

1. it can depict primitive "data" objects (as plain rectangles),
2. the execution of a distinct program is indicated by each distinct "Translating Machine", and
3. the production of interpreters by programs is catered for as well as the production of translators (compilers).
3.1.2 Purpose of SAGA Described Using Rosin’s Notation.

![Diagram of SAGA system with Rosin's notation]

- **Def. of L**
- **Def. of M**
- **SAGA**
- **Interpreter Generator**
- **L → S**
- **I → O**
- **I → S**
- **I → O**
- **Input I**
- **Output O**
- **User program input**
- **User program output**
- **Compiler**
- **ILM machine**
Here L, S and M refer to the high-level language for which the interpreter and compiler are being produced, the intermediate language produced, and the the low-level target machine (language), respectively. I → O refers to a user program originally written in L. The interpreter generator takes as input a definition of the high-level language, L, and the machine, M, and produces as output, an interpreter for S written in M, and a compiler for L, which translates L to S. The implementation of the compiler is not specified here. In the diagram above the compiler and interpreter are shown loaded into the transforming machines of the compilation and execution phases respectively. The method of defining L, the method of defining M, and the interpreter generator constitute the proposed SAGA system.

Two classes of users are envisaged with two quite different sets of requirements. The first is the computer manufacturer, computer centre manager or, perhaps, an individual who wishes to get an implementation of a defined language for a particular machine. The second is the software house, academic, specialist programmer or student who wishes to experiment with the design of a language and needs a method for implementing it automatically.

The first user is not concerned with the expense of running SAGA in the ideal situation where the descriptions of L and M are already available since SAGA would only be used once for each language/machine combination. This is analogous to the “production” programming environment where programs are compiled, again ideally, only once, and
run many times. Where a language implemented using SAGA is to be transported to another machine only a description of the new machine need be created. The description of the language and the new machine description are then supplied to the interpreter generator and the required new implementation is produced. The efficiency of the products: the compiler and the interpreter in the case of SAGA; and the executable module in the case of the production environment, is the main concern. Going a step further, if the compiler and interpreter produced by SAGA are to be used in a production environment then the efficiency of the interpreter is of primary concern.

The second user has a quite different point of view. In order to gain a feel for the language being designed or modified, SAGA may be invoked many times with different descriptions of \( L \); the same description of \( M \) would be used provided the target machine did not change. Thus the efficiency of SAGA itself would be important, but not to the complete detriment of the efficiency of the compiler and interpreter, which should be good enough to allow experiments with the designed language to be carried out realistically.
3.2 Description of SAGA.

SAGA is clearly divided into three parts, each of which is described briefly in the sections below. These three parts are: the definition of L; the definition of M; and the interpreter generator. The last part is called an interpreter generator, although it generates both a compiler and an interpreter, for conciseness and to distinguish it from what are generally known as "compiler generators" (see section 2.5). The intermediate language $S$ is defined implicitly by the compiler and the interpreter. $S$ is considered to be the machine language of the machine with the ideal architecture for executing programs written in $L$.

3.2.1 The Definition of $L$.

$L$ must be described completely (see sections 2.3 and 2.6) since it must be "understood" by the interpreter generator. It is also highly advantageous if, from the definition of $L$, the structure of $L$ can be obtained by simple rules applied to the definition. For example, it should be possible to determine whether a stack oriented storage allocation system is sufficient to implement $L$ by the application of rules, which are concerned with acquisition and release of storage, to the definition of $L$. 
The use of the word "structure" when applied to a high-level language such as L in this thesis is used to convey the concept of the semantics of the language as a whole. By extracting the structure of L, the form of the abstract L-machine is obtained.

A corollary of the above requirement is that the method chosen to define L must not impose any structure of its own. The structure of L must "shine through" untainted.

It is also preferable that the definition of L be easy to use by programmers with a knowledge of programming language semantics. The easier the definition method is to use, the more willing people will be to create and read such language definitions and the fewer the number of mistakes that will be made.

The method proposed consists of a high-level language used to describe the translation of L to a target language (see section 2.4) which, itself, is as high-level as is practical considering its purpose. This method provides a definition by translation, and the term introduced by Pepper to classify this approach is "transformational semantics" [PeppP79]. The method also uses a high-level language to perform the translation; this facilitates its "readability" and "writeability". It clearly allows for a complete definition by the suitable choice of the target language. It should also, by design, facilitate the extraction of the structure of L.
The effect of the definition method on the perceived structure of L is minimized by the use of a high-level target language as, ideally, constructs in L can be translated to constructs in the target language directly without resorting to low-level description. It is the use of low-level target languages in transformational semantic descriptions, which discard high-level programming language structure, that makes the extraction of such structure very difficult. To illustrate, consider the relationship between a program written in a high-level language and its compiled machine language equivalent. The latter can be considered as a definition of the semantics of the program. The process of converting or translating the high-level language to the low-level machine language is relatively easy. Consider, now, the inverse relationship; the process of translating a machine language program to a high-level language program can be of the same order of difficulty as the compilation so long as no optimization has taken place and each language structure can be recognized as a pattern of certain machine instructions. However, if any optimization, especially local optimization [McKeW65], has taken place, or, worse still, the machine language program was written directly in machine language, the "un-compilation" process would have little hope of producing an equivalent high-level language program without modelling, to some extent, the machine architecture.

Thus in translating a high-level language to a low-level language the structure of the high-level language is very likely to be lost and the structure of the low-level language imposed instead. The analogy
to be drawn is that, by describing L in terms of a low-level language, the structure of L is removed before it can be extracted. (It is perhaps relevant to note here that work has been done on restoring control structure in FORTRAN to FORTRAN translations [BakeB77].) It was as an attempt to avoid these problems that the target language was so designed.

The definition method chosen also allows for a simple way of specifying the compiler (see 3.2.3.2).

The language used to define the translation and the language used to act as the target language are called SYNTRAN and GLADYS, respectively. SYNTRAN is an acronym for SYntax TRANslator: the idea being that SYNTRAN translates from one language to another by changing the syntax only and not the semantics. This, of course, applies to any translator, by definition, but seemed to apply particularly well to the use for which SYNTRAN was designed. GLADYS is an acronym for Good Language for Analogously Defining Semantics. This name was motivated by the idea that L is defined by drawing the analogy between constructs of L and their equivalent constructs of GLADYS. SYNTRAN is described in [EdwaS80c] (reproduced as appendix two) and discussed in chapter four and GLADYS is described in [EdwaS80d] (reproduced as appendix three) and discussed in chapter five.
Examples of the use of SYNTRAN/GLADYS definitions can be found in appendices five to eight where definitions of A$\text{SPLE}$ [MarcM76], Pascal [JensK78], rendezvous [IchbJ79] and LISP [MartJ77] are given.

3.2.2 The Definition of $M$.

The definition of the target machine $M$ is required to produce the interpreter for $S$. No attempt is made in this dissertation to fix on a particular method for defining $M$. It is envisaged that one of the systems currently being developed in the field of code generator generators (see section 2.8) will provide a suitable notation for this purpose. The applications of a language derived from ISP [BellC71], including its use in the production of code generators, are discussed in [BarbM81].

3.2.3 The Interpreter Generator.

The task of the interpreter generator is to take the definitions of $L$ and $M$ and produce a compiler and an interpreter.
3.2.3.1 General.

From the description of L, its structure is obtained and embodied in a set of operations defined in terms of the target language used in the definition of L. This set of operations is the "instruction set" of the virtual S-machine and forms the definition of S. The implementation of S requires a choice to be made on the format of the S-instructions which correspond to each operation of S. This format includes the sizes of the various fields of the instructions, the "op-code" fields, the addressing fields, the data type fields, etc.

3.2.3.2 Generation of the Compiler.

Having chosen a definition of L using a translator the automatic production of the compiler becomes, conceptually, at least, very simple. The "front-end" of the compiler, the scanner and parser can be produced from the syntax of L as specified in the translator using techniques used in current "compiler" generators (see section 2.5). The definition of L already specifies the translation of L to some target language, thus a specification of the compiler requires that the constructs of the target language be replaced by the equivalent constructs of the intermediate language, S. This simple mechanism requires that where the target language portions describing a construct in L are scattered throughout the definition, it is in a way that allows each S-language operation to replace one portion. This
requirement could be ensured by defining the S-language operations precisely so that this occurs. Examples in the following two chapters indicate that so defining the S-language operations is not restrictive.

3.2.3.3 Generation of the Interpreter.

The problem of generating the interpreter for S is clearly related to the problem of automatic code generator generation. Instead of just producing a sequence of target machine instructions that carry out a particular S-operation, it is required that the S-instruction associated with that operation be decoded and then carried out by the sequence of machine instructions. Apart from having to choose the format of S-instructions as mentioned above, there is the problem of the difference between the level of S or the complexity of S-operations as they are envisaged and the level of the intermediate languages accepted by code generator generators (see section 2.8).

A solution to the former problem requires the matching of the S-operation and addressing modes with the target machine addressing and field definability, that is, its ability to manipulate data fields of different lengths. This appears fairly straightforward in a target machine with conventional architecture. With the flexibility of a machine designed for S-language interpretation, though, an efficient choice requires knowledge about the static and dynamic frequency of
Chapter 3 - SAGA

each operation so that operations occurring or executed more frequently are assigned smaller instructions. This reduces the requirements on physical memory and also memory bandwidth [WilnW72b]. How this information is to be supplied to SAGA is not investigated in this dissertation.

The latter problem, the gap between the level of S and the level of the language accepted by code generator generators, embodies the area where good machine dependent implementation decisions have to be made to produce an efficient interpreter. This is the major problem left unresolved in SAGA and is a topic for future research. In its most general statement, the problem requires the capabilities of artificial intelligence and "expert" systems (see [BarrH80] and references therein). It has been suggested [HursA80, GlanR77 (pp22-25)] that the generality of such solutions may not be required because of the fairly limited set of data and control structures of current high-level languages and the knowledge of standard implementation techniques of certain high-level language features.

An important example here is the methods used in the implementation of block structured languages. Two standard methods: the use of a "display" [GrieD71] or "current environment vector"; and the use of "static links" [WegnP68] may form a part of, or a complete list of, options in a "menu selection" approach to implementing an ALGOL 60-like block structured language. With such an approach the S language would be tested for certain properties regarding storage
allocation and identifier bindings to ascertain whether particular items of the menu will satisfy the S-language requirements. One is then chosen from amongst them as the most efficient with regard to the target machine.

Suitable data structure representations also have to be chosen. This problem has been tackled in the case where representations are required for data structures in a particular program [LowJ78, SchoE81]. If we assume that data structure representation selection is not going to be performed for each program, the usual case, then the representations need to be selected and fixed in the interpreter. Once again, it is envisaged that the techniques developed by others for program specific cases can be adapted and generalized to the interpreter specific case required in SAGA.
This chapter provides the motivation and use of SYNTRAN which is described in detail in [EdwaS80c] (reproduced in appendix two). SYNTRAN is used to describe the translation from a high-level language, L, to a target language, in this case GLADYS, so that L may be defined (see section 2.6.4). SYNTRAN defines both the context free and context sensitive syntax of L and the semantics of L is described in terms of the target language.

L does not cater for extensible languages in the sense that the languages are not allowed to change their grammar while being parsed. The normal macro facilities found in some implementations of languages are acceptable.

4.1 Aims.

The overall aim of SYNTRAN is to describe translations between high-level languages; it is desirable that SYNTRAN make this process as easy as possible. It is also required that SYNTRAN facilitate the extraction of the source language features from a static definition written as a SYNTRAN program. These aims can be refined to three aims which are more suggestive of practical solutions. SYNTRAN must facilitate:

1. the specification of context free syntax (see section 2.3),
2. the specification of context sensitive syntax (see section 2.3) and

3. the analysis of the output stream of SYNTRAN programs. To be able to extract the language features and structure of the language \( L \), not only does the semantics of the target language have to be known, but also how the target language is used and in what combinations its various constructs appear.

4.2 Features.

The most obvious construct in SYNTRAN designed to facilitate translation of programming languages is the "translator". A translator is a single unit which specifies the strings it will match from the input stream, the conditions under which the translator applies and the output produced. The input stream is matched using "patterns", and the output is produced using "productions". A set of general purpose control structures is available to carry out any computations required.
4.2.1 Specification of Context Free Syntax.

The method chosen to specify the context free syntax of the source language in the input stream of a SYNTRAN program is to use pattern matching of the type available in MACRO 1100 [SperU78, GreeS79]. Patterns in SYNTRAN are very similar to the extended BNF in [EdwaS80b] (reproduced as appendix one) and, in fact, they were developed concurrently. They were designed to allow the context free syntax of a language in the above extended BNF to be transcribed to and from SYNTRAN patterns. The difference between the two notations is that an explicit concatenation operator is used in SYNTRAN patterns to make parsing of SYNTRAN programs easier and the pattern assignment operator in patterns. This operator allows the actual text matched by pattern elements to be assigned to variables in a translator and used in further processing in that translator. So as not to overload square brackets and braces in SYNTRAN these symbols were not used in patterns nor, therefore, in the extended BNF. The symbols plus and asterisk are used in the way they are so as to reflect their usage in generative grammars in formal language theory (see, for example, p16 in [GrieD71]).

The sets and tokens in SYNTRAN come directly from MACRO 1100; they allow the lexical structure of a language to be described separately from the syntax. This makes the patterns in translators much easier to use and read. The separation of lexical and syntactic structures in SYNTRAN allows the definition of an "ignored pattern"
which is used to specify particular character sequences in the input stream which are ignored in the lexical scanning process except as markers. The ignored pattern is used to match blanks, end of line markers, comments and other non-significant lexical units and is placed implicitly between each set, token and literal in a pattern. Any part of the input stream matched by the ignored pattern is discarded immediately. As an example, the following syntax rules of ASPLE can be determined from the first four translators in appendix five:

\[
\begin{align*}
\text{PROGRAM} & = \text{DCL\_TRAIN} \text{ STM\_TRAIN}. \\
\text{DCL\_TRAIN} & = (\text{DECLARATION} \ ' ; ')^+. \\
\text{DECLARATION} & = ' \text{ref} ' ^* ( ' \text{int} ' \mid ' \text{bool} ' ) \text{ ID} / ', '. \\
\text{STM\_TRAIN} & = \\
& (\text{ASG\_STM} \mid \text{COND\_STM} \mid \text{LOOP\_STM} \mid \text{IN\_STM} \mid \text{OUT\_STM}) / '; '.
\end{align*}
\]

4.2.2 Specification of Context Sensitive Syntax.

As well as the ability to obtain relevant portions of the source string via pattern assignment SYNTRAN provides a complete set of control structures for checking context sensitive syntax. The control structures are quite conventional and include conditional and repetitive statements and a function call mechanism. The main primitive data structure is the character string, reflecting the non-numerical nature of the processing required, and a very flexible
data structure allowing important structures such as symbol tables to be built up easily. This data structure is called a node. A node has a number of variables called branches which are selected by string values. This allows data structures similar to arrays and records (ALGOL 68 [WijnA76] 'struct's) to be implemented as well as, in the most general case, directed graphs. In particular, tree or stack structured symbol tables, as well as lists, can be created with ease.

In the ASPLE definition in appendix five the symbol table is a node with selectors, the strings representing identifiers, selecting nodes. These nodes have two selectors, the strings "REFS" and "TYPE"; REFS selects an integer variable containing the number of 'ref's appearing in the identifier declaration plus one, and TYPE selects the string "int" or "bool" depending, again, on the identifier declaration.

The node data structure grew from MACRO 1100 arrays, MASM [SperU77] nodes and SNOBOL4 [GrisR71] tables. A problem arising from the use of the node structure is that sequencing through the various branches of a node is impossible without further information. In MACRO 1100, this is accomplished by ordering the branch selectors and providing functions to obtain the first and last selectors of a node and the next and previous selectors of a given selector of a node. The only way to accomplish this in SNOBOL4 is to convert the table to an array and use the fixed indexing provided by the SNOBOL4 array. SYNTRAN has an operator which when applied to a node returns another
node with branch values the selectors of the argument node and branch selectors the integer values starting at one and increasing by one for each branch in the argument node. This provides an ordering for, and an access path to, the argument node selectors.

Since most high-level languages have recursively defined syntax, translators may be invoked recursively. A trace of the translator invocations provides the basis for a parse tree. It is very useful to be able to pass contextual information up and down this parse tree without resort to global variables which would complicate programs significantly. In SYNTRAN parameters may be passed to and from translators directly imitating the inherited and synthesized attributes of Knuth's attribute grammars [KnutD68].

Ultimately, the context sensitive syntax checking relies on the fail statement to reject illegal input strings. The fail statement causes the translator in which it occurs to fail as though its pattern did not match, that is, the input string is returned to the state it was in before the pattern was applied. For ease of implementation, global variables changed by the translator before it encountered the fail statement are not restored. This means, in general, that the fail statement should be executed as soon as is possible and that, if any restoration of global variables is required, it is done explicitly between the time it is discovered that the translator should fail and the fail statement itself. For example (from appendix five):
Chapter 4 - SYNTRAN - The Translating Language

translator ID

pattern I ← IDE endpattern

production

[ if #I > n4 then fail endif ]

I

endproduction

endtranslator;

ensures that the length of identifiers is not greater than the value of n4.

4.2.3 Analysis of the Output Stream.

In attempting to achieve the third aim of SYNTRAN, identified in section 4.1, the most important and novel feature of SYNTRAN arises. This feature is embodied in the "production" construct and consists of the adaptation of the pattern matching operations of alternation, concatenation, repetition and listing to the production of strings in an algorithmic manner so that they complement the consumption of the input string when used in pattern matching. Thus when the syntax of the operands is known the syntax of the possible output streams are known from the way they are combined with the operators in the production.
4.2.3.1 Context Free Syntax.

The primitive elements of productions are literals and variables with "use mode" E. The syntax of the value of literals is self-evident and, because of the way SYNTRAN is defined, the syntax of the values of variables with use mode E is also very easily found. Variables with use mode E are those which are declared and given their values in patterns or emission declarations. Once these values are set they cannot be changed until re-declared.

The "new" function is used to create unique strings, conforming to a specified syntax, which can be used to create unique names in the target language program if required. Thus each operand in a production has a known syntax.

The syntax of the value of a variable declared in an emission declaration is stated explicitly in the declaration. The syntax of the value of a variable declared in a pattern can be found by inspecting the context in which it occurs (see section 14 in [EdwaS80c]). The syntax of such a variable is basically just the syntax of the value returned by the pattern element on the right-hand side of the pattern assignment operator '←'. If, however, the pattern element was not matched at all, which is possible if it forms part of an argument to an alternation operator ' | ', the right-hand argument of the listing operator '/ ' or the argument to the option operator '? ', then the value of the variable will be null. On the other hand the pattern
element may have been matched more than once, which is possible if it forms part of an argument to the listing operator or the repetition operators '+' and '*', in which case the variable is a node with branch values which conform to the syntax of the pattern element. In this latter case only branches of the variable can occur in productions. The definition of SYNTRAN disallows the case where a variable with a null value due to the fact that its associated pattern element was never matched, can form part of the output stream (see section 15 in [EdwaS80c]). This avoids the situation where a pattern contains:

\[
\begin{align*}
A & \sim P1 \mid B \sim P2
\end{align*}
\]

and a production in the same translator contains:

\[
A \^ B
\]

which is very misleading. The production appears to consist of a concatenation whereas, in fact, when A is matched B will be null and vice versa. It was decided that the above rule was preferable to requiring a more complicated algorithm for determining the syntax of a particular variable declared in a pattern. This algorithm would have to include the information that A and B occurred in different arguments to an alternation operator. Thus the syntax of the value of a variable used in a production and declared in a pattern has the syntax of the pattern element appearing immediately after the pattern assignment operator following that variable. Instead of:

\[
A \^ B
\]

then, SYNTRAN would require a control declaration like:

control A_MATCHED boolean A \sim null
and a production such as:

\[ A_{MATCHED} \rightarrow A \mid B \]

or:

control \( A_{MATCHED} \) boolean \( A \neq \) null;
control \( B_{MATCHED} \) boolean \( \neg A_{MATCHED} \)

and:

\[ A_{MATCHED} \rightarrow A? \]

and:

\[ B_{MATCHED} \rightarrow B? \]

A production expression in a translator can be converted to the right-hand side of a rule in the extended BNF in [EdwaS80b] by the following steps:

1. Variables are replaced by a description of the syntax of their value.
2. Translator calls are replaced by just the name of the translator forming a "non-terminal" symbol.
3. The concatenation operator \('\) is removed from all places.
4. The Boolean and integer controls are removed.
5. The angle brackets used for grouping are replaced by parentheses.
6. Any simplifications that are possible are applied. These include such things as the removal of redundant parentheses and the combination of a number of operators into one. For example, \((A+)\)? can be simplified to \(A^*\), \((A+)\)+ can be simplified to \(A^+\) and \(A \mid \) null can be simplified to \(A?\) etc.
Using the name of the translator in which a production occurs as
the left-hand side of a rule, the following conversions from
translators to syntax production rules can be performed. Thus, from
the translator VARIABLE (page 284 in appendix six):

```plaintext
translator VARIABLE(SCOPE: V_TYPE, V_KIND)
  local TYPE, ENTRY use V
  pattern
    ROOT ← USEDID(: ROOT_ENTRY) ~
    S ← SELECTOR(SCOPE: S_TYPE)*
  endpattern
  production
    [ ...
      control N_SELECTORS integer #S
      ...
    ]
    ROOT ~
    < N_SELECTORS(I) →
      [ control POINTER boolean S.I = '^
          control INDEX boolean S.I ~= null & S_TYPE.I ~= null ]
    < POINTER →
      [ ... ]
      '@'
    < INDEX →
      [ ...
        emission INDEX_TYPE is TYPE_NAME: TYPE.'INDEX'.NAME;
        ...
      ]
    < '@.' ~ INDEX_TYPE ^ '<' ^ S.I ^ '>' ]
    < [ FIELD_SELECTOR → ]
      [ ... ]
    emission RECORD_INDEX_TYPE is TYPE_NAME:
    type.'INDEX'.NAME';
    emission FIELD_NO is UNSN: ENTRY.'#';
      ...
      '@.' ~ RECORD_INDEX_TYPE ^ '<' ^ FIELD_NO_ ~ '>'
    >>>*> ]
  endproduction
endtranslator;
```

the context-free syntax of VARIABLE (in the GLADYS translation) can be
extracted as:
VARIABLE = USEDID ('@' |  
  '@.' TYPE_NAME '<' SELECTOR '>' |  
  '@.' TYPE_NAME '<' UNSN '>')*. 

and from translator USEDID (page 266):

translator USEDID( : ENTRY)  
{ identifier in expressions, supplies symbol table entry }  
local L use V  
pattern I ← IDENT endpattern  
production  
[ ... ]  
I  
endproduction  
endtranslator; 

the following is obtained:

USEDID = IDENT.

and from token definitions (page 261):

IDENT = LETTER ID_CHAR*.

UNSN = DIGIT*.

TYPE_NAME = 'T_' UNSN.
can be obtained.

The above algorithm covers all cases except where a macro call is encountered in a production. In this case the macro call can be expanded so that its production can be converted to the extended BNF as well or it can be left as a parameterized syntactic unit. For example, in the translator SELECTOR (page 283):

\[ \text{SELECTOR} = '^^' \mid \text{IDENT} \mid \text{TO_INT(EXPRESSION)}. \]

\text{TO_INT(EXPRESSION)} can be expanded to:

\[ '##'? \text{EXPRESSION}'@'? \]

using the macro \text{TO_INT} (page 265).

4.2.3.2 Context Sensitive Syntax.

In order to determine the context sensitive syntax of the output stream the expressions determining the number of repetitions of a construct and which alternative of two constructs is to be used, are clearly identified by the syntax of \text{SYNTRAN}. These expressions are found in the control declarations.
The use of controls is illustrated above and by looking at another example. In the translator SIMPLE_EXPRESSION (page 288), the last alternative in its production:

```
...
control N_OPS integer #OP;
...
control NEG boolean $S = '-$;
...
< < N_OPS(L) →
  [ if OP.L = 'OR' | ~COMPATIBLE(T_TYPE.(L+1), INTEGER)
    then fail
    endif;
    control ADD boolean OP.(N_OPS + 1 - L) = '+' ]
< ADD → 'ADD' | 'SUBTRACT' >+ >
< NEG → 'NEGATE' ? > ^
TO_INT(T_KIND.1: T.1) ^
< NEG → ')'? > ^
< N_OPS(M) →
  < ',$ TO_INT(T_KIND.(M+1): T.(M+1)) ^ '$' >+ >
> produces strings with the syntax:
('ADD(' | 'SUBTRACT(')+
  'NEGATE('? TO_INT(TERM) ')')? 'SUBTRACT('? TO_INT(TERM) ')')+

Because the same control, NEG, is used in both cases in the option terms, the two optional syntactic units, 'NEGATE(' and ')' must either both occur or both not occur, in any particular instance of the above syntactic construct. Similarly, both instances of the repetition operator '+', because they are used with the same control, N_OPS, indicate that their left arguments are repeated the same number of times on both occasions of its use. The values of NEG and N_OPS are set in their respective control declarations.

There are cases where separate instances of a single syntactic unit produces the same text and others where it produces different text. Knowledge of the context sensitive syntax requires these different cases to be identified. This is illustrated with a portion of the syntax of the emission from the translator CALL_STATEMENT (page 300). The emission of this translator starts off:

'bind' SCOPE_NAME 'to create_basket next'
'enclosing' SCOPE_NAME 'is' SCOPE_NAME 'next' ...

Three instances of the syntactic unit SCOPE_NAME appear. But because the first two come from the same variable they must be the same string of characters in any instance of the emission from the translator. To
distinguish the various uses of SCOPE_NAME suffixes are added; thus:

'bind' SCOPE_NAME1 'to create_basket next'

'enclosing' SCOPE_NAME1 'is' SCOPE_NAME2 'next' ...

This indicates that the two instances of SCOPE_NAME1 represent the same string in the two contexts which conforms to the syntactic unit SCOPE_NAME; while SCOPE_NAME2, in general, represents a different string from SCOPE_NAME1 conforming to the same syntax. It is possible for SCOPE_NAME1 and SCOPE_NAME2 to have the same string value but in most cases it would be required that they have different values. (In this particular case, an illegal GLADYS construct would be produced if SCOPE_NAME1 had the same value as SCOPE_NAME2.)

Despite these mechanisms, the context sensitive syntax of the output stream will, in general, require some calculation akin to symbolic evaluation or mechanical program analysis (see section 2.9) to be determined completely. The following features of SYNTRAN are included in order to support the symbolic evaluation of SYNTRAN programs.

Since it is easier to determine the value of a variable at some point in a program if assignments to it are restricted, identifiers and their associated variables are given "use" modes. Use modes are an extension of the idea of normal variables, "constant" variables and input and output variables. The use mode of an identifier is defined
when the identifier is declared and this mode is inherited by the variable it denotes and all variables contained in that variable in the case where it is a node variable. These use modes, as their name suggests, restrict the use that can be made of variables. In particular the use modes E and C are very important in allowing the syntax of the emission of productions to be deduced simply. Variables with these use modes have their value set when they are declared and are not thereafter allowed to be assigned.

To allow a variable in a node to be accessed by a single identifier, rather than by selection, aliasing of a node variable is allowed in such a way as to ensure that the use mode of the variable is not violated. This is achieved by the sharing that takes place in assignment statements with nodes with use mode V and by the alias statement for nodes with use mode R or W.

Global node variables are not allowed to be passed as parameters to functions, translators or macros. This removes the possibility of hidden side effects by uncontrolled aliasing and ensures that the values of formal parameters (which have use mode R) do not change during the invocation of the function, translator or macro where the formal parameters occur.

Finally, the simple scoping rules for identifiers and the disallowance of nested declarations reduces the complexity of any
algorithms for determining the value of any variable by inspection of the SYNTRAN source code.

4.2.3.3 An Example.

The nature of the object denoted by ST in the translator EXPRESSION (pp291-293 in the SYNTRAN/GLADYS definition of Pascal in appendix six) will be deduced from that SYNTRAN program. Specifically, given a SYNTRAN description of the type TYPE_NAME1 (mapping some object into a GLADYS object), an intermediate (S-level) representation for TYPE_NAME1 is developed. This will demonstrate how symbolic evaluation and analysis of the output stream of a SYNTRAN program can reveal the pattern of usage of GLADYS constructs and hence the structure of L. The techniques introduced here are used extensively in the next chapter. To make the demonstration more meaningful the reader should have a knowledge of the GLADYS types described in section seven of [EdwaSSOd] (appendix three). From the translator EXPRESSION (page 291), the following portion of GLADYS syntax can be produced by using the methods described above; the object ST appears as TYPE_NAME2.
Chapter 4 - SYNTTRAN - The Translating Language

\[\text{'\(\leftarrow\) TYPE_NAME1[' ENTER('STRING_COMPARE')]}\]

\[\text{'bind S_1 to create_cell(' TYPE_NAME2', ')}\]

\[\text{SIMPLE_EXPRESSION1')} next'\]

\[\text{'bind S_2 to create_cell(' TYPE_NAME2', ')}\]

\[\text{SIMPLE_EXPRESSION2')} next'\]

\[\text{'bind I_ to create_cell(' TYPE_NAME3', ')}\]

\[\text{TYPE_NAME3 '<1>}' next'\]

\[\text{'\(\leftarrow\) TYPE_NAME1['} \]

\[\text{'NEQ(#S_1@.I_@, #S_2@.I_@)→ ' \(\leftarrow\) TYPE_NAME1 '<O> exit'\]

\[\text{'EQL(#I_@, UNSN1')}→ ' \(\leftarrow\) TYPE_NAME1 '<1> exit'\]

\[\text{'assign' TYPE_NAME3 '<INC(#I_@)> to I_ loop'}\]

\[\text{'\]} next'\]

\[\text{EXIT('STRING_COMPARE') '}\]

Underlines are used in the identifiers to the temporary cells so that no conflicts can occur with source language identifiers. The emission statement:

\[\text{emission ST is TYPE_NAME: STRING_TYPES.LENGTH}\]

indicates that TYPE_NAME2 (ST) has the syntax of TYPE_NAME and that the values of the branches of the node STRING_TYPES are the values of TYPE_NAME2. The branches of STRING_TYPES are given various values throughout the definition and are used in a GLADYS binding statement in the distinguished translator PASCAL_PROGRAM (page 305). They are in the GLADYS construct (TYPE_NAME suffix 2 is used for consistency):
'bind' TYPE_NAME2 'to structure(' TYPE_NAME4 ';'

        TYPE_NAME5 ') next'

So TYPE_NAME2 represents a structure type. This could have been
deduced from the fact that the dereferenced identifiers S_1 and S_2,
bound to cells of TYPE_NAME2, were used as the left operand to the
selection operation, viz:

S_1@.I@

in the translator EXPRESSION but this route leads no further.

TYPE_NAME5 (CHAR_TYPE) has the value of CHAR.'NAME' (deduced from
the appropriate emission statement) and appears in the GLADYS
construct:

'bind' TYPE_NAME5 'to interval(0,' UNSN2 ') next'

where UNSN2 has the value CHAR_SET_MAX which is equal to
CHAR_SET_SIZE - 1 and CHAR_SET_SIZE is given some implementation
defined value (page 262), perhaps 128 to support the ASCII character
set. This interval type (CHAR_TYPE) is the base type of the structure
type being determined (TYPE_NAME2). The indexing type, TYPE_NAME4
(SITN) is defined immediately before TYPE_NAME2 in translator
PASCAL_PROGRAM.

'bind' TYPE_NAME4 'to interval(1,' UNSN3 ') next'
Thus the indexing type is a simple interval type also.

Note that, so far, the required symbolic evaluation has been very simple, merely consisting of determining where values of certain variables are used to produce GLADYS output. This is simplified by the guarantee that once a variable, which has to have use mode E to be used in producing GLADYS output directly, has been declared and given a value, that value cannot be changed until after it has been used.

To complete the determination of TYPE_NAME2, the possible values of UNSN3 have to be found as they dictate the number of values of the indexing type of TYPE_NAME2. If this value is limited to "small values" it may make a difference as to how values of TYPE_NAME2 are represented. Thus an upper bound on the value of UNSN3 would be useful. UNSN3 appears as SL which has as values the values of the branches of the node SLS which, in turn, are the selectors of the node STRING_TYPES. This is all deduced from statements in the translator PASCAL_PROGRAM. The node STRING_TYPES is given values in the translators CONSTANT (page 268), UNSIGNED_CONSTANT (page 280) and ARRAY_TYPE (page 273). In the translators CONSTANT and UNSIGNED_CONSTANT the selectors for STRING_TYPES are values of the variable LENGTH which has values the length of the string values emitted by the translator STRING (page 267). The syntax of emitted strings from the translator STRING is just:

\[ \text{ALL_CHAR}^* \]

where \text{ALL_CHAR} is the set of all characters except 'eol'. The number
of characters emitted by STRING is governed by the value of LEN in the translator STRING. The value of LEN is given by the expression:

\[ \#L - 2 - \frac{NSQ}{2} \]

L is the value of a token matched in the Pascal source code and NSQ is a variable initialized to 0 and conditionally incremented in a for loop. Since NSQ is zero or greater and is subtracted from the expression the upper bound of the expression is just:

\[ \#L - 2 \]

This gives some indication as to the maximum size of UNSN3 as the length of L certainly cannot be longer than any Pascal source program and, in fact, since L represents a value conforming to the syntax of LITERAL (page 261) which cannot contain 'eol's, cannot be longer than the length of a Pascal source program line (minus 2)! Again, note that the reasoning so far has been well directed, each step leading to the next in an obvious manner although some steps have been slightly more difficult than previously in that some knowledge of loop and conditional statements has been required.

As stated above, the node STRING TYPES is also given values in the translator ARRAY_TYPE. Again, the selectors are values of a variable LENGTH which, in turn, has the value of STRUCT.'INDEX'. 'MAX'. Moreover, selectors of STRING TYPES have the values of STRUCT.'INDEX'.'MAX' only on the condition IS_STRING_TYPE(STRUCT). This condition does not directly affect the possible values of STRUCT.'INDEX'.'MAX' but could do so indirectly by applying restrictions to values of other branches of the node STRUCT. In fact, the restrictions to the value of
STRUCT are that the:

<table>
<thead>
<tr>
<th>Branch Selected by</th>
<th>Has the Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'FORM'</td>
<td>'ARRAY'</td>
</tr>
<tr>
<td>'BASE'</td>
<td>CHAR</td>
</tr>
<tr>
<td>'INDEX'. 'FORM'</td>
<td>'SUBRANGE'</td>
</tr>
<tr>
<td>'INDEX'. 'MIN'</td>
<td>1</td>
</tr>
<tr>
<td>'INDEX'. 'BASE'</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

The only assignment to STRUCT in ARRAY_TYPE gives STRUCT. 'FORM' the value 'ARRAY'. STRUCT. 'BASE' is given the value of BASE_TYPE. BASE_TYPE is given the value of STRUCT or BASE_ENTRY. The value of STRUCT can never be CHAR because CHAR. 'FORM' has as its value, 'SCALAR', which is set at initialization and, as found before, STRUCT. 'FORM' has the value 'ARRAY'. Therefore, any values given to STRUCT. 'BASE' must come from BASE_ENTRY. BASE_ENTRY is given a value from the translator TYPE (page 274). Any values given to STRUCT. 'INDEX' come from INDEX_ENTRY. (N_INDICES + 1 - 1) which in turn is given a value from the translator SIMPLE_TYPE (page 271).

At this point the search for possible values of STRUCT. 'INDEX' . 'MAX' branches into many paths from just two. They are:

1. what values can be given to STRUCT. 'INDEX' . 'MAX' when STRUCT. 'INDEX' . 'FORM' has the value 'SUBRANGE', 'INDEX' . 'MIN' has the value 1 and 'INDEX' . 'BASE' has the value INTEGER, and

2. can STRUCT. 'BASE' be given the value CHAR?

If the answer to the latter is 'no', then the former is vacuous since
the value of STRUCT.'INDEX'.MAX' is only used on condition IS_STRING_TYPE(STRUCT). To keep the length of this, already lengthy, analysis to a minimum, only one of the successful paths, chosen using knowledge about Pascal, will be followed. Other paths will lead to the same or less strong conclusions or lead to 'dead-end's.

The value returned from TYPE can also be returned from the translator SIMPLE_TYPE (page 271); this is the single path that will be followed. Firstly, the path is followed from SIMPLE_TYPE to TYPE_ID (page 269) which returns the value returned by USED_ID (page 266) selected by 'TYPE'. The value returned by USED_ID is the result of the function SEARCH_ID (page 263) which returns a symbol table entry. The value of CHAR (set once and not changed) has been explicitly placed in a symbol table entry selected by 'TYPE' during initialization. Thus STRUCT.'BASE' can have the value of CHAR. Secondly, from SIMPLE_TYPE the path to the translator SUBRANGE_TYPE is chosen to investigate the values of STRUCT.'INDEX' because STRUCT.'INDEX'.FORM' is required to be 'SUBRANGE'. In the translator the value returned is a node with:

branches: with values:
'FORM'  'SUBRANGE'
'BASE' L_TYPE
'MIN' L_VALUE
'MAX' R_VALUE

L_TYPE, L_VALUE and R_VALUE are values returned by the translator CONSTANT (page 268). In CONSTANT, L_TYPE is given the value of TYPE and L_VALUE and R_VALUE are given the value of VALUE. One of the
possibilities is that TYPE is given the value INTEGER and VALUE the value of UN. UN has the syntax UNSN so can represent any non-negative integer (up to the value of MAXINT). Thus L_TYPE can have the value of INTEGER, at the same time L_VALUE has the value 1 and R_VALUE has any value between 1 (it is specified that L_VALUE <= R_VALUE) and MAXINT.

This completes this path and, in fact, makes all other paths insignificant. This is because the value of R_VALUE, hence STRUCT.'INDEX'. 'MAX', hence LENGTH, hence UNSN3, and hence the number of elements in the indexing type, TYPE_NAME4, of the structure type, TYPE_NAME2, can be any value from 1 to MAXINT. In any reasonable implementation the value of MAXINT would be "large". Thus the hope that the representation of TYPE_NAME2 (the string types of Pascal) could be specially treated because of a small number of elements is dashed.

The nature of TYPE_NAME2 (ST) has been determined: it is a structure type with constructing type an interval type with CHAR_SET_SIZE values and with indexing type an interval type with between 1 and MAXINT values. Many other examples in the next chapter display the use of the ability to determine the syntax of the output stream of a SYNTRAN program.
4.3 Specifying the L Compiler from the SYNTRAN Program.

The method by which the L compiler can be generated from a SYNTRAN program was described in section 3.2.3.2. It remains to show that the replacement of GLADYS language constructs with S-operators as proposed is feasible. By dividing the GLADYS constructs produced by a translator into literal and non-literal elements and replacing the literal elements with tentative S-operators an initial guess at the required S-operators can be made. Those non-literal elements which represent other GLADYS constructs (expressions, for example) can be considered to be arguments to the tentative S-operators, that is, the value of these arguments is determined when the operators are executed (at "run-time") and are usually supplied to an S-operator implicitly; on top of a stack or in machine registers, for example. The non-literal elements which represent just identifiers, and the like, whose value can be determined at "compile-time" can be considered to be parameters to the S-operators either as part of the operator or as an extension to the operator. An example is the function call operator; the arguments to the function are expressions evaluated just before the call of the function whose values are supplied explicitly, whereas the name of the function is determined (logically) at compile-time and will be an explicit argument, often an address immediately following the operator in the target code sequence.
By combining this information with the literal elements corresponding to the tentative S-operators, those which become redundant can be removed entirely or left to act as labels marking positions in the S-code to where jumps might be made and the others are defined in terms of GLADYS. The simple control structure the while statement illustrates this clearly. The emission from the translator WHILE_STATEMENT (page 293) is:

`'[* TO_INI(EXPRESSION) '→ [' STATEMENT '] loop]'`

where EXPRESSION and STATEMENT are syntactic entities and TO_INT is a macro which produces GLADYS code that will give the value of EXPRESSION. The expression must evaluate to an integer value of zero or one. Three S-operations can be named W1, W2 and W3, say, and used to replace the literal parts of the emission from the translator:

W1 TO_INI(EXPRESSION) W2 STATEMENT W3

with W1 the string `[', W2 the string `'→ [' and W3 the string `'] loop]'`. The actual definition of W1, W2 and W3 depends on the decisions regarding the "run-time" structure of S which is influenced greatly by the target machine. As an example of a definition of these operators, in a stack-based run-time structure they could be defined as follows:

W1 No action, merely acts as a marker for W3.

W2 If the value on top of the stack is true then pop the stack
and continue execution with the statement, otherwise pop the stack and continue execution after W3.

W3 Continue execution at W1, also acts as a marker for W2.

The approach of replacing portions of GLADYS constructs with S-operators suitably defined can be extended slightly to allow S-operators to be placed between two constructs concatenated together in the production of a translator. In fact, the approach relies only on the condition that "non-terminal" syntactic units such as EXPRESSION and STATEMENT above do not have to be split by an operator to represent the semantics of the complete construct. Since this situation cannot be represented in SYNTRAN it can cause no problems. Thus the SYNTRAN program itself can be used to specify the L compiler.

4.4 Implementation.

Early versions of SYNTRAN were implemented on a Univac 1100 [BorgB78] using the SNOBOL4 [GrisR71] language. The SNOBOL4 system used was modified by the author to use the ASCII character set and make many other improvements [EdwaS80a]. A SNOBOL4 program was written which translated the SYNTRAN source program to a SNOBOL4 target program. SNOBOL4 was chosen for both purposes because of its pattern matching ability and the useful TABLE data structure, both of which facilitate translation of programming languages. Note that the SNOBOL4 target program is a translating program.
4.5 Original Features.

The following features in SYNTRAN are claimed to be original ideas of the author:

1. The "use modes" of variables;
2. The encapsulation of the string producer and consumer in the translator construct;
3. The separation of the different semantics in the 'assign' and 'alias' statements used in conjunction with the "use modes";
4. The production construct as a practical programming construct developed from the ideas of generative grammars.
GLADYS is the target language of SYNTRAN program translations used to define the semantics of L. GLADYS is described in detail in [Edwa88Od] which is reproduced as appendix three. This chapter discusses the aims of GLADYS and the GLADYS features which attempt to satisfy those aims.

5.1 Aims.

The overall aim for GLADYS is that it be a good target language for the semantics defining translations of SYNTRAN and that it facilitate the mechanical analysis of such translations to determine the structure of the language being defined (see section 2.9). GLADYS must thus provide a framework for describing high-level languages. This framework should not be machine dependent, or even machine oriented, otherwise the language definition will be affected by this orientation. This overall aim is split into four areas for separate discussion:

1. Data structures.
2. Environment, binding and storage allocation.
3. Control structures.
4. Routines.
Provided GLADYS is complete in the sense that it can be used to specify any computation that a Turing machine can carry out, then, theoretically, GLADYS can be used to describe all possible programming languages. For practical purposes, GLADYS was designed with the Pascal-like languages in mind. This makes non-procedural languages (like PLANNER [HewiC72, SussG71]) and specialized languages (like SNOBOL4 [GrisR71] and C [KernB78]) awkward and complex to describe with GLADYS. This is because GLADYS lacks the specialized features in these languages which have to be programmed explicitly and often also because of the strict typing required by GLADYS (see section 5.2.5). The former characteristic is common to all formalized language description methods.

5.2 Data Structures.

The data structuring facilities in GLADYS are abstract and basic and were chosen to form a primitive basis for high-level language data structures. Data typing is strict and all type "conversions" are explicit; thus anonymous types are not allowed. This makes GLADYS program segments easier to analyze mechanically. The primitive storage items are called "cells". A cell may contain references to other cells or itself or values of what are called the "constructed" types. Cells are "typed" in that a particular cell can only take values of a particular type which is specified when the cell is created. The constructed types consist of the "interval" types, the "structure"
types and the "union" types. The references and these types are used to model the data structures in L.

5.2.1 References.

Recursive types are allowed in GLADYS, so that the reference type is not required for the purpose of constructing such types. Cell references are used, however, to:

1. model the reference or pointer types in L;
2. allow specification of call-by-reference parameters to routines, procedures etc. and
3. allow shared data.

There is no restriction inherent in a cell which references another cell as to the type of the referenced cell. If such a restriction in L cannot be resolved at "translation time", that is, it cannot be resolved by the context sensitive restrictions of L enforced by the SYTRAN translating program, then the GLADYS model of such a pointer type would include a cell specifying the type of the referenced cell as well as the cell containing the reference.
5.2.2 The Interval Types.

These types are the primitive constructed types and are so named because they are isomorphic to finite intervals of the set of integers. Cells of an interval type are used to model variables of "simple" or scalar types in L. The number of different values that can be stored is always finite. This requires that, for example, the maximum values of a language "integer" data type be specified in a specific SYNTRAN/GLADYS definition, but may be left unspecified for a more general, but incomplete, description. Operations on interval type values are carried out using integer functions with integer arguments which are defined externally to GLADYS. Thus the semantics of the addition of two "integer" type values of L is defined by a mathematical function. The treatment of a "real" or machine "word" data type of L would be very similar provided the "word" type was not used as a machine address (see section 5.2.5 below). In these cases the values of the interval type in GLADYS would bear little relationship to the normal interpretation of these values, especially as "real" values, and the only properties of the interval type used would be the number of different values and perhaps their ordering. The interpretation of the interval type values as "real" or "word" values would be enforced entirely by the externally defined functions. The use of such functions removes the responsibility of defining them from GLADYS and places it with the well-defined world of mathematics.
Other interpretations can be placed in the interval types by the translating program. This is the case with Pascal [JensK78] user defined scalar types (enumerations [HoarC72]); the compiler forces a specific type interpretation on values that are, in fact, implemented as a subset of the numbers representable as "integers". It was decided that to have a similar mechanism in GLADYS was an unnecessary complication. The character and Boolean types in most Ls would also be modelled using the interval types.

Examples are:

bind BOOLEAN to interval(0, 1)
bind INTEGER to interval(-32768, 32767)
bind CHAR to interval(0, 127)
bind UNDEFINED to interval(0, 0)

5.2.3 The Union Types.

A union type is the discriminated union [HoarC72] of its "member" types. They may be used to model the variant part of variant records in Pascal, union types in ALGOL 68 [WijnA76, LindC77] and similar structures as well as the dichotomy between the undefined value and the defined values of a variable. The last is accomplished by treating the normal values of a variable as one type and the "undefined" value as the single value of a different type. The variable then has the
type which is the union of these two types and is initialized to the single value of the "undefined" type. This is logically consistent as operations on the value of the variable can only be carried out on "defined" values. When the cell modelling the variable is accessed the attempted conversion from its union type to the type of "defined" values will determine whether the variable has a "defined" value or not. For example:

\[
\text{bind INT MODE to union(INTEGER, UNDEFINED)}
\]

5.2.4 The Structure Types.

The structure types may be used to model arrays, records and other structures in a simple manner since they are the set cross product types of other "constructing" types coupled with an indexing type. The powerset type [HoarC72] or the set types in Pascal can be modelled using a structure type constructed of a number of instances of a two valued type giving, in effect, an array of Boolean, or by using an interval type with operations defined on it such that it has a set-like interpretation. The former provides for set types of varying size and is more flexible than the latter which reflects the common implementation method used in most Pascal compilers. For example, the type involved in the FORTRAN declaration "INTEGER X(10)" would be modelled by ARRAY_TYPE as follows:
bind ONE_TO_TEN to interval(1, 10) next
bind ARRAY_TYPE to structure(ONE_TO_TEN; INT_MODE)

The structure types and the union types together provide the means to define recursive types without the use of pointers. A recursive type may be constructed by using a union type as a part of a structure type where one of the members of the union type is the structure type itself. This requires the two type constructors, the union type constructor and the structure type constructor, to be in the same parallel statement unit. Although such a definition specifies a possibly infinite structure only finite objects of that type can be created. (This assumes only that the program is finite in length.) This method can be used to model the sequence type data type [HoarC72], for example, such as files in Pascal. For example:

bind HEAD_REST to interval(0, 1) next
bind EOF to interval(0, 0) next
[ bind FILE to union(FILE_ELEMENT, EOF) with
bind FILE_ELEMENT to structure(HEAD_REST; CHAR, FILE) ]

FILE_ELEMENT is a record type with two fields; one of type CHAR and the other of type FILE (see section 7.3.3 in appendix three).
As with interval types, a compiler can place a particular interpretation on a structure type. For example, a COBOL [ANSI74] record can be modelled by a single structure type with a character type base type with the compiler allowing different ways of specifying various contiguous elements of the structure type. These ways of specification would form a tree-like naming structure.

5.2.5 Modelling Other Types.

One of the more complex types found in some languages is the string. Some languages treat a string as an array of characters (e.g. Pascal) or as descriptors referring to an array of characters, as in SIMULA's [Dahl066] "magic box"s and "text" type values. SNOBOL [GrisR71] strings are perhaps the most difficult to model as the string type is primitive in this language. The SNOBOL string can be modelled, in isolation from the rest of the language, as a union of structure types each corresponding to strings of a particular length (there being an implementation defined maximum length of a string). These structure types would themselves each consist of the characters of the string value.

It is envisaged that the more complex data structures of other high-level languages will be modelled by building up data structures from the basic constructed types of GLADYS in a similar way to that just described. On the other hand, a simpler, low-level type, the
machine word, where it may be used as a machine address and as an argument to real, integer and other types of operations, can only be modelled in GLADYS by modelling the memory space by a structure type. A one-to-one relationship (or any other sort of relationship) cannot be set up between the reference type and any other type. This is because of the obvious machine dependency of addresses; adding one to a machine address may or may not give the address of the next word. This seems more reasonable when it is considered that a language which allows such operations is, strictly speaking, a different language on each different machine. A program may even have a different meaning depending on the physical configuration of a particular machine. A resolution of this problem is not attempted here.

5.2.6 Representation Issues.

Having described the primitive data types and methods of constructing new data types in L using GLADYS constructs, there is the problem of finding a suitable representation for them. Some decisions can be made without regard to the target machine, others cannot; only the former will be discussed.
5.2.6.1

The representation of a reference type is a target machine address in some form and is thus totally machine dependent.

5.2.6.2

The representation for interval types is relatively simple. They may be represented by a bit string long enough to represent the cardinality of the type (for example, using one bit to represent values in the range 1000..1001), the minimal representation, or a bit string long enough to represent the upper and lower bound values as integers (requiring 10 bits to represent values in the range 1000..1001), not, in general, a minimal representation. If both characteristics can be obtained with the same representation, which is quite often, it is the most efficient. In other cases a choice between one or the other depends on the instruction set and the addressability of the target machine and the actual values of the upper and lower bounds.
The representation of union types also falls into two general methods. The first is the more general; a cell of the union type would have the size of its largest member type plus a bit string long enough to represent a different value for each member type. That is, a discriminator field. This discriminator field may be omitted if the type security required can be enforced by some other method such as the syntax of L. If the member types are all interval types or other types with small cardinality, a minimal representation could be used which consists of just a bit string long enough to represent the number of different values that is the sum of the cardinalities of each member type.

The representation of structure types is the most complicated of the constructed types in general. When the cardinality of the indexing type is small, by some measure, as is the case when structures like Pascal records are being modelled, the values of the constructing types can be placed in juxtaposition. This is the usual representation for records and arrays in most language implementations. However, if the cardinality of the indexing type is not small, then a sparse representation is appropriate. The only commonly known high-level
language which would require such a representation for a normal
language feature is SNOBOL4 with its TABLE data structure.

Turning back to the modelling of the SNOBOL string type in the
last section, another model for this type is a structure consisting of
an interval type to hold the length of the string and a structure
consisting of the maximum number of characters possible in a string.
The representation of this model would require a sparse representation
to be practical. The knowledge that most strings are much shorter than
the maximum length cannot be deduced from the language definition.
Thus, SAGA might either "guess" that a particular structure should be
represented sparsely or extra information would have to be included in
the language definition. Neither of these topics is investigated in
this thesis.

Another option for representing structure types is to link single
elements or groups of elements together into a list. This may be
practical in situations where only sequential access of the elements
of the structure type is used.
5.2.6.5

Some abstract data types available in some high-level languages can be defined most naturally recursively e.g. lists in LISP [McCaJ62], files in Pascal and sequences and stacks in general. Since GLADYS allows the definition of recursive types the problem of the representation of these types arises. The size of the data items in these structures, the anticipated methods of usage and their relative frequency can all affect the representation chosen. The last factor, as mentioned above, cannot be deduced from a formal definition. Pointers are often used when the data items are large in comparison to the size of the pointers, or, in the other case, small individual data items might be grouped into blocks and then linked together by pointers. In any case this complex problem is not dealt with in this thesis. See [LowJ78] and references therein for more information.

5.2.7 An Example.

In section 4.2.3.3 it was shown how a SYNTRAN/GLADYS definition can reveal the structure of a particular data type, called ST (and TYPE_NAME2). The following proceeds further with this example by showing that the pattern of usage, if any, of that data type can also be revealed. This can then be used to determine a representation for that structure. The portion of GLADYS syntax appearing in section 4.2.3.3, from the translator EXPRESSION (page 291), is again used. The
object denoted ST (or TYPE_NAME2 in the analysis) is the string type, as deduced previously. Ignoring the syntax of SIMPLE_EXPRESSION, ENTER and EXIT which are not important in this example, the following analysis can be carried out to determine the operations involved on the values of data types specified by TYPE_NAME2.

It can be seen that there are still four unknowns occurring in the portions of GLADYS syntax: TYPE_NAME1, TYPE_NAME3, UNSN1 and INC. TYPE_NAME1 can be traced to:

'bind' TYPE_NAME1 'to interval(0, 1) next'

in translator PASCAL_PROGRAM in the same way as CHAR was traced in section 4.2.3.3. TYPE_NAME3 (SIT in EXPRESSION) has the value STRING_INDEX_TYPES.LENGTH and UNSN1 is also LENGTH. Again in PASCAL_PROGRAM the GLADYS output:

'bind' TYPE_NAME3 'to interval(1,' UNSN1 ')next'

can be found. Thus UNSN1 represents the largest value in TYPE_NAME3. From PASCAL_PROGRAM, the value of STRING_INDEX_TYPES.n is the name of the indexing type to the type named by STRING_TYPES.n. Thus TYPE_NAME3 must be the indexing type for TYPE_NAME2 (and the same as TYPE_NAME4). Finally, INC is an external function. Assuming the semantics of INC are known then the semantics of the operation described in the portion
of GLADYS extracted from EXPRESSION can be deduced from a knowledge of
GLADYS.

It is, indeed, an operation comparing two string values for
equality. This operation could have been modelled directly by a GLADYS
relational operator because an ordering is defined on structure types
where the constructing and indexing types are ordered. But the way
chosen provides a non-trivial example.

A method of usage of TYPE_NAME2 has been determined though it can
be viewed from two levels. At the lower level, TYPE_NAME2 is indexed
using increasing values of its indexing type (starting at the
smallest) comparing constructing type values until a pair is unequal,
in which case the expression yields a "Boolean" value zero, or until
all constructing type values have been tested, in which case the
expression yields a "Boolean" value of one. At the higher level this
operation can be viewed as just a comparison for equality. Whichever
view is more convenient can be used by SAGA to formulate
implementation decisions such as, for example, ensuring that
sequential access to the constructing type values in TYPE_NAME2 values
is facilitated or at least not unduly hindered. If the comparison
operation just described were not part of the language description the
question of efficient sequential access would not arise.
5.3 Environments, Binding and Storage Allocation.

These three important concepts used in high-level languages have been captured with the GLADYS basket. A basket is an object which may contain cells and bindings between identifiers and objects: cells, baskets and types. A basket may also be associated with another basket called the enclosing basket. Baskets act as environments and every process has a basket associated with it called the current basket. Identifiers used in that process are given a value by searching, firstly, the current basket, by default, for a binding with the identifier. If it is not found there, the search continues in the enclosing basket of the current basket and baskets enclosing that one until a binding for the identifier is found. When the enclosing baskets are set up statically, the identifier binding search just described models the normal block structured language environments as in ALGOL 60 [NaurP63] and its descendants. Since all baskets, including the current and enclosing baskets, are under the control of the GLADYS program, the environment can be adjusted dynamically to suit other environment strategies. The basket where the search for an identifier binding is to begin can also be specifically named with the occurrence of the identifier.

Cells are contained in baskets, allowing a simple way of controlling storage. The basket in which each newly created cell or identifier binding is placed can be specified (the default is the current basket) and baskets may be destroyed as a whole. The two
functions of baskets, that of storing identifier bindings and
controlling identifier binding searches and that of storing cells, may
be used quite independently. The normal block structured environments
and storage allocation scheme, though, is aptly modelled by combining
the two functions. On block exit, in an ALGOL 60-like block structured
language, identifier bindings and the storage associated with the
cells created in that block are destroyed. These two operations may be
combined and modelled in GLADYS by the destruction of a basket.

This flexibility and explicit control available are the
advantages of the basket concept over implicit scoping and storage
allocation schema and is the reason for its use in GLADYS.

To illustrate how baskets may be used and how they fit into the
scheme of symbolically evaluating a SYNTRAN/GLADYS definition the
storage allocation patterns of Pascal as revealed by the definition in
appendix six are explained below.

The methods of allocating storage can be split into four types
which shall be called static, stacked, dynamic and transient in this
and the next sections. Static storage allocation corresponds to the
type of allocation found in FORTRAN [ANSI66], with OWN variables in
ALGOL 60 [NaurP63] and STATIC storage in PL/I [ANSI76]. Stacked
storage allocation occurs with normal variables in ALGOL 60, Pascal
[JensK78] and with AUTOMATIC and CONTROLLED storage in PL/I. Dynamic
storage allocation [section 2.5, pp435-451, KnutD73] is the most
general method and occurs in creation of class instances in SIMULA [Dahl066], in newly allocated pointer based variables in Pascal and with all data storage allocation in LISP [McCaj62], APL [FalkA75] and SNOBOL4 [GrisR71]. Transient storage allocation as defined here is only found internal to the implementation of a language.

It is important to note that in the following it is assumed that there is only one expression or statement being executed at one time unless otherwise specified. That is, no parallel execution takes place and collateralaly evaluated constructs are evaluated one at a time.

The static storage allocation type (SAT) is the most restricted and is described first. Static data storage units (SUs) have the property that their size and number at any time during execution can be determined by inspection of the program source code text, that is, they can be determined at "compile time". Thus they can be allocated at the beginning of program execution and released at the end.

A transient SU has the property that no other SUs are allocated or released between its allocation and release. These are truly temporary SUs and occur in implementing some high-level language features such as the Case statement in Pascal. Here a transient SU can be used to hold the value of the expression as it is being compared against each case constant. A number of transient SUs may be allocated and released concurrently but apart from this different transient SUs may use the same physical storage, that is, they can be overlayed. The
actual implementation of a transient SU, because of its restricted properties, may thus be the same as a static SU for efficiency or as a stacked SU for simplicity and consistency with the rest of storage allocation mechanisms (as with most block structured language implementations) or even as a dynamic SU though this would usually be very inefficient.

SU's of the stacked SU chosen so that it in allocated and released in a class, all other SU's in the class are PL/I with languages for machines has its own stack of allocation of AUTOMATIC single stack found in Algol languages. The description of this program is indirect benefits of higher remuneration. Thus, by baskets being associated by enclosure. Stacked SU's are used explicitly in block structured languages for variables and implicitly for some temporary storage. For example, in the Pascal For statement some information has to be stored to indicate the state of each particular instance of a For statement execution. This normally
actual implementation of a transient SU, because of its restricted properties, may thus be the same as a static SU for efficiency or as a stacked SU for simplicity and consistency with the rest of storage allocation mechanisms (as with most block structured language implementations) or even as a dynamic SU though this would usually be very inefficient.

SUs of the stacked SAT are divided into classes, each class chosen so that it includes as many SUs as possible which can be allocated and released within it. The (SAT) class is only used by the first SU, all others are allocated and released from the (SAT) class. Though this is inconvenient, it is useful in languages for many SUs as possible which can be allocated and released implicitly in block structured languages for variables and implicitly for some temporary storage. For example, in the Pascal For statement some information has to be stored to indicate the state of each particular instance of a For statement execution. This normally
actual implementation of a transient SU, because of its restricted
properties, may thus be the same as a static SU for efficiency or as a
stacked SU for simplicity and consistency with the rest of storage
allocation mechanisms (as with most block structured language
implementations) or even as a dynamic SU though this would usually be
very inefficient.

SUs of the stacked SAT are divided into classes, each class
chosen so that it includes as many SUs as possible which can be
allocated and released in a "LIFO" manner on a stack. Thus a (stacked
SAT) class is characterized by the property that for each SU in that
class, all other SUs in that class, if allocated after the first SU is
allocated and before it is released, must then be released before the
first SU is released. More than one class may exist in a language
though this is unusual. Instances of languages with more than one
class are PL/I with its CONTROLLED storage and machine-oriented
languages for machines with multiple stacks. Each CONTROLLED variable
has its own stack of storage. PL/I also has one class for the
allocation of AUTOMATIC variables and this stack corresponds to the
single stack found in ALGOL 60, Pascal and other block structured
languages. The description of this property is facilitated in GLADYS
by baskets being associated by enclosure. Stacked SUs are used
explicitly in block structured languages for variables and implicitly
for some temporary storage. For example, in the Pascal For statement
some information has to be stored to indicate the state of each
particular instance of a For statement execution. This normally
The dynamic SAT is the least restricted. After allocation of a dynamic SU, other SUs of any SAT may be allocated and released in any order. It is possible that dynamic SUs are not released explicitly as the termination of a program implicitly releases all remaining allocated storage. SUs not belonging to any other SAT belong to the dynamic SAT.

A cell may belong to more than one of the static, transient or stacked SATs in which case it should be treated as of the one type which is most efficient. To determine the SAT(s) to which a particular SU belongs again requires the analysis of the SYNTRAN/GLADYS definition. The properties described above concerning each class can be used as the basis for SDRs for this purpose.

5.3.1 Explicit Storage Allocation.

The storage allocation primitives in GLADYS are the 'create_basket' and 'create_cell' functions and the 'destroy' and 'destroy_enclosed' statements. SUs are modelled by cells in GLADYS. Each instance of a cell in a SYNTRAN/GLADYS description corresponds, in general, to any number, including none, of SUs in the execution of a particular program. Storage is allocated by creating a basket and
then creating cells (data storage units) and placing them in the basket. Storage is released by destroying a basket explicitly named, as in the 'destroy' statement, or implicitly named by being enclosed in a specified basket as in the 'destroy_enclosed' statement. The point immediately following the 'create_cell' function in GLADYS corresponds to the point of allocation (PA) of the SU modelled by the created cell. The newly created cell cannot be referred to until after the 'create_cell' function is completed. Specifically, the evaluation of the initial value for the cell occurs before the PA. A number of 'create_cell' functions occurring in succession can be considered to be concurrent if the functions do not have any side-effects on one another, that is, the identifiers bound to the newly created cells are not referenced in arguments to the succeeding functions. The PAs of SUs corresponding to concurrent 'create_cell' functions can be re-ordered to suit the requirements of any analysis provided that any required order of evaluation of the initial values is not violated.

The release of an SU occurs when the basket containing the cell modelling that SU is destroyed. The actual point used as the point of release (PR), though, can be considered to be any point of execution between the last possible use of the SU before the destruction of the relevant basket and the destruction of the relevant basket itself. The PR may be chosen between these bounds, as above, to suit the requirements of any analysis. The "last possible use" of an SU, if required, must be established by analysis of the SYNTRAN/GLADYS description.
5.3.1.1 Transient Storage Allocation.

Two examples of cells of the transient SAT can be found in the Pascal description. One in the string comparison definition given in the translator EXPRESSION (page 291) and used as an example in sections 4.2.3.3 and 5.2.7 and the other in the translator CASE_STATEMENT (page 295).

In the former case, three cells are created and bound to $S_1$, $S_2$ and $I_1$. The creation of these cells corresponds to their PA and occurs after the basket in which they are placed has been created using the GLADYS code produced by the macro ENTER. This code is:

```
'bind' SCOPE_NAME 'to createbasket next'
'enclosing' SCOPE_NAME 'is current_basket next'
'current' SCOPE_NAME 'next'
```

This basket is destroyed when the code produced by the macro EXIT is encountered and in this case can be considered to be the PR of the cells without affecting the result. The code produced by the macro EXIT is:

```
'destroy' SCOPE_NAME
```

Between the PA and the PR of these cells no storage allocation or release can possibly occur. Apart from the use of the cells $S_1$, $S_2$
and I only a GLADYS 'loop' unit and three external functions are used (NEQ, EQL and INC). Thus the SUs modelled by these three cells belong to the transient SAT.

In the latter case, the syntax of the string emitted by the translator CASE_STATEMENT is:

```
ENTER('CASE_SCOPE')

'bind CASE_TEMP to create_cell(' TYPE_NAME1 ',
  TYPE_NAME1 '<' TO_INT(EXPRESSION) '>') next'

'[ ' ( 'OR(' 'EQL(' NUMBER ', '#CASE_TEMP')'
  ( ', EQL(' NUMBER ', '#CASE_TEMP'))')*
    '→' STATEMENT) / 'exit'

'] next'

EXIT('CASE_SCOPE')
```

Because the controls associated with the '*' operators are identical the same number of instances of their left arguments are generated in the production element in which they are a part so that, in this case, the parentheses will always match. The cell CASE_TEMP is created and given a value specified by EXPRESSION and this determines the PA of the SU modelled by the cell. The syntax of the emitted GLADYS code shows that after the execution of any STATEMENT, the GLADYS loop unit is terminated by the 'exit' statement separator. This implies that CASE_TEMP cannot be used after STATEMENT is executed. Also since CASE_TEMP does not conform to any syntactic unit of the source
language (specifically it is not a legal Pascal identifier) it cannot be used in any GLADYS code generated by STATEMENT (the identifier CASE_TEMP is not used elsewhere in the description). Other occurrences of CASE_TEMP can only occur in nested Case statements in which case a new basket named 'CASE_SCOPE' is created thus removing any possibility of conflict.

Thus the PR of CASE_TEMP can be considered to be a point that occurs before the execution of any code generated by STATEMENT and therefore no storage allocation or release occurs between the PA and any PR of the cell CASE_TEMP (only the external functions OR and EQL are used). Therefore CASE_TEMP denotes a transient SU.

Note that if the translator CASE_STATEMENT were changed so that the EXIT macro occurred before each STATEMENT, that is, the third last line of the string emitted above becomes:

\[ \rightarrow [ \text{EXIT(CASE_SCOPE)} \text{ } \text{'next'} \text{ STATEMENT '}] ) / 'exit' \]

then the analysis would have been simpler, since any PR of CASE_TEMP would have clearly been before any STATEMENTS and therefore any opportunity for usage of the cell CASE_TEMP. The problem of choice of methods available to the language definer is discussed in section 6.2.1 but a "proper" SAGA would be indifferent to such choices.
An SDR dealing with transient SUs simply tests the behaviour of cells created by each 'create_cell' function for the properties of a transient SU as described above and marks the PAs and PRs of those transient cells in the SYNTRAN/GLADYS description. These points can then be used to allocate and release high-speed registers or other temporary storage mechanisms suitable to the target machine. It is the responsibility of the interpreter generator to find all the transient SUs in a language description, which may require some analysis as in the Case statement above, to ensure that no opportunity is lost to increase the efficiency of the final implementation.

5.3.1.2 Static Storage Allocation.

An SDR dealing with static SUs is merely required to determine which cells belong to the static SAT (from the properties given above) and mark them so that the compiler produced by SAGA will have the ability to calculate the storage requirements of the static SUs in any source language program.

Because of the strong typing in Pascal the sizes of all SUs are known at "compile" time. Thus if the number of these SUs can also be calculated at compile time then they are all static SUs. The number of SUs becomes unknown at compile time when the depth of recursive function or procedure calls is governed by run-time (input data) considerations. Thus an optimizing Pascal compiler seeking to minimize
procedure call overhead could translate a Pascal program with no recursive function or procedure calls using static allocation for at least all SUs modelling Pascal variables.

Rules regarding the determination of static SUs have not been investigated except to note that the number and size of SUs allocated by the translator VAR_DEC (page 275) in the translator BLOCK (page 304) as it is called from the distinguished translator PASCAL_PROGRAM (page 305) are known. This is because the distinguished translator is invoked only once (implicitly) and does not appear elsewhere in the SYNTRAN program and hence these particular 'create_cell' functions (in the call of VAR_DEC) in the target GLADYS program can only be executed once. Other 'create_cell' functions and those in the translator VAR_DEC in different contexts can, in general, be called any number of times. Also, by inspection of the type related translators, the types of the cells created are built out of interval types which have known bounds at compile time (they use literal tokens UNSN, page 261, and NUMBER, page 261). Thus the cells created by GLADYS output from the translator VAR_DEC when it is called from BLOCK from PASCAL_PROGRAM are static. The following sub-section will show that they are also of stacked SAT; a compiler thus has the choice of implementing them in either way.
5.3.1.3 Stacked Storage Allocation.

The properties of SUs of the stacked SAT are more complicated than those of the static or transient SATs and, as a result, a stacked SU cannot be found by testing each instance of the use of a cell in the SYNTRAN/GLADYS description in isolation. All SUs must be considered together to determine the stacked SAT classes referred to above.

Each PA and PR of an SU modelled by a cell in a GLADYS program corresponds to two points in the SYNTRAN program (the SYNTRAN/GLADYS description) in the string expressions of emission statements. The occurrence of a 'create_cell' function call in the SYNTRAN program identifies a number of cells in an executing GLADYS program produced by that SYNTRAN program. The PAs and PRs of these cells correspond to two sets of points in the SYNTRAN program; one set corresponding to the PAs and the other to the PRs. These two sets are taken to form a vertex in a directed graph [section 2.3.4.2, pp371-375, KnutD73]. One vertex is associated with each and every one 'create_cell' function call in the SYNTRAN program. For a vertex v, A(v) is the set of points in the SYNTRAN program corresponding to the PAs of the cells associated with that vertex and R(v) is the similar set of points corresponding to the PRs of the associated cells.
An arc between two vertices represents the relation that cells associated with the second vertex are allocated and released between the allocation and release of cells associated with the first vertex; the arc being directed from the first vertex towards the second. This implies that the SUs modelled by the cells associated with the second vertex can be allocated on a stack above the SUs modelled by the cells associated with the first vertex. More formally, for two vertices, \( v_1 \) and \( v_2 \), let \( c_1 \) and \( c_2 \) be any cells such that the PA of \( c_1 \) corresponds to a point in \( A(v_1) \) and the PA of \( c_2 \) corresponds to a point in \( A(v_2) \). Hence the PR of \( c_1 \) and the PR of \( c_2 \) correspond to \( R(v_1) \) and \( R(v_2) \) respectively. If, for all such cells \( c_1 \) and \( c_2 \), the PA of \( c_2 \) occurs between the PA of \( c_1 \) and the PR of \( c_1 \) if and only if the PR of \( c_2 \) occurs between the PA of \( c_1 \) and the PR of \( c_1 \) then there is an arc directed from \( v_1 \) to \( v_2 \). Where there is a significant choice to be made regarding the relative order of PAs and PRs allowed by the choice of actual position of the PAs and PRs as specified above they should favour the existence of relations (arcs in the graph). This leads to more SUs in the stacked SAT and hence a more efficient implementation.

From the directed graph, the stacked SAT classes can be determined and each vertex can be assigned to a particular class. The graph is first split into sub-graphs such that each sub-graph is connected (all vertices are joined by arcs, ignoring the direction of the arcs) and the union of any two sub-graphs is not connected. Isolated vertices (those which have no arcs leading in or out) correspond to cells which are not of the stacked SAT and are excluded.
from further discussion in this sub-section.

A sub-graph corresponds to a stacked SAT class if in all cases where two 'create_cell' functions (both corresponding to vertices in the sub-graph) can be executed one after the other without the intervening execution of another 'create_cell' function (corresponding to another vertex in the sub-graph) then the two vertices are joined by an arc leading from the vertex whose corresponding 'create_cell' function is executed first. This ensures that all SUs allocated by 'create_cell' functions corresponding to vertices in the sub-graph can be placed on one stack. This also takes into account the possibility of parallel and collateral execution. This condition is trivially satisfied if there are arcs leading from every vertex to every other vertex in the sub-graph. Other cases have not been investigated.

This method can find the number of stacks required for a language that can be determined from the definition of the language. Other stacks may be required in other circumstances. For example, if the possibility of parallel execution is ignored in determining the stacked SAT classes, then each instance of a process allocating cells in that class will require one stack for that class. This rule is fairly straightforward. Another example is the CONTROLLED variables in PL/I where each variable is associated with a stack of SUs. The discovery of this use of a stack from the language description falls outside the capability of the above method and would be sought for when dealing with dynamic SUs.
The graph obtained from the SYNTRAN/GLADYS description of Pascal can be depicted as follows:

The contexts in which "arcs" leading to the syntactic unit EXPRESSION occur are listed below, numbered to correspond to the numbered arcs:

1. the expression constituting the right-hand side of the Assignment statement;
2. the upper and lower limit expressions;
3. the expression in the Case statement, the Boolean expressions in the If, While and the Repeat statements and array indexing expressions in the variables of With statements and of the left-hand side of Assignment statements;
4. expressions that are actual parameters to procedures; and
5. expressions that are actual parameters to functions.

A number of points should be noted about this depiction of the graph. Firstly, it is drawn so as to be as easy as possible to follow from the SYNTRAN/GLADYS description. To this end the following steps have been taken:

1. The syntactic units EXPRESSION and STATEMENT have been explicitly included and arcs passing through them have been split into more than one "arrow". Thus there is an arc leading from the FOR_STATEMENT vertex to the FUNCTION_CALL vertex which happens to pass through the syntactic units STATEMENT and EXPRESSION.

2. Only those arcs which are deduced directly from the description are drawn, that is, as the calls from translator to translator are followed the search for the destination of an arc stops at the first vertex found. The complete graph is the transitive closure of the graph depicted and has arcs leading from all vertices (except the one isolated vertex) to all other vertices but was deemed to be too complex and untidy to serve as a useful diagram.

Secondly, as noted above, the collateral evaluation of parameters to functions and procedures and sub-expressions has been assumed to have been "serialized" to the extent that each parameter expression and sub-expression is evaluated completely before another is evaluated. (Most Pascal compilers make this simplification, at least.) This allows cells allocated and released during evaluation of
parameters and sub-expressions to use the same stack. Thirdly, the potentially disruptive effects of the Goto statement are shown to be innocuous in section 5.4.2.2 below. Finally, arcs pointing to the node with local variables in BLOCK are traversed by a process "call", that is, the cells are allocated at that node when a procedure or function is called in Pascal. The cycle STATEMENT, ASSIGNMENT_STATEMENT, EXPRESSION, FUNCTION_CALL, BLOCK, STATEMENT will now be described in detail as an example. This is achieved by examining the SYNTRAN/GLADYS description of Pascal to determine that the relation holds between the vertices as specified.

One sort of statement in Pascal is the Assignment statement (translator ASSIGNMENT_STATEMENT is on page 299). To ensure the left-hand side of the Assignment statement is evaluated before the right-hand side a temporary cell (L_H_S) is created and the variable on the left-hand side evaluated and stored in it. The expression (the right-hand side) is then evaluated and placed in the cell specified by the value of L_H_S. No cell named L_H_S is referred to outside the translator ASSIGNMENT_STATEMENT and wherever recursion involves this translator a new basket named TEMP_SCOPE is created to contain any other instance of a cell named L_H_S. Thus SUs modelled by L_H_S cannot be affected by the evaluation of the expression and hence the expression is evaluated between the PA and the PR of L_H_S. An expression may consist of a function call and this leads to the next vertex.
A function call (see the translator FUNCTION_CALL page 282 and an example of its syntax given in section 5.5.3 below) causes the creation of a cell named func-name_INSTANCE in a newly created basket named by LOCAL_SCOPE. Func-name is the name of the called function. This basket is destroyed at the end of the function call by the 'destroy' statement which is the last statement in the expression unit modelling the function call. The complication here is that a new process, referred to by func-name_INSTANCE@.func-name_BASE<O>, is started. This process has the basket named by LOCAL_SCOPE as its initial current basket. The form of this process can be found by noting that the name of the type of func-name_INSTANCE is a string called FUNC_NAME (with the value denoted by "func-name" above), the emission from the translator USEDID. The symbol table entry for this string is returned by USEDID as ENTRY. The value of ENTRY.'MODE' must be 'FUNC' (or the translator FUNCTION_CALL will fail) so that the value of ENTRY can only have been specified by the translator FUNCTION_DEC (page 277) because the variable, also called ENTRY, in that translator, is the only instance where a symbol table entry with branch selected by 'MODE' is given the value 'FUNC'. Using the common symbol table entry called ENTRY in both translators a correspondence can be set up between the values in both translators. For example, the variable NEW_SCOPE in FUNCTION_DEC has the same value as LOCAL_SCOPE in FUNCTION_CALL because they have the same value as the symbol table entry, ENTRY, branch selected by 'LOCAL_SCOPE'. The process type started in the function call consists of the following syntax:
FORMAL PARAMETER LIST?

'bind' func-name 'to'

func-name '_INSTANCE@.' func-name '_BASE<-2> next'

BLOCK 'next'

'transfer' func-name '_INSTANCE@.' func-name '_BASE<-1> @'

where func-name is the name of the called function as before. No storage is allocated in FORMAL PARAMETER LIST or the bind statement so these parts may be ignored. The value of func-name_INSTANCE@ .func-name_BASE<-1>@ is specified in the translator FUNCTION_CALL as 'this_process', that is, the process which started the process currently being investigated. Thus the 'transfer' statement in the called process is, in effect, a "return" operation and execution continues immediately after the 'transfer' statement in the calling process. This leaves only BLOCK to be investigated further.

Emissions from the translator BLOCK (page 304) have the syntax:

TYPE_DEC* VAR_DEC*
(PROCEDURE_DEC | FUNCTION_DEC)*
STATEMENT_SEQUENCE.

TYPE_DEC, PROCEDURE_DEC and FUNCTION_DEC only specify the setting up of bindings between identifiers and types. VAR_DEC, on the other hand, specifies the setting up of bindings between identifiers and newly created cells which are placed in the current basket.
Chapter 5 - GLADYS - The Target Language

VAR_DEC =

(TYPE

('bind' IDENT 'to create_cell(' TYPE_NAME1 ','

('no_cell' | TYPE_NAME1 '<?>') ')next')+ )+

This basket is the current basket of the called process which is destroyed on return to the calling process. STATEMENT SEQUENCE leads the path back to the syntactic unit STATEMENT which completes the cycle.

The complexity of the determination of the arcs in the graph is increased when the context of a particular translator call affects the relation. This requires each 'create_cell' function in the SYNTRAN program to be associated with more than one vertex in the graph depending on the different contexts in which that 'create_cell' function appears.

5.3.1.4 Dynamic Storage Allocation.

The only instance of a create_cell function not included in the stacked storage class is the one in the translator STANDARD PROCEDURE_CALL (page 301) which corresponds to the isolated vertex in the graph above. The create_cell function here explicitly places cells created by it into a separate basket called HEAP_AREA which is not the
same as any basket name used in the SYNTRAN program nor does it
conform to the syntax of the token SCOPE_NAME which is used to
generate basket names. HEAP_AREA is a basket created in the translator
PASCAL_PROGRAM (page 305) and not referred to in any other place other
than the two just mentioned. These characteristics can only be
satisfied by the dynamic storage class.

In some cases the use of the 'create_basket' and 'create_cell'
functions and the 'destroy' statement will suggest an implementation
that is more efficient than a complete free store allocation scheme.
It is envisaged that such is the case with, for example, PL/I
CONTROLLED storage but this has not been investigated.

5.3.2 Implicit Storage Allocation.

Implicit storage allocation refers to the storage required for
intermediate results in expression evaluation. This area is influenced
greatly by the architecture of the target machine but it is still
worthwhile to discuss the evaluation of GLADYS expressions on a stack
since this is a very common method and can form the basis of
allocation schemes on register oriented target machines. As before, it
is assumed that no parallel or collateral execution takes place.
Multiple processes or collateral evaluation require as many stacks as
separate paths of execution. The feature of GLADYS expressions which
makes such a discussion non-trivial is the ability to execute
statements which allocate and release storage within part of an expression, namely, the expression unit. This occurs in the string comparison routine used as an example in sections 4.2.3.3 and 5.2.7 above. The problem that arises is that the transient SUs named $S_1$, $S_2$ and $I$ are allocated, then the value of the expression calculated and then $S_1$, $S_2$ and $I$ are released. If the value calculated is to be left on top of the same stack as is used for the allocation of the transient storage (it is assumed that stacked SUs, transient SUs and the intermediate results are all stored on the same stack as is almost universally the case) then the storage for that value must be allocated before the transient storage. The same applies if the SUs, $S_1$, $S_2$ and $I$ were stacked SUs.

A general solution is quite straightforward: when entering an expression unit in GLADYS a cell of the type specified by the expression unit is created as an SU on the stack. In fact, the SU need only be allocated when entering the outermost expression unit which supplies values for that SU and only when SUs are explicitly allocated in the expression unit. This assumes, of course, that all SUs allocated in an expression unit are also released in the same expression unit. If this were not the case, a single stack would be unsuitable.
5.4 Control Structures.

5.4.1 General.

The control structures in GLADYS allow various sequential and parallel combinations of statements that are flexible enough to model the control structures used in high-level languages. A mechanism which is a generalization of the label and 'goto' is also included because, in general, the 'goto' construct cannot be modelled using normal "structured" control structures. This 'goto' mechanism was introduced reluctantly since it allows a language definer to use the 'goto' instead of the structured control statements, thus making analysis of the control structures of L more difficult. The only alternative was to treat each statement separately and determine the next statement to be executed in each case explicitly (c.f. continuations in Denotational Semantics, see section 2.6). This causes difficulties in the translation phase, that is, in the writing of the SYNTRAN programs, because the next statement to be executed after the last statement in a nested control structure, for example, is not encountered until too late. Ease of use triumphed over elegance in this point. An example of the GLADYS 'goto' is given later in section 5.4.2.2 where 'goto' out of Pascal procedures are described in terms of GLADYS.
There are two types of sequential control structures: one which has looping and conditional primitives; and another which allows labelled statements. Two separate, but similar, constructs were chosen for sequential statements to separate, and clarify, any interaction between the structured control construct, the "loop" unit, and the 'goto'; labels being allowed only on statements in the "serial" unit. The loop unit, by itself, is powerful and flexible enough to model all the normal conditional and repetitive statements in high-level languages. For two examples see COND_STMT and LOOP_STMT in the ASPLE definition in appendix five.

The two parallel control structures are the "parallel" unit and the "clonal" unit. The first has a number of statements which are simply executed in parallel and the second has one statement which is copied a specified number of times and each copy is executed in parallel. These two constructs reflect the difference between the record and array data structures respectively. The first has a fixed number of differing elements while the second has a number of identical elements. These two forms allow the modelling of collateral execution of a number of statements. The former is useful where a number of distinct operations are to be performed in parallel, such as the evaluation of actual parameters and their assignment to formal parameters. The latter form is useful when a number of very similar operations are to be done in parallel, such as the creation of a family of identical processes or the initialization of an array-like structure cell. For example:
bind ARRAY to create_cell(ARRAY_TYPE, ARRAY_TYPE<?>) next
[ONE_TO_TEN copies
    assign INT_MODE<INTEGER<?>> to ARRAY@.index ]

creates an array called ARRAY of type ARRAY_TYPE (see section 5.2.4) and assigns 0 to all components of the array.

The expression statement is a construct that behaves syntactically like a statement but returns a value like an expression. This value is used as either the whole or part of the value of the loop, serial, parallel or clonal unit in which it appears. This allows any sort of unit to return a value; this is especially useful with the parallel and clonal units when generating structure type values. For example:

#( E → ← 0 exit ← 1 ]

where E is an expression which evaluates to 0, causing "← exit" to be skipped, or to 1. In the former case the unit will evaluate to 1 and in the latter to 0 (see section 10.1 in appendix three). On page 129 lines 16 to 18 show an expression statement returning a structure type value of type Q which initializes a variable of that type called Q_INSTANCE.
In GLADYS, as much evaluation as possible is allowed collaterally or in parallel and ordering can be imposed by using sequential units where required. For example, in Pascal procedure and function calls (translators CALL_STATEMENT (page 300) and FUNCTION_CALL (page 282)) the order of evaluation of parameters is not defined in the description and is expressed as a number of statements in a parallel unit. On the other hand, in the translator ASSIGNMENT_STATEMENT (page 299) the left-hand side (LHS) is evaluated before the right-hand side (RHS). The early proposed Pascal standard [RaveB79] implies that either the LHS is evaluated before the RHS or vice-versa, and one or the other has to be chosen for the description. In fact, the former was chosen and enforced by evaluating the LHS and storing the "address" obtained in a temporary cell and then executing the assignment using the previously evaluated "address" after evaluating the RHS. The alternative would have been easier to specify since no temporary cell would need be used.

The overall criterion for choosing the above control structures was to be able to model all normal control structures used in high-level languages in a simple way with the minimum number of GLADYS constructs.

5.4.2 Examples.
5.4.2.1 The For Statement.

The For statement is used as an example here because it is perhaps the most complicated control structure in the Pascal language. It is used to illustrate the extraction of control structure from a SYNTRAN/GLADYS description and the formation of S-language operations for it. It requires more than just the "conditional branch" type of semantics found in the While statement used as an example in section 4.3 in that the For statement also requires two stacked temporary cells. The syntax of the GLADYS output from the translator FOR_STATEMENT (page 296) is:

```
ENTER('FOR_SCOPE')
'bind FOR_B to create_cell(' TYPE_NAME1 ','
    TYPE_NAME1 '<' TO_INT(EXPRESSION1) '>') next'
'bind FOR_E to create_cell(' TYPE_NAME1 ','
    TYPE_NAME1 '<' TO_INT(EXPRESSION2) '>') next'
'[' ('LEQ' | 'GEQ') '('#FOR_B@, #FOR_E@')→'
  [' assign FOR_B@ to 'IDENT1 'next'
  [' STATEMENT 'next'
   'EQL(#' IDENT1 '@, #FOR_B@)→ null exit'
   'assign' TYPE_NAME1 '<' ('INC' | 'DEC')
   '('# IDENT1 '@) to' IDENT1 'loop ]'
'] next'
EXIT('FOR_SCOPE')
```

Thus the Pascal statement:
for I := 1 to 10 do A[I] := 0

would, making some reasonable assumptions about its context, be expanded to:

bind FOR_SCOPE to create_basket next
enclosing FOR_SCOPE is current_basket next
current FOR_SCOPE next
bind FOR_B to create_cell(T_1, T_1<1>) next
bind FOR_E to create_cell(T_1, T_1<10>) next
[LEQ(#FOR_B@, #FOR_E@) →
  [assign FOR_B@ to I next
   [bind TEMP_SCOPE to create_basket next
    enclosing TEMP_SCOPE is current_basket next
    current TEMP_SCOPE next
    bind L_H_S to create_cell(cell, A@.T_9<#I@>) next
    assign T_1<O> to L_H_S@ next
    destroy TEMP_SCOPE next
    EQL(#I@, #FOR_E@) → null exit
    assign T_1<INC(#I@)> to I loop]
  ]
] next
destroy FOR_SCOPE

where T_1 is the GLADYS name for the Pascal integer type and A is
declared to be an array of integers with base type named in GLADYS by T_9.

Perhaps the first thing to note is that the temporary cells FOR_B and FOR_E require a judicious choice for their PRs for them to be included in the stacked SAT (the PR of a cell may be chosen to be anywhere between its last use and its destruction, see section 5.3.1). The simplest choice is to consider the PR of FOR_E to be before the PR of FOR_B. This merely extends the existence of FOR_B longer than is absolutely necessary.

Another important point to note is that because of the way alternation in productions in SYNTRAN is defined the two alternation operators in the syntax of the GLADYS code above are not independent. Since the same control variable, UP in the translator FOR_STATEMENT, is used in both cases and the value of UP cannot change between these two uses the same alternative must be chosen in both cases. That is, if 'LEQ' is produced in the first case then 'INC' must be produced in the second.

Using the same technique as in section 4.3 of placing proposed S-operations between each item returned by a translator in a syntactic description. The S-operation sequence for a For statement would be:

F1 EXPRESSION1 F2 EXPRESSION2 F3 IDENT1 F4
STATEMENT F5 IDENT1 F6 IDENT1 F7 IDENT1 F8
The choices available for defining each of the S-operators, F1 to F8, are numerous and depend upon the run-time structure chosen. Since the item IDENT1 appears a number of times it would probably be advantageous to "evaluate" that item just once, if possible. Since IDENT1 is the name of a cell and assignments are made to it as well as its value referenced it can best be "evaluated" to an address which is stored as a temporary.

By removing the excess references to IDENT1 and instead refer to the same cell by the stored temporary address, the S-operator sequence can be reduced to:

\[
F1 \text{ EXPRESSION1 } F2 \text{ EXPRESSION2 } F3 \\
\text{IDENT1 } F4 \text{ STATEMENT } F5
\]

The PA of such a cell is the first occurrence of IDENT1 and the PR the last occurrence of IDENT1. The PA of IDENT1 is after the PA of FOR_E and the PR of IDENT1 is at the same time as the PR of FOR_E (they are tested for equality before the loop is terminated). Thus the address of IDENT1 can be stored in a stacked SU.

The simplest run-time structure that can be used to allow a definition of the S-operators is a single simple stack with essentially reverse polish format S-operators. This means transforming some actions defined by GLADYS code before an item to the S-operator after that item. It also allows stacked SUs to be simply allocated on
the stack as outlined in section 5.3.2. A possible description of the S-operators F1 to F5 is:

F1 Enter a new scope - could mean the marking of the stack in some way or a null operation if F5 knows how much space will be allocated for temporaries.

F2 Give value of expression just evaluated to the first temporary cell. In practice a null operation in a stack run-time structure.

F3 Give value of expression just evaluated to the temporary cell (null operation). Compare the values of the first and second temporaries and depending on whether the For statement used 'to' or 'downto', jump to the point in F5 or continue here. Give the address of IDENT1 to the third temporary cell and store the value of the first temporary in the contents of this address.

F4 No operation - acts as a marker for F5.

F5 Compare the value of the cell addressed by the third temporary with the value of the second temporary. If they are unequal increment or decrement (depending on which form of the For statement was used) the value of the cell addresses by the third temporary and continue execution at F4. Otherwise, define the point which is mentioned in F3 as "here" and release temporary storage so that the stack is in the same state as it was in when F1 was executed.
F2 and F4 are null operations. F1 can be a null operation provided F5 knows how much space to release on top of the stack. This can be done if the temporary storage required can be calculated at "compile time". This is possible in Pascal because of the strict typing, so F1 can also be assumed to be a null operation. Symbolic evaluation of the type of the first two temporary cells allocated shows that they must be interval types with upper and lower bounds determinable from the program text.

This leaves only two S-operators, F3 and F5 to implement the For statement. F3 is executed once for each execution of the For statement and F5 is executed once for each iteration of the statement inside the For statement. Both F3 and F5 depend on the form of the For statement, that is, whether 'to' or 'downto' is used. This is precisely the form used in the For-loop S-level instructions in a MODULA compiler for the B1700 [JustG81]. Thus the Pascal For statement can be implemented in a stack based run-time structure with two pairs of S-operators; one pair for each form of the For statement.

5.4.2.2 The Goto Statement.

This example illustrates how the Goto statement in Pascal (as described in appendix six) is modelled in GLADYS. This description does not allow gotos out of procedures or functions and this simplifies the semantics significantly.
The syntax of the GLADYS code produced for a Goto statement is:

'assign type_of(this_process)<' LABEL_NAME '> to this_process'

and the code produced for a labelled statement is:

'[' LABEL_NAME ': destroy_enclosed' SCOPE_NAME '] next'

UNLABELLED_STATEMENT

These code segments are taken from translators GOTO_STATEMENT (page 298) and STATEMENT (page 303).

The Goto statement itself is simple and merely changes the point at which execution continues in the process it occurs. Execution always continues at a labelled statement and the action here is more complicated. The action 'destroy_enclosed' has the effect of releasing any storage (allocated by the For statement, for example) and removing any bindings (set up by the With statement, for example) set up between the scope associated with the labelled statement and the scope associated with the Goto statement. The 'destroy_enclosed' statement then, in effect, restores the environment to the condition expected by its surrounding statements because all baskets created in a single process are enclosed by the previous 'current' basket and the newly created basket is made the current basket (see macro ENTER, page 264). This is clearly a null operation if the labelled statement is reached by normal means rather than being "jumped to".
It is important to show that the Goto statement does not invalidate the discussion above regarding, in particular, the stacked storage allocation scheme. It is often the case that constructs such as the Goto statement in Pascal cause serious complications in a language implementation and definition.

Because the restriction in the Pascal description given in appendix six of not allowing gotos out of functions or procedures the potential difficulties that can be caused by the Goto statement are reduced. The concern here is to show that the Goto statement does not change the general discussion above nor the particular examples given regarding the SUs that have been determined to be of the stacked SAT. The only construct that can contain a statement and which also allocates and releases (stacked type) storage is the For statement. It must be established that the relations (arcs) used in the directed graph in section 5.3.1.3 are not affected by a Goto statement within the For statement.

There is clearly no effect on the allocated SUs if a Goto statement within a For statement "jumps" to another statement in the For statement. If, however, a Goto statement "jumps" out of the For statement, the cells created in the basket FOR_SCOPE must be released. A GLADYS "goto" statement always "jumps" to a 'destroy_enclosed' statement in the Pascal definition and this will restore the environment by destroying any baskets enclosed by the basket which has been statically determined to be the current basket when that
statement is executed. This will destroy the basket FOR_SCOPE because of the strict enclosure of baskets which is maintained by the use of the macros ENTER (page 264) and EXIT (page 264). The only times a newly created basket is not made the current basket and not enclosed by the previous current basket is in initialization in the translator PASCAL_PROGRAM (page 305) and in the translators FUNCTION_CALL (page 282) and CALL_STATEMENT (page 300). A "goto" cannot by-pass these situations because it cannot "jump" out of the program and cannot "jump" out of a procedure or function. If gotos out of functions and procedures were allowed then a more complicated method of ensuring the preservation of the relation used in the stacked SAT graph would be required.

To preserve the required relation between vertices of the graph it is also necessary to show that a "goto" cannot be executed which "jumps" into a For statement. If this were allowed the 'destroy' statement at the end of the For statement could be executed without the corresponding 'create_basket' function being executed. This would result in either an error when the destruction of a non-existent basket was attempted or the basket of an enclosing For statement could be prematurely destroyed. The inability to "jump" into a For statement (or any statement) is a syntactic restriction enforced by the SYTRAN program alone. The mechanism achieving this is explained in section 1.2.1 of appendix six and conforms to the proposed standard in [RaveB79]. Showing how this mechanism excludes, in particular, "jumps" into For statements is a complex argument and is only outlined here.
The syntax of statements in Pascal is recursive and this is reflected in the recursive nature of the translator calls in the SYNTRAN description. Where a translator involves the parsing of a statement involving labels or calls of other translators involving statements a node is maintained which represents the set of labels that have been used in Goto statements but have not yet been defined (prefixed to an unlabelled statement). This node is passed back from each translator call by a result parameter (corresponding to a "synthesized attribute" in Knuth's Attribute Grammars [KnutD68]). The translator BLOCK (page 304) tests to make sure that this set is empty after the statements in the block it represents have been parsed. Thus assuming that labels are unique within a block (which is enforced by other mechanisms in the SYNTRAN program) if a label is defined in a For statement, any Goto statement which uses that label must be parsed by a translator called, directly or indirectly, from the translator parsing the For statement. If this is not the case the label will be added to the set after its chance of being removed by the parsing of its definition has occurred. The label will then remain a member of the set to cause failure in the translator BLOCK.
5.5 Processes.

5.5.1 General.

Processes are used to model the 'active' or purely procedural part of high-level language 'routine' constructs such as procedures, functions, SIMULA [Dahl066] classes, MODULA [WirtN77] processes and Ada [IchbJ79] tasks. GLADYS processes have no parameters and it is intended that a process, being a type, and an instance of a process, a value of a process type, will be part of a structure type in which its parameters are kept. This unifies the treatment of processes and data storage under the same type, basket and cell mechanisms. The process is able to access these cells in the structure type in which it is a part by the type reference construct.

The various methods used for passing parameters can be achieved in a similar way to the methods used in ALGOL 68. That is: a value, a reference to a cell or a process with the correct environment may be passed, to model parameter calls by value, by reference or by name, respectively.

The concept of a process as a pure procedure with no parameters, no result nor local data was chosen because of its simplicity and primitiveness. A less primitive basis might lose enough flexibility to become a liability.
5.5.2 Operations.

The four primitive operations on processes are: "start", "stop", "wait" and "transfer". "Start" is used to start processes and "stop" is used to stop processes including the process executing the "stop" primitive. The "wait" operation is the synchronization primitive in GLADYS (see section 2.10). The process executing a "wait" operation, waits until one of the processes specified has stopped when it starts again immediately. If more than one process is waiting for another to stop, only one of the waiting processes (which one, is left unspecified) continues with the process being waited on. Similarly, if more than one process that is being waited on, stops simultaneously, only one (again unspecified) is started when the waiting process continues. "Transfer" is the same as "start" except that the process executing the transfer is stopped immediately.

These primitives can be combined so that they are executed simultaneously and in this way the process control involved in procedure calling and returning can be modelled. A procedure is modelled as a separate process; when the procedure is called the calling process executes a transfer statement. The procedure returns in a similar manner. These operations on processes in GLADYS are very flexible and can model other more complex process control operations of other languages such as "resume" and "detach" in SIMULA.
5.5.3 An Example.

This example deals with specific instances of a Pascal program segment and shows how procedure declarations, calls and returns are handled. The equivalent GLADYS is that as specified by the translation of Pascal to GLADYS in appendix six but with the added ability to allow 'goto's out of procedures. The Pascal program segment:

i procedure P;
ii label 4;
iii procedure Q(I : integer);
iv begin
v goto 4
vi end;

vii begin
viii 4: Q(5)
ix end

is translated to the GLADYS program segment:
1  bind P_BASE to interval(-1, 0) next
2  bind P_PROCESS to process[
3  bind Q_BASE to interval(-1, 1) next
4  bind Q_PROCESS to process[
5      [bind I to Q_INSTANCE@.Q_BASE<1>] next
6      assign P_PROCESS<1> to P_INSTANCE@.P_BASE<0> next
7      transfer P_INSTANCE@.P_BASE<0> next
8      transfer Q_INSTANCE@.Q_BASE<-1>@
9    ] next
10  bind Q to structure(Q_BASE; cell, Q_PROCESS, T_1) next
11  [1: destroy_enclosed S_1] next
12  bind S_2 to create_basket next
13  enclosing S_2 is S_1 next
14  bind Q_INSTANCE to create_cell(
15    Q,
16    Q[ Q_BASE<-1> ← this_process with
17    Q_BASE<0> ← Q_PROCESS<0>/S_2 with
18    Q_BASE<1> ← T_1<5>],
19    S_2
20    ) in S_2 next
21    transfer (Q_INSTANCE in S_2)@.Q_BASE<0> next
22    destroy S_2
23  ] next
24  bind P to structure(P_BASE; cell, P_PROCESS)
The lines of the two program segments are numbered, the Pascal segment with Roman numerals, for ease of reference.

The declaration of procedure P consists of the creation of three types named P_BASE, P_PROCESS and just P (lines 1, 2 to 23 and 24). P is a structure type with base type P_BASE and constructing types cell and P_PROCESS. The cell type cell is used to refer to the calling process and the P_PROCESS type cell is the process corresponding to the procedure P (line i). The declaration of procedure Q is on lines 3, 4 to 9 and 10. The structure type Q is similar to P but has another cell of type T_1 which is the parameter called by value in the procedure Q (line iii). T_1 is used as the name of the GLADYS type corresponding to the Pascal integer type.

The call of procedure Q from procedure P is on lines 12 to 22. The current scope in which procedure P is declared is named S_1. The call of Q starts by creating a new basket, S_2, (line 12) enclosed by S_1 (line 13). A new cell of the structure type Q, called Q_INSTANCE is created (lines 14 to 20) in the new basket, S_2 (line 19). The binding of the name Q_INSTANCE to the new cell is also placed in S_2 (line 20). The value of this new cell (lines 16 to 18) consists of the values: a reference to the currently executing process (line 16), the Q_PROCESS initial value with current basket S_2 (line 17) and the integer value 5 (line 18). The call of Q is completed by simultaneously stopping the currently running process executing procedure P and starting the process in the structure type cell called
Q_INSTANCE in basket S_2 by use of the transfer statement (line 21). When the process executing P is started again (at the point of call of Q) the basket S_2 is destroyed (line 22). The names Q_BASE, Q_PROCESS, Q and Q_INSTANCE and the objects to which they were bound to are also destroyed.

Execution of the process in Q_INSTANCE starts with binding the identifier I to the cell containing the integer value 5 in basket S_2 (line 5) thus modelling a call-by-value parameter. Line 8 shows the normal return code. The process in Q_INSTANCE is stopped and the calling process started by the transfer statement.

The Goto statement (line v) corresponds to lines 6 and 7. Line 7 has the same effect as line 8 in that it returns control to the calling process, but before it does so the calling process (of Q_INSTANCE) has its point of execution changed by the Assignment statement (line 6). The P_PROCESS type value corresponding to the integer one (label "4", line viii) also corresponds to the point just after the colon on line 11. The first statement executed after the abnormal "return" is the destroy enclosed statement which will destroy basket S_2 (it being enclosed by S_1, the current basket for the process executing P) as required for correct semantics.

The extra feature in this GLADYS program segment which is not as described in appendix six is the Assignment statement on line 6. In effect the process containing the target label is identified as P.
This enables a return to be made in the most recently called instance of P_INSTANCE, the baskets being dynamically nested. This dynamic nesting also supports the propagation of ADA exceptions [IchbJ79], for example, which is a special case of the "goto out of procedure" semantics.

5.5.4 Synchronization.

The "wait" synchronization mechanism can model the binary semaphore [DijkE68] very simply. A process consisting of an empty loop, the "wait" operation and the "stop" operation correspond to a binary semaphore, the P operation and the V operation, respectively. The running process corresponds to the semaphore with the value zero and the stopped process to the semaphore with the value one.

The above primitive operations on processes were chosen, again, for their simplicity and flexibility. The synchronization primitive, in particular, was chosen because it introduced no separate concepts such as semaphores or eventcounts [ReedD79] (see section 2.10).
5.6 Original Features.

The following features in GLADYS are claimed to be original ideas of the author:

1. baskets for control of storage and bindings and as a primitive for modelling high-level language scoping rules;
2. the generalized control structure, the loop unit;
3. the clonal statement and expression units; and
4. the synchronization primitive.
Chapter 6 - Conclusion

6 CONCLUSION.

The aim of this thesis is to develop a method of defining a high-level language that is appropriate for use in SAGA.

6.1 What Has Been Achieved.

A method for defining L has been developed which consists of a language, SYNTRAN, which is used to specify the translation between L and the known target language, GLADYS. In a sense SYNTRAN is used to define the syntax of L while GLADYS is used to define the semantics of L. The definition of L by translation facilitates the generation of the compiler (see section 3.2.3.2). The power of GLADYS has been demonstrated by the examples in chapter five.

Some parts of the definition of L can be used to generate S-operators quite readily (for example, control structures) while others (for example, storage allocation) require externally supplied heuristics. Despite the shortcomings listed below, the SYNTRAN/GLADYS method of high-level language definition shows promise as illustrated by the examples given in chapters four and five.
6.2 Shortcomings.

Perhaps the most obvious criticism that can be laid against the SYNTRAN/GLADYS definition of Pascal in appendix six is its length. The SYNTRAN program itself takes up about 40 pages, and it does not even describe Pascal completely. The draft Pascal standard [RaveB79] is only 14 pages long and covers the complete language. The SYNTRAN/GLADYS description is more precise but also requires detailed knowledge of both SYNTRAN and GLADYS. Other formal definitions of Pascal are likely to be quite long as well (e.g. the new draft proposed standard for Pascal [AddyA80] is 55 pages long without appendices) and their relative merits with respect to SYNTRAN and GLADYS are somewhat subjective.

Perhaps the main points to emerge are that there is still quite a lot of work in translating Pascal to GLADYS and that the "higher" level of GLADYS does not save a great deal compared to a low-level target language where no attempt is made at optimization, efficient register allocation etc. and there is no requirement for semantic analysis. Semantic analysis is required, though, of the SYNTRAN/GLADYS definition and its length is justified when this is borne in mind.
6.2.1 SYNTRAN.

From the point of view of using SYNTRAN to describe a compiler, some difficulties can be noted. The first is that the emissions from the various translators are not produced in the same order that would be required to write an object code file in a sequential manner. Emissions of translators are included in the emission of others and so on until the emission of the distinguished translator is decided. It may be possible to overcome this by executing the patterns and productions as co-routines, producing as much output as possible before matching further input. The second difficulty is that the introduction of any global optimization would require such optimization to be explicitly specified in SYNTRAN somehow. These points have not been investigated.

With regard to the requirement that symbolic evaluation be facilitated in SYNTRAN the lack of stronger typing may be a disadvantage. A form of typing is enforced by emission and control declarations and the "use modes" given to variables. This could be extended to a larger class, or perhaps all, variables by requiring the syntax of all variables which are to have string values to be specified when they are declared and requiring declarations of node selectors etc. It is not known whether these sorts of steps will lead to a better translating language or not.
With respect to the last point SYNTRAN shares a problem with most high-level languages which try to encourage good programming practices such as the use of subrange types where possible in Pascal. This is that these languages can do just that, encourage; they cannot force a programmer to declare a variable which can only take the values between 5 and 10, say, in a correctly working program as a variable of type subrange 5 to 10 rather than as a variable of type "integer". This problem manifests itself in SYNTRAN as the inability, for example, to force the programmer to specify in a pattern or a production the '+' operator rather than the '*' operator where the former is sufficient, or, to force the strictest possible token definitions on variables that are emitted in productions. An extreme case of the latter is to use a token defined to include any string of characters. The emission of a variable declared in an emission statement using this token provides no information whatever. Another example appears in section 5.3.1.1.

In order to make the determination of the context free syntax of the string values emitted by a production as simple as possible a rather complicated condition has to be applied to occurrences of variables declared in patterns and used in productions (see section 15 in [EdwaS80c]). Although this can be checked by a SYNTRAN compiler the reason for and the correction of a violation of this rule may be obscure in some cases. There is an analogy here between choosing methods of implementation at the compiler/interpreter generation time being discussed here and choosing methods of implementation at compile
Chapter 6 - Conclusion

Time (program generation time). The latter is referred to as "optimization" and it is felt that optimization techniques can be applied to the former.

Finally, an indication that the translators and patterns of SYNTRAN are not quite powerful enough for some high-level language constructs. In the translator SELECTOR (page 283, appendix six) a global flag, INDEX_FLAG, is required to correctly parse the indexing selectors of a Pascal array. This difficulty arises because of the short form of multiple indexing where \([i_1, i_2, i_3]\) is allowed as an abbreviation for \([i_1][i_2][i_3]\).

6.2.2 GLADYS.

The analysis of GLADYS constructs is made more difficult in some situations because there is more than one way of specifying the same semantics. For example, if the current basket of a process is known at "translation-time" the current basket could be specified either by being named explicitly or by using the reserved identifier 'current_basket'. Any analysis involving current baskets has to take into account both ways of specifying the current basket, making the analysis more complicated.
There are a number of ways in which a collection of expressions may be evaluated. The simplest is to evaluate them serially with their order specified; this is the most common and is handled in GLADYS by the 'next' statement separator. There at least two ways of evaluating a collection of expressions "collaterally". They may all be evaluated simultaneously by multiple processes or interleaved in an arbitrary fashion on a single processor with one set of implications or they may be evaluated serially but in any order with a different set of implications. The parallel evaluation in GLADYS using the 'with' separator corresponds to the former situation and there is no way of specifying the latter.

A more general difficulty in GLADYS is the untidiness of the changes in bindings that occur when baskets are destroyed. The idea behind bindings (the bind statement) was that a binding, once set up, was immutable and only disappeared when the basket in which it was placed was destroyed. Unfortunately, if a binding associates an identifier with an object in another basket which is destroyed leaving a "dangling" pointer or association then that binding changes so that the identifier is bound to the null value of the object type to which it referred. Thus a binding is not immutable as was desired.

The use of external functions was motivated by the desire to remove "irrelevant" arithmetic from GLADYS itself and rely on a mathematical definition for these functions. This leads immediately to the problem that in some situations it is necessary to know the
semantics of some external functions to make a decision in the analysis of a SYNTRAN/GLADYS definition. Such is the case in dealing with the string comparison operators in section 5.2.7. It also leads to a less serious problem in that only a specified number of single integers can be operated on in a single external function. In some situations it would be advantageous to directly convert a structure type value to a vector of integers and define an external function on the vector. This would, for example, allow the implementation of sets in Pascal to be a structure type yet still allow the set operations be externally defined.

The synchronization primitive in GLADYS uses a process as a semaphore. This obviates the need for a separate data type called semaphore which has special properties. It was envisaged that the ability to have a non-trivial process executing and behaving like a semaphore would enable it to carry out other useful processing at the same time. This has not been investigated deeply due to lack of time (but see appendix seven) and if, in fact, no advantage accrues from the use of a process as a semaphore, analysis of a language definition would be easier if the two concepts were separated.
Chapter 6 - Conclusion

6.3 Further Work.

Apart from the need to improve on the work done so far to eliminate, if possible, the shortcomings presented in the last section, the areas of SAGA not investigated in this dissertation provide the topics for further work.

As mentioned before, by defining L by translation, specification of the compiler is made far easier than otherwise (see section 3.2.3.2). The replacement of the target GLADYS constructs in the translators by S-language constructs has been shown to be practical in chapters four and five. To create a practical compiler though, some measure of error recovery in the parser [MiltD79, GrahS79] and global optimization must be to be added to the basic translator definition (see section 2.5).

A particular method for defining M has to be found. It is envisaged that a suitable notation can be devised as an extension of the work currently being carried out in the field of code generator generators (see section 2.8).

There are three areas associated with the production of the interpreter to be investigated (see section 3.2.3.3). Firstly, there is the creation of the target machine code segments to perform the actions specified by each S-operation. This should be able to be achieved by techniques used in code generator generators and
complements the use of the method of description of M from the same area of work. Secondly, a method for deciding on S-instruction formats in a manner suitable to the target machine has to be found. This requires a number of specific choices to be made which can be guided by heuristics involving the target machine addressing modes and word size. Lastly, a method for making the machine dependent implementation decisions required has to be found. This is perhaps the most challenging area for research as it promises a computer architecture implementation "expert" system (see [BarrHBO] and references therein). This would be a system embodying the knowledge and experience of compiler-writers that is used in making the decisions that make an implementation of a language successful and efficient. There is an analogy here between choosing methods of implementation at the compiler or interpreter generation time being discussed here and choosing methods of implementation at compilation time (i.e. program generation time). The latter is referred to as "optimization" and similar optimization techniques can be applied to the former.

Finally, the formulation of complex rules and heuristics in trying to determine the structure of a language from its definition could lead to an "expert" system in this area as well. Thus, from a description of a language, the ideal (virtual) machine architecture for that language would be able to be determined. This would highlight the correspondence between programming languages and their underlying computational mechanisms and hence be a useful tool in future programming language design and implementation.
REFERENCES


References


References


CommPET Commodore Business Machines, "PET" (R), 901, California Avenue, Palo Alto, California, 94304, U.S.A.

CromI79 Cromemco Incorporated, "System Three Computer" (R), 280 Bernardo Avenue, Mountain View, California, 94040, U.S.A., 1979.


DijkE72 Dijkstra, E.W., "Notes on Structured Programming", in [Dahl072], pp1-82.

References


Edwards, S.J., "Notation for the Definition of Context Free Syntax", Department of Computer Science, Department Report TR-CS-80-05, Australian National University, Canberra, 1980. (appendix one)


References


HewiC72 Hewitt, C., "Description and Theoretical Analysis (using Schemata) of PLANNER", M.I.T., A.I. Laboratory, TR-258, Cambridge, Massachusetts, April 1972.
References


Hoar72 Hoare, C.A.R., "Notes on Data Structuring", in [Dahl72], pp83-174.


HursA80 Hurst, A.J., private communication, March 1980.


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
</table>


References


References


References


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title and Authors</th>
</tr>
</thead>
</table>
APPENDICES

Note that appendices one to four are reproduced from other reports and have their own sets of references.
A description of a notation for defining context free syntax is given to act as a reference point amongst the many "extended EBNF"s in use. The notation is also described in terms of itself.
1 INTRODUCTION.

Since BNF (Backus-Naur Form) [Back59] was used in defining the syntax of ALGOL 60 [Naur63] many extensions and dialects of it have appeared under the name "extended BNF". See for example [Grie71, Jens78, Naur63]. These extensions have been devised to reduce the reliance of BNF on recursive definitions and also to reduce the sheer number of productions required; both of which detract from the readability of BNF. This report merely describes one such "extended BNF" used by the author to act as a reference and standard for other works. The ASCII character set is sufficient for use with the notation. No originality is claimed for the ideas presented herein.

2 THE META-SYMBOLS.

The following symbols are used as meta-symbols: the equals sign "="; the period "."; the vertical bar "|"; the slash "/"; the single quote "'"; the asterisk "*"; the plus sign "+"; the question mark "?"; and the left and right parentheses "(" and ")". The use for each is described below:

the equals sign. This is used to separate the non-terminal symbol being defined, on the left, and its definition, on the right.

the period. This is used to terminate a definition.

the vertical bar. This is used to indicate alternation in the same way as in the original BNF.

the slash. This is used to indicate the listing of one or more occurrences of the left argument separated by single occurrences of the right argument.

the single quote. This is used to delimit terminal symbols. If the single quote itself appears in a terminal symbol it is represented by two single quotes as is normal in representing characters strings.

the asterisk. This is used to indicate that its left argument may occur not at all, once, twice or any number of times.

the plus sign. This is used to indicate that its left argument may occur any number of times, but at least once.

the question mark. This is used to indicate that its left argument may occur once or not at all.

the parentheses. These are used to group syntactic constructs for the purpose of using them as one argument to a meta-symbol.
Appendix 1 - Notation for the Definition of CFS

3 NON-TERMINALS AND LAYOUT OF DEFINITIONS.

Non-terminals may consist of any number of characters except blanks or meta-symbols. To enhance readability, meaningful names should be chosen. Juxtaposition is used to indicate concatenation of syntactic entities but at least one blank must appear between two adjacent non-terminals. Blanks apart from this have no meaning except in terminal symbols and should be used to enhance readability. Definitions may be spread over more than one line since the period is used as a terminator. It is not recommended that more than one definition be placed on the same line, but this is not excluded by the notation.

4 META-SYMBOL OPERATIONS.

The three unary operations are defined by using BNF.

\[ A = B^+. \text{ is equivalent to } \langle A \rangle ::= \langle B \rangle \langle T \rangle \]
\[ \langle T \rangle ::= \langle A \rangle \mid \langle \text{empty} \rangle \]
\[ \langle \text{empty} \rangle ::= \]

\[ A = B?. \text{ is equivalent to } \langle A \rangle ::= \langle B \rangle \mid \langle \text{empty} \rangle \]

\[ A = B*. \text{ is equivalent to } \langle A \rangle ::= \langle T \rangle \mid \langle \text{empty} \rangle \]
\[ \langle T \rangle ::= \langle B \rangle \langle A \rangle \cdot \cdot \]

The unary operators cannot appear together - the result of one cannot be used directly as the argument of another.

The three binary operations, in order of highest to lowest precedence, are listing, concatenation and alternation. Unary operations have higher precedence than the binary operations. The binary operators are also described using BNF:

\[ A = B / C. \text{ is equivalent to } \langle A \rangle ::= \langle B \rangle \langle T1 \rangle \]
\[ \langle T1 \rangle ::= \langle T2 \rangle \mid \langle \text{empty} \rangle \]
\[ \langle T2 \rangle ::= \langle C \rangle \langle A \rangle \]

\[ A = B \cdot C. \text{ is equivalent to } \langle A \rangle ::= \langle B \rangle \langle C \rangle \]

\[ A = B \mid C. \text{ is equivalent to } \langle A \rangle ::= \langle B \rangle \mid \langle C \rangle \]

Parentheses are used for grouping as in expressions in most programming languages.

5 SELF-DESCRIPTION.

The syntax of the notation is described below in the notation itself.
Appendix 1 - Notation for the Definition of CFS

SET-OF-RULES = RULE*.
RULE = NON-TERMINAL '=' EXPRESSION? '.'.
EXPRESSION = EXPR1 / '|'.
EXPR1 = EXPR2+.
EXPR2 = EXPR3 / '/'.
EXPR3 = ELEMENT ('+' | '*' | '?')?
ELEMENT = NON-TERMINAL | TERMINAL | ('' EXPRESSION '').
TERMINAL = (''' NOTSQ'''').
NON-TERMINAL = NOTMS+.

NOTSQ denotes any character except a single quote.

NOTMS denotes any character except those used as meta-symbols and blank.

This definition does not, as most do not, take into account the meaning of and requirements for the blank character which were explained in section 3. The following description specifies where blanks may or must occur and is hence more precise than the one above.

SET-OF-RULES = RULE*.
RULE = NON-TERMINAL SPACES '=' SPACES (EXPRESSION SPACES)? '.' SPACES.
EXPRESSION = EXPR1 / (SPACES '|' SPACES).
EXPR1 = EXPR2 / (SPACES + SPACES).
EXPR2 = EXPR3 / (SPACES '/' SPACES).
EXPR3 = ELEMENT (SPACES ('+' | '*' | '?'))?
ELEMENT = NON-TERMINAL | TERMINAL | ('' SPACES EXPRESSION SPACES '').
TERMINAL = (''' NOTSQ''').
NON-TERMINAL = NOTMS+.
SPACES = '*'.

Where NOTSQ and NOTMS have the same meanings as above.
Appendix 1 - Notation for the Definition of CFS

6 REFERENCES.


SYNTRA is a program designed to study the translation of high-level languages into another and thereby facilitate the semantic analysis of the target in terms of the syntax. The initial purpose of the program was to allow the study of the nature of the translation process and its application in understanding the semantics of high-level languages. This involved the analysis of the semantic output of the target language.
SYNTRAN is a language designed to specify the translation of one high-level language to another and thereby define the semantics of the former in terms of the latter. The method of specifying the production of the target language output is designed to allow the syntax of the output to be determined by inspection of the SYNTRAN program. This facilitates the analysis of the possible output strings.
1 INTRODUCTION.

SYNTRAN is a language designed to specify the translation between two programming languages, from an input stream to an output stream. It is designed to be used with the language GLADYS [Edwa80b] as the target of translation and thus define the syntax and semantics of the source language in terms of SYNTRAN and GLADYS.

The input stream is matched and consumed using "patterns" and the output stream is produced using "productions". Patterns and productions have syntactic similarities and allow the context free syntax of the input and output streams to be determined by inspection of the SYNTRAN program. It is an aim of SYNTRAN that the context sensitive syntax of the output stream be able to be determined by symbolic evaluation of the SYNTRAN program.

2 VOCABULARY.

The syntax of SYNTRAN is defined in this report by using an extended BNF described in [Edwa80a]. The basic vocabulary of SYNTRAN consists of the upper case letters, digits and some special symbols and keywords which are described here.

LETTER = 'A' | 'B' | ... | 'Z'.

DIGIT = '0' | '1' | ... | '9'.

SPECIAL SYMBOL = '+' | '-' | '*' | '/' | '#' | '.' | ':' | '=' | '<' | '>' | '|' | '.' | '<=>' | '<-=' | '<-' | '</-' | '[' | ']' | '{' | '}' | '('. | ')'.

RESERVED WORD = 'assign' | 'alias' | 'boolean' | 'call' | 'case' | 'control' | 'distinguished' | 'do' | 'downto' | 'else' | 'emission' | 'endcase' | 'endfor' | 'endfunction' | 'endif' | 'endignore' | 'endloop' | 'endmacro' | 'endpattern' | 'endproduction' | 'endtranslator' | 'eof' | 'eol' | 'exit' | 'fail' | 'for' | 'from' | 'function' | 'global' | 'if' | 'ignore' | 'integer' | 'is' | 'local' | 'loop' | 'macro' | 'new' | 'not' | 'null' | 'of' | 'otherwise' | 'pattern' | 'production' | 'return' | 'set' | 'substr' | 'then' | 'to' | 'token' | 'translator' | 'use'.

Comments in SYNTRAN are any sequence of characters not including a right brace, enclosed in braces. That is:

COMMENT = '{' NOT_RIGHT_BRACE* '}'.

where NOT_RIGHT_BRACE represents any character except a right brace. Comments may be placed just before or after any special symbol or
reserved word without affecting the meaning of the SYNTRAN program.

3 IDENTIFIERS.

Identifiers are used to denote variables, functions, sets, tokens, translators, macros, controls and emissions. Identifiers are associated with an object by declarations, and that association must be unique in the scope of that declaration. Re-declaration of an identifier is not allowed in the scope of that identifier, that is, scopes are not statically nested but augmented.

Identifiers declared as global variables, functions, sets, tokens, translators or macros must be declared outside any other function, translator and macro declaration and have global scope. This means that they are known throughout the SYNTRAN program and cannot be re-declared anywhere. Identifiers declared as local variables have as their scope the function, translator or macro in which they were declared. Identifiers declared as emissions or controls may have either global or local scope depending on whether their declaration occurs outside or inside, respectively, a translator or macro declaration.

 IDENTIFIER = LETTER (LETTER | DIGIT | '_')*.

4 DATA TYPES.

There are three types of data in SYNTRAN. They are the Boolean, string and node types.

4.1 The Boolean Type.

The Boolean type is the simplest type and has just two values, true and false. Values of this type are used in the if statement and the control statement. They have no literal denotation and are only produced as the result of a Boolean expression. The normal Boolean operations of 'and', 'or', 'not' and 'exclusive or' are available and the operator symbols are 'a', 'i', '~' and '!' respectively.

4.2 The String Type.

The string type has as values sequences of characters of length zero or more. The string of length zero has the literal denotation 'null'.

 LITERAL = NUMBER | '""" NOTSQ* '""" | '"" NOTDQ* '"" | 'null'.


where: NOTSQ represents any character except the single quote and NOTDQ represents any character except the double quote.

\[
\text{NUMBER} = (\ ' + ' | ' - ' )? \text{DIGIT} + .
\]

Strings which conform to the syntax of the literal denotation NUMBER may be used as integers and form a sub-type of the string type. When such a string is used as a number leading zeros and a possible leading plus sign are removed. The normal arithmetic operations of addition, subtraction, multiplication, truncated division and exponentiation are available for integer strings and the operator symbols are '+', '-', '*', '/’ and '%', respectively. The unary operations of negation and identity with the operator symbols '-' and '+' are also available. The identity operation has the effect of removing leading zeros and the possible leading plus sign from integer strings as mentioned above.

Operations allowed on all strings are concatenation and determination of length using the operator symbols ' ' and '#'. The latter is a unary prefix operator. A built-in function 'substr' takes three arguments, in turn, a string, and two integer strings. It returns the substring of its first argument starting at the character position (starting at one) specified by the second argument with length specified by the third. If the arguments do not specify a substring an error occurs.

String values may also be compared for equality and related by lexicographic ordering. The equality and inequality operator symbols are '=' and '!=' and the relational operators '<', '<=', '>', and '>=', have their usual meanings. A comparison operation returns a Boolean value also as usual. When two integer string values are compared it is their interpretation as integers, not strings, that is compared.

4.3 The Node Type.

The node type is the most complicated type and is used to form structured data type values. A node value consists of zero or more variables called branches, each with a string value selector. Nodes may be used to build general directed graph data structures but, in practice, directed acyclic graphs are the most complicated structures required. A node with no branches is indistinguishable from the null string.

Branches of a node are selected using the selection operator which has as a left argument, a node value and as a right argument, a string value. The selection operator symbol is the period '. '. If a branch is selected that does not exist, it is automatically created and given the initial value null. Thus a non-existent branch and a branch with the null value are indistinguishable.

The operators applicable to nodes are the "selector determination" operator and the "number of branches determination"
operator which have unary prefix operator symbols '!' and '#', respectively. The former returns a node with branch selectors 1, 2, ... etc. up to the number of branches of its argument and with branch values the selectors of its argument. The order of the branch values with respect to the branch selectors is not defined. The latter operation returns an integer string value which is the number of branches its argument node value has.

5 VARIABLES.

Variables are primitive objects which have a value of type Boolean, string or node. A variable may be denoted by an identifier or by selecting a branch of a node.

\[ \text{VARIABLE} = \text{IDENTIFIER ('.' STRING_EXPRESSION)*} \]

Each variable has a use mode which is one of the use modes: R, W, E, C or V. The use mode of a variable is inherited from the identifier used in the denotation of that variable. The properties of variables of each use mode are described as follows:

Use mode: V

Declared: in global declarations and in local declarations in translators, macros and functions with use mode V specified.

Used: in expressions and as variable in assignment statements.

Use mode: R

Declared: all formal parameters of functions and formal parameters occurring before the colon in translators and macros, the identifier declared in the for statement, the optional index variable of integer controls in productions and in local declarations with use mode R specified.

Used: in expressions only.

Use mode: W

Declared: formal parameters occurring after colon in translators and in local declarations with use mode W specified.

Used: as variables in assignment statements and in expressions in alias statements.
Appendix 2 - SYNTRAN

Use mode: E

Declared: in emission statements, as formal parameters occurring after colon in macros and variables declared in patterns.

Used: in the emission part in productions and in expressions.

Use mode: C

Declared: in control declarations.

Used: in control parts of productions and in expressions.

6 FUNCTIONS.

Functions consist of a statement list, may return a value and may have local variables. A function is invoked by using the identifier declared with that function. Parameters may be passed to a function when it is called and these actual parameters are expressions which are evaluated when the function is called, that is, parameters are "called by value". All functions are declared globally.

7 SETS.

Sets in SYNTRAN are sets of characters. They are used in matching the input stream and a set matches any single character that is a member of that set. All sets are declared globally.

8 TOKENS.

Tokens are syntactic units also used in matching the input stream. They may be built out of string literals, sets and other previously declared tokens. (Recursive definition of tokens is not allowed.) Tokens are also used to generate strings of known syntactic form to form part of the output stream. The primitive SYNTRAN function 'new' takes a token as its argument and returns a string which conforms to the syntax specified by that token and which is different from all other strings generated by the 'new' function with the same token argument. Tokens are used to check the syntax of any string used to form part of the output stream in the emission declaration. All tokens are declared globally.

9 TRANSLATORS.

A translator is a SYNTRAN construct which matches part of the input stream and, in general, emits (or returns) a string to form part of the output stream. The input stream is matched (consumed) by
constructs called patterns and the output stream is produced by constructs called productions. Parameters may be passed to and from translators which are either called from within other translators or implicitly invoked. Exactly one translator must be "distinguished", that is, have the keyword 'distinguished' in its declaration. This translator is invoked after all statements and declarations outside other declarations have been executed. All translators are declared globally.

10 MACROS.

Macros consist of a production and one may be invoked from within productions in other macros or in translators. Parameters may be passed to a macro when it is called. Macros, like functions, are merely methods of abbreviation, allowing parts of productions, or statements, used often to be written once.

11 EXPRESSIONS.

Expressions are constructs which use operands: variables, literals and functions; and operators to produce a new Boolean, string or node value.

\[
\text{EXPRESSION} = \text{BOOLEAN EXPRESSION} | \text{STRING EXPRESSION} | \text{NODE EXPRESSION}.
\]

\[
\text{INTEGER EXPRESSION} = \text{STRING EXPRESSION}.
\]

where integer expression evaluates to a integer type string value. The precedence of operators is as implied by the syntax of expressions given below.

11.1 Node Expressions.

\[
\text{NODE EXPRESSION} = '!' \text{ NODE EXPRESSION} | \text{NODE VARIABLE} | \text{NODE FUNCTION CALL}.
\]

\[
\text{NODE VARIABLE} = \text{VARIABLE}.
\]

\[
\text{NODE FUNCTION CALL} = \text{FUNCTION CALL}.
\]

where node variable and node function call evaluate to a node type value.
Appendix 2 - SYNTRAN

11.2 String expressions.

\[
\text{STRING EXPRESSION} = \text{STRING FACTOR} / 'A'.
\]

\[
\text{STRING FACTOR} = \text{STRING TERM} / \text{ADD_OP}.
\]

\[
\text{STRING TERM} = \text{STRING PRIMARY} / \text{MUL_OP}.
\]

\[
\text{STRING PRIMARY} = \text{STRING UNIT} / '%'.
\]

\[
\text{STRING_UNIT} = '#' ? \text{STRING_ELEMENT} | '#' \text{ NODE_EXPRESSION}.
\]

\[
\text{STRING_ELEMENT} = \text{LITERAL} | \text{STRING_VARIABLE} | \text{STRING_FUNCTION_CALL} | '\(' \text{ STRING_EXPRESSION } '\')'.
\]

\[
\text{STRING_VARIABLE} = \text{VARIABLE}.
\]

\[
\text{STRING_FUNCTION_CALL} = \text{FUNCTION_CALL}.
\]

\[
\text{ADD_OP} = '+' | '-'.
\]

\[
\text{MUL_OP} = '*' | '/'.
\]

where string variable and string function call evaluate to a string type value.

11.3 Boolean Expressions.

\[
\text{BOOLEAN EXPRESSION} = \text{BOOLEAN FACTOR} / (' | | '!').
\]

\[
\text{BOOLEAN FACTOR} = \text{BOOLEAN PRIMARY} / '&'.
\]

\[
\text{BOOLEAN PRIMARY} = \text{NODE_EXPRESSION} \text{ EQ_OP} \text{ NODE_EXPRESSION} | \text{STRING EXPRESSION} \text{ REL_OP} \text{ STRING_EXPRESSION} | '\~'? (\text{BOOLEAN_VARIABLE} | \text{BOOLEAN_FUNCTION_CALL} | '\(' \text{ BOOLEAN_EXPRESSION } '\')).
\]

\[
\text{BOOLEAN_VARIABLE} = \text{VARIABLE}.
\]

\[
\text{BOOLEAN_FUNCTION_CALL} = \text{FUNCTION_CALL}.
\]

\[
\text{REL_OP} = '<' | '> | '<=' | '>=' | \text{EQ_OP}.
\]

\[
\text{EQ_OP} = '=' | '=='.
\]

where Boolean variable and Boolean function call evaluate to a Boolean type value.
11.4 Function Calls.

FUNCTION_CALL = FUNCTION_IDENTIFIER ( '(' EXPRESSION / ',' ')' )?.

A function call invokes the function named with the actual parameters if any, listed in parentheses. The function returns a value which is used in further evaluation of the expression in which the function call appears. All parameters are passed by value. To eliminate aliasing of global node variables, no actual parameter expression may evaluate to a variable which is a global node value. If the function has no parameters the parentheses and colon must not appear.

12 STATEMENTS.

Statements are constructs which cause actions such as changing the value of a variable. They may occur in the main body of a program, in productions and in function declarations.

STATEMENT_LIST = STATEMENT / ';'.

STATEMENT = ASSIGNMENT_STATEMENT | ALIAS_STATEMENT | LOOP_STATEMENT | EXIT_STATEMENT | CASE_STATEMENT | IF_STATEMENT | FOR_STATEMENT | CALL_STATEMENT | RETURN_STATEMENT | FAIL_STATEMENT.

12.1 The Assignment Statement.

The assignment statement takes the value given by an expression and gives it to the variable specified. This variable must have use mode W or V.

ASSIGNMENT_STATEMENT = 'assign' EXPRESSION 'to' VARIABLE.

When node values with use mode V are assigned sharing takes place, that is, a node may be denoted in more than one way. If X and Y are both variables and X denotes a node with use mode V:

assign X to Y

means that X and Y both denote the same node. Thus, adding a branch selected by 'new-node' to the node may be accomplished (when Z is not null) by either:

assign Z to X.'new-node' or
assign Z to Y.'new-node'
Appendix 2 - SYNTRAN

which are equivalent statements. Assigning a value directly to either X or Y removes this sharing. Note that immediately after:

assign A to B
A = B is always true.

If the expression in an assignment statement consists of a variable with use mode R, E or C and a node type value the node value is copied rather than shared. If sharing is required the alias statement should be used. If the expression evaluates to a Boolean or string type value copying takes place.

An abbreviated form of the assignment statement can be used when a number of similar assignment statements are required. If A, B, C, Z, a, b, c and z represent strings in the text of a SYNTRAN program not containing unbalanced and unquoted square brackets ' [ ' and ' ] ' or unquoted commas ',' then:

assign A[B, C, ..., Z] to a[b, c, ..., z]

is equivalent to:

assign AB to ab;
assign AC to ac;
   ...
assign AZ to az

12.2 The Alias Statement.

The alias statement can only be used in productions.

ALIAS_STATEMENT = 'alias' IDENTIFIER 'for' NODE_EXPRESSION.

The identifier must have been declared in a local declaration with use mode R or W. The variable denoted by the identifier is given the node value specified by the expression so that the node is shared. The use mode of the node variable specified by the node expression must be the same as the use mode of the identifier.

12.3 The Loop and Exit Statements.

The loop statement simply repeatedly executes the statements within it until an exit statement is executed when the loop statement is finished.

LOOP_STATEMENT = 'loop' STATEMENT_LIST 'endloop'.
EXIT_STATEMENT = 'exit'.

The exit statement can only appear within a loop statement.

12.4 The Case Statement.

\[
\text{CASE STATEMENT} = \text{'case'}\ \text{STRING EXPRESSION}
- (\text{'of'}\ \text{STRING EXPRESSION} ,\ '':\ \text{STATEMENT_LIST})+
(\text{'otherwise'}\ \text{-STATEMENT_LIST})?\ \text{'endcase'}.
\]

The expression after the keyword 'case' is evaluated and compared with the values of the string expressions after the keywords 'of' in turn, until one is found to be equal or they have all been evaluated. If one is found to be equal, the statement list following that string expression is executed. If none is found to be equal, the statement list following the keyword 'otherwise', if present, is executed. The case statement is then finished.

12.5 The If Statement.

\[
\text{IF STATEMENT} = \text{'if'}\ \text{BOOLEAN EXPRESSION} \text{'then'}\ \text{STATEMENT_LIST}
- (\text{'else'}\ \text{STATEMENT_LIST})?\ \text{'endif'}.
\]

If the Boolean expression evaluates to true the statement list following the keyword 'then' is executed, otherwise the statement list following the keyword 'else', if present, is executed. The if statement is then finished.

12.6 The For Statement.

\[
\text{FOR STATEMENT} = \text{'for'}\ \text{IDENTIFIER} \text{'from'}\ \text{INTEGER EXPRESSION}
- (\text{'to'}|\text{'downto'})\ \text{INTEGER EXPRESSION} \text{'do'}\ -
\text{STATEMENT_LIST} \text{'endfor'}.
\]

The identifier is declared local to the statement list with use mode R and must be different from all other identifiers whose scope includes the for statement. The execution of the for statement is described by the following equivalent SYNTRAN constructs:
for I from E1 to E2 do S endfor; is equivalent to:

assign E1 to I;
assign E2 to TEMP;
if I <= TEMP then
  loop
    S;
    assign I + 1 to I;
    if I > TEMP then exit endif
  endloop
endif;

for I from E1 downto E2 do S endfor; is equivalent to:

assign E1 to I;
assign E2 to TEMP;
if I >= TEMP then
  loop
    S;
    assign I - 1 to I;
    if I < TEMP then exit endif
  endloop
endif;

where I and TEMP are both variables local to the expanded statement sequence and TEMP denotes a variable not otherwise used in the SYNTRAN program.

12.7 The Call Statement.

CALL STATEMENT = 'call' FUNCTION IDENTIFIER
                  '(' EXPRESSION / ',' ')'?.

The call statement simply invokes the function specified by the function identifier with actual parameters, if any, listed after the identifier in parentheses. Parameters are called by value and the expressions are evaluated in order during the execution of the call statement before the function is executed. To eliminate aliasing of global node variables no expression may evaluate to a variable which is a global node value. Any value returned by the function is discarded.

12.8 The Return Statement.

RETURN STATEMENT = 'return' EXPRESSION?.
The return statement may only occur in function declarations. The function is exited with the value given by the expression, if present, and null otherwise.

12.9 The Fail Statement.

FAIL_STATEMENT = 'fail'.

The fail statement may only occur in productions. It causes the immediate failure of the translator in which it appears.

13 DECLARATIONS.

DECLARATION = GLOBAL DECLARATION | LOCAL DECLARATION | FUNCTION DECLARATION | SET DECLARATION | TOKEN DECLARATION | TRANSLATOR DECLARATION | MACRO DECLARATION | IGNORE DECLARATION | CONTROL DECLARATION | EMISSION DECLARATION.

13.1 The Global Declaration.

GLOBAL DECLARATION = 'global' ID_LIST.
ID_LIST = IDENTIFIER / ',',.

The global declaration declares the identifiers listed as global variables with initial value null and use mode V. Global declarations can only appear outside other declarations in the main body of a GLADYS program.

13.2 The Local Declaration.

LOCAL DECLARATION = 'local' ID_LIST 'use' ('R' | 'V' | 'W').

The local declaration declares the identifiers listed as local variables with use mode as specified and with scope the function, translator or macro in which it appears. The identifiers listed must be different from all globally declared identifiers and other identifiers declared locally in the same function, translator or macro including those declared as formal parameters or in patterns.
13.3 The Function Declaration.

FUNCTION DECLARATION = 'function' IDENTIFIER ('(' ID LIST ')')?
LOCAL DECLARATION* STATEMENT_LIST 'endfunction'.

The identifier following the keyword 'function' is declared to be a function. The formal parameters, if any, are listed in parentheses after the identifier. The formal parameters are identifiers declared to be variables local to the function with use mode R. They are initialized with the values of the corresponding actual parameters when the function is called.

After the function has been called, by a call statement or by appearing in an expression, and the parameters passed, the statement list is executed. A return statement in this list causes the function to return control to where it was called from with the value specified by the expression in the return statement. If the statement list is exhausted without encountering a return statement, the function returns with the value null.

13.4 The Set Declaration.

SET DECLARATION = 'set'
   -(IDENTIFIER 'not'? '<' SET_DEF '>' / ')'.

SET_DEF = ((('''' CH '''' ('..'''' CH ''''))? | ''eof' | ''eol') / ''').

where CH is any character.

Each identifier is declared as a set identifier which represents a set of characters and pseudo-characters ('eof' and 'eol'). The characters available depend on the implementation but the pseudo-characters do not. The 'eol' pseudo-character acts as a single character which represents the end of a line on the input stream. The 'eof' pseudo-character also acts as a single character which represents the end of the input stream. The pseudo-characters may or may not be equivalent to characters in the implementation defined character set.

The set of characters represented by a set identifier is built up as follows:

1. A single character in quotes represents the set containing just that character.

2. A pseudo-character represents the set containing just that pseudo-character.
3. Two characters, each in quotes, separated by two dots represents the set of characters containing the two characters and all those between them as defined by the character set ordering in the specific implementation. The first character must occur before the second in the ordering.

4. Each of the set representations in 1, 2 and 3 separated by commas are united (using set union) together.

5. If the keyword 'not' appears after the set identifier, the set calculated so far is complemented with respect to the implementation defined character set united with the pseudo-character 'eol'.

Set identifiers are used in patterns and token declarations.

13.5 The Token Declaration.

```
TOKEN_DECLARATION = 'token' (IDENTIFIER '<' TOKEN_DEF '>') / ','.
TOKEN_DEF = (TOKEN_ELEMENT ('+' | '*' | '?')) / '"'.
TOKEN_ELEMENT = LITERAL | SET_IDENTIFIER | TOKEN_IDENTIFIER.
SET_IDENTIFIER = IDENTIFIER.
TOKEN_IDENTIFIER = IDENTIFIER.
```

Each identifier is declared to be a token which is a simple syntactic entity, built up from string literals, sets and other previously defined tokens. Token definitions may not be recursive, that is, a token identifier may not appear in its own definition. A token is built up as follows:

1. A string literal represents that string, a set identifier represents any one of the characters in that set and a token identifier represents a character string represented by that token.
2. A token element may have a postfix operator '+', '*' or '?' which indicate, in turn, one or more, zero or more, or zero or one occurrences of the string represented by the token element.
3. A number of the above constructs may be concatenated together using the '"' operator to form a token.

Tokens are used in defining patterns and in the emission declaration.
13.6 The Translator Declaration.

TRANSLATOR DECLARATION = 'translator' IDENTIFIER

('distinguished' | '({ ID_LIST? ':' ID_LIST? }')?)

LOCAL DECLARATION* (PATTERN PRODUCTION)+

'endtranslator'.

The identifier following the keyword 'translator' is declared to be a translator. The formal parameter identifiers are declared in parentheses following the translator identifier. The identifiers preceding the colon are declared as variables with use mode R and are given their values when the translator is called. The identifiers following the colon are declared as variables with use mode W and initial value null.

The body of a translator consists of one or more patterns each followed by a production. Patterns and productions are executed in the order that they appear. Patterns match the input stream, and, if successful, allow the execution of the following production. Separate patterns are implicitly concatenated together. A production, when executed, specifies a string value to be returned by the translator. The result of each production is concatenated together to form the result of the translator. If a pattern fails, a production fails or a fail statement is executed the translator as a whole fails, and the input stream is restored to the state it was in when the translator was called but any changes made by the translator to global variables stand.

13.7 The Macro Declaration.

MACRO DECLARATION = 'macro' IDENTIFIER

('({ ID_LIST? ':' ID_LIST? }')?)

LOCAL DECLARATION* PRODUCTION

'endmacro'.

The identifier following the keyword 'macro' is declared to be a macro. Formal parameters, if any, are listed in parentheses after the identifier. The identifiers listed before the colon are declared to be variables with use mode R and those listed after the colon are declared to be variables with use mode E. Formal parameters are given their initial values when the macro is called. Macros can only be called from within a production. The production in a macro declaration along with the actual parameters passed to the macro when called, specifies the emission of, or string returned by, the macro.
Appendix 2 - SYNTRAN

13.8 The Ignored Pattern Declaration.

\[ \text{IGNORE\_DECLARATION} = 'ignore' \text{\ Pattern\_Expression} \text{\ 'endignore'}. \]

The use of the ignored pattern is described at the end of section 14.

13.9 The Control Declaration.

\[ \text{CONTROL\_DECLARATION} = 'control' \text{\ IDENTIFIER} \text{\ ( 'boolean' \ BOOLEAN\_EXPRESSION | 'integer' \ INTEGER\_EXPRESSION) }. \]

The identifier following the keyword 'control' is declared to be a variable with use mode C. Its value and type is given by the Boolean or integer expression. A control declaration may appear outside all other declarations or inside a production. In the former case the identifier is declared globally. In the latter case the variable declared is local to the translator in which it is declared but can only be used after it is declared.

13.10 The Emission Declaration.

\[ \text{EMISSION\_DECLARATION} = 'emission' \text{\ IDENTIFIER \ 'is' \ TOKEN\_IDENTIFIER \ ':' \ STRING\_EXPRESSION}. \]

The identifier following the keyword 'emission' is declared to be a variable with use mode E with value given by the string expression. This value must conform to the syntax of the token specified by the token identifier. An emission declaration may appear outside all other declarations or inside a production. In the former case the identifier is declared globally. In the latter case the variable declared is local to the translator in which it appears but can only be used after it is declared.

14 PATTERNS.

Patterns specify a set of strings that will be successfully matched from the input stream and allow substrings of the matched string to be assigned to variables declared in the pattern.

\[ \text{PATTERN} = 'pattern' \text{\ PATTERN\_EXPRESSION \ 'endpattern'}. \]

\[ \text{PATTERN\_EXPRESSION} = \text{\ PATTERN\_TERM} / ' \mid '. \]
Appendix 2 - SYNTRAN

PATTERN_TERM = PATTERN_FACTOR / '`.`.
PATTERN_FACTOR = PATTERN_PRIMARY ('/' PATTERN_PRIMARY)?.
PATTERN_PRIMARY = (IDENTIFIER '←'? )? PATTERN_ELEMENT ('? ' | '+' | '*')?.
PATTERN_ELEMENT = '<' PATTERN_EXPRESSION '>' | LITERAL | SET_IDENTIFIER | TOKEN_IDENTIFIER | TRANSLATOR_CALL.
TRANSLATOR_CALL = IDENTIFIER ( '(' STRING_EXPRESSION / ',' )? ':' (IDENTIFIER / ',' )? ')')?.

Pattern elements are distinguished by the strings they match in the input stream and the strings they return when they match. A pattern element consisting of a pattern expression enclosed in angle brackets matches and returns the same strings as the pattern expression itself, the angle brackets being used only for grouping. A literal matches and returns itself and set and token identifiers match and return the strings they matched as specified by their respective set and token declarations. A translator call matches the string specified by the pattern or patterns in that translator and returns a string value as specified by the production or productions in that translator.

In a translator call the number of expressions before the colon and the number of identifiers after the colon must match the number of parameters before and after the colon in the declaration of that translator. The expressions are evaluated when the translator is called and passed by value to the corresponding formal parameters. As with functions, no expression may evaluate to a global node value. The identifiers after the colon are declared to be variables with use mode R local to the translator in which the call appears. These variables are given their values by the called translator. If the translator has no parameters the parentheses and colon must not appear.

A pattern primary consists of a pattern element which may have an identifier to the left of the pattern assignment operator '←'. This identifier is declared as a variable with use mode E local to the translator in which it appears and with initial value null. The action of the assignment operator is described below. Also in a pattern primary, the pattern element may have a postfix operator which has a similar meaning to those used in token definitions. A '?' means the pattern element is optional, a '*' means the pattern element may occur zero or more times and a '+' means the pattern element may occur one or more times. A pattern primary returns the concatenation of the strings returned by each application of the pattern element.

Two pattern primaries may be arguments to the listing operator '／', forming a pattern factor, which indicates that one or more occurrences of the left argument are to be matched, each separated by
one occurrence of the right argument. The string returned by a pattern
factor is just the concatenation of the strings returned by each
pattern primary in the order they are matched. Alternatively, a
pattern factor may just consist of a single pattern primary.

A number of pattern factors may be concatenated together by the
concatenation operator 'A', to form a pattern term. The string
returned by a pattern term is the concatenation of the strings
returned by each pattern factor in order.

A number of pattern terms may be joined together by the
alternation operator '|', to form a pattern expression. The
alternation operator has the effect of applying its left argument and
only if this fails to match then its right argument is applied. From a
number of alternative pattern terms either one succeeds (matches
successfully) or the whole pattern expression fails. A pattern
expression returns the string returned by the successful pattern term.

A pattern consists of a pattern expression surrounded by the
keywords 'pattern' and 'endpattern'. The value returned by a pattern
is discarded.

The pattern assignment operator has the highest priority of all
pattern operators and is used to examine parts of the input stream
matched by the pattern expression. The form of the value of the
variable declared by appearing on the left-hand side of the pattern
assignment operator is governed by the number of times the pattern
element on the right-hand side of that operator is matched. If the
pattern element can only be matched zero or one times the value of the
variable is null or the string returned by the pattern element,
respectively. If the pattern element is part of an argument to any of
the operators '+', '*' or '/' then it may be matched more than once.
In this case the variable will have a node value. If, in fact, the
pattern element is not matched at all the value of the variable is
null, as before; if the pattern element is matched once, though, the
variable has as its value a branch selected by the string '1' and with
value the string returned by the pattern element. If the pattern
element is matched twice or more times then the variable has as its
value a node with two or more branches each selected by '1', '2' ..., 
and so on up to the number of branches and the number of times the
pattern element matched, each branch having the string value as
returned by each instance of matching of the pattern element. Thus if
LETTER denotes the set of letter characters and the pattern primary:

\[
L \leftarrow \text{LETTER}^+ \\
\]

matches the string:

'E\text{X}A\text{M}\text{P}\text{L}'

in the input stream the value of L will be a node with seven branches
selected by '1', '2', '3', ..., '7' with values, in turn, the strings
'E', 'X', 'A', ..., 'E'.

Between each pair of pattern elements matched in the input stream an instance of the ignored pattern is matched implicitly. If an ignored pattern has not been declared it is assumed to match only the null string. The strings matched by the ignored pattern are discarded immediately and do not affect the strings returned by the pattern elements. The ignored pattern is usually declared to include the null string. The pattern expression in the ignored pattern declaration cannot contain any translator or macro calls of any pattern assignment operators.

15 PRODUCTIONS.

Productions specify the strings to be returned as strings from the translators they appear in and eventually to be emitted to the output stream. They are, in this way, complementary to patterns.

PRODUCTION = 'production' PRODUCTION_EXPRESSION 'endproduction'.

PRODUCTION_EXPRESSION = STATEMENT_PART? PRODUCTION_TERM.

STATEMENT_PART = ' [ ' (STATEMENT | EMISSION_DECLARATION | CONTROL_DECLARATION) / ';' ' ]'.

PRODUCTION_TERM = CONCATENATION_TERM | ALTERNATION_TERM | OPTION_TERM | LISTING_TERM | REPETITION_OPTION_TERM | REPETITION_TERM.

PRODUCTION_FACTOR = PRODUCTION_PRIMARY / ' ^ '.

PRODUCTION_PRIMARY = STATEMENT_PART? EMISSION_PART.

EMISSION_PART = LITERAL | VARIABLE | MACRO_CALL | '<' PRODUCTION_TERM ' > '.

MACRO_CALL = IDENTIFIER '(' (EXPRESSION / ',')? ':' ((LITERAL | VARIABLE) / ',')? ')')?.

Each production primary consists of an optional statement part and an emission part. The statement part may include the emission and control declarations and any statements except the return statement. The execution of a production element consists of executing any statements and declarations in the statement part and then emitting the string specified by the emission part. If the emission part is a
literal, then the string represented by the literal is emitted. If the emission part is a variable, it must have a string value and use mode E and its value is the value emitted. If the emission part is a production term enclosed in angle brackets, the production part is executed and the emission from it is the emission from the emission part. Lastly, if the emission part is a macro call, the macro specified by the identifier is called and the emission of the production executed in the macro becomes the emission of the emission part. Any actual parameters after the colon in the parameter list of the macro call must be literals or variables with use mode E and any actual parameters before the colon are expressions which are evaluated when the macro is called. As with actual parameters for functions such an expression cannot evaluate to a global node variable. If there are no parameters to the macro the parentheses and colon must not appear.

If it is possible for a variable declared in a pattern to have the null value due to the fact that its associated pattern element was never matched then it must occur in a concatenation term, an alternation term or an option term which has a Boolean control. This Boolean control must guarantee that if the variable has the null value it will not be used to form the emitted string of that term. The use of such variables is not restricted in normal expressions.

A number of production primaries can be concatenated to form a production factor. The string emitted by a production factor is the concatenation of the strings emitted by each production primary in order.

A production expression consists of an optional statement part followed by a production term. Execution of a production expression consists of the execution of the statement part, if present, followed by the execution of the production term. The statement part may contain emission and control declarations as well as normal statements. Although each of these declarations declares a single identifier and variable with local scope they may be executed more than once. Each time they are executed a (usually) different value is given to the variable declared which may be used in following emission part. The action of executing each type of production term is described in the following sub-sections.

15.1 The Concatenation Term.

\[
\text{CONCATENATION_TERM} = (\text{BOOLEAN_CONTROL} \ '→'\)? \text{PRODUCTION_FACTOR}.
\]

\[
\text{BOOLEAN_CONTROL} = \text{IDENTIFIER}.
\]

The identifier in a Boolean control must denote a variable with use mode \( \text{C} \) and have a Boolean value. If the Boolean control is present and evaluates to true or is not present the emission of the
concatenation term is just the emission of the production factor. If the Boolean control evaluates to false the term fails, causing failure of the translator in which it appears.

15.2 The Alternation Term.

\[
\text{ALTERNATION TERM} = \text{BOOLEAN CONTROL} \Rightarrow \text{PRODUCTION FACTOR '}|' \text{PRODUCTION FACTOR}.
\]

If the Boolean control evaluates to true the emission of the alternation term is the emission of the production factor preceding the alternation operator '}', otherwise it is the emission of the production factor following the alternation operator.

15.3 The Option Term.

\[
\text{OPTION TERM} = \text{BOOLEAN CONTROL} \Rightarrow \text{PRODUCTION PRIMARY '}?'.
\]

If the Boolean control evaluates to true the emission of the option term is the emission of the production primary otherwise it is null.

15.4 The Listing Term.

\[
\text{LISTING TERM} = \text{INTEGER CONTROL} \Rightarrow \text{PRODUCTION PRIMARY '} /' \text{PRODUCTION PRIMARY}.
\]

\[
\text{INTEGER CONTROL} = \text{IDENTIFIER ('( IDENTIFIER ')')}?.
\]

The first identifier in the integer control must denote a variable called the control variable with use mode C, and with a positive value. The optional second identifier is declared to be a variable with use mode R local to the listing term. This identifier must be different from all other identifiers in whose scope the listing term is contained. The control variable determines the number of times the production primary following the '→' symbol is executed. In the case of the listing term the production primary following the listing operator '/', is executed one fewer times than the first production primary and these executions are interspersed between the executions of the first production primary. If the second identifier is present, the variable it denotes is called the index variable which is given the values 1, 2, ..., etc up to the value of the control variable, one for each time the production primary following the '→'
is repeated. It may be used in production primaries as an indexing variable. An error occurs if the control variable has a non-positive value. The emission of a listing term is the concatenation of the emissions of each of the production primaries in the order that they are executed.

15.5 The Repetition and Option Term.

\[
\text{REPETITION \_OPTION \_TERM} = \text{INTEGER \_CONTROL} \rightarrow \text{PRODUCTION \_PRIMARY} \rightarrow^* .
\]

The emission from the repetition and option term is null if the control variable has the value zero, otherwise it is the concatenation of the emissions of the production primary which is executed as many times as is indicated by the value of the control variable. An error occurs if the control variable has a negative value. An indexing variable may be used as in the listing term.

15.6 The Repetition Term.

\[
\text{REPETITION \_TERM} = \text{INTEGER \_CONTROL} \rightarrow \text{PRODUCTION \_PRIMARY} \rightarrow^+ .
\]

The repetition term acts in the same way as the repetition and option term except that an error occurs if the value of the control variable is negative or zero.

16 LAYOUT OF A SYNTRAN PROGRAM.

A SYNTRAN program consists of a list of declarations and statements in any order except that variables, sets and tokens must be declared before they are used.

\[
\text{PROGRAM} = (\text{DECLARATION} \mid \text{STATEMENT}) / ';' .
\]

There must also be exactly one translator with the 'distinguished' keyword.

Execution of a SYNTRAN program starts with the execution, in order, of all statements and declarations not inside other declarations. Then the translator with the keyword 'distinguished' is invoked. If this translator fails the SYNTRAN program as a whole fails. If the distinguished translator succeeds then the defined translation has taken place from the input stream to the output stream.
17 REFERENCES.


18 SYNTAX OF SYNTRAN.

PROGRAM = (DECLARATION | STATEMENT) / ';'.

DECLARATION = GLOBAL DECLARATION | LOCAL_DECLARATION |
FUNCTION_DECLARATION | SET_DECLARATION | TOKEN_DECLARATION |
TRANSLATOR_DECLARATION | MACRO_DECLARATION |
IGNORE_DECLARATION | CONTROL_DECLARATION |
EMISSION_DECLARATION.

GLOBAL_DECLARATION = 'global' ID_LIST.

ID_LIST = IDENTIFIER / '','.

LOCAL_DECLARATION = 'local' ID_LIST 'use' ('R' | 'V' | 'W').

FUNCTION_DECLARATION = 'function' IDENTIFIER ('( ' ID_LIST ')')?
LOCAL_DECLARATION* STATEMENT_LIST 'endfunction'.

SET_DECLARATION = 'set' (IDENTIFIER 'not'? '<' SET_DEF '>') / '','.

SET_DEF = ((( ' ' CH ' ' ) ( '.. ' CH '.. ') ) |
'.eof' | 'eol') / '','.

TOKEN_DECLARATION = 'token' (IDENTIFIER '<' TOKEN_DEF '>') / '','.

TOKEN_DEF = (TOKEN_ELEMENT ('+' | '*' | '?')?) / '^'.

TOKEN_ELEMENT = LITERAL | SET_IDENTIFIER | TOKEN_IDENTIFIER.

SET_IDENTIFIER = IDENTIFIER.

TOKEN_IDENTIFIER = IDENTIFIER.

TRANSLATOR_DECLARATION = 'translator' IDENTIFIER
distincted ID_LIST? ':' ID_LIST? ')')?
LOCAL_DECLARATION* (PATTERN PRODUCTION)+
'endtranslator'.

MACRO_DECLARATION = 'macro' IDENTIFIER
token ID_LIST? ':' ID_LIST? ')')?
LOCAL_DECLARATION* PRODUCTION 'endmacro'.

IGNORE_DECLARATION = 'ignore' PATTERN_EXPRESSION 'endignore'.

CONTROL_DECLARATION = 'control' IDENTIFIER
('boolean' BOOLEAN_EXPRESSION |
'integer' INTEGER_EXPRESSION).

INTEGER_EXPRESSION = STRING_EXPRESSION.
EMISSION DECLARATION = 'emission' IDENTIFIER 'is' TOKEN IDENTIFIER ':' STRING EXPRESSION.

PATTERN = 'pattern' PATTERN_EXPRESSION 'endpattern'.

PATTERN_EXPRESSION = PATTERN_TERM / ' | '.

PATTERN_TERM = PATTERN_FACTOR / ' | '.

PATTERN_FACTOR = PATTERN_PRIMARY (' / ' PATTERN_PRIMARY)?.

PATTERN_PRIMARY = (IDENTIFIER ' <- ')? PATTERN_ELEMENT ('?' | '+' | '*')?.

PATTERN_ELEMENT = '<' PATTERN_EXPRESSION '>' | LITERAL | SET IDENTIFIER | TOKEN IDENTIFIER | TRANSLATOR_CALL.

TRANSLATOR_CALL = IDENTIFIER ('( STRING_EXPRESSION / ',' ')? ':' (IDENTIFIER / ',' ')')?.

PRODUCTION = 'production' PRODUCTION_EXPRESSION 'endproduction'.

PRODUCTION_EXPRESSION = STATEMENT PART? PRODUCTION_TERM.

STATEMENT PART = '[' (STATEMENT | EMISSION DECLARATION | CONTROL DECLARATION) / ';' ']'.

PRODUCTION_TERM = CONCATENATION TERM | ALTERNATION TERM | OPTION TERM | LISTING TERM | REPETITION OPTION TERM | REPETITION TERM.

CONCATENATION TERM = (BOOLEAN CONTROL ' -> ')? PRODUCTION_FACTOR.

BOOLEAN_CONTROL = IDENTIFIER.

ALTERNATION TERM = BOOLEAN_CONTROL ' -> ' PRODUCTION_FACTOR | PRODUCTION_FACTOR.

PRODUCTION_FACTOR = PRODUCTION_PRIMARY / ' | '.

OPTION TERM = BOOLEAN_CONTROL ' -> ' PRODUCTION_PRIMARY '?'.

LISTING TERM = INTEGER_CONTROL ' -> ' PRODUCTION_PRIMARY '/'.

INTEGER_CONTROL = IDENTIFIER ('( IDENTIFIER ')')?.

REPETITION OPTION TERM = INTEGER_CONTROL ' -> ' PRODUCTION_PRIMARY '*'.

REPETITION TERM = INTEGER_CONTROL ' -> ' PRODUCTION_PRIMARY '+'.


Appendix 2 - SYNTRAN

PRODUCTION_PRIMARY = STATEMENT_PART? EMISSION_PART.

EMISSION_PART = LITERAL | VARIABLE | MACRO_CALL |
               '"' PRODUCTION_TERM '"'.

MACRO_CALL = IDENTIFIER ('("' (EXPRESSION / ',')? ':'
                   (LITERAL | VARIABLE) / ',')? '))?.

STATEMENT_LIST = STATEMENT / ';'.

STATEMENT = ASSIGNMENT_STATEMENT | ALIAS_STATEMENT |
             LOOP_STATEMENT | EXIT_STATEMENT | CASE_STATEMENT |
             IF_STATEMENT | FOR_STATEMENT | CALL_STATEMENT |
             RETURN_STATEMENT | 'FAIL_STATEMENT'.

ASSIGNMENT_STATEMENT = 'assign' EXPRESSION 'to' VARIABLE.

ALIAS_STATEMENT = 'alias' IDENTIFIER 'for' NODE_EXPRESSION.

LOOP_STATEMENT = 'loop' STATEMENT_LIST 'endloop'.

EXIT_STATEMENT = 'exit'.

CASE_STATEMENT = 'case' STRING EXPRESSION |
               ('of' STRING EXPRESSION | ':' STATEMENT_LIST)+ |
               ('otherwise' STATEMENT_LIST)? 'endcase'.

IF_STATEMENT = 'if' BOOLEAN EXPRESSION 'then' STATEMENT_LIST |
               ('else' STATEMENT_LIST)? 'endif'.

FOR_STATEMENT = 'for' IDENTIFIER 'from' INTEGER_EXPRESSION |
               ('to' | 'downto') INTEGER_EXPRESSION 'do' |
               STATEMENT_LIST 'endfor'.

CALL_STATEMENT = 'call' FUNCTION IDENTIFIER |
                ('("' EXPRESSION / ',')? ').

RETURN_STATEMENT = 'return' EXPRESSION?.

FAIL_STATEMENT = 'fail'.

EXPRESSION = BOOLEAN_EXPRESSION | STRING_EXPRESSION |
             NODE_EXPRESSION.

BOOLEAN_EXPRESSION = BOOLEAN_FACTOR / ('!' | '!'').

BOOLEAN_FACTOR = BOOLEAN_FACTOR / '&'.

BOOLEAN_PRIMARY = NODE_EXPRESSION EQ_OP NODE_EXPRESSION |
                  STRING_EXPRESSION REL_OP STRING_EXPRESSION |
                  '"'? (BOOLEAN_VARIABLE | BOOLEAN_FUNCTION_CALL |
                  ('"' BOOLEAN_EXPRESSION '"')).
Appendix 2 - SYNTRAN

BOOLEAN_VARIABLE = VARIABLE.

BOOLEAN_FUNCTION_CALL = FUNCTION_CALL.

REL_OP = '<' | '>' | '<=' | '=>' | EQ_OP.

EQ_OP = '=' | '=='.

STRING_EXPRESSION = STRING_FACTOR / '^'.

STRING_FACTOR = STRING_TERM / ADD_OP.

STRING_TERM = STRING_PRIMARY / MUL_OP.

STRING_PRIMARY = STRING_UNIT / '%'.

STRING_UNIT = '#'? STRING_ELEMENT | '#' NODE_EXPRESSION.

STRING_ELEMENT = LITERAL | STRING_VARIABLE |
STRING_FUNCTION_CALL | '(' STRING_EXPRESSION ').'

STRING_VARIABLE = VARIABLE.

STRING_FUNCTION_CALL = FUNCTION_CALL.

ADD_OP = '+' | '-'.

MUL_OP = '*' | '/'.

LITERAL = NUMBER | '***' NOTSQ* '***' | '**' NOTDQ* '**' | 'null'.

NUMBER = ('+' | '-')? DIGIT+.

FUNCTION_CALL = FUNCTION_IDENTIFIER ('(' EXPRESSION / ',' ')')?.

NODE_EXPRESSION = '!' NODE_EXPRESSION | NODE_VARIABLE |
FUNCTION_CALL.

NODE_VARIABLE = VARIABLE.

NODE_FUNCTION_CALL = FUNCTION_CALL.

VARIABLE = IDENTIFIER ('.' STRING_EXPRESSION)*.

IDENTIFIER = LETTER (LETTER | DIGIT | '_')*.

LETTER = 'A' | 'B' | ... | 'Z'.

DIGIT = '0' | '1' | ... | '9'.

SPECIAL SYMBOL = '+' | '-' | '*' | '/' | '#' | '|' | '?' | '
' | '=' | '==' | '<=' | '>=' | '<' | '>' | '~' | '&' | '%' | '
' | ':' | '<-' | '<=>' | '[' | ']' | '_' | '1' | '
'

RESERVED WORD = 'assign' | 'alias' | 'boolean' | 'call' | 'case' | 'control' | 'distinguished' | 'do' | 'downto' | 'else' | 'emission' | 'endcase' | 'endfor' | 'endfunction' | 'endif' | 'endignore' | 'endloop' | 'endmacro' | 'endpattern' | 'endproduction' | 'endtranslator' | 'eof' | 'eol' | 'exit' | 'fail' | 'for' | 'from' | 'function' | 'global' | 'if' | 'ignore' | 'integer' | 'is' | 'local' | 'loop' | 'macro' | 'new' | 'not' | 'null' | 'of' | 'otherwise' | 'pattern' | 'production' | 'return' | 'set' | 'substr' | 'then' | 'to' | 'token' | 'translator' | 'use'.

COMMENT = '{' NOT_RIGHT_BRACE* '}'. 
GLADYS is a high levelemerger written in English which was designed to be used as a native grammar of high level programming languages. It is intended to be used in conjunction with a system of the latest language of computer science translation. The concepts used in current high level languages have been used to form the basis for GLADYS so that the difficulty of presenting a computer program in GLADYS is reduced.
GLADYS

a Target Language for Semantics Defining Translations

S. J. Edwards
Department of Computer Science
Australian National University
Canberra

May 1980

GLADYS is a high level abstract machine language which was designed to be used to define semantics of high level programming languages. It is intended to be used in conjunction with SYTRAN as the target language of semantics defining translations. The concepts used in current high level languages have been used to form the basis for GLADYS so that the difficulty of translating a source language to GLADYS is reduced.
1 INTRODUCTION.

GLADYS is intended to be a high level abstract machine language suitable for defining the semantics of high level programming languages. The definition takes the form of a SYNTRAN program which specifies the translation from the language being defined, (L, say) to GLADYS. It is felt that describing the process of translation of a high level language, L, to another high level language, GLADYS in this case, is less tedious, simpler and less error prone than describing the process of translation from L to an intermediate or low level language. For further motivation see [Edwa80a].

GLADYS is designed not only to be easy to use as an object of translations but also for the ease of extracting the structure of the language L. It is envisaged that the inspection of the use of the GLADYS primitives used in any particular definition will enable the choice of efficient data and control structures to be made mechanically, and thus create, or at least describe, a viable implementation of L. Since GLADYS is high level, it imposes less structure of its own on the definition of L than a lower level language with the same purpose.

GLADYS can also be seen as a candidate for the solution of the UNCOL problem [Stee61] though GLADYS was not designed for this purpose. Rather, GLADYS is designed to be a vehicle for defining the 'features' required for an implementation of the language L.
2 OVERVIEW.

The following diagram illustrates the basic concepts of GLADYS and their interaction.

Each class of objects is drawn as a box with arrows depicting interactions between objects and other interactions. Numbers in square brackets throughout the following text are references to the interactions numbered in the above diagram.

Because of these interdependencies it is impossible to describe any part of GLADYS without referring to another part. This means no description of GLADYS can treat each concept in turn without referring to concepts not yet described. As a compromise the concepts depicted above will be described in the order: identifiers, baskets, cells, types and data. Following these, the further topics: assignment and expressions, control structures and primitive operations on data will be discussed.

There are some situations which are described to cause an 'error'. When this occurs the effect of further execution of the GLADYS program is undefined. There are also many places in the following description where phrases such as "can only" and "must" etc. are used: it is implied that if these conditions are not met, the execution of the affected part of the GLADYS program will also cause an error.
3 NOTATION AND VOCABULARY.

The syntax of GLADYS constructs is defined in this report by an extended BNF described in [EdwaS80bJ. The vocabulary of GLADYS consists of the whole printable ASCII character set and some reserved words represented in lower case letters. Some syntactic categories are listed and defined here for later use.

LETTER = 'A' | 'B' | .... | 'Z'.

DIGIT = '0' | '1' | .... | '9'.

ID_SPECIAL = ' _ ' | '$' | '%' | '@' | ' ' | '_' | '.' | '-' |

SPECIAL_SYMBOL = ' ' | '!' | '=' | '<' | '>' | '|' | '*' | '+' | '-' | '@' | '
' | ':' | ';' | '(' | ')' | '[' | ']' | ']' | ']' |

assign | basket | bind | call | cell |

copies | create_basket | create_cell | current |

current_basket | destroy | destroy_closed |

enclosing | error | external | in |

index | interval | is | loop | next |

no_basket | no_cell | no_type | null |

null_type | process | running | start | stop |

stopped | structure | this_process | to |

transfer | type | type_of | union | wait | with |

ASCII_SYMBOL = 'nul' | 'soh' | 'stx' | 'etx' | 'eot' | 'enq' |

ack | 'bel' | 'bs' | 'tab' | 'lf' | 'vt' | 'ff' | 'cr' |

so | 'si' | 'die' | 'dc1' | 'dc2' | 'dc3' | 'dc4' |

nak | 'syn' | 'etb' | 'can' | 'em' | 'sub' | 'esc' |

fs | 'gs' | 'rs' | 'us' | 'del'.

PRINTABLE ASCII CHARACTER = LETTER | DIGIT | ID_SPECIAL |

'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' |

'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'.

Non-terminal symbols are written in upper case characters and are referred to in the text by the same characters but in lower case except that the underline character '_-' is replaced by a space.
Identifiers are written using the following syntax:

\[
\text{IDENTIFIER} = \text{LETTER} \mid \text{DIGIT} \mid \text{ID_SPECIAL}^*.
\]

Bindings of identifiers to cells, baskets or types [1, 4, 6] may be set up and placed in a basket.

An identifier binding is set up using the binding statement:

\[
\text{BINDING \_STATEMENT} = \text{'bind'} \text{ IDENTIFIER} \text{'to'} \text{ OBJECT\_EXPRESSION} \text{'in'} \text{ PLACING\_BASKET}?.
\]

\[
\text{OBJECT\_EXPRESSION} = \text{CELL\_EXPRESSION} \mid \text{BASKET\_EXPRESSION} \mid \text{TYPE\_EXPRESSION}.
\]

\[
\text{PLACING\_BASKET} = \text{BASKET\_EXPRESSION}.
\]

Identifier is bound to the object denoted by object expression and this binding is placed in a basket. If placing basket is present and not 'no_basket' then it specifies the basket where the newly created binding is placed. If it is not present the binding is placed in the current basket (see section 7.4) if one exists. In all other cases an error occurs. An error also occurs if a binding with the same identifier already exists in the same basket.
A basket is an object which:
1. contains identifier bindings to cells, baskets and types [2, 3, 5];
2. contains cells [7]; and
3. may refer to another basket [9], called the enclosing basket.

BASKET_CREATOR = 'create_basket'.

The basket creator is a function which creates and returns a new basket that contains no identifier bindings, no cells and no enclosing basket.

A basket is given an enclosing basket by the enclosing statement.

ENCLOSING_STATEMENT = 'enclosing' ENCLOSED_BASKET_EXPRESSION 'is' ENCLOSED_BASKET_EXPRESSION.

ENCLOSED_BASKET_EXPRESSION = BASKET_EXPRESSION.

BASKET_EXPRESSION = EXPRESSION.

ENCLOSING_BASKET_EXPRESSION = BASKET_EXPRESSION.

where basket expression must evaluate to a basket.

The basket specified by the enclosed basket expression is given the enclosing basket specified by the enclosing basket expression. An error occurs if the enclosed basket expression evaluates to 'no_basket', but the enclosing basket expression may evaluate to
'no_basket'. The effect of the enclosing statement in such case is to remove any enclosing basket for the basket specified by the enclosed basket expression.

A basket may be destroyed by using the destroy statement.

DESTROY_STATEMENT = 'destroy' BASKET_EXPRESSION.

This destroys all identifier bindings and cells in the basket specified. Any identifiers bound to cells, baskets or types that have been destroyed become bound to 'no_cell', 'no_basket' and 'no_type', respectively. In particular, any cells referring to the destroyed basket are given the value 'no_basket' and any bindings between identifiers and the destroyed basket are removed. If the basket destroyed was the current basket then the enclosing basket of the destroyed basket becomes the current basket. If the destroyed basket had no enclosing basket an error occurs.

DESTROY_ENCLOSED_STATEMENT = 'destroy_enclosed' BASKET_EXPRESSION.

This statement destroys all baskets which are enclosed by the specified basket and all baskets enclosed by baskets so destroyed and all baskets enclosed by those and so on. If the current basket is destroyed by this statement then the basket specified becomes the current basket.

The primitive methods of specifying a basket are:

BASKET = 'no_basket' | 'current_basket' | BASKET_CREATOR.

'current_basket' is bound to the current basket of the process using it, if there is one and is equivalent to 'no_basket' otherwise.
Cells are used to store data values [17]. Each cell has a type [16] which determines the type of values that may be stored in that cell. A new cell is created by the cell creator function.

\[
\text{CELL CREATOR} = 'create \text{cell} '( ' \text{TYPE EXPRESSION} ' , ' \text{INITIAL VALUE} ' , ' \text{PLACING BASKET} ) ? ' ) .
\]

\[
\text{INITIAL VALUE} = \text{EXPRESSION} | \text{PROCESS INITIAL VALUE}.
\]

Process initial value is defined in section 7.4. The newly created cell has type specified by the type expression and a value specified by initial value which must be an expression of the type specified for the cell. An error occurs if type expression evaluates to 'no_type'. If placing basket is present and is not 'no_basket', then the cell is placed in that basket so specified. If it is not present, the cell is placed in the current basket, if it exists. In all other cases an error occurs.

The primitive methods of specifying a cell are:

\[
\text{CELL} = ' no_cell ' | \text{CELL CREATOR} | \text{PROCESS REF}.
\]

where process ref is described in section 7.4 below and:

\[
\text{CELL EXPRESSION} = \text{EXPRESSION}.
\]

where cell expression must evaluate to a cell.
Types in GLADYS define a set of values. Each cell has a type and so does each value of data. A cell of a particular type can only have values of that type. The type of a value may be determined by the 'type_of' function which returns the type of the value supplied as its argument.

\[
\text{TYPE\_FUNCTION} = \text{\textquote{type\_of\textquote{' EXPRESSION \textquote{'}}}}.
\]

Types fall into four categories: the integer type; the object types; the constructed types; and the process types. Each of these is dealt with in turn in the following sections. Cells cannot have the type integer.

\[
\text{TYPE\_CREATOR} = \text{OBJECT\_TYPE} \mid \text{CONSTRUCTED\_TYPE}.
\]

The primitive methods of specifying a type are:

\[
\text{TYPE} = \text{\textquote{no\_type\textquote{}\textquote{}}} \mid \text{TYPE\_CREATOR}.
\]

7.1 The Integer Type.

The integer type consists precisely of the mathematical set of integers. There is no construction for this type as no cells can store an integer value. Integer values are used as partial results in expressions and for communicating with the "outside world".
7.2 The Object Types.

\[
\text{OBJECT\_TYPE} = \text{'basket'} \mid \text{'cell'} \mid \text{'type'}.
\]

The basket type \([10]\) consists of all references to baskets which exist at any point in time during execution of a GLADYS program plus the value 'no\_basket', which is used as a null value and does not refer to any basket. The cell type \([12]\) and type type \([13]\) are defined in a similar way with references to cells and types and with null values 'no\_cell' and 'no\_type' respectively.

Thus the value of a cell can be a reference to a basket, type or another cell \([8, 15, 11]\). An object or the value of a cell may be obtained from a reference to that object or cell by the use of the postfix dereference operator '@'.

\[
\text{DEREFERENCE} = \text{REFERENCE\_TERM} \ ' @' .
\]

\[
\text{REFERENCE\_TERM} = \text{TERM}.
\]

where reference term must evaluate to a reference to an object.

Values of the same object type may be compared for equality or inequality by using the infix operator symbol "=" or "\~\~=". If the two basket, cell or type values refer to the same basket, cell or type, the result is the integer value one, else the result is the integer value zero in the former case and vice versa in the latter (inequality) case.

\[
\text{OBJECT\_RELATION} = \text{OBJECT\_FORM} (\ ' = ' \mid \ ' \~\~\~ = ' ) \text{OBJECT\_FORM}.
\]

\[
\text{OBJECT\_FORM} = \text{FORM}.
\]

where object form must evaluate to a basket, type or cell.

7.3 The Constructed Types.

The constructed types define sets of data values using the set of integers and three constructing methods.

\[
\text{CONSTRUCTED\_TYPE} = \text{INTERVAL\_TYPE} \mid \text{UNION\_TYPE} \mid \text{STRUCTURE\_TYPE} \mid \text{NULL\_TYPE}.
\]

\[
\text{NULL\_TYPE} = \text{'null\_type'}.\]

Null type is a type which has no values.
Every execution of a constructed type creates a distinct type.

Values of constructed types are ordered and the determination of equality or ordering of two values of the same type is accomplished by using the infix relational operators.

\[
\text{VALUE\_RELATION} = \text{VALUE\_FORM} \text{ RELATIONAL\_OPERATOR} \text{ VALUE\_FORM}.
\]

\[
\text{VALUE\_FORM} = \text{FORM}.
\]

\[
\text{RELATIONAL\_OPERATOR} = '=' | '~=' | '<=' | '>' | '<' | '>'.
\]

where value forms must evaluate to values of the same constructed type. The relational operators have the usual meanings. The orderings on the values of constructed types is defined in the following sub-sections.

7.3.1 The interval types.

An interval type is a set of values isomorphic to a contiguous finite subset (finite interval) of the integers.

\[
\text{INTERVAL\_TYPE} = 'interval' '(' \text{LOWER\_BOUND} ', ' \text{UPPER\_BOUND} ')'.
\]

\[
\text{LOWER\_BOUND} = \text{INTEGER\_EXPRESSION}.
\]

\[
\text{INTEGER\_EXPRESSION} = \text{EXPRESSION}.
\]

\[
\text{UPPER\_BOUND} = \text{INTEGER\_EXPRESSION}.
\]

where integer expression must evaluate to an integer value. The lower bound must have a value less than or equal to the upper bound so that the created type has at least one value. The cardinality (the number of values) of an interval type is the value of the upper bound minus the value of the lower bound plus one. Each value corresponds to an integer value in the range specified by the bounds. The correspondence, an isomorphism, between the set of values of an interval type and the interval in the set of integers is used by two functions that map or "convert" values between the two sets. The function that accepts a value of an interval type and returns the corresponding integer value is invoked by the prefix operator '#'.

\[
\text{INTEGER\_CAST} = '#' \text{ INTERVAL\_TERM}.
\]

\[
\text{INTERVAL\_TERM} = \text{TERM}.
\]

where interval term must evaluate to a value of an interval type. The partial function which accepts an integer value and returns a value of an interval type is invoked by using the interval type as a prefix to the integer expression enclosed in angle brackets. This construct is called a 'cast' after the similar concept in ALGOL68 [WijnA76,
where type element evaluates to a type and interval type name evaluates to an interval type. If the value of the integer expression does not have a corresponding interval type value, i.e. it is "out of range", an error occurs.

The values of an interval type are ordered in the natural way by their correspondence to integer values.

7.3.2 The union types.

A union type is constructed from other previously defined types and has a value corresponding to each value of each of the types used in its construction, called the member types of the union type. Previously defined types are those that have been created already by the execution of a type creator or those whose type creator is in the same parallel statement unit. The union type corresponds to the set sum or discriminated union of its member types. Thus the cardinality of a union type is the sum of the cardinalities of each of its member types.

UNION_TYPE = 'union' 'MEMBER_TYPE_EXPRESSION / ',' ')'.

MEMBER_TYPE_EXPRESSION = TYPE_EXPRESSION.

TYPE_EXPRESSION = EXPRESSION.

where type expression must evaluate to a type. Each member type expression must evaluate to a distinct type. A value of a member type may be "converted" to the corresponding value of the union type by the use of the cast construct, as with integer to interval type conversion.

UNION_CAST = UNION_TYPE_NAME 'MEMBER_EXPRESSION '>'.

UNION_TYPE_NAME = TYPE_ELEMENT.

MEMBER_EXPRESSION = EXPRESSION.

The union type name must evaluate to a union type and the member expression must evaluate to an value in a member type of the union type. The inverse function is also invoked by using the cast construct.
The member type name must evaluate to a member type of a union type and union expression must evaluate to a value of that union type and have a corresponding value in the member type.

The particular member type to which a value of the union type corresponds can be tested by invoking a function named by the member type.

These functions return zero if the union type value does not correspond to a value in the named member type and one if it does.

The values of a union type are ordered by using the ordering of each member type and the ordering in which the member type appears in the member type list in the definition of the union type. That is, all values corresponding to a member type appearing before another are less than the values corresponding to the second member type. The ordering within values from a single member type is inherited directly from that member type.

### 7.3.3 The structure types.

A structure type is also constructed from previously defined types. It has values isomorphic to each combination of values from each of its constructing types, thus the structure type represents the set product of its constructing types and the cardinality of a structure type is the product of the cardinalities of each of its constructing types. A structure type also has an indexing type which has one value for each of the constructing types. A structure type value consists of a number, equal to the cardinality of the indexing type, of cells. Associated with each cell is an indexing type value, and associated with each indexing type value is a constructing type. Each cell is the same type as its associated constructing type. The correspondence between indexing type values and the constructing types is made using the order in which the constructing types are written and the ordering on the indexing type.

```
STRUCTURE_TYPE = 'structure' '(' INDEXING_TYPE_EXPRESSION ')' / CONSTRUCTING_TYPE_EXPRESSION /

INDEXING_TYPE_EXPRESSION = TYPE_EXPRESSION.

CONSTRUCTING_TYPE_EXPRESSION = TYPE_EXPRESSION.
```
where indexing type expression must not evaluate to 'null_type'.

For example, with a structure type specified by
'structure(I;A,B,C)', I is the indexing type and A, B and C are the
constructing types. The cardinality of I must be 3. In each value of
the structure type the first value of I is associated with the first
cell which is of type A, the second value of I is associated with the
second cell which is of type B, and the third value of I is associated
with the third cell which is of type C.

A syntactic abbreviation is allowed in that if the same
constructing type is used for all values of the indexing type it may
be written once only instead of being repeated the number of times
required by the cardinality of the indexing type.

The ordering of the values of a structure type is inherited from
each constructing type and the order of the constructing types. That
is, the ordering between two structure type values can be determined
by inspecting the cells in order from the first (in the constructing
type list or the indexing type values) to the last and comparing their
values in each of the structure type values. As soon as a non-equal
pair is found the ordering of the two structure type values is defined
by the ordering of the two constructing type values. This is an
extension of the lexicographic ordering usually applied to strings.

A reference to a cell inside a structure type value can be
obtained by using the infix selection operator '.', which associates
to the left. Its left argument is a structure type value and its right
argument is a value from the indexing type of the left argument. The
result of the operation is a reference to the cell in the structure
type value which is associated with the indexing type value.

$$\text{SELECTION} = \text{STRUCTURE TERM} . \text{INDEXING PRIMARY}.$$  
$$\text{STRUCTURE TERM} = \text{TERM}.$$  
$$\text{INDEXING PRIMARY} = \text{PRIMARY}.$$  

where structure term evaluates to a value of a structure type and
indexing primary evaluates to a value of the indexing type of that
structure type.

Thus if S and T are two values of the same structure type with
indexing type values of i1, i2, ..., iN, then the ordering described
above can be defined as follows:

$$S < T = \begin{cases} 
\text{if } S.i1@ = T.i1@ \text{ then } S.i1@ < T.i1@ \text{ else} \\
\text{if } S.i2@ = T.i2@ \text{ then } S.i2@ < T.i2@ \text{ else} \\
\ (... \\
\text{if } S.iN@ = T.iN@ \text{ then } S.iN@ < T.iN@ \text{ else} \\
\text{false} 
\end{cases}$$
where "<" is the inequality operator.

With the use of the structure and union type constructors, recursive types can be specified. Such types may have infinite cardinality, but any instance or value of the type in GLADYS must be finite. Thus constructed recursive types must allow for finite values. This results in the mandatory use of the union constructor to allow a limit to the recursion. For example:

bind HEAD TAIL to interval(0,1)
bind NULL to interval(0,0)
bind CHAR to interval(0,127)
bind STRUCT1 to structure(HEAD TAIL; CHAR, STRUCT1)
bind STRUCT2 to union(NULL, TAIL)
bind TAIL to structure(HEAD_TAIL; CHAR, STRUCT2)

Here STRUCT1 is a type with only infinite values and is of no use in GLADYS whereas STRUCT2 has as values sequences of zero or more values of type CHAR. The single value of type NULL is the end-of-sequence marker which would correspond to an end-of-file or null pointer in some representations.

7.4 The Process Types.

A process type is described by a GLADYS statement and values of a process type correspond to the various possible stages of execution of the statement.

PROCESS_CREATOR = 'process' UNLABELLED_STATEMENT.

Each cell or instance of a process type may be in one of two states: stopped or running. If a process is stopped its value is constant and the process can not cause any changes to occur. If a process is running, its value changes spontaneously as it executes the statement in its definition. It may also cause the values of other cells to change, objects to be created and destroyed etc. as it executes. To start a process means to change its state from stopped to running and to stop a process means to change its state from running to stopped. When the unlabelled statement that defines the process is finished the process stops.

A GLADYS program must have at least one process in the running state; an implicit, un-named process with an un-named current basket starts execution of a GLADYS program.

A correspondence between the set of integers and the values of a process type may be set up as follows: the value of a process type before it starts execution of its statement corresponds to the integer value zero. Other correspondences may be defined in the process type by prefixing statements in the process definition with an integer
literal, when allowed by the syntax. These literals are called labels because of the obvious analogy with normal high-level languages. The values of a process type are not ordered in any way by the correspondence to integer values set up by labels. No values other than 0 and those defined by labels can be used to correspond to values of the process type.

\[ \text{LABELLED STATEMENT} = \text{LITERAL} '::' \text{UNLABELLED STATEMENT}. \]

Literal is defined in the section 8 and unlabelled statement is defined in section 12. The action of a process instance of assigning (assignment is described in section 9) a value to the same process instance is called a 'goto' operation, also as an analogy with normal high level languages. Values of the process type are obtained from integer values by the use of a cast as before:

\[ \text{PROCESS CAST} = \text{PROCESS TYPE NAME} '<' \text{INTEGER EXPRESSION} '>' . \]

\[ \text{PROCESS TYPE NAME} = \text{TYPE ELEMENT}. \]

where process type name must evaluate to a process type.

A process may assign a value to itself with the effect that the process will continue execution at the point specified by the value assigned. When a process assigns a value to another process the latter process must be stopped. The effect of such an assignment is merely to change the value of the stopped process and thus change the place from where it continues execution when it is started.

There are four GLADYS statements which apply to processes. They are:

1. The stop statement.

\[ \text{STOP STATEMENT} = 'stop' (\text{PROCESS EXPRESSION} / ',','). \]

\[ \text{PROCESS EXPRESSION} = \text{EXPRESSION}. \]

where process expression must evaluate to a reference to a process. This statement causes the specified processes to be stopped. If any of the processes is already stopped an error occurs.

2. The start statement.

\[ \text{START STATEMENT} = 'start' (\text{PROCESS EXPRESSION} / ',','). \]

This statement causes the specified processes to be started, that is, to continue execution from where they were stopped. If any of the processes is already running an error occurs.

3. The wait statement.
Appendix 3 - GLADYS

Wait statement = 'wait' (PROCESS_EXPRESSION / ',').

Execution of this statement causes the process to wait before executing the next statement until one of the specified processes is stopped. When this occurs the waiting process continues and the stopped process is immediately started. No other process is able to detect the stopping of the process waited on if it was initially running. If more than one process is waiting for the same process to stop, only one of them continues with the waiting process. Which one is left unspecified. If more than one of the processes being waited on by the same process stop simultaneously, only one of them (again unspecified) is started when the waiting process continues.

4. The transfer statement.

Transfer statement = 'transfer' (PROCESS_EXPRESSION / ',').

This statement stops the executing process and starts the specified processes. If any of the specified processes is already running an error occurs. It is guaranteed that all the started processes will perceive the transferring process as stopped unless it is explicitly started again.

Where a list of processes is specified in a process statement the action is carried out simultaneously on all processes. For example, for three processes A, B and C, where A and B are running, C is waiting on B and the next statement in C is "start A". An error will never occur if A executes "stop B, this_process".

PROCESS_STATEMENT = STOP_STATEMENT | START_STATEMENT | WAIT_STATEMENT | TRANSFER_STATEMENT.

There are also two functions which apply to processes. They are:

1. The stopped function.

Stopped function = 'stopped' '(( PROCESS_EXPRESSION ')').

This function returns one when the specified process is stopped and zero if it is running.

2. The running function.

Running function = 'running' '(( PROCESS_EXPRESSION ')').

This function returns one if the specified process is running and zero if it is stopped.

The primitive methods of specifying a process are:

PROCESS = PROCESS_Creator.
PROCESS_REF = 'this_process'.

'this_process' is bound to the process executing the expression in which it occurs.

Each process may have a basket associated with it called the current basket. The object denoted by an identifier is found by following the method described next or by any equivalent method. The identifier binding is searched for initially in the current basket. If it is not found in the current basket, the enclosing basket of the current basket is searched. Each enclosing basket in turn is searched until the identifier binding is found or until there are no more enclosing baskets. If the identifier binding is found, the object bound to that identifier is used as the object denoted by that identifier. If the identifier is not found or an identifier is used when there is no current basket, an error occurs. The search for an identifier binding will start in a specified basket, rather than the current basket, when the identifier is followed by a basket expression.

BINDING = IDENTIFIER ('in' BASKET_EXPRESSION)?.

A basket is made the current basket by using the current basket statement.

CURRENT_STATEMENT = 'current' BASKET_EXPRESSION.

When a process type cell is created and initialized the current basket for that process may be specified by following the process type expression with a slash followed by a basket expression which specifies the current basket for the newly created process.

PROCESS_INITIAL_VALUE = PROCESS_EXPRESSION ('7' BASKET_EXPRESSION)?.

It is an error to destroy a basket which contains any process type cells when any of them are running.
Data is stored in cells [17] of the constructed types. Values of data are the values of constructed types [19]. The type of values stored in a cell and the type of that cell must be identical [18].

An unspecified value (finite in size) of a particular type can be created by the cast construct as follows:

\[
\text{INDEFINITE\_VALUE} = \text{TYPE\_ELEMENT} \langle \? \rangle.
\]

Data value literals are built up from integer value literals using expression units and casts. In addition to the usual sign and decimal digits representation, the integers from 0 to 127 may be written using ASCII characters either by enclosing a printable character in single quotes or by using a reserved word.

\[
\text{LITERAL} = \text{\'-\?' DIGIT\+} | \text{\'\'\' PRINTABLE\_ASCII\_CHARACTER \'\'\'} | \text{ASCII\_SYMBOL}.
\]
9 ASSIGNMENT AND EXPRESSIONS.

Assignment is the method by which values of cells are changed.

ASSIGNMENT_STATEMENT = 'assign' EXPRESSION 'to' CELL_EXPRESSION.

where expression evaluates to a value of the same type as the cell referenced by the cell expression. The value is given to the cell referenced and the previous value of the cell lost. If the type of the cell is a recursive type the destination cell must have been created with the same structure as the value assigned to it.

The syntax for expressions is described now using the syntactic units defined in the previous sections.

EXPRESSION = VALUE_RELATION | OBJECT_RELATION | FORM.

FORM = INTEGER_CAST | TERM.

TERM = SELECTION | FACTOR.

FACTOR = DEREFERENCE | PRIMARY.

PRIMARY = TYPE_REFERENCE | ELEMENT.

ELEMENT = BASKET | CELL | TYPE | PROCESS | BINDING | EXPRESSION_UNIT | TYPE_FUNCTION | INDEFINITE_VALUE | 'index' | INTERVAL_CAST | UNION_CAST | MEMBER_CAST | PROCESS_CAST | INTEGER_FUNCTION | LITERAL | '(' EXPRESSION ')'.

INTEGER_FUNCTION = PROJECTION_FUNCTION | STOPPED_FUNCTION | RUNNING_FUNCTION | EXTERNAL_FUNCTION.

Integer function is a construct which evaluates to an integer value. External function is defined in section 11.

TYPE_REFERENCE = '!' TYPE_ELEMENT.

A type reference evaluates to a reference to the value of the previously defined type specified which most immediately contains the type reference. For example, if a type reference !X appears in a process type which is a constructing type of a structure type X, then !X evaluates to a reference to the structure type value containing that process.
10 CONTROL STRUCTURES.

The control structures in GLADYS are called units. There are three types of units: the sequential unit; the parallel unit; and the clonal unit. A unit may return a value in which case it must be preceded by a type primary, like casts, indicating the type of value it will return or the ']' symbol if it returns an integer value. A unit which returns a value is an expression unit as opposed to a unit which does not return a value which is a statement unit. An expression unit returns a value by using one or more expression statements in it. The value of an expression unit may be that of a single expression statement or, if the value of the unit is of a structured type, each element of the structure may be provided with a value by executing a number of different expression statements or by repeatedly executing one expression statement or a combination of both.

```
EXPRESSION_STATEMENT = INDEXING_EXPRESSION? '←' EXPRESSION.

INDEXING_EXPRESSION = EXPRESSION.
```

The expression after the left arrow symbol is evaluated to provide the value or part of the value of the unit in which it appears. If it is to form part of a structure value, the indexing expression before the left arrow must evaluate to a value of the indexing type of the structure type of the unit. The other expression must evaluate to a value of the constructing type associated with the value of the indexing expression in the structure type of the unit. All constructing type values must be specified exactly once by expression statements. If the type of the value the unit is to return is not a structure type then the expression must evaluate to a value of that type and there must be exactly one such expression statement, with no indexing expression, executed.

```
EXPRESSION_UNIT = SEQUENTIAL_EXPRESSION_UNIT | PARALLEL_EXPRESSION_UNIT | CLONAL_EXPRESSION_UNIT.

STATEMENT_UNIT = SEQUENTIAL_STATEMENT_UNIT | PARALLEL_STATEMENT_UNIT | CLONAL_STATEMENT_UNIT.
```

10.1 The Sequential Unit.

```
SEQUENTIAL_EXPRESSION_UNIT = (TYPE_ELEMENT | ']' )
SEQUENTIAL_STATEMENT_UNIT.

SEQUENTIAL_STATEMENT_UNIT = SERIAL_UNIT | LOOP_UNIT.

SERIAL_UNIT = '[ ' STATEMENT / 'next' ' ]'.
```
STATEMENT = UNLABELLED_STATEMENT | LABELLED_STATEMENT.

Execution of the serial statement unit normally proceeds from the first to the last statement in sequence, but since the statements may be labelled, execution may continue at any statement in the serial unit from anywhere from within the serial unit or from another part of the process by directly assigning a suitable value to the process. That is, a 'goto' operation could be performed into the serial unit.

LOOP_UNIT = "[" (PART ('exit' | 'loop' | 'next'))* 
PART 'loop'? ']".

PART = (INTEGER_EXPRESSION '→')? UNLABELLED_STATEMENT.

Execution of a loop unit starts with the first part and proceeds as follows:

If the part has an integer expression it is evaluated. If its value is not zero or one an error occurs. If its value is zero, the statement and the following keyword are ignored and execution continues with the next part in sequence. If its value is one or it is not present, the following statement is executed and the following keyword action taken. If the keyword is 'next', the next part in sequence is executed next, if the keyword is 'loop', execution begins again at the first part in the unit. If the keyword is 'exit', execution of the unit is complete.

The execution of the last part of a sequential unit differs slightly from the others because only the keyword 'loop' is allowed after the statement. Execution of the unit is complete when the integer expression in the last part, if present, evaluates to zero or when the statement in the last part is executed and it is not followed by the keyword 'loop'. The action of the last part is the same as all others otherwise. The last part is the only part which is allowed to not have a keyword following it.

The execution of the sequential unit is described again below using a flow chart. The Ei's stand for the integer expressions, the Si's for the statements and the Ki's for the keywords of a unit viz:-

[ E1 → S1 K1 E2 → S2 K2 ...... En → Sn Kn ]
Appendix 3 - GLADYS

START

\( i \leftarrow 1 \)

No

E_i present?

Yes

\( 1 \)

Value of E_i?

Other

execute S_i

error

Ki present?

No

Yes

\( i \leftarrow i + 1 \)

\( i \leftarrow i + 1 \)

next

Ki?

exit

FINISH

\( i \leftarrow 1 \)

loop

\( i \leftarrow n \)
10.2 The Parallel Unit.

Execution in parallel in GLADYS is the independent, concurrent execution of a number of statements. No assumptions can be made about the relative order of execution of any parts of the different statements executed in parallel. This implies, for example, that an infinite loop in one statement may cause the parallel statement in which it occurs never to complete even if another statement in the same parallel statement has a 'goto' operation out of the parallel statement. Also, a cell given different values by more than one statement (each statement using one assignment statement) in a parallel statement may have any of those values at the termination of the parallel statement.

PARALLEL_EXPRESSION_UNIT = (TYPE_ELEMENT | '#')
PARALLEL_STATEMENT_UNIT.

PARALLEL_STATEMENT_UNIT = ['PART 'WITH' PART / 'WITH' '].

Each integer expression of the parts is evaluated in parallel. Parts without integer expressions are treated as though they had an expression which evaluated to one. If any integer expression evaluates to a value other than zero or one an error occurs. The statements in the parts where the integer expression evaluates to zero are ignored, all others are executed in parallel. When all have completed, the execution of the parallel unit is complete. Labels cannot appear on statements in parallel units as such a label could not be associated with a unique value of the process in which the unit was contained. A 'goto' operation occurring inside a statement in a parallel unit causes all executing statements in that unit to cease at whatever point they have reached; execution continues at the point specified by the 'goto' operation.

10.3 The Clonal Unit.

CLONAL_EXPRESSION_UNIT = (TYPE_ELEMENT | '#')
CLONAL_STATEMENT_UNIT.

CLONAL_STATEMENT_UNIT = ['TYPE_ELEMENT 'copies'
UNLABELLED_STATEMENT '].

The type primary in the clonal statement unit is evaluated and the cardinality of the type thus obtained determined. This number of copies of the statement are created and are executed in parallel.

The special symbol 'index' has meaning only in the unlabelled statement inside a clonal unit. It is bound to a different value of the type obtained from the type primary in the clonal unit in each
copy of the unlabelled statement. Since there are the same number of copies of the unlabelled statement as values of the type, all values of the type are used by instances of 'index' in the copies of the unlabelled statement. Labels cannot appear on statements in the unlabelled statement within a clonal unit. A 'goto' operation in a statement in a clonal unit has the same effect as the same operation in a parallel unit.
11 PRIMITIVE OPERATIONS ON DATA.

Primitive data operations in GLADYS can only be carried out on integer values to produce integer values. This provides an unambiguous and well-founded basis for calculations. The actual operators or functions, called external functions, are not defined as part of GLADYS, but by the environment of GLADYS. This gives a well-defined interface between GLADYS and its environment. For a general purpose definition of a language these functions may be any well-defined mathematical functions on the integers. Some arithmetic and Boolean functions would be necessary for most languages. GLADYS specifies that the arguments to these functions are evaluated in parallel and that some argument values may cause an error to occur.

The normal functional notation is used, and the function identifiers must be declared at the beginning of the GLADYS program using the external function declaration.

EXTERNAL FUNCTION = EXTERNAL FUNCTION IDENTIFIER

N  INTEGER_EXPRESSION/ ', ') FN?

EXTERNAL FUNCTION DECLARATION = 'external'

IDENTIFIER FN(' DIGIT+ ') FN / ', '.

The external function identifier must be an identifier that has appeared in an external function declaration. The digit sequence in parentheses following an identifier in the external function declaration specifies the the number of parameters the function associated with that identifier has. External functions with no arguments must not have empty parentheses following the identifier in a call of the function.
Appendix 3 - GLADYS

12 GLADYS PROGRAM STRUCTURE.

A GLADYS program consists of an external function declaration followed by a statement, usually a serial unit. This statement is executed by an un-named process with an un-named current basket.

\[
\text{PROGRAM} = \text{EXTERNAL FUNCTION DECLARATION}'\text{next}' \text{UNLABELLED STATEMENT}.
\]

\[
\text{UNLABELLED STATEMENT} = \text{BINDING STATEMENT} \mid \text{ENCLOSING STATEMENT} \mid \text{DESTRUCTION STATEMENT} \mid \text{DESTRUCTION ENCLOSED STATEMENT} \mid \text{PROCESS STATEMENT} \mid \text{CURRENT STATEMENT} \mid \text{ASSIGNMENT STATEMENT} \mid \text{EXPRESSION STATEMENT} \mid \text{STATEMENT UNIT} \mid \text{ERROR STATEMENT} \mid \text{NULL STATEMENT}.
\]

\[
\text{ERROR STATEMENT} = '\text{error}'.
\]

\[
\text{NULL STATEMENT} = '\text{null}'.
\]

The error statement is used to cause an error and the null statement has no effect.
13 REFERENCES.


14 SYNTAX OF GLADYS.

PROGRAM = EXTERNAL_FUNCTION_DECLARATION 'next'
    UNLABELLED_STATEMENT.

EXTERNAL_FUNCTION_DECLARATION = 'external'
( IDENTIFIER \(' ' DIGIT+ .'\) / '.

UNLABELLED_STATEMENT = BINDING_STATEMENT | ENCLOSED_STATEMENT |
                     DESTROY_STATEMENT | DESTROY_ENCLOSED_STATEMENT |
                     PROCESS_STATEMENT | CURRENT_STATEMENT |
                     ASSIGNMENT_STATEMENT | EXPRESSION_STATEMENT |
                     STATEMENT_UNIT | ERROR_STATEMENT | NULL_STATEMENT.

BINDING_STATEMENT = 'bind' IDENTIFIER 'to' OBJECT_EXPRESSION
('in' PLACING_BASKET)?.

OBJECT_EXPRESSION = CELL_EXPRESSION | BASKET_EXPRESSION | TYPE_EXPRESSION.

CELL_EXPRESSION = EXPRESSION.

BASKET_EXPRESSION = EXPRESSION.

TYPE_EXPRESSION = EXPRESSION.

PLACING_BASKET = BASKET_EXPRESSION.

ENCLOSING_STATEMENT = 'enclosing' ENCLOSED_BASKET_EXPRESSION
                     'is' ENCLOSED_BASKET_EXPRESSION.

ENCLOSED_BASKET_EXPRESSION = BASKET_EXPRESSION.

ENCLOSING_BASKET_EXPRESSION = BASKET_EXPRESSION.

DESTROY_STATEMENT = 'destroy' BASKET_EXPRESSION.

DESTROY_ENCLOSED_STATEMENT = 'destroy_enclosed'
                           BASKET_EXPRESSION.

PROCESS_STATEMENT = STOP_STATEMENT | START_STATEMENT |
                   WAIT_STATEMENT | TRANSFER_STATEMENT.

STOP_STATEMENT = 'stop' (PROCESS_EXPRESSION / '.').

PROCESS_EXPRESSION = EXPRESSION.

START_STATEMENT = 'start' (PROCESS_EXPRESSION / '.').

WAIT_STATEMENT = 'wait' (PROCESS_EXPRESSION / '.').

TRANSFER_STATEMENT = 'transfer' (PROCESS_EXPRESSION / '.').
CURRENT_STATEMENT = 'current' BASKET_EXPRESSION.

ASSIGNMENT_STATEMENT = 'assign' EXPRESSION 'to' CELL_EXPRESSION.

EXPRESSION_STATEMENT = INDEXING_EXPRESSION? '←' EXPRESSION.

INDEXING_EXPRESSION = EXPRESSION.

STATEMENT_UNIT = SEQUENTIAL_STATEMENT_UNIT | PARALLEL_STATEMENT_UNIT | CLONAL_STATEMENT_UNIT.

SEQUENTIAL_STATEMENT_UNIT = SERIAL_UNIT | LOOP_UNIT.

SERIAL_UNIT = '][' STATEMENT / 'next' ']'.

STATEMENT = UNLABELLED_STATEMENT | LABELLED_STATEMENT.

LABELLED_STATEMENT = LITERAL ':' UNLABELLED_STATEMENT.

LITERAL = '-'? DIGIT+ |  '"""" PRINTABLE_ASCII_CHARACTER '"""" | ASCII_SYMBOL.

LOOP_UNIT = '[ (PART ('exit' | 'loop' | 'next'))* PART 'loop'? ']''.

PART = (INTEGER_EXPRESSION '→')? UNLABELLED_STATEMENT.

INTEGER_EXPRESSION = EXPRESSION.

PARALLEL_STATEMENT_UNIT = '][' PART 'with' PART / 'with' ']''.

CLONAL_STATEMENT_UNIT = '][' TYPE_ELEMENT 'copies' UNLABELLED_STATEMENT ']''.

TYPE_ELEMENT = ELEMENT.

ERROR_STATEMENT = 'error'.

NULL_STATEMENT = 'null'.

EXPRESSION = VALUE_RELATION | OBJECT_RELATION | FORM.

VALUE_RELATION = VALUE_FORM RELATIONAL_OPERATOR VALUE_FORM.

VALUE_FORM = FORM.

RELATIONAL_OPERATOR = '=' | '~'= | '<=' | '>=' | '<' | '>'.

OBJECT_RELATION = OBJECT_FORM ('=' | '~=') OBJECT_FORM.

OBJECT_FORM = FORM.
FORM = INTEGER_CAST | TERM.

INTEGER_CAST = '# INTERVALTERM.

INTERVALTERM = TERM.

TERM = SELECTION | FACTOR.

SELECTION = STRUCTURETERM '.' INDEXINGPRIMARY.

STRUCTURETERM = TERM.

INDEXINGPRIMARY = PRIMARY.

FACTOR = DEREFERENCE | PRIMARY.

DEREFERENCE = REFERENCETERM '@'.

REFERENCETERM = TERM.

PRIMARY = TYPEREFERENCE | ELEMENT.

TYPEREFERENCE = '@' TYPEELEMENT.

ELEMENT = BASKET | CELL | TYPE | PROCESS | BINDING | EXPRESSION UNIT | TYPEFUNCTION | INDEFINITENUMBER | 'index' | INTERVALCAST | UNIONCAST | MEMBERCAST | PROCESSCAST | INTEGERFUNCTION | LITERAL | '(', EXPRESSION ').'

BASKET = 'no_basket' | 'current_basket' | BASKETCREATOR.

BASKETCREATOR = 'create_basket'.

CELL = 'no_cell' | CELLCREATOR | PROCESSREF.

CELLCREATOR = 'create_cell' '(' TYPE EXPRESSION ',' INITIALVALUE ',', ' PLACING_BASKET)? ')'.

INITIALVALUE = EXPRESSION | PROCESSINITIALVALUE.

PROCESSINITIALVALUE = PROCESSEXPRESSION ('?' BASKET_EXPRESSION)?.

PROCESSREF = 'this_process'.

TYPE = 'no_type' | TYPECREATOR.

TYPECREATOR = OBJECTTYPE | CONSTRUCTEDTYPE.

OBJECTTYPE = 'basket' | 'cell' | 'type'.
CONSTUCTED_TYPE = INTERVAL_TYPE | UNION_TYPE | STRUCTURE_TYPE | NULL_TYPE.

INTERVAL_TYPE = 'interval' '(' LOWER_BOUND ',' UPPER_BOUND ')'.

LOWER_BOUND = INTEGER_EXPRESSION.

UPPER_BOUND = INTEGER_EXPRESSION.

UNION_TYPE = 'union' '(' MEMBER_TYPE_EXPRESSION / ',' ')'.

MEMBER_TYPE_EXPRESSION = TYPE_EXPRESSION.

STRUCTURE_TYPE = 'structure' '(' INDEXING_TYPE_EXPRESSION / CONSTRUCTING_TYPE_EXPRESSION / ',' ')'.

INDEXING_TYPE_EXPRESSION = TYPE_EXPRESSION.

CONSTRUCTING_TYPE_EXPRESSION = TYPE_EXPRESSION.

NULL_TYPE = 'null_type'.

PROCESS = PROCESS_CREATOR.

PROCESS_CREATOR = 'process' UNLABELLED_STATEMENT.

BINDING = IDENTIFIER ('in' BASKET_EXPRESSION)?.

EXPRESSION_UNIT = SEQUENTIAL_EXPRESSION_UNIT | PARALLEL_EXPRESSION_UNIT | CLONAL_EXPRESSION_UNIT.

SEQUENTIAL_EXPRESSION_UNIT = (TYPE_ELEMENT | '#') SEQUENTIAL_STATEMENT_UNIT.

PARALLEL_EXPRESSION_UNIT = (TYPE_ELEMENT | '#') PARALLEL_STATEMENT_UNIT.

CLONAL_EXPRESSION_UNIT = (TYPE_ELEMENT | '#') CLONAL_STATEMENT_UNIT.

TYPE_FUNCTION = 'type_of' '(' EXPRESSION ')'.

INDEFINITE_VALUE = TYPE_ELEMENT '<' '?' '>'.

INTERVAL_CAST = INTERVAL_TYPE_NAME '<' INTEGER_EXPRESSION '>'.

INTERVAL_TYPE_NAME = TYPE_ELEMENT.

UNION_CAST = UNION_TYPE_NAME '<' MEMBER_EXPRESSION '>'.

UNION_TYPE_NAME = TYPE_ELEMENT.
MEMBER_EXPRESSION = EXPRESSION.
MEMBER_CAST = MEMBER_TYPE_NAME ' < ' UNION_EXPRESSION ' > '.
MEMBER_TYPE_NAME = TYPE_ELEMENT.
UNION_EXPRESSION = EXPRESSION.
PROCESS_CAST = PROCESS_TYPE_NAME ' < ' INTEGER_EXPRESSION ' > '.
PROCESS_TYPE_NAME = TYPE_ELEMENT.
INTEGER_FUNCTION = PROJECTION_FUNCTION | STOPPED_FUNCTION | RUNNING_FUNCTION | EXTERNAL_FUNCTION.
PROJECTION_FUNCTION = MEMBER_TYPE_NAME '( ' UNION_EXPRESSION ' ) '.
STOPPED_FUNCTION = 'stopped' '( ' PROCESS_EXPRESSION ' ) '.
RUNNING_FUNCTION = 'running' '( ' PROCESS_EXPRESSION ' ) '.
EXTERNAL_FUNCTION = EXTERNAL_FUNCTION IDENTIFIER ( ' ( ' INTEGER_EXPRESSION ' ) ' )?.
EXTERNAL_FUNCTION IDENTIFIER = IDENTIFIER.
IDENTIFIER = LETTER ( LETTER | DIGIT | ID_SPECIAL )*.
LETTER = 'A' | 'B' | .... | 'Z'.
DIGIT = '0' | '1' | .... | '9'.
ID_SPECIAL = '_ ' | '$ ' | '%' | '& ' | ' '.
PRINTABLE_ASCII_CHARACTER = LETTER | DIGIT | ID_SPECIAL |
ASCII_SYMBOL = 'nul' | 'soh' | 'stx' | 'etx' | 'eot' | 'enq' | 'ack' | 'bel' | 'bs' | 'tab' | 'lf' | 'vt' | 'ff' | 'cr' | 'so' | 'si' | 'dle' | 'dc1' | 'dc2' | 'dc3' | 'dc4' | 'nak' | 'syn' | 'etb' | 'can' | 'em' | 'sub' | 'esc' | 'fs' | 'ga' | 'rs' | 'us' | 'del'.

Appendix 3 - GLADYS

225
Appendix 3 - GLADYS

SPECIAL SYMBOL

ASCII SYMBOL

'assign' | 'basket' | 'bind' | 'call' | 'cell' |
'copies' | 'create_basket' | 'create_cell' | 'current' |
'current_basket' | 'destroy' | 'destroy_enclosed' |
'enclosing' | 'error' | 'exit' | 'external' |
'in' | 'index' | 'interval' | 'is' | 'loop' |
'next' |
'no_basket' | 'no_cell' | 'no_type' | 'null' |
'null_type' | 'process' | 'running' | 'start' |
'stop' |
'stopped' | 'structure' | 'this_process' |
'to' |
'transfer' | 'type' | 'type_of' | 'union' | 'wait' | 'with'.

...
APPENDIX FOUR

APPENDIX FOUR

Department of Computing Science
Australian National University

Advantages such as portability and code compaction are claimed for the intermediate language while the approach is running high-level language programs. This approach consists of compiling the source language to the intermediate language and an interpreter to interpret the intermediate language. It is proposed that given a suitable description of the high-level language and the target machine, such a compiler and interpreter can be produced automatically. The method proposed for describing the high-level language is outlined and some unresolved problems are discussed.
Advantages such as portability and code compactness are claimed for the intermediate language machine approach in running high-level language programs. This approach requires a compiler to translate the source language to the intermediate language and an interpreter to interpret the intermediate language. It is proposed that given a suitable description of the high-level language and the target machine such a compiler and interpreter can be produced automatically. The method proposed for describing the high-level language is outlined and some unresolved problems are discussed.
1 INTRODUCTION

One approach to the problem of designing computer architectures oriented towards the efficient execution of high-level languages has been the intermediate language machine [5, 9]. This approach is characterized by the definition of an intermediate language that lies between the high-level language L and the machine language M. Notable examples of this type of architecture are found in [1, 18, 19]. Following Wilner [18], we shall identify the intermediate language itself as S.

Systems based upon the intermediate language machine (ILM) approach require two components for the execution of high-level languages. Firstly, a compiler which translates programs written in L to programs written in S, and secondly, an interpreter written in M which interprets S-code. Strictly, L, S, and M define languages, and are distinguished from the machines which execute programs written in those languages. However, it is convenient to refer to these (perhaps virtual) machines also as L, S, and M. Using the notation of Rosin [14], we may represent this diagrammatically as:

![Diagram of the Intermediate Language Machine](image)

The Intermediate Language Machine

Figure 1

This can be compared with the normal approach where L is translated (compiled) directly to M and executed:
As the automatic generation of compilers has to some extent been successful, it is appropriate to explore the feasibility of automatically generating both the compiler and interpreter required for the ILM approach. The big difference is, of course, the creation of the intermediate language S. We will start with the idealistic assumption that all the input required by such a generating system are the definitions of L and M. The proposed system has been named SAGA, System for Automatic Generation of Architectures.

2 OUTLINE OF SAGA

The complete general picture of the way in which SAGA is to be used is given in figure 3, again using Rosin's notation.

SAGA takes the definitions of L and M and produces a compiler and an interpreter. The compiler runs on some unspecified machine (possibly virtual) and translates the user program written in L to a object program written in S. The user program is then interpreted on the virtual S machine formed by the interpreter running on machine M. The user program takes input I and produces output O.

In general, the compiler produced by SAGA could be written in M or S or some other convenient language. In the first case the compiling machine would be drawn as in figure 4a.
Appendix 4 - Proposed Intermediate M/c Generating System

Use of SAGA
Figure 3
Appendix 4 - Proposed Intermediate M/c Generating System

If neither M nor S are suitable languages in which to produce the compiler, the "other convenient language" needs to be defined to SAGA. We shall show that this problem can be side-stepped in the next section.

3 SPECIFICATION OF L AND GENERATION OF THE COMPILER

SAGA requires a complete specification of the language L. Methods for defining the syntax of a programming language are well understood and form a large part of any compiler generating system. Since SAGA will be using the definition of the syntax for exactly the same purpose we are assured of finding a suitable method.

The definition of the semantics of programming language, though, is a far from settled issue. Here we must look closely at our requirements and make a decision based on intuition. The method we choose must:

(a) be written by a human,
(b) be read by a machine,
(c) be able to describe concepts used, and
(d) impose as little structure as possible.

Requirements (a), (b) and (c) are straightforward. Requirement (d) results from the fact that we are creating S and we would like its structure to be determined by the language L alone and not on the method of defining L's semantics or any other extraneous influence. This ideal may not be practical, in particular, the target machine, M,
may affect S to some extent for efficiency reasons, but we shall ignore this for the time being. Methods which use intermediate or low level languages in their definition of L are to be avoided because of the implicit structure imposed by them as in VDL [12] and SEMANOL [2].

In SAGA the specification of L is performed by two languages GLADYS and SYNTRAN. SYNTRAN is used to specify the translation between L and GLADYS, thus defining L in terms of GLADYS. The advantage of this is that (1) GLADYS can be a higher level language and impose less structure than otherwise, in particular, the level of GLADYS can be of the same order as L, (2) S can, in a natural way, be defined in terms of GLADYS, and (3) the program translating L to GLADYS written in SYNTRAN can be used to specify the compiler by substituting the appropriate S constructs for the corresponding GLADYS constructs in the translating program. This results in a program translating L to S written in SYNTRAN. Thus by interpreting or compiling SYNTRAN the compiler is implemented.

\[
\begin{array}{c}
L \rightarrow \text{GLADYS} \\
\text{SYNTRAN}
\end{array}
\quad \text{to} \quad 
\begin{array}{c}
L \rightarrow S \\
\text{SYNTRAN}
\end{array}
\]

Implementation of the compiler
Figure 5

Denotational semantics [15] and W-grammars [17] were rejected as methods for defining the semantics of L because of their primitive bases. This is an advantage if the aim is "merely" to define L, but we require more. GLADYS uses concepts distilled from high-level languages and this allows the definition to be more direct and easier to use from the points of view of both the person writing the definition and from SAGA.

4 SYNTRAN

The central concept and program unit of SYNTRAN is the "translator". A translator consists of a pattern which is used to match the input stream and a production to produce an output stream. This allows the complete definition of a language construct to be in one place. The notation used for specifying the patterns is very similar to that commonly used for specifying context free grammars of programming languages. The notation for specifying the character strings produced in the output stream are also very similar. This allows the context free syntax of both the source language as a whole and the corresponding target language constructs of a given SYNTRAN program to be determined quite simply. A simple example is given to
show how this is done:

Example 1.

```
translator IDLIST
<(' ~ ID_TABLE = ID/',' ~ ')'>
<control NUMBER_OF_IDS integer #ID_TABLE>
[' ~
<NUMBER_OF_IDS(INDEX) = > ID_TABLE.INDEX/'; > ~'
'] > endtranslator
```

Example 1 is a translator which translates a list of one or more occurrences of the syntactic entity ID (not defined here) separated by commas and enclosed in parentheses to the same list of ID's separated by semi-colons and enclosed in square brackets. That is, for example, (FRED,JIM,BERT) to [FRED;JIM;BERT]. The pattern is the first construct enclosed in angle brackets and the production is the second.

The syntax of the expected input stream can be derived from the pattern in a trivial manner and written in the form:

```
IDLIST = '( 'ID/',' ')'.
```

which is a familiar form for expressing syntax where the "/" denotes the listing operator. The SYNTTRAN pattern uses tildes for concatenation instead of juxtaposition (for ease of implementation) and uses the left arrow symbol, '←', as a special assignment operator which results in the variable ID_TABLE being an array whose elements selected by the values 1, 2, 3... etc. are the first, second and third etc. ID's matched in the input stream.

The syntax of the output stream is not quite as trivial. The enclosing square brackets are clear, the listing operator with a semi-colon right argument is also clear, but the syntax of the object ID_TABLE.INDEX is not immediately obvious. The dot is used as the selection operator and by inspecting the pattern we can see that ID_TABLE is an array whose elements belong to the syntactic category named ID. Thus the value ID_TABLE.INDEX must conform to the syntax of an ID. Now, we can write:

```
IDLIST = '[ 'ID/'; ' ']'.
```

as the syntactic definition of IDLIST in the target language. All the constructs allowed by SYNTTRAN in its patterns and productions allow a similar simple analysis to determine the context free syntax of the input and output streams.

Given that one is familiar with this notation for expressing context free syntax and because both source and target language syntax can be expressed readily in this notation, it is clearly advantageous to use the notational constructs of alternation, concatenation, listing, repetition etc. to specify both the input and output streams.
The listing operator is especially useful in this regard.

The parts of example 1 not yet explained deal with the detailed control of the production. NUMBER_OF_IDS is defined to have the value the number of elements of ID_TABLE. This is used to determine the number of times the left argument of the listing operator in the production is to be repeated. INDEX is defined to have the values 1, 2, 3 ... etc. up to the value of NUMBER_OF_IDS so that each element of ID_TABLE can be referenced in order to produce the output required.

The context free syntax of the target language constructs can be used to partition the semantics of language L into operations suitable for the intermediate language, S. Then, as mentioned above, by replacing the productions in each translation in the SYNTRAN program with the suitable combinations of S-operations derived by SAGA, the compiler for L is specified. To see how this applies in practice, consider the following definition of a while-statement construct.

Example 2.

```
translator WHILE_STATEMENT
< 'while' ~ B ← EXP('Boolean':length_of_B) ~
  'do' ~ S ← STATEMENT(:length_of_S) ~> 
< [emission L1 is LABEL(GENERATE_LABEL);
  emission L2 is LABEL(GENERATE_LABEL)]
  'LOCATE' ~ L1 ~
  B ~
  'BRANCH_IF_FALSE' ~ L2 ~
  S ~
  'UNCONDITIONAL_BRANCH' ~ L1 ~
  'LOCATE' ~ L2 endtranslator
```

In example 2 we have used a low level target language, which has operations LOCATE, BRANCH_IF_FALSE and UNCONDITIONAL_BRANCH, to describe the semantics of a while-statement. Simple analysis shows that to implement the while-statement, S could have two operations: a branch forward conditional on a calculated value, thereafter discarded, being "false" (BFF, say) and an unconditional branch backward, (UBB, say). The production in example 2 could then be replaced with something like:

```
< B ~
  'BFF' ~ length_of_S+2 ~
  S ~
  'UBB' ~ length_of_B+length_of_S+1 ~> 
```

where length_of_S and length_of_B are parameters passed back from EXP and STATEMENT giving the respective lengths of the generated code sequences B and S.

SYNTRAN also has the normal control constructs of a general purpose programming language: loops, conditionals, procedures etc. The
data structures are simple and flexible and allow the all important data structure of most translators, the symbol table, to be handled effectively.

5 GLADYS

Let us consider the S-operations derived from example 2, BFF and UBB. Any other combination of operations that would implement a while-statement are precluded unless some much more difficult analysis is done. This is because of the low level of the target language. For example, in a more complicated environment where the statement inside the while-statement is located in a separate code segment, the while-statement could be implemented using one S-operator (WL, say) with the semantics:

- if the expression given evaluates to "true", execute the code in the segment ... and return to repeat this operation,
- otherwise continue execution with the next operation in this code segment.

This is clearly a higher level operation than BFF and UBB.

It would be far easier to analyse a high-level target language to produce the single operation, WL, than to analyse a low level target language, such as the one used in example 2, to produce the same single operation. Conversely it would be reasonable to expect that the lower level operations BFF and UBB would also be derivable easily from a high-level target language. This is a common phenomenon; it is far easier to translate from a higher level language to a lower level language efficiently than it is to translate from a lower level to a higher level. This is another aspect of the reason given in section 3 for GLADYS to have a high-level language flavour.

To express the semantics of the while-statement of example 2 in GLADYS the production can be changed to:

\[
< ' [ ' - B - ' : ' - S - ' ] loop >
\]

which is a fairly high-level construct and does not preclude either combination of S-operations described above.

It is expected that the person writing the description of L will do so in as abstract a way as possible. To do otherwise may cause the created S language to be constrained in an undesirable way. Consider the case where the while-statement is described in GLADYS in a way resembling the low level language used in example 2: the single S-operation, WL, would again be precluded.

The other important concepts of GLADYS are: names, cells, types, scopes and processes.

Names correspond to variable names, type names and references (or pointers) in normal high-level languages. In GLADYS, names are also
used to denote scopes. All names are declared by the same GLADYS "name" construct and during their existence denote exactly one object: a cell, a type or a scope.

Cells correspond to variables and are the elements for storing all data in GLADYS. Types define a set of values that a cell may take. Typing is strict and no automatic conversions occur. Primitive types are names, processes and finite, contiguous subsets of the integers called intervals. These may be combined using the union and structure constructors. The union constructor results in a discriminated union type \([s]\), while the structure types are used to describe records, arrays, sets and sequences found in normal high-level languages. Heavy reliance is placed on the usage of the structure types to determine an efficient physical representation for them. It has been shown \([11]\) that the automatic choice of physical representations for various data structures using dynamic and static factors about their usage is feasible. The method of choosing static representations required here can be drawn from that study.

The operations allowed on constructed types are only assignment, discrimination of unions and selection of components of structures. Names can only be "de-referenced", leaving the intervals to provide the basis for all calculations. Thus the set of integers provides GLADYS with a safe and well understood foundation for calculations.

Processes in GLADYS correspond to procedures, co-routines, tasks etc. of normal high-level languages but their definition is not yet finalized.

Scopes are explicitly handled in GLADYS to allow the description of details of storage management at a fairly high level. Scopes are sets of names and cells with the restriction that at most one scope name can be in any one scope. The scope thus named is called the enclosing scope and with it, nesting of scopes can easily be achieved. All names and cells are placed into a scope upon creation and remain there until the scope is destroyed. Names are unique within one scope but duplicate names are allowed in different scopes as is usual in normal high-level language scoping rules.

To illustrate these concepts consider a trivial ALGOL 60 program and its GLADYS equivalent:

Example 3

ALGOL 60

begin
    integer I;
    begin
        integer J;
        J:=I+1
    end
end
GLADYS (The statements are numbered only for reference)

(1) `scope_name(GLOBAL);`
(2) `current_scope GLOBAL;`
(3) `type_name(INTEGER, GLOBAL, interval(-32768,32767));`
(4) `cell_name(I, GLOBAL, cell(INTEGER, GLOBAL, ?));`
(5) `scope_name(LEVEL1, GLOBAL);`
(6) `current_scope LEVEL1;`
(7) `cell_name(J, LEVEL1, cell(INTEGER, LEVEL1, ?));`
(8) `assign (J, convert (convert (I@, INTEGER, #) + 1, #, INTEGER));`
(9) `destroy LEVEL1;`
(10) `destroy GLOBAL;`

The meaning of each of these statements is:

(1) declare a scope called GLOBAL.

(2) makes GLOBAL the current scope, i.e. name searches start in this scope.

(3) the type INTEGER is declared and put into the GLOBAL scope. Values of type integer are isomorphic to the subset of the integers \{-32768, -32767, \ldots, 32766, 32767\}. The exact limits are chosen in this case to make efficient use of particular hardware. Normal high-level languages do not specify details for things such as "integer" data types because it is assumed that in any implementation they will be determined by the hardware. GLADYS does not make concessions to such matters and requires all types to be explicitly defined.

(4) declares a cell named I of type INTEGER with an unknown (but legal) initial value. Both the name I and the cell are placed in the GLOBAL scope.

(5) a new scope, LEVEL1, is declared enclosed by the scope GLOBAL.

(6) LEVEL1 is made the current scope.

(7) declares a cell named J of type INTEGER with an unknown initial value. Both the name J and the cell are placed in the scope LEVEL1.

(8) the name I, not found in the current scope, LEVEL1, is searched for in LEVEL1's enclosing scope GLOBAL where it is found. The name I is "de-referenced" to get the value of the cell that it names (which is unknown). This value is converted from the type INTEGER to its corresponding integer value. One is added to this value; the operation of addition is the one normally defined on the mathematical set of integers. The result is then converted to type INTEGER, if possible. The value may not be in the range -32768 to 32767 (specifically, it could be 32768), in which case the effect of the
program is left undefined by the GLADYS description. Assuming the result is in range it is given to the cell named J, which is found in the current scope, LEVEL1. Note that the cell named J is of type INTEGER and the value placed in it is of type INTEGER. This must be the case or the effect of the program is left undefined as before.

(9) the scope LEVEL1 is destroyed along with everything in it, that is, the cell named J and the name J itself. The current scope becomes the enclosing scope of LEVEL1 which is GLOBAL.

(10) the scope GLOBAL and everything in it is destroyed.

GLADYS statements (1), (2) and (3) would form part of a "standard prelude" to an ALGOL 60 program and (10) a "standard postlude".

6 SPECIFICATION OF M AND GENERATION OF THE INTERPRETER

The state of methods defining computers at the level we require (register transfer level) is a bit better than that for defining programming language semantics. There are a large number of Computer Hardware Description Languages (CHDLs) that treat the Register Transfer level, for example, ISP [3], APL*DS [6], CDL [4], CSL [16] and OSM [13]. Most of these have been used for computer simulation as well as purely descriptive purposes.

Once the intermediate language has been defined machine code sequences can be produced automatically to perform the execute part of the interpreter fetch/execute cycle. Two practical systems for automatic generation of code generators are given in [7] and [10]. Both these systems accept input in the form of intermediate language code segments and produces corresponding machine code segments and with some adaption the approaches used therein can be applied in SAGA. For simplicity, though, the initial aim is to be less general and produce interpreters only for the Burroughs' B1700 machines [18] since they were designed for interpreting intermediate languages. This removes the requirement for the description of M temporarily.

The design of the fetch part of the fetch/execute cycle determines the S machine instruction and addressing formats. The architecture of the target machine will clearly affect these formats as well as the data formats chosen. The decisions that need to be made here may be more difficult with the B1700 as a target machine precisely because of its flexibility, normal architectures leave little choice in this area.

7 GENERATION OF THE INTERMEDIATE LANGUAGE S

The description of S is not output from SAGA explicitly but is implied in the compiler and interpreter produced. Not only must SAGA recognize the myriad possibilities for the structure of S, it must
choose one, preferably the "best" one. SYNTRAN and GLADYS have been designed with the determination of the bounds of possibilities for a language L in mind. For example, if we have a unit in an L which is translated into a sequence of GLADYS statements matching something like this:

```
UNIT = 'scope_name(' ID1 ', ' ID2 ');'
    'current_scope' ID1 ';
    .
    .
    .
    'destroy' ID1 ';'
```

then there is a strong indication that a stack discipline of storage allocation would be applicable, if not sufficient. Sufficiency could be determined by checking all cells created inside the unit. If they are all placed in scope ID1 then the stack discipline is sufficient otherwise it is not.

Doing this sort of analysis automatically is the essence of SAGA and it has been a design goal of GLADYS and SYNTRAN to make this analysis easy for as broad a spectrum of languages as possible. With this in mind it is hoped that the computation required to do the analysis is not prohibitively complex or expensive.

The choice of one of the possible structures of S is currently left to the skill and subjective judgement of a human programmer and is the nub of the research required for the successful implementation of SAGA. It must be noted that even at this relatively high level, decisions as to the structure of S may be strongly affected by M if only for efficiency reasons. There is an advantage to be gained by not allowing M to affect S and that is portability of S [1]. It is perhaps ironic to observe that if SAGA becomes a fully operational and efficient system, portability will no longer be an issue! One would need only to feed in a different M description to SAGA to produce a compiler and interpreter for a different machine.

8 CURRENT POSITION

At the time of writing, the design of SYNTRAN seems to be stable and GLADYS is almost complete. Work is progressing on how the structure of L can be extracted from the GLADYS/SYNTRAN description and GLADYS is still being optimized for this purpose. An implementation of SYNTRAN is being developed and consists of a SNOBOL4 program that will translate SYNTRAN to SNOBOL4. This, however, will not form part of SAGA directly, it will be used for checking descriptions of languages and for "fine tuning" SYNTRAN and GLADYS.
In order to provide some experience in the use of a system such as SAGA, the initial implementation will use an implicit M description, and the S languages produced will not necessarily be optimal.
REFERENCES


This appendix gives a SYTRAN/GLADYS description of a small language called ASPLE which is used in [MarcM76] where a number of formal definition methods are described. The symbol '#' is used for ASPLE's not equals symbol. The identifiers n1, n2, n3, n4 and n5 represent, respectively, the maximum length of an ASPLE program, the maximum number of declared identifiers, the maximum number of digits in an integer constant, the maximum number of letters in an identifier and the maximum value that can be contained in an integer variable.
Appendix 5 - Definition of ASPLE

set LETTER < 'A' .. 'Z'>,
    DIGIT < '0' .. '9'>,
    SPEOL < ' ', eol >;

token IDE < LETTER+ >,
    NUMBER < DIGIT+ >;

ignore SPEOL* endignore;

global ST;

emission MAXINT is NUMBER: n5;

translator PROGRAM distinguished
pattern
    'begin' " D ← DCL_TRAIN " S ← STM_TRAIN " 'end'
endpattern
production
    [ if #(D " S) > n1 then fail endif ]
    'external NOT(1), EQUAL(2), ADD(2), MULTIPLY(2), OR(2), AND(2),'
    'READ_INT(0), READ_BOOL(0), WRITE_INT(1), WRITE_BOOL(1) next'
    'bind GLOBAL SCOPE to current basket next'
    'bind BOOLEAN_TYPE to interval(0, 1) next'
    'bind INTEGER_TYPE to interval(0, MAXINT) next'
    'bind UNDEF_BOOL to interval(0, O) next'
    'bind UNDEF_INT to interval(0, 0) next'
    'bind INT_MODE to union(INTEGER_TYPE, UNDEF_INT) next'
    'bind BOOL_MODE to union(BOOLEAN_TYPE, UNDEF_BOOL) next'
    'bind TEMP_REF to create_cell(cell, no_cell) next'
    D " 'next' " S
endproduction
endtranslator;

translator DCL_TRAIN
pattern < D ← DECLARATION '"'; '> endpattern
production
    [ _control NDS integer #D ]
    [ ' ' < NDS(I) → D.I / 'with' > '"' ]
endproduction
endtranslator;
translator DECLARATION
local NREFS, ENTRY use V
pattern
  REFS ← 'ref' * TYPE ← < 'int' | 'bool' >
  IDS ← ID / ','
endpattern
production
  [ control NIDS integer #IDS;
    assign #REFS + 1 to NREFS;
    control PNTR boolean NREFS > 1;
    control INT_MODE boolean TYPE = 'int'
  ]
< NIDS(I) →
  [ assign ST.(IDS.I) to ENTRY;
    if ENTRY = null & #ST <= n2
      then
        assign NREFS to ENTRY.'REFS';
        assign TYPE to ENTRY.'TYPE'
      else fail
    endif
  ]
< 'bind' ^ IDS.I 'to create cell(' ^
< PNTR → 'cell, no_cell' |-
  < INT_MODE →
    'INT_MODE, INT_MODE<UNDEF_INT<0>>' |
    'BOOL_MODE, BOOL_MODE<UNDEF_BOOL<0>>' />
    ')' > / 'with' >
endproduction
endtranslator;

translator STM_TRAIN
pattern
  S ← < ASG_STM | COND_STM | LOOP_STM | IN_STM | OUT_STM >
  /':'
endpattern
production
  [ control NSS integer #S ]
< NSS(I) → S.I / 'next' >
endproduction
endtranslator;

translator ID
pattern I ← IDE endpattern
production
  [ if I > n4 then fail endif ]
I
endproduction
endtranslator;
translator ASG_STM
pattern
I ← ID " := '" E ← EXP(: REFS, TYPE)
endpattern
production
[ if ST.I " = null
then
  control DEREFS integer REFS + 1 - ST.I."REFS';
  if DEREFS >= 0 & ST.I."TYPE" = TYPE
then
  control INT_MODE boolean TYPE = 'int';
  control ASG_REF boolean REFS > DEREFS
else fail
endif
else fail
endif ]
'assign'
< ASG_REF→
< 'cell'
  'assign' " E " < DEREFS → '@*' > ^
  'to TEMP_REF next' ^
  '(no_cell = TEMP_REF@) → error next' ^
  '← TEMP_REF@]' > |
< < INT_MODE→ 'INT_MODE< ' | 'BOOL_MODE< ' > ^
DEREF(DEREFS, TYPE: E) "' >'
> ^
'to' ^ I
endproduction
endtranslator;

macro DEREF(REFS, TYPE: EXP)
production
[ control SIMPLE boolean REFS = 0 ]
< SIMPLE →
EXP |[
[ control INT_MODE boolean TYPE = 'int';
control DEREFS integer REFS ]
< < INT_MODE→ 'INTEGER' | 'BOOLEAN' > ^
'– TYPE< ' " EXP " < DEREFS → '@*' > ^ ' >'
> ^
endproduction
endmacro;
translator COND_STM

pattern 'if' ^ E ← EXP(: REFS, TYPE) ^ 'then' ^ TB ← STM_TRAIN ^
< 'else' ^ FB ← STM_TRAIN >? ^ 'fi'
endpattern
production
[ if TYPE = 'bool'
then control ELSE_PART boolean FB = null
else fail
endif ]
'[: ' DEREF(REFS, 'bool': E) ']' ^
< ELSE_PART → < 'exit' ^ FB >? > ^ ']''.
endproduction
endtranslator;

translator LOOP_STM

pattern 'while' ^ E ← EXP(: REFS, TYPE) ^ 'do' ^
S ← STM_TRAIN ^ 'end'
endpattern
production
[ if TYPE = 'bool' then fail endif ]
'[: ' DEREF(REFS, 'bool': E) ']' ^
< ELSE_PART → < 'exit' ^ S >? > ^ ']''.
endproduction
endtranslator;

translator IN_STM

pattern 'input' ^ I ← ID endpattern
production
[ if ST.I ← null
then
control INT_MODE boolean ST.I.'TYPE' = 'int';
control DEREFS integer ST.I.'REFS' - 1
else fail
endif ]
'assign' ^
< INT_MODE →
'INT_MODE<INTEGER_TYPE<READ_INT' |
'BOOL_MODE<BOOLEAN_TYPE<READ_BOOL' > ^
'>> to' ^ I ^ < DEREFS → '@' '> ^
endproduction
endtranslator;

translator OUT_STM

pattern 'output' ^ E ← EXP(: REFS, TYPE) endpattern
production
[ control INT_MODE boolean TYPE = 'int' ]
< INT_MODE → 'WRITE_INT' | 'WRITE_BOOL' > ^
'(#' ^ DEREFS(REFS, TYPE: E) ^ ')'.
endproduction
endtranslator;
Appendix 5 - Definition of ASPLE

translator EXP(: REFS, TYPE)
local T use V
pattern F ← FACTOR(: PR, PT) / '+' endpattern
production
[ assign FT.1 to T;  
  assign T to TYPE;  
  control INT_MODE boolean T = 'int';  
  control NOPS integer #F - 1;  
  control SIMPLE boolean NOPS = 0 ]
  < SIMPLE →
  [ assign FR.1 to REFS ]
  F.1 |
  [ assign O to REFS ]
  < NOPS →
  < INT_MODE →
  'INTEGER_TYPE<ADD(#' |  
  'BOOLEAN_TYPE<OR(#' + > ^
  DEREF(FR.1, T: F.1) ^
  < NOPS(I) →
  [ if T ' = PT.(I+1) then fail endif ]
  < ', # ' ^ DEREF(PR.(I+1), T: P.(I+1)) ^ ' )' >+ >
>>
endproduction
endtranslator;

translator FACTOR(: REFS, TYPE)
local T use V
pattern P ← PRIMARY(: PR, PT) / '*' endpattern
production
[ assign PT.1 to T;  
  assign T to TYPE;  
  control INT_MODE boolean T = 'int';  
  control NOPS integer #P - 1;  
  control SIMPLE boolean NOPS = 0 ]
  < SIMPLE →
  [ assign FR.1 to REFS ]
  P.1 |
  [ assign O to REFS ]
  < NOPS →
  < INT_MODE →
  'INTEGER_TYPE<MULTIPLY(#' |  
  'BOOLEAN_TYPE<AND(#' + > ^
  DEREF(PR.1, T: P.1) ^
  < NOPS(I) →
  [ if T ' = PT.(I+1) then fail endif ]
  < ', # ' ^ DEREF(PR.(I+1), T: P.(I+1)) ^ ' )' >+ >
>>
endproduction
endtranslator;
translator PRIMARY(: REFS, TYPE)

  pattern
  I ← ID | C ← CONSTANT(: TYPE) | (' ^ ' E ← EXP(: EREFS, ETYPE) | P ← COMPARE > ^ ' )'

endpattern

production
  [ control IE boolean I = null;
    control CE boolean C = null;
    control EE boolean E = null ]

< IE →
  [ if ST.I = null
      then
        assign ST.I.'TYPE' to TYPE;
        assign ST.I.'REFS' to REFS
      else fail
      endif ]

I |

< CE →
  [ assign 0 to REFS;
    assign CTYPE to TYPE ]

C |

< EE →
  [ assign 0 to REFS;
    assign ETYPE to TYPE ]

DEREF(EREF, ETYPE: E)
  [ assign 0 to REFS;
    assign 'bool' to TYPE ]

P

>>> endproduction
endtranslator;

translator CONSTANT(: TYPE)

pattern B ← < 'true' | 'false' > | I ← NUMBER endpattern

production
  [ control INT_MODE boolean I = null ]

< INT_MODE →
  [ if #I <= n3
      then assign 'int' to TYPE
      else fail
      endif ]

< 'INTEGER_TYPE' ^ ! I ^ '>' > |
  [ assign 'bool' to TYPE;
    control TRUE boolean B = 'true' ]

< 'BOOLEAN_TYPE' ^ < TRUE → '1' | '0' > ^ '>' >

endproduction
endtranslator;
translator COMPARE

pattern
L ← EXP(: LREFS, LTYPE) ^ OP ← < '=' | '#' > ^
R ← EXP(: RREFS, RTYPE)
endpattern

production
[ if LTYPE = 'int' & RTYPE = 'int'
then control NEQ boolean OP = '#'
else fail
endif ]

'BOOLEAN TYPE< '

< NEQ → 'NOT(' ? > '
'EQUAL('# ^ DEREF(LREFS, 'int': L) ^
'#, #' ^ DEREF(RREFS, 'int': R) ^ ')) ^
< NEQ → ')'? > ^ '>'

endproduction

endtranslator
Appendix 6 - Description of Pascal

APPENDIX SIX

1. The REAL type.

Can be included by adding following GLAIS functions to provide "real" arithmetic e.g.: MULTIPLY, DIVID, MODULO,... etc. The REAL type would be represented by an internal type. Alternatively, the REAL type could be a structure type consisting of internal types representing, for instance, exponent and sign separately. This option would require arithmetic operations to be written in GLAIS.

2. Value Records.

Would use the union type structuring to combine each record variant.

3. Values.

A file would be implemented as a recursive type (see file test, section 5.5.4).

4. Input and Output Function Extensions.

Each would be implemented as a GLAIS function to read and write values of each different type.

5. Functions and Procedures as Parameters.

Would require a special check in GLAIS to check for compatible parameter lists and function types of these procedures and functions passed as parameters each time they were called.

6. Forward Declarations.

Would declare all procedures and functions in a block in one particular variable.

7. Full Controlled Variable Declaring.

The subscript specification that an array can be made to be controlled variable while an indexed can be an indexed (see Fig. 1). This requires that every variable would have a unique name. This variable must have a name that indicates whether it is an entire controlling variable. The GLAIS representation of such a variable would be a structure with the name variable type for the variable value plus an array of variable name index to represent the array. The user would refer the control's subscript parameter index to represent this variable.

8. Value Test of Procedures and Functions.

The type name of the procedure where the target label appears
Appendix 6 - Description of Pascal

1 A SYNTRAN/GLADYS DESCRIPTION OF A SUBSET OF PASCAL

1.1

The description in this appendix follows Pascal as defined in [RaveB79] (a more recent draft proposal appears in [AddyA80]) except for the following omissions which were made to reduce the length and complexity of the description. An indication is given as to what GLADYS features would be used if these omissions were included.

1. The REAL type.
   Can be included by adding external GLADYS functions to provide "real" arithmetic e.g. REAL_ADD(2), REAL_DIVIDE(2), ... etc. The REAL type would be represented by an interval type. Alternatively, the REAL type could be a structure type consisting of interval types representing the mantissa, exponent and sign separately. This option would require arithmetic operations to be written in GLADYS.

2. Variant Records.
   Would use the union type constructor to combine each record variant.

3. Files.
   A file would be implemented as a recursive type (See main text, section 5.2.4).

4. Input and Output Procedures and Functions.
   Most easily implemented by using external GLADYS functions to read and write values of each different type.

5. Functions and Procedures as Parameters.
   Would require a 'run-time' check in GLADYS to check for compatible parameter lists and function types of those procedures and functions passed as parameters each time they were called.

6. Forward Declarations.
   Would declare all procedures and functions in a block in one parallel statement.

7. Full Controlled Variable Checking.
   The Standard specifies that no assignments can be made to a controlled variable while it is being used in an active 'for-loop'. This requires that every variable that is used as a controlled variable must have a flag associated with it indicating whether it is an active controlled variable. The GLADYS representation of such a variable would be a structure type with the normal interval type for the variable value plus an interval type with two values to represent the flag. The more recent draft standard proposal [AddyA80] covers this problem with extra syntactic restrictions.

   The type name of the process where the target label appears
Appendix 6 - Description of Pascal

would be stored along with the label name of the target label. The process containing the goto would be stopped and the process with the target label started achieving, in effect, a procedure or function return.


ABS, SQR, SIN, COS, EXP, LN, SQRT, ARCTAN, TRUNC, ROUND, SUCC, PRED, ODD, EOF and EOLN are omitted. Their inclusion would require either straightforward GLADYS or external function definitions.

Note that this definition has not been verified in any way so that it is likely that it contains errors. Its purpose is only to serve as a source of examples and as an example of the use of SYNTRAN and GLADYS.

1.2

A description of some of the more difficult parts of the definition concerning the context sensitive syntax follows.

1.2.1

Three tables are used to store context sensitive information: the symbol table, ST; the pointer table, POINTERS; and the label table, LABELS. ST, POINTERS and LABELS are SYNTRAN nodes. The first level of branching from these nodes is a positive integer denoting the nesting level of the particular structure. The symbol table level, LEVEL, is incremented on entry to blocks, record field name declarations and with-statement variable occurrences and decremented on exit from blocks, record declarations and with-statements. The level of POINTERS and LABELS, BLEVEL, is incremented and decremented only on entry and exit from blocks.

ST is used to store all information required regarding declared identifiers. POINTERS is used to resolve the forward referencing allowed in pointer type declarations and LABELS is used to ensure that all declared labels are defined.

The selectors used on the second level of branching of ST are the Pascal identifier strings found in the source text. The value of each of these branches is a node with selectors and values are as set out in the table below and is referred to as a symbol table entry.
Appendix 6 - Description of Pascal

<table>
<thead>
<tr>
<th>Selectors</th>
<th>'MODE'</th>
<th>'TYPE'</th>
<th>'#'</th>
<th>'KIND'</th>
<th>'PARAMS'</th>
<th>'ENC SCOPE'</th>
<th>'LOCAL SCOPE'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'TYPE'</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'CONST'</td>
<td>*</td>
<td>value</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'VAR'</td>
<td>*</td>
<td>p</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'PROC'</td>
<td>number of parameters</td>
<td>l</td>
<td>g</td>
<td>g</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'FUNC'</td>
<td>number of parameters</td>
<td>l</td>
<td>g</td>
<td>g</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'FIELD'</td>
<td>*</td>
<td>position in record</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where:

* stands for a type entry (described below)

k stands for kind of variable; value is one of 'ACTUAL', 'VAR', 'VALUE', 'FUNC' or 'CONTROLLED' meaning, respectively, a normal variable, a VAR parameter, a normal parameter, a function name variable used to return the value of the function of the same name and the controlled variable of an enclosing for-statement.

p stands for position in parameter list when associated k = 'VAR' or 'VALUE', null otherwise.

l stands for parameter list. This is a node with selectors the positive integers corresponding to the positions of the formal parameters of the procedure or function that this entry belongs to. The value of each of these branches is a symbol table entry with a branch selected by 'MODE' having a value of 'VAR' and a branch selected by 'KIND' having the value 'VALUE' or 'VAR' depending on whether the parameter is normal or a VAR parameter. This will be written in future as: "a symbol table entry with 'MODE' = 'VAR' and 'KIND' = 'VALUE' or 'VAR' depending ... ."

g stands for a generated name used in the GLADYS output.

A type entry is a node with following form:
Appendix 6 - Description of Pascal

<table>
<thead>
<tr>
<th>Selectors Values</th>
<th>'FORM'</th>
<th>'NAME'</th>
<th>'BASE'</th>
<th>'MIN'</th>
<th>'MAX'</th>
<th>'INDEX'</th>
<th>'FIELDS'</th>
<th>'# FIELDS'</th>
</tr>
</thead>
<tbody>
<tr>
<td>'SCALAR'</td>
<td>g</td>
<td></td>
<td>n</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'SUBRANGE'</td>
<td>g</td>
<td>*</td>
<td>n</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'SET'</td>
<td>g</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'ARRAY'</td>
<td>g</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'POINTER'</td>
<td>g</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'RECORD'</td>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g</td>
<td>s</td>
</tr>
</tbody>
</table>

where:
- g stands for a generated name (as above)
- * stands for a type entry (as above)
- n stands for an integer value
- s stands for a node with selectors equal to the field name selectors of the record type and symbol table entries as values. These entries have 'MODE' = 'FIELD'.

The selectors used on the second level of branching of POINTERS are the identifiers used as names of types in the pointer type declarations. Since these types need not be defined at the time they are used, the type entries for such pointer types cannot be completed in this context. Specifically the 'BASE' selected branch cannot, in general, be given a value. The values of the branches on the second level of POINTERS is the incomplete type entry. When all declarations in a block have been parsed the identifiers used as the names of base types and second level selectors of POINTERS are checked to make sure that they are type identifiers (by a normal symbol table search). The type entries associated with these type identifiers are then included in the type entries of the pointer types which are accessed as the values of the second level of branches of POINTERS. For example, with the declaration:

```pascal
type X: 'Y;
```

POINTERS and ST are augmented:
When the declarations in the current block have been completed, Y will now be declared as a type identifier (in a correct program).

and the type entry associated with X is completed:
Finally, the current level of POINTERS is deleted leaving the symbol table entries intact.

The selectors used in the second level of branching of LABELS are the digit sequences declared as labels. The value of each of these branches is a node with branches selected by 'NAME' and 'STATUS'. The value of the former branch is a generated non-zero integer used as a label in the GLADYS program and the value of the latter branch is initially 'DECLARED'. When a label is encountered as a definition, that is, preceding a colon and a statement, the corresponding value
of the 'STATUS' branch is changed to 'DEFINED'. At the end of the block all such branches are checked: any that still have the value 'DECLARED' have not been defined in that block (and are therefore in error).

To enforce the rules laid out in [RaveB79] regarding the use of labels, two parameters are returned from the translator STATEMENT. The first is a node which is used to represent the set of labels used in goto-statements within that statement. The second is the label, if any, attached to that same statement. A label can only be used within the statement where it is defined or within a statement in a statement sequence where the label is defined on one of the statements of that sequence. The translator STATEMENT_SEQUENCE unites the set of used labels in each of its constituent statements and then removes any labels which are defined on any of its constituent statements. The resulting set is then passed back as a value.

In any place where the translator STATEMENT is used, the label defined in that statement, if any, is removed from the set of those labels used in that statement. The resulting set is then passed back as a value as above. These sets of labels thus specify the labels that have been used but not yet defined at a lower level of statement nesting as required by the standard definition. If, at the end of a block, the set returned by STATEMENT_SEQUENCE of that block is not empty then at least one label has been mis-used. Either it has been used without any definition at all or it was used in an illegal manner.

1.2.2

To handle types which are "compatible" during execution and hence in the GLADYS target program, but anonymous in the source text, a mechanism like that used to handle the 'string' types is required. Here, two separate nodes (STRING_TYPES and STRING_INDEX_TYPES) are used to store the generated type names of all string types that are created in a Pascal program. Specifically, the name of the structure type forming the array and the name of the interval type forming the index type for the array have to be stored. Thus when two string values have compatible types and can therefore be compared in Pascal, they will have the same type GLADYS type. This allows a great deal of simplification in the GLADYS program produced.

1.2.3

A procedure is modelled by a structure type consisting of a cell to store a reference to the "calling" process, the actual "called" process and a number of other cells which correspond to the formal parameters. Formal parameters called by value have a cell of the appropriate type in the structure while VAR parameters have a cell of type cell to refer to the actual parameter variable. A function is
modelled by a similar structure type but with one extra cell to store the result of the function.
Appendix 6 - Description of Pascal

{ definition of sets }
set LETTER < 'A' .. 'Z' >,
    ID_CHAR < 'A' .. 'Z', '0' .. '9' >,
    DIGIT < '0' .. '9' >,
    ONE_TO_NINE < '1' .. '9' >,
    SIGN < '+', '-' >,
    NOTSQ not < '}', eol >,
    COMMENT_CHAR not < ';' >,
    ALL_CHAR not < eol >;

{ syntactic tokens for Pascal }
token IDENT < LETTER ^ ID_CHAR* >, [identifier]
    UNSN < DIGIT+ >, [unsigned integer]
    LIT < "" ^ NOTSQ* ^ "" >, [or number]
    LITERAL < LIT+ >, [string literal]
    NUMBER < SIGN? ^ UNSN >, [optionally signed number]
    COMMENT < "{ " COMMENT_CHAR* ^ "}" >;

{ tokens for GLADYS - generated names }
token TYPE_NAME < 'T' ^ UNSN >,
    LABEL_NAME < ONE_TO_NINE ^ DIGIT* >, [positive integers]
    SCOPE_NAME < 'S' ^ UNSN >;
ignore < ' ' | eol | COMMENT >* endignore;

{ global variables }
global ST, [symbol table]
    STL, [alias for branches of symbol table at]
        LEVEL, [particular level]
        BLEVEL, [current symbol table level]
        MAXINT, [current block level]
        CHAR_SET_SIZE, [maximum Pascal integer value]
        LABELS, [size of Pascal character set and]
            POINTERS, [maximum number of elements in set]
            STRING_TYPES, [type values]
            STRING_INDEX_TYPES, [node to check usage of labels]
            STRING_INDEX_TYPES, [node to check usage of pointers]
            STRING_INDEX_TYPES, [type names for created string types]
            STRING_INDEX_TYPES, [index type names for above]
            STRING_INDEX_TYPES, [aliases for pre-defined Pascal types in]
            STRING_INDEX_TYPES, [symbol table]
            INTEGER, [used in parsing array indices]
            BOOLEAN,
            CHAR,
            INDEX_FLAG;

{ set up alias for pre-defined identifier symbol table entries }
assign ST.0 to STL;
Appendix 6 - Description of Pascal

{ set up pre-defined types in symbol table }

{ INTEGER }
assign ??? to MAXINT;
give MAXINT implementation defined value
assign 'TYPE' to STL.'INTEGER'. 'MODE';
assign STL.'INTEGER'. 'TYPE' to INTEGER;
assign ['SCALAR', new(TYPE_NAME), -MAXINT, MAXINT] to INTEGER.[ 'FORM', 'NAME', 'MIN', 'MAX'];

{ BOOLEAN }
assign 'TYPE' to STL.'BOOLEAN'. 'MODE';
assign STL.'BOOLEAN'. 'TYPE' to BOOLEAN;
assign ['SCALAR', new(TYPE_NAME), 0, 1] to BOOLEAN.[ 'FORM', 'NAME', 'MIN', 'MAX'];

{ CHAR }
assign ??? to CHAR_SET_SIZE;
give CHAR_SET_SIZE implementation defined value
assign 'TYPE' to STL.'CHAR'. 'MODE';
assign STL.'CHAR'. 'TYPE' to CHAR;
assign ['SCALAR', new(TYPE_NAME), 0, CHAR_SET_SIZE] to CHAR.[ 'FORM', 'NAME', 'MIN', 'MAX'];

{ set up pre-defined Pascal constants }
{ FALSE }
assign ['CONST', BOOLEAN, 0] to STL.'FALSE'.[ 'MODE', 'TYPE', '#'];
{ TRUE }
assign ['CONST', BOOLEAN, 1] to STL.'TRUE'.[ 'MODE', 'TYPE', '#'];
{ MAXINT }
assign ['CONST', INTEGER, MAXINT] to
STL.'MAXINT'.[ 'MODE', 'TYPE', '#'];

{ other initialization }
assign 1 to LEVEL;
assign 1 to BLEVEL;
assign 0 to INDEX_FLAG;

{ functions used in manipulating sets of labels }

function EMPTY(NODE)
  return NODE = null
endfunction;
Appendix 6 - Description of Pascal

function REMOVE(NODE, VALUE)
  local N use V
  assign NODE to N;
  if VALUE ≠ null then assign null to N.VALUE endif;
  return N
endfunction;

function UNITE(NODE1, NODE2)
  local N, N2N use V
  assign NODE1 to N;
  assign !NODE2 to N2N;
  for I from 1 to #N2N do assign 1 to N.(N2N.I) endfor;
  return N
endfunction;

{ symbol table searching function }
function SEARCH_ID(ID)
  local L use V
  assign LEVEL to L;
  loop
    if ST.L.ID ≠ null then exit endif;
    if L = 0 then exit else assign L - 1 to L endif
  endloop;
  return ST.L.ID
endfunction;

{ functions for manipulating and comparing types }
function CONFORM(T)
  { returns the type of expression consisting of one operand
    of type T } 
  local SET use V
  case T.'FORM'
    of 'SUBRANGE': return T.'BASE'
    of 'SCALAR': return T
    of 'SET':
      assign 'SET' to SET.'FORM';
      assign CONFORM(T.'BASE') to SET.'BASE';
      return SET
    otherwise return T
  endcase
endfunction;

function IS_STRING_TYPE(T)
  { tests T for string type }
  return T.'FORM' = 'ARRAY' & T.'BASE' = CHAR &
        T.'INDEX'.FORM = 'SUBRANGE' & T.'INDEX'.MIN = 1 &
        T.'INDEX'.BASE = INTEGER
endfunction;
function IDENTICAL(T1, T2)
{ tests for identical types. In fact returns true for compatible string types but such types are never tested for identity }
return T1 = T2 |
T1.'FORM' = 'POINTER' & T2.'FORM' = 'POINTER' &
(T1.'BASE' = null | T2.'BASE' = null)
{ NIL is a member of all pointer types }
T1.'FORM' = 'SET' & T2.'FORM' = 'SET' &
(T1.'BASE' = null | T2.'BASE' = null)
{ [] is a member of all set types }
endfunction;

function COMPATIBLE(T1, T2)
{ tests types for compatibility }
return
IS STRING TYPE(T1) & IS STRING TYPE(T2) &
T1.'INDEX'.MAX' = T2.'INDEX'.MAX'
IDENTICAL(T1, T2) |
CONFORM(T1) = CONFORM(T2)
endfunction;

function ASSIGNMENT_COMPATIBLE(T1, T2)
return IDENTICAL(T1, T2) |
(T1.'FORM' = 'SCALAR' | T1.'FORM' = 'SUBRANGE') &
COMPATIBLE(T1, T2) &
T1.'MIN' <= T2.'MAX' & T1.'MAX' >= T2.'MIN'
T1.'FORM' = 'SET' & ASSIGNMENT_COMPATIBLE(T1.'BASE', T2.'BASE')
endfunction;

{ macro definitions }

macro ENTER(S: )
{ macro to enter new scope for temporaries }
production
[ emission SCOPE is SCOPE_NAME: S ]
'bind' SCOPE 'to create_basket next' 'enclosing' SCOPE 'is current_basket next' 'current' SCOPE 'next'
endproduction
endmacro;

macro EXIT(S: )
{ inverse of ENTER macro }
production
[ emission SCOPE is SCOPE_NAME: S ]
'destroy' SCOPE
endproduction
endmacro;
Appendix 6 - Description of Pascal

macro TO_INT(KIND : EXP)
{ macro to convert variable to its value }
production
[ control VAR boolean KIND = 'ACTUAL' | KIND = 'VAR' |
  KIND = 'VALUE' | KIND = 'CONTROLLED' ]
  < VAR → '#'? > ^ EXP ^ < VAR → '@'? >
endproduction
endmacro;

macro DEREFS(TYPE , KIND : EXP)
{ macro to dereference variable }
production
[ control VAR boolean KIND = 'VAR' | KIND = 'VALUE' | KIND = 'ACTUAL' |
  KIND = 'CONTROLLED' ;
  control SIMPLE boolean TYPE . 'FORM' = 'SCALAR' |
  TYPE . 'FORM' = 'SUBRANGE' | TYPE . 'FORM' = 'SET';
  emission EXP_TYPE is TYPE_NAME : TYPE . 'NAME' ]
  < SIMPLE →
    < EXP_TYPE ^ '<' ^ TO_INT(KIND : EXP) ^ '>' >
  endproduction
endmacro;

macro OPERATION(N, KINDS : OPERANDS , OPERATION)
{ macro to convert list of one infix operator to prefix function form of GLADYS }
production
[ control ONE boolean N = 1;
  control N_OPS integer N - 1 ]
  < ONE →
    OPERANDS . 1 ≥
  < < N_OPS → < OPERATION ^ '(' >+ >
    TO_INT(KINDS . 1 : OPERANDS . 1 ) ^
    < N_OPS ( I ) →
      < ', ' ^ TO_INT(KINDS . ( I + 1 ) : OPERANDS . ( I + 1 )) ^ ';' >+ >
    endproduction
endmacro;

translator ID
{ identifier in declarations }
pattern I ← IDENT endpattern
production
[ if ST . LEVEL . I ≠ null then fail { multiple declaration } endif ]
I
endproduction
endtranslator;
translator USEDID(: ENTRY)
{ identifier in expressions, supplies symbol table entry }
local L use V
pattern I ← IDENT endpattern
production
[ assign SEARCH ID(I) to L;
  if L = null then fail { not declared } endif;
  assign L to ENTRY ]
I
endproduction
endtranslator;

translator ID LIST(MODE : ID)
{ identifier list as in scalar type, VAR declarations and formal parameter lists. Starts symbol table entry for each identifier }
pattern I ← IDENT / ',' endpattern
production
[ for J from 1 to #I do
  if ST.LEVEL.(I.J) = null
    then assign MODE to ST.LEVEL.(I.J)."MODE"
  else fail { multiple declaration }
  endif
  endfor;
  assign I to ID ]
endproduction
endtranslator;

translator IDENT LIST(MODE : ID)
{ identifier list as in record declarations }
pattern I ← IDENT / ',' endpattern
production
[ for J from 1 to #I do
  if ST.(LEVEL+1).(I.J) = null
    then assign MODE to ST.(LEVEL+1).(I.J)."MODE"
  else fail { multiple declaration }
  endif
  endfor;
  assign I to ID ]
endproduction
endtranslator;
translational STRING
{ constant character string }
local NSQ, J use V
pattern L ← LITERAL endpattern
production
[ assign 0 to NSQ;
  for I from 2 to #L - 1 do
    { count number of single quotes in string }
    if substr(L, I, 1) = "" then assign NSQ + 1 to NSQ endif
  endfor;
  control LEN integer #L - 2 - NSQ / 2;
  { calculate length of string }
  assign 2 to J
  < LEN →
[ if substr(L, J, 1) = "" then assign J + 1 to J endif;
  emission C is ALL CHAR: substr(L, J, 1);
  assign J + 1 to J
] C* >
endproduction
endtranslator;
translator CONSTANT(: TYPE, VALUE)
{ constant string, identifier and number }
{ used in CONST and subrange declarations and CASE statement }
local LENGTH use V
pattern
W ← STRING | S ← SIGN? ^
< UI ← USEDID(: ENTRY) | UN ← UNSN >
endpattern
production
[ assign #W to LENGTH;
case LENGTH
of 0: { identifier or number }
   if UI ^= null
      then { identifier }
         if ENTRY.'MODE' = 'CONST'
            then
               assign ENTRY.'TYPE' to TYPE;
               if S = '-'
                  then
                     if ENTRY.'TYPE' = INTEGER
                        then assign -(ENTRY.'#') to VALUE
                        else fail
                     endif
                  else assign ENTRY.'#' to VALUE
               endif
            endif
      else fail
   endif
else { number }
   assign INTEGER to TYPE;
   if UN > MAXINT then fail endif;
   if S = '-'
      then assign -UN to VALUE
   else assign UN to VALUE
   endif
endif
of 1: { constant of type CHAR }
   assign CHAR to TYPE;
   assign W to VALUE
otherwise { string constant }
   if STRING_TYPES.LENGTH = null
      then
         assign new(TYPE_NAME) to STRING_TYPES.LENGTH;
         assign new(TYPE_NAME) to STRING_INDEX_TYPES.LENGTH
      endif;
   assign ['ARRAY', STRING_TYPES.LENGTH, CHAR] to
      TYPE:['FORM', 'NAME', 'BASE'];
   assign ['SUBRANGE', STRING_INDEX_TYPES.LENGTH, INTEGER, 1, LENGTH] to
      TYPE:['INDEX'][['FORM', 'NAME', 'BASE', 'MIN', 'MAX']];
   assign W to VALUE
endcase ]
endproduction
Appendix 6 - Description of Pascal

endtranslator;

translator LABEL DEC
{ label declaration }
local L use V
pattern 'LABEL' ^ LBLs ← UNSN / ' , ' ^ ' ; ' endpattern
production
[ for I from 1 to #LBLs do
  assign LABELS.LEVEL.(LBLs.I) to L;
  if L = null
  then { label not declared at this level }
      assign ['DECLARED', new(LABEL_NAME)] to L.[ 'STATUS', 'NAME' ]
  else fail { multiple declaration }
  endif
endfor ]
end production
endtranslator;

translator CONST DEC
{ declaration of constants }
local ENTRY use W
pattern
  'CONST' ^
  < CID ← ID ^ '=' ^ CONSTANT(: TYPE, VALUE) ^ ' ; ' >+
endpattern
production
[ for I from 1 to #CID do
  assign ST.LEVEL.(CID.I) to ENTRY;
  assign ['CONST', TYPE.I, VALUE.I] to
  ENTRY.[ 'MODE', 'TYPE', '#' ]
endfor ]
end production
endtranslator;

translator TYPE ID(: TYPE)
pattern I ← USEDID(: ENTRY) endpattern
production
[ if ENTRY.'MODE' = 'TYPE'
  then assign ENTRY.'TYPE' to TYPE
  else fail
  endif ]
end production
endtranslator;
translator SCALAR_TYPE(: ENTRY)
local T_ENTRY use V
local ID_ENTRY use W
pattern '(' ^ ID_LIST('CONST': IDS) ^ ')' endpattern
production
[ [ set up type entry for individual constants ]
  emission MAX is UNSN: #IDS - 1;
  emission TYPE is TYPE_NAME: new(TYPE_NAME);
  assign ['SCALAR', TYPE, O, MAX] to T_ENTRY.['FORM', 'NAME', 'MIN', 'MAX'];
  assign T_ENTRY to ENTRY;
] [ set up individual identifier symbol table entries ]
for I from O to MAX do
  assign ST.LEVEL.(IDS.(I+1)) to ID_ENTRY;
  assign [T_ENTRY, I] to ID_ENTRY.['TYPE', '#']
endfor
'bind' ^ TYPE ^ 'to interval(0,' ^ MAX ^ ')' next' >
endproduction
endtranslator;

translator SUBRANGE_TYPE(: ENTRY)
pattern
CONSTANT(: L_TYPE, L_VALUE) ^ ' ' ^
CONSTANT(: R_TYPE, R_VALUE)
endpattern
production
[ if L_TYPE. 'FORM' = 'SCALAR' & L_TYPE = R_TYPE & L_VALUE <= R_VALUE
  then
    emission TYPE is TYPE_NAME: new(TYPE_NAME);
    assign ['SUBRANGE', TYPE, L_TYPE, L_VALUE, R_VALUE] to ENTRY.['FORM', 'NAME', 'BASE', 'MIN', 'MAX'];
    emission L is NUMBER: L_VALUE;
    emission R is NUMBER: R_VALUE
  else fail
    end if
  ]
'bind' ^ TYPE ^ 'to interval(' ^ L ^ ',' ^ R ^ ') next' >
endproduction
endtranslator;
Appendix 6 - Description of Pascal

translator SIMPLE_TYPE(: ENTRY)
{ scalar or subrange type }

pattern
T ← < SCALAR_TYPE(: SC_ENTRY) | SUBRANGE_TYPE(: SU_ENTRY) > | TYPE_ID(: ID_ENTRY)
endpattern

production
[ if ID_ENTRY ^= null
then
  if ID_ENTRY.'TYPE'.FORM = 'SCALAR'
  then assign ID_ENTRY.'TYPE' to ENTRY
  else fail
  endif
else
  if SC_ENTRY ^= null
  then assign SC_ENTRY to ENTRY
  else assign SU_ENTRY to ENTRY
  endif;
control TYPE_OUT boolean T ^= null ]
end production

translator POINTER_TYPE(: ENTRY)

pattern '"' BASE ← IDENT endpattern

production
[ assign POINTERS.BLEVEL.BASE to ENTRY;
  emission TYPE is TYPE_NAME: new(TYPE_NAME);
  assign ['POINTER', TYPE] to ENTRY.[FORM, 'NAME']
< 'bind' ^ TYPE ^ 'to cell next' >
end production

translator SET_TYPE(: ENTRY)

pattern 'SET' ^ 'OF' ^ BASE ← SIMPLE_TYPE(: BASE_TYPE) endpattern

production
[ emission TYPE is TYPE_NAME: new(TYPE_NAME);
  assign ['SET', TYPE, BASE_TYPE] to ENTRY.[FORM, 'NAME', 'BASE']
< BASE ^ 'bind' ^ TYPE ^ 'to SET_TYPE next' >
end production
translator RECORD_TYPE(: ENTRY)
local N_FIELDS, FIELD_INDEX, FIELDS use V

pattern
  'RECORD' ^ < IDENT_LIST('FIELD': FLS) ^ ': ' ^
  FTN ← TYPE(: _F_TYPE) > / ';' ^ 'END'
endpattern

production
[ emission TYPE is TYPE_NAME: new(TYPE_NAME);
  emission BASE_TYPE is TYPE_NAME: new(TYPE_NAME);
  assign ST.(LEVEL+1) to FIELDS;
  control N_TYPES integer #FTN;
  assign 0 to N_FIELDS; assign 0 to FIELD_INDEX;
  { calculate number of fields }
  for I from 1 to N_TYPES do
    assign N_FIELDS + #(FLS.I) to N_FIELDS
  endfor;
  assign ['RECORD', TYPE, BASE_TYPE, N_FIELDS] to
    ENTRY. [IFORM I 'NAME I 'INDEX I '##FIELDS I
  emission N_FIELDS is UNSN: N_FIELDS ]
  < N_TYPES(J) → FTN.J+ > ^
  'bind' ^ BASE_TYPE ^ 'to interval(1, ' ^ N_FIELDS ^ ' ) next' ^
  'bind' ^ TYPE ^ 'to structure(' ^ BASE_TYPE ^ ');' ^
  < N_TYPES(K) →
    [ emission FIELD_TYPE is TYPE_NAME: F_TYPE.K.'NAME' ]
    < N_IDS_TYPE(I) →
      [ assign FIELD_INDEX + 1 to FIELD_INDEX;
        assign [F_TYPE.K, FIELD_INDEX] to
          FIELDS.(FLS.K.I).['TYPE', '#'] ]
      FIELD_TYPE / '/;'' >
    / '/;'' >
  ] assign FIELDS to ENTRY.'FIELDS';
  assign null to ST.(LEVEL+1)
  { clear symbol table level used by field names } ]
endproduction
endtranslator;
translator ARRAY_TYPE(: ENTRY)
local STRUCT , BASE_TYPE, LENGTH use V
pattern
  'ARRAY' ^ '[ ' ^ INDEX ← SIMPLE_TYPE(: INDEX_ENTRY) / ', ' ^ 
  ']' ^ 'OF' ^ BASE ← TYPE(: BASE_ENTRY)
endpattern
production
  [ control N_INDICES integer #INDEX ;
  assign BASE_ENTRY to BASE_TYPE ]
  BASE ^
< N_INDICES(I) →
  [ emission I_TYPE is TYPE_NAME:
    INDEX_ENTRY.(N_INDICES + 1 - I). 'NAME';
 emission B_TYPE is TYPE_NAME: BASE_TYPE. 'NAME';
 emission A_TYPE is TYPE_NAME: new(TYPE_NAME);
 assign ['ARRAY', A_TYPE, BASE_TYPE,
 INDEX_ENTRY.(N_INDICES + 1 - I)] to
 STRUCT.[ 'FORM', 'NAME', 'BASE', 'INDEX'];
 assign STRUCT to BASE_TYPE ]
< 'bind' ^ A_TYPE ^ 'to_structure(' ^ I_TYPE ^ ' '; ^
  B_TYPE ^ ' ') next' >+
^ >
  [ if IS_STRING_TYPE(STRUCT)
    then
      assign STRUCT. 'INDEX'. 'MAX' to LENGTH;
    if STRING_TYPES. LENGTH = null
    then
      assign STRUCT. 'NAME' to STRING_TYPES. LENGTH;
    assign STRUCT. 'INDEX'. 'NAME' to STRING_INDEX_TYPES. LENGTH
    else
      assign STRING_TYPES. LENGTH to STRUCT. 'NAME';
    assign STRING_INDEX_TYPES. LENGTH to STRUCT. 'INDEX'. 'NAME'
    endif
    endif;
 assign STRUCT to ENTRY ]
endproduction
endtranslator;
translator TYPE(: ENTRY)
   pattern
   T ← <SIMPLE_TYPE(: S_ENTRY) | SET_TYPE(: T_ENTRY) |
   ARRAY_TYPE(: A_ENTRY) | RECORD_TYPE(: R_ENTRY) |
   POINTER_TYPE(: P_ENTRY) > | TYPE_ID(: I_ENTRY)
endpattern
production
[ if S_ENTRY == null
   then assign S_ENTRY to ENTRY
else
   if T_ENTRY == null
      then assign T_ENTRY to ENTRY
     else
       if A_ENTRY == null
          then assign A_ENTRY to ENTRY
       else
          if R_ENTRY == null then
              then assign R_ENTRY to ENTRY
         else
          if P_ENTRY == null
             then assign P_ENTRY to ENTRY
          else assign I_ENTRY to ENTRY
       endif
   endif
endif
endif
endif;
control TYPE_OUT boolean T =null
< TYPE_OUT → T? >
endproduction
endtranslator;

translator TYPE_DEC
{ type declaration }
pattern
   'TYPE'
   < TID ← ID ^ '=' ^ T ← TYPE(: ENTRY) ^ ';' >+
endpattern
production
[ control N_IDS integer #TID ]
< N_IDS(I) →
   [ assign ['TYPE', ENTRY.I] to
     ST.LEVEL.(TID.I).[MODE', 'TYPE'] ]
   T.I+ ]
endproduction
endtranslator;
translator VAR_DEC
{ variable declaration }
  local ENTRY use W
  pattern
  'VAR'
  < ID_LIST( 'VAR': VL ) > ': ' ^ T ← TYPE( :: TYPE_ENTRY ) > '
endpattern
production
  [ control N_LISTS integer #VL ]
  < N_LISTS(I) —>
  [ emission CELL_TYPE is TYPE_NAME: TYPE_ENTRY.I.'NAME';
    control N_IDS integer #(VL.I) ]
  < T.I
  < N_IDS(J) —>
  [ emission ID is IDENT: VL.I.J;
    assign ST.LEVEL.ID to ENTRY;
    assign [ TYPE_ENTRY.I, 'ACTUAL' ] to ENTRY.['TYPE', 'KIND'];
    control POINTER boolean ENTRY.['TYPE', 'FORM' = 'POINTER' ];
    < 'bind' ^ ID ^ 'to create cell(' ^ CELL_TYPE ^ ': ' ^
    < POINTER — 'no_cell' | CELL_TYPE ^ '<?>' > ^
    ') next' >+ ]
>>+ >
endproduction
endtranslator;

translator FUNCTION_TYPE_ID( :: TYPE )
{ type identifier used for return type of functions }
  local FORM use V
  pattern I ← USEDID( :: ENTRY ) endpattern
production
  [ if ENTRY.'MODE' = 'TYPE'
    then
      assign ENTRY.'TYPE'.FORM to FORM;
      if FORM = 'SCALAR' | FORM = 'SUBRANGE' | FORM = 'POINTER'
      then assign ENTRY.'TYPE' to TYPE
      else fail { type cannot be returned by function }
    endif
    else fail { identifier is not type identifier }
  endif
endproduction
endtranslator;
translator FORMAL_PARAMETER_LIST(BLOCK_NAME: )
{ bind formal to actual parameters }
local FORMAL_NUM, ID_LIST use V
local ENTRY use W
pattern
'(' ^ < V ← < 'VAR' | null > ^ ID_LIST('VAR': ID_LISTS) ^
'.' ^ TYPE_ID(: TYPE_LIST) / / ';! ^ ')'
endpattern
production
[ control N _ GROUPS integer #V;
assign 0 to FORMAL_NUM;
emission BLOCK is IDENT: BLOCK_NAME;
< '[' ^
< N _ GROUPS(I) →
[ assign ID_LISTS.I to ID_LIST;
control N _ IDS integer #ID_LIST;
control VAR boolean V.I = 'VAR' ]
< N _ IDS(J) →
[ emission NAME is IDENT: ID_LIST.J;
assign FORMAL_NUM + 1 to FORMAL_NUM;
emission POSITION is UNSN: FORMAL_NUM;
{ set up symbol table entries for formal parameters }
assign ST.LEVEL.NAME to ENTRY;
assign [TYPE_LIST.I, FORMAL_NUM] to
ENTRY.['TYPE', '#'];
if VAR
then assign 'VAR' to ENTRY.'KIND'
else assign 'VALUE' to ENTRY.'KIND'
endif ]
< 'bind' ^ NAME ^ 'to' ^ BLOCK ^ 'INSTANCE@.' ^
BLOCK ^ '_BASE<' ^ POSITION ^ '>^ ^ < VAR → '0'? > 
/ 'with' >
/ 'with' > ^
' ] next' >
endproduction
dentranslator;
Appendix 6 - Description of Pascal

translator FUNCTION_DEC(SCOPE: )
{ function declaration }
local PARAM_NAME_LIST, PARAM_LIST, VAR_ENTRY, NEW_SCOPE use V
local ENTRY use W
pattern 'FUNCTION' ^ FUNC_NAME ← ID endpattern
production
[ { set up symbol table entry for function itself }
  assign ST.LEVEL.FUNC_NAME to ENTRY;
  assign new(SCOPE_NAME) to NEW_SCOPE;
  assign ['FUNC', SCOPE, NEW_SCOPE ] to
    ENTRY.['MODE', 'ENC SCOPE', 'LOCAL SCOPE'];
{ increment symbol table level }
  assign LEVEL + 1 to LEVEL;
  assign BLEVEL + 1 to BLEVEL ]
endproduction
pattern
FORMALS ← FORMAL_PARAMETER_LIST(FUNC_NAME: )? ^ ' ' ^
  FUNC_TYPE ID(: RETURN_TYPE) ^ ';' ^
BODY ← BLOCK(NEW_SCOPE: ) ^ ';' ^
endpattern
production
[ assign ST.LEVEL to PARAM_LIST;
  { set up by FORMAL_PARAMETER_LIST }
  control N_PARAMS integer #PARAM_LIST;
  assign !PARAM_LIST to PARAM_NAME_LIST;
  { complete symbol table entry for function itself }
  assign [RETURN_TYPE, N_PARAMS, PARAM_LIST] to
    ENTRY.['TYPE', '#', 'PARMS'];
  { set up function-named return variable }
  assign ST.LEVEL.FUNC_NAME to VAR_ENTRY;
  if VAR_ENTRY = null
    then
      fail { formal parameter with same name as function }
    endif;
  assign ['VAR', RETURN_TYPE, 'FUNC'] to
    VAR_ENTRY.['MODE', 'TYPE', 'KIND'];
  emission N_PARAMS is UNSN : N_PARAMS;
  emission RETURN is TYPE_NAME: RETURN_TYPE.'NAME';
  assign null to ST.LEVEL; { clear symbol table level }
  assign LEVEL - 1 to LEVEL; { decrement symbol table level }
  assign BLEVEL - 1 to BLEVEL ]
  'bind' ^ FUNC_NAME ^ 'BASE to' ^
    'interval(-2, ' ^ N_PARAMS ^ ')' next' ^
  'bind' ^ FUNC_NAME ^ 'PROCESS to process[' ^
    < ANY_PARAMS → FORMALS? > ^
    'bind' ^ FUNC_NAME ^ 'to' ^ FUNC_NAME ^ '_INSTANCE@.' ^
      FUNC_NAME ^ 'BASE<-2> next' ^
    BODY ^ 'next' ^
    'transfer' ^
  FUNC_NAME ^ '_INSTANCE@.' ^ FUNC_NAME ^ '_BASE<-1>@] next' ^
  'bind' ^ FUNC_NAME ^ 'to structure(' ^ FUNC_NAME ^ '_BASE;' ^
RETURN ^ ', cell,' ^ FUNC_NAME ^ '_PROCESS' ^
277
Appendix 6 - Description of Pascal

< n params (j)-->
   [ control call by var boolean
     param list.(param name list.j).kind = 'var';
     emission param type is type name:
       param list.(param name list.j).type.'name' ]
   < ',' ^ < call_by_var --> 'cell' | param_type >>* > ^
  ') next'
endproduction
endtranslator;
translator PROCEDURE_DEC(SCOPE: )
{
procedure declaration
local PARAM NAME_LIST, PARAM_LIST, NEW_SCOPE use V
local ENTRY use W
pattern 'PROCEDURE' ^ PROC_NAME ← ID endpattern
production
[ ]
set up symbol table entry for procedure
assign ST.LEVEL.PROC_NAME to ENTRY;
assign new(SCOPE_NAME) to NEW_SCOPE;
assign [ 'PROC', SCOPE, NEW_SCOPE ] to
ENTRY.[ 'MODE', 'ENC SCOPE', 'LOCAL SCOPE' ];
{ increment symbol table level }
assign LEVEL +1 to LEVEL;
assign BLEVEL +1 to BLEVEL ]
endproduction
pattern
FORMALS ← FORMAL_PARAMETER_LIST(PROC_NAME: )? ^ ',' ^
BODY ← BLOCK(NEW_SCOPE: ) ^ ','
endpattern
production
[ ]
assign ST.LEVEL to PARAM_LIST;
{ set up by FORMAL_PARAMETER_LIST };
control N_PARAMS integer #PARAM_LIST;
assign !PARAM_LIST to PARAM_NAME_LIST;
{ complete symbol table entry for procedure }
assign [N_PARAMS, PARAM_LIST] to ENTRY.[ '#', 'PARAMS' ];
emission N_PARAMS is UNSN: N_PARAMS;
assign null to ST.LEVEL; { clear symbol table level }
assign LEVEL - 1 to LEVEL; { decrement symbol table level }
assign BLEVEL - 1 to BLEVEL ]
'bind' ^ PROC_NAME ^ 'BASE to' ^
'interval(-1, ^ N_PARAMS ^ ') next' ^
'bind' ^ PROC_NAME ^ '_PROCESS to process[' ^
< ANY_PARAMS → FORMALS? > ^ BODY ^ 'next' ^
'transfer' ^
PROC_NAME ^ 'INSTANCE@.' ^ PROC_NAME ^ '_BASE< -1>@' next' ^
'bind' ^ PROC_NAME ^ 'to structure(' ^ PROC_NAME ^ '_BASE; ' ^
'cell,' ^ PROC_NAME ^ '_PROCESS' ^
< N_PARAMS(J) →
[ ]
control CALL_BY_VAR boolean
PARAM_LIST.(PARAM_NAME_LIST.J). 'KIND' = 'VAR';
emission PARAM_TYPE is TYPE_NAME:
PARAM_LIST.(PARAM_NAME_LIST.J). 'TYPE'. 'NAME' ]
< , ^ < CALL_BY_VAR → 'cell' | PARAM_TYPE >>* >
') next'
endproduction
endtranslator:
Appendix 6 - Description of Pascal

translator UNSIGNED_CONSTANT(: TYPE) { used in expressions }

| Pattern |
|-----------------|-----------------|
| $S \leftarrow \text{STRING} \mid \text{UN} \leftarrow \text{UNSN} \mid \text{UI} \leftarrow \text{USEDID}(: \text{ENTRY}) \mid P \leftarrow '\text{NIL}'$ |

| Production |
|-------------|-----------------|
| $\text{control LENGTH integer }#S$; | $\text{control STR boolean LENGTH} > 1$; |
| $\text{control CHR boolean LENGTH} = 1$; | $\text{control INT boolean UN} \sim \text{null}$; |
| $\text{control IDN boolean UI} \sim \text{null}$; | $\text{control PTR boolean P} \sim \text{null}$ |

$< \text{STR} \rightarrow$

- { set up string type }
- if STRING_TYPES.LENGTH = null
  - then
    - assign new(TYPE_NAME) to STRING_TYPES.LENGTH;
    - assign new(TYPE_NAME) to STRING_INDEX_TYPES.LENGTH
  - endif;
- emission STRING_TYPE is TYPE_NAME: STRING_TYPES.LENGTH;
- emission STRING_INDEX_TYPE is TYPE_NAME: STRING_INDEX_TYPES.LENGTH;
- assign ['ARRAY', STRING_TYPE, CHAR] to TYPE['FORM', 'NAME', 'BASE'];
- assign ['SUBRANGE', STRING_INDEX_TYPE, INTEGER, 1, LENGTH] to TYPE['INDEX'][['FORM', 'NAME', 'BASE', 'MIN', 'MAX']]

$< \text{STRING_TYPE}^*['^{ '*}^$}$

$< \text{LENGTH(I)} \rightarrow$

- { emission INDEX is UNSN: I;
  - emission C is ALL CHAR: $S(I,1)$ }

$< \text{STRING_INDEX_TYPE}^*['^*^$ INDEX $^*']^*^'^\sim'^*$ INDEX $^*']^*'$

- '/with' $^*'$

$< \text{CHR} \rightarrow$

- { assign CHAR to TYPE }

$< \text{INT} \rightarrow$

- { assign INTEGER to TYPE }

$\text{UN} |$

$< \text{PTR} \rightarrow$

- { assign 'POINTER' to TYPE.'FORM'
  - 'no_cell' }

$< \text{IDN} \rightarrow$

- { if ENTRY.'MODE' = 'CONST' then
  - assign ENTRY.'TYPE' to TYPE;
  - control CONST_STR boolean ENTRY.'TYPE'.FORM = 'ARRAY';
  - control CONST_CHR boolean ENTRY.'TYPE' = CHAR;
  - control CONST_INT boolean ENTRY.'TYPE' = INTEGER;
  - control CONST_BOOL boolean ENTRY.'TYPE' = BOOLEAN
  - else fail { identifier not constant } }
Appendix 6 - Description of Pascal

```pascal
endif ]
< CONST_STR -->
  [ control C_LENGTH integer ENTRY.'TYPE'. 'MAX';
    emission C_TYPE is TYPE_NAME: ENTRY.'TYPE'. 'NAME';
    emission C_INDEX_TYPE is TYPE_NAME:
      ENTRY.'INDEX'. 'TYPE'. 'NAME'
  ]
< C_TYPE ^ '[' ^
< C_LENGTH(J) -->
  [ emission C.C is ALL CHAR: ENTRY.'#'(J,1);
    emission C_INDEX is UNSN: J ]
< C_INDEX TYPE ^ '<' ^ C_INDEX ^ '>' ^ '<' ^
  C_C ^ '*' >
  /'with' > ^
  ']' > ^
< CONST CHR -->
  [ emission CH is ALL CHAR: ENTRY.'#' ]
  < '..' ^ CH ^ '..' > ^
< CONST INT -->
  [ emission NUM is NUMBER: ENTRY.'#' ]
  NUM |
< CONST_BOL -->
  [ control TRUE boolean ENTRY.'#' = 1 ]
  < TRUE --> '1' | '0' >
>>>>
endproduction
endtranslator;
```
translator FUNCTION_CALL(SCOPE: RETURN_TYPE)
  local PARAM_ENTRY use V
  pattern
    FUNC_NAME ← USEDID(: ENTRY) ~
    < '(' ~ PARAMS ← EXPRESSION(SCOPE: TYPE, KIND) / ', ' ~ ')' ~>?
  endpattern
  production
  [ if ENTRY.'MODE' = 'FUNC'
    then
      control N_PARAMS integer #PARAMS;
      if N_PARAMS = ENTRY.'#' then
        assign ENTRY.'TYPE' to RETURN_TYPE;
        emission RESULT_TYPE is TYPE_NAME: ENTRY.'TYPE'.'NAME';
        emission LOCAL_SCOPE is SCOPE_NAME: ENTRY.'LOCAL SCOPE';
        emission ENC_SCOPE is SCOPE_NAME: ENTRY.'ENC SCOPE';
        else fail { incorrect number of arguments to function } endif
      else fail { identifier not function name } endif
    ]
  < RESULT_TYPE ~ '[' ~
    'bind' ~ LOCAL_SCOPE ~ 'to create basket next' ~
    'enclosing' ~ LOCAL_SCOPE ~ 'is' ~ ENC_SCOPE ~ 'next' ~
    'bind' ~ FUNC_NAME ~ 'INSTANCE to create cell(' ~
    FUNC_NAME ~ 'L ~
    FUNC_NAME ~ '$\cdot$ ~
    FUNC_NAME ~ '| BASE< -2> ← RESULT_TYPE '~ '<> with' ~
    FUNC_NAME ~ '| BASE< -1> ← this_process with' ~
    FUNC_NAME ~ '| BASE<0> ← ' ~
    FUNC_NAME ~ '| PROCESS<0>/ ' ~ LOCAL_SCOPE
    [ < N_PARAMS(I) → ]
  ]
  [ assign ENTRY.'PARAMS' to PARAM_ENTRY;
    emission PARAM_TYPE is TYPE_NAME:
      PARAM_ENTRY.'TYPE'.'NAME';
  ]
  control VAR boolean PARAM_ENTRY.'KIND' = 'VAR';
  if VAR
    then
      if KIND.I = 'VAR' | KIND.I = 'VALUE' | KIND.I = 'ACTUAL'
      then
        if PARAM_TYPE =~ TYPE.I.'NAME'
        then
          fail { VAR parameters must have identical types } endif
        else
          fail { controlled variable or }
          { expression supplied for VAR parameter } endif
        else
          if ~ASSIGNMENT_COMPATIBLE(PARAM_ENTRY.'TYPE',
            TYPE.I)
then fail { VALUE parameters not compatible }
endif
endif;
emission SELECT is UNSN: I ]
< 'with' " FUNC_NAME " _BASE< " SELECT '" ] < ' "
< VAR = PARAMS.I | DEREP(TYPE.I, KIND.I: PARAMS.I) >
>" = ' ' )', " LOCAL_SCOPE '" >
"in' " LOCAL_SCOPE '" next' "
'transfer' "
"(' " FUNC_NAME " _INSTANCE in' " LOCAL_SCOPE '" )@.' "
'FUNC NAME '" _BASE<" next' "
'=" (' " FUNC_NAME '" _INSTANCE in' " LOCAL_SCOPE '" )@.' "
" FUNC_NAME '" _BASE<" next' "
'destroy' " LOCAL_SCOPE '" >
end production
end translator:

translator SELECTOR(SCOPE: TYPE)

pattern
P "' ' | ' ' " FIELD_ID = IDENT |
"[ ' " EXP1 = EXPRESSION(SCOPE: TYPE1, KIND1) = DELIM1 = ' ' ]? |
"[ ' " EXP2 = EXPRESSION(SCOPE: TYPE2, KIND2) = DELIM2 = ' ' ]? 
end pattern
{ multiple indexing expressions are handled one at a time with the aid of the global flag INDEX_FLAG }
production
[ control PNTR boolean P = null;
 control FLDS boolean FIELD_ID = null;
 control EXPR1 boolean EXP1 = null;
 control EXPR2 boolean EXP2 = null ]
< PNTR "' ' |
< FLDS " FIELD_ID |
< EXPR1 "
[ if INDEX_FLAG = 0 
 then 
 if DELIM1 = null then assign 1 to INDEX_FLAG endif;
 assign TYPE1 to TYPE
 else fail
 endif ]
TO_INT(TYPE1, KIND1: EXP1) |
< EXPR2 "
[ if INDEX_FLAG = 1 
 then 
 if DELIM2 = ' ' then assign 0 to INDEX_FLAG endif;
 assign TYPE2 to TYPE
 else fail
 endif ]
TO_INT(TYPE2, KIND2: EXP2) 

endproduction
endtranslator;
translator VARIABLE(SCOPE: V_TYPE, V_KIND)
local TYPE, ENTRY use V
pattern
\[ \text{ROOT} \leftarrow \text{USEDID}(: \text{ROOT_ENTRY}) \]
\[ S \leftarrow \text{SELECTOR}(: \text{S_TYPE}) \]
endpattern
production
\[ \text{[ if } \text{ROOT_ENTRY.'MODE'} = 'FIELD' ] \]
\[ \text{ROOT_ENTRY.'MODE'} = 'VAR' \& \text{ROOT_ENTRY.'KIND'} = 'FUNC' \]
\[ \text{then} \]
\[ \text{assign ROOT_ENTRY.'KIND'} \rightarrow \text{V_KIND}; \]
\[ \text{assign ROOT_ENTRY.'TYPE'} \rightarrow \text{TYPE}; \]
\[ \text{control N_SELECTORS integer #S} \]
\[ \text{else fail } \{ \text{illegal mode of identifier or function name } \} \]
endif
\[ \text{ROOT} \]
\[ < \text{N_SELECTORS(I)} \rightarrow \]
\[ \{ \text{control POINTER boolean S.I} = '"'; \]
\[ \text{control INDEX boolean S.I} = \text{null} \& \text{S.TYPE.I} = \text{null} \} \]
\[ \langle \text{POINTER} \rightarrow \]
\[ \{ \text{if TYPE.'FORM'} = 'POINTER' \]
\[ \text{then assign TYPE.'BASE'} \rightarrow \text{TYPE} \]
\[ \text{else fail } \{ \text{can't dereference a non-pointer } \} \]
endif
\[ '{@}' \]
\[ < \text{INDEX} \rightarrow \]
\[ \{ \text{if TYPE.'FROM'} = 'ARRAY' \]
\[ \text{then} \]
\[ \text{if ASSIGNMENT_COMPATIBLE(TYPE.'INDEX', S.TYPE.I)} \]
\[ \text{then} \]
\[ \text{emission INDEX_TYPE is TYPE_NAME: TYPE.'INDEX'.NAME';} \]
\[ \text{assign TYPE.'BASE'} \rightarrow \text{TYPE} \]
\[ \text{else fail } \{ \text{index expression not compatible type } \} \]
endif
\[ \text{else } \{ \text{can't index non-array } \} \]
endif
\[ '{@}' \]
\[ < \text{INDEX_TYPE ^ '} \langle ^ S.I ^ '<' \rangle \} \]
\[ < \{ \text{FIELD_SELECTOR} \rightarrow \} \]
\[ \{ \text{if TYPE.'FORM'} = 'RECORD' \]
\[ \text{then} \]
\[ \text{assign TYPE.'FIELDS'(S.I) to ENTRY;} \]
\[ \text{if ENTRY = null then fail endif;} \]
\[ \text{emission RECORD_INDEX_TYPE is TYPE_NAME:} \]
\[ \text{TYPE.'INDEX'.NAME';} \]
\[ \text{emission FIELD_NO is UNSN: ENTRY.'#';} \]
\[ \text{assign ENTRY.'TYPE'} \rightarrow \text{TYPE} \]
\[ \text{else fail } \{ \text{can't select a non-record } \} \]
endif
\[ '{@}' \]
\[ < \text{RECORD_INDEX_TYPE ^ '} \langle ^ FIELD_NO ^ '<' \rangle \} \]
\[ >>> \]
\[ < \{ \text{assign TYPE to V_TYPE} \]
translator STANDARD_FUNCTION_CALL(SCOPE: TYPE)
{ must be matched after normal function call }

pattern
   SF_NAME ← IDENT ^ '(' ^
   PARAM ← EXPRESSION(SCOPE: E_TYPE, E_KIND) ^ ')' ^

end pattern

production
[ if SF_NAME = 'ORD' &
  (E_TYPE.'FORM' = 'SCALAR' | E_TYPE.'FORM' = 'SUBRANGE')
  then assign INTEGER to TYPE
  else
    if SF_NAME = 'CHR' & CONFORM(E_TYPE) = INTEGER
      then assign CHAR to TYPE
    else fail
    endif
  endif ]

TO_INT(E_KIND: PARAM)
end production
end translator;

translator SET_RANGE(: TYPE)
local SET_BASE_TYPE use V

pattern
   L ← EXPRESSION(: L_TYPE, L_KIND) ^
   < '••' ^ R ← EXPRESSION(: R_TYPE, R_KIND) >?

end pattern

production
[ assign CONFORM(L_TYPE) to SET_BASE_TYPE;
  if SET_BASE_TYPE.'FORM' = 'SCALAR' &
  SET_BASE_TYPE = CONFORM(R_TYPE)
  then assign SET_BASE_TYPE to TYPE
  else fail { not identical scalar types }
  endif
]
< 'CREATE_SET(' ^ TO_INT(L_KIND: L) ^ ', ' ^
  TO_INT(R_KIND: R) ^ ')') ^

end production
end translator;
translator SET_DENOTATION(: TYPE)
pattern
'[' ^ < SR ← SET_RANGE(: R_TYPE) / ',', ]? ^ ']' endpattern
production
[ assign #SR to N_RANGES;
  control EMPTY boolean N_RANGES = 0;
  for I from 2 to N_RANGES do
    if ~IDENTICAL(R_TYPE.I, R_TYPE.I) then fail endif
  endfor;
  assign 'SET' to TYPE.'FROM';
  if ~EMPTY then assign R_TYPE.I to TYPE.'BASE' endif ]
< 'SET_TYPE'< ^
< EMPTY →
'CREATE_SET(1,0)' |
OPERATION(N_RANGES, null : SR, 'UNION') > ^
'>' >
endproduction
endtranslator;
translator FACTOR(SCOPE: TYPE, KIND)

pattern
UC ← UNSIGNED_CONSTANT(: UC_TYPE) |
FC ← FUNCTION_CALL(SCOPE: FC_TYPE) |
V ← VARIABLE(SCOPE: V_TYPE, V_KIND) |
SFC ← STANDARD_FUNCTION_CALL(SCOPE: SFC_TYPE) |
< '(' ^ E ← EXPRESSION(SCOPE: E_TYPE, E_KIND) ^ ')') > |
< 'NOT' ^ F ← FACTOR(SCOPE: F_TYPE, F_KIND) > |
S ← SET_DENOTATION(: S_TYPE)
end pattern

production
[ control CONST boolean UC "=" null;
control CALL boolean FC "=" null;
control VAR boolean V "=" null;
control SCALL boolean SFC "=" null;
control EXP boolean E "=" null;
control NEG boolean F "=" null;
control SET boolean S "=" null ]
< CONST →
[ assign UC_TYPE to TYPE;
assign 'CONST' to KIND ]
UC |
< CALL →
[ assign FC_TYPE to TYPE;
assign 'FUNC CALL' to KIND ]
FC |
< VAR →
[ assign V_TYPE to TYPE;
if V_KIND = 'FUNC' then fail endif;
assign V_KIND to KIND ]
V |
< SCALL →
[ assign SFC_TYPE to TYPE;
assign 'STANDARD FUNC CALL' to KIND ]
SFC |
< EXP →
[ assign CONFORM(E_TYPE) to TYPE;
assign 'EXP' to KIND ]
E |
< NEG →
[ if F_TYPE "=" BOOLEAN then fail endif;
assign BOOLEAN to TYPE;
assign 'EXP' to KIND ]
< 'NOT' " TO_INT(F_KIND: F) " > |
< SET →
[ assign S_TYPE to TYPE;
assign 'EXP' to KIND ]
S
end production
end translator;
translator TERM(SCOPE: TYPE, KIND)
  local E_TYPE, J, SET, ARITHMETIC use V
  pattern
    F ← FACTOR(SCOPE: F_TYPE, F_KIND) /
    OP ← < '/*' | 'DIV' | 'MOD' | 'AND' >
  endpattern
  production
    [ control N_OPS integer #OP; 
    control SINGLE boolean N_OPS = 0 ]
    < SINGLE →
      [ assign F_TYPE.1 to TYPE;
      assign F_KIND to KIND ]
    F.1 |
    [ assign CONFORM(F_TYPE.1) to E_TYPE;
      assign 'EXP' to KIND;
      assign E_TYPE to TYPE;
      assign E_TYPE = INTEGER to ARITHMETIC ]
    < < N_OPS(I) →
      [ assign N_OPS + 1 - I to J;
      control MULT boolean ARITHMETIC & OP.J = '/*';
      control DIV boolean ARITHMETIC & OP.J = 'DIV';
      control MOD boolean ARITHMETIC & OP.J = 'MOD';
      control AND boolean E_TYPE = BOOLEAN & OP.J = 'AND';
      control INTERSECT boolean
        E_TYPE.'FORM' = 'SET' & OP.J = '/*' ]
    < MULT → 'MULTIPLY(' |
    < DIV → 'DIVIDE(' |
    < MOD → 'REMAINDER(' |
    < AND → 'AND(' |
    < INTERSECT → 'INTERSECT('
      ^ ^
      TO_INT(F_KIND.1: F.1) ^
    < N_OPS(K) →
      [ if "COMPATIBLE(E_TYPE, F_TYPE.(K+1)) then fail endif ]
    < ', ' ^ TO_INT(F_KIND.(K+1): F.(K+1)) ^ ' )' >>
  >>>>
  endproduction
endtranslator;
Appendix 6 - Description of Pascal

translator SIMPLE_EXPRESSION(SCOPE: TYPE, KIND)
local E_TYPE

pattern
S ← SIGN? " T ← TERM(SCOPE: T_TYPE, T_KIND) / 
OP ← < SIGN | 'OR' >
endpattern

production
[ control NO_SIGN boolean S = null;
control N_OPS integer #OP;
control NEG boolean S = '-';
control SINGLE boolean N_OPS = 0;
assign CONFORM(T_TYPE.1) to E_TYPE ]
<SINGLE →
< NO_SIGN →
[ assign T_TYPE.1 to TYPE;
assign T_KIND.1 to KIND ]
T.1
[ assign E_TYPE to TYPE;
assign 'EXP' to KIND;
if NEG & ~COMPATIBLE(T_TYPE.1, INTEGER) then fail endif ]
< < NEG → 'NEGATE(? > ^
TO_INT(T_KIND.1: T.1) ^
< NEG → '-')? > >
>
< assign 'EXP' to KIND;
assign E_TYPE to TYPE;
control SET boolean E_TYPE.'FORM' = 'SET';
control BOOL boolean E_TYPE = BOOLEAN;
control INT boolean E_TYPE = INTEGER ]
< SET →
[ if ~NO_SIGN then fail endif ]
< < N_OPS(I) →
[ control UNION boolean OP.(N_OPS + 1 - I) = '+' ]
< UNION → 'UNION( ! 'WITHOUT( >+ > ^
TO_INT(T_KIND.1: T.1) ^
< N_OPS(J) →
[ if OP.J = 'OR' | ~COMPATIBLE(T_TYPE.(J+1), E_TYPE)
then fail endif ]
< ' ' TO_INT(T_KIND.(J+1): T.(J+1)) ^ ' ' >+ >
>
< BOOL →
[ if ~NO_SIGN then fail endif;
for K from 1 to N_OPS do
if OP.K = 'OR' | ~COMPATIBLE(T_TYPE.(K+1), BOOLEAN)
then fail endif
endfor ]
OPERATION(N_OPS + 1, T_KIND: T, 'OR') ]
< INT →
< < N_OPS(L) →
[ if OP.L = 'OR' | ~COMPATIBLE(T_TYPE.(L+1), INTEGER) ]
then fail
endif;
control ADD boolean OP.(N_OPS + 1 - L) = '+' ]
< ADD → 'ADD(' | 'SUBTRACT(' >+ > ^
< NEG → 'NEGATE(?' > ^
TO_INT(T_KIND.: T.1) ^
< NEG → ')?' > ^
< N_OPS(M) →
<<<'
TO_INT(T_KIND.(M+1): T.(M+1)) ^ ')' >> >
endproduction
endtranslator;
Appendix 6 - Description of Pascal

translator EXPRESSION(SCOPE: TYPE, KIND)

pattern
L ← SIMPLE_EXPRESSION(SCOPE: L_TYPE, L_KIND) ^
< OP ← < '=' | '<' | '<=' | '>=' | '<' | '>' | 'IN' > ^
R ← SIMPLE_EXPRESSION(SCOPE: R_TYPE, R_KIND) >?
endpattern

production
[ control ONE boolean OP = null ]
< ONE →
[ assign L_TYPE to TYPE;
  assign L_KIND to KIND ]
L |
[ control VARL boolean L_KIND = 'VAR';
  control VARR boolean R_KIND = 'VAR';
  control STRING boolean IS_STRING_TYPE(L_TYPE);
  control POINTER boolean L_TYPE.'FORM' = 'POINTER';
  control SET boolean L_TYPE.'FORM' = 'SET';
  control INC boolean OP = 'IN';
  assign BOOLEAN to TYPE;
  assign 'EXP' to KIND;
  control SIMPLE boolean
    L_TYPE.'FORM' = 'SCALAR' | L_TYPE.'FORM' = 'SUBRANGE';
  control EQL boolean OP = '=';
  control NEQ boolean OP = '<>';
  control LSS boolean OP = '<';
  control GTR boolean OP = '>';
  control LEQ boolean OP = '<=';
  control GEQ boolean OP = '>='; ]
< INC →
[ if R_TYPE.'FORM' ~ = 'SET' |
  COMPATIBLE(R_TYPE.'BASE', L_TYPE)
  then fail endif ]
< 'MEMBER(' ^ TO_INT(L_KIND: L) ^ ',' ^
  TO_INT(R_KIND: R) ^ 'T')' > |
< POINTER →
[ if 'IDENTICAL(L_TYPE, R_TYPE) then fail endif ]
< EQL →
< L ^ < VARL → '@' > ^ '=' ^ R ^ < VARR → '@' > > |
< NEQ →
< 'NOT(' ^ L ^ < VARL → '@' > ^ '=' ^
  R ^ < VARR → '@' > > |
> > |
< SET →
[ if COMPATIBLE(L_TYPE, R_TYPE) then fail endif;
  control LGT boolean LSS | GTR;
  control LGQ boolean LEQ | GEQ;
  control REV boolean GTR | GEQ ]
< < EQL → 'EQL' |
< NEQ → 'NEQ' |
< LGT → 'PROPERLY_CONTAINED' |
< LGQ → 'CONTAINED' |
Appendix 6 - Description of Pascal

>>>>

\[ \begin{align*}
&\text{REV TO INT(R KIND: R)} \mid \text{TO_INT(L KIND: L) > ' ' '} > \\
&\text{REV TO INT(L KIND: L)} \mid \text{TO_INT(R KIND: R) > ' ' '} > \\
&\text{SIMPLE} \to \\
&\quad \text{[ if 'COMPATIBLE(L TYPE, R_TYPE) then fail endif ]} \\
&< \text{EQL} \to 'EQL' \mid \\
&\quad \text{NEQ} \to 'NEQ' \mid \\
&\quad \text{LSS} \to 'LSS' \mid \\
&\quad \text{GTR} \to 'GTR' \mid \\
&\quad \text{LEQ} \to 'LEQ' \mid \\
&\quad \text{GEQ} \to 'GEQ' \\
&\quad \text{STRING} \to \\
&\quad \quad \text{[ if 'COMPATIBLE(L TYPE, R_TYPE) then fail endif; } \\
&\quad \quad \text{emission LENGTH is UNSN: L_TYPE.'BASE'.MAX'; } \\
&\quad \quad \text{emission ST is TYPE_NAME: STRING TYPES.LENGTH; } \\
&\quad \quad \text{emission SIT is TYPE_NAME: STRING INDEX TYPES.LENGTH; } \\
&\quad \quad \text{emission BOOL is TYPE_NAME: BOOLEAN.'NAME'; } \\
&\quad \quad \text{'\leftrightarrow' \ BOOL \ [ \ 'ENTER('STRING_COMPARE'): ) \ ^
\quad \quad \text{'bind S1 to create_cell('ST','L')next'} \ ^
\quad \quad \text{'bind S2 to create_cell('ST',R')next'} \ ^
\quad \quad \text{'bind I to create_cell('SIT',SIT')next'} \ ^
\quad \quad \text{'\leftrightarrow' \ BOOL \ [ ] \ ^
&\quad \quad \text{'EQL'} \to \\
&\quad \quad \quad \text{'\text{EQL(#S1@.I @, #S2@.I @)} \to \leftrightarrow' \ BOOL \ ^\text{'<O> exit'} \ ^
&\quad \quad \quad \quad 'EQL(#I@,'LENGTH'\text{'T')} \to \leftrightarrow' \ BOOL \ ^\text{'<1> exit'} \ ^
&\quad \quad \quad \quad 'assign' \ \text{SIT} \ ^\text{'<INC(#I@)} \ \text{to I_loop'} \ ^
&\quad \quad \text{NEQ} \to \\
&\quad \quad \quad \text{'\text{NEQ(#S1@.I @, #S2@.I @)} \to \leftrightarrow' \ BOOL \ ^\text{'<O> exit'} \ ^
&\quad \quad \quad \quad 'EQL(#I@,'LENGTH'\text{'T')} \to \leftrightarrow' \ BOOL \ ^\text{'<1> exit'} \ ^
&\quad \quad \quad \quad 'assign' \ \text{SIT} \ ^\text{'<INC(#I@)} \ \text{to I_loop'} \ ^
&\quad \quad \text{LEQ} \to \\
&\quad \quad \quad \text{'\text{GTR(#S1@.I @, #S2@.I @)} \to \leftrightarrow' \ BOOL \ ^\text{'<O> exit'} \ ^
&\quad \quad \quad \quad 'EQL(#I@,'LENGTH'\text{'T')} \to \leftrightarrow' \ BOOL \ ^\text{'<1> exit'} \ ^
&\quad \quad \quad \quad 'assign' \ \text{SIT} \ ^\text{'<INC(#I@)} \ \text{to I_loop'} \ ^
&\quad \quad \text{GEQ} \to \\
&\quad \quad \quad \text{'\text{LSS(#S1@.I @, #S2@.I @)} \to \leftrightarrow' \ BOOL \ ^\text{'<O> exit'} \ ^
&\quad \quad \quad \quad 'EQL(#I@,'LENGTH'\text{'T')} \to \leftrightarrow' \ BOOL \ ^\text{'<1> exit'} \ ^
&\quad \quad \quad \quad 'assign' \ \text{SIT} \ ^\text{'<INC(#I@)} \ \text{to I_loop'} \ ^
&\quad \quad \text{LSS} \to \\
&\quad \quad \quad \text{'\text{GTR(#S1@.I @, #S2@.I @)} \to \leftrightarrow' \ BOOL \ ^\text{'<O> exit'} \ ^
&\quad \quad \quad \quad '\text{AND(EQL(#S1@.I @, #S2@.I @), LSS(#I@, 'LENGTH'\text{'T}))} \to \ ^
&\quad \quad \quad \quad 'assign' \ \text{SIT} \ ^\text{'<INC(#I@)} \ \text{to I_loop'} \ ^
&\quad \quad \quad \quad '\leftrightarrow' \ BOOL \ ^\text{'<1> } > \\
&\quad \quad \text{GTR} \to \\
&\quad \quad \quad \text{'\text{LSS(#S1@.I @, #S2@.I @)} \to \leftrightarrow' \ BOOL \ ^\text{'<O> exit'} \ ^
&\quad \quad \quad \quad '\text{AND(EQL(#S1@.I @, #S2@.I @), LSS(#I@, 'LENGTH'\text{'T}))} \to \ ^
&\quad \quad \quad \quad 'assign' \ \text{SIT} \ ^\text{'<INC(#I@)} \ \text{to I_loop'} \ ^
&\quad \quad \quad \quad '\leftrightarrow' \ BOOL \ ^\text{'<1> } > \\
\end{align*}\]
translator IF_STATEMENT(SCOPE: USED_LABELS)
    pattern
    'IF' ^^ BE <-> EXPRESSION(SCOPE: TYPE, KIND) ^^ 'THEN' ^
    TB <-> STATEMENT(SCOPE: USED_TLBS, T_LABEL) ^
    < 'ELSE' ^ FB <-> STATEMENT(SCOPE: USED_FLBS, F_LABEL) >?
    endpattern
    production
    [ if TYPE = BOOLEAN
        then
            control ELSE boolean FB ~= null;
            assign UNITE(REMOVE(USED_TLBS, T_LABEL),
                            REMOVE(USED_FLBS, F_LABEL)) to USED_LABELS
        else fail { expression not Boolean }
        endif
        ' [ ' TO_INT(KIND : BE) ^^ ' -> [ ' ^ TB ^ ' ] ' ^
        < ELSE -> < 'exit' ^ FB >? > ^']
    endproduction
endtranslator;

translator WHILE_STATEMENT(SCOPE: USED_LABELS)
    pattern
    'WHILE' ^^ BE <-> EXPRESSION(SCOPE: TYPE, KIND) ^^ 'DO' ^
    S <-> STATEMENT(SCOPE: USED_LBLS, LABEL)
    endpattern
    production
    [ if TYPE = BOOLEAN
        then assign REMOVE(USED_LBLS, LABEL) to USED_LABELS
        else fail { expression not Boolean }
        endif
        ' [ ' TO_INT(KIND : BE) ^^ ' -> [ ' ^ S ^ ' ] loop]'
    endproduction
endtranslator;
Appendix 6 - Description of Pascal

translator REPEAT_STATEMENT(SCOPE: USED_LABELS)
pattern
  'REPEAT' 'S' ← STATEMENT_SEQUENCE(SCOPE: USED_LBLS) 'UNTIL' 'BE' ← EXPRESSION(SCOPE: TYPE, KIND)
endpattern
production
  [ if TYPE = BOOLEAN
    then assign USED_LBLS to USED_LABELS
    else fail { expression not Boolean }
  endif
  'S' 'next NOT(' 'TO_INT(KIND: BE)' ')' → null loop ]
endproduction
endtranslator;

translator CONSTANT_LIST(: TYPES, VALUES)
pattern CONSTANT(: TYPE, VALUE) / '',' endpattern
production
  [ assign TYPE to TYPES;
    assign VALUE to VALUES ]
endproduction
endtranslator;
Appendix 6 - Description of Pascal

translator CASE_STATEMENT(SCOPE: USED_LABELS)
local USED_LS use V
pattern 'CASE' ^ E ⊕ EXPRESSION(SCOPE: E_TYPE, E_KIND) endpattern
production
[ if E_TYPE.'FORM' = 'SCALAR' | E_TYPE.'FORM' = 'SUBRANGE'
  then
    emission CASE_TYPE is TYPE_NAME : E_TYPE.'NAME'
  else fail { case expression type not simple } endif ]
ENTER('CASE_SCOPE': )
'bind CASE_TEMP to create_cell(' ^ CASE_TYPE ^ ', ^ CASE_TYPE ^ '<' ^ TO_INT(E_KIND: E) ^ '>') next'
endproduction
pattern
'OF' ^ < CONSTANT_LIST(: C_TYPES, C_VALUES) ^ ':' ^ S ⊕ STATEMENT('CASE_SCOPE': USED_LBLS, LABEL) > / '; ' ^ 'END'
endpattern
production
[ _control N_CASES integer #S ]
[ if ]< N_CASES(I) →
[ assign UNITE(USED_LS, REMOVE(USED_LBLS.I, LABEL.I)) to to USED_LS;
  control N_ORS integer #C_VALUES.I - 1;
  emission CVI1 is NUMBER: C_VALUES.I.1;
  if 'COMPATIBLE(E_TYPE, C_TYPE.I.1) then fail endif ]
< N_ORS → 'OR("' ^ 'EQL(' ^ CVI1 ^ ', #CASE_TEMP@) ^ 
< N_ORS(J) →
[ if COMPATIBLE(C_TYPES.I.(J+1), E_TYPE)
  then emission CVIJ is NUMBER: C_VALUES.I.(J+1)
  else fail endif ]
< ', EQL(' ^ CVIJ ^ ', #CASE_TEMP@)') >* > ^
'->' ^ S.I > / 'exit' > ^
[ assign USED_LS to USED_LABELS ]
' ] next' ^ EXIT('CASE_SCOPE': )
endproduction
endtranslator;
Appendix 6 - Description of Pascal

translator FOR_STATEMENT(SCOPE: USED_LABELS)
  local CV_ENTRY, CV_TYPE use V
  pattern
    'FOR' ^ CV ← IDENT ^ ';' ^
    LB ← EXPRESSION(SCOPE: LB_TYPE, LB_KIND) ^
    DIR ← < 'TO' | 'DOWNTO' > ^
    UB ← EXPRESSION(SCOPE: UB_TYPE, UB_KIND)
  endpattern
  production
    [ assign ST.LEVEL.CV to CV_ENTRY;
      if CV_ENTRY.'MODE' = 'VAR' & CV_ENTRY.'KIND' = 'ACTUAL'
      then
        assign CV_ENTRY.'TYPE' to CV_TYPE;
        assign 'CONTROLLED' to CV_ENTRY.'KIND';
        if (CV_TYPE.'FORM' = 'SCALAR' | CV_TYPE.'FORM' = 'SUBRANGE') &
          COMPATIBLE(CV_TYPE, LB_TYPE) & COMPATIBLE(CV_TYPE, UB_TYPE)
        then
          emission FOR_TYPE is TYPE_NAME: CV_TYPE.'NAME';
          control UP boolean DIR = 'TO'
        else fail { incompatible types }
      endif
      else fail { controlled variable not declared at current level }
    endif]
  ENTER('FOR_SCOPE': ) ^
    'bind FOR B to create cell(' ^ FOR_TYPE ^ ' ' ^
    FOR_TYPE '<' ^ TO_INT(LB_KIND :- LB) ^ ' )' next ' ^
    'bind FOR E to create cell(' ^ FOR_TYPE ^ ' ' ^
    FOR_TYPE '<' ^ TO_INT(UB_KIND :- UB) ^ ' )' next ' ^
    '[' ^ '< UP → ' 'LEQ' | 'GEQ' > ^ '('FOR_B@, #FOR_E@)-' ^
    [ assign FOR_B@ to ' CV ^ ' next'
  endproduction
  pattern
    'DO' ^ S ← STATEMENT('FOR_SCOPE': USED_LABELS, LABEL)
  endpattern
  production
    [ assign 'ACTUAL' to CV_ENTRY.'KIND';
      assign REMOVE(USED_LABELS, LABEL) to USED_LABELS ]
    '[' ^ S ^ next' ^
    'EQL(# ^ CV ^ '@', #FOR_B@) → null exit' ^
    'assign' ^ FOR_TYPE '^ '< ^ UP → 'INC' | 'DEC' > ^
    '(' # ^ CV ^ '@') to ' CV ^ 'loop' ^
    ']'] next' ^
  EXIT('FOR_SCOPE': )
  endproduction
  endtranslator;
translator WITH VARIABLE_LIST(SCOPE: INNER_MOST_SCOPE, N_VARS)
  local FIELD_NAMES, NEW_SCOPE use V
pattern VAR ← VARIABLE(SCOPE: TYPE, KIND) endpattern
production
  [ if TYPE.'FORM' = 'RECORD'
    then
      assign LEVEL + 1 to LEVEL;
      assign TYPE.'FIELDS' to ST.LEVEL;
      assign new(SCOPE_NAME) to NEW_SCOPE;
      control N_FIELDS integer TYPE :=
        1<br> 1<br> 297<br> 208<br> 802<br><br> 208<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809<br> 802<br> 809

endif ]
ENTER(NEW_SCOPE: ) ^ [' ^
  < N_FIELDS(I) →
  [ emission FIELD_NO is UNSN: I;
    emission FIELD_NAME is IDENT: FIELD_NAMES.I ]
  < 'bind' ^ FIELD_NAME ^ 'to' ^ VAR ^ '.' ^
    FIELD_TYPE ^ '[' ^ FIELD_NO ^ '>' > / 'with' > ^
  ] next' -
endproduction
pattern
  < ',' ^ VAR_LIST ← WITH VARIABLE_LIST(NEW_SCOPE:
    INNER_SCOPE, N_VAR) >?
endpattern
production
  [ control MORE boolean VAR_LIST ^= null ]
  < MORE →
    [ assign INNER_SCOPE to INNER_MOST_SCOPE;
      assign N_VAR + 1 to N_VARS ]
  VAR_LIST |
  [ assign NEW_SCOPE to INNER_MOST_SCOPE;
    assign 1 to N_VARS ]
null >
endproduction
endtranslator;
translator WITH_STATEMENT(SCOPE: USED_LABELS)

  pattern
    'WITH' ^ WVS \<rightarrow\> WITH_VARIABLE_LIST(SCOPE: NEW_SCOPE, N_VARS) / ','
    ^ 'DO' ^ S \<rightarrow\> STATEMENT(NEW_SCOPE: USED_LBLs, LABEL)

endpattern

production
  [ assign REMOVE(USED_LBLs, LABEL) to USED_LABELS;
    for I from LEVEL - N_VARS + 1 to LEVEL do
      assign null to ST.I
    endfor;
    assign LEVEL - N_VARS to LEVEL;
    emission CURRENT_SCOPE is SCOPE_NAME: SCOPE ]
  WVS ^ S ^ 'next'
  'destroy_enclosed' ^ CURRENT_SCOPE
endproduction

endtranslator;

translator GOTO_STATEMENT(: USED_LABELS)

  pattern 'GOTO' ^ LABEL \<--\> UNSN endpattern

production
  [ if LABELS.BLEVEL.LABEL ^= null then
    emission TARGET is LABEL_NAME: LABELS.BLEVEL.LABEL.'NAME';
    assign 1 to USED_LABELS.LABEL
  else fail { label not declared } endif ]
  'assign type_of(this_process)<' ^ TARGET ^ '>' to this_process'
endproduction

endtranslator;
translator ASSIGNMENT_STATEMENT(SCOPE: )
local LHS_TYPE, FORM use V
pattern
LHS (~< VARIABLE(SCOPE: V_TYPE, V_KIND) \ userid(:, LHS_ENTRY) ~> 
'=' ^ RHS (~< EXPRESSION(SCOPE: RHS_TYPE, RHS_KIND)
end pattern
production
[ if LHS_ENTRY ~ null
   then
     if LHS_ENTRY.'MODE' = 'VAR' & LHS_ENTRY.'KIND' = 'FUNC'
     then
        { assigning to function name in function body }
        assign LHS_ENTRY.'TYPE' to LHS_TYPE
     else fail { any other case is an error }
     endif
   else
     if V_KIND = 'CONTROLLED' then fail endif;
     assign V_TYPE to LHS_TYPE
   endif;
   if 'ASSIGNMENT_COMPATIBLE(LHS_TYPE, RHS_TYPE) then fail endif ]
ENTER('TEMP_SCOPE': ) ~
'bind L H S to create_cell(cell, ' LHS ') next' ~
'assign' ^ DEREF(RHS_TYPE, RHS_KIND: RHS) ^ 'to' ^ L_H_S@ ^ 'next'
EXIT('TEMP_SCOPE': )
end production
end translator;
Appendix 6 - Description of Pascal

translator CALL_STATEMENT(SCOPE: )
local PARAM_ENTRY use V
pattern
PROC_NAME ← USEID(: ENTRY) ^
< '(' ^ PARAMS ← EXPRESSION(SCOPE: TYPE, KIND) / ',' ^ ')' >?
endpattern
production
[ if ENTRY.'MODE' = 'PROC'
    then
        control N_PARAMS integer #PARAMS;
        if N_PARAMS = ENTRY.'#'
            then
                emission LOCAL_SCOPE is SCOPE_NAME: ENTRY.'LOCAL SCOPE';
                emission ENC_SCOPE is SCOPE_NAME: ENTRY.'ENC SCOPE'
            else fail { incorrect number of arguments to procedure } endif
        else fail { identifier not procedure name } endif ]
endif
< 'bind' ^ LOCAL_SCOPE ^ 'to create basket next' ^
'enclosing' ^ LOCAL_SCOPE ^ 'is' ^ ENC_SCOPE ^ 'next' ^
'bind' ^ PROC_NAME ^ 'INSTANCE to create cell' ^
PROC_NAME ^ '(' ^
PROC_NAME ^ '_BASE<1> ← this_process with' ^
PROC_NAME ^ '_BASE<0> ←' ^
PROC_NAME ^ '_PROCESS<0>/' ^ LOCAL_SCOPE ^
< N_PARAMS(I) →
[ assign ENTRY.'PARAMS' to PARAM_ENTRY;
    emission PARAM_TYPE is TYPE_NAME :
        PARAM_ENTRY.'TYPE'. 'NAME';
    control VAR boolean PARAM_ENTRY.'KIND' = 'VAR';
    if VAR
        then
            if KIND.I = 'VAR' | KIND.I = 'VALUE' |
                KIND.I = 'ACTUAL'
                then
                    if PARAM_TYPE = TYPE.I.'NAME'
                        then
                            fail
                            { VAR parameters must have identical types }
                        endif
                    else
                        fail
                        { controlled variable or } [ expression supplied for VAR parameter ]
                    endif
                else
                    fail
                    { VALUE parameters not compatible }
                endif
        endif
    endif
    emission SELECT is UNSN: I ]
Appendix 6 - Description of Pascal

< 'with' ^ PROC_NAME ^ '_BASE<^ SELECT ^ '> ^ ' ^
< VAR -> PARAMS.I | DEREF(TYPE.I, KIND.I: PARAMS.I) >
>* ^ ']', ^
LOCAL_SCOPE ^ ')' ^
'in' ^ LOCAL_SCOPE ^ 'next' ^
'transfer' ^_LOCALSECope ^ 'next'
'(' ^ PROC_NAME ^ '_INSTANCE in' ^ LOCAL_SCOPE ^ ')@.' ^
PROC_NAME ^ '_BASE<> next' ^
'destroy' ^ LOCAL_SCOPE >
end production
endtranslator;

translator STANDARD_PROCEDURE_CALL(SCOPE: )

pattern
SP_NAME <- IDENT ^ '(': ^
PARAM <- VARIABLE(SCOPE: TYPE, KIND) ^ ')
endpattern

production
[ control NEW boolean SP_NAME = 'NEW';
control DIS boolean SP_NAME = 'DISPOSE';
if TYPE.'FORM' = 'POINTER' & KIND ^ 'FUNC' then
  emission ALLOC_TYPE is TYPE_NAME: TYPE.'BASE'.NAME
else fail { non-pointer variable used }
endif ]

< NEW ->
< 'assign' create_cell(' ^ ALLOC_TYPE ^ ', ^
ALLOC_TYPE ^ '?', HEAP_AREA) to' ^ PARAM > |

< DIS ->
< 'assign no_cell to ' ^ PARAM >

>>
end production
endtranslator;
translator UNLABELLED_STATEMENT(SCOPE: USED_LABELS)

pattern
'BEGIN' ^ SS <- STATEMENT_SEQUENCE(SCOPE: USED_SSL) ^ 'END' |
IF <- IF_STATEMENT(SCOPE: USED_IFL) |
CASE <- CASE_STATEMENT(SCOPE: USED_CASEL) |
WHILE <- WHILE_STATEMENT(SCOPE: USED_WHILEL) |
REPEAT <- REPEAT_STATEMENT(SCOPE: USED_REPEATL) |
FOR <- FOR_STATEMENT(SCOPE: USED_FORL) |
WITH <- WITH_STATEMENT(SCOPE: USED_WITHL) |
GOTO <- GOTO_STATEMENT(SCOPE: USED_GOTOL) |
ASSIGN <- ASSIGNMENT_STATEMENT(SCOPE: ) |
CALL <- CALL_STATEMENT(SCOPE: ) |
SPC <- STANDARD_PROCEDURE_CALL(SCOPE: ) |
null
endpattern

production
[
control CMPND boolean SS ^= null;
control IFS boolean IF ^= null;
control CASES boolean CASE ^= null;
control WHILES boolean WHILE ^= null;
control REPEATS boolean REPEAT ^= null;
control FORS boolean FOR ^= null;
control WITHS boolean WITH ^= null;
control GOTOS boolean GOTO ^= null;
control ASSIGNS boolean ASSIGN ^= null;
control CALLS boolean CALL ^= null;
control SPCS boolean SPC ^= null ]
< CMPND -> [ assign USED_SSL to USED_LABELS ] SS |
< IFS -> [ assign USED_IFL to USED_LABELS ] IF |
< CASES -> [ assign USED_CASEL to USED_LABELS ] CASE |
< WHILES -> [ assign USED_WHILEL to USED_LABELS ] WHILE |
< REPEATS -> [ assign USED_REPEATL to USED_LABELS ] REPEAT |
< FORS -> [ assign USED_FORL to USED_LABELS ] FOR |
< WITHS -> [ assign USED_WITHL to USED_LABELS ] WITH |
< GOTOS -> [ assign USED_GOTOL to USED_LABELS ] GOTO |
< ASSIGNS -> ASSIGN |
< CALLS -> CALL |
< SPCS -> SPC |

endproduction

endtranslator;
translator STATEMENT(SCOPE: USED_LABELS, DEF_LABEL)
local ENTRY use V
pattern
< LABEL ← UNSN ° ' ' > ? ^
S ← UNLABELLED_STATEMENT(SCOPE: USED_LBLS)
endpattern
production
\[
\text{control LABELLED boolean LABEL} \neq \text{null;}
\text{assign USEDLBLS to USED_LABELS;}
\text{emission CURRENT_SCOPE is SCOPE_NAME: SCOPE }
\]< LABELLED \to
\[
\text{assign LABELS.BLEVEL.LABEL to ENTRY;}
\text{if ENTRY} \neq \text{null then}
\text{if ENTRY.'STATUS'} = \text{'DECLARED'}
\text{then assign 'DEFINED' to ENTRY.'STATUS'}
\text{else fail \{ multiple definition of label \}}
\text{endif;}
\text{assign LABEL to DEF_LABEL;}
\text{emission TARGET is LABEL_NAME: ENTRY.'NAME'}
\text{else fail \{ label not declared \}}
\text{endif }
\]< '/' \^ TARGET \^ ': destroy_enclosed' \^ CURRENT_SCOPE ^ '
\] next' > ?
^ S
endproduction
endtranslator;

translator STATEMENT_SEQUENCE(SCOPE: USED_LABELS)
local LS use V
pattern S ← STATEMENT(SCOPE: USED_LBLS, LABEL) / ';' endpattern
production
[ control N integer \#S ]
< N(I) \to
[ assign UNITE(LS, USEDLBLS.I) to LS ]
S.I / 'next' > ^
[ for J from 1 to N do
assign REMOVE(LS, LABEL.J) to LS
\{ remove labels declared in statement sequence from list of
labels used in constituent statements \}
endfor;
assign LS to USED_LABELS ]
endproduction
endtranslator;
Appendix 6 - Description of Pascal

translator BLOCK(SCOPE: )
  local TEMP, ENTRY use V
  pattern
    LABEL_DEC* ^
    CONST_DEC* ^
    TYPES ← < TYPE_DEC* > ^
    VARIABLES ← < VAR_DEC* > ^
    PROCS ← < < PROCEDURE_DEC(SCOPE: ) | FUNCTION_DEC(SCOPE: ) >* > ^
    'BEGIN' ^
    STATEMENTS ← STATEMENT_SEQUENCE(SCOPE: USED_LABELS) ^
    'END'
  endpattern

production
  [ if ¬EMPTY(USED_LABELS)
    then fail { label(s) used incorrectly }
    endif;
    assign !(LABELS.BLEVEL) to TEMP;
    for I from 1 to #TEMP do
      if LABELS.BLEVEL.(TEMP.I)."STATUS" = 'DEFINED'
      then fail { label(s) declared but not defined or used }
      endif
    endfor;
    assign null to LABELS.BLEVEL; { clear labels from current level
    assign !(POINTERS.BLEVEL) to TEMP;
    for I from 1 to #TEMP do
      assign SEARCH_ID(TEMP.I) to ENTRY;
      if ENTRY."MODE" = 'TYPE'
      then
        assign ENTRY."TYPE" to POINTERS.BLEVEL.(TEMP.I)."BASE" { set base types of pointer types }
      else fail { "pointer" type not pointing to type }
      endif
    endfor;
    assign null to POINTERS.BLEVEL ]
  TYPES ^ VARIABLES ^ PROCS ^ STATEMENTS
endproduction
endtranslator;
translator PASCAL_PROGRAM distinguished

pattern 'PROGRAM' ^ IDENT ^ ';' endpattern

production

[ emission CHAR_TYPE is TYPE_NAME: CHAR.'NAME';
  emission CHAR_SET_MAX is UNSN: CHAR_SET_SIZE - 1;
  emission INT_TYPE is TYPE_NAME: INTEGER.'NAME';
  emission M_MAXINT is NUMBER: -MAXINT;
  emission P_MAXINT is NUMBER: MAXINT;
  emission BOOL_TYPE is TYPE_NAME: BOOLEAN.'NAME';
  emission SET_SIZE is UNSN: 2 % CHAR_SET_SIZE;
  emission GLOBAL_SCOPE is SCOPE_NAME: new(SCOPE_NAME) ]

'external CREATE_SET(2), UNION(2), NOT(1), MULTIPLY(2),
'DIVIDE(2), REMAINDER(2), AND(2), INTERSECT(2), NEGATE(1), ' ^
'WITHOUT(2), OR(2), ADD(2), SUBTRACT(2), MEMBER(2), EQ(2), ' ^
'NEQ(2), PROPERLY_CONTAINED(2), CONTAINED(2), LSS(2), ' ^
'GTR(2), LEQ(2), EQQ(2), INC(1), DEC(1) next'

'[ bind STANDARD_NAMES to current_basket next'

'bind HEAP_AREA to create_basket next'

{ set up predefined types at level 0 }

'bind' ^ CHAR_TYPE ^ 'to interval(0,' ^ CHAR_SET_MAX ^ ') next'

'bind' ^ INT_TYPE ^ 'to interval(' ^ M_MAXINT ^ ',-1' ^

'P_MAXINT ^ ')) next'

'bind' ^ BOOL_TYPE ^ 'to interval(0,1) next'

'bind SET_TYPE to interval(0,' ^ SET_SIZE ^ ') next'

'bind' ^ GLOBAL_SCOPE ^ 'to create_basket next'

'current' ^ GLOBAL_SCOPE ^ 'next'

'enclosing' ^ GLOBAL_SCOPE ^ 'is STANDARD_NAMES next'

endproduction

pattern B ← BLOCK(GLOBAL_SCOPE: ) ^ '.' endpattern

production

[ assign !STRING_TYPES to SLS;
  control N_STR integer #SLS ]

< N_STR(I) →

[ emission SL is UNSN: SLS.I;
  emission SITN is TYPE_NAME: STRING_INDEX_TYPES.SL;
  emission STN is TYPE_NAME: STRING_TYPES.SL ]

< 'bind' ^ SITN ^ 'to interval(1,' ^ SL ^ ') next'

'bind' ^ STN ^ 'to structure(' ^ SITN ^ ');

'CHAR_TYPE ^ ') next'

>]* > ^

B ^ 'next'

'destroy' ^ LOCAL_SCOPE ^ 'next'

'destroy HEAP_AREA next'

'destroy STANDARD_NAMES ]'

endproduction

endtranslator
Appendix 7 - Description of Rendezvous

APPENDIX SEVEN

This rendezvous method of task synchronization without passing is described using two null processes, V and U and an ordinal type variable, N, with range -1 to N, where N is the maximum number of calling processes allowed to wait for a single entry. V and U may be described as two processes, each containing the statements:

```
while true do
  
  if V = 0 then
    (increase counter)
  if U = 0 then
    (increase counter)

  if V = 0 then
    (decrease counter)
  if U = 0 then
    (decrease counter)
```

APPENDIX SEVEN
A DESCRIPTION OF RENDEZVOUS

1.1

The first sub-section below describes the "rendezvous" method of task synchronization without queuing using GLADYS process control and synchronization primitives. The second describes the same method but with queuing as used in ADA [IchbJ79]. The notation used to specify the algorithms is felt to be universal and is augmented with the addition of the 'stop', 'start', 'wait', 'transfer' and 'running' primitives of GLADYS. Task and process are used as synonymous terms in this appendix.

1.1.1

The rendezvous method of task synchronization without queuing is described using two null processes, W and S and an ordinal type variable, C, with range -1 to N, where N is the maximum number of calling processes allowed to wait for a single entry. W and S may be described as two processes, each containing the statement:

while true do endwhile

W is used to synchronize the calling processes, generically named Ti, and the called process, R. S is used as a semaphore to "protect" C which indicates the number of calling processes currently waiting. C having a value of -1 indicates that R is waiting. CALLER is a variable in R which contains a reference to the calling process which is taking part in the rendezvous. Initially C is 0, W is running and S is stopped. The code for a Ti is:

wait S;
C := C + 1;  { Increment count }
if C = 0
then  { R is waiting }
   stop S;  { Clear semaphore }
   stop W  { Allow R to proceed }
else  { R is not waiting }
   stop S;  { Clear semaphore }
   wait W  { Wait }
endif;
{ Pass any parameters to R }
R.CALLER := this_process;  { Give R reference to self }
wait R;  { Wait for R to start entry code }
stop this_process  { Wait for R to finish entry code }

The code for R is:
Appendix 7 - Description of Rendezvous

wait $S$;
$C := C - 1$;  \{ Decrement counter \}
if $C >= 0$
    then \{ A $Ti$ is waiting \}
        stop $S$;  \{ Clear semaphore \}
        stop $W$  \{ Start a $Ti$ \}
    else \{ No $Ti$'s are waiting \}
        stop $S$;  \{ Clear semaphore \}
        wait $W$ \{ Wait \}
endif;
stop this_process;  \{ Wait for $Ti$ to send parameters \}
\{ Execute entry code in mutual exclusion \}
wait CALLER  \{ Continue after entry code \}

1.1.2

The rendezvous method with queuing requires, in the following model, each entry to have associated with it one null process $S$, two pointer (cell) type variables, $F$ and $B$, and a Boolean variable $P$ which indicates whether $R$ is waiting. $S$ acts as a semaphore again, this time "protecting" $P$ and the two pointer variables which form a queue of processes. In addition each calling process, $Ti$, is required to have a pointer variable, $L$, to link the waiting calling processes. The variable CALLER in $R$ is also required here. Initially $F$ is null, $B$ points to $F$ and $S$ is stopped. The code for a $Ti$ is:

wait $S$;
if $P$
    then \{ $R$ is waiting \}
        $P := false$;  \{ Not any more! \}
        stop $S$ \{ Clear semaphore \}
    else \{ $R$ is not waiting \}
        $B := this\_process$;  \{ Enqueue this process \}
        $B := this\_process$;
        $L := null$;
        stop $S$, $this\_process$ \{ Clear semaphore and wait \}
endif;
\{ Pass any parameters to $R$ \}
transfer $R$ \{ "call" $R$ \}

The code for $R$ is:
wait S;
if F = null
    then { No Ti's are waiting }
            P := true; { R waits }
            stop S, this_process { Clear semaphore and wait }
    else { A Ti is waiting }
            CALLER := F; { Dequeue it }
            F := CALLER.L;
            if F = null
                then
                    B := address(F)
                endif;
            stop S; { Clear semaphore }
            transfer CALLER { Get caller to send any parameters }
        endif;
{ Execute entry code in mutual exclusion }
start CALLER { Rendezvous is over }
Appendix 8 - Description of LISP

APPENDIX EIGHT

The description of LISP (MacAdams, Weyhrauch) given in this appendix in the "Application section of MOD" is intended as follows: It is based on two functions, EVAL and APPLY. EVAL is applied to the input expression, the LISP program, and the results are then stored in the output of the program. The translation of the result expression in the syntax which is not known in system LISP is not done. The code which is in LISP must be written in LISP with external routines which are not made.

Some characteristics of the description below are:

- Arguments to the functions EVAL and APPLY are passed by setting up the relevant knowledge in a new bucket with the function is called and setting this new bucket as the current bucket for the process, and listing the called function. The arguments to EVAL are the input expression and APPLY, the environment. Whatever the input argument of APPLY, the arguments to EVAL are in the environment. The input argument of APPLY, the environment.

- Cells and lists in storage are treated in the current bucket, which is used in most LISP objects are contained in a separate bucket called STORE which is never destroyed.

- Lists are modelled in a straightforward way by the recursive type A储能.

- Elementary list words (L) are included in the description although they are not part of MOD. It is felt that they are not when considered because of their use in the language.
A SYNTRAN/GLADYS DESCRIPTION OF LISP

1. The description of LISP [McCaJ62, MartJ77] given in this appendix is the "applicative subset of LISP" as defined in [AlleJ78]. It is based on two functions, EVAL and APPLY. EVAL is applied to the input S-expression, the LISP program, and the S-expression which results is the output of the program. The translation of the result S-expression to the outside world is not defined here. Such an output routine would have to be written in GLADYS with external routines which accept codes to write out left and right parentheses, a dot, a space and the external representation of the atoms. Atoms are represented internally simply as the interval type ATOM.

1.2

Some characteristics of the description below are:

1. Arguments to the functions EVAL and APPLY are passed by setting up the relevant bindings in a new basket when the function is called and setting this new basket as the current basket for the process modelling the called function. The arguments to EVAL are EXP, the input S-expression and ENVIRON, the environment, that is, the list of current atom/S-expression bindings. The arguments to APPLY are FN, the S-expression specifying the function being applied, ARGS, the list of arguments to FN, and ENVIRON, the environment.

2. Cells used as temporary storage are created in the current basket, cells used to model LISP objects are contained in a separate basket called STORE which is never destroyed.

3. Lists are modelled in a straightforward way by the recursive type S_EXPR.

4. Comments in braces ({{}}) are included in the description although they are not part of GLADYS. It is felt that they do not cause confusion because of this.
Appendix 8 - Description of LISP

set LETTER < 'A' .. 'Z' >,
    DIGIT < '0' .. '9' >,
    IGNORE < ' ', eol >;

token ATOM < LETTER+ >,
    NUMBER < DIGIT+ >;

ignore IGNORE* endignore;

global ST, ATOM_COUNT; {Symbol table and count of atoms encountered}

{ Set up "primitive" atoms }
assign 0 to ST.'NIL';
assign 1 to ST.'T';
assign 2 to ST.'ATOM';
assign 3 to ST.'NULL';
assign 4 to ST.'CAR';
assign 5 to ST.'CDR';
assign 6 to ST.'EQ';
assign 7 to ST.'CONS';
assign 8 to ST.'LAMBDA';
assign 9 to ST.'QUOTE';
assign 10 to ST.'COND';

assign 11 to ATOM_COUNT;
Appendix 8 - Description of LISP

translator LISP_PROGRAM distinguished

pattern $S \leftarrow S_{\text{EXPRESSSION}}$ endpattern

\[
\begin{align*}
\text{production} & \quad [ \text{emission NOA is NUMBER: ATOM\_COUNT - 1} ] \\
& \quad \text{bind BASE to interval}(0,1) \text{ next} \\
& \quad \text{bind ATOM to interval}(0, ^\text{NOA} ^\text{'} \text{'}) \text{ next} \\
& \quad [ \text{bind PAIR to structure(BASE; S\_EXPR) with} \\
& \quad \text{bind S\_EXPR to union(ATOM, PAIR) } ] \text{ next}
\end{align*}
\]

\{ The EVAL function \}
bind EVAL\_PROCESS to process[

\[
\begin{align*}
[ & \text{ATOM(\text{EXPR@})} ] \\
& \{ \text{EXP is an atom} \} \\
& \{ \text{Constants NIL and T} \} \\
& [ \#\text{ATOM(\text{EXPR@})} = 0 \rightarrow \text{bind RESULT to NIL exit} \\
& \#\text{ATOM(\text{EXPR@})} = 1 \rightarrow \text{bind RESULT to T exit}
\end{align*}
\]

\{ The other primitive atoms do not have normal bindings \}
#\text{ATOM(\text{EXPR@})} = 2 \rightarrow \text{error exit}
#\text{ATOM(\text{EXPR@})} = 3 \rightarrow \text{error exit}
#\text{ATOM(\text{EXPR@})} = 4 \rightarrow \text{error exit}
#\text{ATOM(\text{EXPR@})} = 5 \rightarrow \text{error exit}
#\text{ATOM(\text{EXPR@})} = 6 \rightarrow \text{error exit}
#\text{ATOM(\text{EXPR@})} = 7 \rightarrow \text{error exit}
#\text{ATOM(\text{EXPR@})} = 8 \rightarrow \text{error exit}
#\text{ATOM(\text{EXPR@})} = 9 \rightarrow \text{error exit}
#\text{ATOM(\text{EXPR@})} = 10 \rightarrow \text{error exit}

\{ For all other atoms, do a linear search of the environment \\
and return the bound S-expression or an error \}
bind ENV\_PTR to create cell(cell,ENVIRON) next

[ \text{ATOM(ENV\_PTR@)} ] \rightarrow \text{error exit}
#\text{ATOM(\text{PAIR<ENV\_PTR@>,BASE<O>},BASE<O>)} =
#\text{ATOM(\text{EXP@})} \rightarrow \\
\text{bind RESULT}
\text{to PAIR<ENV\_PTR@>,BASE<O>,BASE<1> exit}
assign PAIR<ENV\_PTR@>,BASE<1> to ENV\_PTR@ loop
]

\}

\{ EXP is a list \}
bind FN to PAIR<\text{EXPR@},BASE<O>} next
bind ARGS to PAIR<\text{EXPR@},BASE<1>} next

\[
\begin{align*}
[ & \text{ATOM(FN@)} \rightarrow \#\text{ATOM(FN@)} = 8 \text{ exit } \rightarrow \text{0} ] \rightarrow \\
& \{ \text{EXP is a LAMBDA expression, i.e. a "constant" } \} \\
& \text{bind RESULT to EXP@ exit}
\end{align*}
\]

\[
\begin{align*}
[ & \text{ATOM(FN@)} \rightarrow \#\text{ATOM(FN@)} = 9 \text{ exit } \rightarrow \text{0} ] \rightarrow \\
& \{ \text{EXP is (QUOTE ARG), return ARG un-evaluated } \}
\end{align*}
\]
bind RESULT to PAIR<ARGS@>.BASE<O> exit

{ Set up pointer to baskets to be used below } bind SCOPES to create_cell(cell,no_cell) next

[#ATOM(FN@)→ ←#ATOM<FN@>=10 exit ←0]→
EXP is a COND form

{ Set up pointers to COND list form and predicate results }
bind PAIR_PTR to create_cell(cell,ARGS) next
bind PRED to create_cell(cell,no_cell) next

[ATOM(PAIR_PTR@)→ error exit ] No more predicates }

{ Evaluate predicate }
assign create_basket to SCOPES@ next
enclosing SCOPES@ is current_basket next
bind EXP
to create_cell(cell, PAIR<PAIR<PAIR_PTR@@>.BASE<O@>.BASE<O>, SCOPES@) in SCOPES@ next
bind EVAL_FUNCTION to
create_cell(EVAL,

EVAL[BASE<O> ←this_process with
BASE<1> ←EVAL_PROCESS<O>/SCOPES@],
SCOPES@) in SCOPES@ next
transfer PAIR<(EVAL_FUNCTION in SCOPES@@).BASE<1> next
assign (RESULT in SCOPES@) to PRED@ next
destroy SCOPES@ next

[#ATOM(PRED@@)→ ←#ATOM<PRED@@>=0 exit ←0]→
{ Predicate evaluates to NIL, go to next one }
assign PAIR<PAIR_PTR@@>.BASE<1> to PAIR_PTR@ loop

{ Predicate evaluates to non-NIL, evaluate corresponding
S-expression }
bind SCOPE to create_basket next
enclosing SCOPE is current_basket next
bind EXP
to create_cell(cell, PAIR<PAIR<PAIR_PTR@@>.BASE<O@>.BASE<1>,
SCOPE) in SCOPE next
bind EVAL_FUNCTION to
create_cell(EVAL,

EVAL[BASE<O> ←this_process with
BASE<1> ←EVAL_PROCESS<O>/SCOPE],
SCOPE) in SCOPE next
transfer PAIR<(EVAL_FUNCTION in SCOPE@@).BASE<1> next
{ Pass back result }
bind RESULT to (RESULT in SCOPE) next
destroy SCOPE ]
] exit

{ EXP is a list but is not a LAMBDA, QUOTE or COND form }
Appendix 8 - Description of LISP

{ Evaluate arguments }
bind ARG PTR to create_cell(cell,ARGS) next
bind EVALD_ARGS to create_cell(S_EXPR,NIL,STORE) next
bind EV ARG PTR to EVALD_ARGS next
[ ATOM(ARG_PTR@) → exit
  assign create_basket to SCOPES@ next
  enclosing SCOPES@ is current_basket next
  bind EXP to create_cell(cell,
    PAIR<ARG_PTR@>.BASE<O>,
    SCOPES@) in SCOPES@ next
bind EVAL_FUNCTION to
  create_cell(EVAL,
    EVAL[BASE<O> ← this_process with
    BASE<1> ← EVAL_PROCESS<O>/SCOPES@],
    SCOPES@) in SCOPES@ next
transfer PAIR<(EVAL_FUNCTION in SCOPES@)@>.BASE<1> next
assign S_EXPR<
  PAIR[BASE<O> ← create_cell(S_EXPR,
    RESULT in SCOPES@,
    STORE) with
    BASE<1> ← create_cell(S_EXPR,NIL,STORE)>]
  to EV_ARG_PTR@ next
destroy SCOPES@ next
assign PAIR<ARG_PTR@>.BASE<1> to ARG_PTR@ next
assign PAIR<EV_ARG_PTR@>.BASE<1> to EV_ARG_PTR@ loop
]
next

{ Apply function FN to evaluated arguments }
bind SCOPE to create_basket next
enclosing SCOPE is current_basket next
bind ARGS to EVALD_ARGS in SCOPE next
bind APPLY_FUNCTION to
  create_cell(APPLY,
    APPLY[BASE<O> ← this_process with
    BASE<1> ← APPLY_PROCESS<O>/SCOPE],
    SCOPE) in SCOPE next
transfer PAIR<(APPLY_FUNCTION in SCOPE)@>.BASE<1> next
{ Pass back result }
bind RESULT to (RESULT in SCOPE) next
destroy SCOPE
]
next

{ Return to caller }
transfer PAIR<!EVAL@>.BASE<O>@
]
next

{ End of EVAL function }
bind EVAL to structure(BASE; cell, EVALPROCESS) next

{ The APPLY function }
bind APPLY_PROCESS to process[}
NIL and T are not functions }

#ATOM(FN@) → #ATOM<FN@>=0 exit ←0] → error exit

#ATOM(FN@) → #ATOM<FN@>=1 exit ←0] → error exit

#ATOM(FN@) → #ATOM<FN@>=2 exit ←0] →

| ATOM |
| bind ARG1 to PAIR<ARGS@>,BASE<O> next
| #ATOM(ARG1@) → #ATOM<ARG1@>=0 exit ←0] →
| bind RESULT to T exit
| bind RESULT to NIL ] exit

#ATOM(FN@) → #ATOM<FN@>=3 exit ←0] →

| NULL |
| bind ARG1 to PAIR<ARGS@>,BASE<O> next
| #ATOM(ARG1@) → #ATOM<ARG1@>=0 exit ←0] →
| bind RESULT to T exit
| bind RESULT to NIL ] exit

#ATOM(FN@) → #ATOM<FN@>=4 exit ←0] →

| CAR |
| bind ARG1 to PAIR<ARGS@>,BASE<O> next
| bind RESULT to PAIR<ARG1@>,BASE<O> ] exit

#ATOM(FN@) → #ATOM<FN@>=5 exit ←0] →

| CDR |
| bind ARG1 to PAIR<ARGS@>,BASE<O> next
| bind RESULT to PAIR<ARG1@>,BASE<O> ] exit

#ATOM(FN@) → #ATOM<FN@>=6 exit ←0] →

| EQ |
| bind ARG1 to PAIR<ARGS@>,BASE<O> next
| bind ARG2 to PAIR<PAIR<ARGS@>,BASE>O>,BASE<O> next
| #ATOM(ARG1@) → #ATOM<ARG2@> exit ←0] →
| [ #ATOM<ARG1@>=#ATOM<ARG2@> → bind RESULT to T exit
| bind RESULT to NIL ] exit
| error
| exit

#ATOM(FN@) → #ATOM<FN@>=7 exit ←0] →

| CONS |
| bind ARG1 to PAIR<ARGS@>,BASE<O> next
| bind ARG2 to PAIR<PAIR<ARGS@>,BASE>O>,BASE<O> next
| bind RESULT to
| create_cell(S_EXPR,
  S_EXPR<PAIR [BASE<O> ← ARG1@, with
  BASE1> ← ARG2@],
  STORE]) exit

| LAMBDA, QUOTE and COND are special forms which are
| evaluated in EVAL ]

#ATOM(FN@) → #ATOM<FN@>=8 exit ←0] → error exit
Appendix 8 - Description of LISP

```
#ATOM(FN@) ← #ATOM<FN@>=9 exit ←0] → error exit
#ATOM(FN@) ← #ATOM<FN@>=10 exit ←0] → error exit

#PAIR(FN@)←
←[[bind FIRST to PAIR<FN@>.BASE<O> next
←[ATOM(FIRST@) ← #ATOM<FIRST@>=8 exit ←0]] exit
←0]→
{ Function is LAMBDA expression, augment environment with
formals bound to (evaluated) arguments }
[bind FORMAL_PTR
to create_cell(cell,
    PAIR<PAIR<FN@>.BASE<1>@@>.BASE<O>) next
bind ARG_PTR to create_cell(cell,ARGS) next
bind NEW_ENV to create_cell(S_EXPR,NIL@,STORE) next
bind NEW_ENV_PTR to create_cell(cell,NEW_ENV) next
[ATOM<FORMAL_PTR@@>→
[#ATOM<FORMAL_PTR@@>=0→ error] exit
assign S_EXPR<-
PAIR[←BASE<O>
create_cell(
    S_EXPR,
    S_EXPR<
    PAIR[BASE<O> ←PAIR<FORMAL_PTR@@>.BASE<O> with
        BASE<1> ←PAIR<ARG_PTR@@>.BASE<O>],
    STORE) with
    BASE<1> ←
    create_cell(S_EXPR,NIL@,STORE)]>
to NEW_ENV_PTR@ next
assign PAIR<FORMAL_PTR@@>.BASE<1> to FORMAL_PTR@ next
assign PAIR<ARG_PTR@@>.BASE<1> to ARG_PTR@ next
assign PAIR<NEW_ENV_PTR@@>.BASE<1> to NEW_ENV_PTR@ loop
] next
assign ENVIRON@ to NEW_ENV_PTR@@ next
{ Evaluate LAMBDA expression body in new environment }
bind SCOPE to create_basket next
enclosing SCOPE is current_basket next
bind EXP
to create_cell(cell,
    PAIR<PAIR<FN@>.BASE<1>@>.BASE<1>@>.BASE<O>,
    SCOPE) in SCOPE next
bind ENVIRON to NEW_ENV in SCOPE next
bind EVAL_FUNCTION to
create_cell(EVAL,
    EVAL<BASE<O> ←this_process with
    BASE<1> ←EVAL_PROCESS<0>/SCOPE],
    SCOPE) in SCOPE next
transfer PAIR<(EVAL_FUNCTION in SCOPE)@>.BASE<1> next
{ Pass back result ↑
bind RESULT to (RESULT in SCOPE) next
destroy SCOPE
```
exit

{ FN is not a primitive atom or a LAMBDA expression, so evaluate it }
bind SCOPE1 to create_basket next
enclosing SCOPE1 is current_basket next
bind EXP to create_cell(cell, FN, SCOPE1) next
bind EVAL_FUNCTION to
create_cell(EVAL,
EVAL[BASE<0> ← this_process with
BASE<1> ← EVAL_PROCESS<0>/SCOPE1],
SCOPE1) in SCOPE1 next
transfer PAIR<(EVAL_FUNCTION in SCOPE1)@>.BASE<1> next
bind TEMP to (RESULT in SCOPE1) next
destroy SCOPE1 next
{ Apply (evaluated) function to arguments }
bind SCOPE2 to create_basket next
enclosing SCOPE2 is current_basket next
bind FN to create_cell(cell, TEMP, SCOPE2) in SCOPE2 next
bind APPLY_FUNCTION to
create_cell(APPLY,
APPLY[BASE<0> ← this_process with
BASE<1> ← APPLY_PROCESS<0>/SCOPE2],
SCOPE2) in SCOPE2 next
transfer PAIR<(APPLY_FUNCTION in SCOPE2)@>.BASE<1> next
{ Pass back result }
bind RESULT to (RESULT in SCOPE2) next
destroy SCOPE2

{ Return to caller }
transfer PAIR<!APPLY@>.BASE<0>@

{ End of APPLY function }
bind APPLY to structure(BASE; cell, APPLY_PROCESS) next
{ Create basket for LISP object storage }
bind STORE to create_basket next

{ Set up "constant" atoms }
bind NIL to create_cell(S_EXPR, S_EXPR<ATOM<0>>, STORE) next
bind T to create_cell(S_EXPR, S_EXPR<ATOM<1>>, STORE) next

{ Evaluate input S-expression }
bind EVAL_FUNCTION to create_cell(EVAL,
EVAL[BASE<0> ← this_process with
BASE<1> ← EVAL_PROCESS<0>] next
bind EXP to create_cell(cell,
create_cell(S_EXPR, ^ S ^ , STORE) next
transfer PAIR<EVAL_FUNCTION@>.BASE<1> next'
Appendix 8 - Description of LISP

{ RESULT is bound to the S-expression which is the evaluation of the program S-expression in EXP@ }
endproduction
endtranslator;
Appendix 8 - Description of LISP

translator S_EXPRESSION

pattern
A ← ATOM
'( ' ^ ^ S1 ← S_EXPRESSION ^ ^ ' ' ^ ^ S2 ← S_EXPRESSION
SN ← S_EXPRESSION^ ^ >
?? ^ ')
end pattern

production
[ control ATOMIC boolean A = null;
control NULL boolean A = null & S1 = null;
control PAIR boolean S1 = null & S2 = null;
control LIST_LENGTH integer #SN ]

< ATOMIC → { A }
if ST.A = null then
assign ATOM_COUNT to ST.A;
assign ATOM_COUNT + 1 to ATOM_COUNT
endif;
emission ATOM_NUMBER is NUMBER: ST.A ]
< 'S_EXPR<ATOM<^ ^ ATOM_NUMBER ^ ^ '>>' > |

< NULL → { ( ) }
< 'NIL@' > |

< PAIR → { (S1 . S2) }
< 'S_EXPR<PAIR[BASE<0> ← ' ^ S1 ^ 'with
BASE<1> ← ' ^ SN.1 ^ '']> > |

< LIST → { (S1), (S1 SN1), (S1 SN1 SN2 ...) }
< 'S_EXPR<PAIR
[ BASE<0> ← ' ^ S1 ^ 'with
BASE<1> ← ' ^
< LIST_LENGTH(I) →
< 'S_EXPR<PAIR[BASE<0> ← ' ^ SN.I ^ 'with
BASE<1> ← ' ^ > + > ^
'NIL@' ^
< LIST_LENGTH → < ']> > + > ^
', '' ]' >>

end production

end translator