An Improved Tool for Automated Compiler Construction

Michael Andrew Brian Parkes

Doctor of Philosophy

The University of Aston in Birmingham

February 1989

The University of Aston in Birmingham

An Improved Tool for Automated Compiler Construction

Michael Andrew Brian Parkes
Doctor of Philosophy
February 1989

Summary

Since the advent of High Level Programming languages (HLPLs) in the early 1950s researchers have sought ways to automate the construction of HLPL compilers. To this end a variety of Translator Writing Tools (TWTs) have been developed in the last three decades. However, only a very few of these tools have gained significant commercial acceptance.

This thesis re-examines traditional compiler construction techniques, along with a number of previous TWTs, and proposes a new improved tool for automated compiler construction called the Aston Compiler Constructor (ACC). This new tool allows the specification of complete compilation systems using a high level compiler oriented specification notation called the Compiler Construction Language (CCL). This specification notation is based on a modern variant of Backus Naur Form (BNF) and an extended variant of Attribute Grammars (AGs). The implementation and processing of the CCL is discussed along with substantial examples including a CCL specification for the programming langauge Pascal. The CCL is shown to have an extensive expressive power, to be convenient in use, and highly readable, and thus a superior alternative to earlier TWTs, and to traditional compiler construction techniques. The execution performance of CCL specifications is evaluated and shown to be acceptable.

A number of related areas are also addressed, including tools for the rapid construction of individual compiler components, and tools for the construction of compilation systems for multiprocessor operating systems and hardware. This latter area is expected to become of particular interest in future years due to the anticipated increased use of multiprocessor architectures.

Keywords : Compilation, Syntax Analysis, Attribute Grammars, Code Generation, Translator Writing Tools.

Acknowledgements

I would like to take this opportunity to thank some of the people who have helped and encouraged me during my research at Aston University. I would particularly like to thank my supervisor Dr. Edward Elsworth for his direction and assistance, my wife Carol and my mother Alma for their encouragement and Dr. Brian Gay for the use of departmental resources while writing up.

The work described in this thesis was supported by a grant from the Science and Engineering Research Council (SERC).

Table of Contents

Table of Figures

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 8

Appendix 1

Appendix 2

Chapter 1

Introduction

The first High Level Programming Languages (HLPLs) were developed in the early part of the 1950s. The first HLPL compiler was developed between 1954 and 1957 by International Business Machines (IBM), for the IBM 705 computer. Since these early years HLPLs have received wide-spread acceptance and are now used in most areas of computer programming. This wide-spread acceptance has lead to a large demand for HLPL compiler construction.

The construction of HLPL compilers was not well understood, in the early years, and there was little theoretical work to draw upon when designing or constructing a compiler. Three decades of research has helped to relieve many of the problems associated with compiler construction and has led researchers towards the concept of developing automated compiler construction tools, or Translator Writing Tools (TWTs) as they are more usually known.

The ultimate aim of most of the research on TWTs has been to construct systems that were capable of automatically constructing a HLPL compiler from formal specification of a programming language and a target machine. It was believed that the development of such systems would make the construction of compilers much more rapid and would greatly reduce the costs associated with compiler construction.

The construction of HLPL compilers using TWTs is the general theme of this thesis. In the following chapter the components of a traditional HLPL compiler are reviewed. An examination is then made of each of these components to see how existing TWTs can be used to aid their construction. In chapters 3 and 4 a

new form of TWT and compiler specification language are proposed with the aim of further reducing the effort associated with traditional HLPL compiler construction. In chapters 5 and 6 the implementation and operation of this new TWT is examined, with reference being made to a number of small demonstration examples. In chapter 7 a specification for the programming language Pascal is reviewed. Finally, in chapters 8 and 9 the performance of this new TWT is discussed and a number of conclusions are drawn.

Chapter 2

A Review of Traditional Compiler Construction Techniques

Introduction

The traditional techniques used in commercial High Level Programming Language (HLPL) compiler construction have changed little in the past two decades. In outline, these traditional techniques are to define a HLPL compiler in a HLPL, such as C, using a five phase structure.

The overall structure of a typical traditional compiler is as follows.

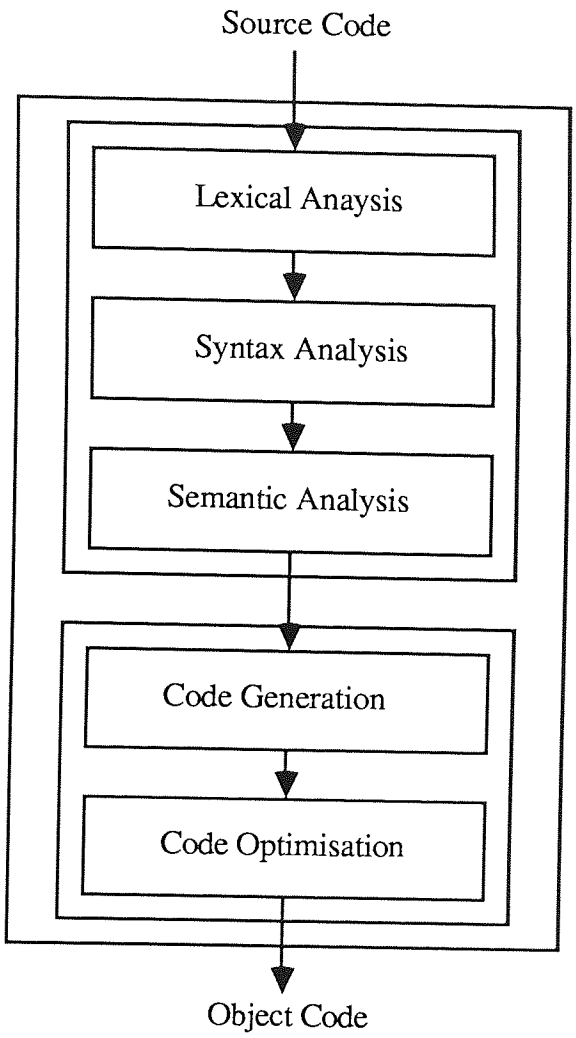


Figure 2.1

The first three compiler components, in a traditional compiler, are usually grouped together and called the compiler 'Front End' or Analysis phase. The last two compiler components are also usually grouped together and called the compiler 'Back End' or Synthesis phase.

The main commercial motivation for using traditional compiler construction techniques is that these techniques allow the construction of high quality compilers. Such compilers are usually compact, run time efficient and produce good quality code. This is important in the commercial world as a poor quality compiler can lead to loss of reputation and sales. However, constructing compilers in this way is both time consuming and expensive. These are major drawbacks and ways to overcome them have long been sought.

Research aimed at overcoming these problems started as long ago as 1962, an early example of such work is the Brooker Morris compiler compiler [Brooker 1962]. It was hoped that it would be possible to construct tools capable of automating the construction of high quality compilers and compiler components. Such tools were called Translator Writing Tools (TWTs), as previously noted. Research in certain areas of TWTs (particulary parser generation) proved so popular that P.J. Brown [Brown 1979] commented "An enormous amount of effort has been spent in developing these automatic tools. The effort has been rather disproportionate, given that parsing is only a small part of a compiler. A huge number of automatic parsing tools have been written, most of which have been dead and forgotten within a few years of their birth". This certainly seems to

be the case. A selective bibliography on TWTs by Meijer and Nijholt [Meijer 1982] contains over a 180 references. A number of other bibliographies are referenced by the authors, such as [Fisher 1981], for further reading in selected associated areas. However, after almost three decades and a great deal of research only a few TWTs have achieved much commercial impact. Again P.J. Brown [Brown 1979], commenting on TWTs, said "A few however are good and have stood the test of time. You may well find one available on your computer".

We can conclude therefore that it would seem worthwhile to re-examine the whole area of TWTs in relationship to traditional compiler construction techniques. It is believed that such a re-examination will highlight a number of areas where improvements can be made to existing traditional construction techniques by developing improved TWTs.

The following sections examine the main phases of traditional compilers and contrast traditional construction techniques with the automated construction techniques offered by a selection of existing TWTs.

Lexical Analysis

The first phase of most traditional HLPL compilers is lexical analysis. The main purpose of this phase is to determine and check the lexical structure of the source code read by a compiler. This source code is usually input by the lexical analysis phase and grouped into lexical categories known as tokens. These tokens are then passed on to the syntax analysis phase of the compiler.

It is widely accepted that the construction of lexical analyzers can be automated by the use of lexical analyzer generators. A great deal of research has been carried out in this area and several lexical analyzer generators already exist (see the bibliography [Meijer 1982] and the review [Ganapathi 1986]). A typical and well known example of such a system is Lex [Lesk 1976]. This system is capable of constructing a lexical analyzer automatically from a formal lexical notation. This notation consists of a series of lexical token specifications defined using a modified form of Regular Expressions (REs).

For example, the crucial part of a typical Lex specification may be as follows.

```
%%

"PROGRAM"                          return( PROGRAM );
"VAR"                              return( VAR );
"BEGIN"                            return( BEGIN );
"IF"                               return( IF );
"THEN"                             return( THEN );
"ELSE"                             return( ELSE );
"FI"                               return( FI );
"END"                              return( ENDS );

":="                               return( ASSIGN );
"+"|"-"                            return( SUM );
"*"|"/"                            return( TERM );
"="|"<>"|"<"|"<="|">"|">="         return( COMPARISON );
","                                return( COMMA );
";"                                return( SEMI_COLON );
"."                                return( DOT );

[A-Z][A-Z0-9]*                     return( IDENTIFIER );
[0-9]+                             return( VALUE );

" "|"\t"|"\n"                      ; /* Skip white space */
.                                  return( OTHER_CHARACTER );

%%
```

Figure 2.2

This specification defines the recognition of a number of keywords, (eg. 'PROGRAM', 'VAR', etc), identifiers, numbers and various other delimiters, (eg. ':=', '+', '-', etc). Such specifications can be processed by Lex to give the source code for a lexical analyzer in the programming languages C or Ratfor. When processing a specification Lex performs the following steps. First the specification is processed to give a Non-deterministic Finite state Automaton (NFA). It is then further processed to give a Deterministic Finite state Automaton (DFA). It is finally reprocessed to produce tables which are incorporated into the final source code generated by Lex. This source code can then be compiled or incorporated into some other software system.

Lex is widely available and has been used in the construction a number of commercial compilers, for example the Portable C Compiler (PCC) [Johnson 1979]. The lexical analyzers generated by Lex usually operate at acceptable speeds. In a test conducted by the author, typical Lex generated analyzers processed source at around 4,000 characters per second on a machine rated at 0.6 MIPS. Although hand crafted lexical analyzers can operate at higher speeds their construction is often more complex and time consuming and inherently less reliable.

It would therefore seem that Lex demonstrates a valid approach for automating the construction of lexical analyzers for traditional HLPL compilers.

Syntax Analysis

The second phase of most traditional HLPL compilers is syntax analysis. The main purpose of this phase is to determine and check the syntactic construction of the statements read by a compiler. The tokens passed from the lexical analysis phase are checked by the syntax analysis phase to make sure they form legal syntactic sentences. The syntax analysis phase and the following semantic analysis phase often work in close co-operation in traditional HLPL compilers.

There is a difference of opinion among researchers as to what is the best way to construct syntax analyzers [Aho 1985]. One point of view is that syntax analyzers are best constructed as a set of recursive descent top-down procedures. It is argued that this approach leads to the construction of more natural syntax analyzers that are easier to understand. The other point of view is that syntax analyzers are best constructed using a bottom-up approach. It is argued that such syntax analyzers are often faster and can deal with more complex grammars. This is a well known contentious area. It is difficult to show here which approach is likely to be the more suitable for TWT construction. Therefore, this matter is deferred and discussed in the next section.

Again, it is widely accepted that the construction of syntax analyzers can be automated by the use of syntax analyzer generators. A great deal of research has also been carried out in this area and several syntax analyzer generators already exist (see the bibliography [Meijer 1982] and the review [Ganapthi 1986]). A typical and well known example of such a system is Yacc [Johnson 1975]. This

system is capable of constructing a syntax analyzer automatically from a formal syntactic notation. This notation is based on a version of Backus Naur Form (BNF) [Naur 1963].

For example, a typical Yacc specification may be as follows.

```
%token PROGRAM VAR BEGINS IF THEN ELSE FI END
%token ASSIGN SUM TERM COMPARISON COMMA SEMI_COLON DOT
%token IDENTIFIER VALUE OTHER_CHARACTER

%%

Compiler        : Program Var Statement End;

Program         : PROGRAM IDENTIFIER;
Var             : VAR Variables SEMI_COLON;
Variables       : IDENTIFIER | Variables COMMA IDENTIFIER;

Statement       : Compound | Assignment | If;

Compound        : BEGIN Statements ENDS;
Statements      : Statement | Statements SEMI_COLON Statement;

Assignment      : IDENTIFIER ASSIGN Expression;
Expression      : Term | Expression SUM Term;
Term            : Primary | Term TERM Primary;
Primary         : IDENTIFIER | VALUE;

If              : IF Boolean THEN Statement FI
                | IF Boolean THEN Statement ELSE Statement FI;
Boolean         : Expression COMPARISON Expression;

End             : DOT;

%%

main()
   { yyparse(); }

#include "lex.yy.c"

static yyerror( value )
      register char       value[];
   { return; }
```

Figure 2.3

This specification defines the context-free syntax of a simple programming language for which a syntax analyzer is required. The specification can be processed by Yacc to automatically give the source code for a bottom-up LALR(1) syntax analyzer [Aho 1974] in the programming languages C or Ratfor. When processing a specification Yacc produces a number of tables. These tables are incorporated into a predefined skeleton syntax analyzer program and output by system. Again, the code generated by Yacc can be compiled or incorporated into other software systems.

Yacc is widely available and has been used in the construction of a number of commercial compilers, for example it was used along with Lex in the construction of the Portable C Compiler (PCC) [Johnson 1979]. The syntax analyzers generated by Yacc usually operate at acceptable speeds. In another test conducted by the author typical Yacc generated analyzers executed about 1,000 rules per second on a machine rated at 0.6 MIPS. Again, hand crafted syntax analyzers can operate at higher speeds but are often more complex and time consuming to construct and inherently less reliable.

It would therefore seem that Yacc demonstrates a valid approach for automating the construction of syntax analyzers for traditional HLPL compilers, and hence that Lex and Yacc together provide a worthwhile tool for automating the construction of the context-free part of a compiler's Analysis phase. Moreover, some researchers have stated that they feel that TWT's have already reached their goal in these areas and that little further research is required [Ganapathi 1986].

Semantic Analysis

The third phase of most traditional HLPL compilers is semantic analysis. The main purpose of this phase is to determine and check the static semantics of the HLPL source code read by a compiler. The static semantics of source code relate to the rules associated with the visibility and accessibility of program objects and type compatibility. This phase will also deal with related elements of execution semantics which can be handled at compile time, typically store allocation. Within the semantic analysis phase certain other operations are also sometimes performed, eg. global optimisation. It can be argued that the semantic analysis phase is one of the most complex phases of a traditional compiler.

The automated construction of semantic analyzers has proved more difficult than the automated construction of lexical and syntax analyzers [Ganapathi 1986]. The researchers working in this area have used a number of differing approaches in order to overcome the difficulties encountered in this domain. One of the two most popular approaches taken has been to define semantic analysers using formal mathematical notations. In particular, a great deal of research has been carried out to evaluate the usefulness of Denotational Semantics (DSs) [Scott 1971] in this area. It has been shown that it is possible using DSs to automate the construction of semantic analyzers for a range of HLPL compilers [Paulson 1982]. However, mathematical and DS specifications are considered complex and unnatural by many computer scientists. Furthermore, in a number of experiments [Paulson 1982] the execution of compilers based on such notations has been shown to be

significantly slower than that of traditional hand crafted compilers. For these reasons many researchers [Ganapathi 1986] at present consider these notations unsuitable for TWTs aimed at realistic rather than purely experimental implementation.

The second and significantly more popular approach [Meijer 1982] taken has been to increase the power of syntax analyzer generators by using alternative more powerful enrichments of CFGs, such as Van Wijngaarden (VW) grammars or Attribute (or Affix) Grammars (AGs). The most popular of the available notations has proved to be AGs. This is probably because AGs are a clean extensions of Context Free Grammars (CFGs) and are readily machine executable [Watt 1980]. This is not the case with many of the alternative notations, for example with VW grammars deciding which production rule to apply after a particular rule is in general unsolvable [McGettrick 1980].

The origins of AGs date back to papers by Knuth [Knuth 1968] and Koster [Koster 1971] who developed Attribute and Affix grammars independently at around the same time. In simple terms AGs (see bibligraphy [Raiha 1980a]) can be described as an augmented variant of CFGs, such as BNF. This augmentation allows context sensitive information and conditions to be expressed and enforced, as well as the normal context free specification.

The following example shows how AG rules can be used to specify some of the context sensitive constraints of a simple assignment statement. It is hoped that this example demonstrates the substantial additional expressiveness of AGs compared

with CFGs, and hence hints at their additional utility.

```
Assignment( ↓Environment,↑(Type1=Type2) ) =
   Identifier( ↓Environment,↑Type1 ), ↑EQUALS,
      Expression( ↓Environment,↑Type2 );

Identifier( ↓Environment,↑Environment [Name].type ) =
   ↑IDENTIFIER( ↑Name );
```

Figure 2.4

Attribute information is given in parentheses following relevant terminals and non-terminals of the underlying CFG. Attribute variables are introduced and constraints may be expressed by expressions over attributes. The symbols '↑' and '↓' suggest (imply) the direction of flow of information up or down a parse tree.

The rule 'Assignment' specifies that an assignment statement consists of an 'Identifier' followed by an 'EQUALS' terminal and an 'Expression'. When the rule 'Assignment' is called it is passed the attribute 'Environment', this is called an **inherited** attribute. The 'Assignment' rule then calls the rule 'Identifier' passing on the 'Environment' attribute. This rule inputs an 'IDENTIFIER' terminal and stores its associated value in the attribute 'Name', this is called a **synthesized** attribute. The rule 'Identifier' then does a look-up in the table stored in the 'Environment' attribute using the 'Name' attribute as a key. This look-up yields a structure containing all the information known about the identifier specified. The value of the field 'type' is then extracted from this structure and returned to the calling rule 'Assignment'. This value is then stored in the attribute 'Type1'. The

rule 'Assignment' then recognises the terminal 'EQUALS' and calls the rule 'Expression'. The value returned from the rule 'Expression' is stored in the attribute 'Type2'. Finally, the attributes 'Type1' and 'Type2' are compared to check that their values are equivalent. If this comparison is successful then the 'Assignment' rule exits, otherwise the 'Assignment' rule would fail and the calling rule would (in a more complete example) cause the generation of an appropriate error report.

When constructing AG-based syntax analyser generators a range of differing implementation techniques may be used [Aho 1985, Engelfret 1984]. The main implementation options available are: to evaluate attributes during parsing; or to construct a syntax tree, decorate it with attribute values and evaluate these attributes at some later stage after parsing. These attribute evaluation schemes are known as 'on-the-fly' and 'decorated parse tree evaluation', respectively, and may be used in conjunction with either top-down or bottom-up syntax analyzer construction. The relative advantages and disadvantages of the above attribute evaluation schemes are discussed below.

The main advantages of 'on-the-fly' attribute evaluation are that: this scheme is usually highly run time efficient; requires minimal storage for attributes; and has the ability to allow attributes to control the execution of the underlying syntax analyzer. This latter feature is considered important by some researchers [Watt 1980] as it often removes the need for the multi-symbol look-ahead used in many traditional parsers. It also allows grammars that are not necessarily LL(1) to be correctly parsed by top-down syntax analysers [Watt 1980]. The main restriction

of this approach is the requirement to evaluate attribute expressions during parsing. This requirement forces left-to-right evaluation of attributes in top-down syntax analyzers and restricted use of inherited attributes in bottom-up syntax analyzers.

An alternative to the scheme outlined above is to construct a complete (or partial) attribute-decorated parse tree during syntax analysis. Then at appropriate later times evaluate this tree. This alternative scheme removes the restrictions on attribute evaluation necessary with the previous scheme for both top-down and bottom-up syntax analyzers. A great deal of research has been carried out on developing algorithms to evaluate such attributed trees [Meijer 1982]. In outline the main algorithms that have been developed are as follows : a single left to right evaluation pass; multiple left to right evaluation passes; alternating left to right and right to left passes; and evaluation based on analysis of the initial AG. The main restrictions of this approach are the efficiency of the attribute evaluation algorithms and the requirement to store all (or significant parts) of the decorated parse tree before attribute evaluation [Raiha 1980b] [Madsen 1983]. This latter requirement is significant as it leads to the consumption of significant amounts of storage on many micro and mini-computers and in certain cases even mainframe computers. Further discussion on some of the more practical implementation considerations of AGs is given in Chapter 5.

A number of TWTs based on the techniques outlined above have been developed, for example CDL [Dehottay 1977], HLP [Raiha 1978] and NEATS [Jesperson 1978]. However, none of these tools have begun to gain the widespread

acceptance of earlier CFG based TWTs, such as Yacc. In the author's opinion this is for one or more of the following reasons: firstly, many of these systems have seemed difficult to use; they have not been able to construct complete compilation systems; they have only been able to produce compilers of inferior quality to traditional compilers; or have offered only limited advantages over traditional construction techniques. This opinion is supported in part by a number of papers, for example DeRemer [DeRemer 1975] said of the CDL system: "CDL falls short of the ideal because part of the language description, as written in CDL, is a set of program fragments that describe what the translator being constructed is to do, rather than what the language being described is to be". The HLP system is acknowledged by Raiha [Raiha 1980b] to be too inefficient for production use. Finally the NEATS system, as demonstrated in [Madsen 1983], showed little support for code generation and optimisation.

It is therefore felt that there is need here for new and improved forms of TWTs. It is believed that may of the difficulties discussed above could be overcome by providing better "hooks" for syntax analysis, code generation and code optimisation within the semantic analysis phase. It is also felt that it is important to make TWTs better oriented towards production use, both in terms of ease of use and reasonable efficiency. Otherwise there is little hope that such tools will be used in commercial compiler construction. The above objectives were some of the early goals of the author's research and were considered during the design of the new TWT demonstrated in this thesis.

Code Generation

The fourth phase of most traditional HLPL compilers is code generation. The main purpose of this phase is to translate the source code read by a compiler into appropriate machine code instructions. The machine code generated by the code generation phase is then usually either passed to a following optional optimisation phase or output to a file ready for linking, loading and execution.

Research aimed at automating the construction of code generators began in the early part of the last decade. A number of systems were produced and called Retargetable Code Generators (RCGs) (see the bibliography [Fisher 1981]). Most of the these early systems were based on interpretive code generation techniques and used macro-processors. The most common approach used was to translate the source code being compiled into some Intermediate Representation (IR) and then to macro-expand this IR into assembly code or machine code. An example of one such early system is [Wilcox 1971]. However, these early systems were slow and required relatively large quantities of processing time due to the macro processing involved.

The interpretive code generation techniques used in early RCGs were gradually replaced during the middle of the last decade by pattern-matching techniques. One well known pattern-matching based system, developed around this time, was the Portable C Compiler (PCC) [Johnson 1979]. The analysis component of this compiler was constructed using Lex and Yacc. This component analysed C source code and translated it into a number of small syntax trees. The synthesis component of this compiler (based on a Template Matching (TM) code generation

system) then examined each of these small trees, a small sub-tree at a time. The TM code generation system attempted to replace each sub-tree analysed with a suitable equivalent machine code instruction. If no suitable replacement could be found the nearest match was used and a further search was made for instructions to plug the gap. Associated with every machine code instruction was a template showing what type of sub-tree it could be used to replace, hence the name TM.

Around this time research on RCGs become somewhat more popular (see the bibliography [Fisher 1981]) and a number of other RCG systems using were developed based on heuristics, such as [Frazer 1977] and [Cattell 1978]. However, most of these early RCGs could not deal with the to the variable nature of many commercial machine architectures [Ganapathi 1986]. Therefore, during the early part of this decade attempts were made to improve these systems. One such development was made by Graham et al [Graham 1980] who improved pattern-matching code generation techniques by further automating them. The system developed by Graham et al was based on a form of CFG bottom-up parser. The system operated by parsing the IR passed to it from the analysis phase of a compiler and generating target code fragments every time certain predefined patterns were recognised.

An outline structure of Graham's system was as shown in Figure 2.5.

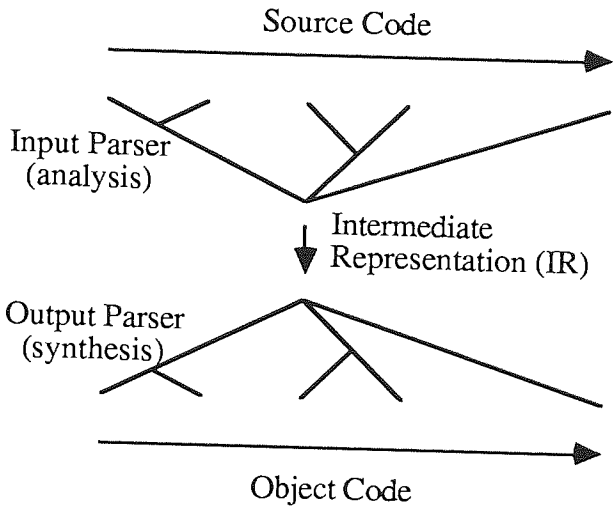


Figure 2.5

Source code is read by the analysis phase and translated into an IR. This IR is then passed to the synthesis phase. The synthesis phase parses this IR and generates target code machine code.

The techniques developed by Graham have been acknowledged as a major advance by some authors and further developed [Horspool 1987] to deal with a wider range of target machine architectures. However, a number of further difficulties have been encountered with these techniques when dealing with certain target machine features, such as non-orthogonal register sets, differing addressing modes, register pairs and overlapping register sets [Ganapathi 1986]. In order to overcome these problems a number of researchers have developed alternative systems using differing algorithms and specification notations. Perhaps one of the most interesting of these was developed by Ganapathi [Ganapathi 1980]. Ganapathi suggested the use of an AG based parser in place of the CFG parser used in Graham's original system. This modification naturally extended the

descriptive power of Ganapathi's RCG system while still retaining the elegance of Graham's original system. This approach has in turn been further developed to by Ganapathi and other researchers [Bird 1982] [Pleban 1984] to produce systems capable of automatically producing code generators of production quality [Ganapathi 1986]. An alternative related development was made by Madsen et al [Madsen 1983] who produced a form of Extended Attribute Grammar [EAG]. Madsen et al then demonstrated how two EAGs could be used "back-to-back" to produce a powerful specification language called an Extended Attribute Translation Grammar [EATG]. This specification language was used in [Madsen 1983] to specify the syntax analysis and semantic analysis phases (and also provide the basis for the code generation phase) for a compiler of a small sub-set of Pascal.

The automated construction of RCGs is still a current research area. Recently, the focus in this area has changed somewhat to include the automated construction of code generators with built-in optimisers. This is because it was realised that certain optimisations were best preformed during code generation. This area is discussed in the following section.

Optimisation

A fifth optional phase in many traditional HLPL compilers is the optimisation phase. The purpose of this phase is to improve the quality of the code emitted by the code generator. Optimisation is usually divided into two categories, namely local optimisations (which are concerned with replacing locally poor instruction sequences with improved sequences) and global optimisations (which concern the

larger code structure of the source). Hence optimisation carried out in a phase following code generation will be concerned with local optimisation. Global optimisation (if attempted) needs more information about the overall program structure and is thus typically carried out in additional phases between semantic analysis and code generation. The optimized machine code, emitted from the final stage of the optimisation phase, is usually output to file ready for linking, loading and execution.

The automated construction of local and global optimizers received little attention from researchers until the latter part of the last decade. Since then a number of researchers have begun to work in this area and a selection of automated systems have been constructed (see the bibliography [Fisher 1981] and the review [Ganapathi 1986]). Unfortunately, many of the systems developed have been tightly coupled to particular RCGs and have therefore been of limited use outside of these systems, for example [Keizer 1983]. The remaining more general systems have mainly focused on the automated construction of local optimizers leaving the area of automated global optimizer construction largely neglected.

A pioneer in the general area of automated optimizer construction was C.W. Frazer. In one of his early papers [Davidson 1980] he and a co-author demonstrated an automated system for the construction of local peephole optimizers. The system used machine instruction set specifications, based on ISP [Bell 1971], to describe the function of assembly language instructions for a range of target machines. Unfortunately, the initial implementation of this system was restricted to the optimisation of assembly language programs and operated at a rate

of between 1 and 10 instructions per second. The execution performance was improved in subsequent versions of the system [Davidson 1984] (Further reading [Kessler 1984b]). However, the scope of the system remained restricted to the optimisation of assembly language programs and was not expanded to embrace the optimisation of numerically represented machine code programs.

An alternative to the above system has been suggested in [Giegerich 1983] (associated work [Kessler 1984a]). This alternative system attempts to decompose complex instruction sequences into simpler ones.

However, in general there are a number of practical problems associated with interfacing independent local and global optimizers with code generators. One major problem is that many optimisations alter the register requirements of the optimized code by either increasing or decreasing the number of registers required. This makes it necessary to integrate such optimisations into the code generation phase, as mentioned earlier.

An attempt to integrate the automated construction of code generators and local optimizers was made in [Ganapathi 1985]. The system described in this paper used AG specifications to define integrated code generator/optimizers. The results obtained from this research were, in this author's opinion, encouraging. However, it is acknowledged that further research is required to investigate the full potential of AGs in this area. Furthermore, additional research is also required to investigate the applicability of AGs in the automated construction of global optimizers. This latter area is of great importance as certain global optimisations

(such as optimal register allocation and code motion) are vital in high quality code generation. However, these areas were beyond the scope of the author's current work and may be suggested as topics for future research.

Linking and Loading

The final phase of most traditional HLPL compilers is code reformatting. The main purpose of this phase is to reformat the code generated by a compiler into a form acceptable to the local Operating System (OS). This phase is seldom an independent compilation phase; it is more usually appended to the code generation or optimisation phase. The reformatting performed by the code reformatting phase is typically highly OS dependent but is important if a compiler is intending to use local OS facilities, such as the linker and loader.

The majority of TWTs avoid OS interfacing difficulties by generating code in local assembly language (eg. The Portable C Compiler (PCC) [Johnson 1979]). The assembly code produced by these systems is then usually assembled, by the local assembler, and linked and loaded before being executed. However, assemblers are often slow and inevitably repeat work previously done by the compiler (eg. lexical analysis). In addition to this, generating assembly language code can, in some cases, be more difficult than generating the same code in pure binary. Ideally, it would be desirable for a TWT to have a choice of interfacing either with an assembler or directly to the native OS. Unfortunately, in practice this is difficult achieve due to the high OS dependency of this area.

There have been a number of attempts to overcome the difficulties outlined above by constructing universal assemblers [Keizer 1983] and machine independent linkers and loaders [Davidson 1982]. However, it has proved difficult to interface these systems to a range of typical OS object libraries, due to the wide range of formats used. Therefore, such systems have not been able to access native OS support routines and functions. Most OSs provide a large number of such routines and functions many of which are of potential use to compilers. For example these may include: access to standard mathematical and input/output libraries; a convenient symbolic reference mechanism; standard means to establish run-time debugging information. Thus, failure to be able to use these routines and functions leads to a significant increase in the effort required by the compiler writer.

The initial objective of the author's research was to construct a system capable of resolving the above problem. However, it was soon realised that any system capable of overcoming this problem was also likely be useful in other areas of automated compiler construction. Thus, the initial aims of the author's research were expanded to include the development of a new TWT suitable for the construction of complete compilation systems.

Summary

We have seen in this chapter that existing TWTs are already capable of processing formal lexical and syntax specifications and automatically producing acceptable lexical and syntax analysers. It was noted that some researchers consider that

research in these areas has now reached its goal [Ganapathi 1986]. We then continued and examined recent research in the areas of automated semantic analyser, code generator and code optimiser construction. It was seen that a considerable amount of research had been carried out in these areas and that a particular specification notation, namely AGs, had been used by a number of researchers in all of these areas. The utility of AGs has also been demonstrated in a number of other current research areas, such as natural language understanding and the automated construction of context-sensitive language-based editors. The latter is understood to be a particularly active research area at the present time (see [Reps 1984] for an example of such work). On the basis of these observations the author believes that AGs provide a suitable notation for the specification of all or significant parts of complete compilation systems. Finally, our examination of recent research was completed by examining the problem of interfacing compilers to operating systems. The significance of this area was noted.

The initial research carried out by the author indicated that it might be possible to construct a small but relatively powerful AG-based TWT suitable for the specification of complete or significant parts of entire compilation systems (eg. lexical analysis, syntax analysis, semantic analysis, code generation and local optimisation). This had not been achieved when the author's research began in 1983 [Paulson 1982]. It was believed that the development of such a system would be a major advance and might be suitable for the design and construction of commercial compilers. It therefore seemed worthwhile to conduct an experiment to evaluate the accuracy of this idea. Such an experiment has been carried out and the system produced is described and evaluated in the following chapters.

Chapter 3

An Overview of the Aston Compiler Constructor

Introduction

The Aston Compiler Constructor (ACC) was the name given to an experimental automated compiler construction tool developed at Aston University, England. This system was developed to investigate the viability of constructing Translator Writing Tools (TWTs) based solely on a Attribute Grammars (AGs) and capable of addressing the issue of constructing complete compilers.

The Aston Compiler Constructor

The development of the ACC was carried out in two main stages. The first stage of development was to design a new AG based specification language suitable for specifying a wide range of compiler components. It was intended that this new specification language would be clear, consistent, easy to use and support modularisation. The new specification language designed during this stage of the ACC project was later called the Compiler Construction Language (CCL). A detailed description of this language is given in Chapter 4 of this thesis.

A small example CCL specification is given in Figure 3.1.

```
TITLE Example

CONSTANTS
   Message1  = "\t : This is a boolean constant\n";
   Message2  = "\t : This is an integer constant\n";

INPUT TOKENS
   Boolean   = "TRUE" | "FALSE";
   Integer   = INTEGER;
   VOID      = '\0'->'\377';

OUTPUT TOKENS
   Message   = TEXT;

TYPES
   Constant  = UNION
               {
                  boolean   : TEXT;
                  integer   : INTEGER;
               };

VARIABLES
   Boolean   boolean;
   Integer   integer;
   Constant  constant;

RULES
   start
      = { input( ↑constant ), output( ↓constant ) };

   input( ↑constant )
      = input_boolean( ↑constant ) |
        input_integer( ↑constant );

   input_boolean( ↑boolean{ boolean } )
      = ↑Boolean( ↑boolean );

   input_integer( ↑integer{ integer } )
      = ↑Integer( ↑integer );

   output( ↓constant )
      = ↓Boolean( ↓constant.boolean ), ↓Message( ↓Message1 ) |
        ↓Integer( ↓constant.integer), ↓Message( ↓Message2 );

END
```

Figure 3.1

The second stage of ACC development was to design and implement a system to validate, translate and execute compiler specifications written in the CCL.

The overall structure of the final system produced was as follows.

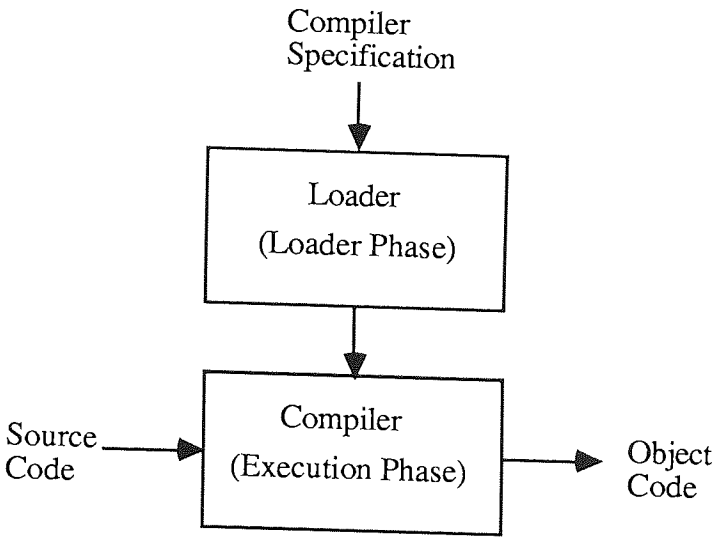


Figure 3.2

The ACC Loader Phase translates compiler specifications, such as the example CCL specification given above, into a representation suitable for execution. A typical simple CCL specification may be translated as shown in Figure 3.3.

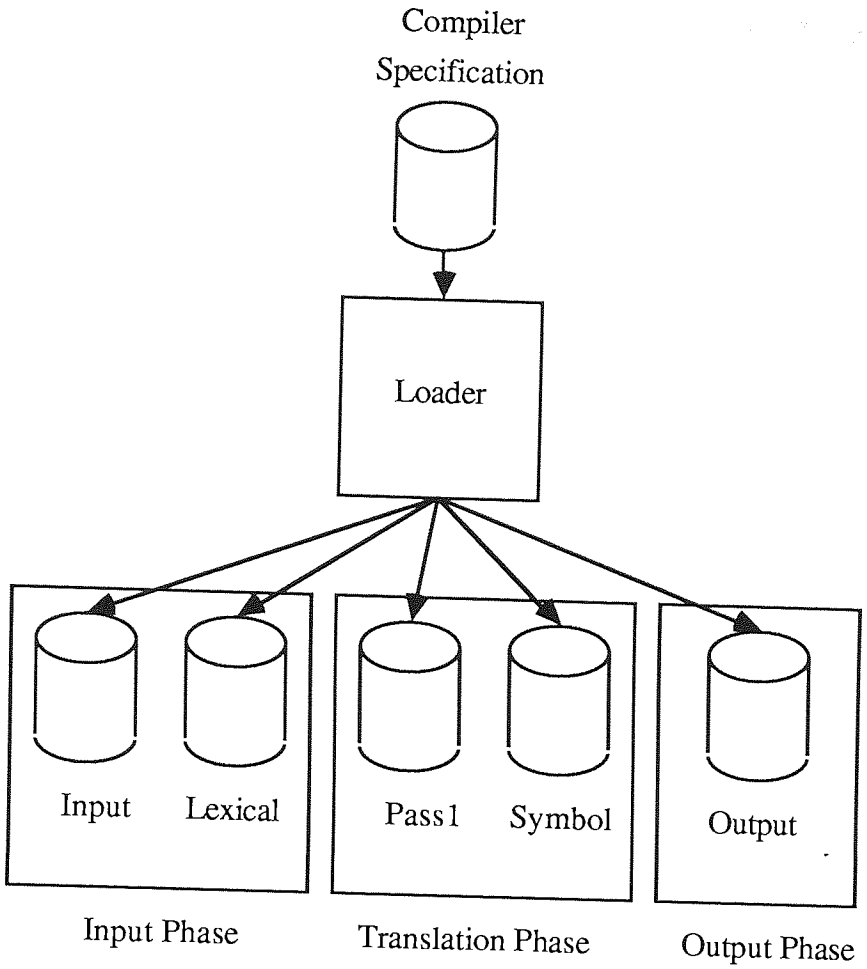


Figure 3.3

The ACC Execution Phase accepts translated CCL specifications and executes them. The execution of a simple CCL specification might be as shown in Figure 3.4, which follows.

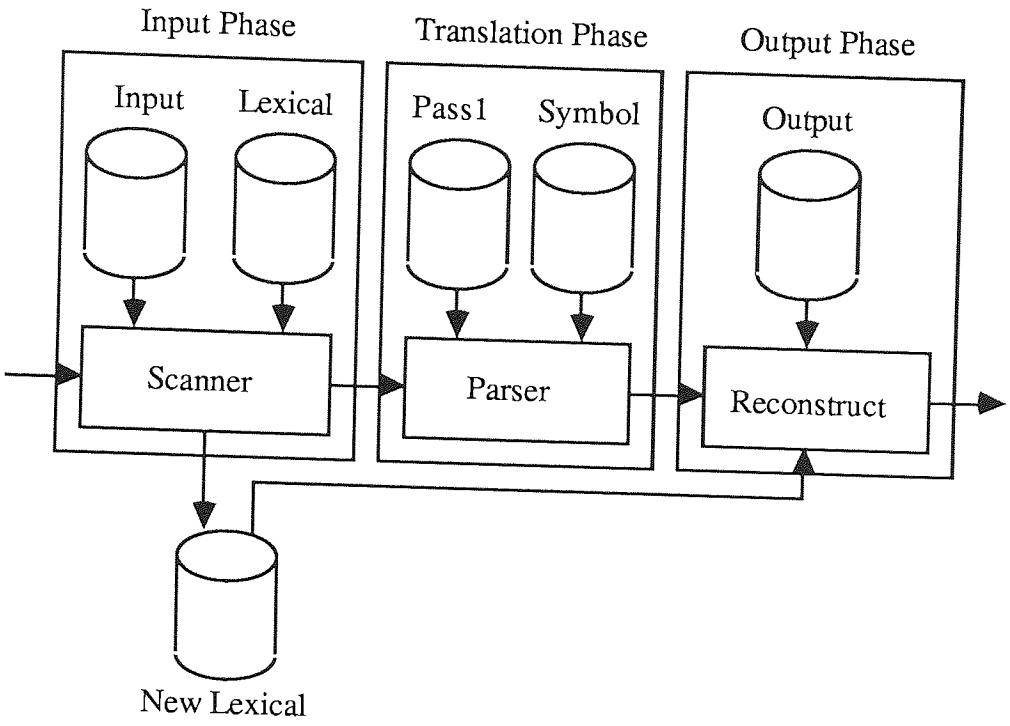
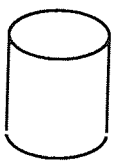


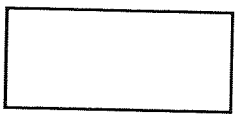
Figure 3.4

A detailed description and discussion of the implementation of the ACC is given in Chapter 5 of this thesis.

Note - in Figures 3.3 and 3.4 and subsequently,



denotes a file containing data to be referenced or processed



(rectangular objects) denote processors.

Chapter 4

The

Compiler Construction Language

Introduction

The Aston Compiler Constructor (ACC) uses a special high level compiler oriented notation called the Compiler Construction Language (CCL). A compiler or compiler component specification defined in the CCL can be processed by the ACC to automatically produce a multi-phase table-driven compiler or compiler component.

The overall structure of systems produced by the ACC is as follows.

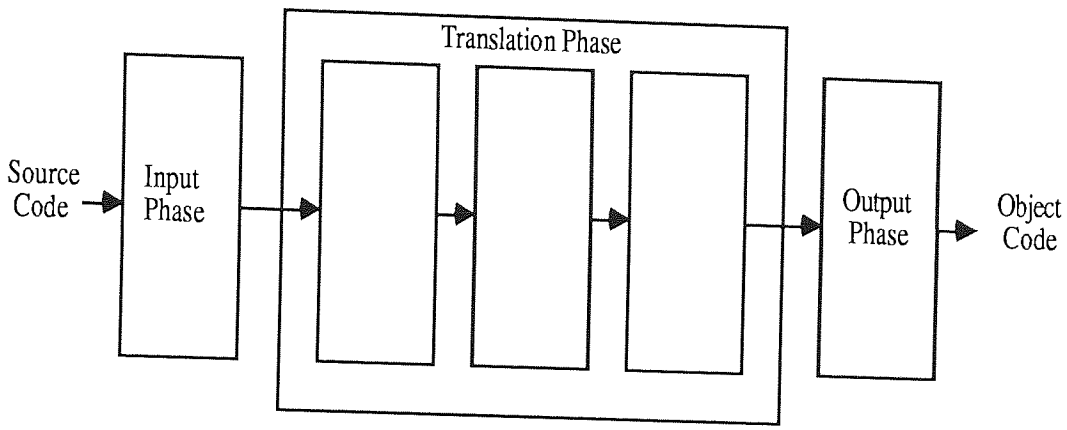


Figure 4.1

The 'Input Phase' reads source code and prepares it for the 'Translation Phase'. The 'Translation Phase' consists of zero or more Attribute Grammar (AG) based translators which translate this source code into some new form. The 'Output Phase' reformats the output from the 'Translation Phase' into some final target representation called object code.

The Compiler Construction Language

The Compiler Construction Language (CCL) was developed from an initial basis provided by the draft British Standards Institute (BSI) meta-language proposed in [Scowen 1982] and a Translator Writing Tool (TWT) specification language proposed in [Madsen 1983].

A complete CCL specification thus consists of an underlying Context Free (CF) specification expressed essentially in Scowen's style augmented by attribute formulas which typically express the static semantics of and the translations to be applied to the source language. For programming convenience and reliability attribute values and variables are typed. Both compound and simple types are supported. For convenience in constructing translators, direct means are supplied for relating CF terminals and attribute values to both source and target language symbols.

A CCL specification consists of a number of separate sections. The structure and content of these sections is discussed in detail in under the following headings : 'Title', 'Constants', 'Input Tokens', 'Output Tokens', 'Types', 'Variables' and 'Rules'. The formal specifications given under these headings are extracts from a complete formal CCL specification given in Appendix 2. The attributed CF specification itself appears in the 'Rules' section of a CCL text. The remaining sections provide the support structure for the typing of attributes and for defining the input and output interfaces to be used.

Title

A CCL specification begins with the 'TITLE' section. This section is used to name a CCL specification and is mandatory.

The 'TITLE' section is formally defined as follows.

```
Title          = "TITLE", Identifier.
Identifier     = ('A'→'Z'|'a'→'z'),
                   {'A'→'Z'|'a'→'z'| '0'→'9'|'_'}.
```

Figure 4.2

A typical CCL title may be as follows.

```
TITLE Pascal
```

The title given to a CCL specification is used as a prefix in the names of the various files generated by the ACC.

Constants

The second section in a CCL specification is the 'CONSTANTS' section. This section is used to define CCL compile-time constants and is optional. The constants defined in the this section may be used throughout the remainder of the CCL specification.

The 'CONSTANTS' section, if present, is formally defined as follows.

```
Constants          = "CONSTANTS", Constant_Rules.
Constant_Rules     = Constant_Rule, { Constant_Rule }.
Constant_Rule      = Identifier, "=", Expression, ";".

Expression         = Sets, { ( "="|"≠"|">"|"≥"|"<"|"≤" ), Sets }.

Sets               = Binary, { ( "∩" | "∪" ), Binary }.
Binary             = Shift, { ( "&" | "|" ), Shift }.
Shift              = Sum, { ( "<<" | ">>" ), Sum }.

Sum                = Term, { ( "+" | "-" ), Term }.
Term               = Power, { ( "*" | "/" ), Power }.
Power              = Unary, { "^", Unary }.

Unary              = { "~" | "-" }, Primary.

Primary            = Constant |
                     Variable |
                     [ Function ], "(", Expression, ")".

Constant           = Character|Hexadecimal|Octal|Real|String|Unsigned.

Character          = '\0'→'\377'.
Hexadecimal        = ('0'→'9'|'A'→'F'|'a'→'f'),
                        {'0'→'9'|'A'→'F'|'a'→'f'}.
Octal              = '0'→'7', {'0'→'7'}.
Real               = '0'→'9', {'0'→'9'},".", '0'→'9', {'0'→'9'}.
String             = '"', {'\0'→'\377'}, '"'.
Unsigned           = '0'→'9', {'0'→'9'}.

Variable           = Identifier |
                     Identifier, ".", Variable |
                     Variable, "[" Expression "]" |
                     "<<", Variable, ">" |
                     "<", Variable, ">>".

Function           = "ABS"|"INTEGER"|"LENGTH"|"REAL"|"UNSIGNED".
```

Figure 4.3

The CCL supports a wide range of constants, variables, operators and functions. These are discussed in the text below, except CCL variables which are discussed

in a later section.

The types of constants supported by the CCL are as follows.

Type Name	Type	Example
CHARACTER	Numeric	'a'
HEXADECIMAL	Numeric	0x12aF or 0X123C
INTEGER	Numeric	123
OCTAL	Numeric	0123
REAL	Numeric	123.456
TEXT	String	"Any Text Sting"
UNSIGNED	Numeric	123

Figure 4.4

Character or string constants may contain special character sequences, as follows.

Character	Meaning
\b	A back space character.
\f	A form feed character.
\n	A new line character.
\r	A return character.
\t	A tab character.
\Any other character	The character.
\Octal number	The character with this Octal character code.

Figure 4.5

The CCL supports a selection of operators to allow a wide range of calculations and manipulations to be specified within CCL specifications. These operators are divided in two main groups, namely binary operators and unary operators.

The CCL binary operators available are as follows.

Priority	Operator	Type	Meaning
2 (Highest)	^	Numeric	Exponential.
3	* /	Numeric	Multiplication and Division.
4	+ -	Numeric	Addition and Subtraction.
5	<< >>	Numeric	Logical Shifts.
6	& |	Numeric	Binary And and Or.
7	∪ ∩	Tables	Table Union and Intersection.
8	= ≠	Any	Comparisons.
8	< ≤	Numeric	Comparisons.
8 (Lowest)	> ≥	Numeric	Comparisons.

Figure 4.6

The CCL binary operators are divided into four main classes. These are arithmetic operators, logical operators, table operators and relational operators. The arithmetic operators are '^', '*', '/', '+' and '-'. These operators are used to perform exponentials, multiplication, division, addition and subtraction respectively. The logical operators are '<<', '>>', '&' and '|'. These operators are used to perform logical left and right shifts and binary AND and OR operations. The table operators are '∪' and '∩'. These operators are used to join and intersect tables (discussed later). Finally, the relational operators are '=', '≠', '<', '≤', '>' and '≥'. These operators are used to set constraints on sub-expressions within CCL rules.

In order to keep CCL specifications concise and readable the CCL relational operators function in a way slightly different than might normally be expected.

Consider the following CCL expression.

```
( Count + 1 > 0 < 10 )
```

When evaluating this expression the ACC would first evaluate the sub-expression 'Count + 1'. The value of this sub-expression would then be compared with 0 and 10. If the value of the sub-expression was in the range 1 to 9 then this value would be returned as the result of the expression. However, should the value of the sub-expression be outside the range 1 to 9 the expression would be deemed invalid and would cause the CCL rule it was associated with to fail.

The CCL unary operators available are as follows.

Priority	Operator	Type	Meaning
1	-	Numeric	Unary Minus
1	!	Numeric	Bitwise Negation

Figure 4.7

These unary operators may be used to negate CCL expressions.

The CCL binary and unary operators shown above may be used in arbitrarily complex numeric expressions. The CCL allows the numeric types within such expressions to be combined as required, with any necessary type conversions being performed automatically by the CCL processor.

Consider the following expression.

```
( 7.321 + 0x4 * 8.76 )
```

The hexadecimal value '0x4' in this expression would be automatically floated to the real value '4.0' by the CCL processor before the '*' operation was carried out.

Finally, the CCL supports a number of built-in functions defined as follows.

Name	Function
ABS	Return the absolute value of following expression.
INTEGER	Truncate the value of the following expression to integer.
LENGTH	Return the length of following string.
REAL	Float the value of the following expression to real.
UNSIGNED	Truncate the value of the following expression to unsigned.

Figure 4.8

These functions operate on the numeric expressions (contained within brackets following them), except the function 'LENGTH' which operates on textual values.

The following example shows a number of typical CCL constant expressions.

```
CONSTANTS
        Character       =   'A';
        Integer         =   -123;
        Real            =   123.456;
        Hexadecimal     =   0x1FF;
        Octal           =   0123;
        Unsigned        =   123;
        Text            =   "A Text String";

        Case            =   'a' - 'A';
        Size            =   Case * 3;
        Truncate        =   INTEGER( Real / 5 );
```

Figure 4.9

The CCL is case sensitive so that, for example, 'Integer' in the above example does not clash with the reserved word 'INTEGER'.

Input Tokens

The third section in a CCL specification is the 'INPUT TOKENS' section. This section is used to define source language input terminals and is mandatory. The input terminals defined in this section may be used as rule terminals, and may also be used to specify CCL variable types, throughout the remainder of the CCL specification.

The 'INPUT TOKENS' section is formally defined as follows.

```
Input_Section      = "INPUT", "TOKENS", Input_Rules.
Input_Rules        = Input_Rule, { Input_Rule }.

Input_Rule         = "VOID", "=", Input_Token, ";" |
                     Identifier, "=", "CONSTANT", String, ";" |
                     Identifier, "=", Input_Token, ";" |
                     Identifier, "=", Input_Structure, ";".

Input_Structure    = "STRUCTURE", "{", Input_Clauses, "}".
Input_Clauses      = Input_Clause, { Input_Clause }.

Input_Clause       = "VOID", ":", Input_Token, ";" |
                     "CONSTANT", ":", String, ";" |
                     Identifier, ":", Input_Token, ";".

Input_Token        = Input_Choice | Simple_Type.

Input_Choice       = Input_Sequence, { "|", Input_Sequence }.
Input_Sequence     = Input_Primary, { ",", Input_Primary }.
Input_Primary      = "(", Input_Choice, ")" | "[", Input_Choice, "]" |
                     "{", Input_Choice, "}", Iterations |
                     Input_Terminal.

Iterations         = [ "*", Expression, [ "→", Expression ] ].

Input_Terminal     = Range | String | "?".
Range              = "'",'\0'→'\377',"'", "→","'",'\0'→'\377',"'".
```

Figure 4.10

The CCL allows input terminals to be defined using Regular Expressions (REs). These REs consists of a number of RE operators and RE terminals.

The RE operators available in the CCL are as shown in Figure 4.11.

Priority	Operators	Meaning
1(Highest)	( )	Enclosed clauses are to be grouped.
1	[ ]	Enclosed clauses are optional.
1	{ }	Enclosed clauses are to be iterated.
2	,	Sequence operator.
3(Lowest)	|	Alternative operator.

Figure 4.11

The RE terminals available in the CCL are as follows.

Example	Meaning
"A keyword or string"	A textual string.
'A'→'Z'	A character range.
?	Any character.

Figure 4.12

A number of typical CCL input terminals are as follows.

```
INPUT TOKENS
   Identifier   = ('A'→'Z'), {'A'→'Z'|'0'→'9'|"_"};
   Term         = "*" | "/";
   Comma        = ",";
```

Figure 4.13

The input terminal 'Identifier' above is defined as a letter followed by a number of letters, digits or underscores. The input terminal 'Term' is defined as either the symbol '*' or '/'. Finally, the input terminal 'Comma' is defined as the single character symbol ','.

When a CCL rule including any of the above input terminals is applied, the text matched by this terminal is be returned to the calling CCL rule. For example, a

CCL rule which applies the terminal 'Identifier' would automatically be returned the name of the identifier matched. This name could then be validated and other information (such as the identifier's type) could be located and extracted. This feature is very useful when writing CCL specifications.

However, in some cases the text matched by an input terminal may not required by the CCL rule which applies it. The terminal 'Comma' above, for example, always matches the text ','. This text is therefore of little interest.

We may redefine the input terminal 'Comma' as follows.

```
Comma           = CONSTANT ",";
```

This alternative specification of the input terminal 'Comma' causes the text matched by this terminal to be discarded. This feature is convenient when defining programming language reserved words.

It is also sometimes necessary to completely ignore certain input terminals and not pass them on for further processing, for example many programming languages ignore comments and white space. Therefore, the CCL supports a special keyword ('VOID') which may be used to cause the voiding (ie. discarding) of certain input terminals or text.

The following example (Figure 4.14) shows how comments and white space may

be voided in a simple programming langauge.

```
VOID            =   " " | "\t" | "\n";
VOID            =   "{", {'A'→'Z'|'a'→'z'|'0'→'9'}, "}";
```

Figure 4.14

To aid the input and output of numeric values the CCL supports a number predefined REs. These predefined REs are associated with the standard CCL variable types (defined later).

The predefined REs supported by the CCL are as follows.

```
Type            Definition
CHARACTER       '\0'→'\377'
HEXADECIMAL     ("0x"|"0X"), {'0'→'9'|'A'→'F'|'a'→'f'}
INTEGER         '1'→'9', {'0'→'9'}
OCTAL           "0", {'0'→'7'}
REAL            '1'→'9', {'0'→'9'}, ".", '0'→'9', {'0'→'9'}
UNSIGNED        '1'→'9', {'0'→'9'}

SHORT FIXED     ? * sizeof(short unsigned) ***
FIXED           ? * sizeof(unsigned) ***
LONG FIXED      ? * sizeof(long unsigned) ***
FLOAT           ? * sizeof(float) ***
LONG FLOAT      ? * sizeof(long float) ***

FILE            ('A'→'Z'|'a'→'z'), {'A'→'Z'|'a'→'z'}
TEXT            ('A'→'Z'|'a'→'z'), {'A'→'Z'|'a'→'z'}

                *** As defined locally in the C programming language
```

Figure 4.15

These predefined REs are divided into three main classes: formatted numerics, unformatted numerics, and strings. The formatted numeric REs are

'CHARACTER', 'HEXADECIMAL', 'INTEGER', 'OCTAL', 'REAL' and 'UNSIGNED'. These predefined REs are used to input characters and character formatted hexadecimal, integer, octal, real and unsigned numerics (character to binary conversion being performed). The unformatted numeric REs are 'SHORT FIXED', 'FIXED', 'LONG FIXED', 'FLOAT' and 'LONG FLOAT'. These predefined REs are used to input integer and real numerics represented in local binary format (here no conversion is required). Finally, the string REs are 'TEXT' and 'FILE'. The predefined RE 'TEXT' is used to input textual alphabetic strings. However, the predefined RE 'FILE' is more unusual. When the ACC applies this predefined RE (at execution time) it assumes that the text that matches it denotes a valid host operating system file name. The ACC automatically opens the named file as a new source file and reads its contents. When the file is exhausted it is automatically closed and input continues from the original source file.

The CCL predefined REs may be used as follows.

```
Integer         =   INTEGER;
Real            =   REAL;
Short           =   SHORT FIXED;
```

Figure 4.16

All input terminals that include predefined REs return a value to the CCL rule which applies them. The value returned by a predefined RE is of the corresponding CCL standard type. As an example the input terminal 'Integer' above would match any formatted integer string found within the source code

being read by the ACC (such as '123') and would return its binary numeric value to the CCL rule which applied it.

Finally, the CCL also supports the definition of structured input terminals. These terminals consist of a number of individual input terminal specifications joined together in series. The 'CONSTANT' and 'VOID' keywords may be used to replace field names within structured terminals. The effect of these keywords is the same as defined earlier.

Consider the following structured input terminal specifications.

```
Include         = STRUCTURE
                  {
                     CONSTANT        : "#include \'";
                     File_Name       : FILE;
                     CONSTANT        : "\'\n";
                  };

Hexadecimal     =   STRUCTURE
                  {
                      VOID           : "0x" | "0X";
                      Value          : HEXADECIMAL;
                  } ;

String          =   STRUCTURE
                  {
                      CONSTANT       : "'";
                      Value          : TEXT;
                      CONSTANT       : "'";
                  } ;
```

Figure 4.17

The fields within a structured input terminal may be accessed individually within CCL specifications (using the CCL '.' operator discussed later). Thus, the field 'Value' in the structured input terminal 'String' above may be accessed and

processed independently of the other fields in this structure. This feature can be very helpful when writing CCL specifications as enables the processing of certain classes of symbols to be specified in a way that is both clear and efficient. This is demonstrated later in Chapters 6 and 7 of this thesis.

Output Tokens

The fourth section in a CCL specification is the 'OUTPUT TOKENS' section. This section is used to define output terminals and is mandatory. The output terminals defined in this section may be used as CCL variable types and rule terminals throughout the remainder of the CCL specification.

The 'OUTPUT TOKENS' section is formally defined as follows.

```
Output_Section    = "OUTPUT", "TOKENS", Output_Rules.
Output_Rules      = Output_Rule, { Output_Rules }.

Output_Rule       = Identifier, "=", "CONSTANT", String, ";" |
                    Identifier, "=", Simple_Type, ";" |
                    Identifier, "=", Output_Structure, ";".

Output_Structure  = "STRUCTURE", "{", Output_Clauses, "}".
Output_Clauses    = Output_Clause, { Output_Clause }.

Output_Clause     = "CONSTANT", ":", String, ";" |
                    Identifier, ":", Simple_Type, ";".
```

Figure 4.18

The 'OUTPUT TOKENS' section is defined, as far as is possible, to be consistent with the 'INPUT TOKENS' section. However, we do need a number of minor differences between the definitions of these two sections, as follows: Firstly, the

use of REs and the keyword 'VOID' are not permitted in the 'OUTPUT TOKENS' section as these clauses are nonsensical when outputting information. Secondly, the keywords 'CONSTANT' and 'FILE' are redefined to denote constant text and files to be output by the ACC.

A number of typical CCL output terminals are as follows.

```
OUTPUT TOKENS
        Value       = INTEGER;
        Header      = CONSTANT "*** Header ***\n";
        Assembler   = STRUCTURE
                        {
                           CONSTANT    : "    ";
                           Function    : TEXT;
                           CONSTANT    : "\t";
                           Operand     : INTEGER;
                           CONSTANT    : "\n";
                        };
```

Figure 4.19

The output terminal 'Value' above can be used to output integers passed from CCL rules in ASCII character format. (eg. '123'). The output terminal 'Header' is a constant output terminal and can be used to output the constant text defined as is required. Finally, the output terminal 'Assembler' is a structured output terminal. This terminal may be used to output structured information from CCL rules.

Types

The fifth section in a CCL specification is the 'TYPES' section. This section is used to define new variable types and is optional. The variable types defined in this section may be used throughout the remainder of a CCL specification.

The 'TYPES' section, if present, is formally defined as shown in Figure 4.20.

```
Types             = "TYPES", Type_Rules.
Type_Rules        = Type_Rule, { Type_Rule }.

Type_Rule         = Identifier, "=", Complex_Type, ";" |
                    Identifier, "=", Type_Structure, ";".

Type_Structure    = "STRUCTURE", "{", Type_Clauses, "}" |
                    "UNION", "{", Type_Clauses, "}" |
                    "TABLE", "[", Complex_Type, "]".

Type_Clauses      = Type_Clause, { Type_Clause }.
Type_Clause       = Identifier, ":", Complex_Type, ";".

Complex_Type      = Identifier | Simple_Type.
Simple_Type       = "CHARACTER" | "INTEGER" | "HEXADECIMAL" |
                    "OCTAL" | "REAL" | "UNSIGNED" |
                    "SHORT", "FIXED" | "FIXED" | "LONG", "FIXED" |
                    "FLOAT" | "LONG", "FLOAT" |
                    "TEXT" | "FILE".
```

Figure 4.20

The CCL supports a number of standard types, as shown in Figure 4.21.

Type	Meaning
CHARACTER	A single Character
HEXADECIMAL	A hexadecimal number
INTEGER	An integer number
OCTAL	An octal number
REAL	A real number
UNSIGNED	An unsigned number

```
SHORT FIXED      A short fixed point number
FIXED            A fixed point number
LONG FIXED       A long fixed point number
FLOAT            A floating point number
LONG FLOAT       A long floating point number

TEXT             A textual string
FILE             A legal file name
```

Figure 4.21

These standard types above correspond to the predefined REs given in Figure 4.15. These types are divided into three main groups. The standard types 'CHARACTER', 'HEXADECIMAL', 'INTEGER', 'OCTAL', 'REAL' and 'UNSIGNED' may be used to define fixed-sized character, hexadecimal, integer, octal, real and unsigned variables respectively. The standard types 'SHORT FIXED', 'FIXED', 'LONG FIXED', 'FLOAT' and 'LONG FLOAT' offer a range of fixed and floating point numeric precision and may be used as alternatives to the standard types 'INTEGER' and 'REAL'. However, it should be noted that the input and output representations of these types are somewhat different (see Figure 4.15). Finally, the string types 'TEXT' and 'FILE' may be used to define variables to contain textual strings and file names respectively.

The CCL, like many traditional high level programming languages, allows the definition of arbitrarily complex data structures. These data structures may consist of structures, unions and tables and are specified using the standard types given above.

A CCL structure consists of a number of related data fields stored together as an entity. Each field within a structure can be accessed and manipulated

independently of the all other fields within the same structure.

An example of a typical structure definition is as shown in Figure 4.22.

```
Structure       = STRUCTURE
                    {
                       Field1   : INTEGER;
                       Field2   : REAL;
                       Fieldn   : TEXT;
                    };
```

Figure 4.22

The fields within a structure are accessed using the CCL dot operator. The second field in the above structure, for example, could be accessed as follows.

```
Structure.Field2
```

A CCL union is an exclusive union of a number of data fields where only one data field may be defined at any time.

An example of a typical union definition is as shown in Figure 4.23.

```
Union           = UNION
                    {
                       Field1   : INTEGER;
                       Field2   : REAL;
                       Fieldn   : TEXT;
                    };
```

Figure 4.23

Again, the defined field within a union may be accessed using the CCL dot

operator, for example the first field may be accessed as follows.

```
Union.Field1
```

The CCL structure and union keywords define objects that are in many ways analogous to the structures defined by the 'struct' and 'union' keywords in the C programming language. However, unlike the C programming language, the ACC maintains a hidden discriminant for CCL unions indicating which alternative within a union is currently defined. This discriminant can be used in conjunction with the RE alternative operator '|' to control the execution of CCL specifications.

Consider the following CCL rule.

```
...     = A(↓Union.Field1) | B(↓Union.Field2) | C(↓Union.Fieldn);
```

This rule would call the rule 'A' if the alternative 'Union.Field1' is defined, rule 'B' if the alternative'Union.Field2' is defined or rule 'C' if the alternative 'Union.Fieldn' is defined.

Finally, a CCL table is a collection of keyed and unkeyed entries stored in a simple hash table. Such tables may contain zero or more entries, each entry being stored in sequential order of arrival.

A definition of typical CCL table is as follows.

```
Table           = TABLE[ Structure ];
```

An entry in a CCL table may be accessed in a number ways. The first or last item in a table may be accessed using the CCL sequence operators, as follows.

Selector	Meaning
<< Table >	Select the first entry in the table
< Table >>	Select the last entry in the table

Figure 4.24

The CCL sequence operators cause the rule they are associated with to fail if the table being accessed is empty.

Alternatively, a keyed entry in a table may be accessed by using the CCL table selection operator.

A typical CCL table selection may be as follows.

```
Table[ 5 ]
```

This table selection clause causes the table entry with the numeric key '5' to be selected and accessed. If no such keyed entry exists within the table, this selection clause would again cause the CCL rule it was associated with to fail. The CCL has no restrictions on the types of table keys that may be used to access table

entries and allows different types of table keys to be mixed within a single table.

A wide range of typical compiler data structures can be easily defined using CCL structures, unions and tables. These data structures include symbol tables, stacks, queues and simple look-up tables. The utility of each of the constructs outlined above is demonstrated in the following specification of a small symbol table.

```
TYPES
   Invariant     = UNION
                   {
                      Character        : CHARACTER;
                      Integer          : INTEGER;
                      Real             : REAL;
                      Unsigned         : UNSIGNED;
                   };

   Variant       = STRUCTURE
                   {
                      Name             : TEXT;
                      Size             : INTEGER;
                      Type             : TEXT;
                   };

   Symbol        = UNION
                   {
                      Constant         : Invariant;
                      Variable         : Variant;
                   };

   Symbol_Table = TABLE[ Symbol ];
```

Figure 4.25

Variables

The sixth section in a CCL specification is the 'VARIABLES' section. This section is used to introduce variables for the following 'RULES' section(s) and is optional. The variables defined in this section may be used throughout the

remainder of the CCL specification.

The 'VARIABLES' section, if present, is formally defined as follows.

```
Variables          = "VARIABLES", Variable_Rules.
Variable_Rules     = Variable_Rule, { Variable_Rule }.

Variable_Rule      = Complex_Type, Variable_Names, ";".
Variable_Names     = Variable_Name, { ",", Variable_Name }.

Variable_Name      = Identifier, [ "=", Complex_Value ].

Complex_Value      = Expression |
                     "{", Complex_Value, { ",", Complex_Value }, "}" |
                     Identifier, "{", Complex_Value, "}" |
                     "[" [ Table_Value, { ",", Table_Value } ] "]".

Table_Value        = [ Expression, "→" ], Complex_Value.
```

Figure 4.26

The CCL supports two main types of variables. These are initialized variables and uninitialized variables.

An initialized variable can most simply be described as a global variable. The contents of such variables do not need to be passed between CCL rules but may be accessed directly by any rule within a CCL specification. These variables are ideal for the storage of static or semi-static data structures and greatly reduce the overhead of passing such data structures between CCL rules.

The following are typical specifications of initialized variables.

```
INTEGER    High=10, Low=2;
TEXT       Words= "*** Hello World ***";
```

Figure 4.27

The CCL also allows data structures to be initialized.

The data structure defined in Figure 4.25 could be initialized as follows.

```
Symbol_Table New_Table =
                [
                   "Constant" → Invariant{ Integer{ 4 } },
                   "Variable" → Variant{ { "Variable",1,"CHAR" } }
                ];
```

Figure 4.28

An uninitialized variable can most simply be described as a local variable. Such variables are used to hold dynamic values as they are passed between CCL rules.

The following are typical specifications of uninitialized variables.

```
INTEGER    First, Second;
REAL       Total;
```

Figure 4.29

The primary function of CCL uninitialized variables is to permit the labelling of dynamic values as they are passed between CCL rules. This improves the clarity of CCL specifications and makes them easier to understand.

Rules

The final section in a CCL specification is the 'RULES' section. This section is used to define the CCL translation phases and may appear zero or more times. If no rules section is used, the power of the system is reduced to a general lexical processor. In any programming language translation application, one rules section would be used for each translation pass required.

The 'RULES' section, if present, is formally defined as follows.

```
Rules             = "RULES", AG_Rules.
AG_Rules          = AG_Rule, { AG_Rule }.

AG_Rule           = Rule_Header, [ "=", Rule_Body ], ";".

Rule_Header       = Identifier, [ "(", Rule_Parameters, ")" ].

Rule_Parameters   = Rule_Parameter, { ",", Rule_Parameter }.
Rule_Parameter    = "↑", Complex_Value | "↓↑", Variable |
                    "↓", Variable | "↓", "<", Variable, ">".

Rule_Body         = Rule_Sequence, { "|", Rule_Sequence }.
Rule_Sequence     = Rule_Primary, { ",", Rule_Primary }.
Rule_Primary      = "(", Rule_Body, ")" | "[", Rule_Body, "]" |
                    "{", Rule_Body, "}", Iterations |
                    Rule_Terminal.

Rule_Terminal     = Rule_Warning | Rule_Error |
                    Rule_IO_Terminal | Rule_Call.

Rule_Warning      = "<<", String, ">>", Rule_Body.

Rule_Error        = "<<", Identifier, ",", Identifier,
                        ",", String, ">>", Rule_Body.
```

```
Rule_IO_Terminal  = "↑", Identifier, [ "(", "↑",Variable, ")" ] |
                    "↓", Identifier, [ "(", "↓",Complex_Value, ")" ].

Rule_Call         = Identifier, [ "(", Rule_Operands, ")" ].

Rule_Operands     = Rule_Operand, { ",", Rule_Operand }.
Rule_Operand      = "↑", Variable | "↑", "<", Variable, ">" |
                    "↓↑", Variable | "↓", Complex_Value.
```

Figure 4.30

Rule Structure

A CCL rule is defined in two main parts. The first part is a rule header and the second part is an optional rule body. The rule header defines the name of the CCL rule being defined and its parameters. These parameters allow information to be passed in and out of the CCL rule to and from the calling rule. Each parameter in a rule header is preceded by an arrow indicating the direction of information flow. An arrow thus '↓' denotes information flowing into the rule (from its caller), an arrow thus '↑' denotes information flowing out of the rule (to its caller), and two arrows thus '↓↑' denote information flowing in and out of the rule (from and to its caller). These arrows in a rule header therefore denote read-only parameters, write-only parameters and read/write parameters. Read-only and read/write parameters are usually assigned to uninitialized variables. A write-only parameter may consist of an arbitrarily complex expression, the result of which is passed back to the calling rule.

The following is a typical CCL rule header.

```
Calculate( ↓A,↑↓B,↑A*B ) = ...
```

This rule must be called with a read-only variable which is assigned to 'A' and a read/write variable which is assigned to 'B'. The values of these variables may used within CCL rule specified after the symbol '='. On completion of this rule the value of 'A*B' is evaluated and the result returned to the calling rule.

The CCL also supports a special table construction parameter to allow the easy construction of stacks and queues. This special parameter type is denoted thus '↓<Table_Name>' and allows the addition of single or multiple, keyed or unkeyed entries into CCL tables.

A queue or a stack, for example, could be formed using the following rules.

```
Add_Queue( ↓Qvalues )    = Queue( ↓Qvalues );
Queue( ↓<Qtable> )       = ... ;
```

Figure 4.31

When the rule 'Queue' is called (due to the application of the rule call in the rule 'Add_Queue') the entries in the table 'Qvalues' would be appended to the entries in the global table 'Qtable'. The entries added to the table 'Qtable' may then be accessed in either queue or stack order. To access the entries in queue order the expression '<<Qtable>' would be used. This operator yields the value of and

removes a single entry from the front of the table 'Qtable' each time it is applied. To access the entries in stack order the expression '<Qtable>>' would be used. This operator yields the value of and removes a single entry from the end of the table 'Qtable' each time it is applied.

A CCL rule body is constructed using REs over (possibly attributed) grammar symbols. These REs consist of RE operators and RE terminals.

It has long been accepted that REs are a significant enhancement to conventional BNF-style Context Free Grammars (CFGs) in the areas of readability and conciseness. DeRemer and Jillig [DeRemer 1984] showed how these advantages could also be extended (in theory) to a class of AGs which had formerly been restricted essentially to the basic BNF-style. The ACC was developed without prior knowledge of this work and demonstrates practically how REs may be used in AGs processed by 'on-the-fly' attribute evaluators.

The RE operators available in the CCL are as shown in Figure 4.32.

Priority	Operators	Meaning
1(Highest)	( )	Enclosed clauses are to be grouped
1	[ ]	Enclosed clauses are optional
1	{ }	Enclosed clauses are to be iterated
2	,	Sequence operator
3(Lowest)	|	Alternative operator

Figure 4.32

The RE terminals available in the CCL are called 'Rule Terminals', 'Rule Calls', and 'Rule Errors'. These terminals along with a number of associated topics are

discussed in the sub-sections below.

Rule Terminals

A CCL rule terminal is used to input or output information to or from a CCL rule. All rule terminals must be defined in either the 'INPUT TOKENS' or 'OUTPUT TOKENS' section of a CCL specification. A rule terminal is denoted within a CCL rule by a leading '↑' or '↓' symbol followed by a input or output terminal name. If the input or output terminal has an associated value, this value may be input or output using a bracketed value clause.

A number of typical CCL rule terminals might be as shown in Figure 4.33.

Grammar Terminal	Meaning
↑IF	Input (ie. recognise from the source text) a terminal of type 'IF'.
↑IDENTIFIER(↑name)	Input a terminal of type 'IDENTIFIER' and store its associated value in the variable 'name'.
↓HEADER	Output a terminal of type 'HEADER'.
↓CUBED(↓A*A*A)	Output a terminal of type 'CUBED' with its associated value of 'A*A*A'.

Figure 4.33

Rule Calls

A CCL rule call is used to invoke other rules within the same CCL 'RULES' section. A rule call consists of the name of the rule to be called followed by an optional parameter list in brackets. In general a rule call parameter will be an

expression composed from declared (attribute) variables, constant identifiers, and appropriate operators and paretheses. The parameter types in a rule call must match the corresponding parameter types in the rule header.

An example of a typical rule call may be as follows.

```
Add_Up( ↓A,↓B,↑C )      = Sum( ↓A,↑↓B,↑C ), ... ;
Sum( ↓W,↑↓X,↑W+X )      = ... ;
```

Figure 4.34

The variables 'A', 'B', 'C', 'W' and 'X' are all defined as integers. When the rule 'Sum' is called the value of the variables 'A' and 'B' will be passed to this rule and will be known locally as 'W' and 'X'. On completion of the rule 'Sum' the values of the variables 'W' and 'X' would be added together and the result returned to the variable 'C'. The value of 'C' might be used in a subsequent part of the rule 'Add_Up', or returned via a parameter to the rule which called this rule.

Visibility within Regular Expression Formats

The RE operators '|', '[', ']' and '{', '}' sometimes affect the visibility of values returned by RE terminals. This is because the execution of an RE terminal enclosed within any of these operators can not always be guaranteed. For this reason, the variables returned by a RE terminal enclosed within any of these operators are often considered undefined outside these of operators.

Consider the following REs.

Regular Expression	Notes
↑A(↑a)	The variable 'a' is defined.
[↑A(↑a)]	The variable 'a' is undefined.
{ ↑A(↑a) }	The variable 'a' is undefined.
{ ↑A(↑a) } * 3	The variable 'a' is defined.
(↑A(↑a,↑c) | ↑B(↑b,↑c))	The variables 'a' and 'b' are undefined. The variable 'c' is defined.

Figure 4.35

The CCL adopts the following rule : 'A variable is only considered visible outside an optional or iterative RE if its definition within that particular RE can be guaranteed'. Suprisingly, this feature turns out to be quite advantageous in practice as it forces the author of a CCL specification to consider carefully the processing of such clauses. Quite often these clauses are associated with special cases within the syntax of a source language. Thus, careful consideration of these cases helps to reduce the likelihood of subsequent errors.

Rule Errors

Finally, a CCL rule error is a special RE terminal used in the detection and reporting of errors discovered during the execution of a CCL specification. There are two types of rule error terminals. These are **warning terminals** and **error terminals**. A **warning terminal** is used to trap minor faults detected during the execution of a CCL specification. This is achieved by placing a warning node before any clause in a CCL rule which may possibly fail, but where that failure is

not necessarily fatal (eg. the use of a non-standard construction). Should the clause fail the failure is trapped, a warning message printed, but no further action is taken - from the point of view of the calling rule the rule in question will have completed successfully. A warning node consists of a '<<' symbol followed by some warning string and a '>>' symbol.

A typical warning terminal might be as follows.

```
...     = ↑MOVE, ↑IDENTIFIER( ↑name ),
                <<"To missing">> ↑TO, ↑IDENTIFIER( ↑name );
```

Figure 4.36

The warning terminal in the above example would trap the omission of the input terminal 'TO' and print a warning message.

An **error terminal** is used to trap more serious failures which require the resetting of the ACC Execution system. Once again the error terminal is placed before any clause that may fail in a serious manner. Should a failure be detected during the execution of this clause the failure would be trapped and an error message printed. The error terminal would then cause a number of input terminals to be skipped to ensure correct recovery from the failure. The format of an error node is a '<<' symbol followed by a recovery token name, a search limit token name, an error string and a '>>' symbol.

A typical error terminal might be as follows.

```
...  =
   <<SEMI_COLON,DOT,"Invalid Statement">>
      ( Assignment( ↑type ) | Compound | If );
```

Figure 4.37

The above error terminal would trap any faults discovered within the grouped input terminals that follow it. Should this terminal be activated a number of input tokens would be skipped until a token of type 'SEMI_COLON' was discovered. However, the search for a token of this type will not proceed past a token of type 'DOT'. If a token of type 'SEMI_COLON' is found then the recovery is considered successful and processing continues. If no token of type 'SEMI_COLON' can be found then the recovery is considered unsuccessful and processing ceases.

An Example of a Complete Rules Section

```
RULES
   Statement =
      <<SEMI_COLON,DOT,"Invalid Statement">>
         ( Assignment( ↑type ) | Compound | If );

   Assignment( ↑( type1 = type2 ) ) =
      Identifier( ↑type1 ), ↑ASSIGN,  Expression( ↑type2 );

   Expression( ↑( type1 = type2 ) ) =
      Identifier( ↑type1 ), Sum( ↓type1,↑type2 );

   Sum( ↓type,↑type ) =
      [ ↑SUM( ↑operator ), Expression( ↑type ) ];

   Identifier( ↑symbol_table[ name ].type ) =
      ↑IDENTIFIER( ↑name );
```

```
Compound =
   ↑BEGINING, Statement, { ↑SEMI_COLON, Statement }, ↑ENDING;

If =
   ↑IF, Boolean( ↑type ),
      ↑THEN,  Statement,
         [ ↑ELSE, Statement ],
   ↑FI;

Boolean( ↑( type1 = type2 ) ) =
   Expression( ↑type1 ), ↑BOOLEAN( ↑bool ), Expression( ↑type2 );
```

Figure 4.38

This example specifies the context-sensitive syntax of a simple programming language. The rule 'Statement' defines the programming language statements available and would deal with any errors discovered during processing of this specification. The rules 'Assignment', 'Compound' and 'If' define the structure of assignment statements, compound statements and an if-statements within this simple language. The rules 'Expression', 'Sum', and 'Identifier' specify the structure of numeric expressions used in assignment statements. Finally, the rule 'Boolean' defines the structure of boolean expressions used within if-statements.

Summary

In this chapter we have examined the design and structure of the CCL. We have seen that the CCL is fundamentally based on attributes but nevertheless has been designed using a Pascal and C style idiom. This idiom has been used in order to give the CCL an easy to use and familiar style. We have also seen that the CCL provides powerful interfacing facilities. It will be shown later that these facilities

allow the ACC to be interfaced to a wide range of operating systems, compilers and other software components.

In the next chapter we shall examine the software components of the ACC and we shall see how these components can be use to translate and execute CCL specifications.

Chapter 5

The Implementation of the Aston Compiler Constructor

Introduction

The implementation of the Aston Compiler Constructor (ACC) was carried out on a High Level Hardware (HLH) Orion mini-computer running the Berkley UNIX operating system. The source code of the ACC consists of around 10,000 lines of code written in the C programming language and about a further 500 lines of code written for Lex and Yacc. The ACC is portable across UNIX systems and in a trial has been ported onto HLH Orion2, a Sun Systems SUN3 and a Digital Equipement (DEC) Vax 11/750.

The ACC has been constructed using a simple two phase structure. The first phase of the ACC is called the Loader Phase. This phase translates compiler specifications into a representation suitable for subsequent execution. The second phase of the ACC is called the Execution Phase. This phase accepts translated compiler specifications and executes them, by interpretation.

The overall structure of an ACC-built compiler was shown in Figure 3.2 but is repeated here for convenience (Figure 5.1).

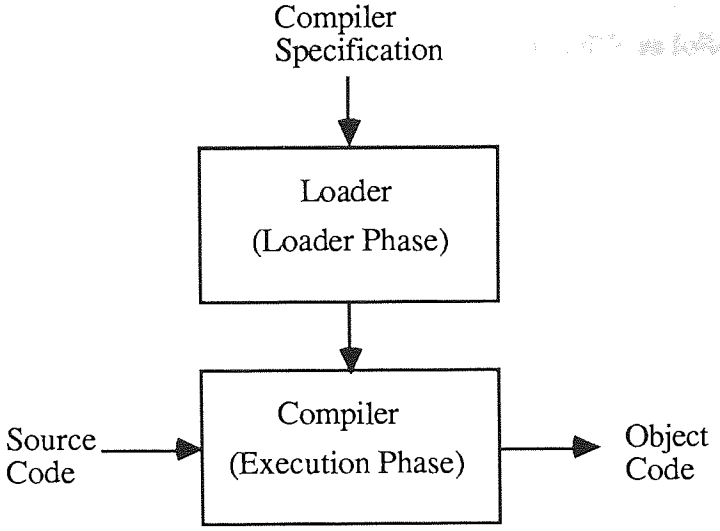


Figure 5.1

The implementation of the ACC Loader and Execution Phases is discussed in detail in the following sections.

The Loader Phase

The ACC Loader Phase translates compiler specifications, written in the Compiler Construction Language (CCL), into a representation suitable for execution, called ACC Intermediate Execution Format (IEF). The ACC Loader Phase consists of a single program called the LOADER.

Loader

The LOADER program reads CCL specifications via the standard C input channel and translates them into ACC IEF.

A small CCL specification may be translated by the LOADER as follows.

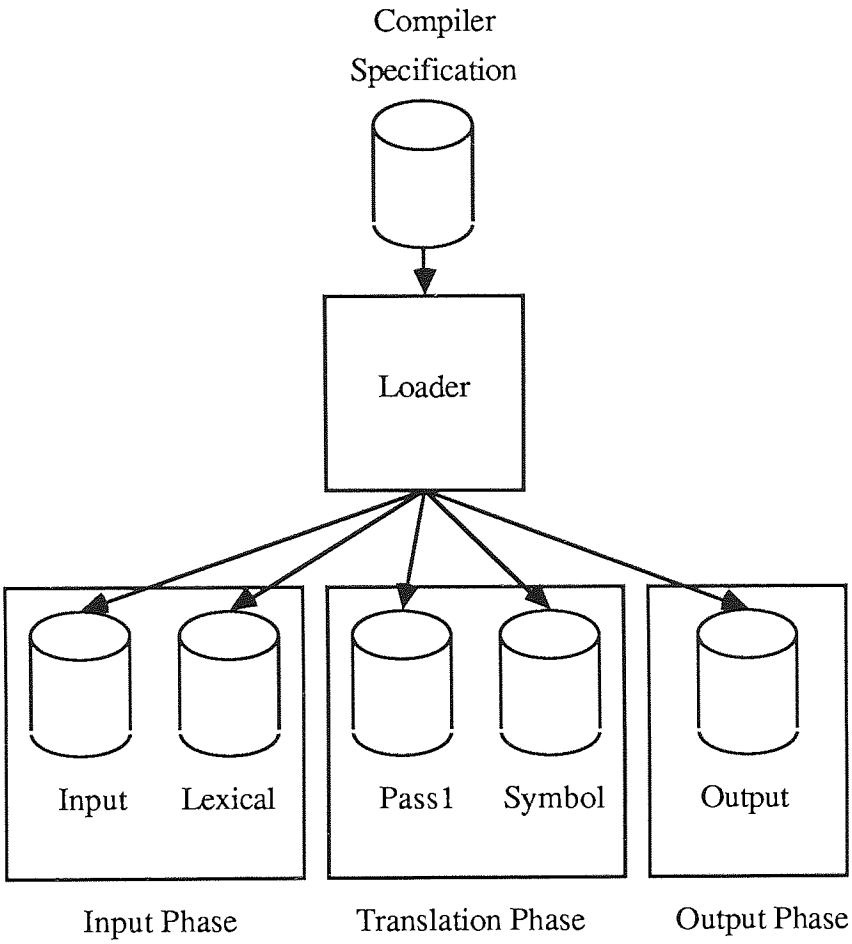


Figure 5.2

(Figure 3.3 repeated for convenience).

The ACC IEF generated by the LOADER is distributed between a number of files. These files can be divided in two main groups, namely control files and information files. The control files in the above example are 'Input', 'Pass1' and 'Output'. These files contain information to control various components of the ACC Execution Phase (discussed later). The information files in the above

example are 'Lexical' and 'Symbol'. These files contain information about CCL constants and variables.

The LOADER checks the lexical, syntactic and static semantic structure of CCL specifications before translating them into ACC IEF. The translation of each section of a CCL specification has a number of particular features (See Chapter 4 for CCL sections). In order to highlight these features and explain the operation of the LOADER we shall examine the translation of a small example CCL specification.

The outline structure of this example is as follows.

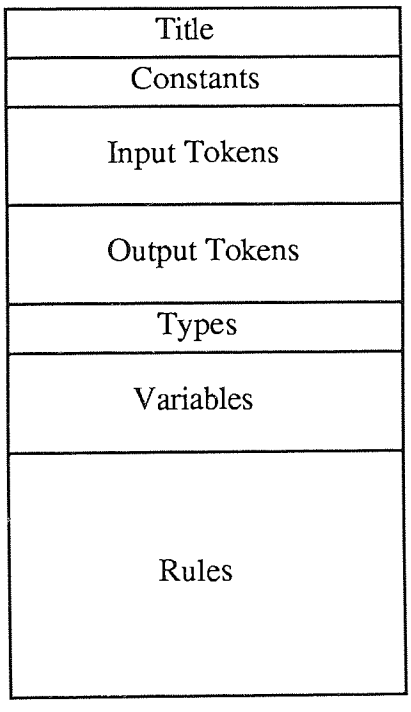


Figure 5.3

We conduct this examination by examining small segments of the example, corresponding to the CCL sections within the specification. We shall hence see how the LOADER translates each particular CCL section into ACC IEF. A complete listing of the example is given in Appendix 3.

Title

The first CCL section in the example specification is the 'TITLE' section. This section is used to name the specification, it is as follows.

```
TITLE Example
```

The title given to a specification is used by the LOADER to generate file names for the ACC IEF files output by the program.

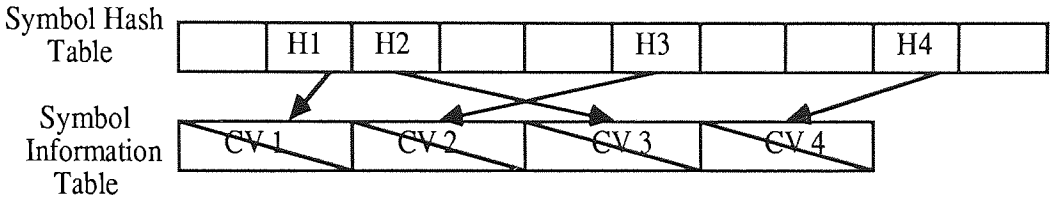
Constants

The second CCL section in the example is the 'CONSTANTS' section. This section is used to allow the specification of CCL compile-time constants, as follows.

```
CONSTANTS
   LDA          = 0x10;
   ADA          = 0x11;
   SBA          = 0x12;
   STA          = 0x13;
```

Figure 5.4

The LOADER evaluates constant expressions and stores them within its internal symbol table. The above constant expressions would be evaluated and stored in the LOADER's internal symbol table as follows.



Key

H1	The hash value of 'LDA'	CV 1	The value of constant 'LDA'
H2	The hash value of 'ADA'	CV 2	The value of constant 'ADA'
H3	The hash value of 'SBA'	CV 3	The value of constant 'SBA'
H4	The hash value of 'STA'	CV 4	The value of constant 'STA'

Figure 5.5

All constants are stored in the LOADER's internal symbol table in ACC value format. The structure of this format is as follows.

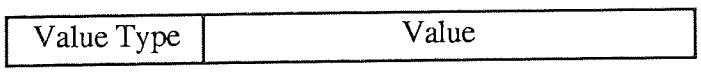


Figure 5.6

Whenever a constant identifier is found within a CCL specification, the LOADER replaces that identifier with the corresponding constant value.

The ACC currently imposes a number of restrictions on CCL constant expressions. The main restriction at present is that the arithmetic range and numeric representation of CCL expressions is tied to the arithmetic range and numeric represention of the host computer system. This restriction leads to increased complexity in CCL specifications of cross-compilers (where the target machine has a different arithmetic range and numeric representation to the host machine). Another current implementation restriction in the LOADER inhibits the use of the CCL table ('∪' '∩') and relational ('=' '≠' '>' '≥' '<' '≤') operators in constant expressions. This restriction was introduced simplify the original implementation of the LOADER and is likely to be removed in future versions of the program.

Input Tokens

The third CCL section in the example is the 'INPUT TOKENS' section. This section is used to allow the specification of ACC input terminals, as follows.

```
INPUT TOKENS
   ASSIGN       = CONSTANT ":=";
   SUM          = "+" | "-";

   IDENTIFIER   = ('A'→'Z'), {'A'→'Z'|'0'→'9'};
```

Figure 5.7

The LOADER translates the set of input terminal specifications into a Deterministic Finite-state Automaton (DFA). A standard algorithm is used to perform this function, see [Aho 1985]. The DFA produced by this algorithm is then stored in a

table, called the Input Terminal Table (ITT). The above input terminals would be translated into a DFA and stored in the LOADER's ITT as shown in Figure 5.8.

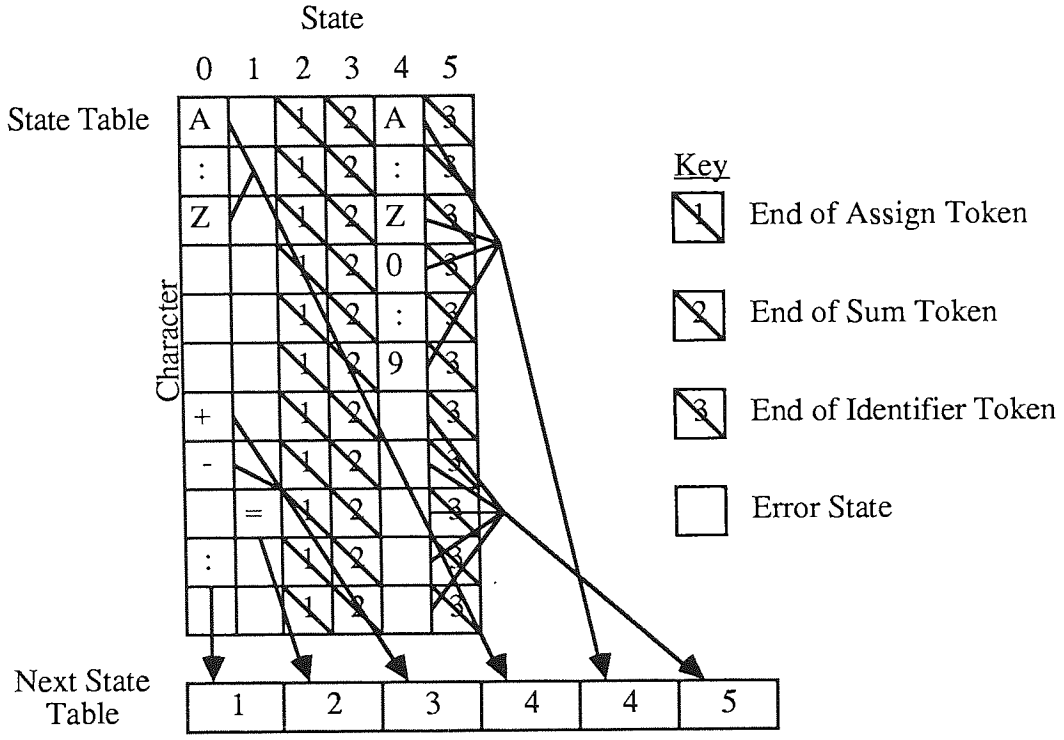


Figure 5.8

The ITT is output by the LOADER as the 'Input' control file. This control file is used to control the ACC SCANNER execution component. The current implemention of the ACC SCANNER carries out lexical analysis using a single character lookahead. Hence, the specification of lexical tokens requiring a multi-character lookahead is not (currently) supported by the ACC. The ACC SCANNER, and also the ACC PARSER and the ACC RECONSTRUCT execution components are described in the final three sections of this Chapter.

Output Tokens

The fourth CCL section in the example is the 'OUTPUT TOKENS' section. This section is used to allow the specification of ACC output terminals, as shown in Figure 5.9.

```
OUTPUT TOKENS
   INSTRUCTION  = STRUCTURE
                     {
                        operation          : INTEGER;
                        CONSTANT           : "\t";
                        operand            : INTEGER;
                        CONSTANT           : "\n";
                     };
```

Figure 5.9

The LOADER uses the input terminals defined in the previous section along with the output terminals defined in this section to produce a simple look-up table, called the Output Terminal Table (OTT). The input terminals defined in Figure 5.7 and the output terminals defined above would be used by the LOADER to produce the OTT shown in Figure 5.10.

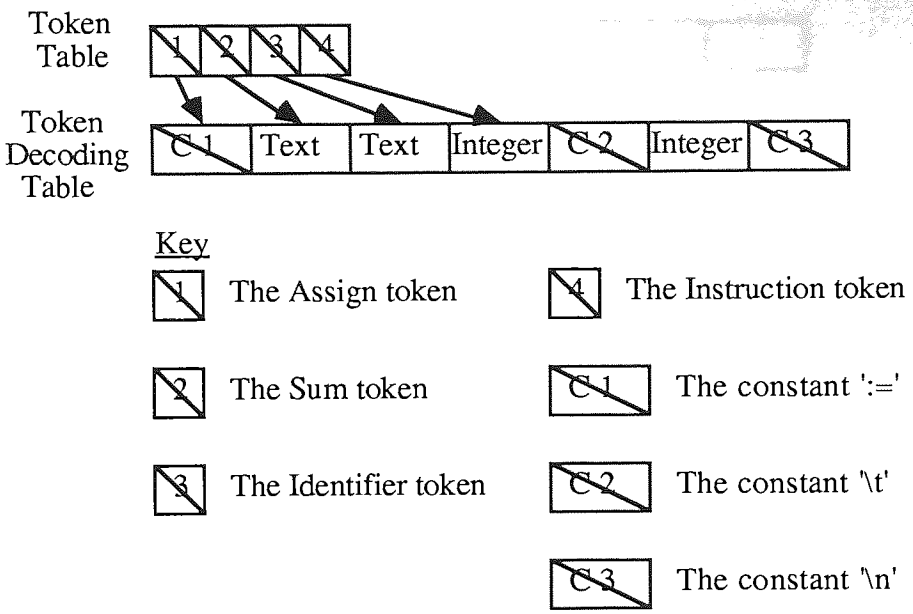


Figure 5.10

The OTT contains information about the structure of all CCL tokens. The 'Token Table' contains an entry for every ACC token defined within a CCL specification. The 'Token Decoding Table' contains one or more entries giving information about the type of each field within these tokens.

The OTT is output by the LOADER as the 'Output' control file. This control file is used by the ACC RECONSTRUCT execution component.

The LOADER also collects all constant CCL strings and enters them into a table, called the Lexical Symbol Table (LST). The CCL sections translated up to this point would be used by the LOADER to produce the LST shown in Figure 5.11.

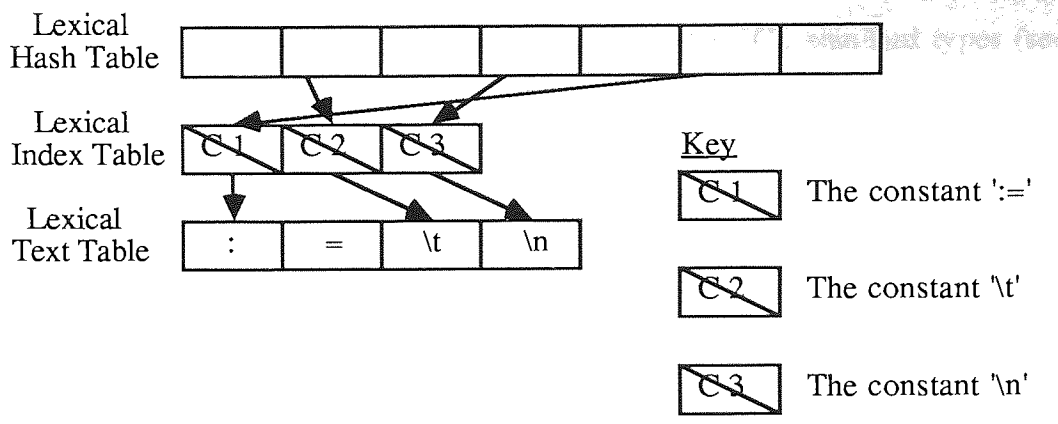


Figure 5.11

The processing of the remainder of the CCL specification might also lead to the addition of number of extra constant strings.

The LST is output by the LOADER as the 'Lexical' information file. This information file is used by the ACC SCANNER and RECONSTRUCT execution components.

Types

The fifth CCL section in the example is the 'TYPES' section. This section is used to allow the specification of CCL data structures; here, as follows.

```
TYPES
   HASH_TABLE    = TABLE[INTEGER];
```

The LOADER enters all type specifications into its internal symbol table (see Figure 5.5) for use in the following CCL section.

The current implementation of the ACC supports the CCL standard types (see Chapter 4) internally as follows.

Type	Size	Representation
CHARACTER	8 bits	Unsigned integer format.
HEXADECIMAL	32 bits	Unsigned integer format.
INTEGER	32 bits	Signed integer format.
OCTAL	32 bits	Unsigned integer format.
REAL	64 bits	Signed floating point format.
UNSIGNED	32 bits	Unsigned integer format.
SHORT FIXED	16 bits	Signed integer format.
FIXED	32 bits	Signed integer format.
LONG FIXED	32 bits	Signed integer format.
FLOAT	32 bits	Signed floating point format.
LONG FLOAT	64 bits	Signed floating point format.
TEXT	32 bits	Unique pointer to text and size.
FILE	32 bits	Unique pointer to name and size.

Figure 5.12

The ACC always performs calculations with enough precision to ensure that there is no loss of numerical accuracy, as far as the host system permits.

The ACC prevents the specification of recursive types (eg. types which are defined in terms of themselves). This was done in order to stop the specification of infinite structures and tables, and consequent infinite storage requirements.

Consider, for example, the following structure specification.

```
Struct    = STRUCTURE { Value1 : INTEGER; Value2 : Struct; };
```

This specification clearly leads to an object of type 'Struct' requiring infinite store, and is therefore not permitted within CCL specifications. However, the

specification of recursive types involving unions is somewhat different.

Consider, for example, the following union specification.

```
Type      = UNION { Subtype : Sub_Type; Pointer : Type; };
```

This specification defines a data structure which could be used within a compiler specification to specify pointer variables.

The type of a C variable 'char **x', for example, might be defined as follows.

```
Type     tx = Pointer{ Pointer{ "char" } };
```

This latter form of recursive types is perfectly meaningful, requires finite store, and is highly useful when writing compiler specifications. However, the current ACC LOADER does not currently permit the specification of such data structures (even though they are supported in the rest of the ACC system). Sadly, this oversight reduces the descriptive power of the present ACC system. Hence, it has been decided this problem requires to be fixed (with some urgency) ready for the next release of the ACC.

Variables

The sixth CCL section in the example is the 'VARIABLES' section. This section is used to specify CCL variables, as shown in Figure 5.13.

```
VARIABLES
   HASH_TABLE      identifiers =
                      [
                         "A" → 0,
                         "B" → 1
                      ],
                   instruction =
                      [
                         "Load"    → LDA,
                         "+"       → ADA,
                         "-"       → SBA
                      ];
   TEXT            name,operator;
```

Figure 5.13

The LOADER enters all CCL variable specifications into a Variable Table (VT) along with their associated initial values. The VT consists of five sub-tables which hold constants, simple variables, CCL structures, CCL unions and CCL tables respectively. The above variable specifications would be stored in the VT as follows.

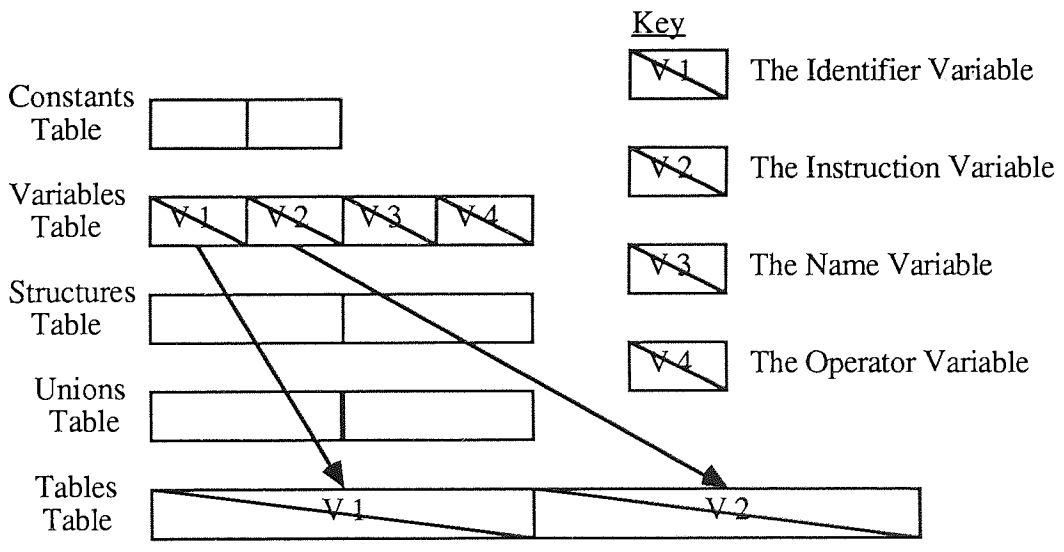


Figure 5.14

The LOADER processes CCL variable specifications and allocates space for each variable defined within the VT. Simple CCL variable specifications are allocated entries in the 'Variables Table' alone, while complex data structures are allocated space within the 'Variable Table', 'Structures Table', 'Unions Table' and 'Tables Table', as required. If a variable has an initial value this value is assigned to the variable automatically by the LOADER.

The VT is output by the LOADER as the 'Symbol' information file. This information file is used by the ACC PARSER execution component.

Rules

The final CCL section in the example is the 'RULES' section. This section is used to specify the actions and manipulations to be carried out by the ACC Execution Phase. These actions and manipulations implicitly include checking of context free syntax and static semantics (eg. type checking). The 'RULES' section from our example is given below in Figure 5.15.

```
RULES
   Assignment =
      ↑IDENTIFIER( ↑name ), ↑ASSIGN,  Expression,
         Store( ↓name );

   Expression =
      Operand( ↓"Load" ),
         { ↑SUM( ↑operator ), Operand( ↓operator ) };

   Operand( ↓operator ) =
      ↑IDENTIFIER( ↑name ),
      ↓INSTRUCTION( ↓{indirect[operator],identifiers[name]} );
```

```
Store( ↓name ) =
   ↓INSTRUCTION( ↓{STA,identifiers[name]} );
```

Figure 5.15

The LOADER translates CCL rules into a tree structure and stores this structure in a table, called the Tree Structure Table (TST). The above rules would be translated by the LOADER as shown in Figure 5.16.

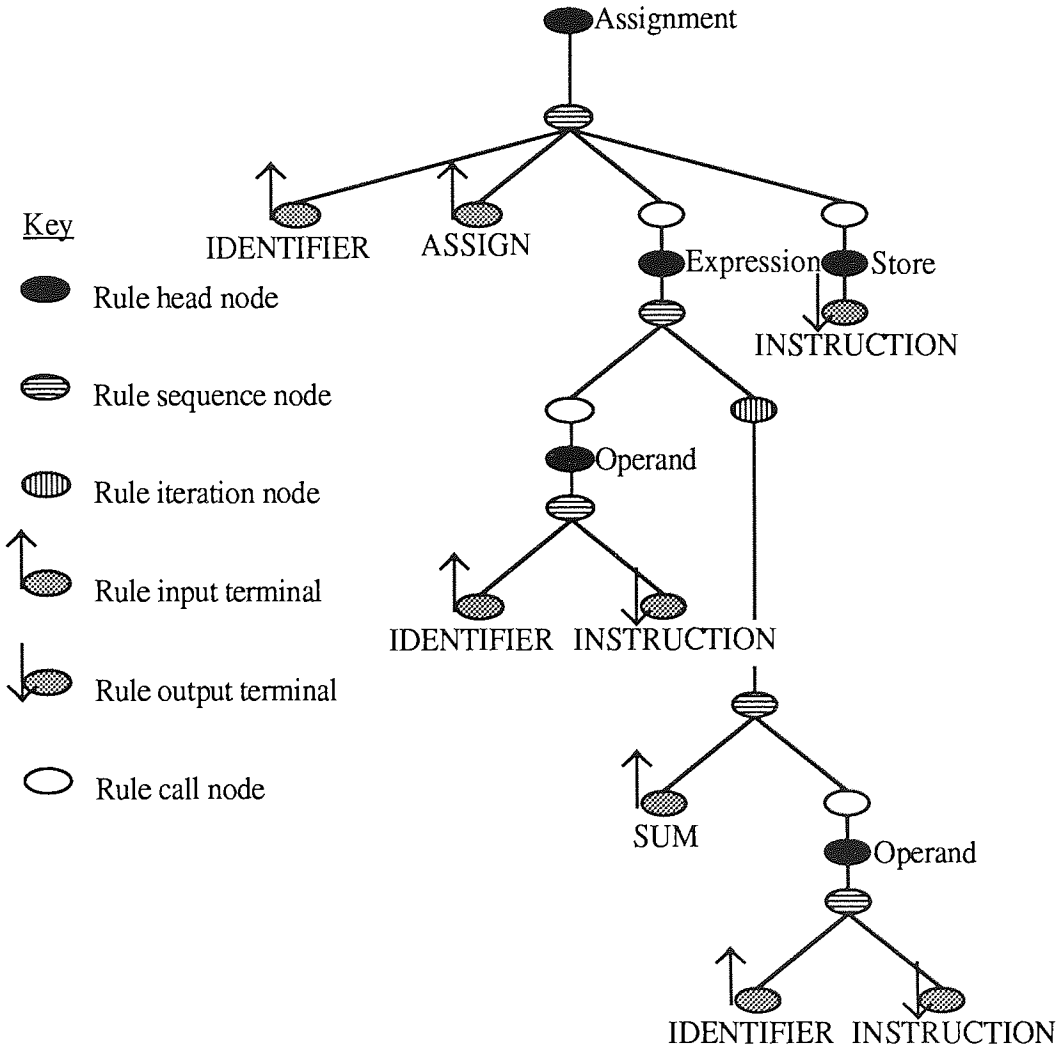


Figure 5.16

Initially, the LOADER translates each CCL rule into a small sub-tree and stores this sub-tree in the TST. When all of the rules within a CCL 'RULES' section have been processed the LOADER joins these sub-trees together to form a complete tree, as shown in the example above (rule calls are correlated with rule definitions and all parameter correspondence is fully checked). The LOADER also verifies that attributes are only referenced after they have been assigned a value (except in a rule header where forward references may be made). In effect, we ensure that the CCL rules are L-attributed [Koskimies 1983]. This (along with other minor checks) ensures that the CCL specification is suitable for top-down parsing and 'on-the-fly' attribute evaluation (discussed later).

The entry and exit points of each CCL rule have small ACC IEF code fragments associated with them. These code fragments are interpreted at execution time and deal with any attribute processing specified within the CCL rule. The structure of the instructions within these interpreted code fragments is as follows.

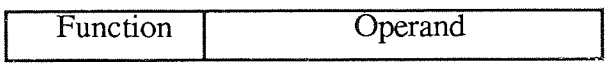


Figure 5.17

The functions available in the code fragment interpreter correspond to the CCL operators and attribute manipulations supported by the ACC. These functions include, for example: load an attribute; add two attributes; subtract two attributes; and store an attribute.

If the underlying Context Free Grammar (CFG) is LL(1), then it is straightforward that L-attributedness permits 'on-the-fly' evaluation. However, strict LL(1)-ness is in practice a rather strong constraint to impose, and by exploiting the fact that attribute evaluation will be done in parallel with parsing, it is possible to work with less severe and more practically acceptable grammar constraints. In particular, the LOADER does not prevent the specification of left recursive rules (or other rules) which might potentially cause the ACC to enter a non-terminating recursive cycle (discussed later). The reason for this is that such constructs are often useful in CCL specifications and may be effectively controlled by CCL attribute expressions. It is common practice, for example, for programming language specifications to use features (eg. a rule for assignments and a rule for procedure calls which both begin with the terminal 'identifier') which are not directly compatible with top down LL(1) style parsing. It is much more convenient if such rules can be directly transcribed into a (suitably attributed) CCL form without the need for a preliminary grammar transformation exercise. Left recursion is also useful in grammars controlling output operations such as code generation, for example, where the length of the lists to be processed can be known and supplied via inherited attributes. Thus, to forbid left recursive rules within the CCL would have been unnecessarly restrictive.

The TST is output by the LOADER as the 'Pass1' control file. This control file is used to control the first pass of the ACC PARSER execution component. The control files generated by subsequent CCL 'RULES' sections are named 'Pass2', 'Pass3', ..., 'Pass(n)' respectively. These control files are used to control subsequent passes of the ACC PARSER execution component.

The Execution Phase

The ACC Execution Phase loads translated CCL specifications in ACC IEF format and executes them. The ACC Execution Phase consists of three program components, called SCANNER, PARSER and RECONSTRUCT. The execution of a small CCL specification, such as the one discussed above, may typically be organised as shown in Figure 3.4, repeated here for convenience (Figure 5.18).

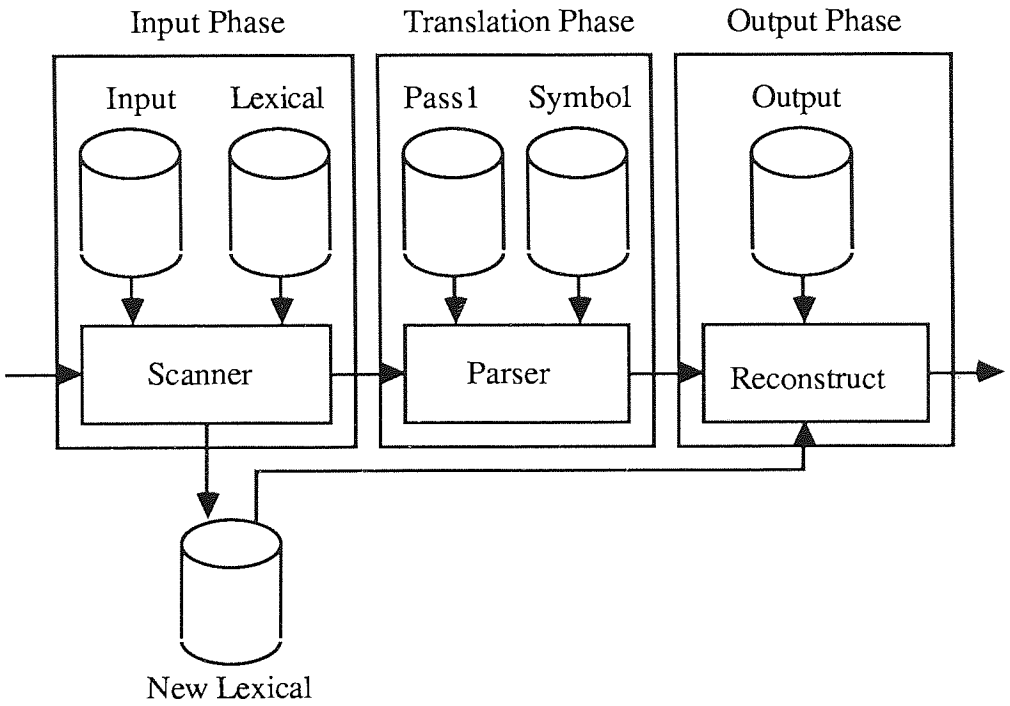


Figure 5.18

The component programs of the ACC Execution Phase may be executed sequentially on uniprocessing operating systems, such as MS/DOS, or in parallel on multiprocessing operating systems, such as UNIX. The operation of each of

these programs is discussed individually in the following sections.

Scanner

The SCANNER program reads source code via the standard C input channel, translates it into ACC tokens and then outputs these tokens via the standard C output channel.

The outline execution structure of the SCANNER program is as follows.

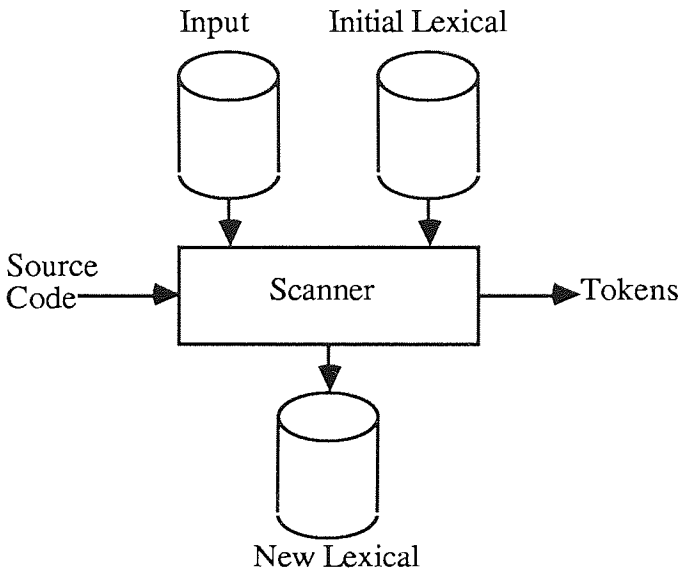


Figure 5.19

Initially, the SCANNER loads the ITT and the LST contained within the 'Input' control file and the 'Initial Lexical' information file respectively. The SCANNER then reads source code via the standard C input channel and translates this source code into ACC tokens using the ITT (see Figure 5.8). The ACC tokens generated

by the SCANNER are output in the following format via the standard C output channel.

Token Number	Line Number	Character Position	Token Size	Token Value(s)

Figure 5.20

The 'Token Number' field identifies the token's type. The 'Line Number' and 'Character Position' fields contain the line number in the source file where the token was discovered and the approximate character position within the source line. These fields are used by the PARSER program when generating warning or error messages. If the token has a number of associated values the 'Token Size' field contains a count of the following 'Token Value' fields, otherwise this field is zero. The structure of each 'Token Value' field is as follows.

Value Type	Value

Figure 5.21

The 'Value Type' field contains an flag indicating the type of value being stored within the structure. The 'Value' field contains the actual value. All numeric values are converted from character format into the corresponding local binary format by the SCANNER. All textual values are entered into the LST (see Figure 5.11) and a pointer to the value is passed instead of the actual text. The updated contents of the LST may be optionally stored and used in subsequent runs of the SCANNER or RECONSTRUCT programs.

All of the component programs of the ACC Execution Phase use the above token structure when communicating with each other.

The current implementation of the SCANNER uses a single character look-ahead when processing lexical symbols (as stated earlier). Although this is sufficient for the specification of a large class of the lexical symbols used in modern High Level Programming Languages (HLPLs) there are cases where it is not.

Consider the following Pascal declaration.

```
Typename = ARRAY[ 1..20 ] OF INTEGER;
```

It is not possible when using a single character lookahead to decide whether the text '1.' is an integer token '1' (followed by the token '..') or a real number token (such as '1.0'). The flexibility of the ACC sometimes allows such problems to be overcome, for example the above problem can be solved by defining the a structured token as shown in Figure 5.22.

```
Real            = STRUCTURE
                    {
                       Integer          : INTEGER;
                       Point            : "." | "..";
                       Fraction         : INTEGER;
                    };
```

Figure 5.22

This CCL token specification would match both real numbers (such as '1.2') and array specifiers (such as '1..20').

However, this solution is not really satisfactory as it increases the complexity of CCL specifications. An improved solution would be to allow the SCANNER to have a multi-character character look-ahead. This is currently under consideration and might be implemented in later versions of the ACC.

Parser

The PARSER program reads tokens from the SCANNER (or a previous PARSER) via the standard C input channel, manipulates these tokens, and outputs a token stream (computed from the execution of a 'RULES' section) via the standard C output channel.

The outline execution structure of the PARSER program is as follows.

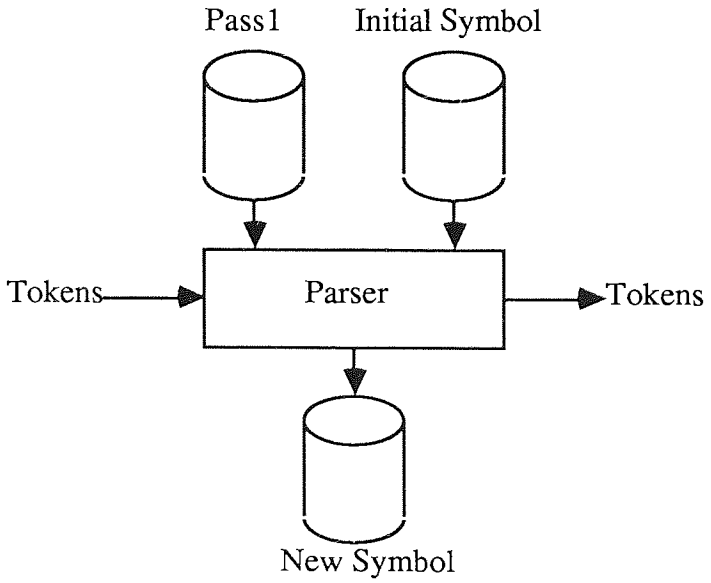


Figure 5.23

Initially, the PARSER program loads the TST and the VT contained in the 'Pass1' control file and the 'Initial Symbol' information file respectively. The PARSER then executes the syntax tree held within the TST (see Figure 5.16) using a standard top down, left to right parsing algorithm. The method used is essentially LL(1) parsing [Aho 1985] (as stated earlier) except that attribute values may be used to control the parse at points where the underlying grammar is not LL(1) [Watt 1980]. The input and output of ACC tokens is performed dynamically by the PARSER as terminals are encountered during the processing of a CCL specification (via the standard C input and output channels). The evaluation of Attribute Expressions (AEs) (which are associated with the entry and exit points of individual CCL rules) is also carried out dynamically. These AEs are evaluated by interpreting the code fragments generated by the ACC LOADER on a small stack. The interpreter uses the VT (see Figure 5.14) to access to CCL constants, CCL variables, and for temporary storage space. When the PARSER terminates the updated contents of the VT may be optionally stored for use by subsequent runs of the program.

The top down, left to right parsing algorithm used by the PARSER was selected because of its run time and storage efficiency coupled with the ability to allow the use of a straightforward left to right attribute evaluation system. The simplicty of the selected parsing and evaluation system is important as CCL rules sometimes have side effects, such as the generation of code. A less straightforward scheme (such as might have been needed by a bottom up syntax analyser) would have made the execution of CCL specifications harder to follow and therefore more

difficult to write and understand.

The code fragment interpreter within the PARSER allows AEs containing the CCL comparison, alternative, sequence and selection operators (see Chapter 4) to control the execution of a CCL specification (as stated earlier). This is achieved by causing individual rules within a CCL specification to fail if the conditions imposed by these operators are not met at execution time. The main motivation for this feature was the belief that this would simplify certain aspects of compiler specifications (see Chapters 4 and 6) as well as extending the descriptive power of the CCL (see Chapter 2 and [Watt 1980]).

Consider, for example, the following Context Free Syntax (CFS).

```
Statement       = ↑IDENTIFIER, ( Assign | Parameters);
Assign          = ↑ASSIGN expression;
Parameters      = [ Parameter, { ↑COMMA, Parameter } ];
```

Figure 5.24

When executing the above CFS with a LL(1) based parser it cannot be decided whether to apply the rule 'Assign' or 'Parameters' after the terminal 'IDENTIFIER' has been accepted in the rule 'Statement'. This kind of difficulty can be overcome in the ACC by directing the ACC PARSER (with AEs) to the rule which should be applied, as shown in Figure 5.25.

```
Statement( ↓env ) =
   ↑IDENTIFIER( ↑name ),
      (
         Assign( ↓env[ name ].identifer ) |
         Parameters( ↓env[ name ].function )
      );

Assign( ↓symbol ) =
   ↑ASSIGN expression;

Parameters( ↓symbol ) =
   [ Parameter, { ↑COMMA, Parameter } ];
```

Figure 5.25

Thus, if the attribute 'name' is defined as an 'identifier' (in the attribute 'env') the rule 'Assign' is applied. Alternatively, if the attribute 'name' is defined as a 'function' (in the attribute 'env') the rule 'Parameters' is applied. In all other cases the rule 'Statement' will fail and return an error condition to the calling rule. However, left to right attribute evaluation can sometimes be restrictive.

Consider, for example, the following Pascal declaration.

```
A,B,C,D,E        : INTEGER;
```

A parser using a left to right attribute evaluation cannot collect any information about the type of the variables 'A', 'B', 'C', 'D' and 'E' (above) until the symbol 'INTEGER' was encountered. Thus, it would usually be necessary to collect information about such variables and pass it forward through the syntax tree until the type of the variables was encountered. Only then could complete entries for these variables be made in a symbol table of a compiler.

A parser using an alternative attribute evaluation scheme (such as an alternating left to right, right to left attribute evaluator [Raiha 1983]) might seem more suitable for compiling the above statement. Such an evaluator would, for example, allow the type information in the previous example to be passed backwards to the variables instead of vice versa. However, it is very difficult to implement such evaluation schemes without the construction of an attributed syntax tree. This imposes a considerable overhead at run time as it is usually necessary to represent all (or at least a significant part) of such attributed trees in a computer's primary memory, for reasons of efficiency (see Chapter 2). In addition to this, the use of this class of attribute evaluation techniques often leads to a number of further difficulties. It is possible, for example, to define attributes with circular dependencies (where attribute A is defined in terms of attribute B and vice versa). Such Attribute Grammars (AGs) are generally considered pathological and are therefore excluded from use in practical systems. A number of tests for circularity are known [Meijer 1982] but most increase exponentially in complexity as the grammar size increases. It was therefore decided that top-down, left to right parsing with 'on-the-fly' attribute evaluation would be the most suitable implementation scheme for a tool like the ACC, which is essentially aimed at practical compiler construction.

Reconstruct

The RECONSTRUCT program reads tokens from the PARSER (or the SCANNER) via the standard C input channel, translates these tokens into a binary

or textual object format, and then outputs this object format via the standard C output channel.

The outline execution structure of the RECONSTRUCT program is as follows.

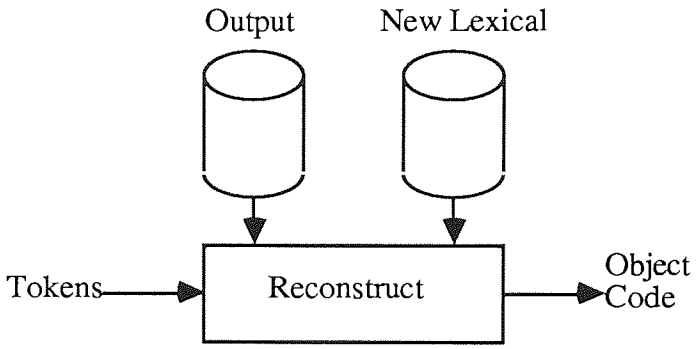


Figure 5.26

Initially, the RECONSTRUCT program loads the OTT and the updated LST contained in the 'Output' control file and the 'New Lexical' information file respectively. The RECONSTRUCT program then reads tokens via the standard C input channel and translates these tokens back into a textual format using the OTT (see Figure 5.10). All numeric values are translated directly into their target character representation and output. The textual values are extracted from the updated LST before being finally output as ASCII text.

Summary

In this chapter we have discussed the implementation of the ACC. It was seen (in outline) how the ACC LOADER translates CCL specifications into an executable ACC IEF. It was also seen how subsequent execution involves: analysing source

text in accordance with CCL input token specifications; manipulations determined by parsing and attribute evaluation in accordance with CCL rules sections; and the production of output text as determined by CCL output specifications. In addition to this, it was seen how parsing and attribute evaluation was achieved via a single-pass top-down 'on-the-fly' mechanism, rather than by building an explicit syntax tree. This approach had certain limitations regarding the form of attributes which can be handled, but this was more than offset by the resource economy of our algorithm compared with tree building approaches. The constraints appear in practice to be acceptable.

The practical application of attributes to the control of top-down parsing, and the consequent benefit of not being restricted to LL(1) grammars, is an area that has been little studied. The majority of earlier work has been concerned with store-hungry bottom-up multi-pass methods. The general benefits of top-down parsing are well-established. However, one important additional benefit (with regard to AGs) is that top-down parsing allows an AG to be viewed as a sequential program which is executing during parsing. This makes it much easier for a compiler writer to understand the processing of AGs by such parsers and to determine how and where semantic processing should be included.

In the next chapter we shall examine a larger CCL specification of a small but complete compilation system, in order to further evaluate the practical value of the ACC. We shall also see how this larger CCL specification could be executed sequentially or in parallel under the UNIX operating system.

Chapter 6

A Demonstration of the Aston Compiler Constructor

Introduction

We have examined many of the theoretical and implementation aspects of the Aston Compiler Constructor (ACC) in the previous chapters. In this chapter we shall attempt to assess the practical value of the ACC by examining a more extensive Compiler Construction Language (CCL) specification. This specification defines a compiler of a small programming language, called The Small Language (TSL).

The Small Language

The programming language TSL is modelled on a restricted subset of Pascal. The Context Free Syntax (CFS) of TSL is defined in CCL as follows.

```
TITLE Example

INPUT TOKENS
   PROGRAM      = CONSTANT "PROGRAM";
   VAR          = CONSTANT "VAR";
   BEGIN        = CONSTANT "BEGIN";
   IF           = CONSTANT "IF";
   THEN         = CONSTANT "THEN";
   ELSE         = CONSTANT "ELSE";
   FI           = CONSTANT "FI";
   ENDS         = CONSTANT "END";

   ASSIGN       = CONSTANT ":=";
   SUM          = "+" | "-";
   TERM         = "*" | "/";
   COMPARISON   = "=" | "≠" | "<" | "≤" | ">" | "≥";
   COMMA        = CONSTANT ",";
   SEMI_COLON   = CONSTANT ";";
   DOT          = CONSTANT ".";

   IDENTIFIER   = ('A'→'Z'), {'A'→'Z' | '0'→'9'};
   VALUE        = INTEGER;

   VOID         = " " | "\t" | "\n";
```

```
RULES
   Compiler      = Program, Var, Statement, End;

   Program       = ↑PROGRAM, ↑IDENTIFIER;

   Var           = ↑VAR, Variables, ↑SEMI_COLON;
   Variables     = ↑IDENTIFIER, { ↑COMMA, ↑IDENTIFIER };

   Statement     = Compound | Assignment | If;

   Compound      = ↑BEGIN, Statement,
                      { ↑SEMI_COLON, Statement }, ↑ENDS;

   Assignment    = ↑IDENTIFIER, ↑ASSIGN, Expression;
   Expression    = Term, { ↑SUM, Term };
   Term          = Terminal, { ↑TERM, Terminal };
   Terminal      = ↑IDENTIFIER | ↑VALUE;

   If            = ↑IF, Boolean, ↑THEN,  Statement,
                      [ ↑ELSE, Statement ], ↑FI;
   Boolean       = Expression, ↑COMPARISON, Expression;

   End           = ↑DOT;
END
```

Figure 6.1

As is shown, the CCL CFS specification above uses a modified superset of standard BNF [Naur 1960]. As noted earlier, the main modifications and additions to standard BNF are the seperation of input terminals from the main syntax specification and the inclusion of Regular Expression (RE) operators for optional clauses (eg. '[' and ']') and iteration cluases (eg. '{' and '}'). The TSL CFS specification above carries the same content as the Lex and Yacc specifications given in Figures 2.2 and 2.3 taken together. The CCL specification is only a little more compact, but has the advantages of greater clarity and readability due to the use of a notation which is more powerful (eg. the operators {} etc) and also more consistent.

A Target Machine Architecture

The Target Machine Architecture (TMA) to be considered in our assessment is based on a hypothetical stack based computer. This computer has a large byte addressed main memory and a single general purpose data stack, which is used in the evaluation of numeric expressions. The TMA supports a range of zero and one address instructions all of which operate on 16 bit integer values. All instruction addresses are assumed to be relative to the base of the machine's data stack, except for the addresses of jump instructions which are assumed to be relative to the start of the current machine code program.

The structure of machine code instructions in the TMA is as follows.

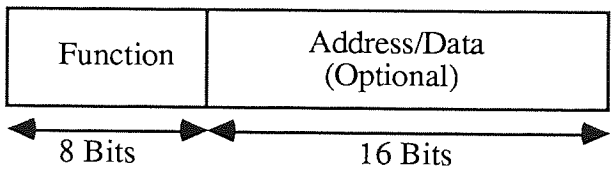


Figure 6.2

The machine code instructions available are as follows.

Instruction	Op code	Size	Meaning
LD	0x00	24 Bits	Overwrite Top Of Stack (TOS) with contents of following address.
ST	0x01	24 Bits	Copy TOS to following address.
LDA	0x10	24 Bits	Load TOS from following address.
ADA	0x11	24 Bits	Add address to TOS.
SBA	0x12	24 Bits	Subtract address from TOS.
MLA	0x13	24 Bits	Multiply TOS by address.
DVA	0x14	24 Bits	Divide TOS by address.

STA	0x15	24 Bits	Store TOS at following address.
LDC	0x20	24 Bits	Load constant to TOS.
ADC	0x21	24 Bits	Add constant to TOS.
SBC	0x22	24 Bits	Subtract constant from TOS.
MLC	0x23	24 Bits	Multiply TOS by constant.
DVC	0x24	24 Bits	Divide TOS by constant.
ASF	0x30	24 Bits	Adjust stack front.
ADD	0x40	8 Bits	Add the top two stack items.
SUB	0x41	8 Bits	Subtract the top two stack items.
MUL	0x42	8 Bits	Multiply the top two stack items.
DIV	0x43	8 Bits	Divide the top two stack items.
HLT	0x50	8 Bits	Halt execution.
JMP	0x60	24 Bits	Jump to following address.
JEQ	0x61	24 Bits	Jump to address if TOS zero.
JNE	0x62	24 Bits	Jump to address if TOS not zero.
JLT	0x63	24 Bits	Jump to address if TOS < zero.
JLE	0x64	24 Bits	Jump to address if TOS ≤ zero.
JGR	0x65	24 Bits	Jump to address if TOS > zero.
JGE	0x66	24 Bits	Jump to address if TOS ≥ zero.

Figure 6.3

The execution of TMA programs begins with the first machine code instruction of a program and terminates with the execution of a 'HLT' instruction.

A Compiler Specification

We shall now examine the implementation of a compiler capable of translating TSL source programs into TMA machine code programs. We shall examine this compiler in a number of segments, corresponding to CCL sections within the specification (see Chapter 4 for CCL sections). A complete listing of the CCL specification is given in Appendix 4.

Title

The first CCL section in the TSL compiler specification is the 'TITLE' section. This section is required in order to name the specification, as follows.

```
TITLE Example
```

The title given to the specification is used to prefix in the file names of the ACC IEF files produced by the ACC LOADER, hence enhancing the security and flexibility of the system.

Input Tokens

The second CCL section in the TSL compiler specification is the 'INPUT TOKENS' section. This section is required to define the mapping of the terminal symbols of the TSL to appropriate input tokens, as follows.

```
INPUT TOKENS
   PROGRAM      = CONSTANT "PROGRAM";
   VAR          = CONSTANT "VAR";
   BEGIN        = CONSTANT "BEGIN";
   IF           = CONSTANT "IF";
   THEN         = CONSTANT "THEN";
   ELSE         = CONSTANT "ELSE";
   FI           = CONSTANT "FI";
   ENDS         = CONSTANT "END";

   ASSIGN       = CONSTANT ":=";
   SUM          = "+" | "-";
   TERM         = "*" | "/";
   COMPARISON   = "=" | "≠" | "<" | "≤" | ">" | "≥";
   COMMA        = CONSTANT ",";
   SEMI_COLON   = CONSTANT ";";
   DOT          = CONSTANT ".";
```

```
IDENTIFIER    = ('A'→'Z'), {'A'→'Z' | '0'→'9'};
VALUE         = INTEGER;

VOID          = " " | "\t" | "\n";
```

Figure 6.4

The above example demonstrates, in principle, how the terminal symbols of modern programming languages, such as Pascal and C, might be mappped to appropriate ACC input tokens. However, input symbols for some older programming languages, such as FORTRAN or COBOL, cannot be specified in CCL as it is not possible for the ACC to divide the source text of these programming langauges into separate symbols. Lex [Lesk 1976] is able to deal with such features; however since most modern programming languages are (for good reasons) designed to avoid such difficulties, it is not worthwhile (in a new system) to include the substantial extra complications required to handle the more obscure lexical features of these languages. It would be necessary for a compiler writer to construct a source code preprocessor in order to translate the source text of such languages into a form suitable for the ACC SCANNER.

An interesting feature in the above example is the grouping of certain operators, such as the operators '=', '≠', '<', '≤', '>', '≥' in the 'COMPARISON' token. It will be seen later that the processing of all of these operators by the ACC is essentially identical, eventhough each individual operator gives rise to the generation of a different machine code jump instruction. It is also apparent that the input token specifications above are no more than the lexical part of the context free specification in Figure 6.1.

Output Tokens

The third CCL section in the TSL compiler specification is the 'OUTPUT TOKENS' section. This section is required in order to map the compiler's output terminals to appropriate output tokens, as shown in Figure 6.5.

```
OUTPUT TOKENS
   FILE_NAME     = FILE;
   CODE          = STRUCTURE
                   {
                      function          : TEXT;
                      CONSTANT          : "\t";
                      operand           : INTEGER;
                      CONSTANT          : "\n";
                   };

   FINAL1        = CHARACTER;
   FINAL2        = SHORT FIXED;
```

Figure 6.5

The output tokens specified above may be grouped into two pairs. The first pair of tokens (ie. FILE_NAME and CODE) define the interface between the front end of the compiler and the code optimisation and linking phases. The second pair of tokens (ie. FINAL1 and FINAL2) define the binary machine code format and are needed for the output of machine code instructions. The use of these terminals is discussed more fully in a later section of this chapter.

The flexibility of the ACC's interfacing capabilities are demonstrated in Figures 6.4 and 6.5 above. It can be seen that the ACC permits a range of input and output terminals to be specified (see Chapter 4) in a way that is both clear and concise. This is possible due to the large selection of data types available in the

ACC and the ability to structure input and output terminals. This generality makes it possible to interface the ACC to a wide range of other software components, as well as providing a convenient method of passing information between different phases of ACC-built compilers.

Types

The fourth CCL section in the TSL compiler specification is the 'TYPES' section. This section is used to specify compiler data structures, as follows.

```
TYPES
   HASH_TABLE   = TABLE[INTEGER];
   INSTRUCTION  = TABLE[TEXT];
   OPTIMISATION = TABLE[INSTRUCTION];
```

Figure 6.6

The ACC allows the specification of a considerable range of data structures (see Chapter 4). However, the TSL compiler requires only a number of simple data structures, because of the elementary nature of the source language. The purposes of these data structures are clarified and discussed in the following section.

Variables

The fifth CCL section in the TSL compiler specification is the 'VARIABLES' section, which naturally is used to specify compile-time variables. These variables are used for a number of distinct purposes, for example: to hold the values associated with particular tokens; to map source language operators to symbolic

instructions; to contain symbolic instruction optimisation tables; and to contain information to map symbolic instructions to binary function codes. The variables needed in this compiler specification are as shown in Figure 6.7.

```
VARIABLES
        /*   Simple variables                                          */
    CODE            code,code1,code2;
    INTEGER         csize=0,dsize=0,label=0,new_label,operand,value;
    TEXT            function,name,operator;

        /*   The symbol table and forward jump resolution table */
    HASH_TABLE      identifier = [],reference = [];

        /*   The operator to function mapping table             */
    INSTRUCTION     instruction =
                        [
                            "+"  → "ADD",        "-"  → "SUB",
                            "*"  → "MUL",        "/"  → "DIV",

                            "="  → "JNE",        "≠"  → "JEQ",
                            "<"  → "JGE",        "≤"  → "JGR",
                            ">"  → "JLE",        "≥"  → "JLT"
                        ];

        /*   The function size look up table                   */
    HASH_TABLE      size =
                        [
                            "LD"  → 3,           "ST"  → 3,

                            "LDA" → 3,           "ADA" → 3,
                            "SBA" → 3,           "MLA" → 3,
                            "DVA" → 3,           "STA" → 3,

                            "LDC" → 3,           "ADC" → 3,
                            "SBC" → 3,           "MLC" → 3,
                            "DVC" → 3,

                            "ASF" → 3,

                            "ADD" → 1,           "SUB" → 1,
                            "MUL" → 1,           "DIV" → 1,

                            "HLT" → 1,

                            "JMP" → 3,           "JEQ" → 3,
```

```
                    "JNE" → 3,          "JLT" → 3,
                    "JLE" → 3,          "JGR" → 3,
                    "JGE" → 3,

                    "LABEL" → 0
                ];

    /*   The peephole optimisation tables                    */
    /*   (for redundant load removal and constant folding)   */
OPTIMISATION   fold1 =
                [
                    "STA" →
                        [ "LDA" → "ST" ]
                ];
OPTIMISATION   fold2 =
                [
                    "LDA" →
                        [
                            "ADD" → "ADA",
                            "SUB" → "SBA",
                            "MUL" → "MLA",
                            "DIV" → "DVA"
                        ],
                    "LDC" →
                        [
                            "ADD" → "ADC",
                            "SUB" → "SBC",
                            "MUL" → "MLC",
                            "DIV" → "DVC"
                        ]
                ];
OPTIMISATION   fold3 =
                [
                    "LDC" →
                        [
                            "ADC" → "LDC",
                            "SBC" → "LDC"
                        ],
                    "ADC" →
                        [
                            "ADC" → "ADC",
                            "SBC" → "ADC"
                        ],
                    "SBC" →
                        [
                            "ADC" → "SBC",
                            "SBC" → "SBC"
                        ]
```

```
                    ];
HASH_TABLE      fold4 =
                    [
                        "ADC" → 1,
                        "SBC" → -1
                    ];

    /*   Symbolic function to machine code mapping tables   */
HASH_TABLE      class1 =
                    [
                        "LD"  → 0x0,        "ST"  → 0x1,

                        "LDA" → 0x10,       "ADA" → 0x11,
                        "SBA" → 0x12,       "MLA" → 0x13,
                        "DVA" → 0x14,       "STA" → 0x15,

                        "LDC" → 0x20,       "ADC" → 0x21,
                        "SBC" → 0x22,       "MLC" → 0x23,
                        "DVC" → 0x24,

                        "ASF" → 0x30
                    ];
HASH_TABLE      class2 =
                    [
                        "ADD" → 0x40,       "SUB" → 0x41,
                        "MUL" → 0x42,       "DIV" → 0x43,

                        "HLT" → 0x50
                    ];
HASH_TABLE      class3 =
                    [
                        "JMP" → 0x60,       "JEQ" → 0x61,
                        "JNE" → 0x62,       "JLT" → 0x63,
                        "JLE" → 0x64,       "JGR" → 0x65,
                        "JGE" → 0x66
                    ];
```

Figure 6.7

It can be seen that the CCL supports, in a natural fashion, the specification of tables. The ability to tabularise compiler specifications (as demonstrated above and also more extensively later in connection with the 'RULES' section) is an important advantage. Such specifications are in practice usually clear, concise,

easy to define, and modify.

The simple variables specified above are used as follows.

Variable	Function
code	This variable is used to hold machine instructions being output by out example compiler's front end.
code1 & 2	These variables are used to hold the current machine instructions being optimised by the peephole optimiser.
csize	This global variable contains the size of the current code segment.
dsize	This global variable contains a count of the data variables defined.
label	This global variable contains a the number of the next available jump label.
new_label	This variable contains the number of the jump label currently in use.
value	This variable contains the value of any constants within a numeric expressions.
function	This variable contains the machine code function mnemonic before it is finally translated into binary and output.
name	This variable contains the name of the current variable identifier being processed within a numeric expression.
operator	This variable contains the current operator being processed a within numeric expression.

Figure 6.8

The tables are used as shown in Figure 6.9.

Tables	Function
identifier	This table contains the addresses of all the variables defined within the source code being processed.
reference	This table contains the addresses of the labels within the machine code generated by our example compiler.
instruction	This table is used to translate programming language operators into target machine instructions.
size	This table gives the size of each machine instruction to allow an accurate label look-up table to be kept in the variable 'reference'.
fold1,2,3 & 4	These tables contain all of the instruction reductions recognised by the peephole optimiser.
class1,2 & 3	These tables contain the numeric machine code equivalent for each textual machine instruction used within the compiler.

Figure 6.9

Rules

The final three CCL sections in the TSL compiler specification are 'RULES' sections. These sections specify the Context Free Syntax (CFS) and static semantics of the TSL, and define the translation of TSL source text to TMA machine code. The first 'RULES' section (the 'Translator') specifies the CFS and static semantics of the TSL and specifies the translation of TSL source code to simple Mnemonic Instructions (MIs). The second 'RULES' section (the 'Optimiser') specifies a local optimiser for the MIs output by the Translator. The final 'RULES' section (the 'Loader') translates the MIs output by either the Translator or the Optimiser into pure binary machine code for the TMA.

The Translator

The Translator is the largest 'RULES' section in the TSL compiler specification and is therefore discussed below in several parts. The specification of the Translator begins as shown in Figure 6.10.

```
RULES
   Compiler =
      Program, Var, Statement, End;

   Program =
      ↑PROGRAM, ↑IDENTIFIER( ↑name ),
         ↓FILE_NAME( ↓name );

   End =
      ↑DOT,
         ↓CODE( ↓{"HLT",0} );
```

Figure 6.10

The rule 'Compiler' specifies the overall structure of TSL programs and references the rules 'Program', 'Var', 'Statement' and 'End'. The rule 'Program' defines the name part of the a source program (the ↓PROGRAM and ↓IDENTIFIER(↓name) part) and (using the ↓FILE_NAME(↓name) part) that the name identifier is to be output to the next phase of the compiler. The rules 'Var' and 'Statement' will process TSL variable specifications and statements and are discussed later. The rule 'End' will accept the input terminal 'DOT' (which appears at the end of a TSL program) and generate a 'HLT' instruction.

The rule 'Var' references a number of further rules, as shown in Figure 6.11.

```
Var =
   ↑VAR, Variables, ↑SEMI_COLON,
      Code( ↓{"ASF",dsize},↑csize );

Variables =
   Variable, { ↑COMMA, Variable };

Variable =
   <<COMMA,SEMI_COLON,"Invalid variable identifier">>
      Variable_Name( ↑dsize,↑<identifier> );

Variable_Name( ↑dsize+1,↑[name → dsize] ) =
   ↑IDENTIFIER( ↑name );

Code( ↓code,↑csize+size[code.function] ) =
   ↓CODE( ↓code );
```

Figure 6.11

The rule 'Var' references the rule 'Variables', which in turn references further rules ('Variable' and 'Variable_Name') related to the processing of TSL variable declarations. The variable names declared in the source text will be stored in the global table 'identifier'. The space required by these variables will be calculated and stored in the global variable 'dsize'. Following this, the TSL compiler will generate an 'ASF' (Adjust Stack Front) instruction to allocate storage on the run-time data stack for source program variables.

The variable declarations allowed in the TSL are very limited and require only a simple symbol table structure. However, a compiler for a commercial programming language, such as Pascal or C, would require a much larger and more complex symbol table structure. This can be dealt with quite easily by the ACC but for the sake of reasonable brevity cannot be demonstrated here. An example of a somewhat more realistic symbol table structure was given earlier in Figure 4.24 in Chapter 4.

The rule 'Statement' also references a number of further rules ('Compound, 'Assignment' and 'If') which deal with the translation of the corresponding TSL statements. 'Statement' and 'Compound' are as follows.

```
Statement =
   <<SEMI_COLON,DOT,"Invalid statement">>
      ( Compound | Assignment | If );

Compound =
   ↑BEGIN, Statement, { ↑SEMI_COLON, Statement }, ↑ENDS;
```

Figure 6.12

The translation of compound statements is straightforward and does not require the generation of any MIs, as shown above. However, the translation of assignment statements and if-statements is more complex. The rule 'Assignment' again references a number of further rules, as follows.

```
Assignment =
   ↑IDENTIFIER( ↑name ),
      <<SEMI_COLON,DOT,"Illegal assignment">>
         ( ↑ASSIGN,  Expression ),
            Code( ↓{"STA",identifier[name]},↑csize );

Expression =
   Term, { ↑SUM( ↑operator ), Term,
      Code( ↓{instruction[operator],0},↑csize ) };

Term =
   Terminal, { ↑TERM( ↑operator ), Terminal,
      Code( ↓{instruction[operator],0},↑csize ) };
```

```
Terminal =
   ( ↑IDENTIFIER( IN name ),
        Code( ↓{"LDA",identifier[name]},↑csize ) )
     |
   ( ↑VALUE( ↑value ),
        Code( ↓{"LDC",value},↑csize ) );
```

Figure 6.13

The rule 'Assignment' references the rule 'Expression', which deals with TSL numeric expressions. This rule in turn references the rule 'Term' which references the rule 'Terminal'. These rules deal with the translation of various sections of TSL numeric expressions into MIs. Again, a greater range of expressions would be permitted in a commercial programming language than the simple numeric expressions dealt with here. However, in principal the translation of such expressions could be achieved with by expanding the rules given here. An experiment carried out by the author indicated that the translation of expressions in the programming language Pascal could be defined in around 30 rules, varying somewhat according to the details of the target machine architecture.

Finally, the rule 'If' references a number of further rules, as follows.

```
If =
   ↑IF, <<SEMI_COLON,DOT,"Illegal if statement">>
      ( Boolean( ↑new_label ), ↑THEN, Statement,
         [ ↑ELSE, Else_Statement( ↓↑new_label ) ],
            Assign_Label( ↓new_label,↑new_label,↑<reference> ) ),
   ↑FI;

Boolean( ↑new_label ) =
   Expression, ↑COMPARISON( ↑operator ), Expression,
      Allocate_Label( ↑new_label,↑label ),
      Code( ↓{"SUB",0},↑csize ),
      Code( ↓{instruction[operator],new_label},↑csize );
```

```
Allocate_Label( ↑label,↑label+2 );

Else_Statement( ↓↑new_label ) =
   Code( ↓{"JMP",new_label+1},↑csize ),
   Assign_Label( ↓new_label,↑new_label,↑<reference> ),
      Statement;

Assign_Label( ↓new_label,↑new_label+1,↑[new_label → csize] ) =
   ↓CODE( ↓{"LABEL",new_label} );
```

Figure 6.14

The rule 'If' references the rules 'Boolean', 'Statement' (as defined earlier), 'Else_Statement' and 'Assign_Label'. The rule 'Boolean' references the rules 'Expression' (as defined earlier) and 'Allocate_Label'; also, it will insert a 'SUB' (subtract) instruction into the generated code (to compare the two expressions defined) followed by an appropriate conditional jump. The rule 'Allocate_Label' is used to provide a sequence of unique jump labels. The rule 'Else_Statement' will insert a 'JMP' (jump) instruction into the code and references the rules 'Assign_Label' and 'Statement'. Finally, the rule 'Assign_Label' will insert the appropriate jump label into the code.

The code generation rules in the first section of TSL compiler above are clear and simple, but are also very naive (causing the compiler to produce poor quality code). However, this was intentional in order to demonstate that ACC-built compilers (with an optional optimisation phase) can be used to produce low quality code (quickly) for program development purposes and better quality code (more slowly) for production programs (thus highlighting the flexibility of the ACC).

The Optimiser

The second 'RULES' section in the TSL compiler specification defines a peephole optimiser designed to optimise the MIs generated by the previous section. This optimiser may be optionally included in the execution of the final ACC-built compiler. The specification of the Optimiser is as shown in Figure 6.15.

```
RULES
   Optimiser =
      Title, Optimisation;

   Title =
      ↑FILE_NAME( ↑name ), ↓FILE_NAME( ↓name );

   Optimisation =
      ↑CODE( ↑code1 ), Optimise( ↓↑code1 ), ↓CODE( ↓code1 );

   Optimise( ↓↑code1 ) =
      {
         ↑CODE( ↑code2 ),
            (
               Fold
                  (
                     ↓fold1[code1.function][code2.function],
                     ↓( code1.operand = code2.operand ),
                     ↑code1
                  )
            |
               Fold
                  (
                     ↓fold2[code1.function][code2.function],
                     ↓code1.operand,
                     ↑code1
                  )
            |
               Fold
                  (
                     ↓fold3[code1.function][code2.function],
                     ↓code1.operand +
                        code2.operand * fold4[code2.function],
                     ↑code1
                  )
            |
               Output( ↓code1,↓code2,↑code1,↑csize )
         )
```

```
    );

Fold( ↓function,↓operand,↑{function,operand} );

Output( ↓code1,↓code2,↑code2,↑csize+size[code1.function] ) =
    [Label( ↓code1.function="LABEL",↓code1.operand,↑<reference> )],
        ↓CODE( ↓code1 );

Label( ↓function,↓operand,↑[operand → csize ] );
```

Figure 6.15

The rule 'Optimiser' references the rules 'Title' and 'Optimisation'. The rule 'Title' merely passes on the name of the current source program to the next phase of compiler. The rule 'Optimisation' will obtain an MI and reference the rule 'Optimise'. The rule 'Optimise' will obtain a second MI and perform references to the rules 'Fold' or 'Output', as required. Each reference to the rule 'Fold' involves a table lookup to see if the current two instructions can be replaced by a single more simple instruction. If this is possible the replacement instruction is passed to the rule 'Fold'. This call makes use of the tables 'fold1', 'fold2' and 'fold3' as defined in the 'VARIABLES' section (see Figure 6.7). The optimisations that can be carried out are constant folding and redundant load elimination. The rule 'Optimise' iterates until all of the current program is processed. The rule 'Fold' returns any instruction replacement found. The rule 'Output' references the rule 'Label' and outputs the oldest instruction (held in code1) replacing it with the following instruction (held in code2). The rule 'Label' is used to build a new table of labels against instructions; whilst not directly relavant to optimisation, this is necessary here since optimisation in general will have removed instructions and hence altered the label assignments.

The Optimiser specified above is limited to two instruction peephole optimisations. Again, there is no reason why the ACC can not be used to construct more powerful local optimisers. However, it is hoped that the above example demonstrates, in principle, the applicability of the ACC to the construction of local optimisers.

The Loader

The final 'RULES' section in the TSL compiler specification translates the MIs output by either the Translator or Optimiser into binary machine code for the TMA, resolving all forward references. The specification of the Loader is as follows.

```
RULES
   Loader =
      Title, Fixup;

   Title =
      ↑FILE_NAME( ↑name ), ↓FILE_NAME( ↓name );

   Fixup =
      {
         ↑CODE( ↑code ),
         [ Final( ↓code.function,↓code.operand ) ]
      };

   Final( ↓function,↓operand ) =
      (↓FINAL1( ↓class1[function] ), ↓FINAL2( ↓operand )) |
      (↓FINAL1( ↓class2[function] )) |
      (↓FINAL1( ↓class3[function] ), ↓FINAL2( ↓reference[operand] ));
```

Figure 6.16

The 'Loader' references the rules 'Title' and 'Fixup'. The rule 'Title' passes on

the name of the current source program to the output phase of the ACC (where it is used to name the final object file). The rule 'Fixup' reads an MI and references the rule 'Final' (the rule 'Fixup' iterates until the all of the current program is processed). The rule 'Final' translates this instruction into TMA machine code and outputs it. The operands of all jump instructions are determined using the table 'reference', generated the earlier.

The Aston Compiler Constructor under UNIX

The TSL compiler can be executed on a range of operating systems and host computers, due to the portability of the ACC. In this section we shall consider how the TSL compiler might be executed under the UNIX operating system on a SUN3 minicomputer.

The first step is to translate the TSL compiler specification into ACC Intermediate Execution Format (IEF). This translation can be carried out using the following command.

```
LOADER 200 <TSL
```

This command would cause the ACC LOADER to allocate 200 units of internal storage space and then translate the CCL specification in the file 'TSL' into ACC IEF. The outline operation of the ACC LOADER would in this case be as shown in Figure 6.17.

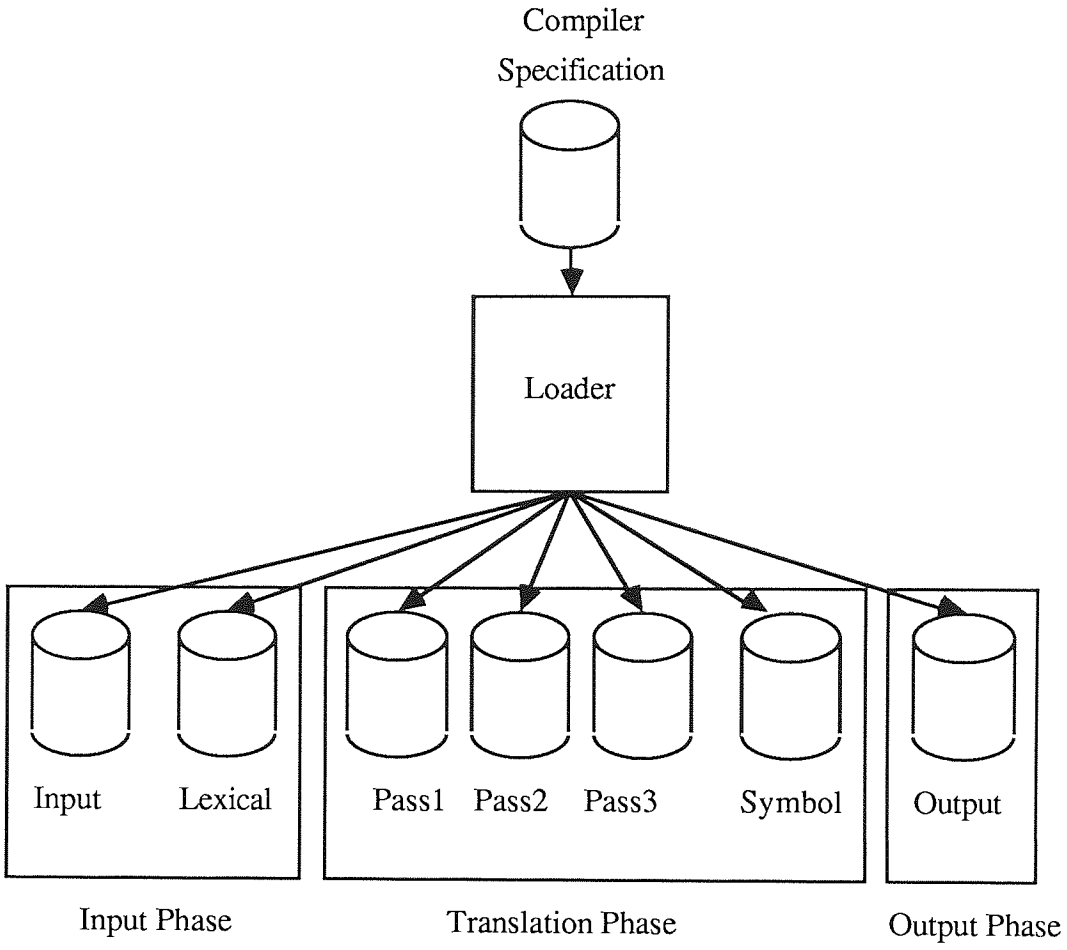


Figure 6.17

The ACC IEF files produced by the ACC LOADER (including the TSL compiler title 'Example' prefixed to the generic file names used above) are as follows.

File Name	Contents
Example_Input	This is a control file and is used to control the execution of the ACC SCANNER program.
Example_lexical	This is an information file containing information about all constant lexical strings defined within the CCL specification.
Example_pass1	This is a control file generated from the first 'RULES' section within the CCL specification. This file is used to control the execution of the first copy of the ACC PARSER program.
Example_Pass2	This is a control file generated from the second 'RULES' section within the CCL specification.

```
                    This file is used to control the execution of the
                    second copy of the ACC PARSER program.
Example_pass3       This is a control file generated from the third
                    'RULES' section within the CCL specification.
                    This file is used to control the execution of the
                    third copy of the ACC PARSER program.
Example_symbol      This is an information file containing information
                    about all the constants and variables defined
                    within the CCL specification.
Example_output      This is a control file and is used to control the
                    execution of the ACC RECONSTRUCT program.
```

Figure 6.18

The TSL compiler is now ready for execution. However, this execution may be carried out in a number of ways. The most straightforward method is to execute each section of the TSL compiler sequentially, as follows.

```
SCANNER Example_input Example_lexical New_lexical 500 <Source >Temp1
PARSER Example_pass1 Example_symbol New_symbol 500 <Temp1 >Temp2
PARSER Example_pass3 New_symbol NONE 500 <Temp2 >Temp3
RECONSTRUCT Example_output New_lexical <Temp3
```

Figure 6.19

These commands would be suitable for executing the TSL compiler (in this case excluding the optimisation phase) on a small uni-processor system. However, the components of the TSL compiler could also be executed in parallel (use is made of the UNIX shell operator '|' which indicates parallel execution of two or more processes, linked by 'pipes'), as follows.

```
SCANNER Example_input Example_lexical New_lexical 500 <Source | \
PARSER Example_pass1 Example_symbol New_symbol 500 >Temp1

PARSER Example_pass3 New_symbol NONE 500 <Temp1 | \
RECONSTRUCT Example_output New_lexical
```

Figure 6.20

To exploit parallelism in the execution of the TSL compiler it is necessary to partition the compiler into two stages of execution. Otherwise, it would not be possible to resolve forward references. The first stage of execution reads the source text, translates it into a stream of MIs, and collects information about forward references. The outline operation of the this stage is as follows.

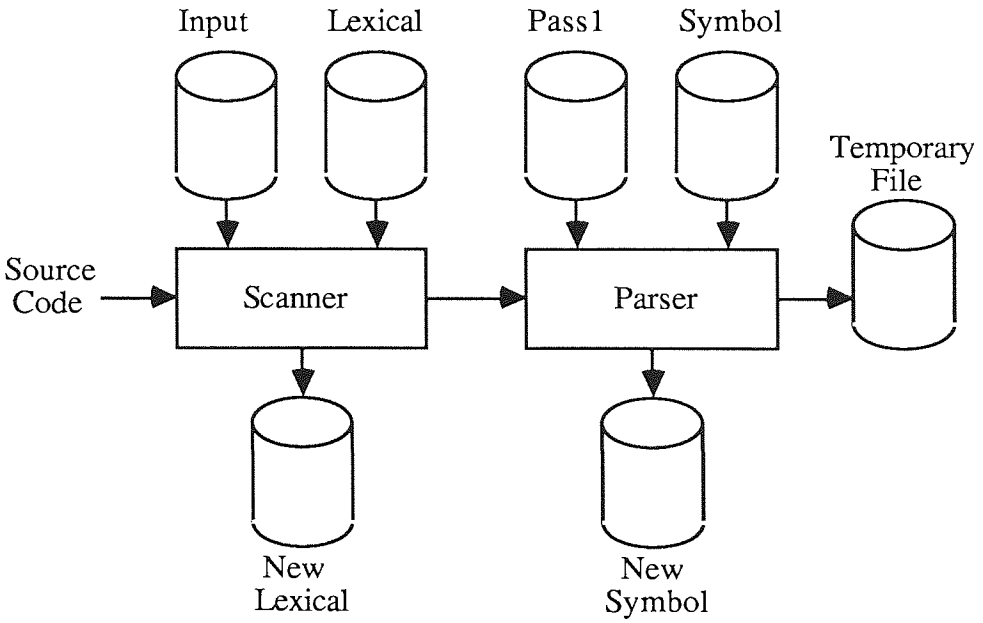


Figure 6.21

The first stage uses the control files 'Input' and 'Pass1' and the information files 'Lexical' and 'Symbol' in order to process the source text, and produces two new files called 'New_lexical' and 'New_symbol' (these files are updated versions of the files 'Lexical' and 'Symbol').

The second stage of execution translates the MIs generated by the first stage into machine code for the TMA and fixes up all forward references. The outline

operation of the this stage is as shown in Figure 6.22.

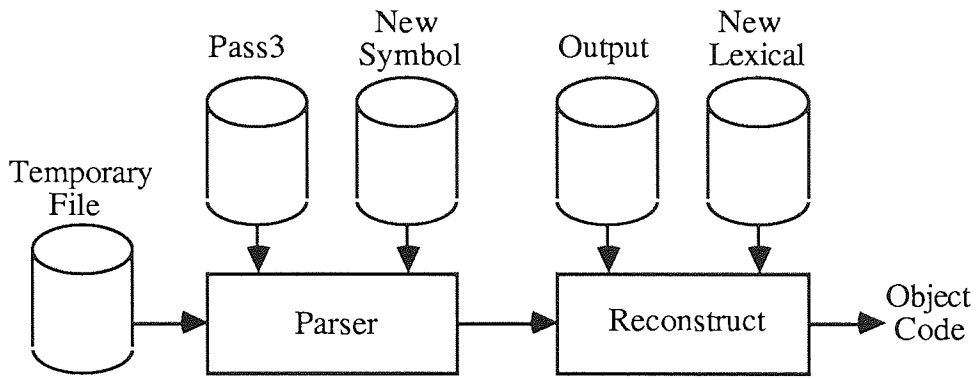


Figure 6.22

The second stage uses the control files 'Pass3' and 'Output' and the updated information files 'New_lexical' and 'New_symbol'. Forward references can now be resolved using the information collected by the first stage.

A Sample Compilation

We shall examine how the TSL compiler would translate a small example program. We shall once again assume UNIX as the host operating system. The example program we shall consider is as shown in Figure 6.23.

```
PROGRAM EXAMPLE

VAR A,B;

BEGIN
   A := 1 + 7;
   B := A + 2;

   IF B-4 = A+B-7 THEN
      BEGIN
         B := A / 5;
         A := B * B
      END
   ELSE
      IF A≠B THEN
         BEGIN
            A := 0;
            B := 0
         END
      FI
   FI;

   A := A + B + 10 - 3
END.
```

Figure 6.23

The commands we shall use to compile the example program (including this time the optimisation phase) are as follows.

```
SCANNER Example_input Example_lexical New_lexical 500 <Source | \
PARSER Example_pass1 Example_symbol NONE 500 | \
PARSER Example_pass2 Example_symbol New_symbol 500 >Temp1

PARSER Example_pass3 New_symbol NONE 500 <Temp1 | \
RECONSTRUCT Example_output New_lexical
```

Figure 6.24

Once again the execution of the TSL compiler will be carried out in two stages. The outline operation of the first stage of execution is as shown in Figure 6.25.

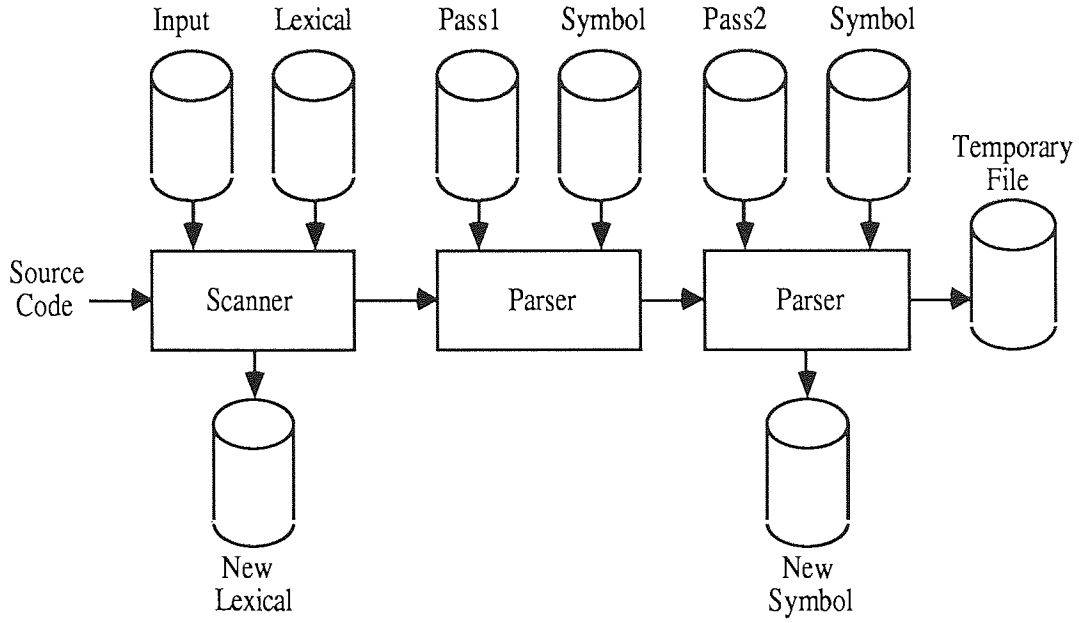


Figure 6.25

The ACC SCANNER would read the example program given in Figure 6.23 and translate it into ACC tokens, passing these on to the first pass of the ACC PARSER. This pass of the ACC PARSER would then translate these tokens into MIs, as follows.

```
Function    Operand        Source Code
   ASF          2          VAR A,B

   LDC          1
   LDC          7          A = 1 + 7
   ADD          0
   STA          0

   LDA          0
   LDC          2          B = A + 2
   ADD          0
   STA          1

   LDA          1
   LDC          4
   SUB          0
   LDA          0
   LDA          1          IF B-4 = A+B-7 THEN
   ADD          0
```

```
LDC         7
SUB         0
SUB         0
JNE         0

LDA         0
LDC         5               B = A / 5
DIV         0
STA         1

LDA         1
LDA         1               A = B * B
MUL         0
STA         0

JMP         1               ELSE
LABEL       0

LDA         0
LDA         1               IF A ≠ B THEN
SUB         0
JEQ         2

LDC         0               A = 0
STA         0

LDC         0               B = 0
STA         1

LABEL       2               FI

LABEL       1               FI

LDA         0
LDA         1
ADD         0
LDC         10              A = A + B + 10 - 3
ADD         0
LDC         3
SUB         0
STA         0

HLT         0
```

Figure 6.26

At a glance the above (naively generated) code will show its poor quality (eg. there are many redundant 'LDA' (load) instructions). Thus, the second pass of the ACC PARSER optimises these MIs, as shown in Figure 6.27.

```
Function    Operand          Source Code
   ASF         2             VAR A,B

   LDC         1
   ADC         7             A = 1 + 7
   ST          0             (Combined 'LDC 7' and 'ADD', and
                             'STA 0' and 'LDA 0')

   ADC         2             B = A + 2
   ST          1             (Combined 'LDC 2' and 'ADD', and
                             'STA 1' and 'LDA 1')

   SBC         4
   LDA         0
   ADA         1             IF B-4 = A+B-7 THEN
   SBC         7             (Combined 'LDC 4' and 'SUB', 'LDA 1'
   SUB         0             and 'ADD', and 'LDC 7' and 'SUB')
   JNE         0

   LDA         0
   DVC         5             B = A / 5
   ST          1             (Combined 'LDC 5' and 'DIV', and
                             'STA 1' and 'LDA 1')

   MLA         1             A = B * B
   STA         0             (Combined 'LDA 1' and 'MUL')

   JMP         1             ELSE
   LABEL       0

   LDA         0
   SBA         1             IF A ≠ B THEN
   JEQ         2             (Combined 'LDA 1' and 'SUB')

   LDC         0             A = 0
   STA         0

   LDC         0             B = 0
   STA         1

   LABEL       2             FI

   LABEL       1             FI

   LDA         0
   ADA         1
   ADC         10            A = A + B + 10 - 7
   SBC         3             (Combined 'LDA 1' and 'ADD', 'LDC 10'
   STA         0             and 'ADD', and 'LDC 3' and 'SUB')

   HLT         0
```

Figure 6.27

These optimised instructions would then be stored in a temporary file ready for the second stage of compilation. It can be seen that although the TSL compiler's optimisation phase has removed a number of inefficient instruction sequences, a small number of further opportunities for optimisations have arisen. Most of these inefficient instruction sequences could be removed by a second pass of the very same optimiser, producing the following MIs.

```
Function    Operand          Source Code
   ASF          2            VAR A,B

   LDC          8            A = 1 + 7
   ST           0            (Combined 'LDC 1' and 'ADC 7')

   ADC          2            B = A + 2
   ST           1

   SBC          4
   LDA          0
   ADA          1            IF B-4 = A+B-7 THEN
   SBC          7
   SUB          0
   JNE          0

   LDA          0
   DVC          5            B = A / 5
   ST           1

   MLA          1            A = B * B
   STA          0

   JMP          1            ELSE
   LABEL        0

   LDA          0
   SBA          1            IF A ≠ B THEN
   JEQ          2

   LDC          0            A = 0
   STA          0            (A less simple minded optimiser may
                             combine 'STA 0' and following 'LDC 0'
                             to 'ST 0'.)

   LDC          0            B = 0
   STA          1
```

```
LABEL       2               FI

LABEL       1               FI

LDA         0
ADA         1               A = A + B + 10 - 3
ADC         7               (Combined 'ADC 10' and 'SBC 3')
STA         0

HLT         0
```

Figure 6.28

Examination shows that these optimisation passes have (in total) reduced the number of instructions by around 30% and the total code size by around 25% (compared with the original naive code shown in Figure 6.26).

The second stage of the TSL compiler translates the optimised MIs output from the first stage into machine code for the TMA. The outline operation of this stage is as follows.

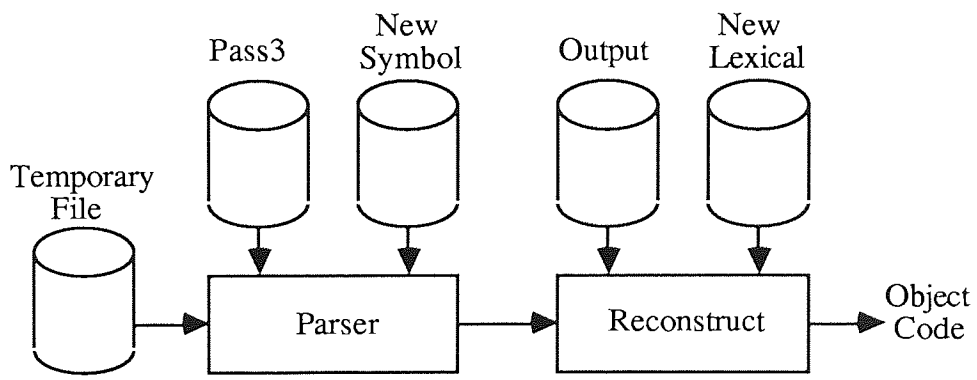


Figure 6.29

The machine code output from this stage (in hexadecimal) would be as shown in Figure 6.30.

Function	Operand	Source Code
30	00 02	VAR A,B
20	00 08	A = 1 + 7
01	00 00	
21	00 02	B = A + 2
01	00 01	
22	00 04	
10	00 00	
11	00 01	IF B-4 = A+B-7 THEN
22	00 07	
41		
62	00 31	
10	00 00	
24	00 05	B = A / 5
01	00 01	
13	00 01	A = B * B
15	00 00	
60	00 46	ELSE
10	00 00	
12	00 01	IF A ≠ B THEN
61	00 46	
20	00 00	A = 0
15	00 00	
20	00 00	B = 0
15	00 01	
10	00 00	
11	00 01	A = A + B + 10 - 3
21	00 07	
15	00 00	
50		

Figure 6.30

The compilation process is now entirely complete.

Summary

We have seen in this chapter how the ACC can be used to construct a compiler for a small example programming language. It is apparent that the ACC allows such compilers to be specified in a way that is much clearer and much more concise than traditional high level programming languages. It is also substantially shorter and more self-documenting than a compiler would be obtained by employing Lex and Yacc together with the augmentation that would be necessary to handle the context sensitive aspects and the translation to machine code.

In the next chapter we shall examine how the ACC may be applied to the specification of a commercial programming language, namely Pascal. The specification discussed in this chapter is a partial implementation of the lexical, syntactic, semantic and code generation (excluding optimisation) phases of an ISO Pascal compiler.

Chapter 7

A Pascal Specification using the Aston Compiler Constructor

Introduction

In the previous chapter we saw how the Aston Compiler Constructor (ACC) could be used to implement a complete compilation system for a small example programming language. In this chapter we shall focus our attention on how the ACC could be used to implement a typical commercial programming langauge. In particular, we shall examine a Pascal translator defined in the Compiler Construction Langauge (CCL) and given in Appendix 5 of this thesis. The translator given is designed to process a substantial subset of the International Standards Organisation (ISO) Pascal [BSI 1982] into a target language modelled on P-code [Ammann 1981].

An Overview of the Translator

The translator specified in Appendix 5 accepts ISO Pascal (Level 0) with the following main restrictions: that no input or output functions are provided; and that functions and procedures may not be passed as formal arguments. The translator generates P-code as described in [Ammann 1981] with the principal exception that value parameters passed to functions and procedures are stacked before entry to the routine (using an additional special instruction 'CPY'). The deviations from the standard may be justified by observing that the translator discussed in this chapter is based on a Pascal specification given in [Watt 1980]. This specification was written prior to the official ISO Pascal standard and hence is in some places a minor subset and in other places a minor superset of the ISO standard. The main reason for using this particular Pascal specification was to allow the reader to make direct comparisons between this specification and the CCL specification

given here. In most cases the notation used in [Watt 1980] is a subset of the ACC specification defined in CCL. Hence, it is possible for the reader to assess the improvements offered by the ACC system, particularly in the areas of specification conciseness and completeness. The main differences between the two notations dissussed here are principally in the areas of input/output specification, code generation and error processing. This is particularly noteworthy because the specification given in [Watt 1980] was implemented using a multi-pass bottom-up Attribute Grammar (AG) evaluator, where as the ACC notation is implemented using a single-pass top-down AG evaluator.

Development Issues

While developing the Pascal specification given in this thesis a number of implementation issues arose that appear worthy of disscussion here. The most obvious of these issues concerns the CCL operator set. At the time when the CCL was designed one of the main design aims was to develop a compiler specification language that was both small and easy to learn, yet powerful enough to allow the description of entire compilation systems. Although it is believed that this has been largely achieved it now appears that it would have been advantageous to have included a small number of extra operators within the original CCL operator set, for example operators such as: exclusive union, exclusive or, alignment functions and constant folding functions. Fortunately, the operation of many of these operators (and functions) can already be specified by using sequences of the existing CCL operators; however, this leads to more verbose and less efficient compiler specifications. It is hoped that this deficiency of the ACC will soon be

rectified by the addition a number of extra operators in a future version of the system.

Another quite separate issue that arose recently concerning the debugging of compiler specifications. Athough languages like CCL are somewhat more formal than many traditional high level programming languages, it is still often difficult to trace errors within compiler specifications. This problem was first noted quite early on in the ACC project. However, only more recently has it been realised that debugging tools for systems like ACC are also likly to be useful for teaching students about the internal operation of compilers. With this in mind, a new version of the ACC system has been provided with a number of special debugging modes capable of displaying the internal state of the ACC system.

Finally, a further issue has been discovered in the implementation of the attribute evaluator used in the ACC. The problem arises because of the top-down left to right attribute evaluation scheme used by the system when processing CCL specifications. Consider the following CCL rules.

```
Number( ↑value ) =
   Boolean( ↑value ) | Integer( ↑value ) | Real( ↑value );

Boolean( ↑boolean(boolean_value) ) =
   ↑BOOLEAN( ↑boolean_value );

Integer( ↑integer(integer_value) ) =
   ↑INTEGER( ↑integer_value );

Real( ↑real(real_value) ) =
   ↑REAL( ↑real_value );
```

Figure 7.1

The current ACC attribute evaluator evaluates rules (such as 'Number') in a left to right order. Unfortunately, in the case where the next symbol is 'REAL' this causes the rules 'Boolean' and 'Integer' to be evaluated needlessly. In this simple case this inefficency is not really important, however in cases where a larger number of rules would need to be evaluated the significance of this difficulty increases. It is possible to rewrite such rules and overcome this problem, for example by moving the terminals 'BOOLEAN', 'INTEGER' and 'REAL' into the rule 'Number'. However, this makes such rules more opaque and hence reduces the clarity of the specification, merely for the sake of efficiency.

One solution to this problem is to propagate information about the terminals in the rules 'Boolean', 'Integer' and 'Real' into the rule 'Number'. It is then possible for the attribute evaluator to decide which rule to activate next (by simply examining at the next input token). This method is analogous to the method used for deciding between alternatives in recursive descent compilers, which must proceed without any backtracking. However, this scheme does not resolve a similar problem which arises for rules which involve only output terminals. Fortunately, this is of no consequence as the selection of the next rule to activate in such a case can usually be made at the base of an appropriate rule subtree (by examining local attribute information). Therefore, it can be seen that our methods allow the compiler writer to produce clearer compiler specifications for automated compiler construction systems (like the ACC) while avoiding most implemention penalties. The generality of our approach makes it applicable to many top-down single pass attribute evaluators and therefore it is commended to the reader for

application in this area.

With the exception of the infelicity outlined above, extensive testing of the ACC has only shown a few other minor problems within the implementation of the system itself. Mostly these are to do with algorithms that are somewhat more store profligate that they need be. At the time of writing recent tuning of the lastest ACC system has improved the execution performance of the system by about 25% and store utilisation by about 50%. Furthermore, it is believed that additional improvements are still available by careful optimisation of selected algorithmns.

A Brief Review

The Pascal specification discussed here was developed by the author over a period of about 12 months during his spare time (equivalent to about 3 months full time work). Although at first difficult, the development of the specification seemed to become progressively easier as work continued. This extensive experiment showed that although there are initial hurdles to overcome, CCL does indeed support a most expressive, fluent and convenient programming style. The most difficult parts of the specification were the implementation of the Pascal type system and numeric expressions. As mentioned above, a number of difficulties were encountered during the design and coding of the specification. However, despite these it is felt that final result is a clear and concise specification of the language, although it is acknowledged that modest extensions to the CCL could improve this further. The size of the final specification was somewhat larger than expected at around 2,000 lines. Nevertheless, this is still about 5-7 times smaller

than an equivalent specification in a tradition high level programming language, such as C or Pascal.

The complete Pascal compiler specification (which is given in Appendix 5) has been input to and processed successfully by the ACC system, producing files containing lexical, parsing and control information (as described in Chapter 5). Taken together with the standard ACC compiler execution system, the outcome is a complete compiler for the Pascal dialect described at the beginning of this chapter.

The resulting compiler has been tested successfully on a variety of Pascal programs. For illustration, a simple example of Pascal source and the P-code generated is given in Appendix 6. It is apparent that the quality of code generated is in its raw state quite poor. However, a simple optimiser (similar to the example given in Chapter 6) could easily improve this considerably. Moreover, there is no known fundamental reason why code generated by automated compiler construction systems (like the ACC) need be inferior to the code generated by traditional hand crafted compilers. Also, it is clear that compilers developed using tools such as the ACC (and in particular the Pascal specification described here) are more retargetable than traditional hand crafted compilers. The observation is that the increased clarity, reduced size and tabular nature of specifications for systems like the ACC naturally make compiler retargeting easier.

Summary

In conclusion, it has been shown in this chapter (and Appendix 5) that specifying a compilation system for programming language such as Pascal is a soluble problem for AG based systems such as the ACC. Furthermore, it has been noted that the source of the resultant specification is likely to be (and in the case of Pascal demonstrably is) significantly smaller and more manageable than the source for a traditional hand crafted compiler. It is therefore believed that the ACC demonstrates a basis for considerably improved alternatives to traditional compiler construction techniques. The only aspect of compiler construction where this is not likely to be the case is in the domain of compiler execution speed. This area is examined in the following chapter.

Chapter 8

A Performance Evaluation of the Aston Compiler Constructor

Introduction

We have seen that the Aston Compiler Constructor (ACC) is a Translator Writing Tool (TWT) offering an automated alternative to traditional compiler construction techniques, and that the ACC supports a special compiler oriented specification language called as the Compiler Construction Language (CCL), which was described in Chapter 4. It is apparent from the demonstrations of the ACC in Chapters 6 and 7 that the ACC offers an alternative compiler construction technique that is both easier to use and more rapid than traditional compiler construction techniques. However, the ACC is based upon an interpretive execution system whereas a traditional compiler written in a compiled high level language will benefit from the superior execution speed of a 'native' machine code. Therefore, in this chapter we shall examine the performance of an ACC-built compiler and compare its performance with that of a traditional compiler.

Performance Details

An ACC performance test was carried out using the example compiler specification described in Chapter 6. The performance of the ACC-built compiler derived from this specification was analysed on a number of mini-computers. The mini-computers used were a High Level Hardware (HLH) Orion (rated at 0.6 MIPS), a Sun Systems SUN3 (rated at 2.1 MIPS) and a HLH Orion2 (rated at 5.7 MIPS).

The measured performance of the ACC was as follows.

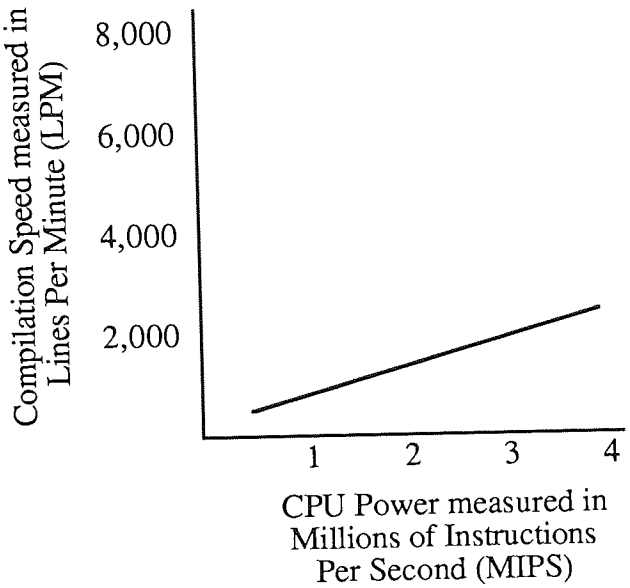


Figure 8.1

The performance of the ACC was discovered to be approximately proportional to the power of the processor being used in the test. It was calculated that the ACC was able to translate source code at speeds of around 700 LPM per 1 MIPS of processing power available. Later investigation showed that the ACC was largely processor intensive with about 80% of the processor's mill time being spent equally on computation and inter-process communication.

A further performance evaluation test was carried out using two SUN3 mini-computers connected together with a high speed ethernet. This test was conducted in order to assess what performance improvements might be gained by using the ACC on multi-processor configurations. It was discovered that the

dual-processor configuration used increased the performance of the ACC by about 20%. However, it was also discovered that 20% of the processing time taken was consumed by the operating system preparing the compilation. Thus, the ACC actually executed at about 40% faster on this dual-processor configuration than on a similar uni-processor configuration.

The measured execution performance of the ACC was as follows.

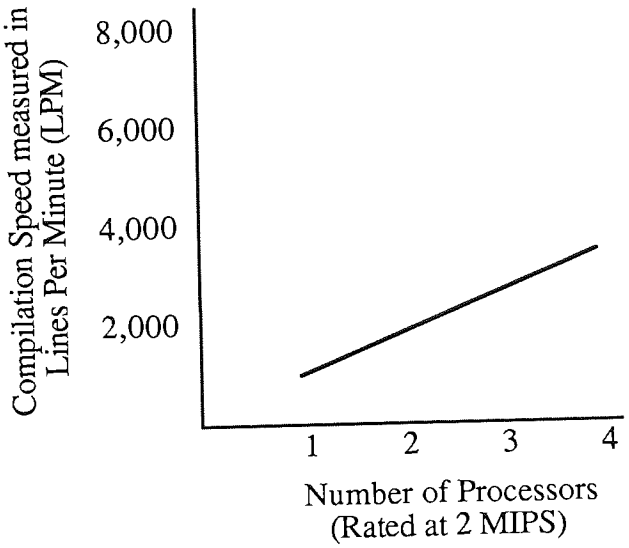


Figure 8.2

These performance figures were considerably worse than expected due to significant transmission delays caused by the buffering of inter-processor messages by the host operating system. It is believed that a total improvement of up to 60% could be expected with purpose-built multi-processor operating systems and hardware. However, the enevitable serial nature of a number of compilation operations, together with the consequent communication

requirements, place a limit on the benifit available from multi-processor execution.

Finally, in order to be able to fully appreciate the of the performance of the ACC it is necessary to compare its performance with a traditional compiler. The compiler chosen for this comparison was the Berkley Pascal compiler available under Berkley UNIX 4.2.

The measured performance of this compiler was as follows.

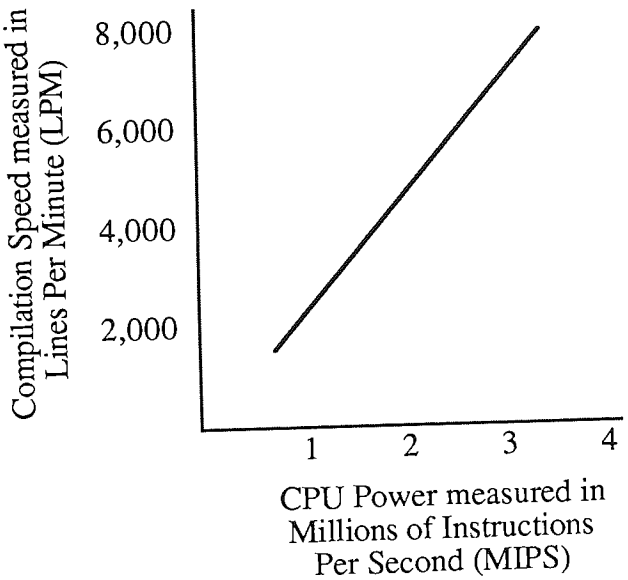


Figure 8.3

(Note: 'Lines of source code' is obviously a rather nebulous measure. To improve the comparability of the tests, the Pascal source was in principal identical to the TSL source code, altering only that which was necessary to produce legal Pascal code)

Again, the performance of this compiler was discovered to be approximately proportional to the power of the processor being used in the test. It was found that this compiler was able to translate source code at speeds of around 2,300 LPM per 1 MIPS of processing power available, thus making it around 3 times faster than the ACC.

Summary

In this chapter we have examined the performance of an ACC-built compiler and have seen that although it is inferior to that of compilers constructed using traditional compiler construction techniques, the difference in performance is only a low order of magnitude. Already a number of improvements to the ACC are underway in order to reduce this performance gap, for example measures to achieve worthwhile improvements in the efficiency of the input and output token translation operations appear to be available. Moreover, a number of simple additions to the CCL (such as extra predefined operators) would increase its specification power, leading to shorter, simpler and faster executing rules. It is believed that in the future that this performance differential can be narrowed to around a factor of 1.5. It is further believed that increases in the processing power of future computers and a move towards multi-processor architectures will make systems like the ACC even more attractive, compared to traditional compiler construction techniques. The continuing rise in the cost of human software construction effort relative to hardware costs also increasingly favours ACC-type tools.

Finally, we shall now attempt to see what conclusions can be drawn from the ACC project and discuss the future of automated compiler construction.

Chapter 9

Conclusions

Introduction

The early chapters of this thesis examined traditional compiler construction techniques and revealed that many phases of traditional High Level Programming Language (HLPL) compilers could be conveniently specified using a single notation, namely Attribute Grammars (AGs). It was concluded that it was worthwhile investigating whether a largely AG-based Translator Writing Tool (TWT) could be constructed which would be capable of producing complete compilation systems, so as to offer an improved automated alternative to traditional compiler construction techniques. As described in Chapters 3,4 and 5 an experimental AG based TWT called the Aston Compiler Constructor (ACC) was developed. This new TWT was then used to construct a small but complete compilation system (as described in Chapter 6) and a Pascal compilation system (as described in Chapter 7). Finally, the performance of the compilation system produced was tested, as documented in Chapter 8, and was shown to be acceptable.

Below we review the results obtained from the experimental ACC system, and draw a number of conclusions.

A Review of the Aston Compiler Constructor

A large number of TWTs have been developed in the last three decades. However, only a very few of these tools have even come near to supporting the formal specification and construction of complete compilation systems. Thus the ACC has a significantly greater scope of applicability than most TWTs (the best

known of course being the Lex-Yacc combination). It also provides a worthwhile qualitative enhancement on the few 'near-complete' systems (eg. [Keizer 1983] and [Madsen 1983]) which all leave some greater or lesser external interfacing or other coding effort to the user and inevitably lack the benefits of self-containedness.

In a few cases, techniques developed in certain earlier TWTs are similar to those used within the ACC. For example, the lexical processing carried out by the ACC LOADER is similar to the processing carried out in Lex, the operators available in the CCL are a modified superset of those suggested in [Madsen 1983], and the AG parsing techniques used in the ACC PARSER are comparable to the parsing technique used in [Koskimies 1983]. However, the ACC also includes a number of specific features that are believed to be new and interesting. These include: extended input and output facilities (incorporating input and output data structuring); natural support for the separate specification and interfacing of individual compiler phases (including the ability to specify systems ranging from complete compilation systems to small compiler components); and the ability to readily organise the execution of compiler phases in parallel (a feature that is lightly to become especially significant with the anticipated increase of multiprocessor architectures over the next decade).

Any use of TWTs based on formal notations brings advantages in the areas of self-documentation, clarity of specification, maintainability and reliablity. The ACC lexical and syntax specification is entirely formal and thus enjoys such advantages. Further, the ACC uses a more modern form of BNF, which is

generally easier to use and understand than traditional BNF, employed (for example) in Yacc. It hence enjoys greater advantages in clarity, etc, especially in connection with syntax.

However, perhaps the main gain of the ACC is that (to a substantial degree) it extends the above noted advantages to the whole of the compilation process. The attribute-based method of specifying context-sensitive constraints and translation operations allows these to be associated with the appropriate syntax rules in a straightforward manner. These context-sensitive constraints and translation operations are typically rather complex items and are thus broken down into small components which are clearly tied to the syntactic items to which they relate. It is acknowledged that the CCL is indeed a programming system, rather than a fully formal notation, and that its execution imposes some constraints on the attribute processing which can be performed. Nevertheless, the notation used is sufficiently close to the underlying formal system to give worthwhile gains in expressivity, clarity, and reliability.

The compiler specification style supported by the ACC has two further features which assist in promoting clarity and maintainability, namely support of tabular presentation and modularisation.

Tabular presentation is particularly valuable in compiler construction, for example due to the typical need to enumerate and organise the processing of collections of items which may be large in number but are each relatively simple (eg. operation codes and keywords). The TSL example of Chapter 6 also demonstrated the value

of straightforward tabular presentation in organising and specifying peephole optimisation.

Modularisation arises through the ability to include a number of 'RULES' sections, where each section corresponds to a transformation of the program being compiled from one representation to another (presumably closer to the intended final form). Thus, the division of a compiler into a number of passes is readily achieved. The CCL's token and type specification mechanisms allow the construction of clear and straightforward pass interfaces.

It is hence believed that the ACC has been demonstrated to have several significant advantages over both earlier TWTs and traditional HLPLs in the construction of complete compilation systems. It is accepted that in the area of performance the ACC is somewhat slower but in many cases this will be outweighed by the ACC's advantages in the areas of speed of construction and reliability. The ACC is already a well advanced practical system. We have noted that a small number of improvements could further enhance its performance, and it seems a quite reasonable hypothesis that this will result in a commercially viable tool.

In conclusion, this thesis has shown that by building on an appropriate formal foundation, a truely practical tool has been obtained for the construction of complete compilers, and thus we have thereby more closely tied together the formal specification of programming languages and the software engineering objectives of reliable and easily maintainable compiler construction.

Appendix 1

The

Compiler Construction

Language

Quick Reference Manual

Introduction

The design and application of the Compiler Construction Language (CCL) is discussed in detail in the main text of this thesis. The most important aspects of CCL syntax are briefly reviewed in this appendix to form a 'stand-alone' quick reference guide to the language.

The Compiler Construction Language

A complete CCL specification is specified in terms of a number of individual CCL sections. These sections are : 'Title', 'Constants', 'Input Tokens', 'Output Tokens', 'Types', 'Variables' and 'Rules'.

Title

The first section of a CCL specification is the 'TITLE' section. This section is used to name a CCL specification and is formally defined as follows.

```
Title          = "TITLE", Identifier.
Identifier     = ('A'→'Z'|'a'→'z'),
                 {'A'→'Z'|'a'→'z'| '0'→'9'|'_'}.
```

Figure A1.1

Constants

The second section of a CCL specification is the 'CONSTANTS' section. This section is used to define CCL compile-time constants and is optional, and formally

defined as shown in Figure A1.2.

```
Constants          = "CONSTANTS", Constant_Rules.
Constant_Rules     = Constant_Rule, { Constant_Rule }.
Constant_Rule      = Identifier, "=", Expression, ";".

Expression         = Sets, { ( "="|"≠"|">"|"≥"|"<"|"≤" ), Sets }.

Sets               = Binary, { ( "∩" | "∪" ), Binary }.
Binary             = Shift, { ( "&" | "|" ), Shift }.
Shift              = Sum, { ( "<<" | ">>" ), Sum }.

Sum                = Term, { ( "+" | "-" ), Term }.
Term               = Power, { ( "*" | "/" ), Power }.
Power              = Unary, { "^", Unary }.

Unary              = { "~" | "-" }, Primary.

Primary            = Constant |
                     Variable |
                     [ Function ], "(", Expression, ")".

Constant           = Character|Hexadecimal|Octal|Real|String|Unsigned.

Character          = '\0'→'\377'.
Hexadecimal        = ('0'→'9'|'A'→'F'|'a'→'f'),
                        {'0'→'9'|'A'→'F'|'a'→'f'}.
Octal              = '0'→'7', {'0'→'7'}.
Real               = '0'→'9', {'0'→'9'},".", '0'→'9', {'0'→'9'}.
String             = '"', {'\0'→'\377'}, '"'.
Unsigned           = '0'→'9', {'0'→'9'}.

Variable           = Identifier |
                     Identifier, ".", Variable |
                     Variable, "[" Expression "]" |
                     "<<", Variable, ">" |
                     "<", Variable, ">>".

Function           = "ABS"|"INTEGER"|"LENGTH"|"REAL"|"UNSIGNED".
```

Figure A1.2

The CCL supports a variety numeric, character and string constants, as follows.

Type Name	Type	Example
CHARACTER	Numeric	'a'
HEXADECIMAL	Numeric	0x12aF or 0X123C
INTEGER	Numeric	123
OCTAL	Numeric	0123
REAL	Numeric	123.456
TEXT	String	"Any Text Sting"
UNSIGNED	Numeric	123

Figure A1.3

Character or string constants may contain special character sequences, as follows.

Character	Meaning
\b	A back space character.
\f	A form feed character.
\n	A new line character.
\r	A return character.
\t	A tab character.
\Any other character	The character.
\Octal number	The character with this Octal character code.

Figure A1.4

The CCL supports a selection of binary operators, as follows.

Priority	Operator	Type	Meaning
2 (Highest)	^	Numeric	Exponential.
3	* /	Numeric	Multiplication and Division.
4	+ -	Numeric	Addition and Subtraction.
5	<< >>	Numeric	Logical Shifts.
6	& |	Numeric	Binary And and Or.
7	$\cup$ $\cap$	Tables	Join and Intersect tables.
8	= $\neq$	Any	Comparisons.
8	< $\leq$	Numeric	Comparisons.
8 (Lowest)	> $\geq$	Numeric	Comparisons.

Figure A1.5

And unary operators, as shown in Figure A1.6.

Priority	Operator	Type	Meaning
1	-	Numeric	Unary Minus
1	!	Numeric	Binary Not

Figure A1.6

Finally, a selection of built-in functions are supported, as follows.

Name	Function
ABS	Return the absolute value of following expression.
INTEGER	Truncate the value of the following expression to integer.
LENGTH	Return the length of following string.
REAL	Float the value of the following expression to real.
UNSIGNED	Truncate the value of the following expression to unsigned.

Figure A1.7

Input Tokens

The third section of a CCL specification is the 'INPUT TOKENS' section. It is used to specify input terminals and is formally defined as follows.

```
Input_Section     = "INPUT", "TOKENS", Input_Rules.
Input_Rules       = Input_Rule, { Input_Rule }.

Input_Rule        = "VOID", "=", Input_Token, ";" |
                    Identifier, "=", "CONSTANT", String, ";" |
                    Identifier, "=", Input_Token, ";" |
                    Identifier, "=", Input_Structure, ";".


Input_Structure   = "STRUCTURE", "{", Input_Clauses, "}".
Input_Clauses     = Input_Clause, { Input_Clause }.

Input_Clause      = "VOID", ":", Input_Token, ";" |
                    "CONSTANT", ":", String, ";" |
                    Identifier, ":", Input_Token, ";".
```

```
Input_Token       = Input_Choice | Simple_Type.

Input_Choice      = Input_Sequence, { "|", Input_Sequence }.
Input_Sequence    = Input_Primary, { ",", Input_Primary }.
Input_Primary     = "(", Input_Choice, ")" | "[", Input_Choice, "]" |
                    "{", Input_Choice, "}", Iterations |
                    Input_Terminal.

Iterations        = [ "*", Expression, [ "→", Expression ] ].

Input_Terminal    = Range | String | "?".
Range             = "'",'\0'→'\377',"'", "→","'",'\0'→'\377',"'".
```

Figure A1.8

The input terminals are specified in this section of a CCL specification and are defined using Regular Expressions (REs). These REs consists of RE operators and RE terminals. The RE operators available are as follows.

Priority	Operators	Meaning
1(Highest)	( )	Enclosed clauses are to be grouped.
1	[ ]	Enclosed clauses are optional.
1	{ }	Enclosed clauses are to be iterated.
2	,	Sequence operator.
3(Lowest)	|	Alternative operator.

Figure A1.9

And the RE terminals, as follows.

Example	Meaning
"A textual string"	Any textual string.
'A'→'Z'	Any character range.
?	Any character.

Figure A1.10

The 'CONSTANT' keyword may be used to cause the value associated with a

particular token to be ignored and deleted. The 'VOID' keyword performs the same function on entire tokens.

A number predefined REs are supported by the CCL, as follows.

```
Type            Definition
CHARACTER       '\0'→'\377'
HEXADECIMAL     ("0x"|"0X"), {'0'→'9'|'A'→'F'|'a'→'f'}
INTEGER         '1'→'9', {'0'→'9'}
OCTAL           "0", {'0'→'7'}
REAL            '1'→'9', {'0'→'9'}, ".", '0'→'9', {'0'→'9'}
UNSIGNED        '1'→'9', {'0'→'9'}

SHORT FIXED     ? * sizeof(short unsigned) ***
FIXED           ? * sizeof(unsigned) ***
LONG FIXED      ? * sizeof(long unsigned) ***
FLOAT           ? * sizeof(float) ***
LONG FLOAT      ? * sizeof(long float) ***

FILE            ('A'→'Z'|'a'→'z'), {'A'→'Z'|'a'→'z'}
TEXT            ('A'→'Z'|'a'→'z'), {'A'→'Z'|'a'→'z'}

                *** As defined locally in the C programming language
```

Figure A1.11

All of these predefined REs return a value to the CCL rule which applies them.

Finally, the CCL supports the definition of structured input terminals. These are specified using the 'STRUCTURE' keyword and allow structured information to be easily input to and output from CCL specifications.

Output Tokens

The fourth section of a CCL specification is the 'OUTPUT TOKENS' section. It is used to define output terminals and is formally defined as follows.

```
Output_Section    = "OUTPUT", "TOKENS", Output_Rules.
Output_Rules      = Output_Rule, { Output_Rules }.

Output_Rule       = Identifier, "=", "CONSTANT", String, ";" |
                    Identifier, "=", Simple_Type, ";" |
                    Identifier, "=", Output_Structure, ";".

Output_Structure  = "STRUCTURE", "{", Output_Clauses, "}".
Output_Clauses    = Output_Clause, { Output_Clause }.

Output_Clause     = "CONSTANT", ":", String, ";" |
                    Identifier, ":", Simple_Type, ";".
```

Figure A1.12

The 'OUTPUT TOKENS' section is defined to be consistent with the 'INPUT TOKENS' section, as far as is possible. However, REs and the 'CONSTANT' and 'VOID' keywords may not be used in the 'OUTPUT TOKENS' section.

Types

The fifth section of a CCL specification is the 'TYPES' section. This section is used to define new variable types and is optional, it is formally defined as shown in Figure A1.13.

```
Types              = "TYPES", Type_Rules.
Type_Rules         = Type_Rule, { Type_Rule }.

Type_Rule          = Identifier, "=", Complex_Type, ";" |
                     Identifier, "=", Type_Structure, ";".

Type_Structure     = "STRUCTURE", "{", Type_Clauses, "}" |
                     "UNION", "{", Type_Clauses, "}" |
                     "TABLE", "[", Complex_Type, "]".

Type_Clauses       = Type_Clause, { Type_Clause }.
Type_Clause        = Identifier, ":", Complex_Type, ";".

Complex_Type       = Identifier | Simple_Type.
Simple_Type        = "CHARACTER" | "INTEGER" | "HEXADECIMAL" |
                     "OCTAL" | "REAL" | "UNSIGNED" |
                     "SHORT", "FIXED" | "FIXED" | "LONG", "FIXED" |
                     "FLOAT" | "LONG", "FLOAT" |
                     "TEXT" | "FILE", "NAME".
```

Figure A1.13

The standard types available in the CCL are as follows.

Type	Meaning
CHARACTER	A single Character
HEXADECIMAL	A hexadecimal number
INTEGER	An integer number
OCTAL	An octal number
REAL	A real number
UNSIGNED	An unsigned number
SHORT FIXED	A short fixed point number
FIXED	A fixed point number
LONG FIXED	A long fixed point number
FLOAT	A floating point number
LONG FLOAT	A long floating point number
TEXT	A textual string
FILE	A legal file name

Figure A1.14

The 'STRUCTURE', 'UNION' and 'TABLE' keywords may be used to defined CCL structures, unions and tables. These data structures can be accessed using

the CCL dot, sequence and selection operators, as follows.

Operator	Meaning
Variable.Name	Select the field 'Name' in the structure or union 'Variable'.
<< Symbols >	Select the first entry in the table 'Symbols'.
< Symbols >>	Select the last entry in the table 'Symbols'.
Symbols["A"]	Select the entry with the key "A" in the table 'Symbols' .

Figure A1.15

Variables

The sixth section of a CCL specification is the 'VARIABLES' section. This section is used to define new variables for the following 'RULES' section(s) and is optional, it is formally defined as follows.

```
Variables          = "VARIABLES", Variable_Rules.
Variable_Rules     = Variable_Rule, { Variable_Rule }.

Variable_Rule      = Complex_Type, Variable_Names, ";".
Variable_Names     = Variable_Name, { ",", Variable_Name }.

Variable_Name      = Identifier, [ "=", Complex_Value ].

Complex_Value      = Expression |
                     "{", Complex_Value, { ",", Complex_Value }, "}" |
                     Identifier, "{", Complex_Value, "}" |
                     "[" [ Table_Value, { ",", Table_Value } ] "]".

Table_Value        = [ Expression, "→" ], Complex_Value.
```

Figure A1.16

The CCL supports two main types of variables. These are initialized variables and uninitialized variables. An initialized variable can be most simply described as a global variable. The contents of such variables do not need to be passed between

CCL rules but may be accessed directly by any rule within a CCL specification. These variables are ideal for the storage of static or semi-static data structures and greatly reduce the overhead of passing such data structures between CCL rules. An uninitialized variable can most simply be described as a local variable. Such variables are used to hold dynamic values as they are passed between CCL rules.

Rules

The final section of a CCL specification is the 'RULES' section. This section is used to define the CCL translation phases and may appear zero or more times, it is formally defined as follows.

```
Rules              = "RULES", AG_Rules.
AG_Rules           = AG_Rule, { AG_Rule }.

AG_Rule            = Rule_Header, [ "=", Rule_Body ], ";".

Rule_Header        = Identifier, [ "(", Rule_Parameters, ")" ].

Rule_Parameters    = Rule_Parameter, { ",", Rule_Parameter }.
Rule_Parameter     = "↑", Complex_Value | "↓↑", Variable |
                     "↓", Variable | "↓", "<", Variable, ">".


Rule_Body          = Rule_Sequence, { "|", Rule_Sequence }.
Rule_Sequence      = Rule_Primary, { ",", Rule_Primary }.
Rule_Primary       = "(", Rule_Body, ")" | "[", Rule_Body, "]" |
                     "{", Rule_Body, "}", Iterations |
                     Rule_Terminal.

Rule_Terminal      = Rule_Warning | Rule_Error |
                     Rule_IO_Terminal | Rule_Call.

Rule_Warning       = "<<", String, ">>", Rule_Body.

Rule_Error         = "<<", Identifier, ",", Identifier,
                         ",", String, ">>", Rule_Body.
```

```
Rule_IO_Terminal  = "↑", Identifier, [ "(", "↑",Variable, ")" ] |
                    "↓", Identifier, [ "(", "↓",Complex_Value, ")" ].

Rule_Call         = Identifier, [ "(", Rule_Operands, ")" ].

Rule_Operands     = Rule_Operand, { ",", Rule_Operand }.
Rule_Operand      = "↑", Variable | "↑", "<", Variable, ">" |
                    "↓↑", Variable | "↓", Complex_Value.
```

Figure A1.17

A CCL rule is specified in two main parts, a rule header and an optional rule body. The rule header defines the CCL rule name and its parameters, if any. Each parameter is preceded by an arrow indicating the direction of information flow.

The optional CCL rule body is specified using REs. These REs again consist of RE operators and RE terminals. The RE operators available are as follows.

Priority	Operators	Meaning
1(Highest)	( )	Enclosed clauses are to be grouped
1	[ ]	Enclosed clauses are optional
1	{ }	Enclosed clauses are to be iterated
2	,	Sequence operator
3(Lowest)	|	Alternative operator

Figure A1.18

The RE terminals available are called **rule terminals**, **rule calls** and **rule errors**.

A CCL **rule terminal** is used to input or output information to or from a CCL rule. A rule terminal is denoted within a CCL rule by a leading '↑' or '↓' symbol followed by a input or output terminal name. If the input or output terminal has an associated value, this value may be input or output using a bracketed value clause.

associated value, this value may be input or output using a bracketed value clause.

A CCL **rule call** is used to invoke other rules within a CCL specification. A rule call consists of the name of the rule to be called followed by an optional parameter list in brackets. The parameter types in a rule call must match the corresponding parameter types in the rule header. The RE operators '|', '[', ']' and '{', '}' sometimes affect the visibility of values returned by RE terminals. The following rule is adopted by the CCL : 'A variable is only considered visible outside an optional or iterative RE if its definition within that RE can be guaranteed'.

Finally, a CCL **rule error** is a special RE terminal used to detect and trap errors discovered during the execution of a CCL specification. There are two types of **rule error** terminals, namely warning terminals and error terminals.

A warning node consists of a '<<' symbol followed by some warning string and a '>>' symbol. A warning terminal is used to deal with any minor faults detected during the execution of a CCL rule.

An error node consists of a '<<' symbol followed by two token names, an error string and a '>>' symbol. An error terminal is used to deal more serious faults detected during the execution of a CCL rule.

End

The 'END' keyword is used to mark the end of a CCL specification.

Appendix 2

A Formal Specification

for the

Compiler Construction Language

Introduction

The Compiler Construction Language (CCL) is the compiler specification language processed the Aston Compiler Constructor (ACC). The formal CCL syntax specification given in this appendix is defined using the draft British Standards Institute (BSI) meta-language as proposed in [Scowen 1982]. An extension to this meta-language has been used in the following specification to simplify the definition of character ranges. For example, a range of characters, say 'A' to 'Z', is defined in the formal specification below as 'A'→'Z'.

The Formal Syntax Specification

```
Loader          = Title, [ Constants ], Tokens, [ Types ],
                      [ Variables ], { Rules }, End.

        (*      The Title Section                          *)

Title           = "TITLE", Identifier.
Identifier      = ('A'→'Z'|'a'→'z'),
                      {'A'→'Z'|'a'→'z'|'0'→'9'|'_'}.

        (*      The Constants Section                      *)

Constants       = "CONSTANTS", Constant_Rules.
Constant_Rules  = Constant_Rule, { Constant_Rule }.
Constant_Rule   = Identifier, "=", Expression, ";".

        (*      The Expression Section                     *)

Expression      = Sets, { ( "="|"≠"|">"|"≥"|"<"|"≤" ), Sets }.

Sets            = Binary, { ( "∩" | "∪" ), Binary }.
Binary          = Shift, { ( "&" | "|" ), Shift }.
Shift           = Sum, { ( "<<" | ">>" ), Sum }.

Sum             = Term, { ( "+" | "-" ), Term }.
Term            = Power, { ( "*" | "/" ), Power }.
Power           = Unary, { "^", Unary }.

Unary           = { "~" | "-" }, Primary.
```

```
Primary          = Constant | Variable | [ Function ], "(", Expression, ")".

Constant         = Character|Hexadecimal|Octal|Real|String|Unsigned.

Character        = '\0'→'\377'.
Hexadecimal      = ('0'→'9'|'A'→'F'|'a'→'f'),
                     {'0'→'9'|'A'→'F'|'a'→'f'}.
Octal            = '0'→'7', {'0'→'7'}.
Real             = '0'→'9', {'0'→'9'},".", '0'→'9', {'0'→'9'}.
String           = '"', {'\0'→'\377'}, '"'.
Unsigned         = '0'→'9', {'0'→'9'}.

Variable         = Identifier |
                   Identifier, ".", Variable |
                   Variable, "[" Expression "]" |
                   "<<", Variable, ">" |
                   "<", Variable, ">>".

Function         = "ABS"|"INTEGER"|"LENGTH"|"REAL"|"UNSIGNED".

        (*       The Tokens Section                          *)

Tokens           = Input_Section, Output_Section.

        (*       The Input Token Section                     *)

Input_Section    = "INPUT", "TOKENS", Input_Rules.
Input_Rules      = Input_Rule, { Input_Rule }.

Input_Rule       = "VOID", "=", Input_Token, ";" |
                   Identifier, "=", "CONSTANT", String, ";" |
                   Identifier, "=", Input_Token, ";" |
                   Identifier, "=", Input_Structure, ";".

Input_Structure  = "STRUCTURE", "{", Input_Clauses, "}".
Input_Clauses    = Input_Clause, { Input_Clause }.

Input_Clause     = "VOID", ":", Input_Token, ";" |
                   "CONSTANT", ":", String, ";" |
                   Identifier, ":", Input_Token, ";".

Input_Token      = Input_Choice | Simple_Type.

Input_Choice     = Input_Sequence, { "|", Input_Sequence }.
Input_Sequence   = Input_Primary, { ",", Input_Primary }.
Input_Primary    = "(", Input_Choice, ")" | "[", Input_Choice, "]" |
                   "{", Input_Choice, "}", Iterations |
                   Input_Terminal.

Iterations       = [ "*", Expression, [ "→", Expression ] ].

Input_Terminal   = Range | String | "?".
Range            = "'",'\0'→'\377',"'", "→","'",'\0'→'\377',"'".
```

```
        (*        The Output Token Section                *)

Output_Section    = "OUTPUT", "TOKENS", Output_Rules.
Output_Rules      = Output_Rule, { Output_Rules }.

Output_Rule       = Identifier, "=", "CONSTANT", String, ";" |
                    Identifier, "=", Simple_Type, ";" |
                    Identifier, "=", Output_Structure, ";".

Output_Structure  = "STRUCTURE", "{", Output_Clauses, "}".
Output_Clauses    = Output_Clause, { Output_Clause }.

Output_Clause     = "CONSTANT", ":", String, ";" |
                    Identifier, ":", Simple_Type, ";".

        (*        The Type Section                        *)

Types             = "TYPES", Type_Rules.
Type_Rules        = Type_Rule, { Type_Rule }.

Type_Rule         = Identifier, "=", Complex_Type, ";" |
                    Identifier, "=", Type_Structure, ";".

Type_Structure    = "STRUCTURE", "{", Type_Clauses, "}" |
                    "UNION", "{", Type_Clauses, "}" |
                    "TABLE", "[", Complex_Type, "]".

Type_Clauses      = Type_Clause, { Type_Clause }.
Type_Clause       = Identifier, ":", Complex_Type, ";".

Complex_Type      = Identifier | Simple_Type.
Simple_Type       = "CHARACTER" | "INTEGER" | "HEXADECIMAL" |
                    "OCTAL" | "REAL" | "UNSIGNED" |
                    "SHORT", "FIXED" | "FIXED" | "LONG", "FIXED" |
                    "FLOAT" | "LONG", "FLOAT" |
                    "TEXT" | "FILE".

        (*        The Variable Section                    *)

Variables         = "VARIABLES", Variable_Rules.
Variable_Rules    = Variable_Rule, { Variable_Rule }.

Variable_Rule     = Complex_Type, Variable_Names, ";".
Variable_Names    = Variable_Name, { ",", Variable_Name }.

Variable_Name     = Identifier, [ "=", Complex_Value ].

Complex_Value     = Expression |
                    "{", Complex_Value, { ",", Complex_Value }, "}" |
                    Identifier, "{", Complex_Value, "}" |
                    "[" [ Table_Value, { ",", Table_Value } ] "]".

Table_Value       = [ Expression, "→" ], Complex_Value.
```

```
        (*      The Rules Section                    *)

Rules           = "RULES", AG_Rules.
AG_Rules        = AG_Rule, { AG_Rule }.

AG_Rule         = Rule_Header, [ "=", Rule_Body ], ";".

Rule_Header     = Identifier, [ "(", Rule_Parameters, ")" ].

Rule_Parameters = Rule_Parameter, { ",", Rule_Parameter }.
Rule_Parameter  = "↑", Complex_Value | "↓↑", Variable |
                  "↓", Variable | "↓", "<", Variable, ">".

Rule_Body       = Rule_Sequence, { "|", Rule_Sequence }.
Rule_Sequence   = Rule_Primary, { ",", Rule_Primary }.
Rule_Primary    = "(", Rule_Body, ")" | "[", Rule_Body, "]" |
                  "{", Rule_Body, "}", Iterations |
                  Rule_Terminal.

Rule_Terminal   = Rule_Warning | Rule_Error |
                  Rule_IO_Terminal | Rule_Call.

Rule_Warning    = "<<", String, ">>", Rule_Body.

Rule_Error      = "<<", Identifier, ",", Identifier,
                      ",", String, ">>", Rule_Body.

Rule_IO_Terminal = "↑", Identifier, [ "(", "↑",Variable, ")" ] |
                   "↓", Identifier, [ "(", "↓",Complex_Value, ")" ].

Rule_Call       = Identifier, [ "(", Rule_Operands, ")" ].

Rule_Operands   = Rule_Operand, { ",", Rule_Operand }.
Rule_Operand    = "↑", Variable | "↑", "<", Variable, ">" |
                  "↓↑", Variable | "↓", Complex_Value.

        (*      The End Section                      *)

End             = "END".
```

Lexical Coventions

The CCL uses the following lexical conventions.

Comments and White Space

Comments and white space within a CCL specification are ignored by the ACC. They are defined as follows.

```
Comment          =   "(*", { '\0'→'\377' }, "*)".
White_Space      =   " " | "\t" | "\n".
```

Figure A2.1

Reserved Words

The CCL reserves a number of keywords for internal use. These keywords may not be used as identifiers within CCL specifications. The reserved keywords are as follows.

```
ABS
CONSTANT
FILE
HEXADECIMAL
INPUT
LENGTH
OCTAL
OUT
RULES
TABLE
TOKENS
UNSIGNED

AND
CONSTANTS
FIXED
IN
INTEGER
LONG
OR
OVERRIDE
SHORT
TEXT
TYPES
VARIABLES

CHARACTER
END
FLOAT
INOUT
JOIN
NOT
OUTPUT
REAL
STRUCTURE
TITLE
UNION
VOID
```

Figure A2.2

Alternative Symbols

The CCL supports a range of alternative operator symbols. These symbols may be used on ranges of computer hardware where the usual CCL operator symbols are unavailable.

The alternative operator symbols are supported by the CCL are as follows.

Symbol	Alternatives
^	**
&	AND
|	OR
≠	<>
≥	>=
≤	<=
~	NOT
→	->
∪	u
∩	n
[	(/
]	/)
{	(:
}	:)
↑	IN
↓↑	INOUT
↓	OUT

Figure A2.3

Appendix 3

A Simple

Compiler Construction Language

Specification

Introduction

The example Compiler Construction Language (CCL) specification discussed in Chapter 5 of this thesis is a formal specification of a simple assignment statement and its compilation to a simple machine code. A complete listing of this specification is given below.

The Simple Assignment Specification

```
TITLE Example

CONSTANTS
   LDA          = 0x10;
   ADA          = 0x11;
   SBA          = 0x12;
   STA          = 0x13;

INPUT TOKENS
   ASSIGN       = CONSTANT ":=";
   SUM          = "+" | "-";

   IDENTIFIER   = ('A'→'Z'), {'A'→'Z'|'0'→'9'};

OUTPUT TOKENS
   INSTRUCTION  = STRUCTURE
                  {
                     operation          : INTEGER;
                     CONSTANT           : "\t";
                     operand            : INTEGER;
                     CONSTANT           : "\n";
                  };

TYPES
   HASH_TABLE   = TABLE[INTEGER];

VARIABLES
   HASH_TABLE     identifier =
                  [
                     "A" → 1,
                     "B" → 2
                  ],
```

```
                    instruction =
                        [
                            ":=" → LDA,
                            "+"  → ADA,
                            "-"  → SBA
                        ];
    INTEGER         value;
    TEXT            name,operator;

RULES
    Assignment =
        ↑IDENTIFIER( ↑name ), ↑ASSIGN,  Expression,
            Store( ↓name );

    Expression =
        Operand( ↓":=" ),
            { ↑SUM( ↑operator ), Operand( ↓operator ) };

    Operand( ↓operator ) =
        ↑IDENTIFIER( ↑name ),
        ↓INSTRUCTION( ↓{instruction[operator],identifier[name]} );

    Store( ↓name ) =
        ↓INSTRUCTION( ↓{STA,identifier[name]} );

END
```

Appendix 4

A Larger

Compiler Construction Language

Specification

Introduction

The example Compiler Construction Language (CCL) specification discussed in Chapter 6 of this thesis is a formal specification of compiler for a simple programming language. A complete listing of this specification is given below.

A Small Programming Language

```
TITLE Example

       /*************************************************************/
       /*                                                           */
       /*   The specification of the compiler interface             */
       /*                                                           */
       /*************************************************************/

INPUT TOKENS
   PROGRAM       = CONSTANT "PROGRAM";
   VAR           = CONSTANT "VAR";
   BEGIN         = CONSTANT "BEGIN";
   IF            = CONSTANT "IF";
   THEN          = CONSTANT "THEN";
   ELSE          = CONSTANT "ELSE";
   FI            = CONSTANT "FI";
   ENDS          = CONSTANT "END";

   ASSIGN        = CONSTANT ":=";
   SUM           = "+" | "-";
   TERM          = "*" | "/";
   COMPARISON    = "=" | "≠" | "<" | "≤" | ">" | "≥";
   COMMA         = CONSTANT ",";
   SEMI_COLON    = CONSTANT ";";
   DOT           = CONSTANT ".";

   IDENTIFIER    = ('A'→'Z'), {'A'→'Z' | '0'→'9'};
   VALUE         = INTEGER;

   VOID          = " " | "\t" | "\n";
```

```
      /*************************************************************/
      /*                                                           */
      /*   The specification of the Optimiser & Loader interface   */
      /*                                                           */
      /*************************************************************/

OUTPUT TOKENS
   FILE_NAME     = FILE;
   CODE          = STRUCTURE
                     {
                        function          : TEXT;
                        CONSTANT          : "\t";
                        operand           : INTEGER;
                        CONSTANT          : "\n";
                     };

   FINAL1        = CHARACTER;
   FINAL2        = SHORT FIXED;

      /*************************************************************/
      /*                                                           */
      /*   The specification of Compiler types                     */
      /*                                                           */
      /*************************************************************/

TYPES
   HASH_TABLE    = TABLE[INTEGER];
   INSTRUCTION   = TABLE[TEXT];
   OPTIMISATION = TABLE[INSTRUCTION];

      /*************************************************************/
      /*                                                           */
      /*   The specification of Complier variables & tables        */
      /*                                                           */
      /*************************************************************/

VARIABLES                                                          */
      /*   Simple variables
   CODE          code,code1,code2;
   INTEGER       csize=0,dsize=0,label=0,new_label,operand,value;
   TEXT          function,name,operator;

      /*   The symbol table and forward jump resolution table     */
   HASH_TABLE    identifier = [],reference = [];
                                                                   */
      /*   The operator to function mapping table
   INSTRUCTION   instruction =
                    [
                       "+"  → "ADD",        "-"  → "SUB",
                       "*"  → "MUL",        "/"  → "DIV",

                       "="  → "JNE",        "≠"  → "JEQ",
                       "<"  → "JGE",        "≤"  → "JGR",
                       ">"  → "JLE",        "≥"  → "JLT"
```

```
                ];

    /*   The function size look up table                          */
HASH_TABLE      size =
                [
                   "LD"  → 3,          "ST"  → 3,

                   "LDA" → 3,          "ADA" → 3,
                   "SBA" → 3,          "MLA" → 3,
                   "DVA" → 3,          "STA" → 3,

                   "LDC" → 3,          "ADC" → 3,
                   "SBC" → 3,          "MLC" → 3,
                   "DVC" → 3,

                   "ASF" → 3,

                   "ADD" → 1,          "SUB" → 1,
                   "MUL" → 1,          "DIV" → 1,

                   "HLT" → 1,

                   "JMP" → 3,          "JEQ" → 3,
                   "JNE" → 3,          "JLT" → 3,
                   "JLE" → 3,          "JGR" → 3,
                   "JGE" → 3,

                   "LABEL" → 0
                ];

    /*   The peephole optimisation tables                         */
    /*   (for redundant load removal and constant folding)        */
 OPTIMISATION   fold1 =
                   [
                      "STA" →
                         [ "LDA" → "ST" ]
                   ];
 OPTIMISATION   fold2 =
                   [
                      "LDA" →
                         [
                            "ADD" → "ADA",
                            "SUB" → "SBA",
                            "MUL" → "MLA",
                            "DIV" → "DVA"
                         ],
                      "LDC" →
                         [
```

```
                        "ADD" → "ADC",
                        "SUB" → "SBC",
                        "MUL" → "MLC",
                        "DIV" → "DVC"
                      ]
                  ];
OPTIMISATION    fold3 =
                  [
                    "LDC" →
                      [
                        "ADC" → "LDC",
                        "SBC" → "LDC"
                      ],
                    "ADC" →
                      [
                        "ADC" → "ADC",
                        "SBC" → "ADC"
                      ],
                    "SBC" →
                      [
                        "ADC" → "SBC",
                        "SBC" → "SBC"
                      ]
                  ];
HASH_TABLE      fold4 =
                  [
                    "ADC" → 1,
                    "SBC" → -1
                  ];

    /*   Symbolic function to machine code mapping tables        */
HASH_TABLE      class1 =
                  [
                    "LD"  → 0x0,        "ST"  → 0x1,

                    "LDA" → 0x10,       "ADA" → 0x11,
                    "SBA" → 0x12,       "MLA" → 0x13,
                    "DVA" → 0x14,       "STA" → 0x15,

                    "LDC" → 0x20,       "ADC" → 0x21,
                    "SBC" → 0x22,       "MLC" → 0x23,
                    "DVC" → 0x24,

                    "ASF" → 0x30
                  ];
HASH_TABLE      class2 =
                  [
                    "ADD" → 0x40,       "SUB" → 0x41,
```

```
                    "MUL" → 0x42,       "DIV" → 0x43,

                    "HLT" → 0x50
                ];
HASH_TABLE      class3 =
                [
                    "JMP" → 0x60,       "JEQ" → 0x61,
                    "JNE" → 0x62,       "JLT" → 0x63,
                    "JLE" → 0x64,       "JGR" → 0x65,
                    "JGE" → 0x66
                ];

    /*************************************************************/
    /*                                                           */
    /*   The specification of a simple compiler                  */
    /*                                                           */
    /*************************************************************/

RULES
  Compiler =
    Program, Var, Statement, End;

  Program =
    ↑PROGRAM, ↑IDENTIFIER( ↑name ),
      ↓FILE_NAME( ↓name );

    /*************************************************************/
    /*                                                           */
    /*   The specification of program variables                  */
    /*                                                           */
    /*************************************************************/

  Var =
    ↑VAR, Variables, ↑SEMI_COLON,
      Code( ↓{"ASF",dsize},↑csize );

  Variables =
    Variable, { ↑COMMA, Variable };

  Variable =
    <<COMMA,SEMI_COLON,"Invalid variable identifier">>
      Variable_Name( ↑dsize,↑<identifier> );

  Variable_Name( ↑dsize+1,↑[name → dsize] ) =
    ↑IDENTIFIER( ↑name );

  Code( ↓code,↑csize+size[code.function] ) =
    ↓CODE( ↓code );
```

```
    /*************************************************************/
    /*                                                           */
    /*   The definition of program statements                    */
    /*                                                           */
    /*************************************************************/

Statement =
   <<SEMI_COLON,DOT,"Invalid statement">>
      ( Compound | Assignment | If );

Compound =
   ↑BEGIN, Statement, { ↑SEMI_COLON, Statement }, ↑ENDS;

    /*************************************************************/
    /*                                                           */
    /*   The Assignment statement                                */
    /*                                                           */
    /*************************************************************/

Assignment =
   ↑IDENTIFIER( ↑name ),
      <<SEMI_COLON,DOT,"Illegal assignment">>
         ( ↑ASSIGN,  Expression ),
            Code( ↓{"STA",identifier[name]},↑csize );

Expression =
   Term, { ↑SUM( ↑operator ), Term,
      Code( ↓{instruction[operator],0},↑csize ) };

Term =
   Primary, { ↑TERM( ↑operator ), Primary,
      Code( ↓{instruction[operator],0},↑csize ) };

Primary =
   ( ↑IDENTIFIER( IN name ),
      Code( ↓{"LDA",identifier[name]},↑csize ) )
      |
   ( ↑VALUE( ↑value ),
      Code( ↓{"LDC",value},↑csize ) );

    /*************************************************************/
    /*                                                           */
    /*   The If statement                                        */
    /*                                                           */
    /*************************************************************/

If =
   ↑IF, <<SEMI_COLON,DOT,"Illegal if statement">>
      ( Boolean( ↑new_label ), ↑THEN, Statement,
         [ ↑ELSE, Else_Statement( ↓↑new_label ) ],
            Assign_Label( ↓new_label,↑new_label,↑<reference> ) ),
```

```
      ↑FI;

   Boolean( ↑new_label ) =
      Expression, ↑COMPARISON( ↑operator ), Expression,
         Allocate_Label( ↑new_label,↑label ),
         Code( ↓{"SUB",0},↑csize ),
         Code( ↓{instruction[operator],new_label},↑csize );

   Allocate_Label( ↑label,↑label+2 );

   Else_Statement( ↓↑new_label ) =
      Code( ↓{"JMP",new_label+1},↑csize ),
      Assign_Label( ↓new_label,↑new_label,↑<reference> ),
         Statement;

   Assign_Label( ↓new_label,↑new_label+1,↑[new_label → csize] ) =
      ↓CODE( ↓{"LABEL",new_label} );

   End =
      ↑DOT,
         ↓CODE( ↓{"HLT",0} );

      /*************************************************************/
      /*                                                           */
      /*   The specification of a simple Optimiser                 */
      /*                                                           */
      /*************************************************************/

RULES
   Optimiser =
      Title, Optimisation;

   Title =
      ↑FILE_NAME( ↑name ), ↓FILE_NAME( ↓name );

   Optimisation =
      ↑CODE( ↑code1 ), Optimise( ↓↑code1 ), ↓CODE( ↓code1 );

   Optimise( ↓↑code1 ) =
      {
         ↑CODE( ↑code2 ),
            (
               Fold
                  (
                     ↓fold1[code1.function][code2.function],
                     ↓( code1.operand = code2.operand ),
                     ↑code1
                  )
            |
```

```
            Fold
               (
                  ↓fold2[code1.function][code2.function],
                  ↓code1.operand,
                  ↑code1
               )
         |
            Fold
               (
                  ↓fold3[code1.function][code2.function],
                  ↓code1.operand + code2.operand * fold4[code2.function],
                  ↑code1
               )
         |
            Output( ↓code1,↓code2,↑code1,↑csize )
      )
   };

Fold( ↓function,↓operand,↑{function,operand} );

Output( ↓code1,↓code2,↑code2,↑csize+size[code1.function] ) =
   [ Label( ↓code1.function="LABEL",↓code1.operand,↑<reference> ) ],
      ↓CODE( ↓code1 );

Label( ↓function,↓operand,↑[operand → csize ] );


    /*************************************************************/
    /*                                                           */
    /*   The specification of a simple Loader                    */
    /*                                                           */
    /*************************************************************/

RULES
   Loader =
      Title, Fixup;

   Title =
      ↑FILE_NAME( ↑name ), ↓FILE_NAME( ↓name );

   Fixup =
      {
         ↑CODE( ↑code ),
         [ Final( ↓code.function,↓code.operand ) ]
      };
```

```
    Final( ↓function,↓operand ) =
        ( ↓FINAL1( ↓class1[function] ), ↓FINAL2( ↓operand ) ) |
        ( ↓FINAL1( ↓class2[function] ) ) |
        ( ↓FINAL1( ↓class3[function] ), ↓FINAL2( ↓reference[operand] ) );

END
```

Appendix 5

A

Specification of

Pascal

Introduction

This appendix contains a Compiler Construction Language (CCL) specification of a translator which is capable of translating a large subset of the International Standards Organisation (ISO) Pascal (Level 0) [BSI 1982] to P-code [Ammann 1981]. A discussion of this specification appears in Chapter 8 of this thesis.

A Specification for Pascal

```
TITLE Pascal

       /*************************************************************/
       /*                                                           */
       /*       Compiler constants                                  */
       /*                                                           */
       /*************************************************************/

CONSTANTS
   BOOLEAN_MIN  = 0;
   CHAR_MIN     = 0;
   INTEGER_MIN  = -1000000;
   REAL_MIN     = -10000000;

   BOOLEAN_MAX  = 1;
   CHAR_MAX     = 255;
   INTEGER_MAX  = 1000000;
   REAL_MAX     = 10000000;

   BOOLEAN_SIZE = 1;
   CHAR_SIZE    = 1;
   INTEGER_SIZE = 4;
   REAL_SIZE    = 4;
   POINTER_SIZE = 4;
   SET_SIZE     = 4;

   MAX_ARRAY    = 100000; /* Trap things like 'ARRAY[INTEGER] OF'  */
```

```
        /*************************************************************/
        /*                                                           */
        /*       Pascal keywords                                     */
        /*                                                           */
        /*************************************************************/

INPUT TOKENS

    Program       = CONSTANT "PROGRAM";
    Label         = CONSTANT "LABEL";
    Const         = CONSTANT "CONST";
    Type          = CONSTANT "TYPE";
    Array         = CONSTANT "ARRAY";
    Of            = CONSTANT "OF";
    Packed        =          "PACKED";
    Record        = CONSTANT "RECORD";
    Set           = CONSTANT "SET";
    Var           =          "VAR";
    Function      = CONSTANT "FUNCTION";
    Procedure     = CONSTANT "PROCEDURE";
    Forward       =          "FORWARD";
    Begin         = CONSTANT "BEGIN";
    Case          = CONSTANT "CASE";
    For           = CONSTANT "FOR";
    To            =          "TO";
    Downto        =          "DOWNTO";
    Goto          = CONSTANT "GOTO";
    If            = CONSTANT "IF";
    Then          = CONSTANT "THEN";
    Else          = CONSTANT "ELSE";
    Repeat        = CONSTANT "REPEAT";
    Until         = CONSTANT "UNTIL";
    While         = CONSTANT "WHILE";
    With          = CONSTANT "WITH";
    Do            = CONSTANT "DO";
    End           = CONSTANT "END";

        /*************************************************************/
        /*                                                           */
        /*       Pascal symbols                                      */
        /*                                                           */
        /*************************************************************/

    Bracket1      = CONSTANT "(";
    Bracket2      = CONSTANT ")";
    Bracket3      = CONSTANT "[";
    Bracket4      = CONSTANT "]";

    Colon         = CONSTANT ":";
    Comma         = CONSTANT ",";
    Dot           = CONSTANT ".";
    Dot_dot       = CONSTANT "..";
    Nil           = CONSTANT "NIL";
    Pointer       = CONSTANT "^";
    Semi_colon    = CONSTANT ";";
```

```
    Assign          = CONSTANT ":=";
    Relation        = "=" | "<>" | "<" | ">" | "<=" | ">=" | "IN";
    Sum             = "+" | "-" | "OR";
    Term            = "*" | "/" | "MOD" | "DIV" | "AND";
    Not             = CONSTANT "NOT";

        /*************************************************************/
        /*                                                           */
        /*       Special symbols                                     */
        /*                                                           */
        /*************************************************************/

    Character       = STRUCTURE
                        {
                           CONSTANT       : "'";
                           value          : CHARACTER;
                           CONSTANT       : "'";
                        };
    Integer         = INTEGER;
    Real            = REAL; /* Clash between '1.2' and '1..2' */

    Identifier      = ('A'→'Z'|'a'→'z'), {'A'→'Z'|'a'→'z'|'0'→'9'|"_"};
    String          = "\'", {'\0'→'&'|'('→'\377'}, "\'";

    VOID            = " " | "\n" | "\t";

        /*************************************************************/
        /*                                                           */
        /*       P-code output symbols                               */
        /*                                                           */
        /*************************************************************/

OUTPUT TOKENS

    Constant        = STRUCTURE
                        {
                           CONSTANT       : "CONSTANT\t\t\t";
                           label          : INTEGER;
                           CONSTANT       : "\n";
                        };

    Label1          = STRUCTURE
                        {
                           CONSTANT       : "LABEL\t\t\t";
                           label          : INTEGER;
                           CONSTANT       : "\n";
                        };
```

```
    Instruction0 = STRUCTURE
                        {
                           CONSTANT        : "\t";
                           function        : TEXT;
                           CONSTANT        : "\n";
                        };

    Instruction1 = STRUCTURE
                        {
                           CONSTANT        : "\t";
                           function        : TEXT;
                           CONSTANT        : "\t\t";
                           operand1        : REAL;
                           CONSTANT        : "\n";
                        };

    Instruction2 = STRUCTURE
                        {
                           CONSTANT        : "\t";
                           function        : TEXT;
                           CONSTANT        : "\t";
                           operand1        : REAL;
                           CONSTANT        : "\t";
                           operand2        : REAL;
                           CONSTANT        : "\n";
                        };

    InstructionS = STRUCTURE
                        {
                           CONSTANT        : "\t";
                           function        : TEXT;
                           CONSTANT        : "\t\t";
                           string          : TEXT;
                           CONSTANT        : "\n";
                        };

        /*************************************************************/
        /*                                                           */
        /*      Structure of label entries                           */
        /*                                                           */
        /*************************************************************/

TYPES

    LABEL         = STRUCTURE
                        {
                           defined         : TEXT;
                           label           : INTEGER;
                        };

    LABELS        = TABLE [ LABEL ];
```

```
    /*************************************************************/
    /*                                                           */
    /*       Structure of the symbol table                       */
    /*                                                           */
    /*************************************************************/

KIND          = UNION
                {
                   procedure      : PROCEDURE;
                   type           : INTEGER;
                   value          : REAL;
                   variable       : VARIABLE;
                };

TYPE          = UNION
                {
                   array          : ARRAY;
                   ordinal        : ORDINAL;
                   pointer        : TEXT;
                   record         : RECORD;
                   set            : SET;
                };

SYMBOL        = STRUCTURE
                {
                   kind           : KIND;
                   type           : TYPE;
                };

SYMBOLS       = TABLE [ SYMBOL ];

    /*************************************************************/

ORDINAL       = STRUCTURE
                {
                   basic          : TEXT;
                   lower          : INTEGER;
                   upper          : INTEGER;
                };

STOREMAP      = STRUCTURE
                {
                   level          : INTEGER;
                   size           : INTEGER;
                };

    /*************************************************************/

ARRAY         = STRUCTURE
                {
                   packing        : TEXT;
                   index          : ORDINAL;
                   type           : TYPE;
                   size           : INTEGER;
                };
```

```
PROCEDURE     = STRUCTURE
                    {
                       defined          : TEXT;
                       label            : INTEGER;
                       parameters       : SYMBOLS;
                       storemap         : STOREMAP;
                    };

RECORD        = STRUCTURE
                    {
                       packing          : TEXT;
                       fields           : SYMBOLS;
                       size             : INTEGER;
                    };

SET           = STRUCTURE
                    {
                       packing          : TEXT;
                       type             : ORDINAL;
                    };

VARIABLE      = STRUCTURE
                    {
                       class            : TEXT;
                       offset           : INTEGER;
                       level            : INTEGER;
                       name             : TEXT;
                       size             : INTEGER;
                       withoffset       : INTEGER;
                    };

    /*************************************************************/
    /*                                                           */
    /*      Other type definitions                               */
    /*                                                           */
    /*************************************************************/

REAL_TABLE    = TABLE [ REAL ];
REAL_LOOKUP   = TABLE [ REAL_TABLE ];

TEXT_TABLE    = TABLE [ TEXT ];
TEXT_LOOKUP   = TABLE [ TEXT_TABLE ];
TEXT_SEARCH   = TABLE [ TEXT_LOOKUP ];
```

```
        /*************************************************************/
        /*                                                           */
        /*      Translator variables                                 */
        /*                                                           */
        /*************************************************************/

VARIABLES

    ARRAY array;

    Character character;

    INTEGER label,label1,label2;
    INTEGER unique_number = 1;
    INTEGER value,value1,value2;
    INTEGER size,offset,max_offset,next_offset;
    INTEGER lower,upper,setbits;

    KIND kind;

    LABEL locallabel;

    LABELS labelenv,locallabels,nonlocallabels;

    ORDINAL ordinal;

    PROCEDURE procedure;

    REAL_LOOKUP sign =
        [
            "+" → [ "INTEGER" → 1, "REAL" → 1 ],
            "-" → [ "INTEGER" → -1, "REAL" → -1 ]
        ];

    REAL_TABLE caselist;

    RECORD record;

    STOREMAP storemap;

    SYMBOL parameter,symbol;

    SYMBOLS env,fields,locals,nonlocals,parameters,withenv;
    SYMBOLS initial =
        [
            "BOOLEAN" → {type(BOOLEAN_SIZE),
                ordinal({"BOOLEAN",BOOLEAN_MIN,BOOLEAN_MAX})},
            "CHAR" → {type(CHAR_SIZE),
                ordinal({"CHAR",CHAR_MIN,CHAR_MAX})},
            "INTEGER" → {type(INTEGER_SIZE),
                ordinal({"INTEGER",INTEGER_MIN,INTEGER_MAX})},
```

```
      /*                                                    */
      /* Real is counted as an ordinal in order to keep */
      /* things simple.  Any attempt to use REAL as an  */
      /* ordinal is trapped by the syntax               */
      /*                                                    */
      "REAL" → {type(REAL_SIZE),
         ordinal({"REAL",REAL_MIN,REAL_MAX})},

      "FALSE" → {value(0),
         ordinal({"BOOLEAN",BOOLEAN_MIN,BOOLEAN_MAX})},
      "TRUE" → {value(1),
         ordinal({"BOOLEAN",BOOLEAN_MIN,BOOLEAN_MAX})}
   ];

TEXT defined,enumname,packing,var;
TEXT name,operator,text,text1,text2;

TEXT_TABLE ordinal_type =
   [
      "BOOLEAN" → "BOOLEAN",
      "CHAR" → "CHAR",
      "INTEGER" → "INTEGER"
      /* Enums need to be added to      */
      /* this list as they are declared */
   ];
TEXT_TABLE load_constant =
   [
      "BOOLEAN" → "LDCB",
      "CHAR" → "LDCC",
      "INTEGER" → "LDCI",
      "REAL" → "LDCR",
      "NULLPOINTER" → "LDCA"
   ];
TEXT_TABLE load_value =
   [
      "BOOLEAN" → "INDB",
      "CHAR" → "INDC",
      "INTEGER" → "INDI",
      "REAL" → "INDR"
      /* Enums need to be added to      */
      /* this list as they are declared */
   ];
TEXT_TABLE pointers =
   [
      ":=" → "STOA",
      "=" → "EQUA",
      "<>" → "NEQA",
      "NULLPOINTER" → "NULLPOINTER"
   ];
```

```
TEXT_TABLE sets =
   [
      ":=" → "STOS",
      "=" → "EQUS",
      "<>" → "NEQS",
      ">=" → "GEQS",
      "<=" → "LEQS",
      "+" → "UNI",
      "-" → "DIF",
      "*" → "INT",
      "NULLSET" → "NULLSET"
   ];
TEXT_TABLE strings = [ /* Initially empty */ ];
TEXT_TABLE table;
TEXT_TABLE uniquename =
   [
      "$1","$2","$3","$4","$5","$6","$7","$8","$9","$10",
      "$11","$12","$13","$14","$15","$16","$17","$18","$19","$20",
      "$21","$22","$23","$24","$25","$26","$27","$28","$29","$30"
      /* And so on.  This is soon to be replaced by a function */
   ];

TEXT_LOOKUP float =
   [
      "INTEGER" → [ "REAL" → "FLO" ],
      "REAL" → [ "INTEGER" → "FLT" ]
   ];
TEXT_LOOKUP unary =
   [
      "+" → [ "INTEGER" → "NOP","REAL" → "NOP" ], /* No operation */
      "-" → [ "INTEGER" → "NGI","REAL" → "NGR" ],
      "NOT" → [ "BOOLEAN" → "NOT" ],
      /* Enums need to be added to      */
      /* this list as they are declared */
      "DOWNTO" → [ "INTEGER" → "DEC" ],
      "TO" → [ "INTEGER" → "INC" ]
   ];

TEXT_SEARCH arithmetic =
   [
      ":=" →
         [
            "BOOLEAN" → [ "BOOLEAN" → "STOB" ],
            "CHAR" → [ "CHAR" → "STOC" ],
            "INTEGER" → [ "INTEGER" → "STOI" ],
            "REAL" → [ "INTEGER" → "STOR","REAL" → "STOR" ]
         ],
```

```
"=" →
    [
        "BOOLEAN" → [ "BOOLEAN" → "EQUB" ],
        "CHAR" → [ "CHAR" → "EQUC" ],
        "INTEGER" → [ "INTEGER" → "EQUI" ],
        "REAL" → [ "INTEGER" → "EQUR","REAL" → "EQUR" ]
    ],
"<>" →
    [
        "BOOLEAN" → [ "BOOLEAN" → "NEQB" ],
        "CHAR" → [ "CHAR" → "NEQC" ],
        "INTEGER" → [ "INTEGER" → "NEQI" ],
        "REAL" → [ "INTEGER" → "NEQR","REAL" → "NEQR" ]
    ],
"<" →
    [
        "BOOLEAN" → [ "BOOLEAN" → "LESB" ],
        "CHAR" → [ "CHAR" → "LESC" ],
        "INTEGER" → [ "INTEGER" → "LESI" ],
        "REAL" → [ "INTEGER" → "LESR","REAL" → "LESR" ]
    ],
"<=" →
    [
        "BOOLEAN" → [ "BOOLEAN" → "LEQB" ],
        "CHAR" → [ "CHAR" → "LEQC" ],
        "INTEGER" → [ "INTEGER" → "LEQI" ],
        "REAL" → [ "INTEGER" → "LEQR","REAL" → "LEQR" ]
    ],
">" →
    [
        "BOOLEAN" → [ "BOOLEAN" → "GRTB" ],
        "CHAR" → [ "CHAR" → "GRTC" ],
        "INTEGER" → [ "INTEGER" → "GRTI" ],
        "REAL" → [ "INTEGER" → "GRTR","REAL" → "GRTR" ]
    ],
">=" →
    [
        "BOOLEAN" → [ "BOOLEAN" → "GEQB" ],
        "CHAR" → [ "CHAR" → "GEQC" ],
        "INTEGER" → [ "INTEGER" → "GEQI" ],
        "REAL" → [ "INTEGER" → "GEQR","REAL" → "GEQR" ]
    ],
```

```
      "+" →
         [
            "INTEGER" → [ "INTEGER" → "ADI" ],
            "REAL" → [ "INTEGER" → "ADR","REAL" → "ADR" ]
         ],
      "-" →
         [
            "INTEGER" → [ "INTEGER" → "SBI" ],
            "REAL" → [ "INTEGER" → "SBR","REAL" → "SBR" ]
         ],
      "*" →
         [
            "INTEGER" → [ "INTEGER" → "MPI" ],
            "REAL" → [ "INTEGER" → "MPR","REAL" → "MPR" ]
         ],
      "/" →
         [
            "INTEGER" → [ "INTEGER" → "DVI" ],
            "REAL" → [ "INTEGER" → "DVR","REAL" → "DVR" ]
         ],
      "AND" →
         [
            "BOOLEAN" → [ "BOOLEAN" → "AND" ]
         ],
      "OR" →
         [
            "BOOLEAN" → [ "BOOLEAN" → "IOR" ]
         ],
      "DIV" →
         [
            "INTEGER" → [ "INTEGER" → "DVI" ]
         ],
      "MOD" →
         [
            "INTEGER" → [ "INTEGER" → "MOD" ]
         ]
      /* Enums need to be added to      */
      /* this list as they are declared */
   ];

TYPE index,type,type1,type2;
TYPE boolean_type = ordinal({"BOOLEAN",BOOLEAN_MIN,BOOLEAN_MAX});
TYPE char_type = ordinal({"CHAR",CHAR_MIN,CHAR_MAX });
TYPE integer_type = ordinal({"INTEGER",INTEGER_MIN,INTEGER_MAX });
TYPE real_type = ordinal({"REAL",REAL_MIN,REAL_MAX });
TYPE null_pointer = ordinal({"NULLPOINTER",0,0});
TYPE null_set = ordinal({"NULLSET",0,0});

VARIABLE variable;
```

```
        /*************************************************************/
        /*                                                           */
        /*       Main program block                                  */
        /*                                                           */
        /*************************************************************/

RULES

   Start =
      Program,
         <<Dot,Dot,"Malformed main block">>
            (
               ↓Instruction1( ↓{ "UJP",0 } ),
               Block( ↓initial,↓[],↓[],↓{0,0},↓0 ),
               ↑Dot
            );

   Program =
      <<Semi_colon,Semi_colon,"Malformed PROGRAM header">>
         (
            ↑Program, ↑Identifier( ↑name ), /* ignore name */
               Program_parameters, ↑Semi_colon
         );

   Program_parameters =
      ↑Bracket1, ↑Bracket2;

   Block( ↓nonlocals,↓nonlocallabels,↓locals,↓storemap,↓label ) =
      Label_declarations( ↓[],↑locallabels ),
      Constant_declarations( ↓nonlocals,↓↑locals ),
      Type_declarations( ↓nonlocals,↓↑locals ),
      Variable_declarations( ↓nonlocals,↓↑locals,↓storemap ),
      Procedures_and_statements
         (
            ↓nonlocals ∪ locals, ↓nonlocallabels ∪ locallabels,
            ↓locallabels,↓storemap,↓label
         ),
      ↓Instruction0( ↓{ "RTS" } );
```

```
    /*************************************************************/
    /*                                                           */
    /*      Declare local labels                                 */
    /*                                                           */
    /*************************************************************/

Label_declarations( ↓locallabels,↑locallabels ) =
   [
      ↑Label,
      <<Semi_colon,Semi_colon,"Malformed label list">>
         (
            Label_list( ↓↑locallabels ),
            ↑Semi_colon
         )
   ];

Label_list( ↓↑locallabels ) =
   <<Semi_colon,Semi_colon,"Malformed label">>
   Label_declaration( ↑<locallabels> ),
      {
         ↑Comma,
         <<Semi_colon,Semi_colon,"Malformed label">>
         Label_declaration( ↑<locallabels> )
      };

Label_declaration( ↑[label1 → { "UNDEFINED",label2 }] ) =
   ↑Integer( ↑label1 ), Unique_number( ↑label2 );

    /*      Allocate unique internal label numbers               */
    /*    (This will soon be replaced by a function)             */

Unique_number( ↑unique_number ) =
   Allocate_number( ↑unique_number );

Allocate_number( ↑unique_number + 1 );
```

```
    /*************************************************************/
    /*                                                           */
    /*       Declare local constants                             */
    /*                                                           */
    /*************************************************************/

Constant_declarations( ↓nonlocals,↓↑locals ) =
   [
      ↑Const,
      <<Semi_colon,Semi_colon,"Invalid constant expression">>
            Constant_list( ↓nonlocals,↓↑locals )
   ];

Constant_list( ↓env,↓↑locals ) =
   Constant_declaration( ↓env ∪ locals,↓0,↓integer_type,↑<locals> ),
     { Constant_declaration( ↓env ∪ locals,↓0,↓integer_type,↑<locals> ) };

Constant_declaration( ↓env,↓value,↓type,↑[name → {value(value),type}] ) =
   ↑Identifier( ↑name ),
      <<"Missing '=' symbol">> Equals( ↑operator ),
         <<Semi_colon,Semi_colon,"Invalid constant expression">>
            (
               Signed_constant( ↓env,↑value,↑type ) |
               Constant_or_identifier( ↓env,↑value,↑type )
            ),
         ↑Semi_colon;

Equals( ↑operator="=" ) =
   ↑Relation( ↑operator );

Signed_constant
   (
      ↓env,
      ↑value(sign[operator][type.ordinal.basic]*value,
      ↑type
   ) =
↑Sum( ↑operator ),
Constant_or_identifier( ↓env,↑value,↑type );

Constant_or_identifier( ↓env,↑value,↑type ) =
   Constant_identifier( ↓env,↑value,↑type ) |
   Unsigned_constant( ↑value,↑type );

Constant_identifier( ↓env,↑env[name].kind.value,↑env[name].type ) =
   ↑Identifier( ↑name );
```

```
    /*************************************************************/
    /*                                                           */
    /*       Define unsigned constant values                     */
    /*                                                           */
    /*************************************************************/

Unsigned_constant( ↑value,↑type ) =
   Character_constant( ↑value,↑type ) |
   Integer_constant( ↑value,↑type ) |
   Nullpointer( ↑value,↑type ) |
   Real_constant( ↑value,↑type ) |
   String_constant( ↑value,↑<strings>,↑type );

Character_constant( ↑character.value,↑char_type ) =
   ↑Character( ↑character );

Integer_constant( ↑value,↑integer_type ) =
   ↑Integer( ↑value );

Nullpointer( ↑0,↑null_pointer ) =
   ↑Nil;

Real_constant( ↑value,↑real_type ) =
   ↑Real( ↑value );

String_constant
   (
      ↑value, ↑[value → text],
      ↑array({"PACKED",{"INTEGER",1,LENGTH(text)},
         char_type,LENGTH(text)*CHAR_SIZE})
   ) =
Unique_number( ↑value ), ↑String( ↑text );

    /*************************************************************/
    /*                                                           */
    /*       Declare new types                                   */
    /*                                                           */
    /*************************************************************/

Type_declarations( ↓nonlocals,↓↑locals ) =
   [
      ↑Type,
      <<Semi_colon,Semi_colon,"Invalid type expression">>
            Type_list( ↓nonlocals,↓↑locals )
   ];
```

```
Type_list( ↓env,↓↑locals ) =
   Type_declaration
      (
         ↓env ∪ locals, ↓INTEGER_SIZE, ↓integer_type,
         ↓↑locals, ↑<locals>
      ),
   {
      Type_declaration
         (
            ↓env ∪ locals, ↓INTEGER_SIZE, ↓integer_type,
            ↓↑locals, ↑<locals>
         )
   };

Type_declaration
   (
      ↓env, ↓size, ↓type,
      ↓↑locals,
      ↑[name → {type(size),type}]
   ) =
↑Identifier( ↑name ),
   <<"Missing '=' symbol">> Equals( ↑operator ),
      <<Semi_colon,Semi_colon,"Invalid type expression">>
         Type( ↓env,↑<locals>,↑size,↑type ),
      ↑Semi_colon;

    /*************************************************************/
    /*                                                           */
    /*      Parse a complete Pascal type specification           */
    /*                                                           */
    /*************************************************************/

Type( ↓env,↑locals,↑size,↑type ) =
   Packed_type( ↓env,↑locals,↑size,↑type ) |
   Ordinal_type( ↓env,↑locals,↑size,↑type ) |
   Pointer_type( ↑locals,↑size,↑type );

Packed_type( ↓env,↑locals,↑size,↑type ) =
   Packing( ↓"UNPACKED",↑packing ),
      (
         Array_type( ↓env,↓packing,↑locals,↑size,↑type ) |
         Record_type( ↓env,↓packing,↑locals,↑size,↑type ) |
         Set_type( ↓env,↓packing,↑locals,↑size,↑type )
      );

Packing( ↓packing,↑packing ) =
   [ ↑Packed( ↑packing ) ];
```

```
    /*************************************************************/
    /*                                                           */
    /*       A array type declaration                            */
    /*                                                           */
    /*************************************************************/

Array_type( ↓env,↓packing,↑locals,↑size,↑type ) =
   ↑Array,
      Subscript_list( ↓env,↓packing,↑locals,↑size,↑type );

Subscript_list( ↓env,↓packing,↑locals,↑size,↑type ) =
   ↑Bracket3,
      Subscript_element( ↓env,↓packing,↑locals,↑size,↑type );

Subscript_element
   (
      ↓env, ↓packing,
      ↑locals, ↑size*(index.ordinal.upper-index.ordinal.lower),
      ↑array({packing,index.ordinal,type,
         size*(index.ordinal.upper-index.ordinal.lower)<MAX_ARRAY})
   ) =
Ordinal_type( ↓env,↑locals,↑size/*Not needed*/,↑index ),
   (
      ↑Comma,
         Subscript_element( ↓env,↓packing,↑<locals>,↑size,↑type )
   |
      ↑Bracket4, ↑Of,
         Type( ↓env,↑<locals>,↑size,↑type )
   );

    /*************************************************************/
    /*                                                           */
    /*       A record type declaration                           */
    /*                                                           */
    /*************************************************************/

Record_type
   (
      ↓env,↓packing,
      ↑locals,↑size,↑record({packing,fields,size})
   ) =
↑Record,
   Record_structure( ↓env,↑locals,↑size,↑fields ),
<<End,Semi_colon,"Misplaced END in RECORD">> ↑End;

Record_structure( ↓env,↑locals,↑offset,↑fields ) =
   Field_list( ↓env,↓0,↑locals,↑offset,↑fields ),
      [ Variant_list( ↓env,↓offset,↑locals,↑offset,↑<fields> ) ];
```

```
Field_list( ↓env,↓offset,↑locals,↑offset,↑fields ) =
   Field
      (
         ↓env, ↓offset,
         ↑locals, ↑offset ,↑size, ↑type, ↑fields, ↑<fields>
      ),
   {
      Field
         (
            ↓env, ↓offset,
            ↑locals, ↑offset ,↑size, ↑type, ↑<fields>, ↑<fields>
         )
   };

Field
   (
      ↓env, ↓offset,
      ↑locals, ↑offset + size, ↑size, ↑type, ↑fields,
      ↑[name → {variable({"FIELD",offset,0,name,size,0}),type}]
   ) =
↑Identifier( ↑name ),
   (
      ↑Comma,
      Field( ↓env,↓offset,↑locals,↑offset,↑size,↑type,↑fields,↑<fields> )
   |
      ↑Colon,
      Field_type( ↓env,↑locals,↑size,↑type,↑fields ),
      ↑Semi_colon
   );

Field_type( ↓env,↑locals,↑size,↑type,↑[] ) =
   Type( ↓env,↑locals,↑size,↑type );

Variant_list( ↓env,↓offset,↑locals,↑max_offset,↑fields ) =
   ↑Case,
      Variant_tag( ↓env,↓offset,↑offset,↑type,↑fields ),
   ↑Of,
      Variants( ↓env,↓offset,↓type,↑locals,↑max_offset,↑<fields> );
```

```
Variant_tag
   (
      ↓env, ↓offset,
      ↑offset + size, ↑type, ↑[]
   ) =
(
   ( ↑Identifier( ↑name ), ↑Colon ) |
   Unique_name( ↑name )
),
   Variant_type( ↓env,↑text,↑size,↑type );

Unique_name( ↑<<uniquename> );  /* Replaced soon by a function */

Variant_type
   (
      ↓env,
      ↑ordinal_type[ env[name].type.ordinal.basic ],
      ↑env[name].kind.type,                  /* Test key is a type name */
      ↑env[name].type
   ) =
↑Identifier( ↑name );

Variants( ↓env,↓offset,↓type,↑locals,↑max_offset,↑fields ) =
   Variant( ↓env,↓offset,↓type,↑locals,↑max_offset,↑fields ),
      {
         Variant( ↓env,↓offset,↓type,↑<locals>,↑next_offset,↑<fields> ),
         Max_offset( ↓max_offset,↓next_offset,↑max_offset )
      };

Variant( ↓env,↓offset,↓type,↑locals,↑offset,↑fields ) =
   Case_list( ↓env,↓type,↑fields ),
      ↑Colon, ↑Bracket1,
         Field_list( ↓env,↓offset,↑locals,↑offset,↑fields ),
      ↑Bracket2, ↑Semi_colon;

Case_list( ↓env,↓type,↑fields ) =
   ↑Case,
      Case_item( ↓env,↓type,↑text,↑fields ),
         {
            ↑Comma,
            Case_item( ↓env,↓type,↑text,↑<fields> )
         };
```

```
Case_item
   (
      ↓env, ↓type1,
      ↑type1.ordinal.basic = type2.ordinal.basic,
      ↑[value → {type(0),pointer("CASE")}]   /* Prevent tag reuse */
   ) =
Constant_or_identifier( ↓env,↑value,↑type2 );

Max_offset( ↓max_offset,↓offset,↑max_offset ) = /* Function soon */
   [ Max_offset( ↓offset>max_offset,↓max_offset,↑max_offset ) ];

    /*************************************************************/
    /*                                                           */
    /*      A set type declaration                               */
    /*                                                           */
    /*************************************************************/

Set_type
   (
      ↓env, ↓packing,
      ↑locals,
      ↑SET_SIZE >= (type.ordinal.upper-type.ordinal.lower)/8,
      ↑set({packing,{ordinal_type[type.ordinal.basic],
            type.ordinal.lower,type.ordinal.upper }})
   ) =
↑Set, ↑Of,
   Ordinal_type( ↓env,↑locals,↑size,↑type );

    /*************************************************************/
    /*                                                           */
    /*      Ordinal type declaration                             */
    /*                                                           */
    /*************************************************************/

Ordinal_type( ↓env,↑locals,↑size,↑type ) =
   Identifier_or_range( ↓env,↑locals,↑size,↑type ) |
   Enumeration_list( ↑locals,↑size,↑type );

Identifier_or_range( ↓env,↑[],↑kind.type,↑type ) =
   (
      Identifier( ↓env,↑kind,↑type ),
      [ Subrange( ↓env,↓kind.value,↓type,↑kind,↑type ) ]
   )
|
   (
      Constant_or_identifier( ↓env,↑value,↑type ),
      Subrange( ↓env,↓value,↓type,↑kind,↑type )
   );
```

```
Identifier( ↓env,↑env[name].kind,↑env[name].type ),
   ↑Identifier( ↑name );

Subrange
   (
      ↓env, ↓value1, ↓type1,
      ↑type(INTEGER_SIZE),
      ↑ordinal({ordinal_type[type1.ordinal.basic=type2.ordinal.basic],
            value1<=value2,value2 })
   ) =
↑Dot_dot, Constant_or_identifier( ↓env,↑value2,↑type2 );

Enumeration_list
   (
      ↑locals, ↑INTEGER_SIZE,
      ↑ordinal({enumname,0,value-1})
   ) =
↑Bracket1,
   Enum_name
      (
         ↑enumname, ↑<arithmetic>, ↑<load_value>,
       ↑<ordinal_type>,  ↑<unary>
      ),
   Enumeration( ↓0,↓enumname,↑locals,↑value ),
      {
         ↑Comma,
         Enumeration( ↓value,↓enumname,↑<locals>,↑value )
      },
↑Bracket2;

Enumeration
   (
      ↓value, ↓enumname,
      ↑[name → {value(value),ordinal({enumname,0,value})}],
      ↑value+1
   ) =
↑Identifier( ↑name );
```

```
Enum_name
   (
      ↑enumname,
      ↑[
         "=" → [ enumname → [ enumname → "EQUI" ] ],
         "<>" → [ enumname → [ enumname → "NEQI" ] ],
         "<" → [ enumname → [ enumname → "LESI" ] ],
         "<=" → [ enumname → [ enumname → "LEQI" ] ],
         ">" → [ enumname → [ enumname → "GRTI" ] ],
         ">=" → [ enumname → [ enumname → "GEQI" ] ]
       ],
      ↑[enumname → "INDI"],
      ↑[enumname → enumname],
      ↑[
         "DOWNTO" → [ enumname → "DEC" ],
         "TO" → [ enumname → "INC" ]
       ]
   ) =
/* Get a unique name for this enum definition */
/* and add it to the ordinal types list       */
Unique_name( ↑enumname );

    /*************************************************************/
    /*                                                           */
    /*      Pointer type declaration                             */
    /*                                                           */
    /*************************************************************/

Pointer_type( ↑[],↑POINTER_SIZE,↑pointer(name) ) =
   ↑Pointer, ↑Identifier( ↑name );

    /*************************************************************/
    /*                                                           */
    /*      Declare new variables                                */
    /*                                                           */
    /*************************************************************/

Variable_declarations( ↓nonlocals,↓↑locals,↓storemap ) =
   [
      ↑Var( ↑var ),
      <<Semi_colon,Semi_colon,"Invalid var declaration">>
      Variable_list( ↓nonlocals,↓↑locals,↓storemap )
   ];

Variable_list( ↓env,↓↑locals,↓storemap ) =
   Variable_declaration( ↓env ∪ locals,↓↑locals,↓↑storemap ),
      { Variable_declaration( ↓env ∪ locals,↓↑locals,↓↑storemap ) };
```

```
Variable_declaration( ↓env,↓↑locals,↓↑storemap ) =
   Variables( ↓env,↓storemap,↓↑locals,↑storemap,↑size,↑type,↑<locals> );

Variables
   (
      ↓env, ↓storemap, ↓↑locals,
      ↑storemap, ↑size, ↑type, ↑[]
   ) =
Variable_name( ↑name,↑size,↑type ),
<<Semi_colon,Semi_colon,"Invalid variable declaration">>
   Variable_type
      (
         ↓env, ↓name, ↓storemap, ↓↑locals,
         ↑storemap, ↑size, ↑type, ↑<locals>
      );

Variable_name( ↑name,↑INTEGER_SIZE,↑integer_type ) =
   ↑Identifier( ↑name );

Variable_type
   (
      ↓env, ↓name, ↓storemap, ↓↑locals,
      ↑{storemap.level,storemap.size + size},
      ↑size, ↑type,
      ↑[ name → {variable(
         {"VAR",storemap.size,storemap.level,name,size,0}),type}]
   ) =
(
   ↑Comma,
      Variable( ↓env,↓storemap,↓↑locals,↑storemap,↑size,↑type,↑<locals> )
|
   ↑Colon,
   Type( ↓env,↑<locals>,↑size,↑type ),
   ↑Semi_colon
);
```

```
    /*************************************************************/
    /*                                                           */
    /*      Declare functions and procedures and statements      */
    /*                                                           */
    /*************************************************************/

Procedures_and_statements
   (
      ↓env,↓labelenv,↓locallabels,
      ↓storemap,↓label
   ) =
Procedure_declarations( ↓↑env,↓labelenv,↓storemap ),
<<End,End,"Malformed block">>
   (
      ↓Label1( ↓{ label } ),
      Compound_block( ↓env,↓labelenv,↓↑locallabels,↓storemap )
   ),
Verify_labels( ↓locallabels );

Procedure_declarations( ↓↑env,↓labelenv,↓storemap ) =
   { Procedure_declaration( ↓↑env,↓labelenv,↓storemap ) };

Procedure_declaration( ↓↑env,↓labelenv,↓storemap ) =
      (
         Function( ↓↑env,↓labelenv,↓storemap ) |
         Procedure( ↓↑env,↓labelenv,↓storemap )
      );

    /*************************************************************/
    /*                                                           */
    /*      Deal with function declarations                      */
    /*                                                           */
    /*************************************************************/

Function( ↓↑env,↓labelenv,↓storemap ) =
   ↑Function, ↑Identifier( ↑name ),
   <<Semi_colon,Semi_colon,"Malformed function header">>
      (
         Predefined_function( ↓env,↓labelenv,↓↑env[name],↑<env> ) |
         Undefined_function( ↓name,↓env,↓labelenv,↓storemap,↑<env> )
      );
```

```
Predefined_function( ↓env,↓labelenv,↓↑symbol,↑[] ) =
   Forward_function
      (
         /* Check function correctly defined */
         ↓symbol.kind.procedure.defined,
         ↓symbol.type.ordinal.basic <> "VOID",
         ↑symbol.kind.procedure.defined
      ),
   <<End,End,"Malformed block">>
      Block
         (
            ↓env, ↓labelenv,
            ↓symbol.kind.procedure.parameters,
            ↓symbol.kind.procedure.storemap,
            ↓symbol.kind.procedure.label
         );

Forward_function( ↓defined,↓text,↑"DEFINED" <> defined ) =
   <<"Semi colon missing">> ↑Semi_colon;

Undefined_function
   (
      ↓name, ↓env, ↓labelenv, ↓storemap,
      ↑[name → {procedure({defined,label,parameters,storemap}),type}]
   ) =
Parameters
   (
      ↓env, ↓{storemap.level+1,0}, ↓[],
      ↑defined, ↑storemap, ↑parameters
   ),
↑Colon,
Result_type( ↓env,↑<parameters>,↑size,↑type ),
↑Semi_colon, Unique_number( ↑label ),
<<Semi_colon,Semi_colon,"Malformed directive or block">>
   (
      ↑Forward( ↑defined ) |
      Block( ↓env,↓labelenv,↓parameters,↓storemap,↓label )
   ),
↑Semi_colon;

Result_type( ↓env,↑locals,↑size,↑type ) =
   Ordinal_type( ↓env,↑locals,↑size,↑type ) |
   Pointer_type( ↑locals,↑size,↑type );
```

```
    /*************************************************************/
    /*                                                           */
    /*       Deal with procedure declarations                    */
    /*                                                           */
    /*************************************************************/

Procedure( ↓↑env,↓labelenv,↓storemap ) =
   ↑Procedure, ↑Identifier( ↑name ),
   <<Semi_colon,Semi_colon,"Malformed procedure header">>
      (
         Predefined_procedure( ↓env,↓labelenv,↓↑env[name],↑<env> ) |
         Undefined_procedure( ↓name,↓env,↓labelenv,↓storemap,↑<env> )
      );

Predefined_procedure( ↓env,↓labelenv,↓↑symbol,↑[] ) =
   Forward_function
      (
         /* Check procedure is correctly defined */
         ↓symbol.kind.procedure.defined,
         ↓symbol.type.ordinal.basic = "VOID",
         ↑symbol.kind.procedure.defined
      ),
   <<Semi_colon,Semi_colon,"Malformed block">>
      Block
         (
            ↓env, ↓labelenv,
            ↓symbol.kind.procedure.parameters,
            ↓symbol.kind.procedure.storemap,
            ↓symbol.kind.procedure.label
         );

Undefined_procedure
   (
      ↓name, ↓env, ↓labelenv, ↓storemap,
      ↑[name → {procedure({defined,label,parameters,storemap}),
         ordinal({"VOID",0,0})}]
   ) =
Parameters
   (
      ↓env,↓{storemap.level+1,0},↓[],
      ↑defined,↑storemap,↑parameters
   ),
↑Semi_colon, Unique_number( ↑label ),
<<Semi_colon,Semi_colon,"Malformed directive or block">>
   (
      ↑Forward( ↑defined ) |
      Block( ↓env,↓labelenv,↓parameters,↓storemap,↓label )
   ),
↑Semi_colon;
```

```
    /*************************************************************/
    /*                                                           */
    /*       Process function and procedure parameters           */
    /*                                                           */
    /*************************************************************/

Parameters
   (
      ↓env, ↓storemap, ↓parameters,
      ↑"DEFINED", ↑storemap, ↑parameters
   ) =
[
   ↑Bracket1,
   <<Bracket2,Semi_colon,"Malformed parameter list">>
      (
         Parameter_list( ↓env,↓"VALUE",↓↑storemap,↑<parameters> ),
            {
               ↑Comma,
               Parameter_list( ↓env,↓"VALUE",↓↑storemap,↑<parameters> )
            }
      ),
   ↑Bracket2
];

Parameter_list( ↓env,↓var,↓↑storemap,↑parameters ) =
   [ ↑Var( ↑var ) ], /* No function or procedure arguments as */
      Parameter      /* these are not supported by P-Code     */
         (
            ↓env,↓var,↓[],↓storemap,
            ↑size,↑type,↑storemap,↑parameters
         );
```

```
Parameter
   (
      ↓env, ↓var, ↓parameters, ↓storemap,
      ↑size, ↑type,
      ↑{storemap.level,storemap.size+size},
      ↑[name → {variable({var,storemap.size,storemap.level,
        name,size,0}),type}]
   ) =
↑Identifier( ↑name ),
   (
      ↑Comma,
      Parameter
         (
            ↓env,↓var,↓[],↓storemap,
            ↑size,↑type,↑storemap,↑<parameters>
         )
   |
      ↑Colon,
      Parameter_type( ↓env,↑size,↑type )
   );

Parameter_type( ↓env,↑env[name].kind.type,↑env[name].type ) =
   ↑Identifier( ↑name );

    /*************************************************************/
    /*                                                           */
    /*       Read and process Pascal statements                  */
    /*                                                           */
    /*************************************************************/

Compound_block( ↓env,↓labelenv,↓↑locallabels,↓storemap ) =
   ↑Begin,
      Statements( ↓env,↓labelenv,↓↑locallabels,↓storemap ),
   <<End,End,"Misplaced END">> ↑End;

Statements( ↓env,↓labelenv,↓↑locallabels,↓storemap ) =
   Labelled_statement( ↓env,↓labelenv,↓↑locallabels,↓storemap ),
      {
         ↑Semi_colon,
         Labelled_statement( ↓env,↓labelenv,↓↑locallabels,↓storemap )
      };
```

```
Labelled_statement( ↓env,↓labelenv,↓↑locallabels,↓storemap ) =
   [
      ↑Integer( ↑label ),
      <<Semi_colon,Semi_colon,"Malformed label">>
         (
            ↑Colon,
            Statement_label( ↓↑locallabels[label] )
         )
   ],
   Statement( ↓env,↓labelenv,↓↑locallabels,↓storemap );

Statement_label( ↓↑locallabel ) =
   Output_statement_label
      (
         ↓locallabel.defined = "UNDEFINED",
         ↓locallabel.label,
         ↑locallabel.defined
      );

Output_statement_label( ↓text,↓label,↑"DEFINED" ) =
   ↓Labell( ↓{label} );

Statement( ↓env,↓labelenv,↓↑locallabels,↓storemap ) =
   Assignment_or_procedure( ↓env ) |
   Case( ↓env,↓labelenv,↓↑locallabels,↓storemap ) |
   Compound_block( ↓env,↓labelenv,↓↑locallabels,↓storemap ) |
   For( ↓env,↓labelenv,↓↑locallabels,↓storemap ) |
   If( ↓env,↓labelenv,↓↑locallabels,↓storemap ) |
   Goto( ↓labelenv ) |
   Repeat( ↓env,↓labelenv,↓↑locallabels,↓storemap ) |
   While( ↓env,↓labelenv,↓↑locallabels,↓storemap ) |
   With( ↓env,↓labelenv,↓↑locallabels,↓storemap );

    /*************************************************************/

Assignment_or_procedure( ↓env ) =
   Identifier( ↓env,↑kind,↑type ),
   <<Semi_colon,Semi_colon,"Malformed assignment or function">>
      (
         Call_function /* defined later */
            (
               ↓env, ↓kind.procedure,
               ↓type.ordinal.basic="VOID"
            )
      |
         Assignment( ↓env,↓kind,↓type )
      );
```

```
Assignment( ↓env,↓kind,↓type1 ) =
   Assignment_variable( ↓env,↓kind.variable,↓↑type1 ),
      ↑Assign, Expression( ↓env,↑type2 ),
   Store( ↓type1,↓type2 );

Assignment_variable( ↓env,↓variable,↓↑type ) =
   (
      Load_ordinal( ↓variable,↓type.ordinal.basic ) |
      Load_complex( ↓env,↓variable,↓↑type )
   );

    /*************************************************************/

Case( ↓env,↓labelenv,↓↑locallabels,↓storemap ) =
   ↑Case,
   <<End,End,"Malformed CASE statement">>
      (
         Expression( ↓env,↑type ), ↑Of,
         Case_block
            (
               ↓env,↓labelenv,↓type.ordinal,
               ↓↑locallabels,↓storemap
            ),
         ↑End
      );

Case_block( ↓env,↓labelenv,↓ordinal,↓↑locallabels,↓storemap ) =
   Unique_number( ↑label1 ), Unique_number( ↑label2 ),
   ↓Instruction1( ↓{"XJP",label1} ),
      Complex_case_list
         (
            ↓env, ↓labelenv, ↓label2, ↓↑ordinal,
            ↓↑locallabels, ↓storemap, ↑caselist
         ),
   /* Output case jump table */
   ↓Label1( ↓{label1} ),
   ↓Constant( ↓{ordinal.lower} ), ↓Constant( ↓{ordinal.upper} ),
   { ↓Instruction1( ↓{"UJP",<caselist>>} ) },
   ↓Label1( ↓{label2} );
```

```
Complex_case_list
   (
      ↓env, ↓labelenv, ↓label2, ↓↑ordinal,
      ↓↑locallabels, ↓storemap, ↑caselist
   ) =
New_label( ↑label1 ),
Complex_case_item
   (
      ↓env, ↓labelenv, ↓label1, ↓label2,
      ↓↑ordinal, ↓↑locallabels,
      ↓storemap, ↑caselist
   ),
{
   Complex_case_item
      (
         ↓env, ↓labelenv, ↓label1, ↓label2,
         ↓↑ordinal, ↓↑locallabels,
         ↓storemap, ↑<caselist>
      )
};

Case_item
   (
      ↓env, ↓labelenv, ↓label1, ↓label2,
      ↓↑ordinal, ↓↑locallabels,
      ↓storemap, ↑caselist
   ) =
Case_label
   (
      ↓env,↓label1,↓ordinal,
      ↑ordinal.lower,↑text,↑caselist
   ),
{
   ↑Comma,
   Case_label
      (
         ↓env,↓label1,↓ordinal,
         ↑ordinal.lower,↑text,↑<caselist>
      )
},
↑Colon,
   Statement( ↓env,↓labelenv,↓↑locallabels,↓storemap ),
   ↓Instruction1( ↓{"UJP",label2} );
```

```
Case_label
   (
      ↓env, ↓label1, ↓ordinal,
      ↑value = ordinal.lower+1 <= ordinal.upper,
      ↑ordinal.basic = type.ordinal.basic,
      ↑[value → label1]
   ) =
Constant_or_identifier( ↓env,↑value,↑type );

    /*************************************************************/

For( ↓env,↓labelenv,↓↑locallabels,↓storemap ) =
   ↑For,
   <<Semi_colon,Semi_colon,"Malformed FOR statement">>
      (
         For_assignment( ↓env,↑kind,↑type ),
         For_compare( ↓env,↓kind,↓type,↑operator,↑label1,↑label2 ),
      ↑Do,
         Statement( ↓env,↓labelenv,↓↑locallabels,↓storemap ),
         For_endloop( ↓env,↓kind,↓type,↓operator,↓label1,↓label2 )
      );

For_assignment( ↓env,↑kind,↑type1 ) =
   Identifier( ↓env,↑kind,↑type1 ),
      ↑Assign, Expression( ↓env,↑type2 ),
   Store( ↓type1,↓type2 );

For_compare( ↓env,↓kind,↓type1,↑operator,↑label1,↑label2 ) =
   New_label( ↑label1 ), Unique_number( ↑label2 ),
   ↓Instruction2( ↓{"LOD",kind.variable.level,kind.variable.offset} ),
      (
         ↑To( ↑operator ),
         Expression( ↓env,↑type2 ),
         Relation( ↓"<",↓type1,↓type2,↑type )
      |
         ↑Downto( ↑operator ),
         Expression( ↓env,↑type2 ),
         Relation( ↓">",↓type1,↓type2,↑type )
      ),
   ↓Instruction1( ↓{"FJP",label2} );
```

```
For_endloop( ↓env,↓kind,↓type,↓operator,↓label1,↓label2 ) =
   ↓Instruction2
      (
         ↓{
           unary[operator][type.ordinal.basic],
           kind.variable.level,
           kind.variable.offset
         }
      ),
   ↓Instruction1( ↓{"UJP",label1} ),
   ↓Label1( ↓{label2} );

New_label( ↑label ) =
   Unique_number( ↑label ),
   ↓Label1( ↓{label} );

    /*************************************************************/

If( ↓env,↓labelenv,↓↑locallabels,↓storemap ) =
   ↑If, Unique_number( ↑label ),
   <<Semi_colon,Semi_colon,"Malformed IF statement">>
      (
         Condition( ↓env,↓label,↑text ),
      ↑Then,
         Statement( ↓env,↓labelenv,↓↑locallabels,↓storemap ),
            [ ↑Else, <<Semi_colon,Semi_colon,"Malformed ELSE part">>
               (
                  Else_labelling( ↓label,↑label ),
                  Statement( ↓env,↓labelenv,↓↑locallabels,↓storemap )
               )
            ]
      ),
   ↓Label1( ↓{label} );

Condition( ↓env,↓label,↑type.ordinal.basic="BOOLEAN" ) =
   Expression( ↓env,↑type ),
   ↓Instruction1( ↓{"FJP",label} );

Else_labelling( ↓label1,↑label2 ) =
   Unique_number( ↑label2 ),
   ↓Instruction1( ↓{"FJP",label2} ),
   ↓Label1( ↓{label1} );

    /*************************************************************/

Goto( ↓labelenv ) =
   ↑Goto, ↑Integer( ↑label ),
      ↓Instruction1( ↓{"UJP",labelenv[label].label} );
```

```
Repeat( ↓env,↓labelenv,↓↑locallabels,↓storemap ) =
   ↑Repeat, New_label( ↑label ),
   <<Semi_colon,Semi_colon,"Malformed REPEAT statement">>
      (
         Statements( ↓env,↓labelenv,↓↑locallabels,↓storemap ),
         ↑Until, Condition( ↓env,↓label,↑text )
      );

While( ↓env,↓labelenv,↓↑locallabels,↓storemap ) =
   ↑While, Unique_number( ↑label1 ), New_label( ↑label2 ),
   <<Semi_colon,Semi_colon,"Malformed WHILE statement">>
      (
         Condition( ↓env,↓label1,↑text ), ↑Do,
            Statement( ↓env,↓labelenv,↓↑locallabels,↓storemap )
      ),
   ↓Instruction1( ↓{"UJP",label2} ),
   ↓Label1( ↓{label1} );

    /*************************************************************/

With( ↓env,↓labelenv,↓↑locallabels,↓storemap ) =
   ↑With,
   <<Semi_colon,Semi_colon,"Malformed WITH statement">>
      (
         With_variable( ↓env,↓storemap,↑env,↑storemap.size ),
            {
               ↑Comma,
               With_variable( ↓env,↓storemap,↑env,↑storemap.size )
            },
      ↑Do,
         Statement( ↓env,↓labelenv,↓↑locallabels,↓storemap )
   );

With_variable
   (
      ↓env, ↓storemap,
      ↑env ∪ withenv, ↑storemap.size + POINTER_SIZE
   ) =
Variable( ↓env,↑type ),
   ↓Instruction2( ↓{"STRA",storemap.level,storemap.size} ),
With_item( ↓<type.record.fields>>,↓storemap.size,↑withenv ),
   { With_item( ↓<type.record.fields>>,↓storemap.size,↑<withenv> ) };
```

```
Variable( ↓env,↑type ) =
   Identifier( ↓env,↑kind,↑type ),
      (
         Load_ordinal( ↓kind.variable,↓type.ordinal.basic ) |
         Load_complex( ↓env,↓kind.variable,↓↑type )
      );

With_item
   (
      ↓symbol,
    ↓symbol.kind.variable.withoffset,
    ↑[symbol.kind.variable.name → symbol]
   );

    /*************************************************************/
    /*                                                           */
    /*       Verify that all labels defined in the header        */
    /*          have been set                                    */
    /*                                                           */
    /*************************************************************/

Verify_labels( ↓locallabels ) =
   { Verify_label( ↓<<locallabels> ) };

Verify_label( ↓locallabel ) =
   <<"Warning label defined but not set">>
      Defined_label( ↓locallabel.defined="DEFINED" );

Defined_label( ↓text );

    /*************************************************************/
    /*                                                           */
    /*       Parse a Pascal expression                           */
    /*                                                           */
    /*************************************************************/

Expression( ↓env,↑type1 ) =
   Unary_sign( ↓env,↑type1 ),
      {
         ↑Relation( ↑operator ),
         Unary_sign( ↓env,↑type2 ),
         Relation( ↓operator,↓type1,↓type2,↑type1 )
      };
```

```
Unary_sign( ↓env,↑type ) =
      (
         ↑Sum( ↑operator ),
         Unary_sign( ↓env,↑type ),
         Unary( ↓operator,↓type )
      )
   |
      Sum( ↓env,↑type );

Sum( ↓env,↑type1 ) =
   Term( ↓env,↑type1 ),
      {
         ↑Sum( ↑operator ),
         Term( ↓env,↑type2 ),
         Binary( ↓operator,↓↑type1,↓type2 )
      };

Term( ↓env,↑type1 ) =
   Factor( ↓env,↑type1 ),
      {
         ↑Term( ↑operator ),
         Factor( ↓env,↑type2 ),
         Binary( ↓operator,↓↑type1,↓type2 )
      };

Factor( ↓env,↑type ) =
      (
         ↑Not,
         Factor( ↓env,↑type ),
         Unary( ↓"NOT",↓type )
      )
   |
      Terminal( ↓env,↑type );

Terminal( ↓env,↑type ) =
   Constant( ↑type ) |
   Variable_or_function( ↓env,↑type ) |
   Set( ↓env,↑type ) |
   ↑Bracket1, Expression( ↓env,↑type ), ↑Bracket2;
```

```
    /*************************************************************/
    /*                                                           */
    /*       Load a constant                                     */
    /*                                                           */
    /*************************************************************/

Constant( ↑type ) =
   Unsigned_constant( ↑value,↑type ),
   Load_constant_value( ↓value,↓type );

Load_constant_value( ↓value,↓type ) =
   ↓Instruction1( ↓{load_constant[type.ordinal.basic],value} ) |
   ↓InstructionS( ↓{"LDCA",strings[value]} );

    /*************************************************************/
    /*                                                           */
    /*       Load a constant, variable or function               */
    /*                                                           */
    /*************************************************************/

Variable_or_function( ↓env,↑type ) =
   Identifier( ↓env,↑kind,↑type ),
      (
         Load_constant_value( ↓kind.value,↓type ) |
         Load_variable_value( ↓env,↓kind.variable,↓↑type ) |
         Call_function( ↓env,↓kind.procedure,↓type.ordinal.basic<>"VOID" )
      );

Load_variable_value( ↓env,↓variable,↓↑type ) =
   (
      Load_ordinal( ↓variable,↓type.ordinal.basic ) |
      Load_complex( ↓env,↓variable,↓↑type )
   ),
   [
      ↓Instruction1( ↓{load_value[type.ordinal.basic],0} ) |
      ↓Instruction1( ↓{"INDS"<>type.set.type.basic,0} )
   ];

Load_ordinal( ↓variable,↓text ) =
   ↓Instruction2( ↓{"LDA",variable.level,variable.offset} );

Load_complex( ↓env,↓variable,↓↑type ) =
   Load_base( ↓variable ),
      {
         ↑Bracket3, Load_array( ↓env,↓↑type ), ↑Bracket4 |
         Load_record( ↓type.record,↑type ) |
         Load_pointer( ↓env[type.pointer].type,↑type )
      };
```

```
Load_base( ↓variable ) =
   ↓Instruction2( ↓{"LDA",variable.level,variable.withoffset>0} ),
      ↓Instruction1( ↓{"LDCI",variable.offset} ),
      ↓Instruction0( ↓{"ADI"} ) |
   ↓Instruction2( ↓{"LDA",variable.level,variable.offset} );

Load_array( ↓env,↓↑type ) =
   Load_subscript( ↓env,↓type.array,↑type ),
      {
         ↑Comma,
         Load_subscript( ↓env,↓type.array,↑type )
      };

Load_subscript( ↓env,↓array,↑array.type ) =
   Ordinal_expression( ↓env,↓array.index,↑text ),
   ↓Instruction1( ↓{"IXA",array.size} );

Ordinal_expression( ↓env,↓ordinal,↑ordinal.basic=type.ordinal.basic ) =
   Expression( ↓env,↑type );

Load_record( ↓record,↑record.fields[name].type ) =
   ↑Dot, ↑Identifier( ↑name ), /* No checking of variant fields yet */
   <<Semi_colon,Semi_colon,"Undefined field identifier">>
      Load_record_item( ↓record.fields[name].kind.variable );

Load_record_item( ↓variable ) =
   ↓Instruction1( ↓{"IXA",variable.offset} );

Load_pointer( ↓type,↑type ) =
   ↑Pointer, ↓Instruction1( ↓{"INDA",0} );

Call_function( ↓env,↓procedure,↓text ) =
   ↓Instruction0( ↓{"MST"} ),
   Load_actual_parameters( ↓env,↓procedure.parameters ),
   ↓Instruction1( ↓{"CUP",procedure.label} );

Load_actual_parameters( ↓env,↓parameters ) =
   [
      ↑Bracket1,
         Load_parameter( ↓env,↓<<parameters> ),
            {
               ↑Comma,
               Load_parameter( ↓env,↓<<parameters> )
            },
      ↑Bracket2
   ];
```

```
Load_parameter( ↓env,↓parameter ) =
   <<Bracket2,Semi_colon,"Malformed parameter list">>
      (
         Value_parameter
            (
               ↓env,
               ↓parameter.kind.variable.class="VALUE",
               ↓parameter.type
            ) |
         Var_parameter
            (
               ↓env,
               ↓parameter.kind.variable.class="VAR",
               ↓parameter.type
            )
      );

Value_parameter( ↓env,↓text,↓type ) =
   Copy_expression( ↓env,↓type );

Copy_expression( ↓env,↓type1 ) =
   Expression( ↓env,↑type2 ),
   Copy( ↓type1,↓type2 );

Var_parameter( ↓env,↓text,↓type ) =
   Parameter_variable( ↓env,↓type );

Parameter_variable( ↓env,↓type ) =
   Identifier( ↓env,↑kind,↑type ),
      (
         Load_ordinal( ↓kind.variable,↓type.ordinal.basic ) |
         Load_complex( ↓env,↓kind.variable,↓↑type )
      );

    /*************************************************************/
    /*                                                           */
    /*      Evaluate a set expression                            */
    /*  (P Code only supports constant set expressions )         */
    /*                                                           */
    /*************************************************************/

Set( ↓env,↑type ) =
   ↑Bracket3,
      Set_elements( ↓env,↓0,↓null_set,↑type ),
   ↑Bracket4;
```

```
Set_elements( ↓env,↓setbits,↓typel,↑set({"PACKED",typel.ordinal}) ) =
   [ <<Bracket4,Semi_colon,"Malformed set">>
      (
         Ordinal_type( ↓env,↑locals,↑size,↑typel ),
         Set_bits
            (
               ↓setbits, ↓"NOTHING",
               ↓typel.ordinal.lower < typel.ordinal.upper >= 0,
               ↓typel.ordinal.upper < SET_SIZE*8,
               ↑setbits
            ),
         {
            ↑Comma,
            Ordinal_type( ↓env,↑locals,↑size,↑type2 ),
            Set_bits
               (
                  ↓setbits,
                  ↓typel.ordinal.basic = type2.ordinal.basic,
                  ↓type2.ordinal.lower < type2.ordinal.upper >= 0,
                  ↓type2.ordinal.upper < SET_SIZE*8,
                  ↑setbits
               )
         }
      )
   ],
      ↓Instruction1( ↓{"LDCS",setbits} );

Set_bits
   (
      ↓setbits, ↓text, ↓lower, ↓upper,
      ↑((1 << (upper-lower+1))-1) << lower | setbits
   );
```

```
/*************************************************************/
/*                                                           */
/*      Code generation rules for expressions                */
/*                                                           */
/*************************************************************/

Store( ↓type1,↓type2 ) =
   Arthmetic
      (
         ↓":=",
         ↓↑type1.ordinal.basic,
         ↓type2.ordinal.basic
      ) |
   Set_or_pointer
      (
         ↓":=",
         ↓type1.pointer,
         ↓type2.pointer,
         ↓pointers
      ) |
   Set_or_pointer
      (
         ↓":=",
         ↓type1.set.type.basic,
         ↓type2.set.type.basic,
         ↓sets
      ) |
   Strings
      (
         ↓":=",
         ↓type1.array.packing=type2.array.packing="PACKING",
         ↓type1.array.index.basic=type2.array.index.basic="INTEGER",
         ↓type1.array.index.lower=type2.array.index.lower,
         ↓type1.array.index.upper=type2.array.index.upper,
         ↓type1.array.type.ordinal.basic=type2.array.type.ordinal.basic=
            "CHAR"
      );
```

```
Relation( ↓operator,↓type1,↓type2,↑boolean_type ) =
   Arthmetic
      (
         ↓operator,
         ↓↑type1.ordinal.basic,
         ↓type2.ordinal.basic
      ) |
   Set_or_pointer
      (
         ↓operator,
         ↓type1.pointer,
         ↓type2.pointer,
         ↓pointers
      ) |
   Set_or_pointer
      (
         ↓operator,
         ↓type1.set.type.basic,
         ↓type2.set.type.basic,
         ↓sets
      ) |
    Strings
      (
         ↓operator,
         ↓type1.array.packing=type2.array.packing="PACKING",
         ↓type1.array.index.basic=type2.array.index.basic="INTEGER",
         ↓type1.array.index.lower=type2.array.index.lower,
         ↓type1.array.index.upper=type2.array.index.upper,
         ↓type1.array.type.ordinal.basic=type2.array.type.ordinal.basic=
            "CHAR"
      );

 Binary( ↓operator,↓↑type1,↓type2 ) =
    Arthmetic
       (
          ↓operator,
          ↓↑type1.ordinal.basic,
          ↓type2.ordinal.basic
       ) |
    Set_or_pointer
       (
          ↓operator,
          ↓type1.set.type.basic,
          ↓type2.set.type.basic,
          ↓sets
       );
```

```
Unary( ↓operator,↓type ) =
   ↓Instruction0( ↓{unary[operator][type.ordinal.basic]} );

Copy( ↓type1,↓type2 ) =
   No_copy( ↓type1.ordinal.basic=type2.ordinal.basic ) |
   No_copy( ↓type1.pointer=type2.pointer ) |
   No_copy( ↓type1.set.type.basic=type2.set.type.basic ) |
   Copy_complex( ↓type1,↓type2 );

Arthmetic( ↓operator,↓↑text1,↓text2 ) =
   [ Float( ↓float[text1][text2],↑text1 ) ],
   ↓Instruction0( ↓{arithmetic[operator][text1][text2]} );

Float( ↓text,↑"REAL" ) =
   ↓Instruction0( ↓{text} ); /* Float ordinal value if required */

Set_or_pointer( ↓operator,↓text1,↓text2,↓table ) =
   /* Below we examine sets and pointers for null values */
   /* if so it matches and set or pointer as neccessary  */
   [ Null( ↓table[text1],↑text2 ) | Null( ↓table[text2],↑text1 ) ],
   Null( ↓text1=text2,↑text ),
   ↓Instruction0( ↓{table[operator]} );

Null( ↓text,↑text );

Strings( ↓operator,↓text,↓text,↓lower,↓upper,↓text ) =
   ↓Instruction1( ↓{pointers[operator],upper-lower} );

No_copy( ↓text );

Copy_complex( ↓type1,↓type2 ) =
   /* CPY replaces a pointer at TOS with its value */
   ↓Instruction1( ↓{"CPY",type1.array.size=type2.array.size} ) |
   ↓Instruction1( ↓{"CPY",type1.record.size=type2.record.size} );

END
```

Appendix 6

A Small

Pascal Example

Introduction

This appendix contains a small example Pascal program along with object code generated by Aston Compiler Constructor (ACC). The object code was produced by executing the Compiler Construction Language (CCL) specification for Pascal given in Appendix 5.

The Example Pascal Program

```
PROGRAM Difference_of_two_Squares();

TYPE
   Squares_Data = RECORD
      Min : INTEGER;
      Max : INTEGER;
   END;

VAR
   Initial : Squares_Data;
   Result  : INTEGER;

PROCEDURE Squares( Value : Squares_Data,VAR Result : INTEGER );
BEGIN
   Result := (Value.Min * Value.Min) - (Value.Max * Value.Max)
END;

BEGIN
   Initial.Min := 12;
   Initial.Max := 24;

   Squares( Initial,Result )
END.
```

The Generated Object Code

```
                UJP             0       /* Jump to main block */
LABEL                           2
                LDA     1       8       /* Procedure Squares */
                LDA     1       0
                IXA             0
                INDI            0
                LDA     1       0
                IXA             0
                INDI            0
                MPI
                LDA     1       0
                IXA             4
                INDI            0
                LDA     1       0
                IXA             4
                INDI            0
                MPI
                SBI
                STOI
                RTS
LABEL                           0
                LDA     0       0       /* main block */
                IXA             0
                LDCI            12
                STOI
                LDA     0       0
                IXA             4
                LDCI            24
                STOI                    /* Mark stack ready for call */
                MST
                LDA     0       0
                CPY             8
                LDA     0       8
                CUP             2
                RTS
```

References

Introduction

The articles and papers cited within this thesis are listed in this Appendix in alphabetical order of author and ascending year of publication.

References

[Aho 1974] Aho, A.V., Johnson, S.C.
'LR parsing', Computing surveys, Volume 6, Number 2,
Pages 99-124, New York, U.S.A.

[Aho 1985] Aho, A.V., Sethi, R., Ullman, J.D.
'Compilers, principles, techniques and tools'.
Addison-Wesley, Reading, U.S.A.

[Ammann 1981] Ammann, U., Jacobi, C., Jensen, K., Nageli, H., Nori, K.
'Pascal - The langauge and its implementation' (Chapter 9).
John Wiley and Sons, Chichester, U.K.

[Bell 1971] Bell, C.G., Newell, A.
'Computer structures readings and examples'.
McGraw-Hill, New York, U.S.A.

[BSI 1982] British Standards Institute (BS 6192).
'Specification for computer programming language Pascal'.
2, Park Street, London, U.K.

[Brooker 1962] Brooker, R.A., Morris, D.
'A general translation program for phrase structured languages', Journal of the ACM, Volume 9, Number 1, Pages 1-10, New York, U.S.A.

[Bird 1982] Bird, P.
'An implementation of a code generator specification language for table driven code generators', Proceedings of the SIGPLAN symposium on compiler construction, Boston, Pages 44-55, New York, U.S.A.

[Brown 1979] Brown, P.J.
'Writing interactive compilers and interpreters'.
John Wiley and Sons, Chichester, U.K.

[Cattell 1978] Cattell, R.G.G.
'Formatization and automatic derivation of code generators', Ph.D thesis, Department of computer science, Carnegie-Mellon University, Pittsburgh, U.S.A.

[Davidson 1980] Davidson, J.W., Frazer, C.W.
'The design and application of a retargetable peephole optimizer', ACM transactions on programming languages and systems, Volume 2, Number 2, Pages 191-202, New York, U.S.A.

[Davidson 1982] Davidson, J.W., Fraser, C.W.
'A machine independent linker', Software practice and experience, Volume 12, Pages 351-366, Chichester, U.K.

[Davidson 1984] Davidson, J.W., Frazer, C.W.
'Automatic generation of peephole optimisations', SIGPLAN notices, Volume 19, Number 1, Pages 111-116, New York, U.S.A.

[Dehottay 1977] Dehottay, J.P., Feuerhahn, H., Koster, C.H.A., Stahl, H.
'Syntaktische beschreibung von CDL2', Technishe, Universitaet Berlin, Fachbereich 20, Forschungsgruppe softwaretechnik, Germany.

[DeRemer 1975] DeRemer, F.L.
'On compiler structure and translator writing tools', Proceedings of the 8th Hawaii international conference on system sciences, University of Hawaii, U.S.A.

[DeRemer 1984] DeRemer, F.L.
'Regular right-part attribute grammars', SIGPLAN notices,
Volume 19, Number 6, Pages 171-178, New York, U.S.A.

[Engelfret 1984] Engelfret, J.
'Attribute evaluation methods' in
'Methods and tools for compiler construction', [Ed. Lohro],
Cambridge University Press, Cambridge, U.K.

[Frazer 1977] Frazer, C.W.
'Automatic generation of code generators', Ph.D Thesis,
Department of computer science, Yale University, U.S.A.

[Fisher 1981] Fisher, C.N., Ganapathi, M.
'Bibliography on automated retargetable code generation',
SIGPLAN notices, Volume 16, Number 10, Pages 9-12,
New York, U.S.A.

[Ganapathi 1980] Ganapathi, M.
'Retargetable code generation using attribute grammars'.
Ph.D. dissertation, Technical report number 406, University
of Winsconsin, Madison, U.S.A.

[Ganapathi 1985] Ganapathi, M.
'Affix grammar driven code generation', ACM transactions on programming languages and systems, Volume 7, Number 4, Pages 560-599, New York, U.S.A.

[Ganapathi 1986] Ganapathi, M., Hennessy, J.
'Advances in compiler technology', Annual reviews in computer science, Number 1, Pages 83-106, New York, U.S.A.

[Giegerich 1983] Giegerich, R.
'A formal framework for the derivation of machine specific optimizers', ACM transactions on programming languages and systems, Volume 5, Number 3, Pages 478-498, New York, U.S.A.

[Graham 1980] Graham, S.L.
'Table driven code generation', IEEE computer, Volume 13, Number 8, Pages 25-33, Encino, U.S.A.

[Horspool 1987] Horspool, R.N.
'An alternative to the Graham-Grandville code generation method', IEEE computer, Volume 20, Number 5, Pages 33-39, Encino, U.S.A.

[Jespersen 1978] Jespersen, J.P., Madsen, O.L., Riis, H.
'New extended attribute system (NEATS)', DAIMI, Aarhus University, Denmark.

[Johnson 1975] Johnson, S.C.
'YACC - Yet Another Compiler Compiler', Computer science technical report number 32, Bell laboratories, Murray hill, New Jersey, U.S.A.

[Johnson 1979] Johnson, S.C.
'A tour through the Portable C Compiler' (in the UNIX programmers manual), Bell laboratories, Murray hill, New Jersey, U.S.A.

[Keizer 1983] Keizer, E.G., Staveren, H.V., Stevenson, J.W., Tanenbaum, A.M.
'A practical tool kit for making portable compilers', Communications of the ACM, Volume 26, Number 9, Pages 654-660, New York, U.S.A.

[Kessler 1984a] Kessler, P.
'Automated discovery of machine specific code improvements', Ph.D thesis, University of California at Berkeley, U.S.A.

[Kessler 1984b] Kessler, R.
'Peep - An architectural description driven peephole optimizer', SIGPLAN notices, Volume 19, Number 1, Pages 106-110, New York, U.S.A.

[Knuth 1968] Knuth, D.E.
'Semantics of context free languages', Mathematical systems theory, Volume 12, Pages 127-145, New York, U.S.A.

[Koskimies 1983] Koskimies, K.
'Extensions of one pass attribute grammars', Report A-1983-4, December 1983, Helsinki University, Department of computer science, Finland.

[Koster 1971] Koster, C.H.A.
'Affix grammars' in
'ALGOL68 implementation', [Ed. Peck],
North Holland, Amsterdam, The Netherlands.

[Lesk 1976] Lesk, M.E.
'LEX - A lexical analyser generator', Computer science technical report number 39, Bell laboratories, Murray hill, New Jersey, U.S.A.

[Madsen 1983] Madsen, O.L., Watt, D.A.
'Extended attribute grammars', Computer Journal, Volume 26, Number 2, Pages 142-153, London, U.K.

[McGettrick 1980] McGettrick, A.D.
'The definition of programming languages', Page 191, Cambridge University Press, Cambridge, U.K.

[Meijer 1982] Meijer, H., Nijholt, A.
'TWT's since 1970 : A selective bibliography', Faculty of science, Department of informatics, Nijmegen University, The Netherlands.

[Naur 1963] Naur, P.
'Revised report on the algorithmic language ALGOL 60', Communications of the ACM, Volume 16, Number 1, Pages 1-17, New York, U.S.A.

[Paulson 1982] Paulson, L.
'A semantics directed compiler generator', Conference Record of the 9th ACM symposium on the principles of programming languages. Pages 224-233, New York, U.S.A.

[Pleban 1984] Pleban, U.
'Compiler prototyping using formal semantics', Proceedings of the SIGPLAN symposium on compiler construction, Boston, Pages 84-105, New York, U.S.A.

[Raiha 1978] Raiha, J.K., Saarinen, M., Soininen, E., Tienari, M.
'The compiler writing system HLP', Report A-1978-2, Department of computer science, University of Helsinki U.S.A.

[Raiha 1980a] Raiha, J.K.
'Bibliography on attribute grammars', SIGPLAN notices, Volume 15, Number 3, Pages 35-44, New York, U.S.A.

[Raiha 1980b] Raiha, J.K.
'Experiences with the compiler writing system HLP', in 'Semantics directed compiler generation', [ed. Jones], Springer-Verlag, The Netherlands.

[Reps 1987] Reps, T., Teitelbaum, T.
'Language processing in program editors', IEEE computer, Volume 20, Number 11, Pages 29-40, Encino, U.S.A.

[Scott 1971] Scott, D.S., Strachey, C.
'Towards the mathematical semantics of a computer languages', Proceedings of the symposium on computer automation, Pages 19-46, New York, U.S.A.

[Scowen 1982] Scowen, R.S.
'A standard syntactic meta-language', SIGPLAN notices, Volume 17, Number 3, Pages 68-73, New York, U.S.A.

[Watt 1980] Watt, D.A.
'Rule splitting and attribute directed parsing', in 'Semantics directed compiler generation', [ed. Jones], Springer-Verlag, The Netherlands.

[Wilcox 1971] Wilcox, T.R.
'Generating machine code for high level programming languages', Ph.D Thesis, Cornell University, Ithaca, New York, U.S.A.