On the automatic construction of program translators with minimal read/write storage requirement.

by

Majid Azarakhsh

A thesis submitted to the University of Glasgow in accordance with the regulations for the award of the degree of Doctor of Philosophy.

ACKNOWLEDGEMENTS

I would like to thank Professor D.C. Gilles and Pahlavi University for giving me the apportunity to study for a higher degree.

I would like to express my sincere thanks and appreciation to Mr. D.G. Jenkins who supervised this work, for his continued support and interest which were of great value in bringing this thesis to completion.

I also thank all members of the staff of the Computing Science Department of the University of Glasgow who have helped me in so many ways.

Majid Azarakhsh

M. Azarakhsh

CONTENTS

ABSTRACT

We present a translator system, KHAR, which is designed to use a minimum amount of read/write storage in environments where this is a scarce resource. The system may be used for languages which are LL(1).

We describe the system and use its application to the checking of the syntax of a machine oriented language, AML/1, to illustrate KHAR's handling of syntax and error recovery and similarly, use its application to the checking of the semantics of Wirth's mini-language, PL/0, to illustrate KHAR's handling of semantics. We show, too, how features not found in PL/0 can be handled.

The interface to the KHAR system provided for the designer/implementer of a language is a set of semantic graphs, after Cordy, to which may be added error recovery and code emitting actions. These graphs are encoded in a development of BNF, called here, Syntax Languages. The linearized graph, with its actions, is translated into two sets of tables, one to drive a push-down automaton to recognise the CFG of the language, with cross-linkage to the second which defines the action to be taken at that point in the syntax. These actions operate on registers and a read-only stack, which handle integer numbers as the encoded form of language symbols. The simplicity of this mechanism is due to the multipass nature of KHAR. We compare this simple mechanism with those used by Cordy.

We report on the degree to which KHAR meets its design objective of minimizing work storage requirements.

We also note the applicability of KHAR to research in language design, because of its clear and flexible interface. We discuss the portability of the KHAR system and its implications for the production of compilers for microcomputers. We also compare the features of KHAR with a compiler writing system.

CHAPTER 1: INTRODUCTION

MOTIVATION

At the start of this work, the need for small, multipass compilers to work in small background partitions within on-line, time critical, process control systems was known to exist [PIERCE]. As no clear candidate existed, or exists, for a standard real-time language, any method adopted would have to be language independent and, if possible, portable. Secondly the necessity to research language designs for use by electrical engineers moving from logic subsystems to microcomputer subsystems is urgent. High level languages such as PL/1 and PASCAL are inappropriate unless severely sub-setted.

Any language seeking to be of use to engineers must allow direct access to the byte and multiple length features typical of microprocessors while imposing the necessary conditions for reliability, say, type checking, restriction of access to variables and code structures. Such a language, called A Microprocessor Language, is under development [JENKINS]. The development process consists of the production of a series of refined languages, AML/1, AML/2 and so on. Refinement requires the use of each language by engineers to generate feedback to the designer. Since subjective factors are highly important, the language design must be quick to implement, be able to respond to user feedback, while at the same time providing good error recovery and reporting from the outset, for,

otherwise,the language would be unusable and no feedback obtained.

Good languages, such as PASCAL, were designed to run on large machines [WIRTHa] and the majority of implementations of PASCAL still are for large machines. The smallest known implementation at present is one for the PDP-11 which runs in a 16k word machine. It is thought that this compiler is capable of compiling itself in a 56k byte machine [PUG]. This may be contrasted with the CORAL 66 compiler for the INTEL 8080 which runs in 48k byte. Neither would satisfy the need to make a high level language available on a typical development system for a small electronics company investigating microprocessors. A reasonable starter kit might be expected to contain, say, 8k of RAM, but upto 32k byte of ROM. If a translator system were made available which was low on work space, yet used relatively little code, relying on tables held in ROM or pageable into RAM as required, a range of languages could be made available economically.

PHILOSOPHY

This thesis presents the results of re-applying table-driven methods to the construction of a complete translator scheme, in which the actions of each pass are encoded as entries into tables for interpretation by a single "translator engine", a "donkey" engine, which laboriously translates source code into object code while consuming as little work space as possible. This engine has been named KHAR, the Farsi for "donkey".

The desire to be able to develop stand-alone "good" compilers was advanced as one reason for adopting a multipass-approach. A second

issue is that of portability: a "good" electrical engineering language must be rapidly adaptable and transportable as new microprocessors appear on the market. The general issue of portability and one specific approach to it, the "abstract" machine, is discussed by Poole in [BAUERed]. The PASCAL compiler produced by the Free University of Amsterdam generates optimised code [TANENBAUMb], which is interpreted by a machine EM-1, designed to be portable across a range of microprocessors. A similar approach, to use pseudo-code as a step in the production of machine code was presented by [PASK0].

These approaches just outlined are examples of the use of intermediate abstract or virtual machines to map the programming environment and accessible abstract machine of PASCAL onto different actual hardware. The implementation of the AML sequence of languages cannot use this approach since the virtual machine manipulated by the programmer must be the same as the actual underlying hardware. We require a high-level language whose virtual machine manipulated by the programmer is as close to the actual machine as prudence and security of design permit. We thus present the KHAR system first in terms of such a language, the first of the AML series, AML/1. AML/1 is, in fact, a context free language since its specification calls for no semantic checking. Thus it is ideal for presenting the basic features of KHAR before considering a language of the first type, PL/0, which has been described in [WIRTHb].

The cost of implementing a new language is high, since the "state of the art" is to use the error recovery technique proposed by Amman [AMMAN]. This is used in both the Vrije and Delfast compilers for PASCAL. Study of the presentation of the PL/0 compiler in [WIRTHb]

reveals the amount of effort required to code the compiler for this minimal language on an interpreted abstract machine designed for the language.

An alternative approach to the use of a conventional compiler is to use a macroprocessor, as discussed in [TANENBAUMa]. One undesirable feature is that the syntax of the language may have to be altered to make the task of writing the ML/1 macros possible [BROWNa] [BROWNb], an influence which appears in the design of AML/1 [JENKINS]. The addition of any type checking and any attempt at error recovery complicates the task enormously. As a result, it becomes comparable to that of writing a conventional compiler.

Table driven compiler techniques are not a recent or new development. KHAR, however, is table driven throughout: that is, not only in syntactic checking, semantic and code generation actions but also in error recovery. Notably, the error recovery and reporting are driven by the syntax information. Conventionally, major attention has been paid to the syntax phases of compilation when considering table-driven methods. Wirth discusses this [WIRTHb], and this was the basis of Glennie's pioneering work [GLENNIE]. Cordy [CORDY] has introduced semantic graphs as a means of defining the semantic actions to be taken by a semantic phase of a compiler, given that the input stream is guaranteed to be free of syntactic errors. This table driven approach has been included in the system described here, although in KHAR the number of semantic primitive operations can be reduced due to the multi-pass approach adopted.

As discussed by Bauer in [BAUERed], and by Wirth in [WIRTHb], a

systematic method can be used to derive the coding of a translator from the grammar of its language into the code for a suitably defined abstract machine. Both state that error recovery cannot be treated in such a systematic way and that ad hoc methods are required, necessarily requiring an understanding of the errors most likely to be made and of the syntax of the language, so that sensible recovery can be attempted.

Error handling and recovery in an efficient manner has been attacked using table-driven methods by James [JAMES]. Currently, for production PASCAL compilers, the method of Amman [AMMAN] is adopted, which has definite limitations and involves the programming of the explicit addition of symbols to the "follow set" on entry to a procedure written to recognise a non-terminal of the language. The approach to error recovery in KHAR is to allow the designer to define error recovery productions. No problems arise over semantic information as discussed in [GRIES] since KHAR rigidly separates syntax and semantics.

The system is constrained to accept languages with an LL(1) grammar. LL(1) grammars and their application are discussed by Griffiths in [BAUERed]. Here, it is enough to say that

a) they make table driven methods useable,

b) error recovery is simpler since even a small degree of "look ahead" provides sufficient redundant information to enable good recovery,

c) automatic syntax improving techniques (e.g. Foster's SID) [FOSTER] exist to improve grammars so that they are LL(1),

d) LL(1) languages are easier to read and use [HOARE].

This last point is of great importance when the final design objective of KHAR, its use in language design, is considered. As remarked by Watt [WATTb], one of the most useful tools available to designers and implementors of programming languages has been the Context Free Grammar(CFG). A CFG is capable of defining a large part of the syntax of a typical programming language, and the existence of a wide variety of syntax-directed parsing techniques [GRIES] has facilitated the construction of efficient deterministic parsers from such syntax definitions.

He remarks further that CFGs are deficient in two respects. Firstly, they are incapable of defining context-sensitive syntax features. Secondly, they provide no explicit means of linking semantics to syntax. One approach to this problem has frequently been adopted in translators, that is, a set of "semantic routines" is provided, and names of semantic routines are inserted in the RHS of production rules, an approach conventionally associated with top-down parsing.

The KHAR system uses this approach, KHAR itself being a table driven recogniser of a CFG, which may have verbs placed at appropriate points of the grammar. Our work relies heavily on the use of the syntax graph, as in [WIRTHb] and the semantic graph, as introduced in [CORDY].

However, we have minimised the number of verbs (actions) required and simplified the internal structure of the semantic mechanism of KHAR by dealing with semantics (context sensitivities) one aspect at a time. This simplification arises from the multi-pass philosophy

adopted throughout the work. The meaning and implications of context sensitivities are defined in terms of a semantic graph with the necessary actions added at the appropriate points. This has the advantage that the designer is constrained to describe these sensitivities one at a time and thus the user can find out the exact way in which a sensitivity is handled.

CHAPTER 2 : APPROACH

The objective of this work is to construct a flexible and portable translator system for a variety of high level programming languages to be implemented on small computers.

We also intend to generate this system in such a way to be able to use it for real time applications and to be highly configurable both in terms of its compile time environment and in terms of the object machine for which it compiles any particular language.

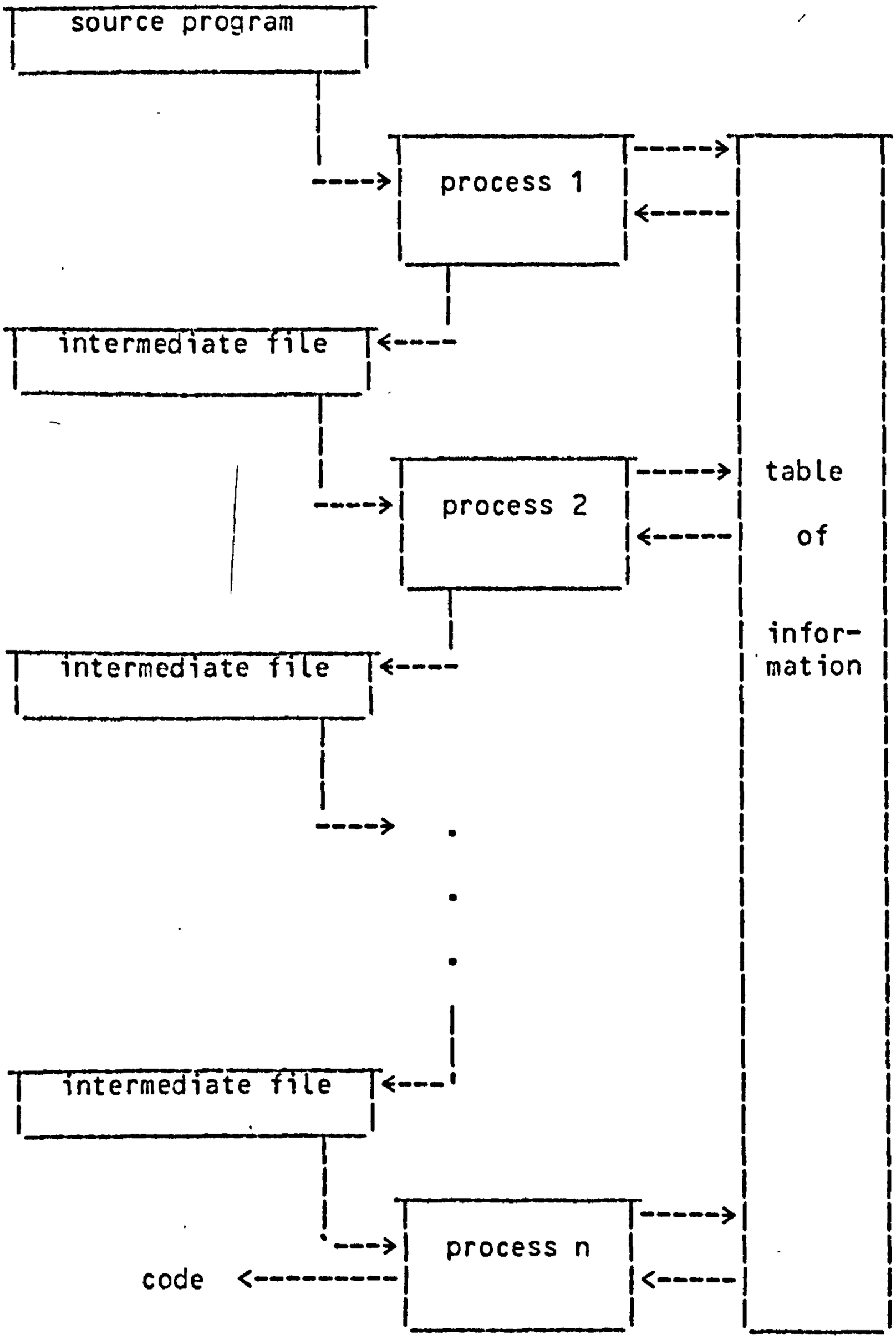
Regarding the efficiency of the intended translator system we are more concerned about the space requirement of any compiler produced rather than its run-time demand.

Some of the programs in this system are executed only once for each language implemented. Their task is to create a few files before any source program can be run. These are permanent files as long as no change is made to the language.

The basic approach to this work is that of Kernighan and Plauger[KERNIGHAN]; a system is constructed of a sequence of sub-processes each of which consumes as input the output of its predecessor, if any.

PRINCIPLES OF THE APPROACH OF KHAR

In our approach we aim to break the compiling process into as many separate processes as possible. Ideally, every separable task is to be carried out by a process with one input and one output stream. We aim to reduce the maximum memory requirement in any pass of the compiler to a minimum and make it possible to use computers with little available read-write storage. A process might itself be set up to read one or more input streams in a prelude or setting-up phase. Thus the general model of the work can be illustrated as shown in the following figure.



This shows a number of subprocesses which transform a stream of symbols (which can be understood by the programmer as a program) to a stream of symbols which is a program for the actual machine.

The input consumed by one process is the output of a previous process and is the concrete linear representation of a data structure.

This data structure is transformed into output in some specified way by the process, that is through the execution of some set of instructions, a program or a procedure. The table of information on the right of the figure given above is a set of files associated with processes.

It should be noted that this base of information changes state as the result of the process. The individual processes may need access to only a subset of the information in this table and access is controlled accordingly.

These processes numbered 1-n can either be the execution of distinct programs or several executions of the KHAR machine each controlled by a different syntax graph, or under the control of one graph, with KHAR operating successively in each of three of its four modes. (Modes 1, 2 and 3 are used to translate programs, and modes 1 and 4 used to encode transition and action matrices by translating syntax graphs encoded in one of the SL languages.)

We do not show the syntax graphs in the figure since they are a separate set of structures determining the processes. We show this aspect of the system more fully in chapter 3.

GRAMMAR AND FORMAL DEFINITIONS

All language is based on a vocabulary. For programming languages the elements of this vocabulary are called "language symbols" or "language keywords". For each of these languages is defined a set of rules or formulas which define the set of well-formed sentences.

These rules are called productions because they determine how a formally correct sentence of the language is produced.

A grammar G(Z) of any programming language of this class is defined to be a nonempty finite set of productions. Z must appear as left part of at least one rule and is called the distinguished symbol. Those symbols appearing as a left part of the rules are called nonterminals and the other symbols are called terminals. Therefore the vocabulary 'V' of a language is the union of its terminals and nonterminals. We will use underlined angular brackets '<u><</u>' and '<u>></u>' for nonterminals to distinguish them from terminals.

We only consider those languages whose grammars satisfy the following rules :

1) The initial symbols of alternative right parts of productions (the director symbols) must be disjoint. (The initial symbol of A is the set of all terminals that can appear in the first position of sentences derived from A.)

2) For every nonterminal symbol A, which generates the empty sequence 'e', the set of its initial symbols must be disjoint from the set of symbols that may follow any sequence generated from A. (This point is discussed by Griffiths in ch 2.b of [BAUERed] and by Wirth in [WIRTHb].)

NOTATIONAL SYSTEM

To describe these productions we need a notational system, or in other words a metalanguage- a language in which we can describe another language. The notation we use is called Modified Backus-Naur

Form (MBNF). It presents an exact explanation of the language construction. These notations are as follows:

1) underlined braces { and } are used to enclose multiples from which a choice must be made. Expressions may be presented vertically like this

<label part>

<const part>

.

.

.

or horizontally like this

{ <label part>|<const part>| ... }

(note the meta symbol 'vertical stroke' between expressions)

2) underlined square brackets [and] enclose optional statements or expresions such as [label :]

3) three dots "..." indicate repetition of preceding item, one or more times.

CONVENTIONAL COMPILATION PROCESS

A compiler or, better, translator, is a program which accepts as its input, a source program written in some high-level language, such as Algol 60 or FORTRAN, and produces as its output, the appropriate code for a specific computer. This output is called the "object program". This process is traditionally divided into analysis of the source program and then synthesis of the object code.

In the simpler analysis part, the compiler accepts the source

program, discards those parts of the source which are not to be compiled (such as comments) and transforms the source into tokens. The source program is now a linear string of symbols, the input characters having been grouped into tokens of the language such as language symbols, identifiers, etc.

The compiler also builds several tables of information during the analysis part for the definitions of the new tokens, (for example identifiers). These tables are used during both analysis and synthesis phases.

After the execution of these lexical tasks, the source program has been converted into its basic tokens and is ready for syntax and semantic analysis, in which the string of tokens is scanned using the syntax rules, the tokens are grouped in order to make sentences of the corresponding language and, usually simultaneously, a complete semantic check of the source program is performed. Runtime storage and addresses are then allocated to variables and the internal representation of the program is used to produce assembly or machine language of the target computer by the code generator. This code generator is the hardest task in a compiler, the most systematic approach presented in the literature being that of [WILCOX]. A fuller discussion may be found in [GRIES] or [BAUERed].

LEXICAL SCANNING IN KHAR

Source text is read as a string of characters, each basic symbol of the language is recognised and this will be passed to the output file in an internal form, that is to say,as an integer number.

Language symbols i.e. reserved words, standard names, special symbols etc. are represented by predefined codes. User-defined identifiers, constants and integers are represented by an index into the appropriate dictionary, plus an offset (2000, 3000, ...) which identifies their class.

This encoding minimises the overhead in transmitting information between processes, as it allows the rest of the processes to operate with fixed-length symbols rather than variable length strings of characters.

It is good communications practice to use the shortest encoding for the most frequent bits of information encountered. Programming language design, properly, takes no account of this since it has other wider concerns. In KHAR a form of intermediate text is used which is chosen after balancing the requirements of readability (to aid development) and compactness (to save space).

In principle further research could be undertaken to determine the most effective form of encoding for the system based on known or measurable statistical information about programs written in each language, but we do not consider this further here. Integer representation requires two bytes of storage at most for each code in read/write storage and between 3 and 5 characters on backing storage, (including the space separating integers).

SYNTAX IN KHAR

As remarked in [WATTa], a CFG grammar serves as a means of

communicating both between the language designer and the programmer, and between the language designer and the implementor. Watt states that a well designed CFG can similtaneously satisfy all the requirements but that, unfortunately, typical programming languages are not strictly context-free. Examples of features of a typical language which defy description by CFGs are the correspondence between declarations and applications of identifiers, and the compatibility of formal and actual parameters. In a programming language with generalised data types, even type compatibility cannot be defined by a CFG.

Watt argues that "syntax" should be extended to cover all criteria for well-formedness of a program which can be determined algorithmically. Our view is the contary, that "syntax" encompasses precisely those features of a language that can be defined by a CFG and checked by a context-free parser. In KHAR we deal with "context sensitivities" by including in KHAR a limited set of semantic and code emitting verbs, which can be placed where required in the linearized form of the syntax and then using semantic graphs, which are syntax graphs augmented with semantic actions, [CORDY], to define the behaviour of KHAR so as to produce a pass of the "compiler" which can deal with a particular context sensitivity.

The advantages of this approach seem to us to be that KHAR need use only one stack of integers to handle semantics as contrasted to the algorithm presented in [WATTb] which requires the stack to handle contexts, expressed as sets and, further, that specific sensitivities are described graphically and individually to the user of the language. Again, as we shall see, this approach produces a much

simpler semantic mechanism than those of [CORDY].

This graphical approach contrasts with the formalisms of two-level grammars and extended affix grammars, but, of course, is unlikely to bear the burden of the proof of correctness which is the distinguishing property of the latter [WATTa]. However, the presentation in [BOCHMAN] of a compiler writer shows the advantages of the separated, graphical presentation which results in KHAR.

Further, we observe here that designer and implementor are the same person in the KHAR system, since the definition of the graphs can be encoded, translated by the system, and used to carry out a compiling process for his language.

Gries discounts some error recovery techniques since they involve semantic information being discarded which does not occur in this approach since no semantic content has been handled. So a wide range of methods could be applied because of separation of the checking of syntax and semantics. This has not been developed in KHAR but as was remarked above, error correction can often be achieved by using an existing production or by adding a few error productions. This allows the designer to concentrate on language definition at the separate levels and to ignore the error problem, although he can improve performance by adding error productions if he wishes.

SEMANTICS

Semantic analysis is a part of compilation which comes between

syntax checking and code generation for the purpose of checking the semantic structure of the source program.

To check for semantic correctness we need information about attributes of all identifiers, which are found in the declaration part of the source program. In semantic processing we use information tables which we created in the process of encoding the symbols in internal form. These tables contain information about identifiers, integers and character strings. In this process some semantic information such as attributes, addresses, dimensionalities and so on will be collected and added to these tables which will be referenced at a later time in the same process, to check for semantic errors.

For each identifier we have one entry in the table and the amount of information we need for that, depends on the type of that element, therefore we have variable length entries in the table. For Algol-like languages in which procedures may be nested, the same identifier may be declared in different procedures and each such declaration must have a unique table entry associated with it.

Each time an identifier appears in the input stream, it carries some information. This information will be checked against the information we have already in the table by our semantic operations.

SEMANTICS IN KHAR

Four kinds of semantic operation are presented in the semantic charts of [CORDY]. These operations, which provide different kinds of actions are :

1) Normal Operation,

2) Parametrized Semantic Operation,

3) Emitting Semantic Operation and

4) Semantic Choice Operation.

Tables, stacks, queues and so on are called "semantic data structures". The operations are called "semantic operations" and together a semantic data structure and its associated operations are refered to as a "semantic mechanism".

The semantic operations, named above, are meant to provide a complete set for accessing and maṇaging a semantic data structure, and the operations on any of the data structures are restricted to these four classes. This restriction makes possible a generalized automatic chart interpreter.

Semantic mechanisms which are commonly needed in producing a set of semantic charts for a programming language are discussed. These are :

1) symbol table mechanism,

2) type stack mechanism,

3) count stack mechanism and

4) address fix mechanism.

The multipass structure of the KHAR system reduces the mechanisms to one, plus a group of registers. The operations on the single stack are fewer in number and are distinguished by having no knowlege of the attributes of identifiers. This is discussed fully in chapter 5.

CODE GENERATION

In KHAR we are dealing with a language in which the programmer desires to control the code exactly, this being a design objective of the system. Thus code generation is simple as the high-level language must map one for one into the machine order code. This mapping may be constrained by our semantic typing and some orders, especially transfer of control orders, hidden by the flow of control structures of the language. At the other extreme, say, PASCAL, we may generate code for any suitable intermediate pseudo-machine [PASK0] [TANENBAUMb]. We have, at present, generated code for a high-level assembler (AML/1) and PL/0 [WIRTHb].

CHAPTER 3 : DESCRIPTION OF KHAR

We describe the KHAR system by presenting the structure of the system and briefly presenting its processing of syntax and semantics, before presenting the designer's interface, the information stuctures and process involved in converting the external representation of a graph or program to an internal one, carrying this through to a practical example of the use of this interface for AML/1.

KHAR consists of a pushdown automaton, which traverses the syntax graph, coupled, via the obeying of verbs, to a pushdown transducer. Errors are dealt with by error productions. Since KHAR deals separately with syntactic error recovery and the other tasks, error recovery can be often achieved by using an existing production in the language. This is done for PL/0 where the emphasis is on the semantic checking and code emission tasks. The pushdown transducer is similar to that of Cordy [CORDY] but is structurally simpler since only one aspect of the semantic task is dealt with in one pass. For example, at no time does KHAR access tables of information about identifiers. All transmission of semantic or environmental information is via semantic tokens emitted by a previous pass. Error situations do not have to be defined. Semantic checking is achieved by defining acceptable syntax graphs for valid combinations of types and operators. Transmission of information up and down the Abstract Parse Tree is achieved by successive alternating passes through two complementary phases to check the semantics of expressions. The only

information placed on the stack of the transducer is in the form of the internal code for tokens. This enables the handling of considerable programs within modest read/write store requirements.

Diagram A shows a primitive KHAR system.

This structure requires that the language designer constructs the driving tables by hand. Effectively he has to machine-code KHAR. This is an unacceptable interface for general use, and undesirable for the development work on KHAR. We clearly require a translator from an external representation to an internal one, as discussed in [WIRTHb]. This gives the revised diagram, B.

These diagrams follow on the next page.

Diagrams A & B

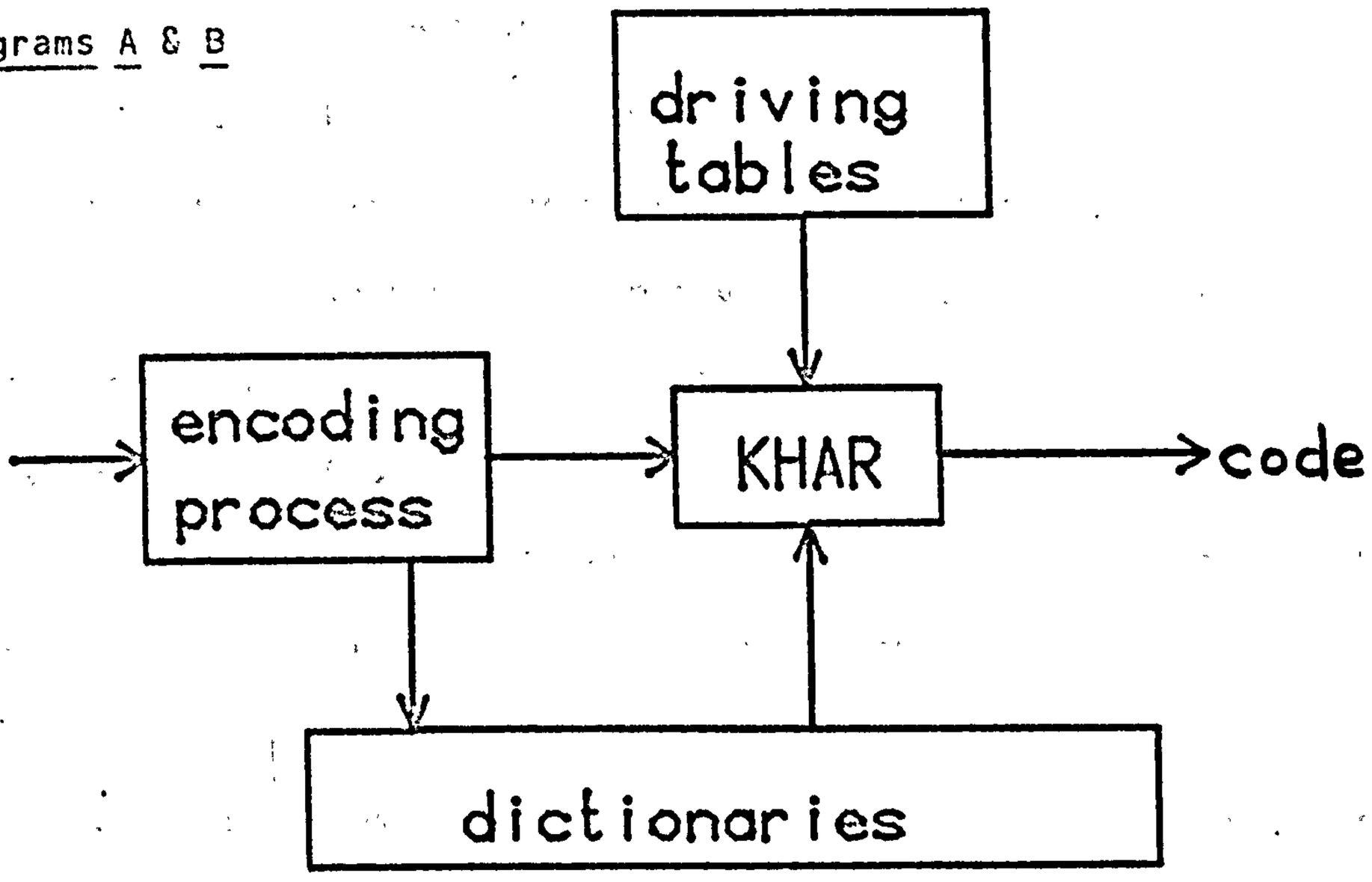
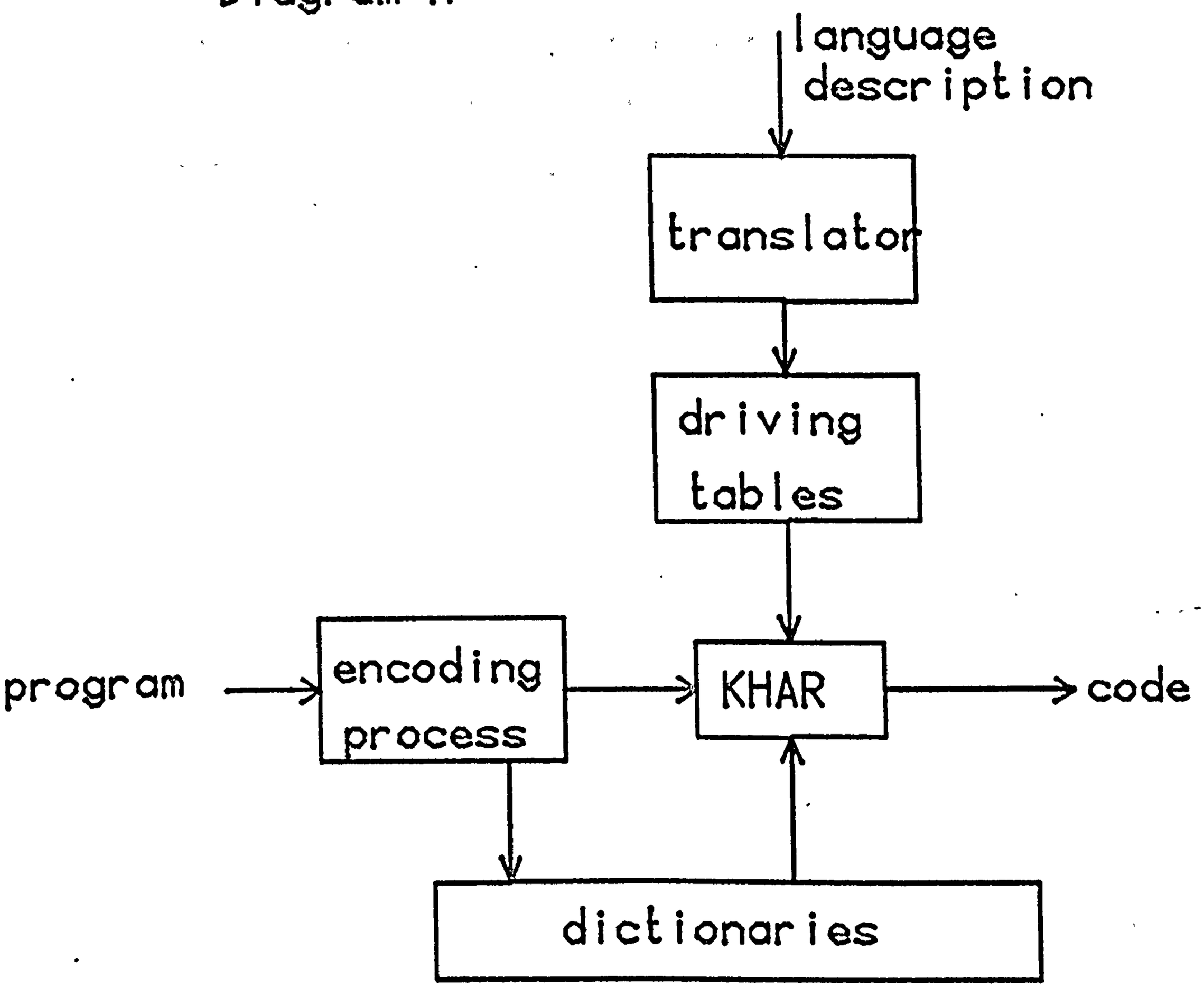


Diagram A



Diacram B

We now have to supply the translator. Wirth gives the PASCAL coding of a compiler which recognises a modified BNF to construct the data structure required to drive a syntax analyser for a language. In the KHAR system, we use the primitive version, supplying a hand coded pair of tables, to implement a zeroth version of a Syntax Language (or Small Language), SL0, and add table building actions to KHAR. SL0 has the smallest possible syntax and allows access to the minimum features of KHAR needed to allow the definition of tables for a larger language with error recovery, SL1. This successfully minimised the encoding which we had to do by hand to about 140 integer codes. SL1 was used to define successive versions of SL as facilities were added to KHAR. The present interface to the KHAR system is by using SL4. These SL languages are more fully described in chapter 6. Diagram C, on the following page, now presents a picture of the KHAR system set up to translate AML/1 [JENKINS].

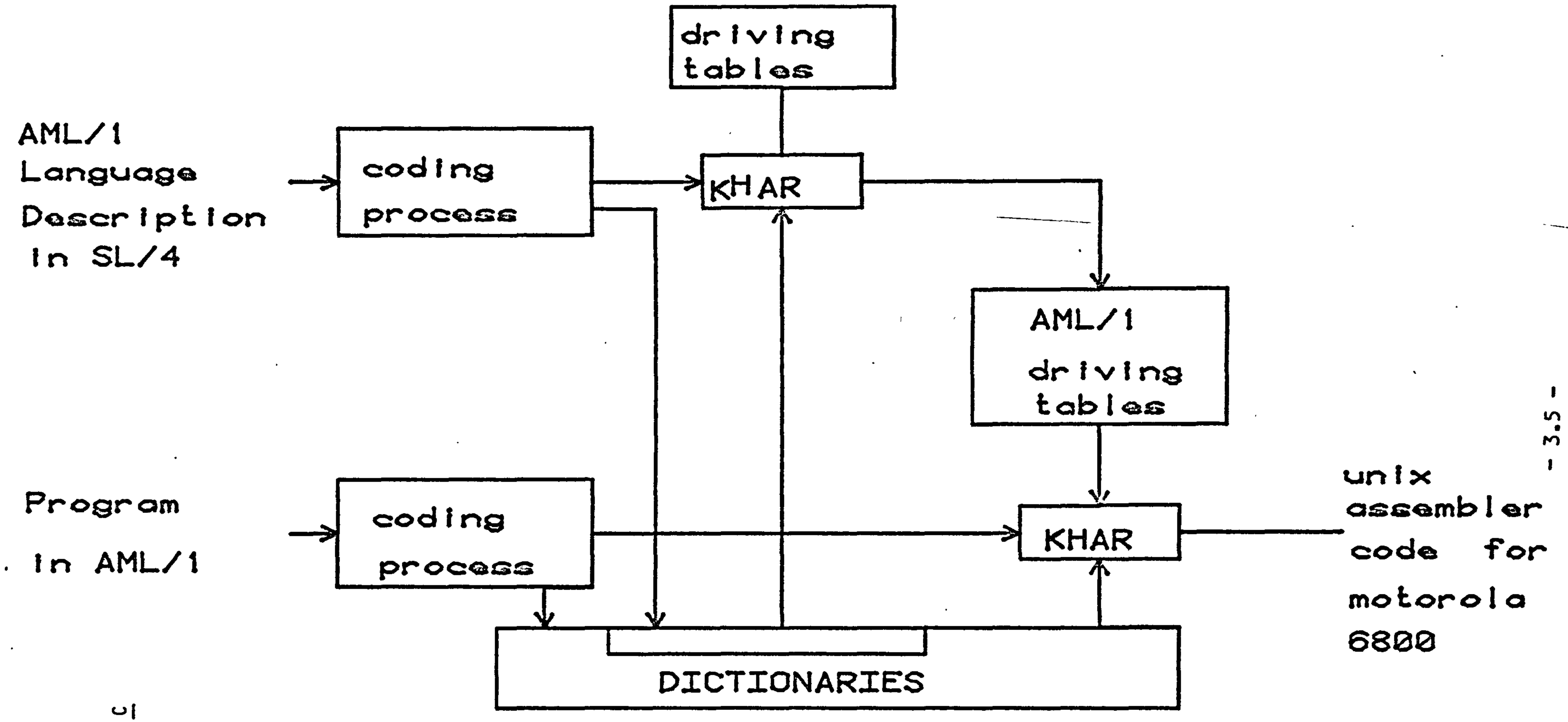


Diagram C

Diagram C

KHAR can deal with languages which have a context free grammar(CFG), such as AML/1, in which no semantic knowledge is needed to parse sentences successfully, in one pass. The system up to this point is only concerned with establishing that the syntax of the program is correct, and syntax here is used in its narrowest sense. As remarked by Watt in the introduction to his thesis [WATTa], practical languages do not have Context Free Grammars. We see this easily enough by inspecting the syntax of PASCAL as given in [JENSEN] or [WIRTHb], and find it true also of PL/0 [WIRTHb]. We see immediately productions like

<typeidentifier>::=<identifier> and

<callclause>::=<CALL><procedureidentifier><;>.

We thus have to modify the syntax of PL/0, the language used to demonstrate the treatment of semantics in the KHAR system in this thesis (chapter 5) to remove this semantic intrusion to produce the syntax of a CFG. We use the version of the system outlined so far to check the syntax of a PL/0 program.

To deal with the semantics of PL/0, and any other language, we focus our attention on one particular aspect of the semantics and draw a syntax diagram with actions placed to deal with that aspect. This is then translated so as to drive KHAR to form a pass of the "compiler" that the KHAR system will become for PL/0.

This approach allows a designer of a language to focus attention one aspect of it at a time, and also allows a user to see the effect of a feature of the language specification by inspection of a diagram.

Where the internal structure of KHAR is too simple to permit full semantic checking , as in the propagation of leaf attributes up the abstract parse tree of an expression, we define two passes through KHAR. A previous pass has appended every identifier with its type. These two passes then run alternatively, the first amending the presentation of the first (operand,operand,operator) triple encountered so that the second pass can use a simple syntax graph to check for the valid (operand,type,type) combinations. The approach is discussed in more detail in chapter 5.

We emphasise that this multi-pass approach makes it unnecessary to include in the KHAR machine itself any concept of "type". In fact, KHAR never requires to handle the representation of the attributes of objects in a program. Comparision of types as implied above is achieved by seeking to match the symbol read from the input stream with the current expected token. That these may both represent "REAL", or one "REAL" and the other "INTEGER", is of no consequence. That (REAL,REAL,+) is valid in ANSI FORTRAN and (REAL,INTEGER,+), not, is not included in KHAR. It is expressed by including the branch *-REAL-REAL in the syntax graph of the second of these two passes and omitting *-REAL-INTEGER and *-INTEGER-REAL. We need only state what is valid. The properties of KHAR as a recogniser will do the rest. Further, since each pass, or group of passes, deals with one aspect of a language, semantic errors of the same type are reported together, which should assist the user in program debugging.

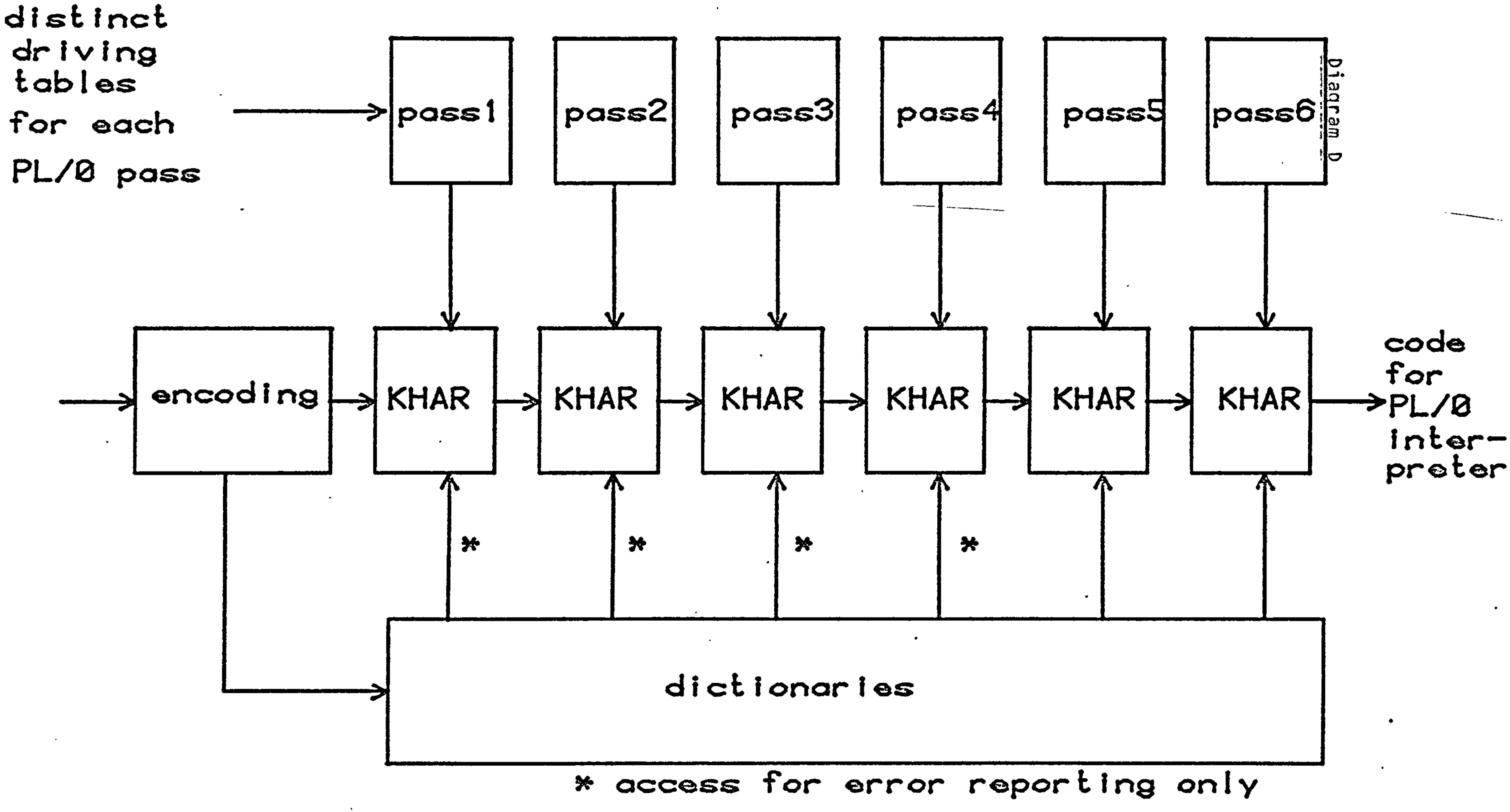
We note also in passing that error recovery actions are also included in KHAR so that error recovery is under the control of the language designer and the majority of syntax errors are reported in

one pass. We discuss error checking in chapter 4.

Our picture of the KHAR system as set up to form a PL/0 compiler is now as in Diagram D, on the following page.

The syntax graphs and actions for each of these passes are, of course, all defined using SL4.

We discuss syntax graphs and our basic checking technique in chapter 4.


distinct driving tables for each PL/0 pass
pass1
pass2
pass3
pass4
pass5
pass6
Diagram D
encoding
KHAR
KHAR
KHAR
KHAR
KHAR
KHAR
code for PL/0 inter-preter
*
*
*
*
dictionaries
* access for error reporting only

(Diagram D)

PREPARATION OF INFORMATION TABLE TO BE USED IN CODING THE SOURCE LANGUAGE SYMBOLS

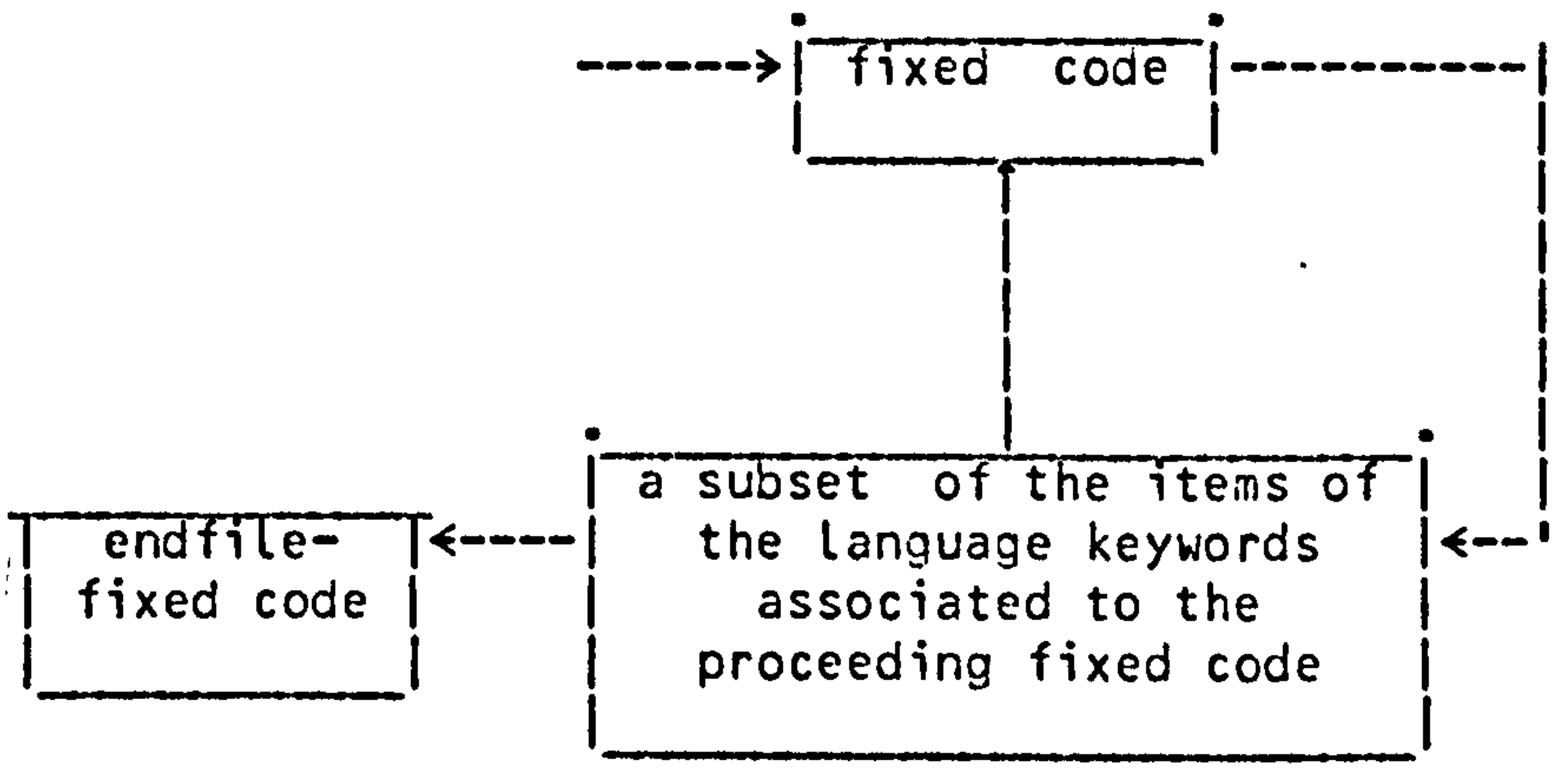
For each language to be used by our system, we need to prepare two sets of information. These are the "language keywords" and the "language syntax". We call them the Language Symbol Data (LSD) and the Transition Table Data (TTD) respectively. The former, LSD, is made from the terminal symbols used in the language syntax and the latter, TTD, is the information derived from the syntax itself, that is from the transition matrices of the language. Both of them have their own special syntax in data preparation, which we explain separately.

Language Symbol Data

The set of elements on the top of columns in all transition matrices of a language is called "the valid elements of that language", or "language symbols".

For each programming language we group terminals according to the classification, (reserved words, ... ,triple character delimiters), given above, and call it "language symbol data". This data is a string of characters divided into groups, starting with the associated fixed code for each group and one group following the other. Each construct or symbol, however, is separated from the next by at least one blank or newline and the same applies to the associated group code number encountered in the beginning of each group.

The layout of this data is as follows:



Each item of the language keywords is to appear in one and only one of the above subsets so that each has a unique code. These subsets can appear in any order as long as they are preceded by their fixed codes. The number of different parts is not fixed.

It is in this file that we introduce the symbols which may start comment and terminate comment. They come under the fixed code, 17. For example in the symbols of the language Algol W we may write

17 comment ; or in Pascal

17 (* *) These two symbols are separated by a blank and can be any character string. The length of this string could be up to the maximum length of identifiers in the language.

This means in process of encoding of the source text, the system reads and copies all characters until it finds the "comment-start" symbol, the first symbol after fixed code 17, then reads and ignores until it finds "comment-end" symbol (the second symbol after fixed code 17).

It is also in "language symbol data" that we introduce constant string ends, "start constant string" and "end constant string" symbols. So any number of characters appearing between these two symbols are considered as one constant string and will be coded to a single internal code. If the "end const string" symbol is itself a part of a constant string it must be typed two consecutive times inside the constant string. For example if these symbols for a programming language are START and END then

write(START ENDEND is the symbol to end a const. string END);

could be a write statement to output

"END is the symbol to end a const. string"

In this data, the fixed code, '0', as shown in the table of fixed codes, means "comment". So any text appearing in the data after '0' is treated as comment. A typical example of this data follows which belongs to the programming language AML/1. Please note that the separator between symbols is at least one blank and the format is free.

```
0  from this line until the first semicolon encountered
   text is treated as  comment and is ignored by the
   program which uses this data.
  The following are the keywords of the AML language.
   Numeric fixed codes are :

     1  for reserved words.
     2  for standard functions.
     5  for single character delimiters.
    17  for comment ends.
    18  for constant string ends.
   100  for end of file.
;


1
a b d q x
at do if in
end out ref rep
body byte call code data else here main proc then
bytes while
define
endcall endcode endproc program
endprogram


2
cc cs ge gt hi le ls lt
mi ne pl ra sr vc vs


5
{ [ ( > ; , : ) ] }
= '


17
(* *)


18
 " "


100
```

There is a program to read this data and make two files, the "code name file" (CNF) and the "code name index file" (CNIF), which are permanent files for their corresponding language.

CODE NAME FILE & CODE NAME INDEX FILE

CNF is constructed so that it has only one blank as a separator between symbols, and the group codes are not located along with the information. The group codes are separately located in the first column of the two dimensional array CNIF which is simultaneously created to access the information in the Code Name File. The second column of this table contains the pointer which locates the start of each group of information according to the corresponding group code from column one of the table while the third column contains the starting code for each group.

The codes in column three of this table take into consideration the complete symbol within the grouped information. These codes are based on an arbitrary starting code, the "base-code".

This information on CNF accesed via CNIF, regarding the various symbolS found in programming languages, is then used to code the symbols encountered in the source programs.

Similarly the symbols located in the transition matrices are replaced by the codes determined from this information and the same applies to the transition tables of the SL languages.

To find what code should be associated with any symbol, the following steps are to be considered

1) a check is made to see if the first character is alphabetic. Two possibilities may occur :

a) the first character is alphabetic in which case we form a tentative idea that the symbol may be in any of the first four groups in CNF.

b) The first character is not alphabetic in that case the symbol falls in one of the other two groups i.e. group with fixed code 5 and 6. If case (a) happens all four groups may be searched one after the other , if required, until such times that the symbol is matched, otherwise the symbol is treated as a user's specified identifier.

Although a similar procedure would be undertaken in case (b) above, by considering the length in characters of the source symbol first, and one of the two groups can be skipped right in the beginning. While the matching of symbols is continued, the relevant code information from case (b) is continuously updated so that by the time the search is completed the appropriate code is also determined. In this way the digital coding of the symbols is accomplished automatically and can very easily be altered. Any alteration in the base code will be accounted for automatically in determining the subsequent codes.

INTERNAL FORM OF THE SOURCE PROGRAM

As we mentioned in previous sections, standard names and special symbols will be coded as 3-digit integers. Integers apearing in the source text are put in the dictionary of integers and in their places the appropriate code. Integers are coded from 1000 to 1999 , that is their index plus 1000.

Users' defined identifiers are coded from 2000 to 2999 and

constant strings from 3000 to 3999.

ENCODING

This encoding is done in six steps. The input to step one is the source text and the output from step six would be the internal form of the source text. Each step, (other than the first), takes the output of the previous step as its own input. The steps are in the following order :

1) delete all commentary and code the newline character;
2) code all constant strings and integers;
3) code standard names and user's defined identifiers;
4) code triple special-characters;
5) code double special-characters;
6) code single special-characters.

The output from step six is a sequence of integers.

FIXED CODES AND VARIABLE CODES

The vocabulary T of terminal symbols of languages we consider can be categorized as follows :

a)	Reserved Words	(RW)
b)	Standard-Procedure-Identifiers	(SP)
c)	Standard-Function-Identifiers	(SF)
d)	Standard-Constant-Identifiers	(SC)
e)	Single-CHaracter-Delimiters	(SCHD)
f)	Double CHaracter-Delimiters	(DCHD)
g)	Triple-CHaracter-Delimiters	(TCHD)

(We assume that in the class of languages we use, there exists no symbol consisting of more than three special characters.)

These can be assumed to be classes or groups and a code can be asigned to each in order to identify them.

The assigned codes to the above classes of objects remain invariant for all the programming languages we consider. Three types of coding arise as undernoted.

1) Fixed codes as shown in the table on the next page. Although the choice of these codes is arbitrarily associated to the language constructs, we assume that for our purpose they remain invariant for all the programming languages to be considered in the work.

FIXED CODE	MEANING
0	undefined (in code files) comment delimiter (in Language-Symbol Data)
1	reserved word
2	standard-procedure-identifier
3	standard-function-identifier
4	SL-reserved-words
5	single-character-delimeter
6	double-character-delimeter
7	triple-character-delimeter
8	identifier in general
9	constant identifier
10	function identifier
11	procedure identifier
12	type identifier
13	variable identifier
14	field identifier
17	comment delimiters
18	constant string delimiters
19	unsigned integer
20	constant string
21	transition matrix
22	state number
23	valid element
24	action number
25	nil symbol
26	any symbol
100	end file
999	end line

2) Semi-fixed codes which change from one language to another but are fixed within one language, e.g. codes for language symbols.

3) Variable codes which change from program to program, e.g. user-defined identifiers.

The code association is accomplished automatically by a program, called "codefile-maker", starting from a base-code.

We reserve all one-digit and 2-digit integers for use as fixed codes, all 3-digit integers for semi fixed codes, 4-digit integers for variable codes, and have defined 100 as the base-code. It should

be noted that this is arbitrary and can be readily changed to increase space efficiency.

A PRACTICAL EXAMPLE

When we want the system for a particular language L :

1) Switch the system to SL language;
2) Write a program in SL for the language L;
3) Compile and execute this program,
   (this builds up tables which control the recognition and parsing of statements in L);
4) Replace previous tables by these new ones and the system is ready for the language L.

In this part we explain how to prepare the system for AML language. All details are included and can be used as a pattern for using the system. Finally we do some changes in the syntax of AML and see the consequent and necessary changes in the system.

AML Programming Language

AML is a high level assembler designed[JENKINS], to relieve the programmer from some of the tedium of knowing the exact syntax required to use the addressing modes of the microprocessor via the manufacturer's assembler.

The following table gives the grammar of AML/1 in BNF.

```
<program>::=PROGRAM<identifier><location><program body>ENDPROGRAM

<program body>::={DEFINE<identifier>=<constant>}*
                  {DATA<data brick>ENDDATA}*
                  {PROC<identifier>BODY<block>ENDPROC}*
                  MAIN<block>ENDMAIN

<data brick>::=<identifier><location><declarations>

<location>::=HERE|<at clause>

<at clause>::=AT<number><suffix>

<suffix>::=B|Q|D|H

<number>::=<digit>{<digit>}*

<digit>::=1|2|3|4|5|6|7|8|9|0|A|B|C|D|E|F

<declarations>::=BYTES<size>|{BYTE<identifier>}*|{REF<identifier>}*

<size>::=<integer number>|<string>

<string>::="<printing character>{<printing character>}*"

<block>::={<statement>}*

<statement>::=<simple statement>|<while statement>|<if statement>

<simple statement>::=<code brick>|<call clause>

<code brick>::=CODE<target machine assembler>ENDCODE

<call clause>::=CALL<identifier>ENDCALL

<while statement>::=WHILE<cond>DO<block>REP

<if statement>::=IF<cond>THEN<block>{ELSE<block>}END

<cond>::={<simple statement>}<conditional test>
```

We first construct a set of syntax graphs from the grammar, (we assume that the reader is familiar with the rules of graph construction). For each nonterminal we may have one syntax graph. It is a good practice to reduce the number of graphs to as few as possible by merging them together. That is if a nonterminal is only used once in the syntax graphs of other nonterminals, we replace it

by its graph. Also it is better to replace the nonterminals by their graphs if they have a small syntax graph even if they are used a few times. This minimizes the amount of information we need to keep and also reduces the compilation time.

Figures 3a to 3c show the syntax graphs of this language.

FIGURE 3a Matrix 99 for AML/1

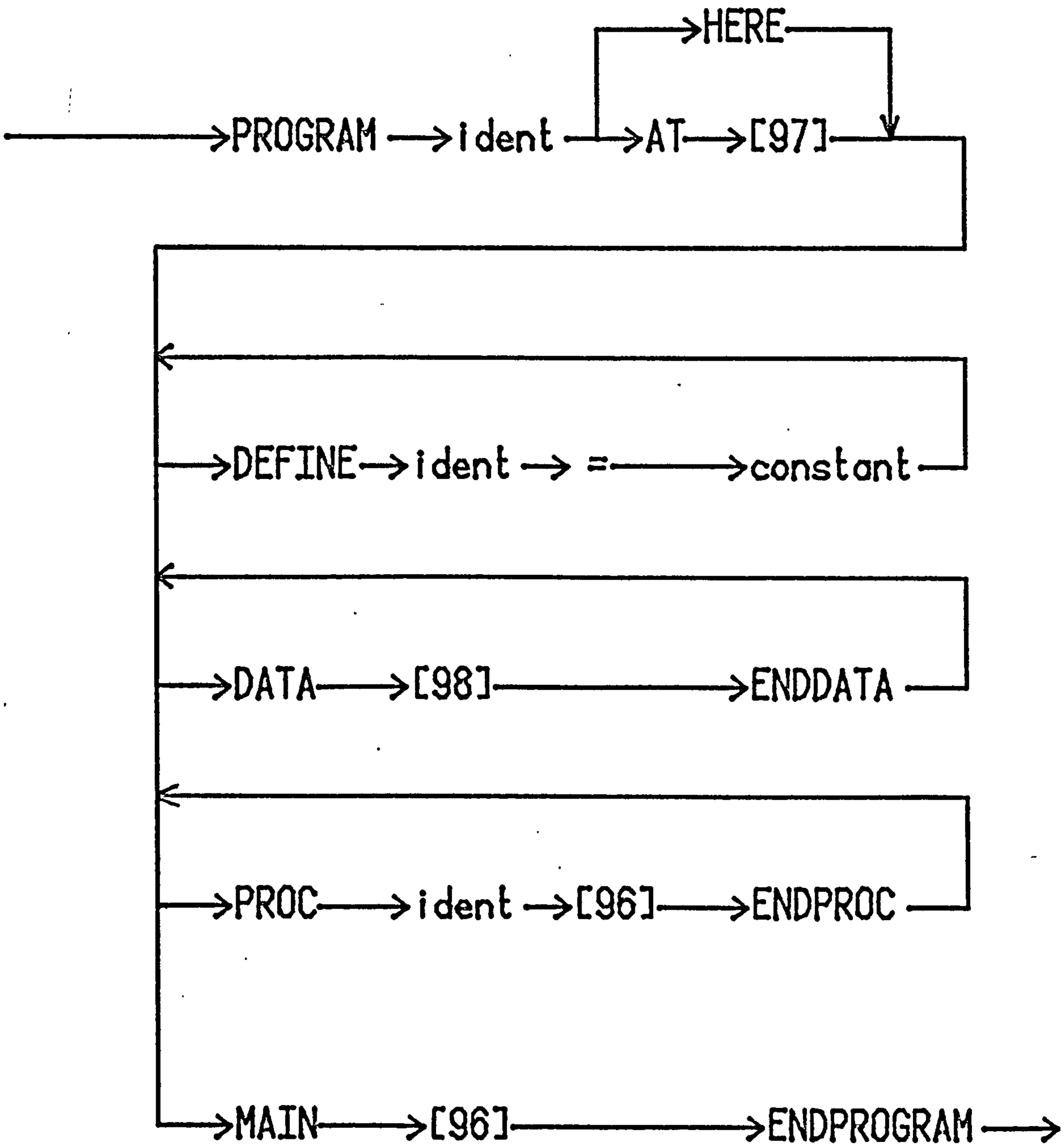
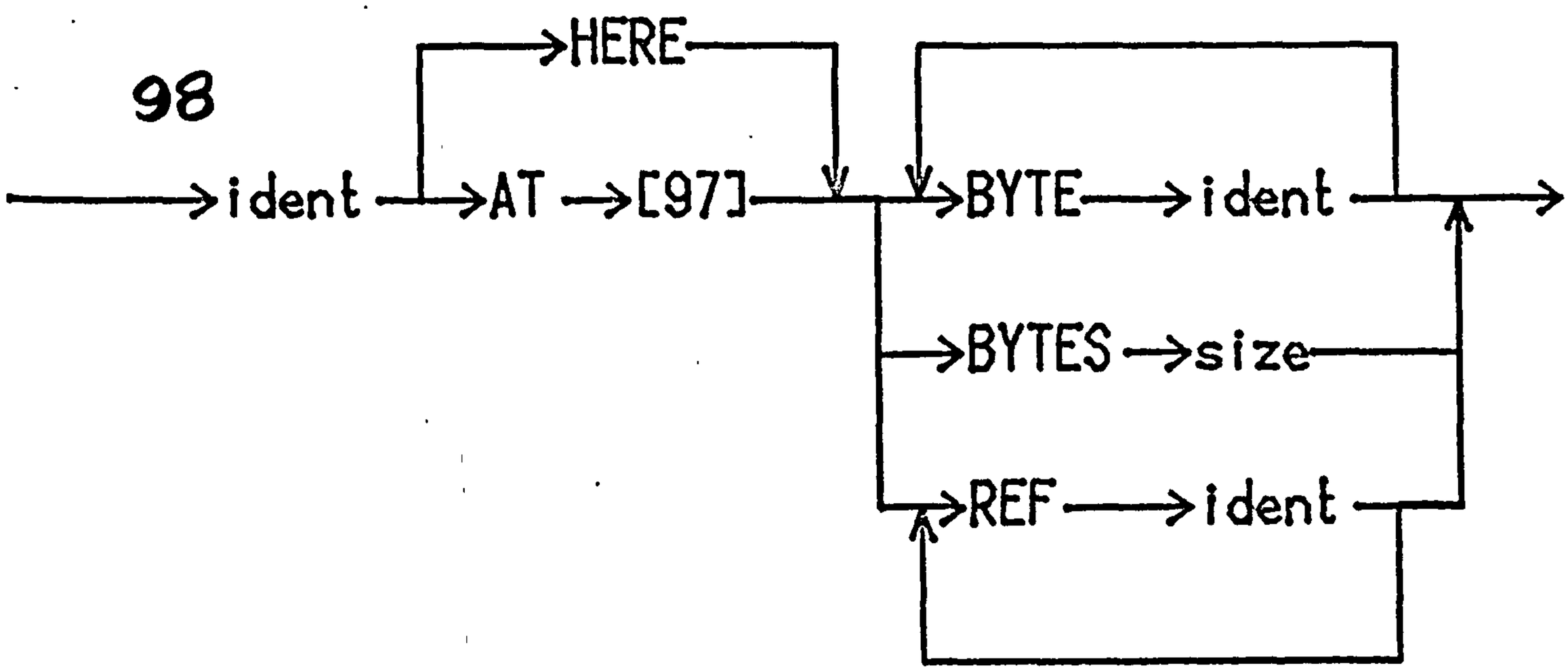
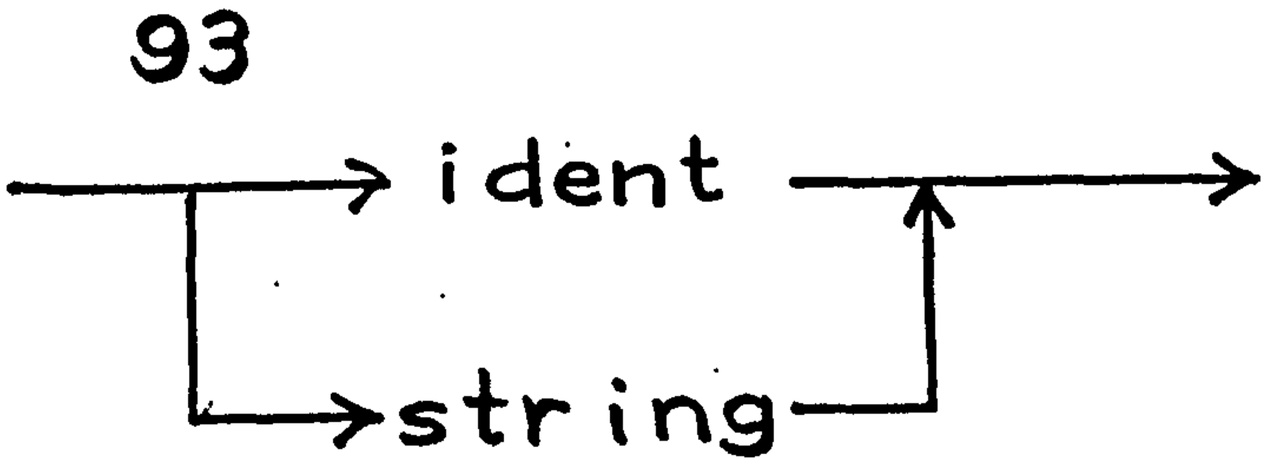


FIGURE 3b Matrices 98, 93 and 97 for AML/1

98



93



97

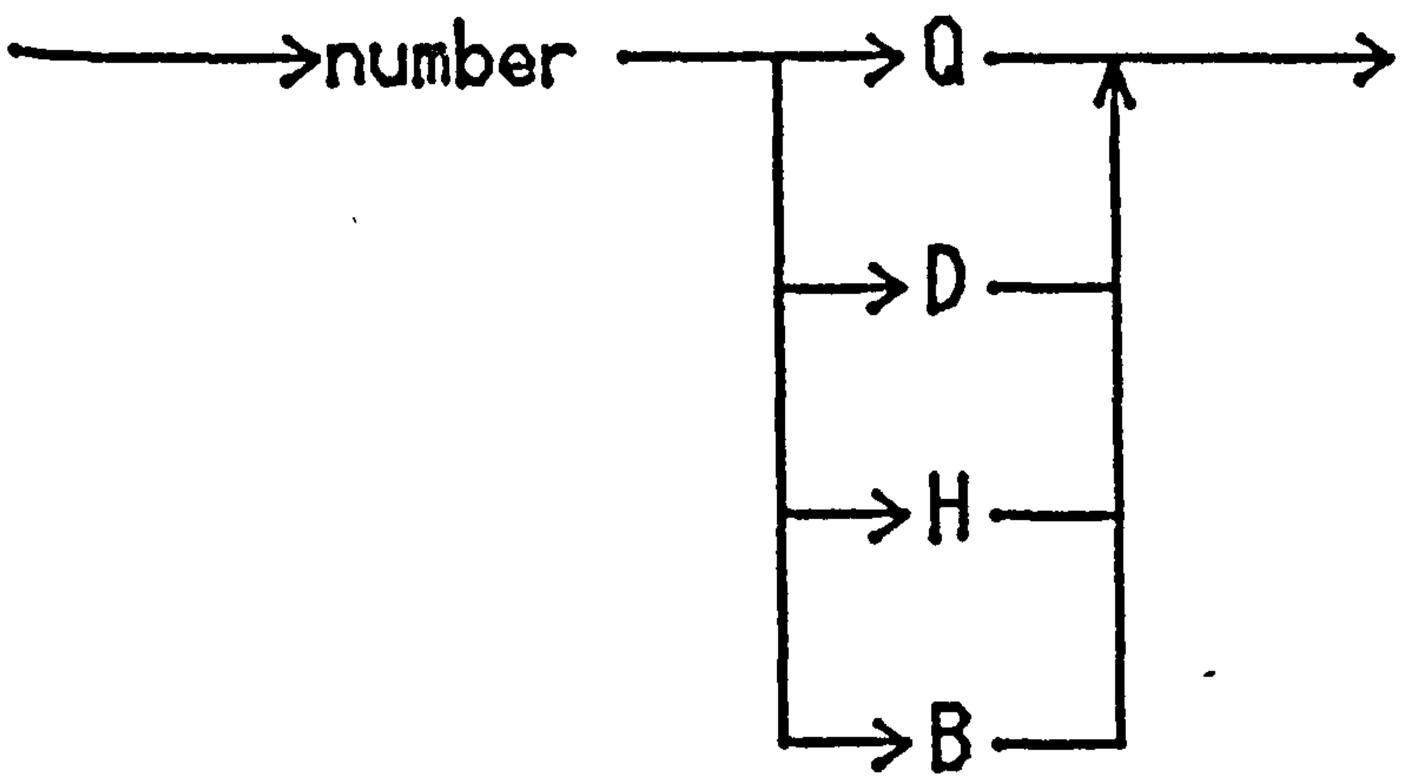
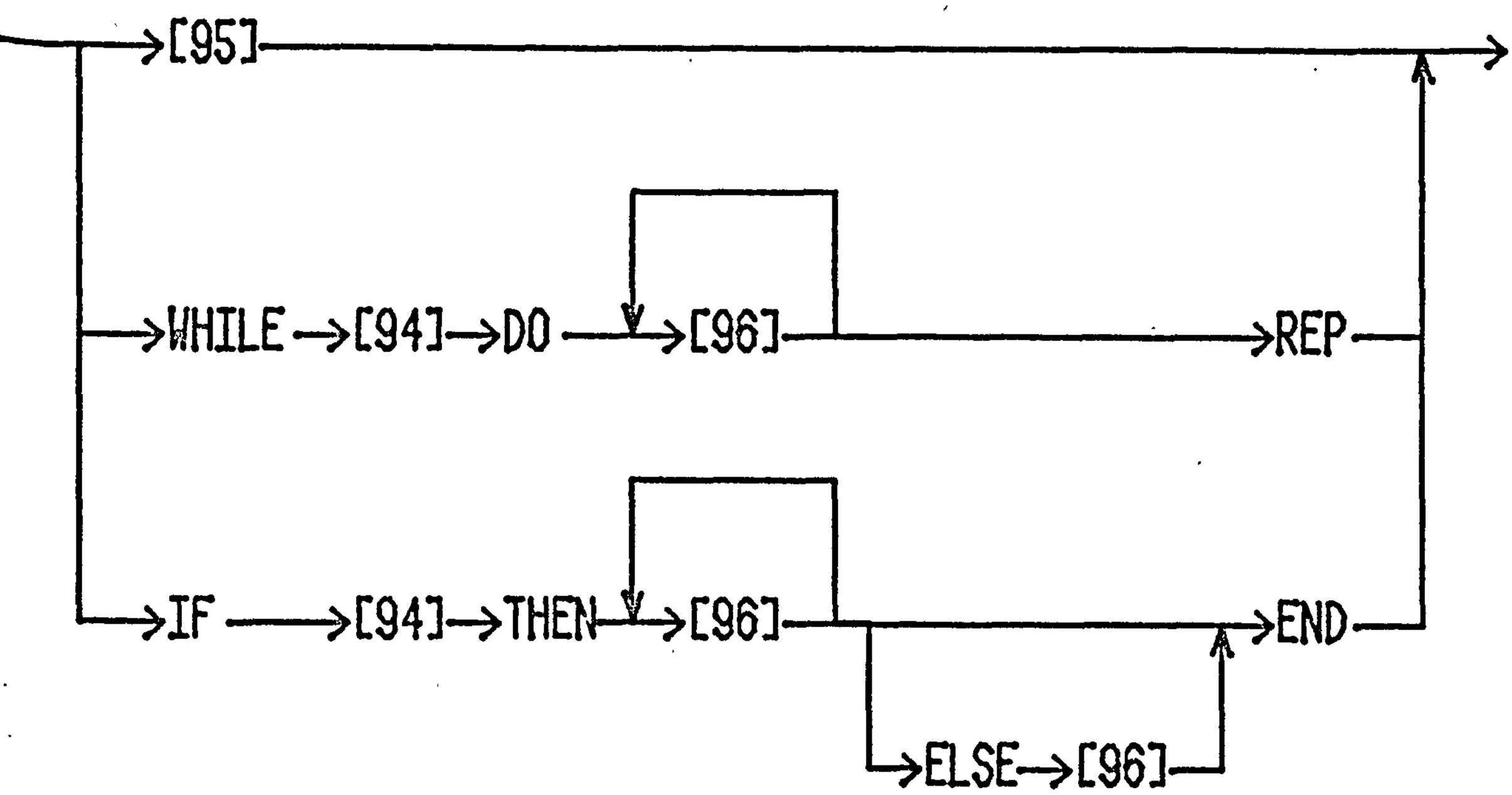
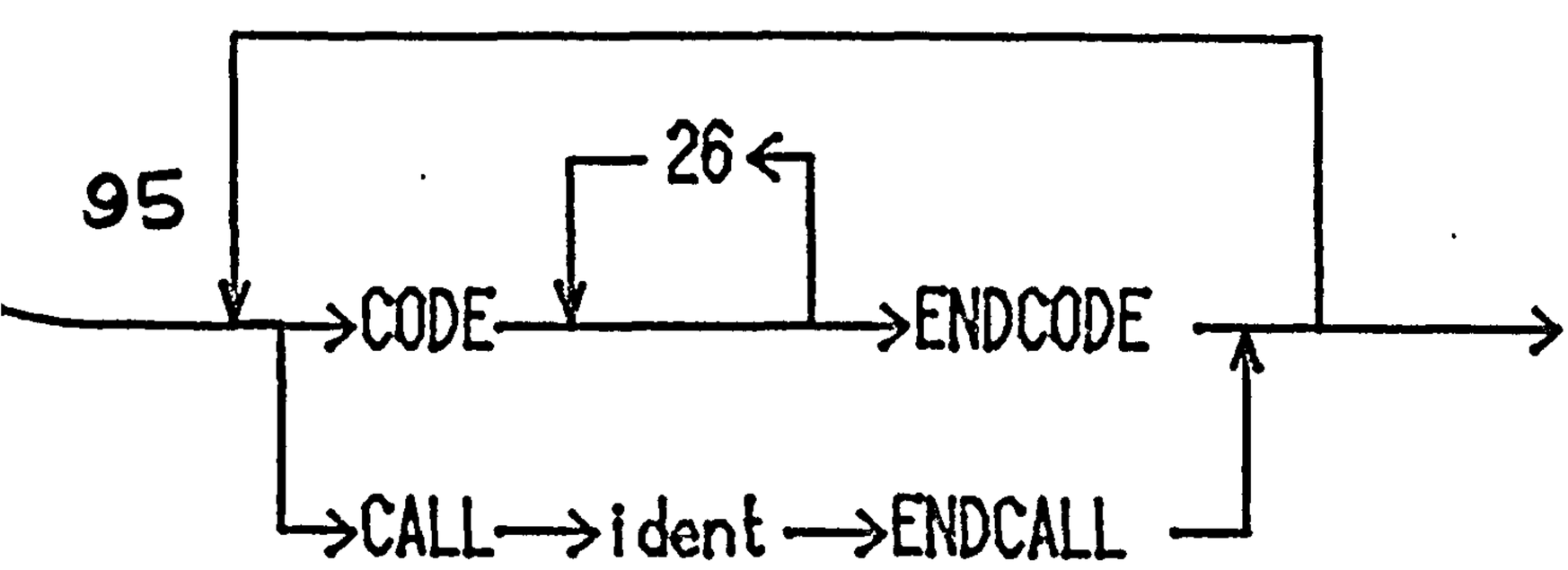


FIGURE 3c Matrices 96, 95 and 94 for AML/1

96



95



94

The terminal symbols are represented by their denotations and non-terminals by placing their name within square brckets,([]). Graph number 99 shows the production of the distinguished symbol of the language.

To every graph we assign a code number starting from 99 downwards. Code 99 is devoted to the main graph and the allocation of graph codes to the other graphs is otherwise free.

The table below shows the codes associated with these seven nonterminals:

	Nonterminal	Graph code
1)	Program	99
2)	Declaration	98
3)	Location	97
4)	Statement	96
5)	Simple statement	95
6)	Conditions	94
7)	Size	93

In each square box of the above graphs also is written the appropriate graph code next to their nonterminal names. The next table is also useful which shows the number of occurrences of each

nonterminal in the other graphs.

Syntax Graph	Number of Occurrences
99	-
98	once in 99
97	once in 99 and once in 98
96	twice in 99 and three times in 96
95	once in 96 and once in 94
94	twice in 96
93	once in 98

CHAPTER 4: SYNTAX & ERROR RECOVERY

In this chapter we discuss the use of syntax graphs, and the intermediate step in their encoding, the transition matrix. We give the algorithm used to check syntax graphs, supported by an example before discussing error recovery and its encoding.

SYNTAX GRAPH

We always represent the given syntax of a language as a set of graphs, the so called "syntax graph", as in [WIRTHb].

We assign a label to each graph, from 99 down to 50. This allows for 50 graphs.

Label 99 is devoted to the main graph. Allocation of labels to the other graphs is otherwise free, but conventionally from 98 downwards.

It is useful to make a table of occurrence of nonterminals in the set of graphs. This has been done for the graphs of language AML/1 in the previous chapter.

This table shows that the graph 98 is only called once in the graph 99 and it is thus more efficient to include it in that graph and have one graph fewer.

The set of elements, terminals and nonterminal symbols, appearing in a syntax graph is the "valid elements" of that graph and consequently, the union of all of these sets is the set of "language symbols".

Valid elements of a syntax graph are connected by directed lines. For each point of these lines there exists at least one destination. Two points having the same destinations are called "the same condition points". A piece of line consisting of such points is called a "state-line".

While in a state, the set of elements which may appear as the next terminal or nonterminal in the input stream, to cause a change of state are valid elements of that state. We have a function to take the state number and return these valid elements. We will see this function later.

For each syntax graph there exists one "initial line" and one or more "exit lines". The initial line is the line which comes to the graph and terminates at one or more alternative elements within the graph. (We call these elements the "initial symbols" of that graph.)

We allocate number "0" to the "initial line" and call this the "initial state" and EXIT to exit lines and call each of them an "exit state". The allocation of state numbers to state lines is otherwise free.

Syntax graphs have to be translated into a data structure. To do this in the KHAR system, we use the transition matrix as an

intermediate step in preparing the input to the translator.

Transition Matrix

In order to translate syntax graphs into their appropriate data structures we first interpret each graph to a transition matrix. This is best explained through an example. Therefore we translate one graph from AML/1, say syntax graph 98, to its corresponding transition matrix.

We refer to directed lines as states and these are already numbered on the graphs, so we know how many states we have, say m. Also we can count the number of valid elements of the graph, say n. First we draw an m X n matrix. Assign valid elements to the columns and state numbers 0 to m-1 to rows. In this particular example we have 7 states, 0 to 6, and 6 valid elements, vis:
"BYTES", "BYTE", "REF", "IDENT", "STRING" and "ENDDATA", since "size" (matrix 93) is "IDENT" or "STRING".

A line leaves each valid element of each state. This line is either an "exit line" or a "state line". In case of a state line, it has a state number and, according to the state we are in, we call this the "next state".

So, for each state i, we have some valid elements and, corresponding to each valid element, there is a next state or exit code. In our example the valid elements of state 0 are

"BYTES", "BYTE" and "REF" and the corresponding next states are 1, 3 and 5 respectively.

To fill the elements of row i of a transition matrix we put the corresponding next states under the appropriate columns. Then the set of elements corresponding to these columns are valid elements of that state. The other elements of this row would be empty. We do the same for all rows and we have a transition matrix for that graph. We associate the graph number with its transition matrix and call it the "matrix number". The corresponding transition matrix of graph 98 would be the following matrix.

	BYTES	BYTE	REF	IDENT	STRING	ENDDATA
0	1	3	5			
1				2	2	
2						EXIT
3				4		
4		3				EXIT
5				6		
6			5			EXIT

(fig 4, transition matrix number 98)

This is not the final form of our transition matrices. There is a "syntax-error action part" for each state to be used for error recovery and also associated with each valid element of each state there would be some semantic and code generation actions.

These actions will be added to the above example matrix after we have discussed these topics.

However, the linearised form of the graph can now be shown as:

```
98 [ 0 (BYTES>1;BYTE>2;REF>5);
     1 (IDENT>2;STRING>2);
     2 (ENDDATA>EXIT);
     3 (IDENT>4);
     4 (BYTE>3;ENDDATA>EXIT);
     5 (IDENT>6);
     6 (REF>5;ENDDATA>EXIT) ]
```

This syntax corresponds to code such as (scanning from "|"):

```
DATA string | BYTES "A CHARACTER STRING" ENDDATA
DATA space  | BYTES size ENDDATA (* size is a manifest constant *)
DATA work1  | BYTE a BYTE b BYTE c BYTE d ENDDATA
DATA refs   | REF index REF top_of_stack ENDDATA
```

We also show where actions can be placed by showing a code-emitting action and an error recovery action. This latter is the worst possible, it just stops the scan. Semantic actions are discussed in chapter 5 while we consider Error Recovery in the second part of this chapter.

We show part of this linear form with error recovery actions added as well as code emitting actions in the following diagram.

```
98 [ 0(BYTES>1;BYTE>2;REF>5 ? STOP);
     1(IDENT>2|emit(" rmb ",css.);
       STRING>2|emit(" fcc /",css.,"/",nl," fcb 0" ? STOP);

     ......
```

"STOP" is the error recovery action, placed after a "?" at the end of the state. The verb "emit(.....)" is one of the semantic and code generating actions we have access to in KHAR through the SL language. As we show in chapter 5, "STOP" is correct in state 0, which only needs a dummy action, but it is of no use in state 1. The correct action is given in chapter 5.

VALID ELEMENTS OF A TRANSITION MATRIX

The valid elements appearing on the top of columns in transition matrices fall in one of the following groups :

1) language keywords: these elements appear exactly as they are and would be coded automatically by a program using the language keyword file;

2) nonterminals or transition matrices: these are already coded to matrix numbers 99 to 50 as mentioned before;

3) identifier:

4) constant string:

5) integer:

6) nil symbol:

7) anything symbol.

For groups 3 to 7 above we put the fixed code obtained from the table of fixed codes, (chapter 3), into the matrix.

If we have fixed code 26 (anything-symbol) in a transition matrix, this is placed in the rightmost of the used entries in a row, so that it is encountered after any specific symbols expected in that

state.

CHECKING TECHNIQUE USED IN KHAR

For each language we have one or more matrices: of these one is the main matrix, matrix 99. The matrices are used by KHAR to check the syntax of any source program written in the corresponding programming language.

For each matrix we have one entry point, the first state of that matrix but there may be an exit from any point in any state of that matrix. Associated with any valid element, terminal or nonterminal, within any state there is a pointer to another state of the same matrix.

The program execution begins with access to the first valid element of the first state of the main matrix, and stops by exiting from this same matrix. The other matrices may be called, directly or indirectly, from this main matrix.

We have two types of valid elements; either they are terminal symbols in which case the source symbol will just be checked with that, or they are nonterminals (a matrix-number). In the latter case before entering this new matrix, the source symbol will be checked with the initial symbols (the director symbols) of that matrix and if it does not match with any of them the next valid element of the current state will be checked against the current source symbol but if a match occurred, checking continues by pointing to the beginning of the new matrix. Upon the exit from this new matrix, access is

regained to that point of the previous matrix from which access was transfered. This recovery is achieved by using a stack for the accessing information. Information is pushed onto the stack on entering a matrix and popped off on exit.

The process always expects the source symbol to match with one of the valid elements of the current state. If a match occurs the next state is the state indicated by the entry after that valid element. Another source symbol is read and access is transfered to the new state and the same process continues until the "end of file" occurs or the source symbol does not match with any of the valid elements of the current state, and an error occurs.

As soon as an error occurs scanning of the input text is no longer controlled by the syntax graph alone. The error actions accessed by placing verbs in the error action entry in the table direct the scanner to take the error action chosen by the language designer.

The errors are caused by unexpected, missing or wrongly spelt symbols. A good compiler should find all errors in a source program and correct as many of them as possible to reduce the number of submissions of a job before it is finally completed. For other errors which the compiler can not correct, it should be able to determine how to continue the analysis when these errors occur. For this process the term "Error Recovery" is used.

In this system we have one general action and some special actions which we use as the error action part of states in transition

matrices.

The error actions can be general for all the different states of all the matrices or different error actions can be added to each state. In the latter case usually the recovery is quicker than the first case.

Each state terminates with an error action part, which also serves as the "end of state" marker. That is, we check the source symbol with valid elements from the start of the state and if it does not match we check it with the next valid element in the table and carry on until we reach the error action part. Then we know the symbol has been checked with all the valid elements of the current state and an error has been detected.

As we mentioned earlier, when the expected symbol is a matrix number the program syntax checker, before entry to this new matrix, checks the current symbol with the valid element of the first state of that matrix and enters if and only if a match occurs. It is obvious here that when we enter a new matrix the current source symbol will definitely match with one of the valid elements. In other words, we can not have an error so that an error part is not needed. The main matrix is an exception to this. Conventionally, we use "STOP" as a dummy action to satisfy the syntax of SL.

We give on the following page the formal algorithm.

FORMAL ALGORITHM

```
push entry point to matrix no. 99 onto stack;
read first symbol; end of program:=FALSE;
WHILE matrix on stack AND NOT end of program DO
BEGIN
  WHILE NOT exit AND NOT end of program DO
  BEGIN
    IF entry in transition data under pointer indicates end of state THEN
    BEGIN
      IF NOT looking for director symbol in first state of matrix THEN
      BEGIN report error and call error recovery action END
      ELSE
      BEGIN
        return (via stack) to point of departure in transition data
        from which entry was made to this matrix and access next entry;
        pop stack
      END
    END
    ELSE
    BEGIN
      IF symbol under pointer into transition data ( the expected symbol )
         is a matrix code THEN
      BEGI
        IF this stack is not already on the stack as unsatisfied goal THEN
        BEGIN
          push value of pointer onto stack;{used to recover this point of
          departure into matrix} access new goal matrix via index table to
          matrices;{record another matrix as an unsatisfied goal on the stack}
          level:=level + 1
        END
        ELSE access next expected symbol in this state
      END
      ELSE {not matrix code, could match current symbol}
      BEGIN
        IF current symbol matches symbol under pointer, the expected symbol THE
        BEGIN
          IF next state NOT EXIT THEN access next state, finding new symbol
                                 ELSE exit:=TRUE;
          level:=0;{sets all matrices on stack as satisfied goals}
          read next symbol
        END
        ELSE
          access next expected symbol in current state
      END
  END
  END;
  IF NOT end of program THEN {exit from matrix has occured}
  BEGIN
    recover point of departure from stack; pop stack;
    IF next state is EXIT THEN exit:=TRUE
    ELSE
    BEGIN
      access next state to find an expected symbol; exit:=FALSE
    END
  END
END.
```

AN EXAMPLE

We describe the action of the algorithm for the small matrix described above. Consider the following program which satisfies the syntax of AML/1, but has no practical meaning, and make reference to the transition matrix given on the next page, which is part of the encoding of graph 99 for AML/1.

```
PROGRAM example HERE
DATA work_space HERE
  BYTE work_1
ENDDATA
MAIN
  CODE ENDCODE
ENDPROGRAM
```

	PROGRAM	IDENT	AT	HERE	97	DEFINE	DATA	PROC	98	MAIN		
0	1											
1		2										
2			3	4								
3					4							
4						5	8	13		23		
5		6										
6												
7					4							
8		9										
9			10	35								
10					11							
11									12			
12							8	13				
13												

		96	ENDPROGRAM	
23		24		
24		24	EXIT	

33												
35									12			
36												

The index to the matrix number 99 is pushed onto the stack and the first symbol "PROGRAM" is read. The outer WHILE loop is entered since neither condition is FALSE, note that "end of program" is set

TRUE by encountering the "end of file" sentinel code while expecting another language symbol: it is an error condition. The inner WHILE loop is also entered. The entry in the transition data first accessed corresponds to PROGRAM so the ELSE part of the IF statement is taken. As the symbol under the pointer is PROGRAM, the ELSE part of the IF statement in the ELSE part is taken. The first symbol read matches PROGRAM so the next state entry is examined. This is the same as the entry in the transition matrix under PROGRAM so that the next state is 1. The THEN clause of "IF state NOT EXIT" is taken so that this next state is accessed.

The next symbol "example" is read and the inner WHILE loop repeated since both conditions are still TRUE. The flow of control follows the same path as just described and "example" matches IDENT. This moves access to state 2 and the same flow occurs until the "IF current symbol matches symbol under pointer" statement is reached, when the ELSE part is taken and the next expected symbol in the current state is accessed. The loop is repeated but now a match occurs, state 4 is accessed, and the next symbol, "DATA", is read.

The loop is repeated, with a match occurring which takes access to state 8, and the next symbol is "work_space"; the loop is repeated, "work_space" matches IDENT, stae 9 is accessed and "HERE" is read; on looping, HERE does not match "AT", the first expected symbol so the next symbol in the current state is accessed, and on looping, this does match so state 35 is accessed, and the next symbol BYTE is read.

On looping, the ELSE part of the first "IF" is taken again but the THEN part of the next "IF ..... matrix code" statement is obeyed.

The point of departure is pushed onto the stackand the first state of matrix 98 accessed via the the index table. The register, level, is increased by one to indicate that a matrix which is a potential goal has just been placed on the stack. The loop is repeated so that the usual ELSE...ELSE route is followed. This does not result in a match, as the first expected symbol is BYTES so the next expected state is accessed. The loop is repeated and results in a match as this symbol is BYTE. The next state is not exit so the next entry is used to indicate state 3; level is set to zero showing that a director symbol of matrix 98 has been found, and the next symbol, "work_1", is read.

The loop is repeated resulting in a match with IDENT and the accessing of state 4. The next symbol is now ENDDATA. The loop is repeated, with no match, since BYTE is expected. The next symbol in state 3 is therefore accessed, which is ENDDATA so that on the next loop a match occurs. But the next state is EXIT so exit becomes TRUE and the next symbol MAIN is read.

The IF NOT end of program encountered on leaving the inner WHILE loop has its THEN clause obeyed, so the point of departure is recovered from the stack which is then popped. The next state entry associated with this entry point to 98 is examined and is 12. Access is made to the first expected symbol in this state, DATA. Since the entry point to 99 is on the stack and end of program is FALSE, the outer WHILE loop is repeated; the inner WHILE is entered and a match made with MAIN in state 12. The state 23 is accessed and the next symbol read, CODE.

On looping, the expected symbol is found to be a matrix number,

96; the departure point to 98 is stacked, level increased by one, and the first expected symbol, WHILE, of matrix 96 accessed via the index table. On looping, there is no match, and the expected symbol IF accessed; again, no match is found, and matrix 95 is accessed as the next expected symbol. On looping, the matrix departure point is stacked, level increased and the first state of 95 accessed, giving CALL as the expected symbol.

On looping, CALL is not matched, as the current symbol is CODE, and access made to CODE. A match occurs on the next loop and results in 3 becoming the next state. The next symbol is read, ENDCODE, and on looping, matches since this is the only expected symbol in that state. The state code after ENDCODE is EXIT so exit becomes TRUE, and the next loop exits from the inner WHILE loop, level having been set to 0 to show that matrices 96 and 95 are now satisfied goals, that is, a member of a director set has been matched.

The point of departure into 95 is recovered and the next state entry for 95 is found to be EXIT; exit becomes TRUE. On looping, the same flow of control occurs, so that the next state entry after 96 is accessed, the stack popped, but the entry is 24, does not equal EXIT, and the loop is obeyed again, with only 99 left on the stack, exit being FALSE. The symbol to be found, however, is matrix 96 so the complete process described above from "On looping, the expected symbol is found to be a matrix number" is repeated until CODE becomes the expected symbol. Since ENDPROGRAM is the current symbol, there is no match, and the next expected symbol is accessed. As this is 0, end of state, on looping, the "IF ... end of state" statement has its THEN clause executed (for the first time) and the "IF NOT looking for

director symbol" statement obeyed. Since we are seeking a director symbol, the ELSE clause is executed, and the point of departure into 95 recovered from the stack, which is popped and level is decremented (so that only one matrix remains as an unsatisfied goal) and the next entry accessed as the expected symbol. On looping, this "IF NOT" statement is obeyed again since the end of state marker, 0, lies under the pointer. The departure point into 96 is recovered and the next entry in that state accessed, the stack being popped and level decremented, becoming 0. The accessed expected symbol is ENDPROGRAM so, on looping, a match occurs. The next state is EXIT, so exit becomes TRUE, and, on looping, the inner WHILE loop is left.

As exit caused the termination of the loop, the "exit from matrix" part of the "IF NOT end of program" statement is obeyed. The point of departure is recovered, the stack is popped and becomes empty. The accessed next state is EXIT so exit becomes TRUE. On looping the outer WHILE loop is left since the stack is empty, and execution concludes with reporting that the analysis is finished.

ERROR RECOVERY

A possible implementation of error recovery could be made by writing the appropriate error message and accessing an indicated state in an error recovery table where the error recovery process reads source symbols one by one, ignoring them until a specific symbol is read. For example ignoring symbols until the end-of-statement symbol is found. This is often called "panic mode" recovery. In this case it fails to report further failures in that statement, if any, and also may cause many other errors. For example if an error occurs in "VAR"

statement in PASCAL program and we ignore to the next semicolon at the end of that statement, we have ignored some identifiers and wherever they appear in the rest of program they are undefined and will cause new errors. Sometimes in many compilers it occurs that because of a single error several error messages will be generated which should be suppressed in the error recovery process.

After detecting an error a good compiler should try to determine what correct symbol had been intended initially. For example if in a PASCAL program the reserved word VAR be misspelt, usually all the identifiers will be undefined, whereas it could be checked for a misspelling of one of the reserved words valid at that state and there is a good chance it would be corrected in the right manner.

In our error recovery, we have one general action and some specific actions. Each of these actions may be used for each of the different languages we implement. These error actions are, in effect, some very flexible tools which may be employed. These tools are like features of a programing language, in that they are very flexible. Because of the simplicity of KHAR and the flexible interface provided via the SL languages, one can simply add his own new features to the system and use them.

These error actions are accessed by calling an action interpreter at the point in the formal algorithm where an error is detected. Either one of the specific actions or the general action may be called. Their action is described in the following sections.

SPECIFIC ACTIONS

Reserved words, or "verbs", in the SL languages are used to call the error actions of the KHAR machine. They are as shown in the following table.

Action	Meaning
GO	go to error state, see below
STOP	stop checking
SUCC	point to next source symbol
LOOP	go to another state, stay in that state, read and ignore source symbols until one of the valid elements of that state appears in the input stream, then go to appropriate state.

AN EXAMPLE OF THEIR USE

The usage of these error actions will become clear by an example. In our previous example we made a transition matrix from syntax graph no. 98 of AML/1. This matrix is not complete and in case of error the program syntax checker would stop. It needs some more information to be able to continue. Therefore we add error actions to each state of that matrix. We discuss the action added for each state seperately.

State 0 :

There will be no error in the first state of our transition matrices. For these kinds of states which do not have errors we place the action STOP as a dummy error action to satisfy the syntax of SL.

State 1 :

In this state we expect either "identifier" or "string". If any error happens we check for ENDDATA or one of the elements valid after exiting from this matrix. As table no.4 shows, the graph of this matrix is called only once in graph 99 and by refering to that we see that the follow elements of this matrix are

DATA PROC INTERRUPT MAIN

So we have a set of symbols and corresponding to each of them there is a next state. We read and ignore the source symbols until one of the elements of this set appears, then we know our next state. In the case of ENDDATA we access state 2 of this matrix. ENDDATA is a valid element in that state and its corresponding state is EXIT which means exit from this matrix.

In the case of the other elements of the set we force exit from this matrix. To this effect we introduce a new column for the "nil" symbol (code 25) and place a new state 7 in the matrix. Its only valid element is this "nil" symbol for which the next state is EXIT.

The "nil" symbol causes a refinemement in the behaviour of

KHAR. "Nil" matches any symbol but no new symbol is read as the next symbol. Thus the use of "nil" makes a key difference in the treatment of ENDDATA and the other keywords on which recovery is made. Since a match with "nil" occurs after the scanner has read one of these, this symbol is still the current symbol and may be used to satisfy the syntax of the outer graph from which 98 was called.

State 2 :

In this state we expect the "ENDDATA" symbol and the corresponding next state is EXIT. But if the current symbol did not match we will not loose much if we go out of this matrix and leave the error recovery to take place in the matrix from which we were sent here. To do this we just add an exit code under the column of the "nil" symbol so that if there is no match with ENDDATA, exit will occur. This state will never produce error, so the error recovery part of that is the same as state '0', that is STOP.

State 3 :

If any error happened in this state we search for BYTE and go to 4, ENDDATA and go to state 4, or for one of the follow symbols (as the error action part of state 2) and go out of this matrix.

State 4 :

The same as state 3.

State 5 :

Very similar to state 3 but we replace BYTE by REF.

State 6 :

We take the same action as state 5.

State 7 :

There would be no error in this state as its only element is the "nil" symbol.

To put these error actions in the table we place the appropriate language keywords and SL verbs and symbols in the linearized form of the matrix, as shown briefly in chapter 3. But at this stage to show these error actions in the matrix we use symbol '>' to link symbols with their next state and ',' to separate the alternative symbols. So the error action part of state 1 can be written as

"ENDDATA>2, DATA>7, PROC>7, INTERRUPT>7, MAIN>7"

and, after putting the error actions into the matrix 98, it would be

as follows:

	BYTES	BYTE	REF	IDENT	STRING	ENDDATA	25	
0	1	3	5					STOP
1				2	2			ENDDATA>2, DATA>7,PROC>7, INTERRUPT>7, MAIN>7
2						EXIT	EXIT	STOP
3				4				BYTE>4, ENDDATA>4, DATA>7,PROC>7, INTERRUPT>7, MAIN>7
4		3						BYTE>4, ENDDATA>4, DATA>7,PROC>7, INTERRUPT>7, MAIN>7
5				6				REF>6,ENDDATA>6, DATA>7,PROC>7, INTERRUPT>7, MAIN>7
6			5					REF>6,ENDDATA>6, DATA>7,PROC>7, INTERRUPT>7, MAIN>7
7							EXIT	STOP

ERROR STATES

An alternative implementation of adding error action parts to transition matrices is to add "error states" to the matrix, and to add the set of follow symbols to the head of the matrix. Since these matrices are an intermediate representation between graph and encoded form, we used a compressed representation which shows these additional

states and symbols, together with any of the ordinary symbols of the matrix used in error recovery in an "error matrix". Access is made to the indicated error state when an error occurs, and the entries in these states direct the scanning process back to a normal state in which recovery will occur.

In the following diagram we show matrix 98 together with its error action matrix in complete form.

	BYTES	BYTE	REF	IDENT	STRING	ENDDATA	
0	1	3	5				STOP
1				2	2		GO>7
2						EXIT	GO>8
3				4			GO>9
4		3				EXIT	GO>9
5				6			GO>10
6			5			EXIT	GO>10

	ENDDATA	BYTE	REF	DATA	PROC	INTERRUPT	MAIN	26	25
7	2			EXIT	EXIT	EXIT	EXIT	7	
8									EXIT
9	4	4		EXIT	EXIT	EXIT	EXIT	9	
10	6		6	EXIT	EXIT	EXIT	EXIT	10	

Note that the "anything" symbol, 26, is always used to force looping in an error state until a symbol is read and matched.

To send the pointer to an error state after error detection we

have action "GO". For example at the end of state 2 we have "GO>8" which means "go to state 8 for error recovery".

Separating error action parts from the states and putting them into error states is useful when the same action patrs are repeated for several states. We can adopt a mixed policy. If an action is used once we leave it in the state but if it is used several times we introduce an error state and use the GO verb.

GENERAL ACTION

The use of error states and the GO verb allow the user of KHAR to introduce the kind of error recovery used in [AMMAN], an improved form of "panic action" in which a set of symbols is kept in existence on which recovery may occur.

The technique of [AMMAN] is such that recovery can only take place on a symbol within the current non-terminal or on a symbol within a non-terminal from which this one was called. Thus substantial sections of valid text can be skipped in certain circumstances.

The general action uses an appropriate normal state within the matrix as an error recovery state. This is possible because of KHAR's rigid distinction between syntax and semantics. The verb LOOP is used to access a state for use in this way. The use of the state is exactly as in the normal syntactic scan except that encountering the end of state is not the signal for taking error action, but to read a new next symbol and repeat the scan from the beginning of the state.

Since this state may contain non-terminals, and so on, the follow set on which recovery may occur is augmented automatically by all the director symbols of these nonterminals.

SUMMARY OF ERROR ACTIONS

USE n

the state "n" should be an error state;

LOOP n

"n" is any normal state within the matrix, not an error state;

GO n

this action forces access to another state without recovery;

EXIT

forces exit from the present matrix to the calling point, with recovery left to the higher level;

STOP

stops the process of syntax checking;

SUCC

read the next symbol and resume scanning as indicated.

CHAPTER 5: SEMANTIC PROCESSING & CODE GENERATION

We have already outlined the approach we take in the KHAR system. In this chapter we discuss the actual mechanism used in KHAR to handle semantics (or context sensitivities). We then show how these are applied by showing their use in the production of a compiler for PL/0. This covers most of the semantic problems encountered in a block structured language but not the propagation of attributes up the abstract parse tree of an arithmetic expression, as PL/0 has only integer variables. We illustrate this problem by considering the semantics of expressions in a language such as ANSI FORTRAN.

THE SEMANTIC MECHANISMS OF KHAR

The expression "semantic mechanism" was used in [CORDY] to cover a semantic data structure and the operations upon it. His first example of such a mechanism is the symbol table mechanism. He asserts that

1) it is universally used,

2) contains name of object,

3) contains its data type,

4) indicates its structure, variable, array, procedure, etc.,

5) contains addressing information, and

6) contains auxiliary information, dimensionality or number of parameters.

Typical operations upon the symbol table are given as

1) enter name,

2) enter address, and so on, one for each attribute.

This immediately brings us to the difference between the approach of KHAR and that of conventional compilation. In KHAR we do not construct a symbol table. We do have dictionaries of identifiers, strings and constant denotations but the objects handled within KHAR are the integer indices into these dictionaries. Thus KHAR has no symbol table in the sense of [CORDY].

The mechanisms of KHAR are:

1) A symbol stack

this is a stack, the elements of which are integer variables; as we shall see, items may be pushed onto or popped off the stack; the structure is a read-only stack, rather than a push-down store, in which only the top element would be accessible.

2) Current Source Symbol register (CSS)

this is a register whose content is the last symbol read from the input stream by the recogniser, it is used in a read-only manner;

3) A Register (RG)

this is a working register whose contents may be set using the SET verb from a range of sources, or may be incremented or decremented;

4) A Label register (LABEL)

this is used to provide a set of integer values which are used to generate unique labels;

5) A Level register (LEVEL)

this is incremented or decremented to provide a level count within a block structured language, used to generate level,address pairs for the PL/0 machine;

6) An Index register (INDEX)

this register is used to index into the stack, and is set by the use of the SCOPE or SEARCH verbs;

7) Top of stack register

this is used to access the top of the stack, it is not explicitly available to the user of KHAR.

SEMANTIC ACTIONS

We may now describe the semantic actions or verbs which operate upon the semantic structures of KHAR.

MARK

This action increases LEVEL by one and puts "MARK" on the stack.

FLUSH

This action removes all entries down to and including the MARK from the stack, and LEVEL is decremented by one. If

MARK was not found on the stack, the stack pointer is set to 0, and LEVEL becomes 0.

SCOPE, SEARCH & CHECK

These three actions have similar syntax as shown in SL4. Each one has one argument and two sets of actions. One of these two sets of actions will be carried out depending on the result of the SCOPE, SEARCH or CHECK action. "SCOPE" first searches if the argument is matched with one of the elements down to the last "mark" put on the stack, and accordingly, if found the first set of actions is obeyed, otherwise the second set of actions would be done. This verb is used to check for duplicated declarations.

"SEARCH" searches the complete stack for a match with its argument. Otherwise, it behaves just like "SCOPE", and is used to locate declared items on the stack.

Both these actions change the value of INDEX so that values pushed onto the stack next to the matched symbol may be accessed.

"CHECK" just checks the argument; if it is not zero the first set of actions is carried out, otherwise, the second.

POP

This action removes items from the stack. POP by itself removes one item, "POP,0" removes all items, and "POP,n"

removes "n" items.

PUSH

This action places one item on the stack. PUSH RG puts the contents of RG on the stack, PUSH CSS, the current source symbol, PUSH LABEL the current value of LABEL, PUSH LEVEL, the value of LEVEL and PUSH alone the value of the following symbol, an integer value which is either an integer, as written, or the index to an encoded item.

SET

This action alters the value of the working register RG. SET RG pops a value off the stack into RG, SET CSS places the current sorce symbol into RG, SET LABEL and SET LEVEL, the values of LABEL and LEVEL, while SET + n adds "n" to RG and SET - n subtracts "n" from RG. SET alone, as for PUSH, places the value of the following symbol in RG.

SEMANTIC CHECKING IN PL/0

Semantic checking in PL0 is done in three passes. In the first pass we concentrate on "CONST" part, in the second pass on "VAR" part and in the third pass on procedure calls. The linearized SL encoding is given in the supporting material, LISTING OF THE KHAR SYSTEM.

PASS 1 : syntactic processing

This pass checks the syntax of PL/0 and outputs expressions in

postfix notation. We do not show the syntax diagrams seperately as they are repeated with the addition of semantic actions in the following sections. The only actions placed in the linearized coding are the error correcting actions and may be seen by inspecting the appendix.

PASS 2 : dealing with manifest constants

In matrix 98 each time on entering [block], we "mark" the stack and upon exit we "flush" the stack, that is to remove all entries down to and including the "mark". For each identifier, we use SCOPE to check if it is declared already. If it is, we use ERROR to output a message. If not declared at this level, we push the identifier, actually its index, onto the stack. For each constant identifier encountered we push two entries onto the stack, its code and its value. For other identifiers, we push the code and "0", as a "don't care" marker.

When we are in "factor", we SEARCH the stack for identifier entries. If the matching identifier has zero as a value above its code, then it is not a constant in that scope. Remember that constants are represented by indices to their denotations. If the value is not zero, then it is used in EMIT to replace the identifier code, otherwise we emit the current source symbol. Note that we set RG to EMIT so that symbols are copied unless otherwise required by the semantic action.

We report an error if any of the constant identifiers on the stack appears

in "VAR" part,

as a procedure ident in matrix 98,

at the left hand side of ":=" in matrix 97, or

after "CALL" in matrix 97.

We tabulate the actions for the pass and then present the graphs with the index number of the action placed to show where it would be carried out.

Semantic Actions: Manifest Constants

0. set 0

1. mark

2. set css

3. scope rg(error(ln,": ",rg.," declared"); push rg push css;)

4. scope css(error(ln,": ",css.," declared"); push css push 0;)

5. flush

6. search css(;;)
   check %index+1(error(ln,": ",css.," const in LHS");;)

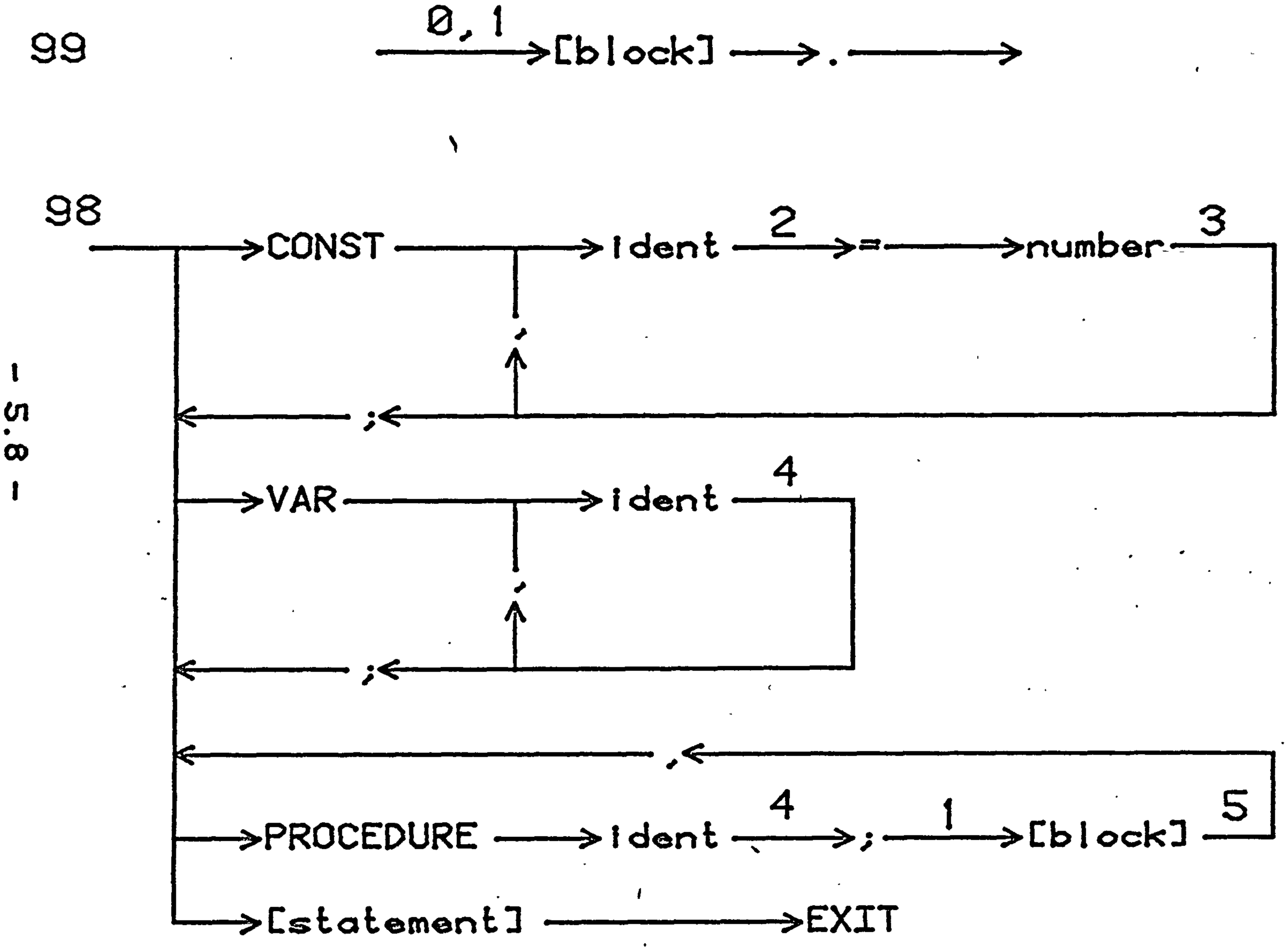
7. search css(;;)
   check %index+1(error(ln,": ",css.," not a procedure");;)

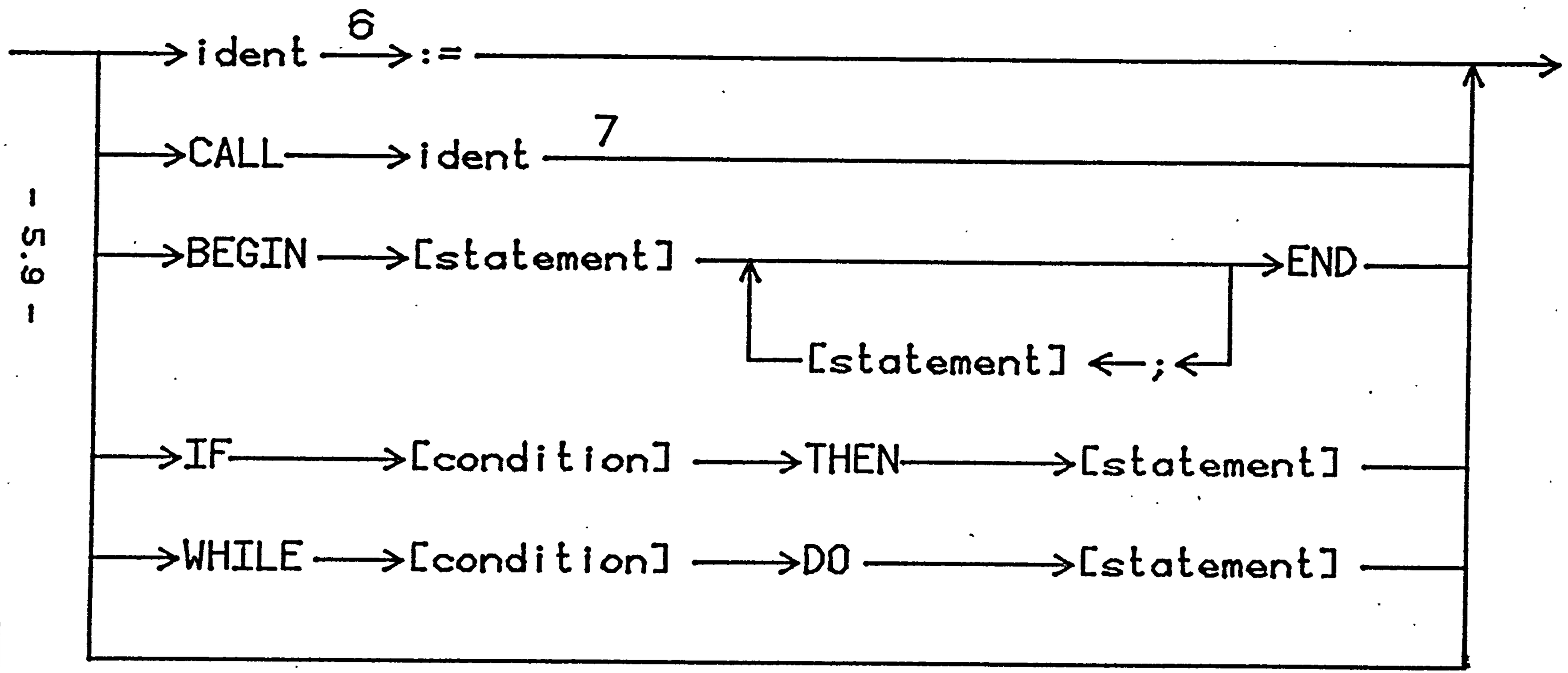
8. set emit

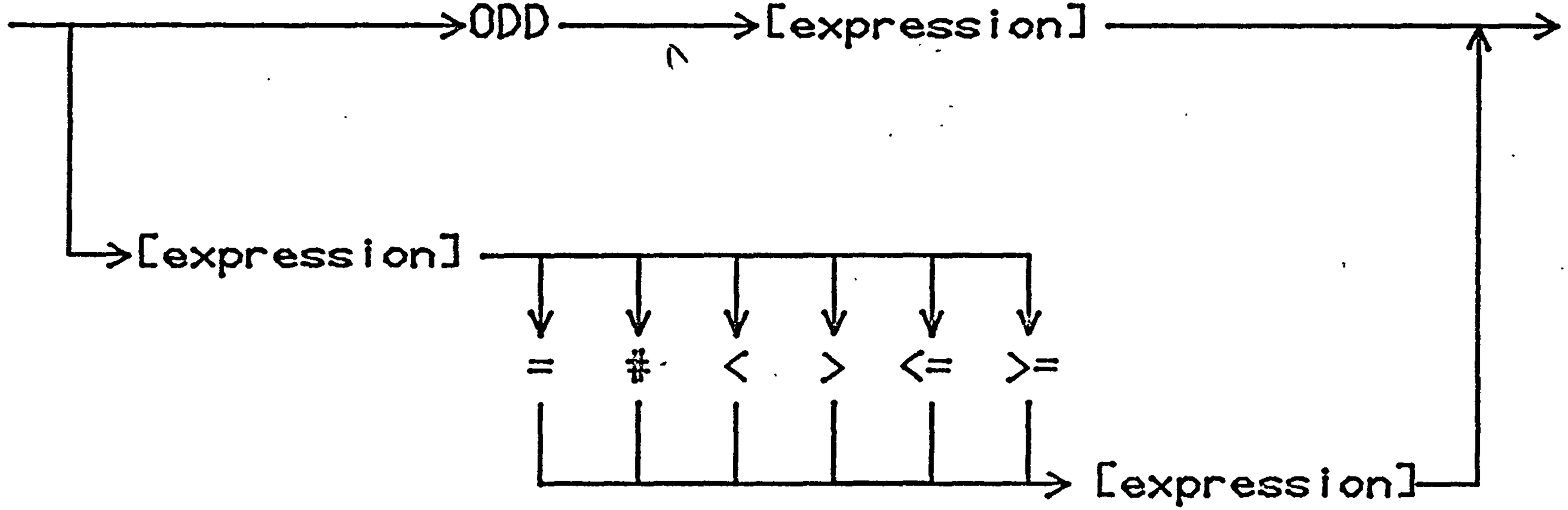
9. search(;;)
   check %index(check %index+1(emit(%index+1);;);;)

10. emit(css)

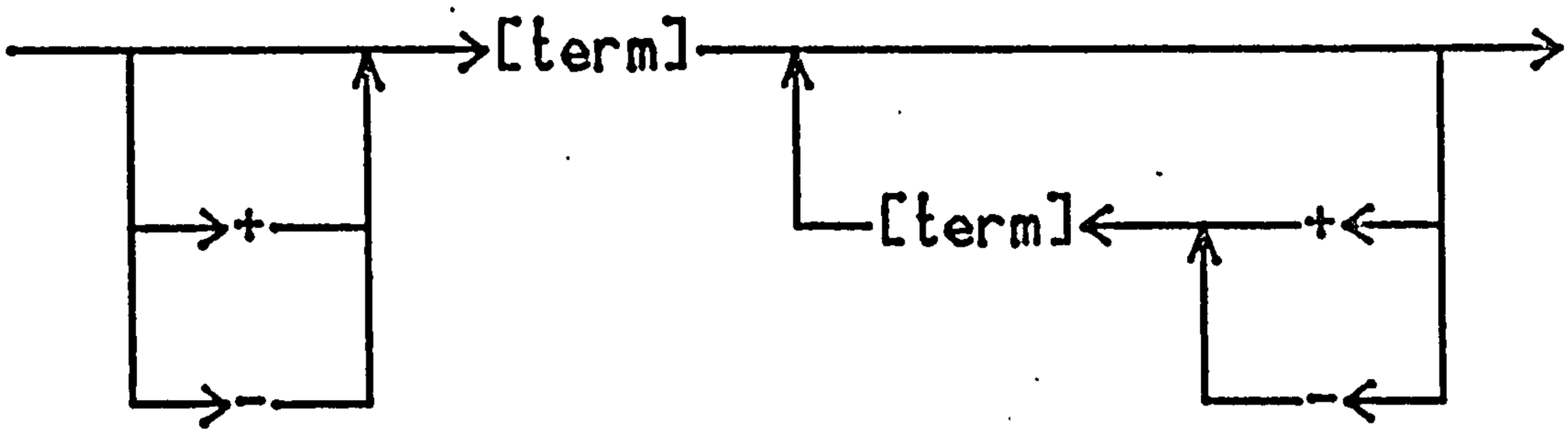
11. emit(css) set emit


99
0, 1
[block]
.
98
CONST
Ident
2
=
number
3
;
VAR
Ident
4
;
,
PROCEDURE
Ident
4
;
1
[block]
5
[statement]
EXIT

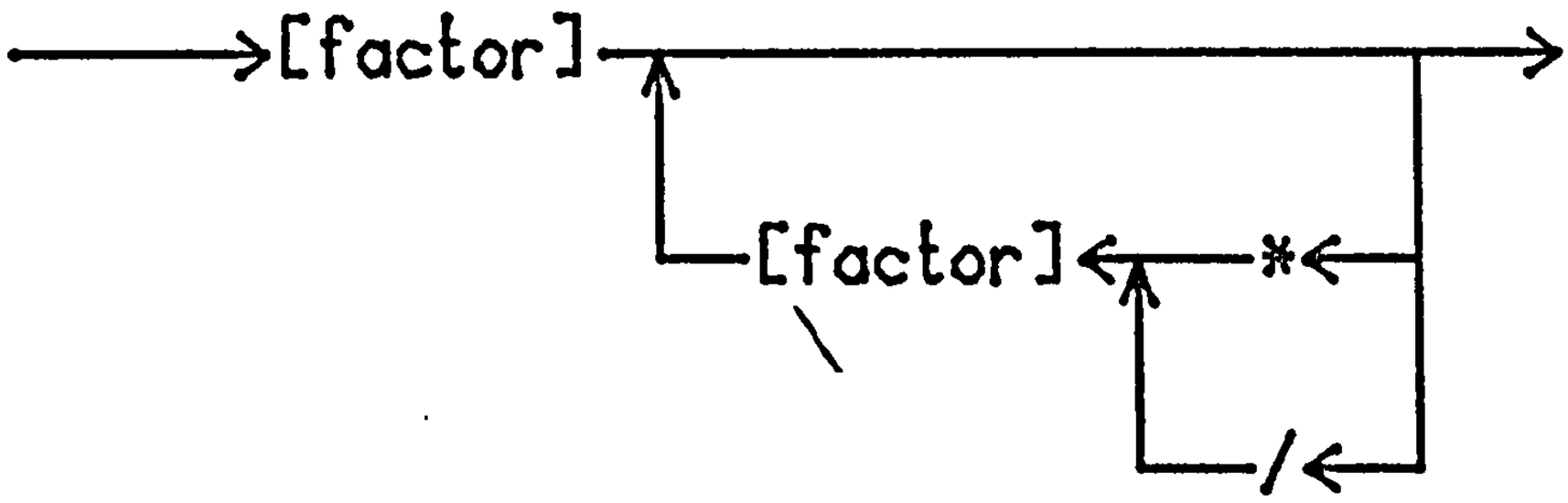

6
ident
:=
7
CALL
ident
BEGIN
[statement]
END
;
[statement]
IF
[condition]
THEN
[statement]
WHILE
[condition]
DO
[statement]


ODD
[expression]
[expression]
=
#
<
>
<=
>=
[expression]

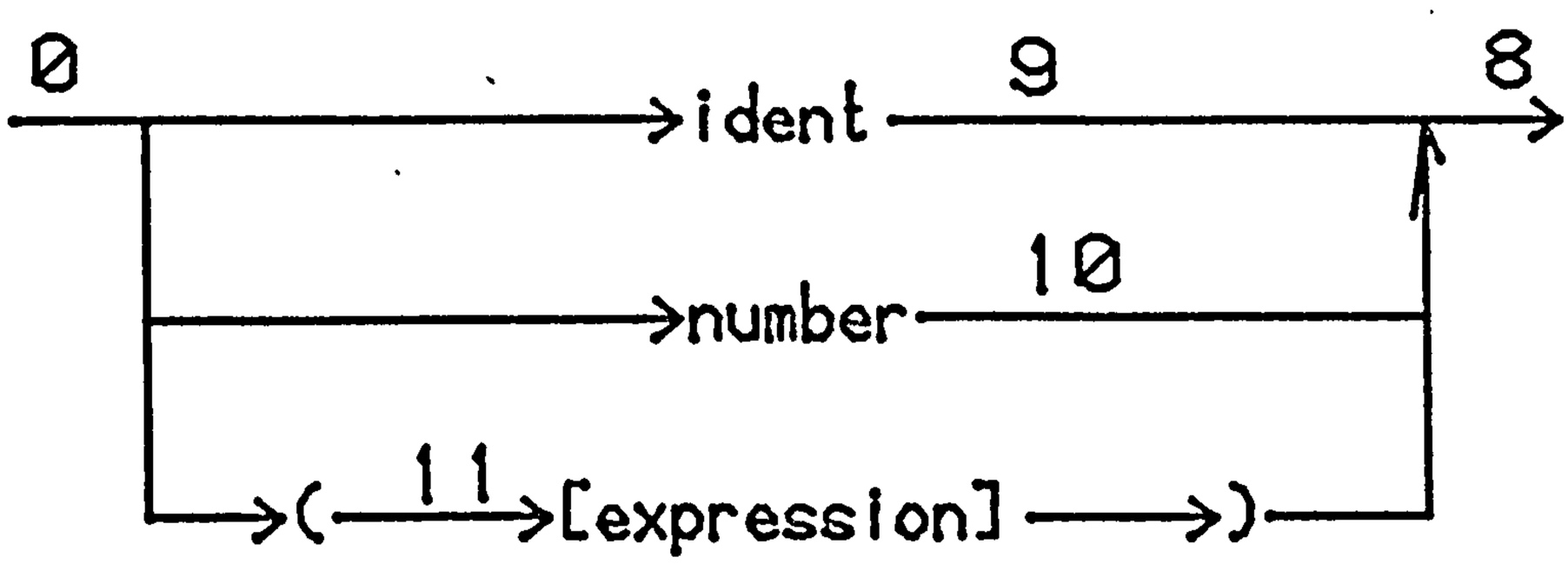
95



94



93



PASS 3 : checking variables

As in pass 1, in matrix 98, each time on entering [block], we MARK the stack and on leaving, we FLUSH.

For each variable identifier encountered we use SCOPE to check if it is already on the stack. If found on the stack, there is an error, "variable identifier redeclared", but if it is not found, we push that identifier on the stack with "1" above it, to mark it as of interest.

Report an error if any of the variable identifiers on the stack appears

in "CONST" part in matrix 98,

as a procedure name in 98, or

after "CALL" in 97.

Report an error if "ident" before ":=" in 97 is not on the stack, i.e. it is not declared and marked with "1". All other identifers will have been marked with "0" as in the first pass.

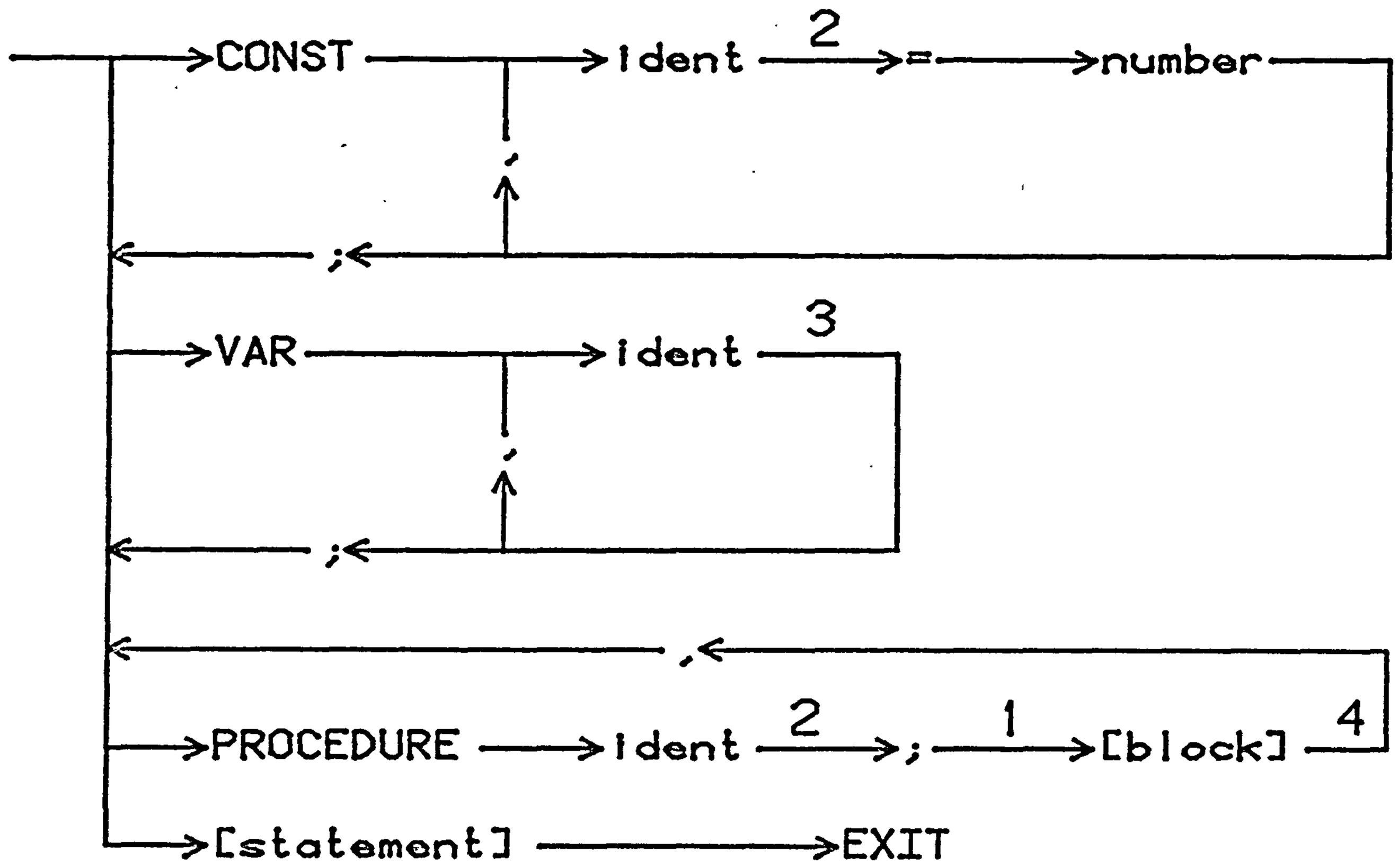
Semantic actions: Variable Identifiers

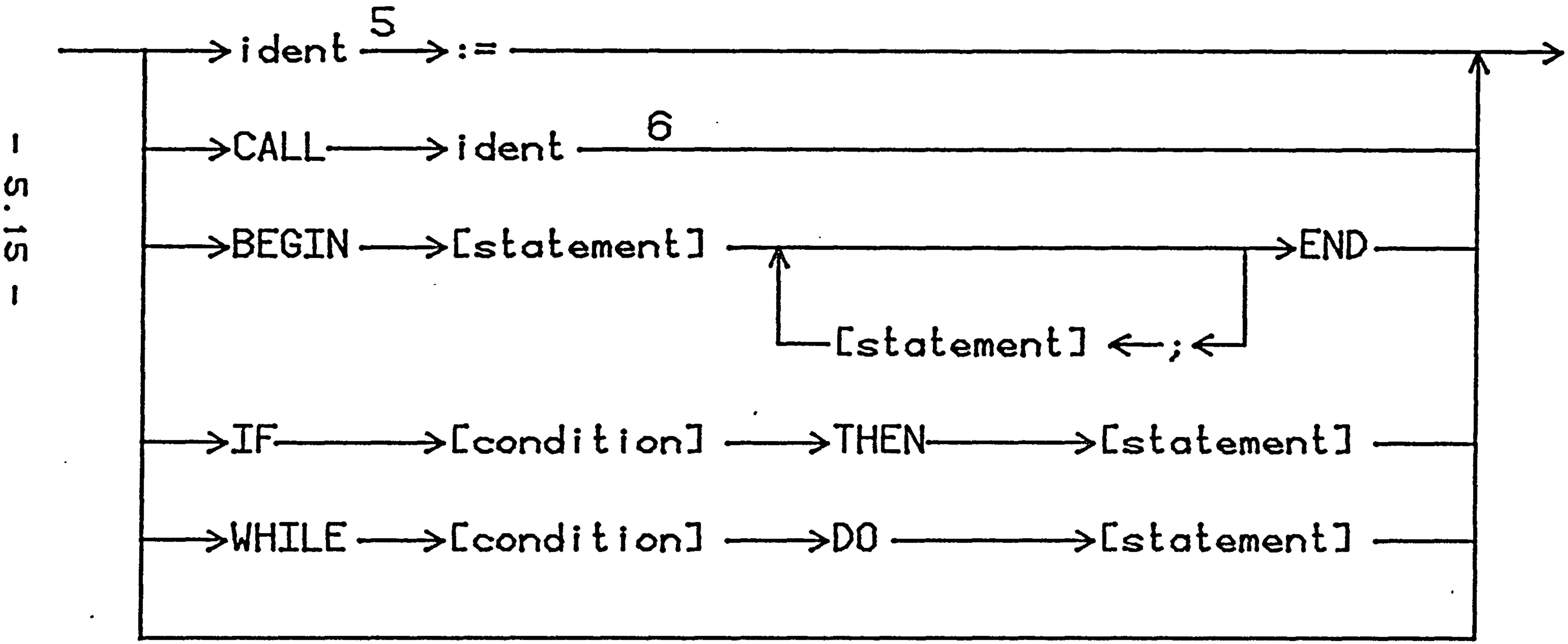
```
1. mark

2. scope css(error(ln,": ",css.," declared",nl); push css push 0;)

3. scope css(error(ln,": ",css.," declared",nl); push css push 1;)

4. flush

5. search css(;;)
   check %index(
     check %index+1(; error(ln,": ",css.,"cannot appear in LHS"););
     error(ln,": ",css.," undeclared");
               )

6. search(;;)
   check %index(
     check %index+1(error(ln,": ",css.," not a procedure");;);
     error(ln,": ",css.," undeclared");
               )

7. search css(; error(ln,": ",css.," is not defined",nl);)
```

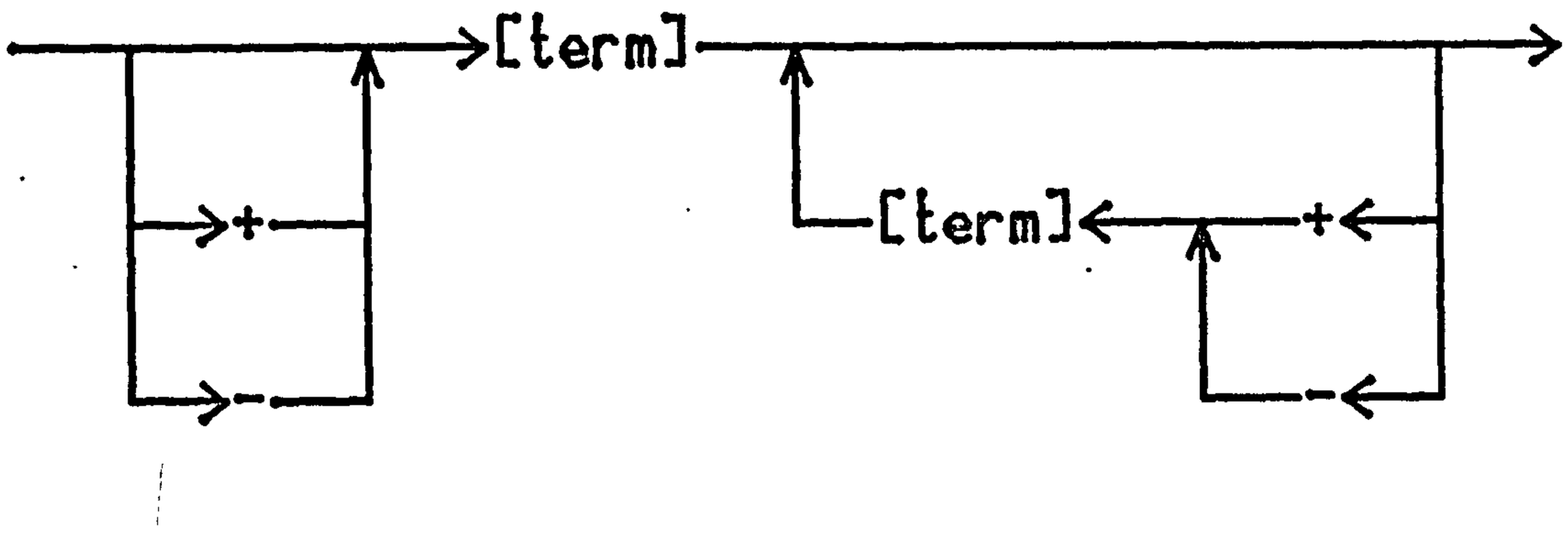
99

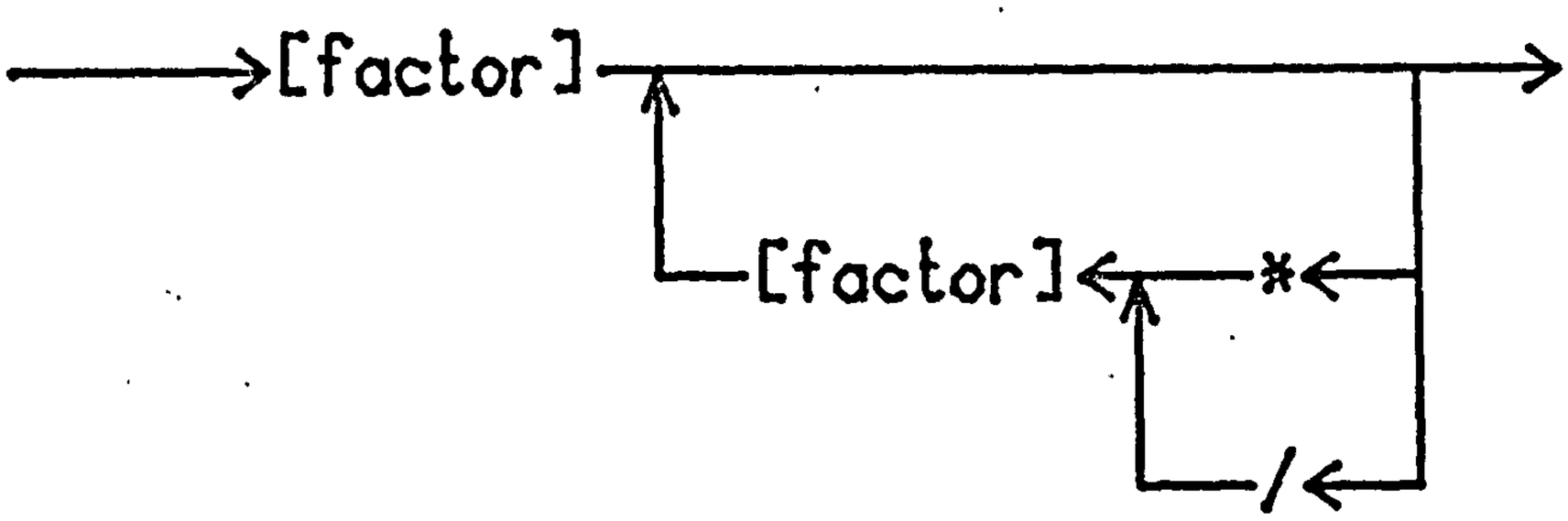

1
[block]
.

98

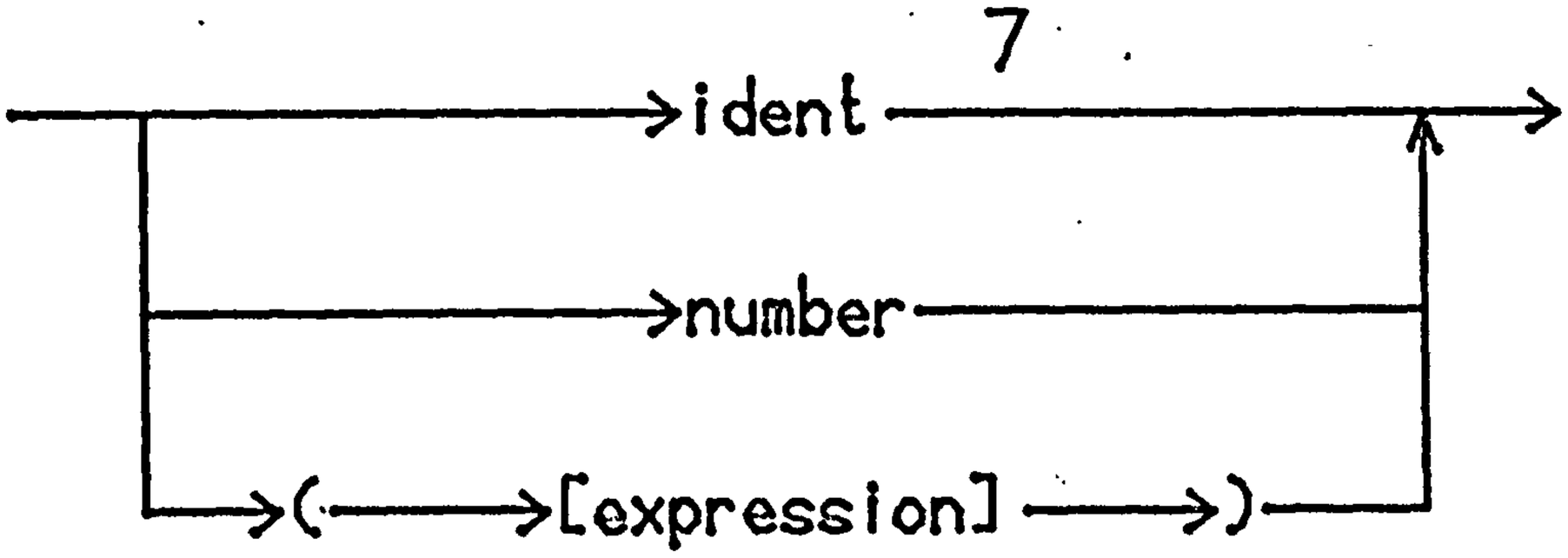

CONST
Ident
2
=
number
,
;
VAR
Ident
3
,
;
,
PROCEDURE
Ident
2
;
1
[block]
4
[statement]
EXIT


ident
5
:=
CALL
ident
6
BEGIN
[statement]
END
;
[statement]
IF
[condition]
THEN
[statement]
WHILE
[condition]
DO
[statement]

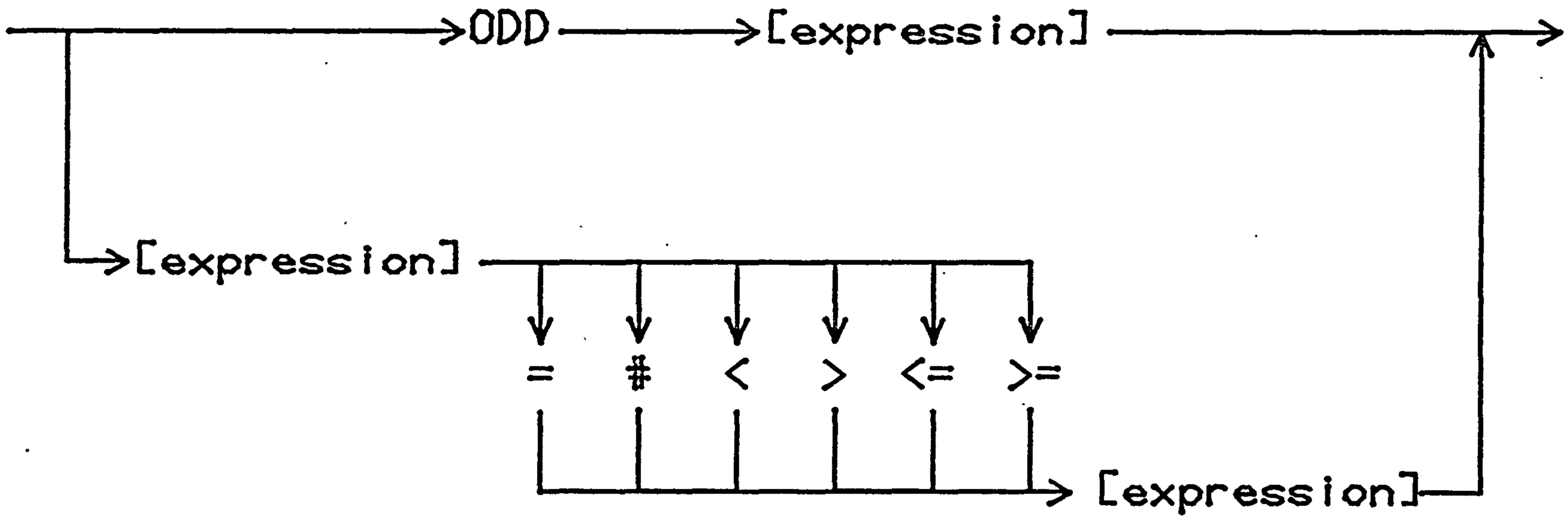




7



- S.17 -



PASS 4 : checking procedures

As in previous passes, "mark" and "flush" the stack before entering and upon the exit each time a [block] is encountered in matrix 98.

Push procedure identifier on the stack if it is not already there, and if it is, report an error that "procedure identifier is redeclared". We mark other identifiers as of no interest.

Report an error if any of the identifiers of interest on the stack appears

in "CONST" part in matrix 98,

in "VAR" part in 98, or

before ":=" in 97.

Report an error if the ident after CALL in 97 is not on the stack, i.e. is not declared or is declared but marked as not of interest.

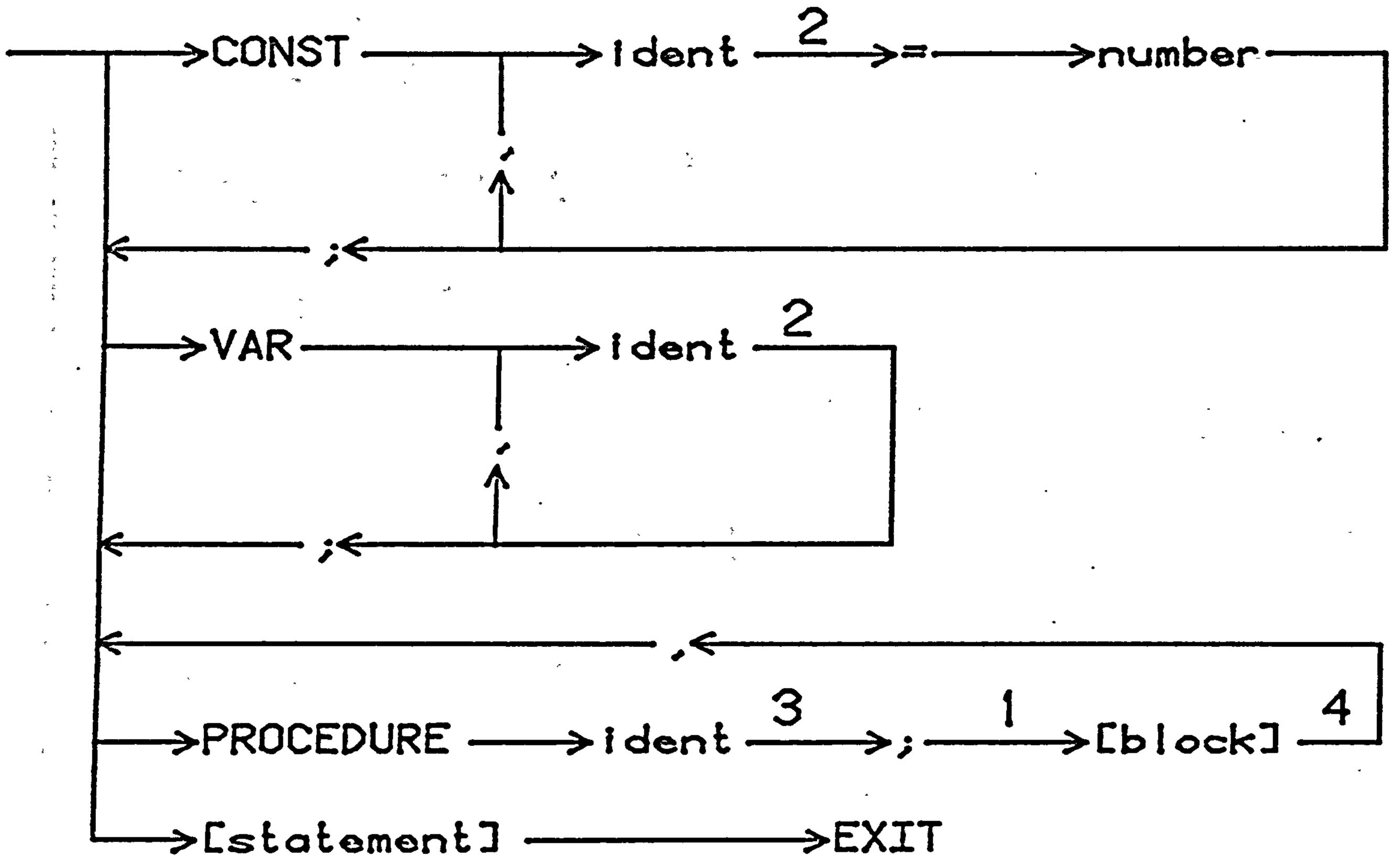
Semantic actions: Procedure Identifiers

```
1. mark

2. scope css(error(ln,": ",css.," declared",nl); push css push 0;)

3. scope css(error(ln,": ",css.," declared",nl); push css push 1;)

4. flush

5. search css(;;)
   check %index(
      check %index+1(error(ln,": ",css.," is a procedure name");;);;)

6. search(;;)
   check %index(
     check %index+1(;error(ln,": ",css.," is not a procedure"););
     error(ln,": ",css.," undeclared");
               )
```

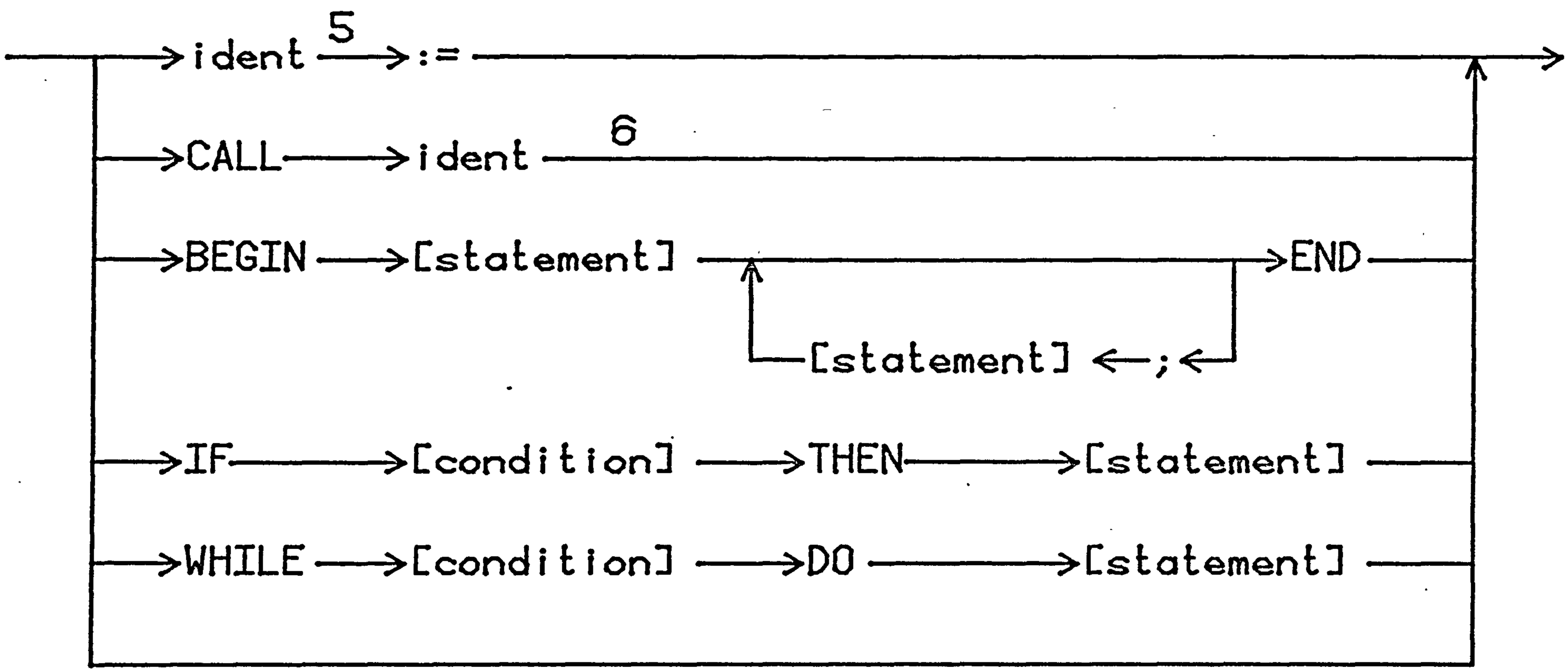
99


1
[block]
.

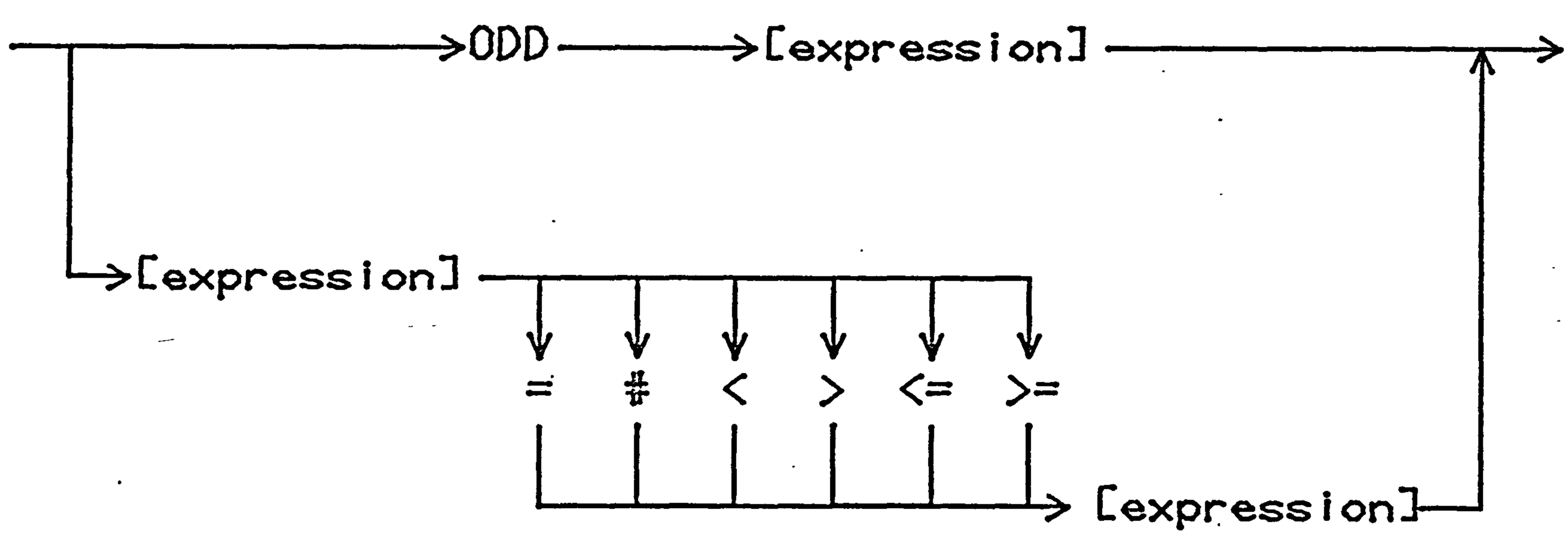
98


CONST
Ident
2
=
number
,
;
VAR
Ident
2
,
;
,
PROCEDURE
Ident
3
;
1
[block]
4
[statement]
EXIT

97


ident
5
:=
CALL
ident
6
BEGIN
[statement]
END
;
[statement]
IF
[condition]
THEN
[statement]
WHILE
[condition]
DO
[statement]

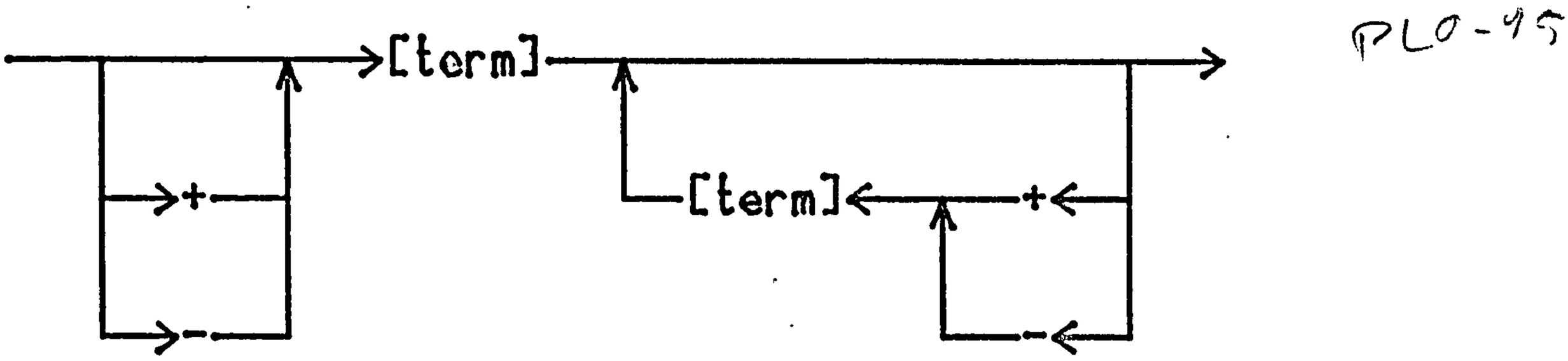
96



Matrix 96

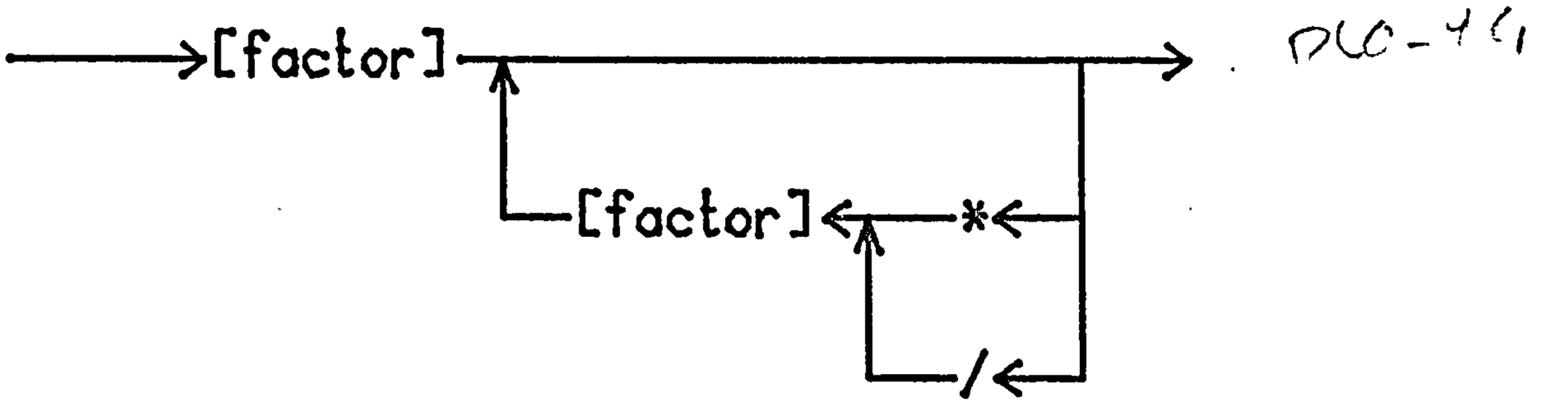
Matrices 95, 94 & 93

95



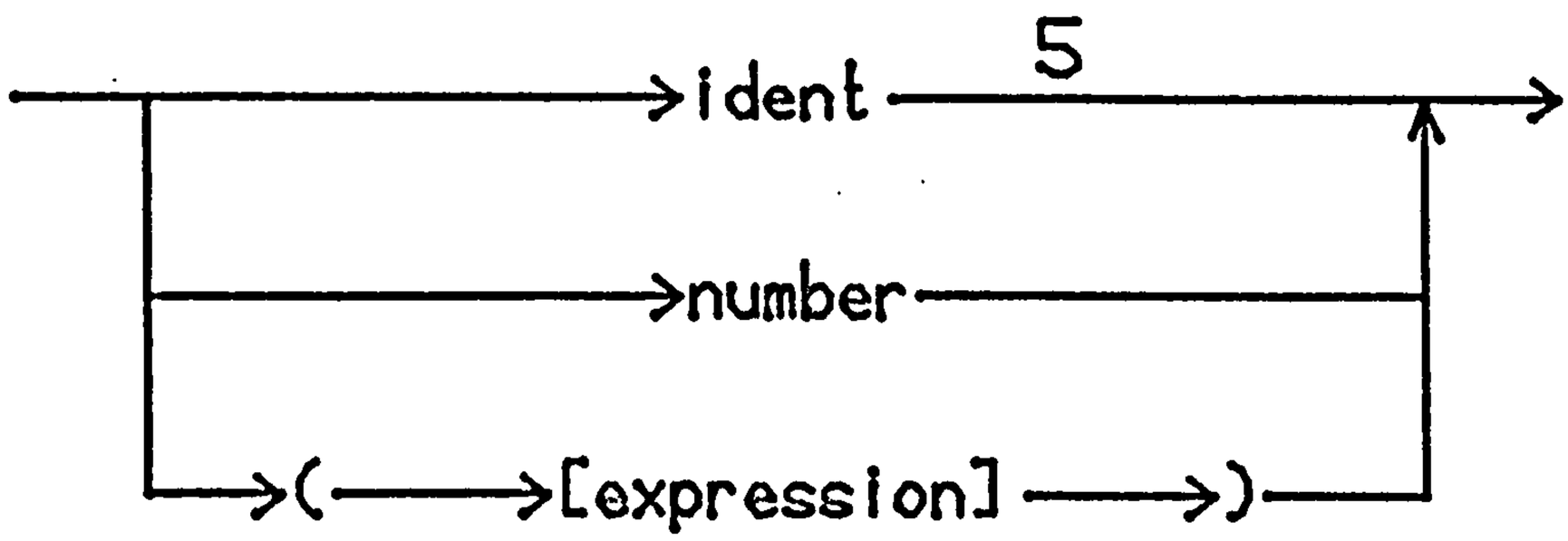
PLO-95

94



PLO-94

93



PLO-93

ATTRIBUTE PROPAGATION IN EXPRESSIONS

As outlined in the introduction to this chapter, PL/0 does not require that attributes be propagated within the Abstract Parse Tree. We therefore discuss this problem for a language with similarities to ANSI FORTRAN.

We assume that an earlier pass of KHAR has emitted reverse polish, or post-fix, code and has appended type tokens to the identifiers in expressions. As in [CORDY], we assume that expressions are bracketed in some convenient way.

The attributes of the leaves (identifiers) have to be propagated up the tree so that semantic checking can take place. This is done by repeatedly scanning the linearized expression from left to right, dealing with one (operand, operand, operator) triple at a time. As language surveys have shown [TANENBAUMb], the majority of expressions are very simple, so this labourious approach will only occasionally result in heavy overheads. We present a simple example and then give the corresponding semantic graphs.

Consider the expression

```
- W + X * Y + Z.
```

We assume that it has been translated into an internal form which might be externally represented as

```
+++ W REAL -- X REAL Y REAL * + Z REAL + ---.
```

Note the use of -- to represent unary minus and the start-of-

expression, end-of-expression markers, +++ and ---.

We give the semantic graphs and actions for the two passes below, while explaining the action of the passes here.

The first pass transforms our example to

+++ < -- W REAL > X REAL Y REAL * + Z REAL + ---.

The action of the semantic graph is to stack identifiers with their types until an operator is encountered but emitting operands and their type further than two deep as it does so. If the operator is monadic it is emitted followed by its operand, else it is followed by two operands. In either case this prefix fragment is bracketed by "<" and ">". Then the rest of the expression is copied until "---" is reached and emitted.

The action of the second pass is to match the bracketed prefix operator and its operand(s) with a syntax graph and semantic (code emitting) actions. The syntax graph defines the valid combinations of operator and types. The actual identifiers can be ignored. The second graph gives the checking needed for a language which requires explicit type changing. The effect of this on our example is to produce

+++ W REAL X REAL Y REAL * + Z REAL + ---.

Repeating the two scans gives,

first,

+++ W REAL < * Y REAL X REAL > + Z REAL + ---

and, then,

+++ W REAL TEMP REAL + Z REAL + ---.

A third pair of scans gives

+++ < + W REAL TEMP REAL > Z REAL + ---

and

+++ TEMP REAL Z REAL + ---,

which, in turn, gives,

+++ < + Z REAL TEMP REAL > ---

and

+++ TEMP REAL ---,

which becomes on the second scan

TEMP REAL

and is no longer an expression, since it is not bracketed as such.

Note that we rely on the syntactic processing to detect and report errors. We can also place code emission actions in the error action part of the matrix to repair the error to allow further processing if we wish.

We tabulate the actions for the two passes and then present the graphs.

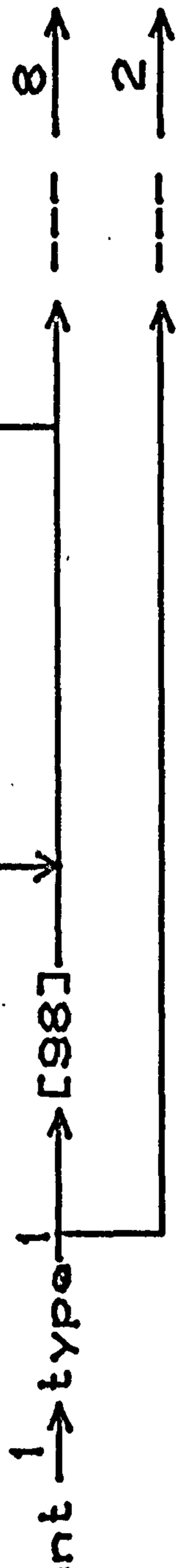
Semantic Actions : first pass of attribute propagation

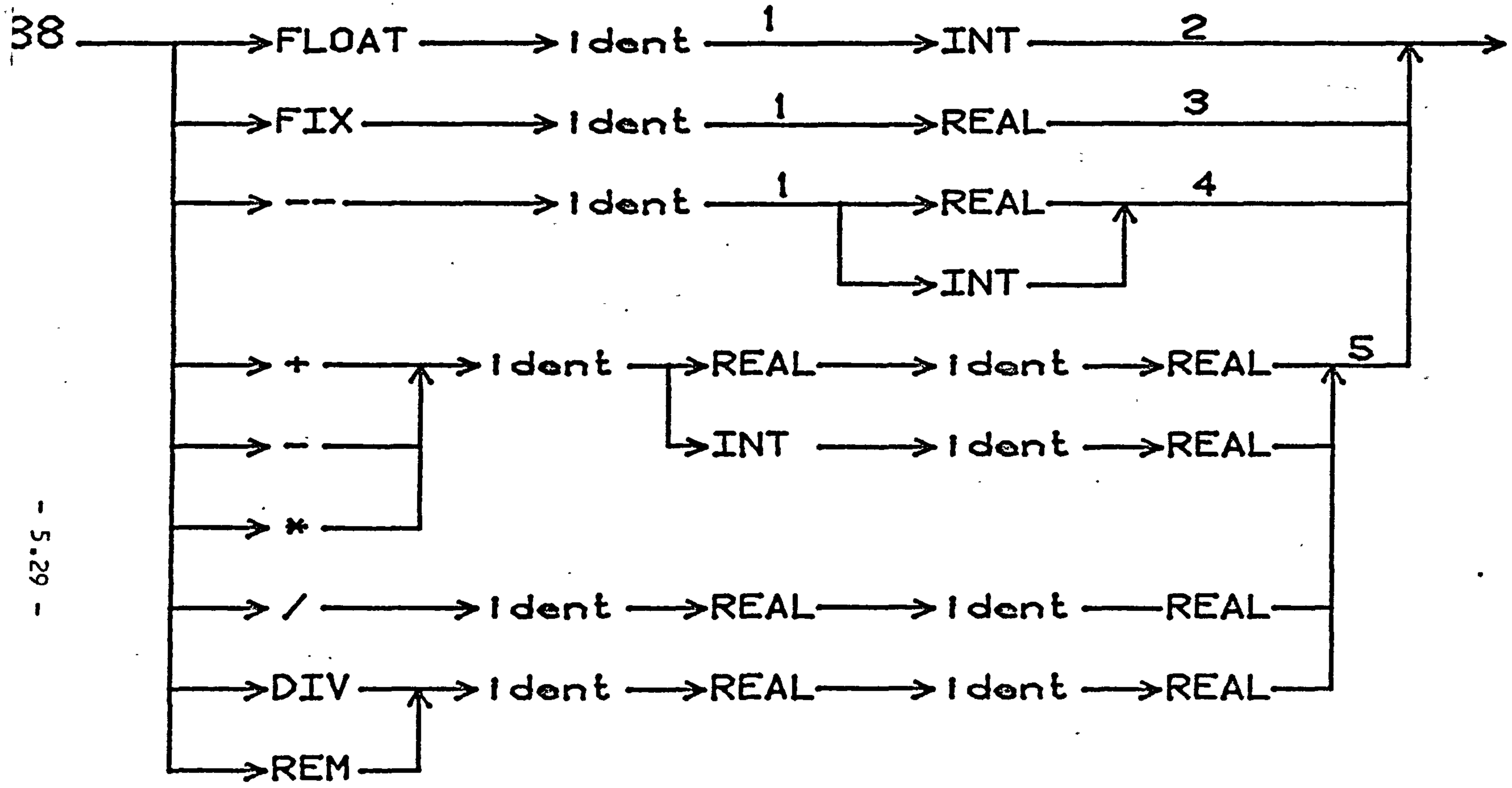
1. push css
2. emit(%2, %1) pop
3. emit(%3)
4. emit("<", css, %2, %1, ">")
5. emit(%4, %3, "<", css, %2, %1, ">")
6. emit("<", css, %4, %3, %2, %1, ">")
7. emit(%3, %4) push css
8. emit(css)

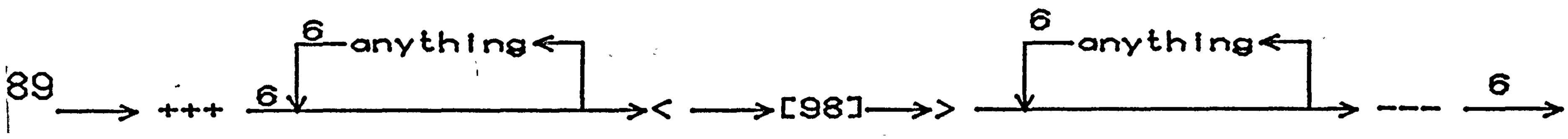
Semantic Actions : second pass of attribute propagation

1. push css
2. emit(%1,"real")
3. emit(%1,"int")
4. emit(%1,css)
5. emit("temp",css)
6. emit(css)

First Pass : Matrices 89 & 88




38
FLOAT
Ident
1
INT
2
FIX
Ident
1
REAL
3
--
Ident
1
REAL
4
INT
+
Ident
REAL
Ident
REAL
5
INT
Ident
REAL
-
*
/
Ident
REAL
Ident
REAL
DIV
Ident
REAL
Ident
REAL
REM


89
+++
6
anything
6
<
[98]
>
6
anything
6

CODE GENERATION

We now discuss code generation in terms of generating code for a simple assembler for the PL/0 machine, that is, we assume a two-pass process which will generate code addresses. A simple assembler could be constructed as two KHAR passes but we take the discussion of assembly code as sufficient for the work at this stage.

The placing of the code emitting actions in the syntax is derived by inspection of the compiler code in [WIRTHb].

The use of the KHAR semantic mechanism to build the information needed for code emission echoes [CORDY] but, again, the separation of the semantics out from the emission simplifies the graphs and the understanding of what is happening.

The method uses the register LABEL to generate labels for use in transfer of control instructions, and LEVEL and RG to construct the address information on the stack required to generate storage references. A triple (CSS,LEVEL,RG) is pushed onto the stack for each identifier encountered in the VAR part. The triple is located using SEARCH when the load or store order is generated while scanning <statement>.

CHAPTER 6 : THE INTERFACE TO KHAR

As was introduced in chapter 3, the interface to KHAR for the language designer is a Syntax Language which he will use to encode a linearized form of the syntax graph and error recovery actions, etc., needed for a particular pass.

We discuss the Syntax Languages in general, then present SL0 in detail, giving the hand coded tables needed to implement SL0, having first discussed the Special Actions which are all that may be used in SL0. We then present the linearized graph for SL1 which is translated using KHAR set up for SL0 to produce the more useable SL1. We then show the syntax of SL1 and coclude the chapter by presenting that of SL4, the current interface to KHAR.

SYNTAX LANGUAGES

In this part we introduce a family of languages SL0, SL1,... designed for creating the transition table and the action table of any other language we may implement.

One of the key ideas in the design of SL series is to make possible the automatic creation of these tables. The transition table and action table of at least one of SL languages must be made manually and we do this for our smallest language in the SL series, SL0. Using SL0 we may define the SL1 tables which may be created

automatically by SL0 and so on. Eventually SL(i) is suitable as the user's interface to KHAR. That is to say that SL(i) language can be improved to produce SL(i+1), to deal with the features of a new language to be implemented.

A program in any SL is the linearized form of syntax information, semantic information and code emitting actions for a language L according to the syntax of SL, a linearized form of the transition matrices of the language L.

A program in SL has one or more blocks bounded in curly brackets "{" and "}" with the following structure :

```
{
    main block
    other blocks
}
```

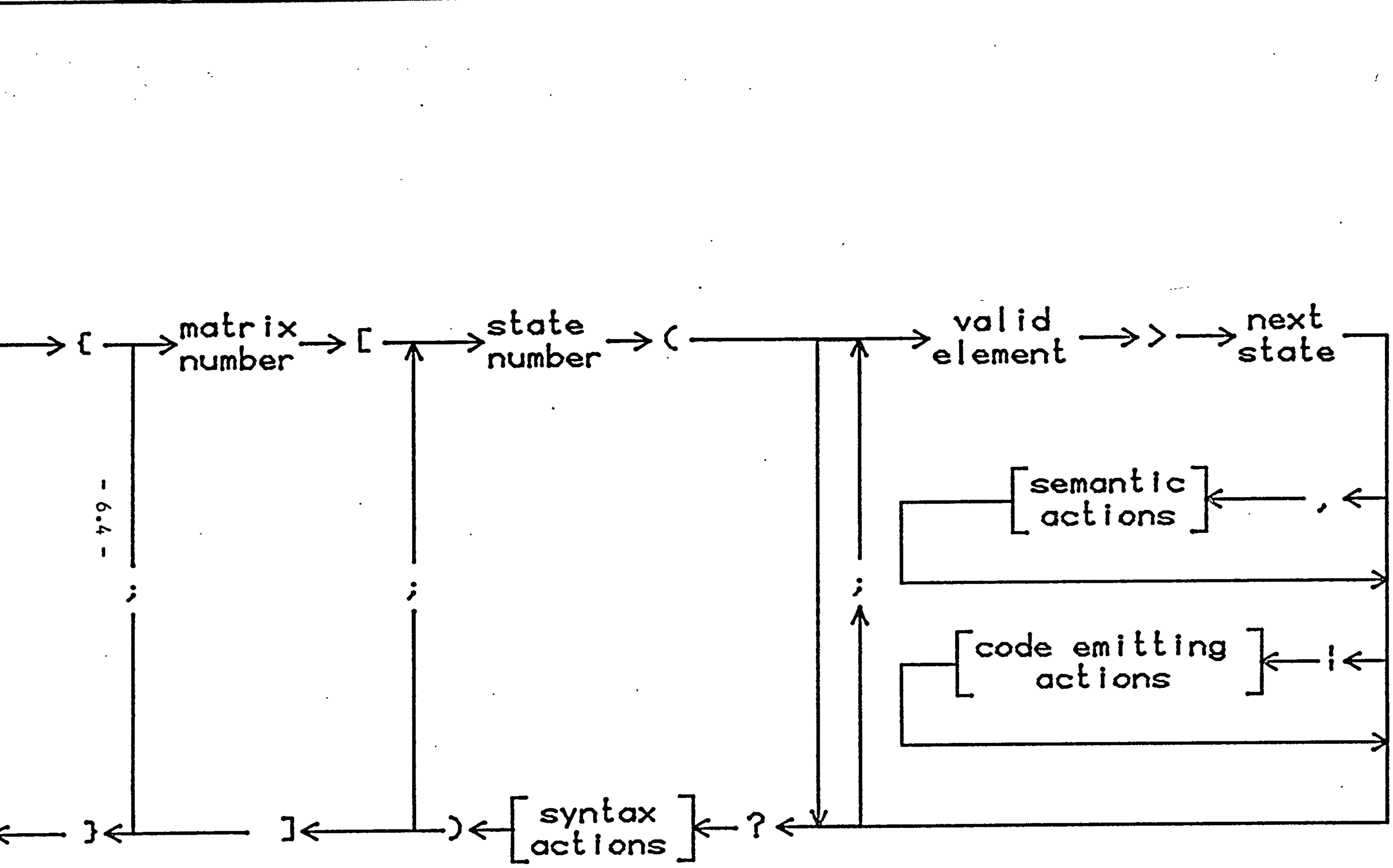
"Main block" is the information constructed from the main transition matrix and "other blocks" contain the information from the other matrices, each of which has a similar layout to the main block. Each block has one or more compound statements bounded by square brackets "[" and "]". Each compound statement has one or more simple statements and one syntax-error action part, bounded by parenthesis "(" and ")". Blocks and compound statements are all labeled. The labels are numeric.

THE GENERAL SYNTAX OF SL LANGUAGES

```
<program>          ::={<main block>[;<block>...  }
<main block>       ::=<block>
<block>            ::=<matrix label>[<state>[;<state>...  ]
<state>            ::=<state label>([<statement>[;<statement>]...]
                      ? <syntax error actions> )
<statement>        ::=<valid element> > <state label>
                      [,<semantic actions>][ | <code emitting actions> ]
<matrix label>     ::="integers between 99 and 50 "
<state label>      ::= "integers between 0 to number of states"
<valid element>    ::= "all language symbols"
<syntax error actions> ::= " depends on which SL(i) is being defined."
[semantic actions] ::="as defined in chapter 5"|ERROR|EMIT|
[code emitting actions] ::=[special actions]|[semantic actions]
[special actions] ::=SA1|SA2|.....|SA11|SA12
```

Different SL languages vary only in the three nonterminals <syntax error actions>, <semantic actions>, and <code emitting actions> used in the overall syntax of the SL series. For SL0 these three nonterminals are as follows.

```
[error syntax action] ::= STOP
[semantic action] ::= empty
[code emitting actions] ::=[special actions]
```


{
matrix number
[
state number
(
valid element
>
next state
semantic actions
,
code emitting actions
!
;
}
]
)
syntax actions
?

LANGUAGE SYMBOLS OF SL

The present language symbols of SL are:

4

tab nl ln sp rg css set pop push emit error label

sa0 sa1 sa2 sa3 sa4 sa5 sa6 sa7 sa8 sa9 sa10 sa11 sa12

use find go or stop get unget loop exit

check search flush index level mark scope

5

! " # $ % & ' ( ) =

^ - ~ | ` [_ @ {

+ *] ; : } < > ? , . /

SEPERATION OF SL KEYWORDS & LANGUAGE KEYWORDS

In our small language we have a set of keywords used in its syntax. As mentioned earlier a program in SL is the linearized form of a language syntax and it contains all keywords of that language. So the system to be able to compile an SL program should know both sets of keywords. These two sets of keywords are unchanged (except when we add a feature to the language SL in which case we update the set). The other set is the keywords of the language for which we intend to set up our system. We concatanate these two and call it "keywords". Sl keywords always come first. This ensures that the codes for SL keywords are always the same.

USE OF POINTERS IN TRANSITION TABLE

In our transition table each valid element is followed by three pointers:

1) pointer to the same table to indicate the next state;

2) pointer to semantic actions;

3) pointer to code-emitting actions.

If any of the last two pointers is zero, it means there will be no such actions for that element.

However if the valid element was a matrix number the case is a litle different as follows

1) the same as above

2) pointer to either type of action before entering the matrix

3) pointer to either type of action after entering the matrix.

SPECIAL ACTIONS SA0 TO SA12

These are the only ones available in SL0. They may, of course, be used in all other SLs and in any programming language if necessary.

SA0

This is the null action, used to satisfy syntax of SL0.

SA1

Changes the state numbers to their actual addresses of the beginning of the appropriate state in transition table. This is done at the end of each block in an SL language at the time of code

emitting.

SA2

This is normally called after each state number is read. The action is to remember starting point of the current state in transition table.

SA3

This puts the current symbol on the first avalable item of the array transition table.

SA4

This changes the sign of the "next state symbol" and puts it in the transition table array, so that at the end of current matrix they are recognized (as they are negative) and changed to their actual address by action SA1.

SA5

This action remembers the beginning of each matrix and initializes all elements of "state addresses" array into a negative number.

SA6

This action comes at the end of the last block in an SL program, counts the number of matrices, records matrix codes with the address of the beginning of each of them in the transition matrix, records the length of transition matrix, writes transition table on the appropriate file, and writes the action table on its file.

SA7

Puts the current symbol on the action table.

SA8

Puts the "next state" on the action table.

SA9

Puts a zero on transition table.

SA10

Puts a zero on action table.

SA11

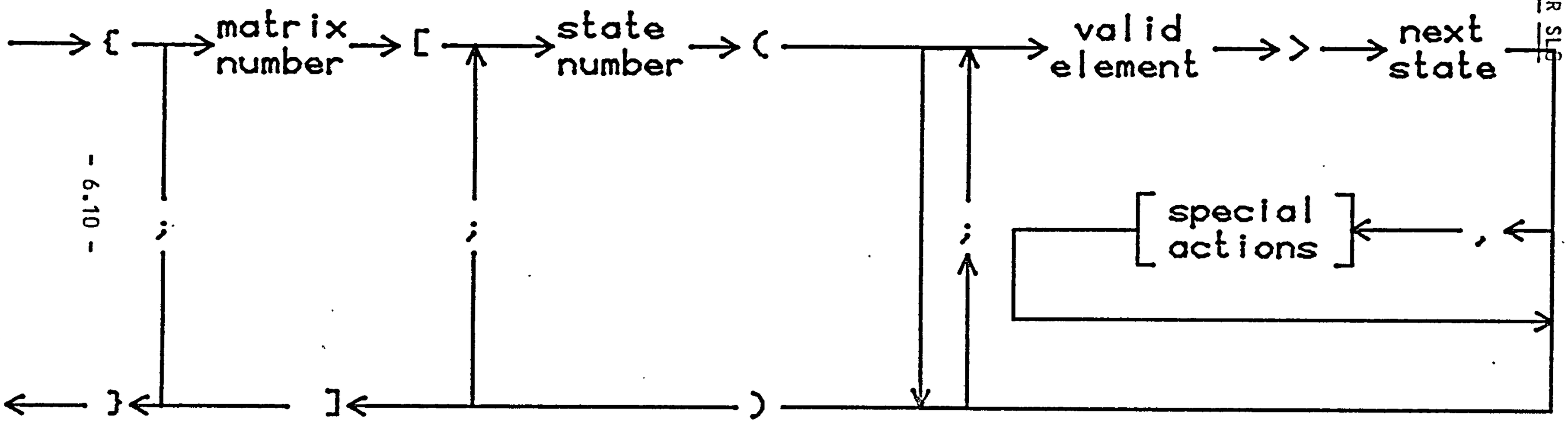
Puts the current pointer of action table on transition table.

SA12

Put "rw-stop" on the transiton table. This is the only syntax error action in SLO.

Syntax error action

In this part for SLO we have no syntax error recovery. Any error causes the compilation to stop at that point.

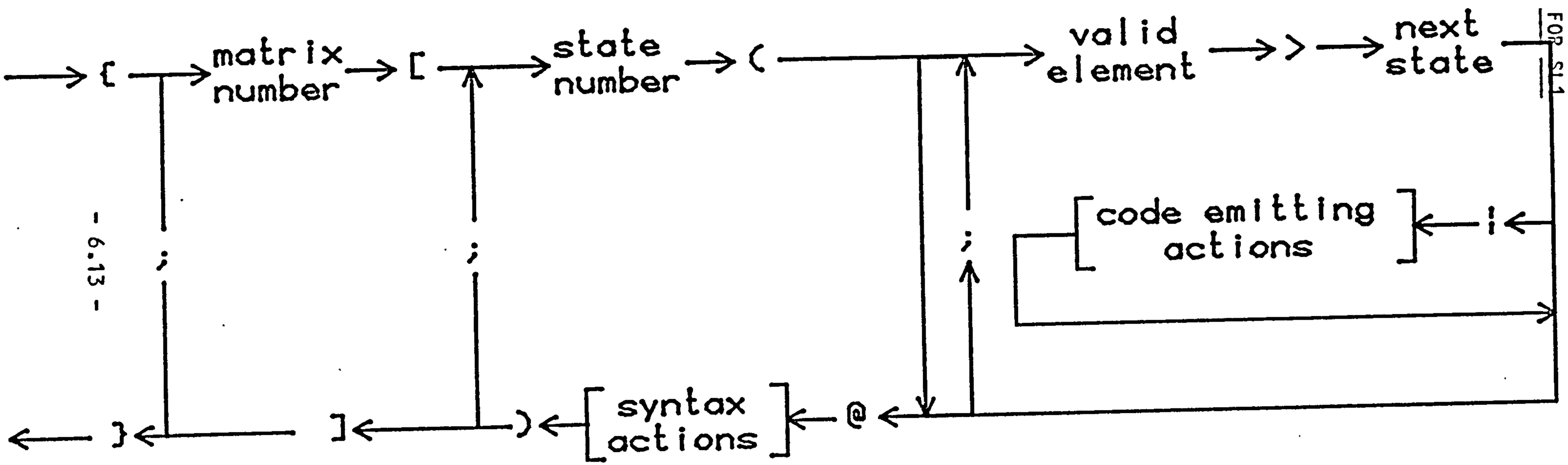

{
matrix number
[
state number
(
valid element
>
next state
SYNTAX GRAPH FOR SLC
special actions
,
;
}
]
)

TRANSITION TABLE FOR SLO

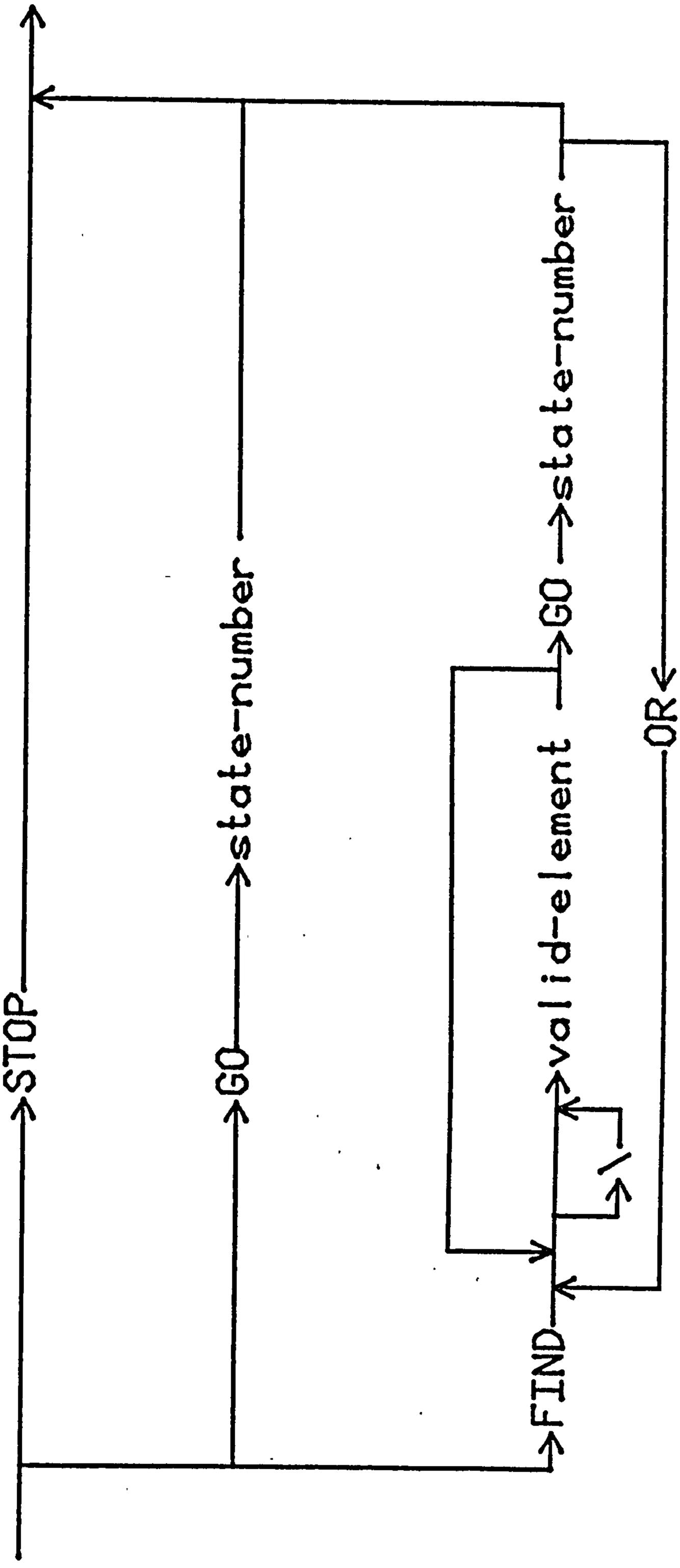
1		36	>	76	)
99		37	42	77	82
0		38	0	78	0
102		39	0	79	22
0	{	40	0	80	0
1	6	41	1	81	1
2	0	42	22	82	;
3	34	43	48	83	18
4	0	44	0	84	0
5	1	45	10	85	0
6	21	46	0	86	]
7	12	47	1	87	92
8	0	48	|	88	0
9	3	49	62	89	0
10	0	50	0	90	0
11	1	51	12	91	1
12	[	52	;	92	;
13	18	53	30	93	6
14	0	54	0	94	0
15	0	55	15	95	0
16	0	56	)	96	}
17	1	57	82	97	998
18	22	58	0	98	0
19	24	59	29	99	26
20	0	60	0	100	0
21	6	61	1	101	1
22	0	62	27	102	
23	1	63	68		
24	(	64	0		
25	30	65	18		
26	0	66	0		
27	0	67	1		
28	0	68	27		
29	1	69	68		
30	23	70	0		
31	36	71	18		
32	0	72	;		
33	8	73	30		
34	0	74	0		
35	1	75	20		

ACTION TABLE FOR SL/O

	37
0	0
1	stop
2	0
3	sa1
4	sa2
5	0
6	sa2
7	0
8	sa3
9	0
10	sa4
11	0
12	sa9
13	sa11
14	0
15	sa9
16	sa9
17	0
18	sa7
19	0
20	sa10
21	0
22	sa9
23	sa0
24	sa10
25	0
26	sa1
27	sa6
28	0
29	sa9
30	sa9
31	sa9
32	sa0
33	0
34	sa12
35	sa10
36	0
37	

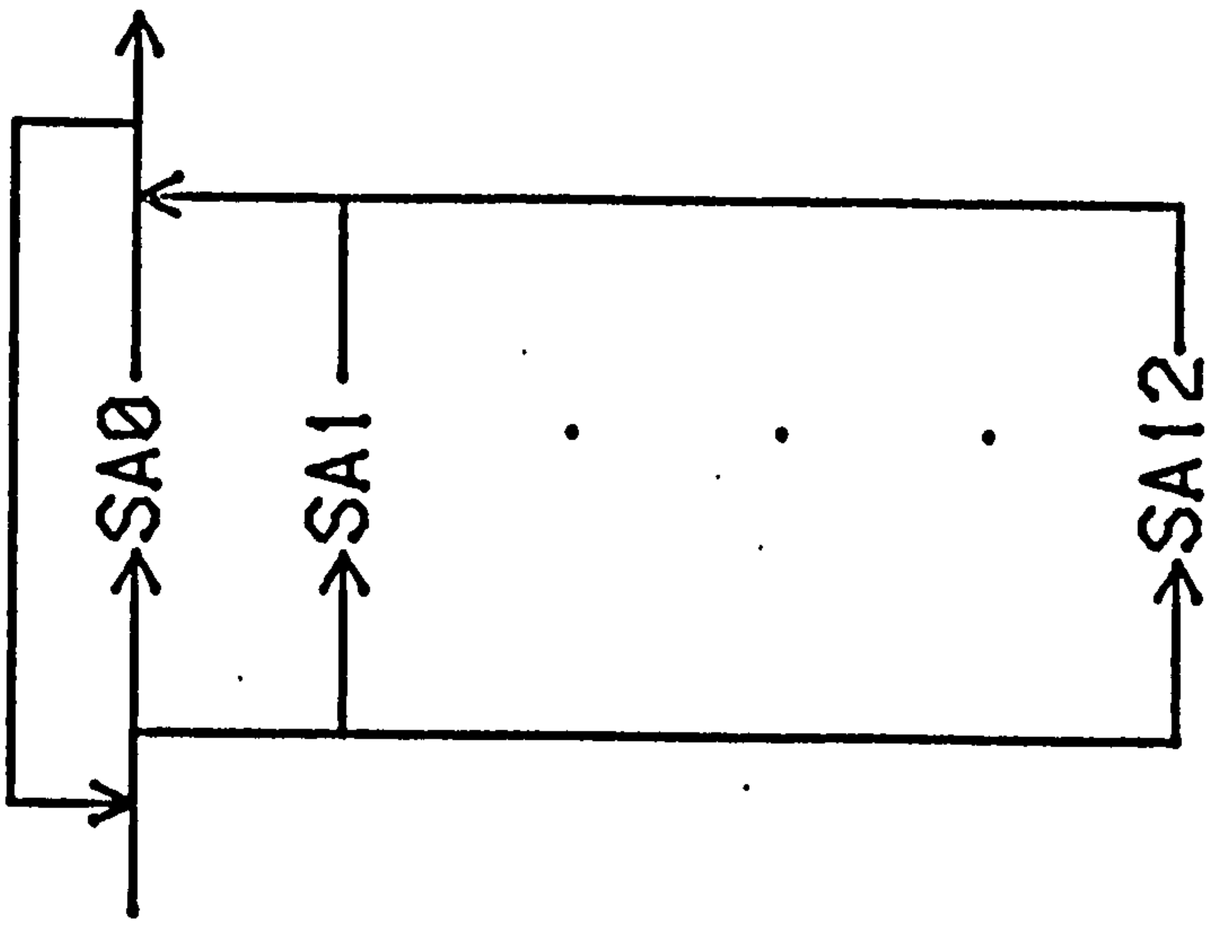

{
matrix number
[
state number
(
valid element
>
next state
code emitting actions
!
;
}
]
)
syntax actions
@

Matrix 98



Matrix 97

SA-97



CODING OF SL1 IN SLO

The following is a complete program written in SLO, for our present small language, SL1.

```
{
  99  [  0({>1 );
         1(21>2|sa1 sa2 );
         2([>3 );
         3(22>4|sa2 );
         4((>5 );
         5(@>12|sa9 sa11; 23>6|sa3 );
         6(>>7 );
         7(22>8|sa4 );
         8(|>9|sa9 sa11; ;>11|sa9 sa9; @>12|sa9 sa9 sa9 sa11);
         9(97>10|sa10 );
        10(;>11; @>12|sa9 sa11);
        11(23>6|sa3 );
        12(98>13|sa10 );
        13()>14 );
        14(;>3; ]>15 );
        15()>exit|sa1 sa6; ;>1 )
      ];
  98  [  0(exit>exit|sa7;stop>exit|sa7; go>1|sa7; find>2|sa7 );
         1(22>exit|sa8 );
         2(>3|sa7; 25>3 );
         3( 23>4|sa7 );
         4(go>5|sa7; >3|sa7; 23>4|sa7 );
         5(22>6|sa8 );
         6(or>2|sa7; 25>exit )
      ];
  97  [  0(sa0>0|sa7; sa1>0|sa7; sa2>0|sa7; sa3>0|sa7; sa4>0|sa7;
         sa5>0|sa7; sa6>0|sa7; sa7>0|sa7; sa8>0|sa7; sa9>0|sa7;
         sa10>0|sa7; sa11>0|sa7; sa12>0|sa7; 25>exit )
      ]
}
```

THE END-STATE SYMBOL

We choose a special character which is rarely used in programming languages and assign it as "end-state" in the SL languages. In this implementation we use "?" as our end-state symbol.

If it happened that "?" was one of the valid elements of a language in use, we may change that to another character, if we like. The ambiguity arises only if we have "?" as the first valid element of a state. For example in the following SL statement

```
5(end>3; ?>1 ?stop)
```

the first "?" is recognized as a valid element because after ";" we expect valid element but the second one is end-state because it is not preceded by a ";". But in the following SL statement

```
5(?>4; end>5 ?stop)
```

after "(" we can have either valid element or end-state followed by syntax-error-actions. So having read the first "?" there is an ambiguity if this is end-state (that is if this state is an error-state) or a valid element. Using "\" before "?" takes its special meaning, (end of state symbol). So if "?" is to be used as the first valid element of a state it should be preceded by a "\".

DESCRIPTION OF THE STATEMENTS

All statements in an SL program have the same layout. Being in a particular state we expect the current source symbol to match with one of the valid elements of that state and the "next state" after that valid element is the address of another statement where we can find the next expected symbol.

The error action part at the end of each statement is for error recovery if none of the valid elements matched the current source symbol.

A valid element in statement can be

a reserved word

an integer

an identifier

a constant string

a matrix label

a statement label

any special action

null-symbol

any-symbol

A "next state" in statements is an existing statement label.

ACTIONS

The actions are listed here but the majority of them have already been explained elsewhere. The chapter is indicated.

USE

Described in chapter 4.

SUCC

Described in chapter 4.

STOP

The whole process would stop, see chapter 4.

LOOP n

See chapter 4: "n" is any state number in the same transition matrix.

GO n

See chapter 4; "n" is any state number in the same transition matrix.

POP, PUSH & SET

These three actions are semantic actions described in chapter 5.

EMIT and ERROR

These two actions have the same arguments. The difference is that "EMIT" outputs on "output-file" and "ERROR" outputs on "error-file". They can have any number of arguments seperated by commas, in parantheses. Arguments and meanings are as follows :

CSS

outputs the current source symbol

RG

outputs the content of register

"constant string"

outputs the pointer to "constant string"

%n

outputs the n'th. element down the stack.

%INDEX

outputs the element in the stack accessed by the current value of INDEX.

%INDEX+n

%INDEX-n

outputs the element +/- n from that accessed by INDEX.

If the argument is followed by a full-stop (".), then the actual symbol would be output.

For example

EMIT CSS

outputs the internal code of current source symbol but

EMIT CSS.

outputs the actual symbol.

NL

outputs a new line

SP

outputs a space

LN

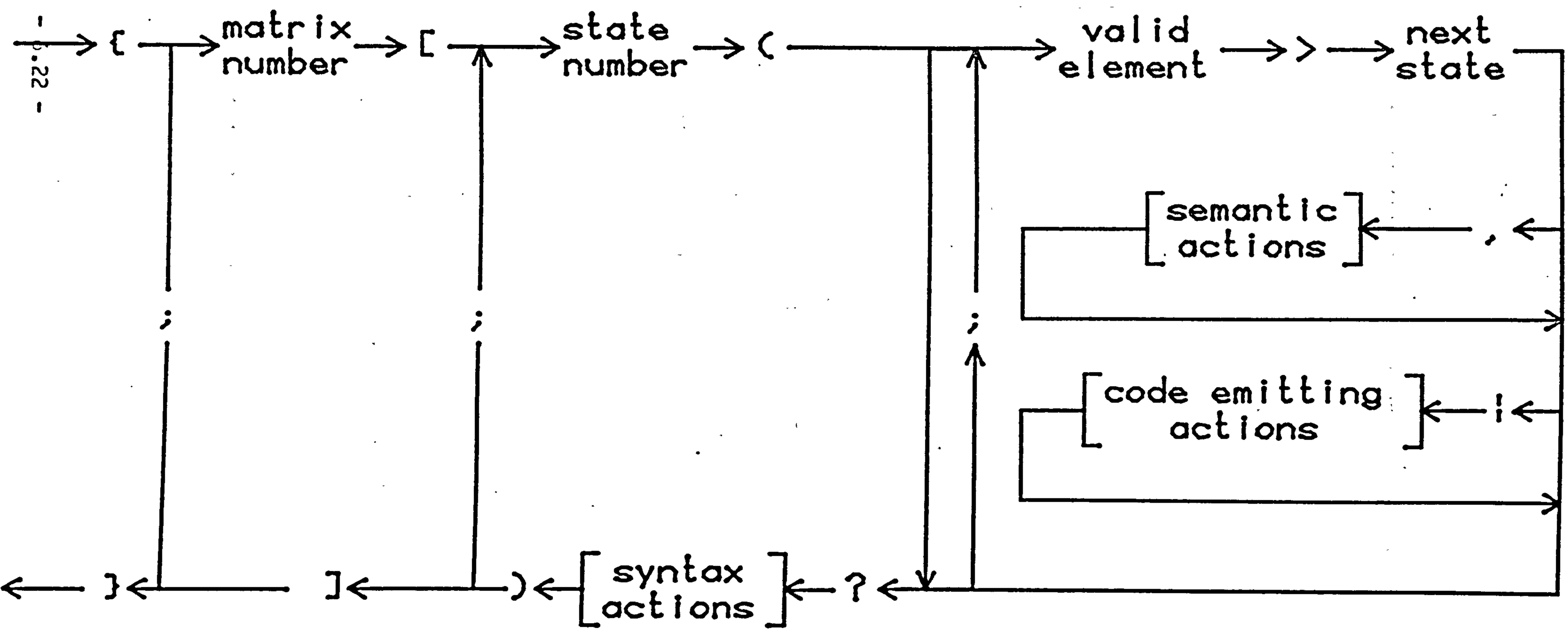
outputs the line number

TAB

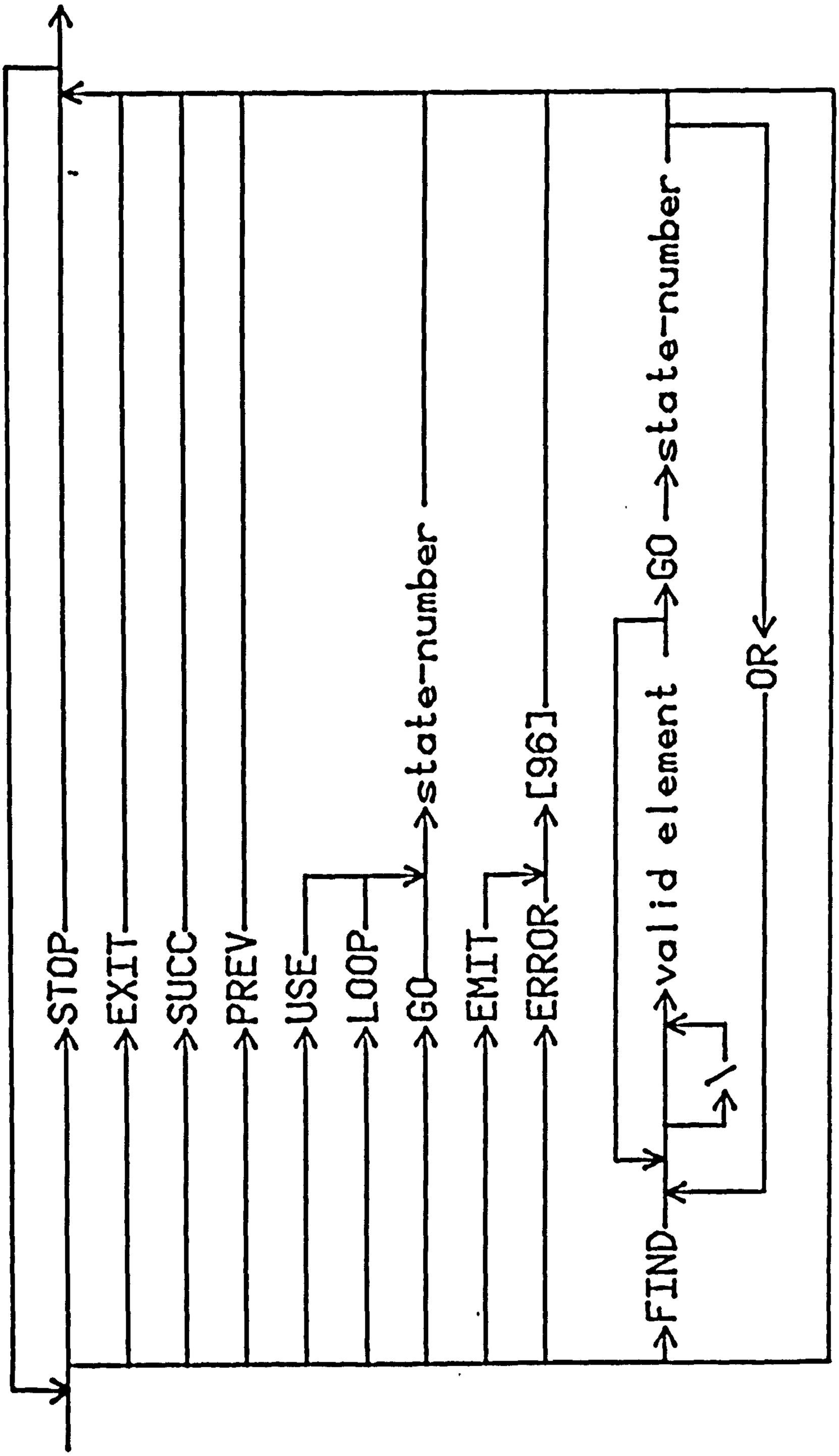
outputs "tab"

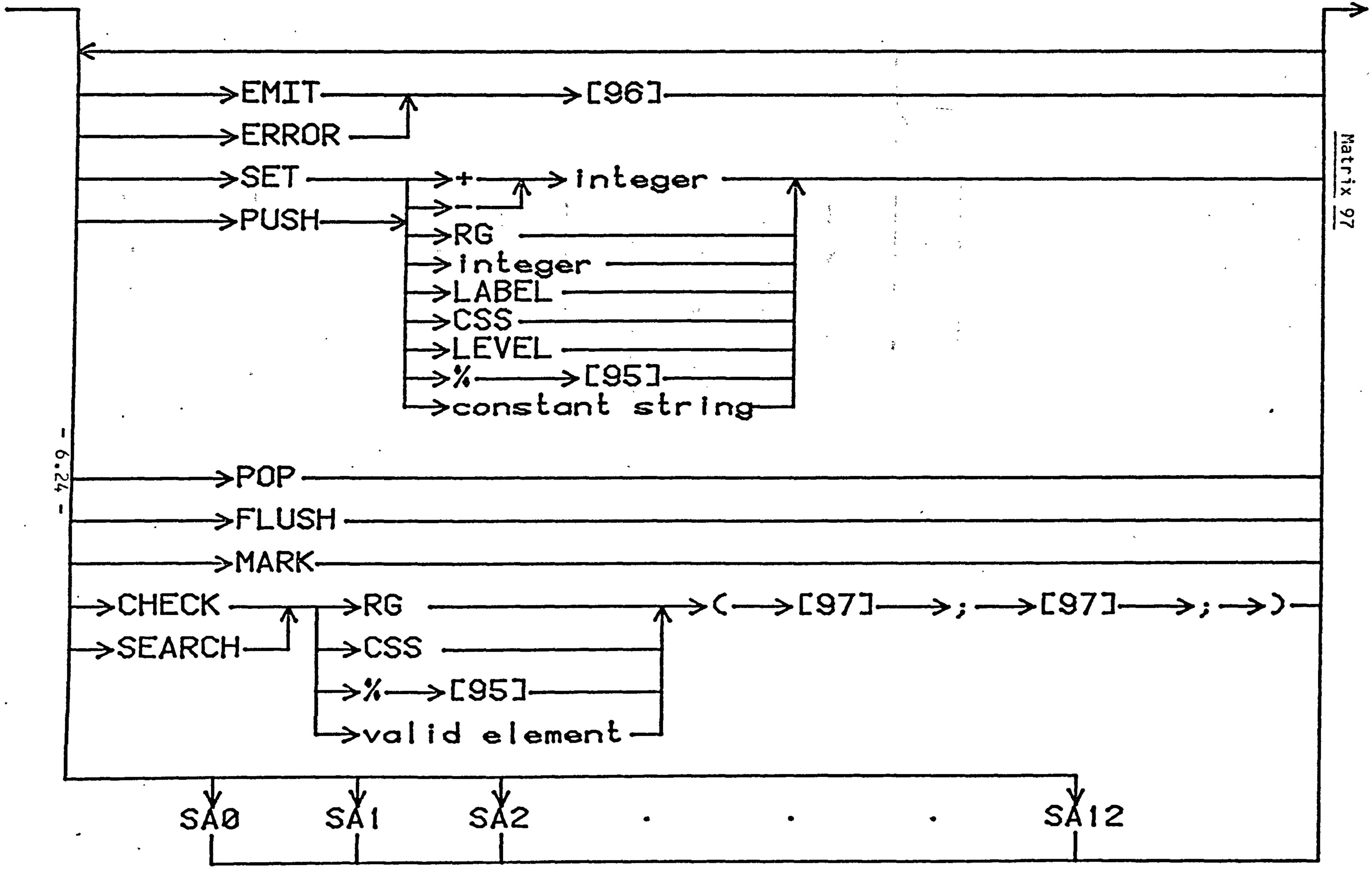
SYNTAX OF SL4

SL4 is the current interface to KHAR. We present its syntax as a set of graphs and give its encoding in SL1.

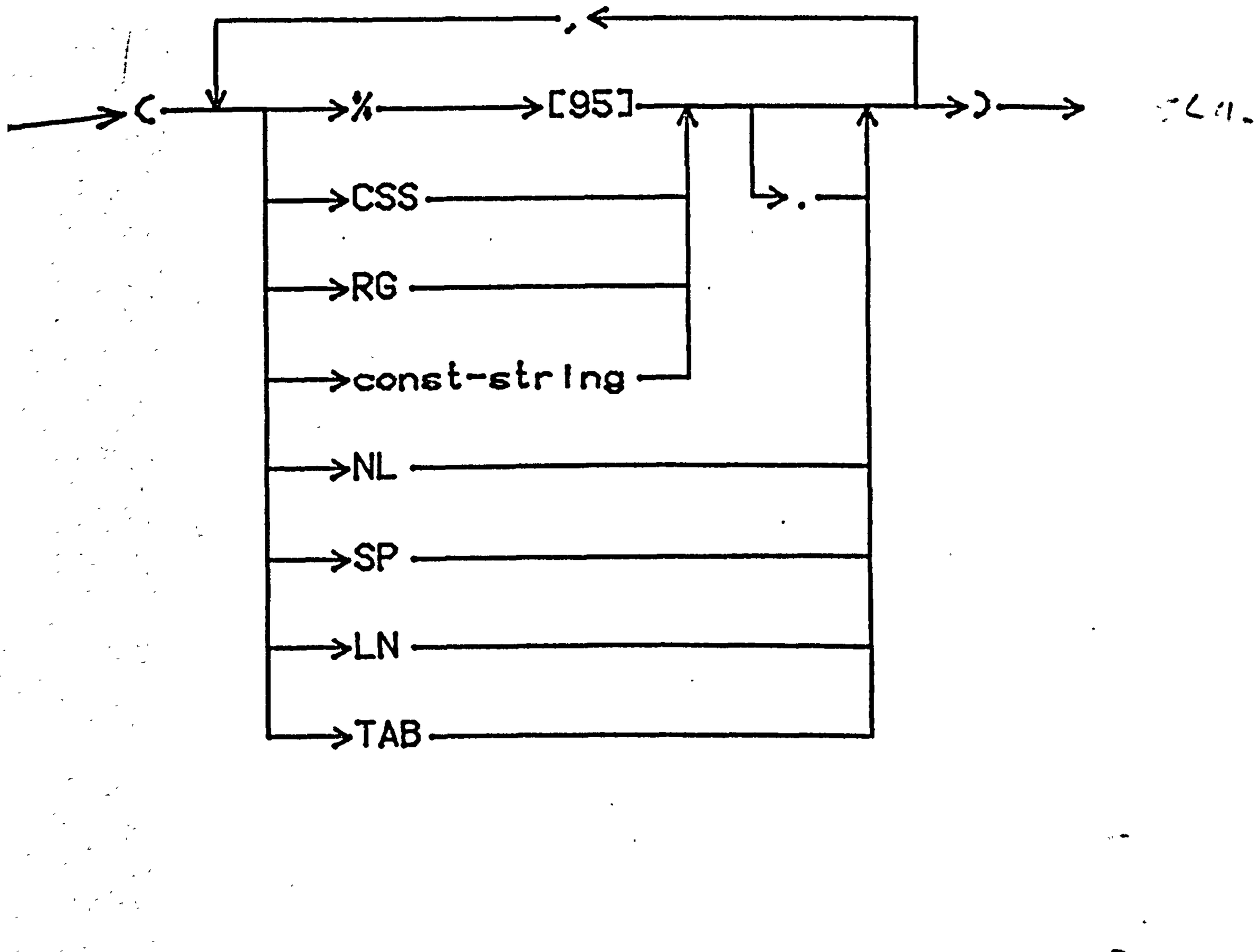

{
matrix number
[
state number
(
valid element
>
next state
semantic actions
,
code emitting actions
!
;
;
;
}
]
)
syntax actions
?

Matrix 98

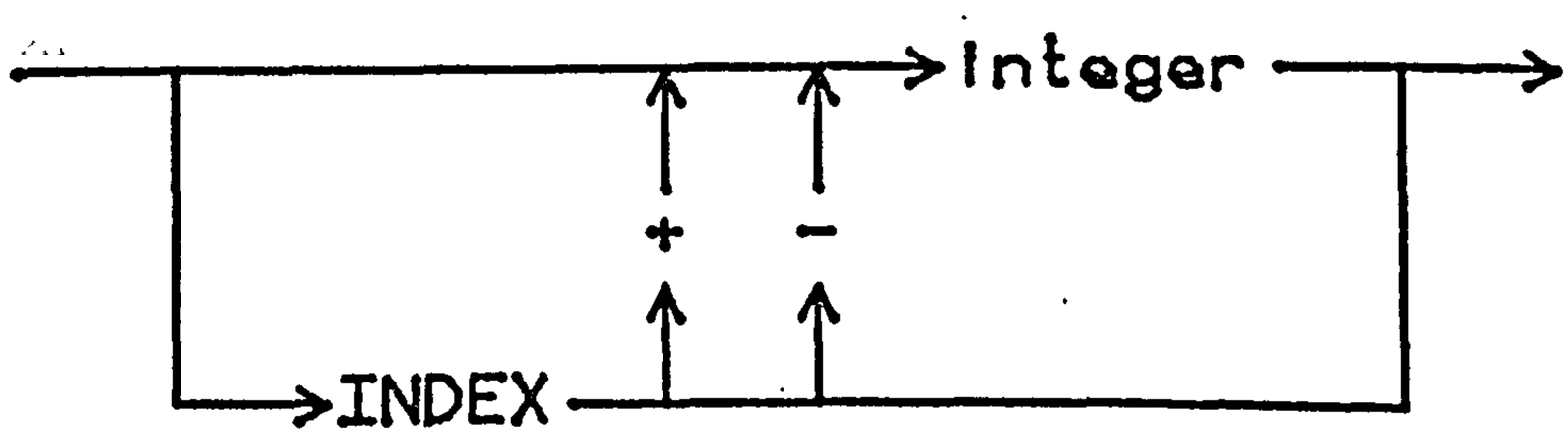

STOP
EXIT
SUCC
PREV
USE
LOOP
GO
state-number
EMIT
ERROR
[96]
valid element
GO
state-number
OR
FIND


EMIT
[96]
ERROR
SET
+
integer
-
PUSH
RG
integer
LABEL
CSS
LEVEL
%
[95]
constant string
Matrix 97
- 6.24 -
POP
FLUSH
MARK
CHECK
SEARCH
RG
CSS
%
[95]
valid element
(
[97]
;
[97]
;
)
SA0
SA1
SA2
SA12

Matrices 96 & 95



SL4-95



CODING OF SL4 IN SL1

```
{
  99  [  0({>1  @ go 18 );
         1(21>2|sa1 sa2  @ go 18 );
         2([>3  @ go 18 );
         3(22>4|sa2  @ go 18 );
         4((>5  @ go 18 );
         5(?>17; 23>6|sa3 @ go 18 );
         6(>>7  @ go 18 );
         7(22>8|sa4  @ go 18 );
         8(,>9|sa11; |>11|sa9 sa11; ;>13|sa9 sa9;
           ?>17|sa9 sa9 sa9 sa11  @ go 18 );
         9(97>10|sa10  @ go 18 );
        10(|>11|sa11; ;>13|sa9; ?>17|sa9 sa9 sa11  @ go 18 );
        11(97>12|sa10  @ go 18 );
        12(;>13; ?>17|sa9 sa11  @ go 18 );
        13(23>6|sa3  @ go 18 );
        14()>15  @ go 18 );
        15(;>3; ]>16  @ go 18 );
        16(}>exit|sa1 sa6; ;>1  @ go 18 );
        17(98>14|sa10  @ go 18 );
        18( @ find  );22( go 4 or 23>22 go 7 or )] go 16 )
     ];
  98 [  0(stop>exit|sa7; get>0|sa7; unget>0|sa7; loop>1|sa7;
           go>1|sa7; use>1|sa7; find>2|sa7 @ exit );
        1(22>exit|sa8 @ exit );
        2(23>3|sa7 @ exit );
        3(>2; go>4|sa7 @ exit );
        4(22>5|sa8 @ exit );
        5(or>2|sa7; 25>exit @ exit )
     ];
  97 [  0(emit>1|sa7;error>1|sa7;push>2|sa7;set>2|sa7;pop>0|sa7;
           27>0|sa7; flush>0|sa7; search>4|sa7; check>4|sa7;
           25>exit                @ exit );
        1(96>0                    @ exit );
        2(%>3|sa7; 26>0|sa7       @ exit);
        3(26>0|sa7                @ exit);
        4( %>5|sa7; 26>6|sa7      @ exit );
        5( 26>6|sa7     @ exit);
        6( (>7          @ exit);
        7( 97>8         @ exit);
        8( ;>9|sa7      @ exit);
        9( 97>10        @ exit);
       10( ;>11|sa7     @ exit);
       11( )>0          @ exit)
     ];
  96 [   0( (>1|sa7      @ exit);
         1( %>2|sa7;css>5|sa7;rg>5|sa7;20>5|sa7;26>6|sa7 @ exit);
         2( index>3|sa7; 26>5|sa7 @ exit);
         3(+>4|sa7; ->4|sa7; 25>5  @ exit );
         4( 26>5|sa7 @ exit );
         5(.>6|sa7; 25>6 @ exit);
         6(,>1|sa7; )>exit|sa7 @ exit)
     ]
}
```

CHAPTER 7: IMPLEMENTATION

Two versions of the KHAR system were developed. The first was on an ICL1904S in PASCAL and the second on a DEC PDP11/40 written in the C language, a derivative of BCPL. This second version relies on many of the features of the UNIX operating system, in particular, the file system and the ability to write command macros. This enables the large number of files involved in the use of KHAR, see below, to be handled by providing commands for the user.

It must be emphasised that the present version is a prototype designed for use in the research, and to exploit the UNIX environment. The main aim of the implementation is to enable the study of the KHAR passes and the development of the primitives required.

It is therefore different from the final form required for its intended working environment in several ways.

OUTLINE OF PROTOTYPE

First, all the information held in the file store about a program and the language is read from the serial files of the UNIX filestore into a simulated indexed random file organisation. This information would be retained on backing store in a stand-alone system using floppy discs.

Second, all possible KHAR actions, error recovery, semantic, etc., are available in each pass. The discussion concluding chapter 5

showed that only a few passes, principally code generation, needed to access these files except when reporting errors.

Third, the present KHAR machine contains a full set of trace statements, which can be selectively enabled to give a full trace of the behaviour of the machine. This has proved to be of great value in tracing errors.

The implementation has attempted to be as simple-minded as possible. Only one recursive procedure call is made, and no use is made of local variables. Thus the basic code of the machine does not require that a procedural language with recursion be used. This means that little overhead is imposed and the machine is easily transportable.

The main components of the KHAR machine are the recogniser, the algorithm of which has been described, and the action interpreter, which uses the case statement of C to parse the linear action code when an action is called for. Each action is implemented as a separate segment of code. Actions are easily added and can be made available to the user via the SL languages.

Transition Table (TT) and Action Table (AT)

These tables of information are interpreted by KHAR to parse and emit code for the appropriate language. This is an important part of the system and all syntax checking, semantic checking and code emitting revolves about it. Once these tables are made for our smallest language in the SL series by hand, we are able to create them automatically for the succeeding languages. These two tables are arranged as a one dimensional array of elements. In the following section we give the structure of these tables.

Structure of TT

Suppose we have m matrices in the language, then TT consists of m parts and we have pointers to beginning of each part. Each part consists of a number of states, all with the same structure. In the figure on page 7.7 we have enlarged one state, in which a number of "valid element part"s is followed by a "0" to indicate the end of a state and followed by a pointer to a syntax error recovery part in AT. Each valid element part in TT has four elements

1) a language symbol, valid at this point.

2) a pointer to another state of the same table, for use if this element matched the current source symbol.

3) a pointer to AT for semantic actions.

4) a pointer to AT for code emitting actions.

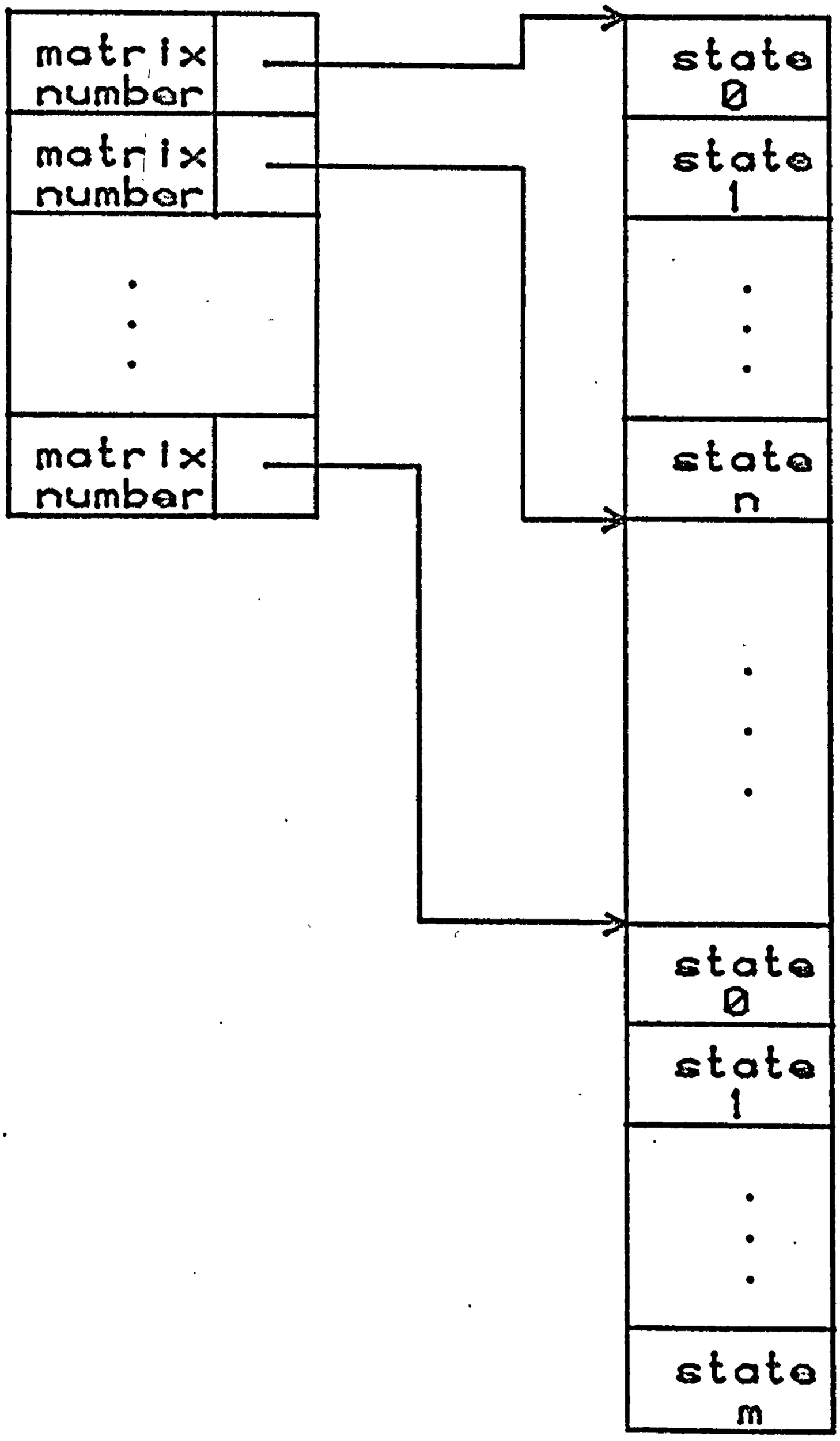
Although the system includes these in the table whether they are

required in a "sub pass" or not the structure could be subdivided if further space reduction were needed.

Sometimes the error recovery part of a few states in a transition matrix is the same. In that case instead of repeating that part at the end of each state we put it in one state at the end of the matrix, calling it the "error state" and refer to it from the other states. "Error state" has no valid element and in TT it consists of two elements, a "0", which can not match anything, and a pointer to AT.

A State of "TT"

"TT"



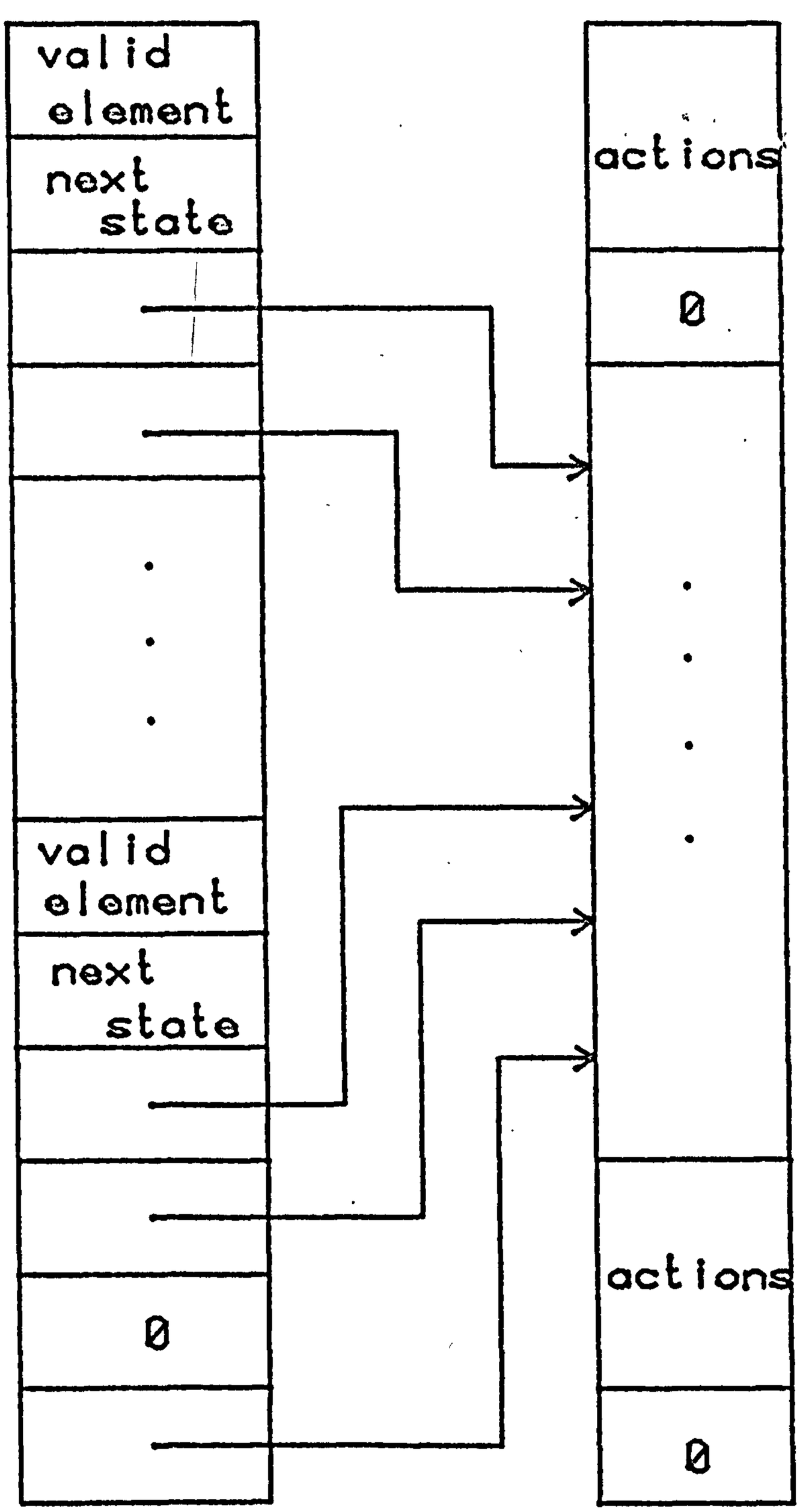
Structure of "AT"

The structure of "AT" is simpler than "TT". It consists of a number of "action parts". Each has some actions followed by a "0" to indicate the end of that action part. The pointers to this table from TT are to the start of these parts. Execution of an action part terminates on encountering the "0". The diagram on the next page shows this.

The Structure of "AT"

one state

of "TT"



FILES & PROGRAMS USED

In this part we explain the programs and the files we use in the system.

A complete listing of the source programs is given in the additional material, LISTINGS OF THE KHAR TRANSLATOR SYSTEM.

FILES used in the system

In this system we use 26 files as follows

1) *codename-file (code name file)
2) *cnindex-file (code name index file)
3) *define-file (define file)
4) *cndefine-file (code name define file)
5) nocomment (all comment removed)
6) const-file (constant file)
7) nostring (all strings encoded)
8) cs-file (constant string file)
9) csindex-file (constant string index file)
10) intgr-file (integer file)
11) noidentifier (all identifiers encoded)
12) id-file (identifier file)
13) notriplechar (all triple characters encoded)
14) nodoublechar (all double characters encoded)
15) code-file (code file)
16) action-rw-file (action reserved word file)
17) *define-rw (define reserved word)

18) *ttable-file (transition table file)

19) *ttdefine-file (transition table define file)

20) *ttaction-file (transition table action file)

21) *ttcsindex-file (transition table constant string index file)

22) *ttcs-file (transition table constant string file)

23) output-file

24) error-file

25) lang-sym (language symbols)

26) any (text)

this file is any input text for which we wish to convert all its basic symbols to their corresponding codes.

We refer to the above files by their numbers, 1 to 26. Only those files whose names are preceded by an '*' are permanent to the system and the others are temporary files.

File 16, action-rw-file, consists of all the reserved words used in syntax-error, semantic and code-emitting actions, and is made by hand. That is, these are reserved words used in our small language, SL.

File 25, is also made by hand and it consists of all the symbols used in the language we wish to implement plus all the symbols used in our small language. Apart from these two files (16 and 25), the others are made by executing programs in the system, which are explained in the next section.

PROGRAMS used in the system.

There are 14 programs used in the system. In this section we explain their tasks, the files they use and the files they create. We refer to these programs as A,Б, ... ,N.

program A

This program reads the "language symbols", file 25, and makes three files

1) codename-file (1)

2) cnindex-file (2)

(these two files are explained in chapter 3)

3) define-file1 (3)

On this third one we write the lengths of the first two files and the base-code. We append more information to this file in the other programs.

program B

In chapter 6 we saw that the terminal symbols of a language can be categorized into different classes. This program reads files 1 and 2 then appends some information to file 3, such as the number of items in each class, the position of that class in file 2 and so on. For those classes of objects which are not present in the language, negative numbers are placed for the number of items in that class. Also this program creates file 4 on which "code-name-table length", "base-code" and "number of code names" are written. This information is needed in program N.

program C

This program using files 1 and 2 finds out the comment delimiters of the language. It then reads source text, deletes all comment and outputs the remaining text on file 5.

programs D and E

These two programs are very similiar. They both read and write the same files. Their main task is to find the constant string delimiters of the language, find constant strings in the input text and replace them by their codes. The actual constant string is put in a dictionary. To distinguish between these codes (which are numbers between 3000 and 3999) and the actual integers used in the text we also have to code integers simultaneously along with coding constant strings. But there are files like "language symbols" or "transition table data" in which the integers used are codes themselves, and we do not wish to code them again. This is why we have two similar programs. Program D codes all integers along with constant strings while program E leaves integers as they are and only codes the constant strings.

program F

This program reads file 7, replaces all standard names and user defined identifiers by their codes and outputs the result on file 11. Also it puts all user defined identifiers on file 12.

program G

This program will check files 1 and 2 to see if any triple-character special symbols (symbols made from three special characters), exist in the language and if so it reads file 11, replacing all triple-character special symbols by their codes and outputs the result on file 13.

program H

This program codes double-character special symbols in the same way as program G did for triples.

program I

This program codes all single-characters left in the text and finally creates file 15. On this file we only have integers.

program J

There is a file of all those reserved words which are used in syntax-error recovery, semantic and code emitting actions used in our small language. This file is called "action-rw-file" and numbered 16. We arrange for this file to be coded using programs C through I. Now we have "action-rw-file" and its appropriate code file, file 15. This program reads these two files and writes some "constant definitions" to be included in program N. As an example, suppose we have our action file looking as follows:

nl sp rg

and its appropriate code file as

132 135 139

the output of this program would be as follows

```
#define  nl  132
#define  sp  135
#define  rg  139
```

which are constant definitions valid in the programming language "C".

program K

The transition tables of all the languages we implement are made automatically by program N, except for the smallest language in the SL series which is made by hand. Once this is done we have the transition table for this language (SL0) in table in file 15. This program reads this code file, and outputs it on file 18 with the actual structure usable by program N.

program L

This program is used to make the action table of our smallest language in SL series as in program K.

program M

This program reads file 6 and amends some constant definitions on file 19.

program N

This program has 4 modes

1) syntax checking
2) semantic checking
3) code emitting
4) transition table making.

In mode 1 the program reads source text (which is now converted into integer numbers), from "code-file", and using the transition table, checks for syntax validity.

We use mode 2 when there are no syntax errors and mode 3 when there are neither syntax nor semantic errors.

PROCESSES USED IN THE SYSTEM

In this system we have the following processes each using a sequence of programs.

Process 1

Executing programs A and B. This process is used each time a change is made to "language symbols".

Process 2, Coding

As we explained earlier, there are two ways of coding a file

1) code all symbols: in this case we execute programs C, D, F, G, H and I; C reads the input text and I terminates with having file 15, all codes.

2) Code all symbols but integers. We do the same as above but we use program E instead of D.

Process 3

Code "action-rw-file" using process 2 (do not code integers), and then execute program J. File 17 is made at the end of this process. This process is used after any change in "language symbols" or our Small Language.

Process 4

Code transition table of SLO and then execute program K. This process terminates with having transition table of SLO made with exactly the same structure as those made automatically by program N.

Process 5

Code "action table data" of SLO and then execute program L. This process creates the "action table" for SLO, with the final structure usable in program N.

CHAPTER 8: DISCUSSION & CONCLUSIONS

We first discuss the extent to which KHAR meets the objectives of the research as regards the use of working storage, and the overall size of the system.

We then consider how the multipass approach leads to a simple structure for KHAR, which in turn gives it both extensibility and portability. The need to have a clear and flexible interface for the language designer is discussed.

The multipass approach adopted, together with the graphical location of semantic actions within the syntax of a language, defined as operations on a simple stack mechanism leads us to consider the potential of this approach for the definition of language semantics.

We reconsider the limited range of languages for which KHAR was intended in the light of its flexibility and suggest that the approach might be extended to languages such as PASCAL, or into other fields of application.

We conclude by considering the application of KHAR to the compilation of languages for microprocessors.

SIZE

The present KHAR implementation occupies about 12000 bytes of code, with 2000 bytes of constants, and requires 16000 bytes of working storage. Only 4000 bytes of this working storage are required for the KHAR machine itself. The remainder is used for data which can be kept on secondary memory. The encoded graphs for PL/0, including code generation, require about 7000 bytes, and may be regarded as read-only constants.

Thus, we estimate that a working compiler for PL/0 could be implemented using 24k bytes of ROM for the fixed tables and code of KHAR. 8k bytes of RAM would leave about 6000 bytes free for program text and dictionaries, since KHAR uses less than half its work space in compiling PL/0.

SIMPLICITY & EXTENSIBILITY

The semantic mechanisms proposed in [CORDY] for SP/6, a severe subset of PL/1, consist of a symbol table, modified to behave as a stack, and three other stack mechanisms, one of which also used entries of the same class as those in the symbol table. The mechanisms consist of four stack structures and over 50 semantic actions defined for the structures. This is an order higher than KHAR. Cordy has to take account of type within his mechanisms, and introduces semantic choice actions, which choose which path to take. We avoid semantic decisions based on knowing within KHAR the type being handled. We reduce the choice to "care" or "do not care" about the type of object being handled in the pass. Further, the outcome only affects the

semantic action taken, not the path through the graph. Thus our graphs are a reduced form of those in [CORDY].

The effect of this is to introduce complexity into the graphs rather than into the internal structure of KHAR. This complexity can be handled successfully because of the multipass approach adopted. Thus use of KHAR to act as a translator for a language with new features, say, an additional type, COMPLEX, does not require the introduction of new mechanisms or semantic actions into KHAR.

For example, at one stage in the consideration of attribute propagation within expressions, we considered the introduction of a new primitive to operate on the stack. However, careful reconsideration of the problem showed that syntax graphs and a set of semantic actions could be defined using the existing basic operations to handle this extension.

The most significant changes between AML/1 and PL/0 are the introduction of scope and the need for type-checking. These changes required the change of KHAR from a machine capable of generating code for a language with a CFG to a machine capable of generating code for a typed, block structured language. The change required the addition of six new operations defined on the stack, and the ability to index the stack, that is, MARK, FLUSH, SCOPE, SEARCH and CHECK, together with the INDEX register and its use in other actions. These features were added to KHAR in about eight working hours, requiring the addition of 60 or so lines of code to the program of KHAR. The syntax of SL was redefined and the SL translator (or table builder) recompiled within this time.

PORTABILITY

We have implemented KHAR using global variables, so that the stack mechanism of the C language is used for subroutine entry and return only. The depth of nesting used is below eight. Thus the coding of KHAR as it stands demands only a minimal support from the hardware for nesting of subroutines.

The data structures in KHAR are all one-dimensional integer arrays. Thus a simple machine architecture with limited indexing capibility should be able to support the KHAR machine.

The only other hardware requirement would be inexpensive secondary storage, capable of random access from KHAR.

The final requirement for portability would be a version of KHAR written in a language supported by itself. The code generation passes would be redefined to generate code for the new machine and the system recompiled.

CLEAR & FLEXIBLE INTERFACE

The interface to KHAR is essentially graphical and its encoding only requires a knowledge of the syntax of SL and the semantics of the simple KHAR mechanisms.

The interface itself is completely independent of the language used to implement KHAR, and thus remains invariant across implementations.

The flexibility of the interface, and the minimal nature of the semantic mechanisms available to the designer, requires skill in the use of the system on the part of the language designer. Once these are mastered, the semantic graphs produced, as claimed in [CORDY], become a ready means of communicating the exact semantics of the language to its users. Indeed, Cordy claims the use of semantic graphs to be superior to other means of achieving this.

DEFINITION OF LANGUAGE SEMANTICS

The reliance on PASCAL to express the semantics of a language defined using an LL(1) affix grammar in [BOCHMAN] produces a textual definition which can only be read if one understands PASCAL. Also, semantics and code generation are considered together, although the details of the latter are concealed by the use of a procedure "generate" which has to be supplied by the user of the compiler writing system. The result is that it is hard to see what the semantic meaning is.

We consider that the essentially graphic nature of the presentation of semantics in KHAR, and the individual refinement of the semantics imposed by the multipass approach, make it possible for both designer and user to select an aspect of semantics and isolate its effects precisely.

Further, KHAR implements a language once it has been expressed as a set of SL "programs" and translated so that a definitive implementation is immediately available. The translator may well be relatively inefficient but it is available as a standard by which to

judge other compilers for the language.

POTENTIAL FOR DEVELOPMENT

We feel that there is considerable potential for development based on KHAR. Our experience shows that a minor change in KHAR makes a wide range of application possible. The system was intended to deal with relatively simple and small languages, approximately subsets of PASCAL, but with low-level operations on the machine architecture, rather than the more abstract operations of PASCAL. KHAR can be used to translate such languages.

We suggest that KHAR could be adapted to tackle the problem of strict checking of user defined scalar types, which has been avoided in PASCAL, and to handle the evaluation of constant expressions at compile time.

A possible approach to the former is to use the special (table-building) actions of KHAR, which are available at all times, to construct at compile time an additional type checking pass or passes derived from the type declarations in the program. This extension is well beyond the original objectives of the work but the possibility has been noted.

The evaluation of constant expressions at compile time would be a simple extension of KHAR if only integer arithmetic were allowed. The stack mechanism would need to be extended by adding arithmetic operations to KHAR and the corresponding operators to SL. A technique similar to that suggested for checking the type of expressions shold

be sufficient.

APPLICABILITY TO PROGRAMMING LANGUAGES FOR MICROPROCESSORS

The problem of language design for microprocessors has been briefly outlined in the introduction. In summary, we may say that two conflicting design goals have to be attained. The language must provide the user with access to all the features of the machine yet provide him with all the protection which can be given by a modern, high-level, language.

Yet another aim of the designer is to make the language useful for the programming of more than one microprocessor. If he succeeds in doing this, then the user will have lost the semantic clues given to him by the peculiarities of the assembler about the semantics of the machine for which he is programming. We think that the ability to introduce separate definitions of the semantics appropriate to different machine architectures as separate passes, associated closely with the code generation for that machine, which exists in KHAR, would allow the language designer to check the static semantics of the program by defining an appropriate pass.

CONCLUSIONS

This work on KHAR which began by reconsidering compiler technology to achieve a highly multipass translator system has resulted in a system which, although not fully extended in this work, both will be useful in research into the design of high-level languages for low-level programming, because of the clear interface provided for the designer, and will provide a translator system for such languages which has a low read/write storage requirement.

REFERENCES

[AMMAN]

Amman, U., "The Method of Structured Programming Applied to the Development of a Compiler", International Computing Symposium 1973, A. Gunther et al, eds., (Amsterdam: North-Holland, 1974), pp93-99.

[BAUERed]

Bauer, F.L., & Eickel, J., eds., Compiler Construction, An Advanced Course, 2nd ed., (Berlin: Springer-Verlag, 1976).

[BOCHMANN]

Bochmann, G.V., & Ward, P., "Compiler Writing System for Attribute Grammers", Comp J vol 21 no 2 pp144-148

[BROWNa]

Brown, P.J., "The ML/1 Macro Processor", Comm ACM vol 10 Oct 1977, pp618-623.

[BROWNb]

Brown, P.J., "Macro Processors and Software Implementation", Comp. J. vol 12, Nov 1969, pp327-331.

[CORDY]

Cordy, J.R., "A Diagrammatic Approach to Programming Language Semantics", Technical Report CSRG-67, (University of Toronto, Computer Systems Research Group, 1976).

[GLENNIE]

Glennie, A., "On the Syntax Machine and the Construction of a Universal Compiler", Technical Report No 2 (AD-240512), (Carnegie-Mellon University, 1960).

[GRIES]

Gries, D., Compiler Construction for Digital Computers, (New York: Wiley 1971).

[HOARE]

Hoare, C.A.R., "Hints on Programming Language Design", ACM Symposium on principles of programming languages, (Boston 1973).

[JAMES]

James, L.R., "A Syntax Directed Error Recovery Method", Technical Report CSRG-13, (University of Toronto, Computer Systems Research Group, 1976).

[JENKINS]

Jenkins, D.G., "A Microprocessor Language: version 1", private communication, (Dept. of Computing Science, Univ. of Glasgow, 1976).

[JENSEN]

Jensen, K., & Wirth, N., PASCAL User Manual and Report, Lecture Notes in Computer Science, vol 18, (Berlin: Springer-Verlag 1974).

[KERNIGHAN]

Kernighan, B.W., & Plauger, P.J., Software Tools, (Reading,Mass.: Addison-Wiley, 1976).

[LECARME]

Lecarme, O., & Bochmann, G.V., "A Compiler Writing System, User's Manual", (Departement d'Informatique, Univ. de Montreal, Dec 1974, revised July 1975).

[PASKO]

Pasko, H.J., "Pseudo-Machine for Code Generation", Tech. Report CSRG-30, Univ. of Toronto, 1973.

[PIERCE]

Pierce, R.H., "Source language debugging on a small computer", Comp J vol 17 no 4 pp313-317 1974

[PUG]

Pascal Users Group, DEC PDP-11 (ESI), Implementation Notes, Pascal News, nos 9 & 10, p83, Sept 1977

[TANENBAUMa]

Tanenbaum, A.S., "A General Purpose Macroprocessor as a Poor Man's Compiler-Compiler", IEEE Transactions on Software Engineering, SE-2, vol 2, June 1976, pp121-125.

[TANENBAUMb]

Tanenbaum, A.S., "Implications of Structured Programming for Machine Archtecture", CACM vol 21 no 3 pp237-246, 1978

[WATTa]

Watt, D.A., "Analysis Orientated Two Level Grammars", PhD Thesis, Univ. of Glasgow, 1974

[WATTb]

Watt, D.A., "The Parsing Problem for Affix Grammars", Acta Informatica, vol 8 no 1 pp1-20 1977

[WILCOX]

Wilcox, T.R., "Generating Machine Code for High-Level Programming Languages", PhD Thesis, Cornell University, 1971.

[WIRTHa]

Wirth, N., "The Programming Language Pascal", Acta Informatica, vol 1 pp35-63, 1971.

[WIRTH]

Wirth, N., Algorithms + Data Structures = Programs, (Englewood Cliffs, N. J.: Prentice-Hall, 1976).