[illegible]

THEORY AND PRACTICE IN THE CONSTRUCTION OF EFFICIENT INTERPRETERS

Nigel Paul Chapman

Submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

The University of Leeds
Department of Computer Studies

October 1980

ABSTRACT

Various characteristics of a programming language, or of the hardware on which it is to be implemented, may make interpretation a more attractive implementation technique than compilation into machine instructions. Many interpretive techniques can be employed; this thesis is mainly concerned with an efficient and flexible technique using a form of interpretive code known as indirect threaded code (ITC). An extended example of its use is given by the Setl-s implementation of Setl, a programming language based on mathematical set theory. The ITC format, in which pointers to system routines are embedded in the code, is described and its extension to cope with polymorphic operators. The operand formats and some of the system routines are described in detail to illustrate the effect of the language design on the interpreter.

Setl must be compiled into indirect threaded code and its elaborate syntax demands the use of a sophisticated parser. In Setl-s an LR(1) parser is implemented as a data structure which is interpreted in a way resembling that in which ITC is interpreted at runtime. Qualitative and quantitative aspects of the compiler, interpreter and system as a whole are discussed.

The semantics of a language can be defined mathematically using denotational semantics. By setting up a suitable domain structure, it is possible to devise a semantic definition which embodies the essential features of ITC. This definition can be related, on the one hand to the standard semantics of the language, and on the other to its implementation as an ITC-based interpreter. This is done for a simple language known as X10. Finally, an indication is given of how this approach could be extended to describe Setl-s, and of the insight gained from such a description. Some possible applications of the theoretical analysis in the building of ITC-based interpreters are suggested.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTERPRETERS

1.1 INTERPRETATION AND COMPILATION

An interpreter-based implementation of a programming language is a system in which some representation of the source program is used at runtime to direct the flow of execution through system routines. This is commonly distinguished from a compiler-based implementation, in which the source program is translated into a sequence of instructions which can be executed by hardware. Idiomatically, the words 'compiler' and 'interpreter' are used loosely: 'compiler' is sometimes used to mean the same as a compiler-based implementation, and sometimes used to mean any program or part of a system which translates from a source language into any lower-level representation. Similarly, 'interpreter' can mean that part of a system which performs the runtime operations, but can also be used to mean an entire interpreter-based implementation. The way in which the two words are used at particular points in this thesis should be obvious from their context.

These two terms provide an important distinction between implementations of programming languages, because of the different sorts of dependence on the machine hardware which they imply, although this cannot be a sharp distinction because most implementations possess some of the characteristics of each and certain techniques defy this classification. (For example, the machine code produced by some compilers consists almost entirely of subroutine calls.) In both cases, the execution of a program ultimately consists of the execution of a series of machine instructions, but, as will be seen, an interpretive system provides an extra degree of flexibility and independence from the facilities of the hardware.

The main advantage attributed to systems generating hard code is that no software is required to decode the compiled program and so the maximum execution speed is obtained. However, the instruction set of a particular computer is unlikely to be optimal for implementing the operations required by any particular language, and the memory organisation will not necessarily be well-suited to representing the data types the language provides. As a consequence of this, it can be the case that conceptually simple operations in the source language require long sequences of machine instructions for their implementation; consequently, the compiled version of a program can be very bulky. This effect can be mitigated by the use of optimisation techniques in the compiler, but this may slow down the compilation process considerably and increase the amount of workspace required by the compiler.

The organisation of interpreter-based systems varies considerably; several possible organisations are described in Section 1.3. If the source language is appropriate it can be interpreted directly, but more often it is translated into an internal form (referred to as 'interpretive code') which is then used by the interpreter to direct the execution. The individual elements of the interpretive code will be chosen so that the source program can conveniently be expressed in terms of them. The runtime system will consist of a set of routines corresponding to the elements of the interpretive code. (These routines may be written in a high level language rather than the machine code of the computer on which the system is to run.) Since the runtime routines comprise code sequences to perform the operations of the source language, they will be as bulky as the compiled code produced for the same operations. However, only one copy of each routine is required and the interpretive code itself will be compact. Therefore, if each operation occurs many times in a program (as is usually the case) the total size of the interpretive system and code will be smaller than the corresponding compiled version of the program.

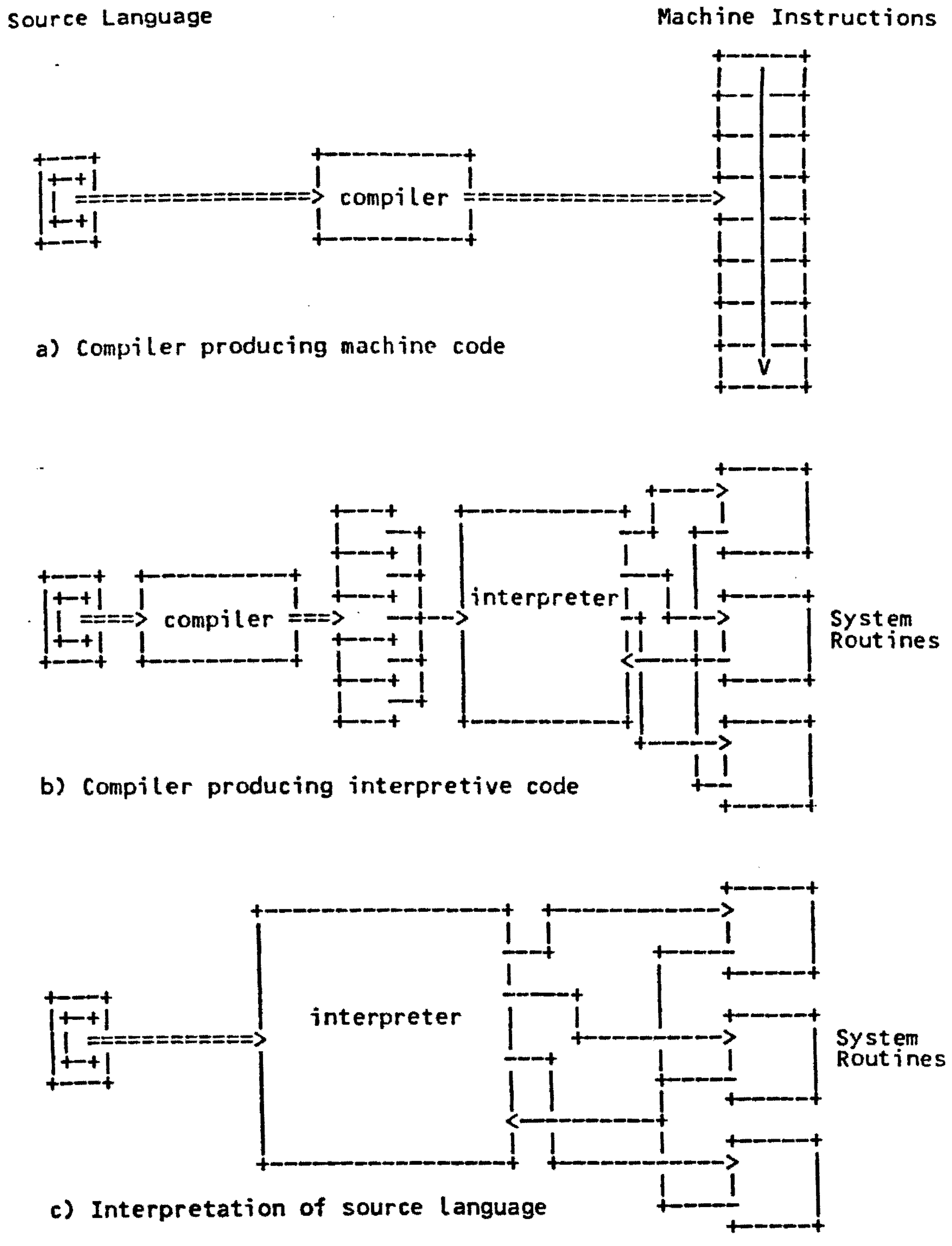
Although it will be practically important to the implementor of a language to retain the distinction between compiler-based and interpreter-based systems, it can be instructive to consider all implementations as including both a compiler and an interpreter. The compiler extracts a certain amount of information from the source program and uses it to perform a translation into some interpretive code which is then used to direct execution. At one extreme, the interpretive code is identical with machine language, the 'system routines' are implemented in hardware, and the flow of execution simply consists of the sequential execution of the instructions, which may include branching instructions. At the opposite extreme, the interpretive code is identical with the source language and all the information in the program is extracted by the interpreter. In between these extremes, is a continuum of implementations with different levels of interpretive code, in which different amounts of information are extracted by the compiler and interpreter.

Since any system must extract the syntactic information contained in the source in order to determine the structure of te program, all implementations must deal somehow with the problems associated with parsing. If the internal form used by the implementation is not identical with the source language some form of code generation will also be required.

Figure 1.1 is intended to illustrate the variety of implementations.

Before going on to practical matters relating to implementations, there is one other aspect of interpreters which deserves mention. One way of defining the semantics of a programming language is by appealing to the behaviour of a particular compiler. This has obvious drawbacks, in particular, the dependence on the operation of a particular computer and the difficulty of relating the behaviour of any other compiler to the standard in anything other than an experimental manner. Therefore, the idea arose of defining semantics by the use of an <u>abstract</u> <u>interpreter</u>, the operations of which can be precisely defined mathematically. Early work on such definitions goes back to [McC66,LAN64]; the most successful application is the Vienna definition of PL/I [LW71]. An account of different approaches to the use and specification of abstract interpreters, as well as some of the difficulties arising therefrom is given in [REY72].

Figure 1.1 Compilers and Interpreters



a) Compiler producing machine code

b) Compiler producing interpretive code

c) Interpretation of source language

==> data transmission
--> control flow

1.2 THE CHOICE OF IMPLEMENTATION TF^HNIQUE

A number of programming languages may be said to constitute a mainstream in language design at present. This would certainly include Algol68, Cobol, Fortran, Pascal and PL/I. There is also an identifiable mainstream in hardware design and, perhaps not surprisingly, the facilities provided by the latter are well-suited to implementing mainstream languages. These facilities include integer and real arithmetic, conditional branching, loops controlled by counters, subroutine calls, and index registers suitable for implementing array-like data structures. All other things being equal, the superior execution speed obtained in a compiler-based system will make this the preferred implementation for such a language on such a machine. It may be the case, though, that there are restrictions on space on a small machine or one with an addressing mechanism which imposes a limit on the address space. If this is so, then an interpreter-based system may be preferred because of the compactness of interpretive code.

For this reason, the choice between an interpreter and a compiler is often presented as a choice between a small, slow system and a large, fast one. This naive view is inadequate. Both the nature of the language and of the machine may affect the choice in a variety of ways and so may the environment in which the system is to be developed and used.

Outside the mainstream of language design there are some languages which deal with objects which are not directly implemented by hardware; the phrase 'high level data types' is used to describe such objects. Snobol4 (strings, tables) and APL (vectors) are well-known examples of such languages. Primitive operations in these languages require complex machine operations for their implementation, compared with which the decoding overhead of a suitably designed interpretive scheme may be acceptable, whereas the size of the machine code produced by a compiler may be unacceptable on all but the largest machines.

The choice between compilation and interpretation is also influenced by the amount of manifest information (information contained in the source text) available to a compiler. If sufficient information is available for the compiler to select a sequence of machine instructions which will not be affected by the runtime values of variables in the program, compiled code of very high quality can be produced. A particular requirement is for the type of variables to be determinable at compile time, since, if they can vary, type checking code must be generated. For programming languages with dynamic data types interpretation is usually preferred, particularly since dynamic typing is often found in languages with high level data types such as those mentioned in the previous paragraph.

In summary, if, for reasons such as those just outlined, the overhead of executing a system routine considerably exceeds the decoding overhead associated with the use of interpretive code, then the use of a compiler producing hard code ceases to be necessarily the best method of implementing the language. Whether this is so will depend on the machine or machines on which the system is to run, as well as on the language. The introduction of the first generation of micro processors has seen a return to primitive instruction sets, which are not adequate for implementing high level languages: a prime example is the lack of floating point operations. For this reason, as well as the restricted addressing space, implementations of high level languages for micros are almost always interpreter-based.

Interpreters possess other advantages. It is easier to write an interpretive system which performs acceptably than it is to write a code generator to produce high quality machine code. Furthermore, a system generating machine code is *ipso facto* machine dependent, but an interpreter need not be so. The system routines can be written in a widely available programming language, and the interpretive code format can be designed so that it is not dependent on particular machine characteristics. There is no need to redesign the generated code sequences or produce register allocation schemes to cope with new processor configurations, so the task of transporting a system becomes considerably less complex.

Finally, the environment in which the system is to be used must be taken into consideration. Interpreters are usually thought preferable for interactive use (see, for example [BRO79]). Even in a non-interactive environment, if more time is spent developing programs than actually running them when they work, then compilations will be frequent. The obvious example of such an environment is a university, in which students' programs are rarely run at all once they work sufficiently well to satisfy a course requirement. Under these circumstances, a fast translation into a suitable interpretive code may be preferable to a compilation involving considerable optimisations aimed at producing the most efficient machine code possible. The latter option would, however, be preferred in a production environment, where the costs of compilation will only rarely be incurred and it is desirable that execution be as efficient as possible.

Interpreters can usually produce better run-time diagnostics than compiler-based systems and in an environment where much debugging goes on this may be considered important. There are many languages in existence for which a compiler cannot provide complete security; since an interpreter retains at run-time much information (such as the symbol table) which is thrown away before execution of a compiled program, it is able to produce more helpful post-mortem dumps if a run-time error occurs and can often provide sophisticated tracing facilities.

1.3 INTERPRETIVE STRATEGIES

If the source language itself is used as the interpretive code, the path of execution is determined by the syntax of the language. For example, the interpreter might use some form of shift-reduce parser, with the interpretive routines being called whenever a reduction was made. (The use of any parsing algorithm which requires back-tracking is ruled out for this purpose.) This approach is particularly simple and does not require the generation of any form of intermediate code; it can be implemented in such a way that parts of the program which are not executed on a particular run will not get parsed at all. The objection to interpretation of the source is that if the program contains a loop or a procedure which is called more than once, then the overhead of syntax analysis is incurred every time the body of the loop is executed or the procedure is called. Unless the language is very unusual, the information extracted by this analysis will be the same every time, and it is clearly preferable to perform the analysis once only. For this reason, direct interpretation of the source is rarely used.

More often, lexical and syntactic analysis are carried out on the source language before execution. If it is appropriate to the particular language, semantic attribute processing (e.g. type checking and declaration processing) will also be carried out. The output from this phase may take several forms. Perhaps the simplest to produce is a parse tree: the corresponding interpreter would be a tree automaton, which performed a tree walk. This form of internal representation is suitable for incremental compilers and systems where the user can edit the program during execution.

Greater efficiency of execution can be obtained by using an interpretive code which is a flattened representation of the parse tree. The obvious choice is a reverse Polish string, which can be evaluated on a stack. The operators appearing in the string can be the same as the source language operators, with system routines being written to correspond to them. Such a Polish string probably provides the most economical representation of the program. If the operators and operands are encoded carefully, the interpretive code can be made extremely compact, but the ultimate in code compression can only be obtained at the expense of portability, as it depends on machine features such as the number of bits in a word and the address size.

If more processing is performed by the compiler, a lower level of interpretive code can be generated, in which the source language operations are represented by the composition of more primitive operations which more closely resemble machine instructions. This may permit the detection of special cases for which shorter sequences of code can be generated. As the level of the interpretive code becomes lower, the code becomes bulkier and, up to a point, the execution speed increases. There comes a point where the effect of the interpreter's decoding overhead has to be taken into account and the system's performance deteriorates.

The lowest level of interpretive code which is widely used is designed in imitation of the machine codes of real computers. A virtual machine is defined, sometimes with a stack architecture, but often with one or more virtual registers. A set of instructions is provided for the virtual machine. An instruction is packed into a virtual machine word (which has to be mapped onto real machine words) and contains fields with an opcode, possibly a register and some form of effective operand address, which may involve indexing and indirection. The interpreter performs the function of the control hardware or microprogram in decoding the instruction and performing the operation. This form of interpretive code is most useful for specialised applications, connected with the generation of actual machine code. Examples are provided by Intcode [RIC72], which is used during the bootstrapping of BCPL, and by the Janus interpretive testbed [POO78].

The remainder of this thesis is concerned with investigating the properties of a form of interpretive code known a indirect threaded code, which differs somewhat from those just described. Although the code is organised as a reverse Polish string, pointers to the system routines are embedded in the code in such a way that no interpreter is required to select the routine - in a sense, it may be said that the code interprets itself, as will become clear. The next four chapters provide a detailed description of one system in which indirect threaded code has been successfully used. Following this, a more abstract description of such interpreters will be presented.

CHAPTER 2

INTRODUCTION TO SETL-S

2.1 PRELIMINARY DESCRIPTION

Setl-s is the name given to a portable and compact implementation of a large subset of the language Setl. The language derives its name from the fact that finite sets are one of its basic data types, and notations derived from mathematical set theory appear in the syntax, allowing operations such as union and intersection to be written in programs in a natural way using infixed operators (although the restricted character set available on most computers does not permit the full range of symbols). The language is designed to relieve the programmer of the job of specifying the storage structures to be employed to represent sets in the memory of a computer, and the access and updating algorithms which go with them. No mechanism is provided for the specification of programmer-defined data types nor for the use of pointers, so the viability of Setl rests on the contention that algorithms are most conveniently expressed as operations on sets. Also, typing is dynamic so that Setl programs do not contain the redundant information required for extensive compile-time checking or for verifying the correctness of programs. The design philosophy has been summed up as one of 'making it easy to write good programs' as against 'making it hard to write bad programs' [DEW78]. An account of the design of Setl can be found in [SCH76] and a full description of the current version in [DEW79].

Setl was designed and first implemented at New York University. The NYU system (which will be referred to as NYU Setl or full Setl) is large and slow, consisting of four separate, overlaid phases. It was written in a specially designed implementation language known as Little, which is not widely available and although it was designed to be portable has proved to be difficult to transport. (Furthermore, since the language is particularly unattractive there is no great incentive to implement it widely.) Consequently, NYU Setl itself is not very portable and, indeed, its excessive size rules out the possibility of implementing it on small machines. (See Chapter 5 for some figures on the size of the system.)

The -s in Setl-s stands primarily for 'subset' but it also stands for 'small', reflecting the basic design objective of producing a compact system, especially suitable for use on mini computers and large micros. The subset of the language which is implemented comprises roughly 75% of full Setl, the most important omissions being the 'representation sub-language', a system of declarations which gives the programmer some control over data structure choice, and the ability to break programs into separately compiled modules.

The following section describes the language features which are implemented in Setl-s; Figure 2.2, which appears at the end of the description, lists the differences between Setl-s and the full language. The listings of several Setl programs, which will be discussed in Chapter 5, appear in Appendix 1.

2.2 THE SETL LANGUAGE

2.2.1 Data Types

Setl provides the familiar types integer, real, boolean and string. In Setl-s, both numeric types have a range limited to that available on the host machine; literals are written in the normal way (e.g. 124, 3.14159, 2.0E7). Strings are sequences of characters of arbitrary length (subject to an implementation dependent limit); they can be written enclosed in single quotes (e.g. 'hello'). Tuples are ordered sequences of values, which again may be of arbitrary length. They resemble one dimensional arrays or vectors; the values need not all be of the same type. Sets are unordered collections of values which do not include duplicates - an attempt to add a value which is already present to a set has no effect. Both sets and tuples may include sets and tuples among their members. Literals are provided for both types: a list of values separated by commas is written between set brackets { and } or tuple brackets [and] (e.g. {1, 3.4, {'a', 'b'}}, [1, 2, 3]). The values may be any sort of expression.

Sets all of whose elements are tuples of length 2 (pairs) are referred to as maps. The first element of each pair is treated as a domain value, with the second element providing its corresponding range value. A functional notation (see 2.2.2) is provided for accessing and updating maps. It is by using maps which represent the relationships between elements of a structure that a Setl programmer can represent structures such as graphs.

Objects known as atoms are used to build data structures when unique tags are required, for example to label nodes of a tree containing duplicated values. The only property of an atom is that it has a value which is different from anything else. A supply of atoms is provided by the system function NEWAT.

Values of all these types can be assigned to variables. The type of a variable is dynamic and depends only on the last value assigned to it. Initially, all variables have an undefined value, omega, (written OM); this value is also yielded by certain erroneous operations. A test for equality is the only operation which can be performed on omega without an error.

2.2.2 Expressions

Expressions can be built out of literals and identifiers using the monadic and dyadic operators listed in Figure 2.1 (full Setl provides a larger number of more elaborate operators). These operators provide a full range of integer and real arithmetic and reasonably sophisticated string manipulations. The operators on sets perform the usual set-theoretic operations as explained in the table. The With and Less operators provide a convenient shorthand for adding and removing single elements; s With x is equivalent to s + {x}. This operator is commonly used to build up sets one element at a time.

Figure 2.1 Setl-s Operators

a) Monadic Operators

Operator	Operand Type	Meaning
+	integer real	affirmation
-	integer real	negation
#	set tuple string	cardinality number of elements number of characters
ABS	integer real string	absolute value character code value (#string = 1 only)
ARB	set	arbitrary element
DOMAIN	set	if set is a map, yields domain set
FIX	real	convert to integer
FLOAT	integer	convert to real
NOT ~	boolean	logical negation
RANGE	set	if set is a map, yields range set
STR	any	yields string representation
TYPE	any	yields string giving type
VAL	string	for suitable strings, converts to numeric value

Figure 2.1 Setl-s Operators

b) Dyadic Operators

Left opnd	Operator	Right opnd	Meaning
integer real set string tuple	+	integer real set string tuple	integer addition floating point addition union concatenation
integer real set	-	integer real set	integer subtraction floating point subtraction set difference
integer real set string	*	integer string real set integer	integer multiplication replication floating multiplication intersection replication
integer real	/	integer real	floating division
any	=	any	equality test
any	/=	any	inequality test
integer) string) real)	< <= > >=	(integer (string (real	integer comparisons lexical comparisons floating point comparisons
integer	DIV	integer	integer division
string any	IN	string set tuple	sub-string test membership test
set	INCS	set	inclusion test
set	LESS	any	removes element
set	LESSF	any	if set is a map, removes pairs for given domain
string any	NOTIN	string set tuple	inverse of IN
integer	REM	integer	remainder after division
set	SUBSET	set	subset test

Several additional forms of basic operand are available. Elements of a tuple can be written as, for example, t(i) which selects the ith element. If i exceeds the current length of the tuple omega is yielded as the value. Similarly, map references can be written: f(x) searches the set f for a pair whose first element is equal to x and returns the second element of the pair. If no such pair exists omega is returned; if the set contains more than one such pair or any elements which are not pairs an error occurs. In the former case the notation f{x} is used to give the set of all range elements for the domain value x.

Set and tuple operands can be formed using special notations. Tuples which consist of sequences of integers are written in a style exemplified by [2...100], which gives a tuple whose members are the integers from 2 to 100. The general form is [first,next...last], where first is an expression giving the initial value, next is an expression giving the second value, which indicates the step size and direction, and last is an expression giving the final value. This notation can also be used to form sets, the ordering not being significant; thus, an idiosyncratic way of forming a set of even integers less than 100 would be {100,98...1}.

A more general type of set and tuple former uses an iterator. The general form of iterator is identifier IN expression, where expression yields a set or tuple. The notation {expression: iterator} yields a set whose elements are the successive values of the expression obtained as the identifier in the iterator takes on the value of each element in its expression, in turn. Thus {[a*a, a]: a IN [1...10]} gives a square root map (e.g. $\dagger(16) = 4$). The iterator can be followed by a test to indicate that only some elements are to be used; thus to skip the pair [25, 5] the previous example would have to be {[a, a*a]: a IN [1...10] | a/=5}. The bar | is used to separate the test; it may be read as 'such that'.

Function calls, both of system functions and user-defined functions (see 2.2.4) can be used in expressions.

2.2.3 Statements

Most of the statement forms in Setl are conventional, resembling a hybrid of Pascal and Algol68. An assignment is of the form left_hand := expression, where left_hand is either an identifier, a tuple reference or map reference. It is legitimate to assign to a non-existent element of a tuple or map - the effect is to create an extra member. Assigning operators, such as +:= are also supplied. One special assigning operator is From. x From s is equivalent to the sequence x := Arb s ; s Less:= x. It is thus unusual in that it implicitly assigns to both operands. Both assigning operators and the assignment can be used within an expression as well as standing alone as a statement.

Conditional execution is provided by an If-statement and a Case-statement with the familiar semantics. A point of syntax to note is that Setl does not have blocks delimited by Begin and End. Instead keywords such as If and Then act as block delimiters as in Algol68. The keyword End is used to close all constructs (as against Fi, Od etc.). If-expressions and Case-expressions are also available for convenience.

There are two forms of loop in Setl. The first, sometimes called a 'full iterator' is a portmanteau construct with several clauses: an Init clause to be performed on entry, a Doing block to be performed at the start of each iteration; this is followed by a While test, after which the body of the loop is performed. Next comes a Step block, which is performed at the end of every iteration, followed by an Until test. Finally, a Term block is performed on exit from the loop. Any, or all, of these clauses may be omitted, which permits the synthesis of a wide variety of loops.

The other form of loop is controlled by an iterator of the type described in connection with set formers. It has the syntax:
(For identifier IN expression) body End. The effect is to execute the body of the loop with the variable in the iterator taking the value of each member of the set or tuple yielded by expression; if the expression is a tuple the values are yielded in order, if a set they are yielded non-deterministically. As with formers, an iterator controlling a loop may include a 'such that' test. Iterators may be combined (e.g. (For x IN s, y IN ss)) giving the effect of nested loops.

Loops can also be used as Boolean expressions in the so-called 'quantified tests'. These are of the form keyword iterator | test, where keyword is any of Exists, Notexists or Forall. The effect is to perform the test on each value yielded by the iterator; the final value agrees with the intuitive meaning of the keyword. Once the value has been established an exit is taken from the loop and the variable in the iterator may have a value depending on the condition causing the exit. These tests are not the quantifiers known in symbolic logic.

Within the body of any loop the commands Quit and Continue may be obeyed. The former causes immediate exit from the loop, the latter causes the rest of the body to be skipped and the next iteration to be performed.

Commands can be made to yield a value by the use of Expr. Within the scope of an Expr commands are obeyed until the command Yield expression is encountered when the value of the expression becomes the value of the Expr block.

2.2.4 Miscellaneous

A Setl program consists of a header followed by the main body followed by procedure declarations, which are conventional in format. Setl-s insists that procedure names be pre-declared in a Procedure statement following the heading, to facilitate one-pass compilation. All procedures return a result, which is omega if there is no explicit one; the procedures may be called either as functions within expressions or as routines standing alone as statements, in which case the result is ignored. In Setl-s all parameters are passed by reference, which is more a bug than a design feature.

In reading the programs in Appendix 1 it should be noted that Setl allows the keywords Program, Procedure and Continue to be abbreviated Prog, Proc and Cont respectively. Also, to cope with the possibility that {, }, [,] and | may not be available these may be replaced by <<, >>, (/, /) and ST. Iterators at the head of loops may be surrounded by parentheses as described in previous sections or by the keyword pair Loop and Do. In full Setl the keyword End may be followed by tokens copied from the head of the construct it closes, to aid readability and provide a check on matching of Ends. Setl-s does not support this feature in full, but it allows If, Case and Loop to be matched by End If, End Case and End Loop; likewise Prog and Proc may be appended to the matching End. If the keywords do not match an error is reported.

This description of Setl has necessarily been rather sketchy; interested readers are referred to the references. Some other detailed points will be described as required in the account of the Setl-s system.

2.3 THE SETL-S SYSTEM

2.3.1 Background

The design of Setl-s was influenced by the macro Spitbol implementation of the Spitbol dialect of Snobol4[DM77]. This has proved to be an efficient and highly portable implementation of that language. Given the similarities between the Spitbol and Setl languages, and the design objective of Setl-s, it would seem feasible

Figure 2.2 Differences between Setl-s and full Setl

Major Omissions

Module structure

Representation sub-language

Macros

Backtracking

Compound operators

Labels and GOTOs

User-defined operators

Some operators

String and tuple slices

Arbitrary precision integers

Restrictions

No tuples on left hand of assignments

No multi-dimensional map references (e.g. f(a, b))

Only simplest iterator forms

Procedures must be pre-declared

Arguments passed by reference

Limitations on tokens following End

Constants in Const and Init declarations restricted to numbers and strings to suggest that the implementation approach used in macro Spitbol could profitably be adapted for use in Setl-s. A comparison of the two languages, which brings out the relationship between the two systems is summarised in Figure 2.3. The main point of similarity is the high level data types supported, which in both cases demand the use of a heap-based storage allocation scheme, which in turn calls for an efficient garbage collector. To correspond with these data types, both languages provide operations which are not supported by conventional hardware, so that interpretive code is an attractive implementation technique. This is confirmed by the fact that both languages are dynamically typed.

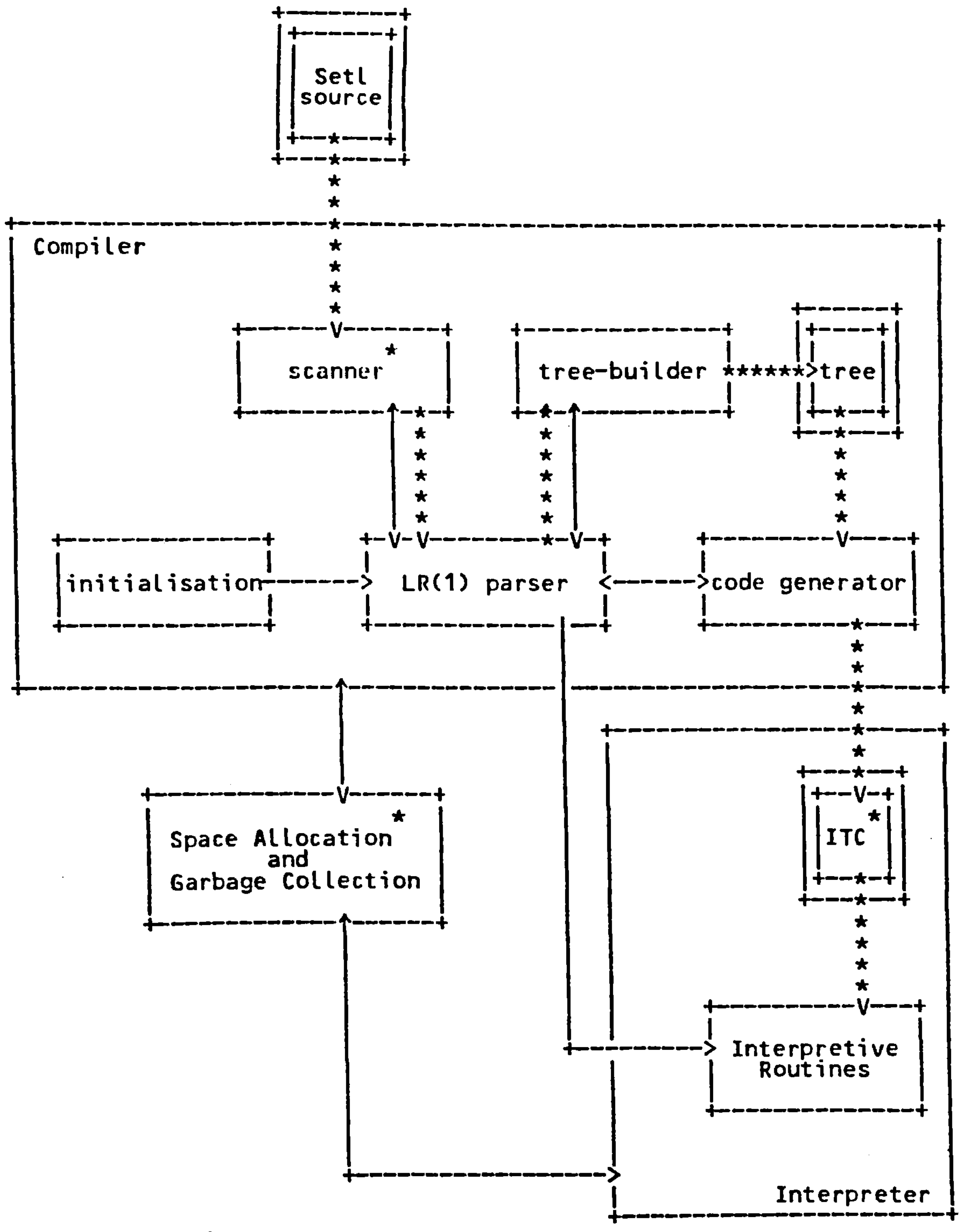
There are however sufficient differences between Setl and Spitbol to demand extensive modifications to the design of macro Spitbol if a sensible implementation of Setl is to be produced. The precise nature of the languages' data types differs considerably, and the approach to polymorphism and mixed-mode operations is quite different. The biggest difference is that Setl requires a much more sophisticated syntax analysis than Spitbol does, although since there is no runtime compilation of code, this analysis can be entirely separated from the runtime system.

Figure 2.4 shows the structure of Setl-s. For reasons outlined in Chapter 1, the source is translated into a form of interpretive code. The part of the system which performs the translation is referred to as the Setl-s compiler, and forms the subject matter of Chapter 4; it uses a novel form of LR(1) parser to perform syntax analysis. The code which is generated from the parse tree is the

Figure 2.3 A Comparison of Snobol4 and Setl

Snobol4	Setl
dynamic types	dynamic types
sets, maps, tuples	strings, patterns, tables
set operations, iterators map references	pattern matching
many polymorphic operators	some polymorphic operators
no mixed mode operations no coercions	coercions
conventional program structure	modified Markov algorithm
complex syntax	simple syntax
	run time compilation of code

Figure 2.4 Structure of the Setl-s System



--> control flow
**> data transmission
* components derived from macro Spitbol

indirect threaded code, briefly mentioned in 1.3; the code format and interpretive routines will be described in Chapter 3. A space allocation scheme interacts with all part of the system.

2.3.2 Minimal

Setl-s is written in the implementation language Minimal, which had been developed for macro Spitbol. This has allowed parts of the former system to be directly incorporated in the new one. 'Minimal' is an acronym for Machine Independent Macro Assembly Language. As this suggests, it resembles the assembly language of real machines, but is defined in a machine-independent way, as the assembly language for a non-existent virtual machine. It is implemented on a particular machine by macro-expansion into the target machine-code. This is a fairly efficient process, with expansion ratios from Minimal to target code as low as 1:1.2 for machines with a modest set of registers , and so Minimal programs execute with an efficiency approaching that of machine code. It is important for the runtime interpretive routines of a system such as Setl-s to be coded as efficiently as possible, hence the attraction of Minimal, despite the inconvenience of programming in it.

Minimal is described in some detail in [DM77]. Its design was influenced by the original application, so it has features which are particularly useful for string processing. The Minimal virtual machine resembles most conventional machine architectures (particularly that of the PDP11). It has two general-purpose index registers, XL and XR, a stack pointer register XS , three work registers and separate integer and real accumulators. Operations are provided for integer, address and (optionally) real arithmetic, jumps and tests as well as the character handling operations alluded to above.

Some features of the language are provided specifically to assist with the coding of interpreters, notably a code pointer CP, and appropriate operations on it. Machine-dependent quantities, such as the number of bits in a word, and the size of basic addressing unit and address, are incorporated as parameters in the Minimal program.

2.3.3 Memory Organisation

Given the basic design approach to Setl-s, it is possible to take parts of the actual code of macro-Spitbol and incorporate it into Setl-s, thereby saving a certain amount of routine coding. The major saving afforded in this way comes from using the Spitbol garbage collector, an efficient, compacting garbage collector, which is already debugged and well understood. It does, however, impose certain restrictions, which have to be rigorously observed. The ones which affect the implementation strategy are:

1. no data object in collectable memory may have pointers to it, other than to its first word.

2. a structure containing pointers must have them all in a contiguous block, possibly interspersed with recognisable non-pointers.

3. small integers must be distinguishable from pointers; this is done by restricting dynamic memory to start at some threshold value: all pointers therefore exceed this threshold, with small integers falling below it. This imposes a limit on the maximum permissible size of objects whose size is specified by such an integer.

Additional restrictions on the contents of registers when the collector is called have specific effects on the coding.

Implicit in the adoption of the garbage collector is the adoption of the particular memory organisation used with it in macro Spitbol. Memory is divided into a static area, which is allocated once and for all and not subject to garbage collection, and a dynamic area in which space is allocated and released as execution proceeds, with freed space being recovered by the collector.

As well as the collector-imposed restrictions, a further restriction, imposed by Minimal, to assist portability, is that pointers must occupy a full word. (Hence, it is not possible to pack a pointer and, for example, some marker bits into a single field of a block.) The combination of these factors exerts a powerful influence on data structure choice, but, in fact, the structures fit in comfortably with the interpretive scheme, and starting from such a set of restrictions avoids the problem of having to fit a garbage collector into the pattern of data structures chosen by the implementor according to other criteria.

CHAPTER 3

THE SETL-S INTERPRETER

3.1 REPRESENTATION OF CODE

3.1.1 Indirect Threaded Code

The distinguishing feature of the Setl-s interpretive scheme is the format used for the interpretive code, which is known as indirect threaded code, abbreviated ITC. The code consists of a series of codewords, arranged as a reverse Polish string, with operands preceding their operators. Evaluation proceeds by loading operands onto a stack, and applying operators to the top stack items.

Each codeword is a word which contains the address of a word which in turn contains the entry point address of a system routine, which performs the function of loading operands onto the stack or applying operators. Such a pointer to a pointer will sometimes be referred to as an indirect pointer. The code pointer register (CP) points to the current codeword, in a manner analogous to a hardware program counter. Each system routine ends in a sequence of code which increments the code pointer, loads the new current codeword into a register (XR), and makes an indirect branch through the pointer in that register, i.e. control passes to the address in the word pointed at by the codeword, which will be the system routine entry point. This is achieved by the following sequence of Minimal instructions:

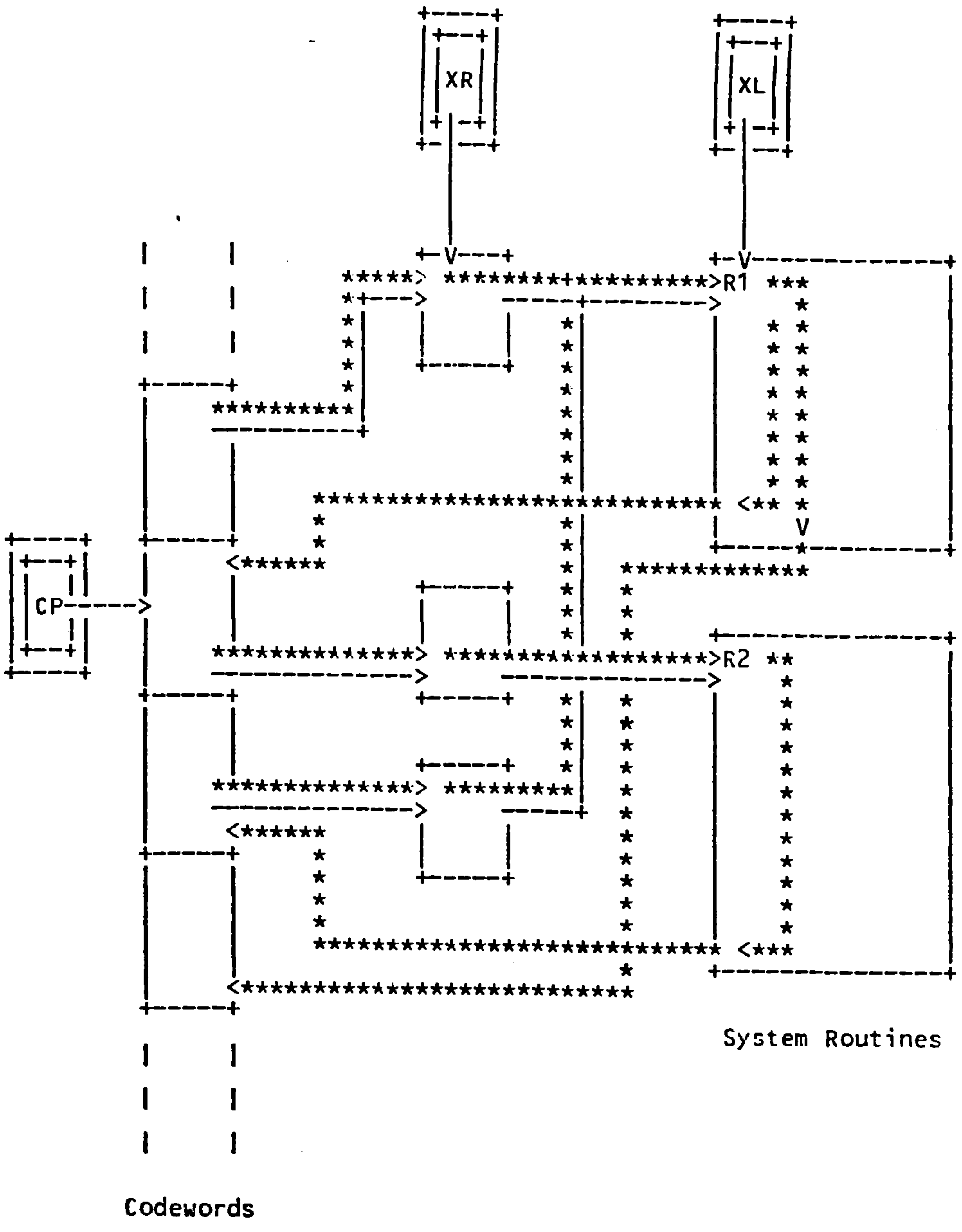
```
LCW  XR        load the codeword, incrementing CP, and possibly destroying XL
                                   into XR
MOV  (XR),XL   XL now points to entry point
BRI  XL        make indirect branch
```

The BRanch Indirect instruction transfers control to the location pointed at by its operand - Minimal restricts such jumps to locations which are explicitly defined in the program to be 'entry points'.

Figure 3.1 illustrates this flow of control.

Note that this interpretive cycle leaves a pointer in XR. This is exploited, in the case of operands, by representing each value by a block, whose first word points to a routine to load the value. This load routine can access the value (i.e. the block) via XR, so that the code produced to load any value is merely a pointer to the first word of the block. Similarly, if an operand is a variable, the codeword points to the first word of a variable block (VRBLK, see figure 3.4h), which points to the routine to load the value, while another field in the VRBLK points to the block which is the current R-value of the variable. (Following Strachey [STR67], I use the terms L-value and R-value of a variable, to distinguish between the 'location' denoted by the variable in a particular environment, its L-value, and the contents of that location, its R-value.) In the case of variables, however, another pointer is required - one to a routine to store a new R-value into it upon assignment. This pointer is held in the VRSTO field of the VRBLK, which is its second word. A simple

Figure 3.1 ITC flow of control



--> pointers
**> control threads

The registers are shown on entry to R1, through the first codeword.

assignment to a variable is represented in the code by a pointer to this field of the appropriate VRBLK. Since this means that there are pointers into the middle of VRBLKs, which violates a basic garbage collector restriction, VRBLKs are kept in the static region, and are not collected.

Figure 3.2 provides a simple example of the code format; it shows operators as simple indirect pointers to operator routines, which was the format used in macro-Spitbol. The modified operator representation used in Setl-s will be described later.

Although this format seems to be extremely elaborate, it possesses several desirable features:

the code consists of nothing but addresses, and since most computers have words which can hold an address, the code format is portable.

the code is compact.

only one copy of each system routine is required.

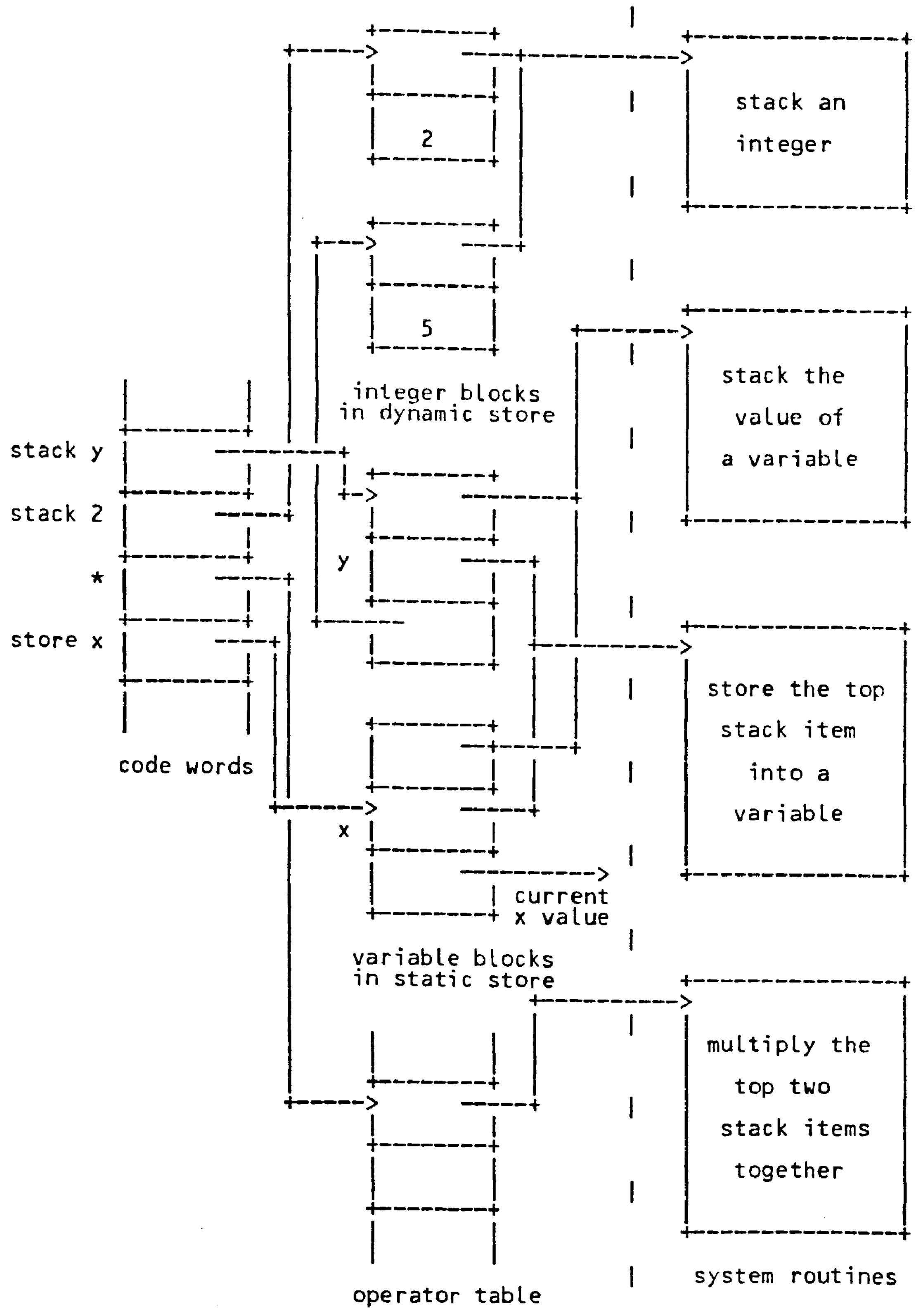
the decoding overhead is small.

These points will be elaborated in Chapter 5.

3.1.2 Transfer Of Control

So far , the ITC has been described as if it were a linear series of words, with control passing through each codeword as the code pointer is incremented. In order to represent the control structures of Setl it must be possible for control to be transferred, by conditional or unconditional jumps. A branch of this nature will be

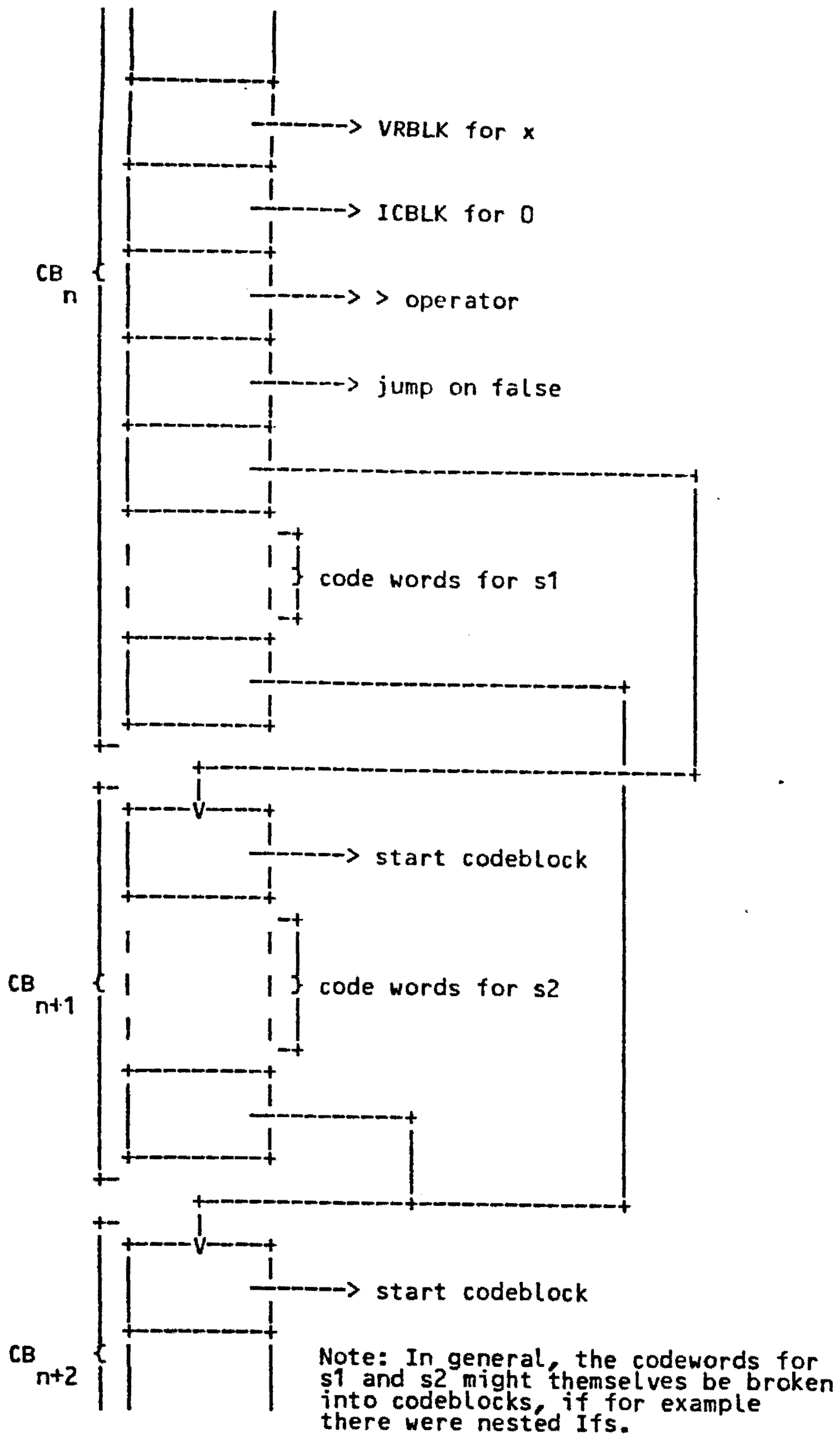
Figure 3.2 ITC generated for x:=y*2 when y=5


2
5
integer blocks in dynamic store
stack y
stack 2
*
store x
code words
y
x
current x value
variable blocks in static store
operator table
stack an integer
stack the value of a variable
store the top stack item into a variable
multiply the top two stack items together
system routines

represented by an indirect pointer to a routine to reset the code pointer and perform an indirect branch through the new codeword. This routine will have to be supplied with the destination codeword for the jump. To do this conveniently, codewords are arranged in codeblocks, the first word of each of which contains a pointer to the routine to transfer control. An unconditional jump is simply a pointer to the codeblock which is its destination; a conditional jump is implemented as an indirect pointer to a skip routine followed by an unconditional jump. The skip routine tests the truth value of the top stack item and either increments the code pointer or not accordingly, so as to obey the jump or carry on with the next word of the current codeblock. Both forms of conditional jump - jump on true and jump on false - are used, as well as more specialised conditional jumps used in iterators and set formers. Figure 3.3 shows the arrangement of codeblocks corresponding to the Setl IF-THEN-ELSE construct. Notice that control cannot drop through to CB_{n+2}, because pointers may only be to the heads of blocks.

This format has two interesting effects: it is quite easy to ensure that these codeblocks obey the restrictions imposed by the garbage collector, so that codeblocks can be built by the code generator in dynamic memory. This means that, as code becomes unreachable (e.g. after the last exit from a loop, or the final return from a procedure) the space occupied by it can be recovered by the garbage collector, and made available as workspace to the executing program. This may prove a valuable feature where memory is limited. The other thing to note about the code format is that the code can be looked at statically as a directed graph representing the

Figure 3.3 Codeblocks built for If x>0 Then s1 Else s2 End



Note: In general, the codewords for s1 and s2 might themselves be broken into codeblocks, if for example there were nested Ifs.

flow of control in the program. This graph, therefore, contains information which would be useful in performing optimisations on the generated code, and it is hoped that it will eventually be possible to make some such use of it, or else to perform optimisation dynamically as the code pointer 'traverses' the graph.

The representation of procedures has been chosen to extend these benefits to procedure calls. There is an entry in the symbol table, corresponding to the name of each procedure, which is created when the forward declaration of the procedure is processed and which has its value set to be a pointer to a procedure block (PCBLK). This PCBLK is filled in when the procedure body is encountered with a pointer to the code and certain administrative information required for handling the procedure call. When a call is found in the Setl program, a codeword is generated which points to the relevant PCBLK - the first word of the PCBLK points to a routine to call a procedure, the rest of the block providing the information required to preserve the calling environment, and associate arguments with their values (see 3.4.2). Setl does not permit function-valued variables, and so there is never any need to load a function value (i.e. a PCBLK) onto the stack; this means that the first word of the PCBLK is free to be used for the call instead of holding a load routine pointer, as is the case with most other blocks. The chosen code format is sufficiently flexible for this arrangement to fit in comfortably. After code generation has been completed, the value fields of the symbol table entries for procedures (which are chained together for easy access) are cleared to zero. Hence the PCBLKs are 'set loose' and become accessible only through the code. This means that the PCBLKs and their associated code can be reclaimed by the garbage collector when they become unreachable.

Labels and GOTOs are not permitted in Setl-s, since the language contains adequate control syntax for them to be unnecessary as well as undesirable; furthermore, the implementation of labels involves restoring the environment after a jump, a problem which would complicate the system to little purpose.

3.2 REPRESENTATION OF VALUES

3.2.1 Operand Blocks

Every value is represented by a contiguous block of two or more words, divided into fields; there is a separate block type for each Setl datatype, with pairs and maps being treated as distinct types. Figure 3.4 shows the various block formats. Several conventions are used in these diagrams and the following text.

The block and field names follow the Minimal rules for names, each being five characters long, the first three of which must be alphabetic. There is a 2 letter code for each type, e.g. IC for integer (constant), ST for set, and the block is referred to as an xxBLK, where xx is the code. Similarly, all fields' symbolic names begin with the type code, the remaining three letters being a mnemonic for the field's use. In the diagrams, fields shown delimited by solid vertical lines are one machine word long, those delimited with * are one or more words long, depending on the value of some machine parameter (e.g. the length of the RCVAL field of an RCBLK will be the

Figure 3.4 Setl-s Block Formats

a) ICBLK - integer

```
+---------+
|  ICGET  |       type pointer
+---------+
*  ICVAL  *       value
+---------+
```

b) RCBLK - real

```
+---------+
|  RCGET  |       type pointer
+---------+
*  RCVAL  *       value
+---------+
```

c) SCBLK - string

```
+---------+
|  SCGET  |       type pointer
+---------+
|  SCLEN  |       no of characters
+---------+
/         /
/  SCHAR  /       characters (packed)
/         /
+---------+
```

d) ATBLK - atom

```
+---------+
|  ATTYP  |       type pointer
+---------+
*  ATVAL  *       print value (integer)
+---------+
```

e) STBLK - set
(MPBLK - map is identical)

```
+---------+
|  STTYP  |       type pointer
+---------+
|  STLEN  |       block length
+---------+
|  STNEL  |       number of elements (cardinality)
+---------+
/         /
/  STELS  /       pointers to elements
/         /
```

Figure 3.4 Setl-s Block Formats

f) TPBLK - tuple

```
+---------+
|  TPTYP  |      type pointer
+---------+
|  TPLEN  |      block length
+---------+
|  TPNEL  |      number of elements (cardinality)
+---------+
/         /
/  TPELS  /      pointers to elements
/         /
+---------+
```

g) PRBLK - pair

```
+---------+
|  PRTYP  |      type pointer
+---------+
|  PRDOM  |      pointer to first element
+---------+
|  PRRNG  |      pointer to second element
+---------+
|  PRNXT  |      link pointer
+---------+
```

h) VRBLK - variable

```
+---------+
|  VRGET  |      pointer to load routine
+---------+
|  VRSTO  |      pointer to store routine
+---------+
|  VRVAL  |      pointer to current value
+---------+
|  VRNXT  |      hash chain link pointer
+---------+
|  VRNML  |      name length (proc number in leftmost bits)
+---------+
/         /
/  VRCHS  /      characters of name (packed)
/         /
+---------+
```

number of words required to hold a floating point number). Areas delimited with / indicate a variable number of identical, one-word fields (e.g. the members of a set in a STBLK). Blocks with such an area have also a fixed number of administrative fields; the variable s is used for the number of these fields, in the following descriptions. Comments on the significance of each field are appended to the right of each diagram.

Most blocks are allocated in dynamic memory and are subject to relocation by the garbage collector, as well as being reclaimed by it when they cease to be active.The exceptions are system constants, such as OM, TRUE, FALSE and the null set (map), tuple and string, and also atoms created by NEWAT. These all reside in static, in a single copy, so they can readily be identified or compared using only their addresses.

As explained above, the first field of every block contains a pointer to a system routine to load its value onto the stack. Since the stacked value is merely a pointer to the block, and all pointers are just words containing addresses, it might be thought that a single load routine was all that was required. In practice, a different routine entry point is defined for every type, although the routines all share code. In this way, the pointer serves as a type code for the block, as well as performing its function as part of the interpretive scheme.

The blocks can be divided into two groups. The first of these consists of <u>atomic</u> values whose internal structure is not decomposable, and which are never modified once built. This group comprises ICBLKs, RCBLKs, SCBLKs and all of the static blocks. Any operation yielding an atomic value creates a new block to hold it: e.g. performing the addition 2 + 2 leads to the creation of a new ICBLK with value 4. The cost of this is low, and a great deal of potential trouble with shared pointers is avoided. All the other blocks have a decomposable structure, and can be modified in place - the latter is inevitable on efficiency grounds, since creating, for example, a new MPBLK every time a map was modified would rapidly exhaust memory, as well as slowing down execution intolerably.

3.2.2 Sets And Tuples

The structures chosen for sets and tuples are very simple. Sets are represented as linear hash tables. Pointers to the elements of the set are kept in the STELS fields, with empty slots being occupied by OMs. The fields at the head of the block hold the total block length, which is used by the garbage collector as well as by the routines for accessing set elements, and the number of elements, which is the cardinality of the set. Overflow of the hash table is dealt with by rehashing the entire set into a new, larger block. When a block of length N + s becomes full, a new block of length 2N + 1 + s is allocated. The smallest value of N is 11, giving the sequence 11, 23, 47, 95, 191, 383, ... for the number of hash slots in sets. Most of these are primes, which should help reduce collisions. A block is deemed full if n/(N+s) >= 0.7 which varies between 70 to 80% occupancy as the effect of s changes. This would seem to be fairly optimal [HOP69]. Since a linear regime is used to deal with collisions, a deletion marker must be lef. after an element has been removed. 0 is used for this purpose.

A tuple has a length field, giving the total number of words in the block, a cardinality field, which gives the index of the highest element which is not OM, and then pointers to its elements, in order. A tuple with cardinality n will be a block of length N >= n+s, with elements beyond the nth filled with pointers to OM. When the block gets full (n > N-s), a larger one is allocated, using the same allocator as for sets, with all the values being copied from the original. Accessing t(x) is easy - if x <= n, then return the element at the appropriate offset, otherwise return OM. Updating t(x) is more complex,since the cardinality may be affected, if either x>n, or if x=n and the value assigned is OM, when it is necessary to find the last non-OM value, which need not be t(x-1). The details are straightforward, albeit tedious.

Set and tuple blocks are never contracted again once they have expanded.

3.2.3 Pairs And Maps

Maps are a distinctive feature of Setl, and the way in which they are defined in the language presents particular implementation difficulties. According to the Setl language definition, a pair is merely a tuple of length 2, and a map is a set containing only pairs. However, elements of a map can be accessed using the map notations f(x) and f{x}. It is desirable to use a representation for maps which facilitates such references without complicating the treatment of maps as sets where the context requires it. This is achieved by using the standard linear hash table set representation for maps, and using a special pair representation which is suitable for map references when stored in the table. The type word of the map block serves as an indicator that the set is known to consist entirely of pairs.

The pair representation is shown in figure 3.4g. When added to a set or map, the pairs are hashed on the domain value only, so that pairs with the same domain value will hash to the same location in the MPBLK; the PRNXT field is used to chain such entries together for the multi-valued map case. To facilitate iteration through maps, such chains are terminated by a pair whose PRNXT field contains the offset to the next entry in the MPBLK - a simple test for a pointer will detect the end condition, so iteration can be controlled by a location which contains either an offset into the block, or a pointer to a pair in a chain.

Although this scheme means that special action must always be taken to add a pair to any set or map, and that iteration is complicated somewhat, sets and maps are indistinguishable for most purposes, and map references are straightforward. (In the case of multi-valued map references f{x} it is necessary to form the result set explicitly, but the alternative of keeping the range as a set has its own disadvantages, such as the need to distinguish between {[1, {1,2}]} and {[1, 1], [1, 2]}.) A slight problem with not making the distinction between single-valued and multi-valued maps is that a reference f(x) to a multi-valued map is supposed to produce an error, whether or not the range for x is itself multi-valued. In Setl-s the error cannot be detected until reference is actually attempted to a range with more than one element.

Making pairs and maps into different internal types introduces the problem of conversion between types.

Whenever a pair or tuple is created or modified it is straightforward to detect whether its cardinality has become equal to 2; however, changing a tuple into a pair (or vice versa) involves a restructuring of the object which will not always be worthwhile, since a tuple which is being built up either by an iterator or a series of assignments will pass through the stage of being a pair. Instead of always doing this check, therefore, tuples which are created explicitly with two elements are made into pairs. This reflects the typical ways of forming maps:

```
{[1, 1], [2, 4], [3, 9], [4, 16]} or

{[a, a*a]: a IN [1...4]}
```

If elements are added to a pair, or OM is assigned to either of its elements, the pair is converted to a tuple. By suitable additions to and deletions from tuples, it is possible to create one of length 2 which is not in pair format, so when context demands a pair, and the value is a tuple, a check is made, and if it turns out to be a pair, it is rebuilt as a PRBLK.

A similar strategy is adopted with respect to maps. The null set can be treated as a map whose range is everywhere undefined; sets are always created by adding elements to the null set or to an empty block which will also have type map and the procedure which adds the elements can easily determine whether an element being added is a pair. If it is, the object continues to be a map, and this will be reflected by the type word. If, however, a non-pair is added, the type is changed to set. It may be the case that all non-pairs are subsequently removed, but it is difficult for the system to keep track of this. The result is that the type word of a map serves to indicate that the object is known to contain only pairs, and so map references may be performed. If the type word indicates that an object is a set, this merely implies that it is not known whether it is a map, so if context demands a map a check is made before an error is announced. If the object turns out to contain only pairs its type word is reset. Union and intersection are implemented in such a way that the result set is formed by adding elements to an initially empty set. The checks are performed as each element is added, in the usual way so the result automatically has its type word correctly set.

It seems quite clear that the implementation problems associated with maps, which are quite out of proportion to their importance, derive from a muddled piece of language design and that maps and sets should be separate types. This may offend some purists, but is a minor compromise compared to some which the language already makes.

3.2.4 Assignment And Copying

Conceptually, the way in which Setl defines the assignment of aggregate values is straightforward. Dewar describes it as follows: 'Setl treats tuples [and sets] as values when it comes to assignment.' [DEW79]. For example, after execution of the following sequence:

```
    t1 := [1, 2, 3, 4] ;
    t2 := t1 ;
 t2(4) := 999 ;
```

the value of t1 is still [1, 2, 3, 4]. Notionally, the right hand side of an assignment is evaluated to produce a value which is then used to update the location associated with the left hand. This location must be considered elastic in order to accomodate type changes and objects whose size may vary dynamically. In practice, a heap-based storage allocation scheme is used, and locations hold only pointers. However, in order to preserve the semantics defined for assignment it is insufficient to copy a pointer, the entire block which is pointed to must be copied, and a pointer to the copy used to update the location. Since the copying of objects such as sets is expensive it is desirable to avoid this operation whenever possible.

.The difference between a copying assignment and a pointer assignment only becomes important when the value to be assigned is already the R-value of some variable, or else is a member of an object which is the R-value of some variable, because it is only when subsequent modifications can affect the value of more than one variable that any difference will be detected in the behaviour of a program. Fortunately, this condition can be detected at compile-time by inspection of the parse tree, and the code generator can insert 'copy' instructions into the code, before the assignment. The syntax of expressions which must be copied is given by the following mini-grammar, where the grammar symbols represent nodes of the parse tree, rather than symbols of the input, hence precedence and parentheses need not be considered.

```
    copyvalue ::= name |
                  copyvalue assigning_operator expression |
                  copyvalue subscript
```

Examples are t1, t1(5), y +:= 3, (x := y)(i).

The syntactic forms of right hands which can safely be assigned or added to a set or tuple by a simple pointer-moving operation are expressions with operators, set and tuple formers and enumerations, or constants. Examples would be:

```
    {1, 2, 3}
    {x: x IN s | x > 4}
    x + y
    x + y +:= 3
    x WITH y
```

The correctness of this copying rule depends on the fact that an expression produces a new block to hold its value. Most of the routines for operators have been coded so that they build this block and do not modify their operands. However, the routines for WITH and LESS have been written so that the operation is performed in place on the left operand: these operators are frequently used in assigning form, when it is undesirable to produce a new block. This is exemplified by the following:

```
    s := {} ;
    (For x IN ss) s With:= x ;  End ;
```

which represents a typical way of building a set. It is more convenient to insert an instruction to copy the left operand on occasions when the operator is not in assigning form, than to try and optimise the assigning case, by generating a special codeword. Thus, in the last example given above, the result of the expression can still be safely assigned, because an explicit copying operation will have been inserted into the code.

Although this discussion has been restricted to assignment, it should be apparent that similar considerations apply to any operation whose execution can result in the storing of a value. In particular, the addition of an element to a set or tuple must be performed so that shared pointers to aggregate values do not occur.

It will be noticed that the rules just given are unnecessarily strict, inasmuch as the R-value of a variable might be an atomic value, and, as has been described, atomic values cannot be modified, so there is no harm if pointers to them are shared. It is not,in general, possible for the compiler to detect this situation, so a copy instruction is always inserted as described. The runtime system can make use of the extra type information available to it in order to lessen the number of copying operations performed. If, on entry to the copy routine, the top stack item which is to be copied is an atomic value, then no copying is performed. The cost of the operation is just one indirect threading cycle, and the production of superfluous blocks is avoided. In a similar way, when a set or tuple is to be copied, normally all of its elements must be copied. In fact, atomic objects are never copied, and pointers to them inside objects such as sets may be shared without trouble.

The foregoing discussion should indicate that the problem of copying on assignment is one of the greatest sources of complexity and potential insecurity in the whole system. Depending on one's feelings on these matters, this can either be taken as a lesson on the dangers of pointers, or an indication of the inappropriateness of a strict value semantics in practically implemented languages. It is felt, however, that the strategy adopted (which is original to Setl-s) provides a workable solution; alternatives will be examined in chapter 5, to illustrate further the complexity of the implementation issues raised.

3.3 REPRESENTATION OF OPERATORS

3.3.1 Polymorphism In Setl

Referring back to Figure 2.1, it can be seen that certain operator signs in Setl represent several entirely different operations depending on the type of operand to which they are applied. Such operators are referred to as polymorphic operators; an example is + (dyadic) which can mean integer addition, real addition, string or tuple concatenation or set/map union. Since the type of a variable changes dynamically, it is not possible for the compiler to determine it and select the appropriate operation to be performed (or signal a type incompatibility error). The type checking must be done at runtime. This fact has already been mentioned, since it is of importance in the design of the system, being a major factor in the decision to use an interpretive scheme.

With polymorphic operators it is not sufficient to determine whether an operand is of a particular type or not,the actual type must be found, in order to select the required operation. In contexts where a simple check of a particular type is sufficient, the indirect pointer in the first word of the block can be tested, since there are different entry points for each type. A series of tests would also be sufficient to establish the type of an object, but this would involve the coding of such a series of tests at the head of each operator routine, with each routine having to deal with all possible cases. This would, of course, be straightforward, but the stereotyped nature of such code suggests that a more systematic approach could be used. The scheme which has been devised and implemented for this purpose provides a reasonably efficient, flexible method of type determination, as well as improving the structure of the runtime routines.

Two points should be noticed. There is an element of hidden polymorphism from the implementor's point of view, since sets and maps and also tuples and pairs have different internal representations and are separate types to the runtime system, although they are not distinct in Setl. Secondly, it is the case that once the type of one operand of a dyadic operator has been found it severely limits the valid types for the other operand, so that a type determination on the second operand can reasonably be left as a series of tests of the indirect pointer (at most, there will only be two of these).

3.3.2 Operator Implementation

An operator is represented by a block (OPBLK), the first word of which points to a routine to apply an operator, the remaining words being pointers to routines which perform the operation appropriate to that particular operator for each type of operand (some of these will be error actions). There are two routines, APPL1 and APPL2 which apply monadic and dyadic operators respectively. They are referred to collectively as APPLn. The code generated for an operator is a pointer to the first word of its OPBLK; there is one block for each operator provided in the Setl language, an example is the block for + shown in fig 3.5. The OPBLKs are set up by data definition statements in the Minimal code and hence are built in static, although in principle there is no reason why they should not be built dynamically and garbage collected.

It should be apparent that this organisation leads to a modular arrangement of small operator routines, each performing a limited function, and that the type determination has been separated from the operation. Nothing has been said yet about the way in which the actual routine to be performed is selected. If the APPLn routines only performed a series of tests on the type pointer of a value the scheme would be somewhat inefficient. Instead, use is made of the fact that, on entry to the routine, XR contains a pointer to the OPBLK. If types can be mapped onto (small) integers t_i, $1 <= t_i <= t_n$,

Figure 3.5 OPBLK for +

```
             +---------+
             |         |
             |     ----------> APPL2
             |         |
             +---------+
             |         |
      atom   |     ----------> + error routine
             |         |
             +---------+
             |         |
   integer   |     ----------> integer addition
             |         |
             +---------+
             |         |
       map   |     ----------> map/set union
             |         |
             +---------+
             |         |
     omega   |     ----------> + error routine
             |         |
             +---------+
             |         |
      pair   |     ----------> pair/tuple concatenation
             |         |
             +---------+
             |         |
      real   |     ----------> real addition
             |         |
             +---------+
             |         |
    string   |     ----------> string concatenation
             |         |
             +---------+
             |         |
       set   |     ----------> map/set union
             |         |
             +---------+
             |         |
     tuple   |     ----------> pair/tuple concatenation
             |         |
             +---------+
             |         |
truth-value  |     ----------> + error routine
             |         |
             +---------+
```

where t_n is the number of types, then a simple indexing operation on XR selects the correct routine. Such a mapping is provided by the entry point id, a constant which Minimal associates with each entry point. This can be loaded into a register and used as required. This does mean that all OPBLKs have to have entries for all types, even though some of these will correspond to errors; this overhead seems to be acceptable. (See[CM79])

Figure 3.6 is an example of this scheme.

Since there is a significant number of operators for which only one operand type is legitimate, an escape mechanism is provided to cut down on the space overhead. An alternative OPBLK format has a pointer to the routine APPX1 or APPX2 (collectively APPXn), then the type code (indirect pointer) for the legitimate type, followed by a pointer to the operator routine, and a pointer to an error routine. APPXn merely compares the first word of the operand block with the second word of the OPBLK, and transfers control to either of the two routines, as appropriate. This format is shown in Fig 3.7, for the operator VAL.

The code for APPLn and APPXn is presented in fig 3.8.

Figure 3.6 Modified ITC generated for x:=y*2 when y=5

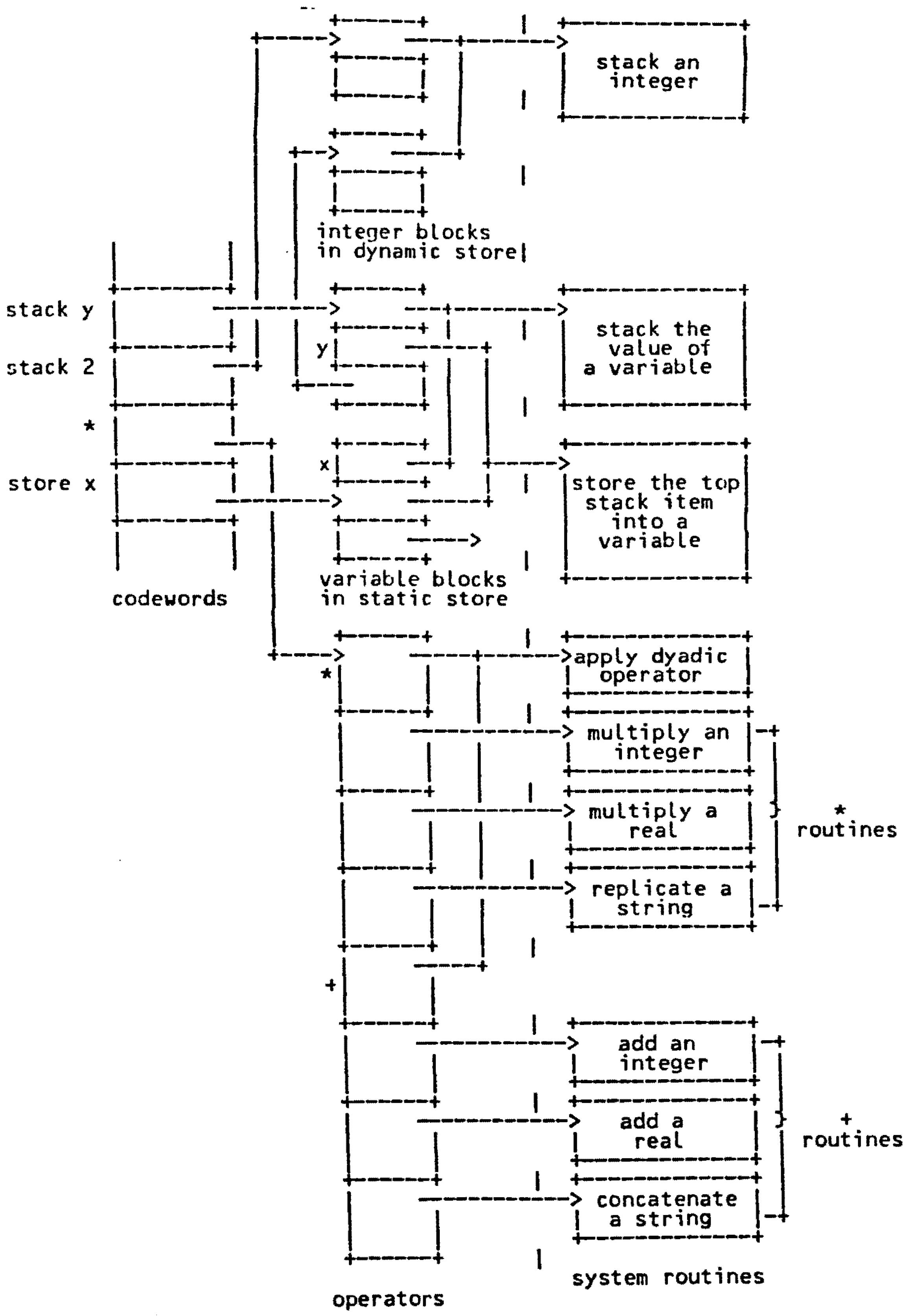


Figure 3.7 OPBLK for VAL

```
+----------+
|          |
|        ---------> APPX1
|          |
+----------+
|          |
|        ---------> load string routine
|          |
+----------+
|          |
|        ---------> VAL operation routine
|          |
+----------+
|          |
|        ---------> VAL error routine
|          |
+----------+
```

Figure 3.8 APPLn and APPXn Routines

```
*
*       APPL1 -- apply a monadic operator
*
*       The operator block has a vector of routine entry points
*       On exit merges to APPLY to select the correct one
*
APPL1   ENT
        MOV  (XS),XL          load operand
        BRN  APPLY
        EJC
*
*       APPL2 -- apply a dyadic operator
*
*       The operator block has a vector of routine entry points.
*       On exit, merges to APPLY to select the correct one,
*       according to the type of the LEFT operand (at 1(XS))
*
APPL2   ENT
        MOV  OFFS1(XS),XL     load left operand
        BRN  APPLY
        EJC
*
*       APPX1 -- apply a monadic operator
*
*       The operator block has the legitimate type code,
*       the operator routine address, and an error routine
*       address
*
*       On exit merges to APPX to do the checking.
*
APPX1   ENT
        MOV  (XS),XL          load operand
        BRN  APPLX
        EJC
*
*       APPX2 -- apply a dyadic operator
*
*       As APPX1, but checked on LEFT operand
*
APPX2   ENT
        MOV  OFFS1(XS),XL     load left operand
        BRN  APPLX
        EJC
```

Figure 3.8 APPLn and APPXn Routines (continued)

```
*
*       APPLX -- apply an operator
*
*       In the case where an operator is applicable to only
*       one type of operand, APPLX is entered to check the
*       type and enter the evaluating routine if ok.
*
*       (XL) operand of monadic operator or
*            left operand of dyadic operator
*
APPLX   RTN
        ICA  XR                  point to ok type word
        BEQ  (XL),(XR),APP10     is it what we have
        ICA  XR                  no bump pointer
APP10   ICA  XR                  point to routine
        MOV  XR,XL               copy entry point
        BRN  APPEX               merge to enter
        EJC
*
*       APPLY -- apply an operator
*
*       APPLY selects an appropriate operator routine, by
*       chhoosing one of the entries from its jump vector,
*       indexing by the EPI obtained from the operand in XL
*
*       (XL) operand of monadic operator or
*            left operand of dyadic operator
*
APPLY   RTN
        MOV  (XL),XL             get entry point
        LEI  XL                  load EPI
        WTB  XL                  convert to BAU offset
        ADD  XR,XL               point to appropriate routine
*
*       Continue by falling into APPEX
        EJC
*
*       APPEX -- enter operator routine
*       This is the common exit for APPLX and APPLY
*
*       (XL) operator routine entry point
*
APPEX   RTN
        MOV  (XS)+,XR            pop right operand
        MOV  (XL),XL
        BRI  XL
        EJC
```

3.4 SOME FEATURES OF THE RUNTIME ROUTINES

3.4.1 Iterators

The Setl-s runtime system includes specialised routines to implement loops controlled by iterators of the form: (For name in expression). The code generated for such loops may be represented in a pseudo-machine language as follows:

```
        {code to evaluate expression}
          PRPIT
          ->L0
    L0: JNEXT
          ->L1
          STORE name
        {code for loop body}
          ->L0
    L1:
```

The labels indicate the heads of codeblocks and the notation ->L indicates a pointer to the corresponding block; note that the blocks have to be chained together explicitly. The pseudo-opcodes PRPIT and JNEXT are indirect pointers to system routines which each perform a fairly complicated function. The routine for PRPIT prepares a temporary to control the loop. The form of this temporary depends on the type of the controlling expression, which is the top stack item on entry. For pairs, tuples and strings, the temporary is a small integer which is initially the index of the first item (1 for tuples and pairs, 0 for strings as a result of the internal representation used by the system). For set-like objects the temporary is an offset into the block, which gives the first word which can potentially hold an element.

The routine for JNEXT performs a composite function: it checks whether all elements of the expression have been exhausted and, if so, cleans off the top two stack items and executes the next codeword, which causes a jump out of the loop; otherwise, it extracts the next element, updates the temporary and increments the code pointer so that the assignment to the loop variable and the body of the loop will be executed. In the case of tuples, pairs and strings the action is simply performed, since the blocks for such values contain a field giving the number of elements and this can be compared with the temporary to see whether all elements have been used. If they have not, a procedure to access the appropriate element is called and the value which it returns is stacked. When the controlling expression is a set or map the situation is more complicated. Initially, the temporary is an offset into the block, and a procedure is called to return the value held at that offset. If this is a pair chain, the first item is returned and the iterator is updated to point to the next, as explained in 3.2.2; otherwise it is incremented to point to the next hash slot. If the offset exceeds the length of the block it is reset to the initial offset and a flag is set, which can be used by the calling routine to determine whether the set is exhausted. Since there will usually be 'empty' slots in a set block filled with OM values, the JNEXT routine must test for these and skip over them.

One special case of the iterator involves the arithmetic former: e.g (For i in [first,next...last]). It is obviously inappropriate to build a tuple and then extract its elements, so this case is detected by the code generator which produces code for a conventional arithmetic loop resembling a Fortran DO-loop. Loop temporaries for the current value, the limit and the step are held on the stack; special routines are used to load their values onto the top when required. The only point of interest in this is that the form of loop permitted in Setl is so general that it is not possible to determine until runtime whether the loop variable is increasing or decreasing. This leads to the bizarre necessity of including in the code a test to determine whether the step value is negative and, if it is, to swap the current and limit values before comparing them at the end of the loop, since the condition to be satisfied on termination is reversed by a negative step.

Set and tuple formers are implemented as special cases of loops. That is, an expression of the form {expression: iterator} is expanded as if it were:

```
Expr
    temp0 := {} ;
    (For iterator)
        temp0 with:= expression ;
     End ;
     Yield temp0 ;
```

Similarly, an arithmetic former is expanded into a loop in which the current value of the loop variable is added to an initially empty set or tuple.

These iterators offer an example of the flexibility of the interpretive code format, and the compact representation of loops which is permitted can be contrasted with the machine code which would be required.

3.4.2 Procedure Call And Return

It was noted in 3.1.2 that a procedure call is represented in the code as a pointer to a PCBLK the first word of which is a pointer to a routine to call procedures. On entry to this routine, the values of the arguments being passed to the procedure are on the stack and the next codeword holds an argument count (thus slightly upsetting the purity of the code format). Because of the way in which space is allocated in the static region during compilation, the VRBLKs for the arguments and local variables of the procedure being called will be contiguous; during compilation of the procedure body, pointers are set in the PCBLK to point to the start and end of this contiguous region of static. The procedure call routine performs the unconventional action of stacking the pre-entry values of the arguments and locals of the _called_ procedure, and then initialising the arguments to the values passed on the stack. On first entry to a procedure a pointer is set in the PCBLK to point to the stacked values of the locals; on recursive entries, this pointer is used to obtain initial values for any locals which had been set by Const or Init declarations. This mechanism means therefore that such declarations can be dealt with by the compiler and do not produce any code to be executed.

Certain link information has to be placed on the stack. The old stack pointer is required to identify the start of the information stacked by the call. A return link is also needed: this is stacked as a pointer to the calling codeblock and an offset to the codeword to be executed next, because no pointers into the middle of blocks may be stacked. A pointer to the PCBLK is also stacked. Setl does not permit reference to non-local variables except globals, so no static chain or display is needed. Finally, the code pointer is reset from the PCBLK to point to the procedure's entry point and the procedure is entered.

On return, the link information is recovered from the stack, the locals are restored to their pre-entry values, the stack is cleaned up and execution of the calling code is resumed.

CHAPTER 4

THE SETL-S COMPILER

4.1 THE PARSER

4.1.1 Parsing Algorithm

Syntactic analysis in Setl-s is carried out by a parser for an SLR(1) grammar, using essentially the algorithm given by DeRemer in [DER71]. Since the formulation of this algorithm in Setl-s is somewhat different from the best-known implementations of LR(k) parsers (see, for example [AJ74], [AU77] or [JOH78]), it will be useful to review the basic ideas behind this class of parser. The notation and terminology used follow those of [DER71] and [AJ74].

An LR(0) parser for a context-free grammar with start symbol S is constructed by computing its _configuration sets_ - each member of a configuration set is known, not surprisingly, as a _configuration_, and consists of a production with a special marker, indicated by a dot, in its right part. The sets are computed as follows: the grammar is augmented by a production S' -> |- S -|, where |- and -| are special terminal start- and end-markers, and S' is a new start symbol. The _initial configuration set_ is s0 = {S' -> .|- S -|}. Each configuration set which is not empty has one or more _successor_ configuration sets, computed from a basis set, obtained by moving the dot to the right over one symbol; in general, a configuration set has an s-successor for each symbol s that is preceded by a dot in any configuration of the set. If the dot in a configuration in the basis set precedes a non-terminal, N say, then a closure set is added to the basis set. The closure set consists of configurations where N is the subject of the production, and the dot precedes the first symbol in the right part. This closure operation is repeated until no new configurations are required. If the dot appears at the right hand end of a configuration, then the successor is the empty configuration set, called the #p successor, where p is the number of the corresponding production.

Figure 4.1 shows the configuration sets for the grammar:

S -> |- E -|	0
E -> E + T	1
E -> T	2
T -> P / T	3
T -> P	4
P -> i	5
P -> (E)	6

(This example originally appeared in [DER71])

The parser for an LR(0) grammar can be represented by a deterministic pushdown automaton, consisting of a finite control known as the characteristic finite state machine (CFSM) and a stack - the states of the CFSM correspond to configuration sets, which, in a sense, represent 'states of the parse', and the transitions correspond

Figure 4.1 Configuration Sets for Example Grammar

State	Configuration set	Successor
0	S -> .|- E -|	|- ==> 1
1	S -> |- .E -|	E ==> 2
	E -> .E + T	
	E -> .T	T ==> 6
	T -> .P / T	P ==> 7
	T -> .P	
	P -> .i	i ==>10
	P -> .(E)	(==> 11
2	S -> |- E .-|	-| ==> 3
	E -> E .+ T	+ ==> 4
3	S -> |- E -|.	#0 ==> {}
4	E -> E + .T	T ==> 5
	T -> .P / T	P ==> 7
	T -> .P	
	P -> .i	i ==> 10
	P -> .(E)	(==> 11
5	E -> E + T.	#1 ==> {}
6	E -> T.	#2 ==> {}
7	T -> P ./ T	/ ==> 8
	T -> P.	#4 ==> {}
8	T -> P / .T	T ==> 9
	T -> .P / T	P ==> 7
	T -> .P	
	P -> .i	i ==> 10
	P -> .(E)	(==> 11
9	T -> P / T.	#3 ==> {}
10	P -> i.	#5 ==> {}
11	P -> (.E)	E ==> 12
	E -> .E + T	
	E -> .T	T ==> 6
	T -> .P / T	P ==> 7
	T -> .P	
	P -> .i	i ==> 10
	P -> .(E)	(==> 11
12	P -> (E .)	) ==> 13
	E -> E .+ T	+ ==> 4
13	P -> (E).	#6 ==> {}

to the successor relations. Figure 4.2 shows the CFSM for this example. There are three kinds of state in the CFSM :

1. Shift State: all transitions are under symbols in the vocabulary.

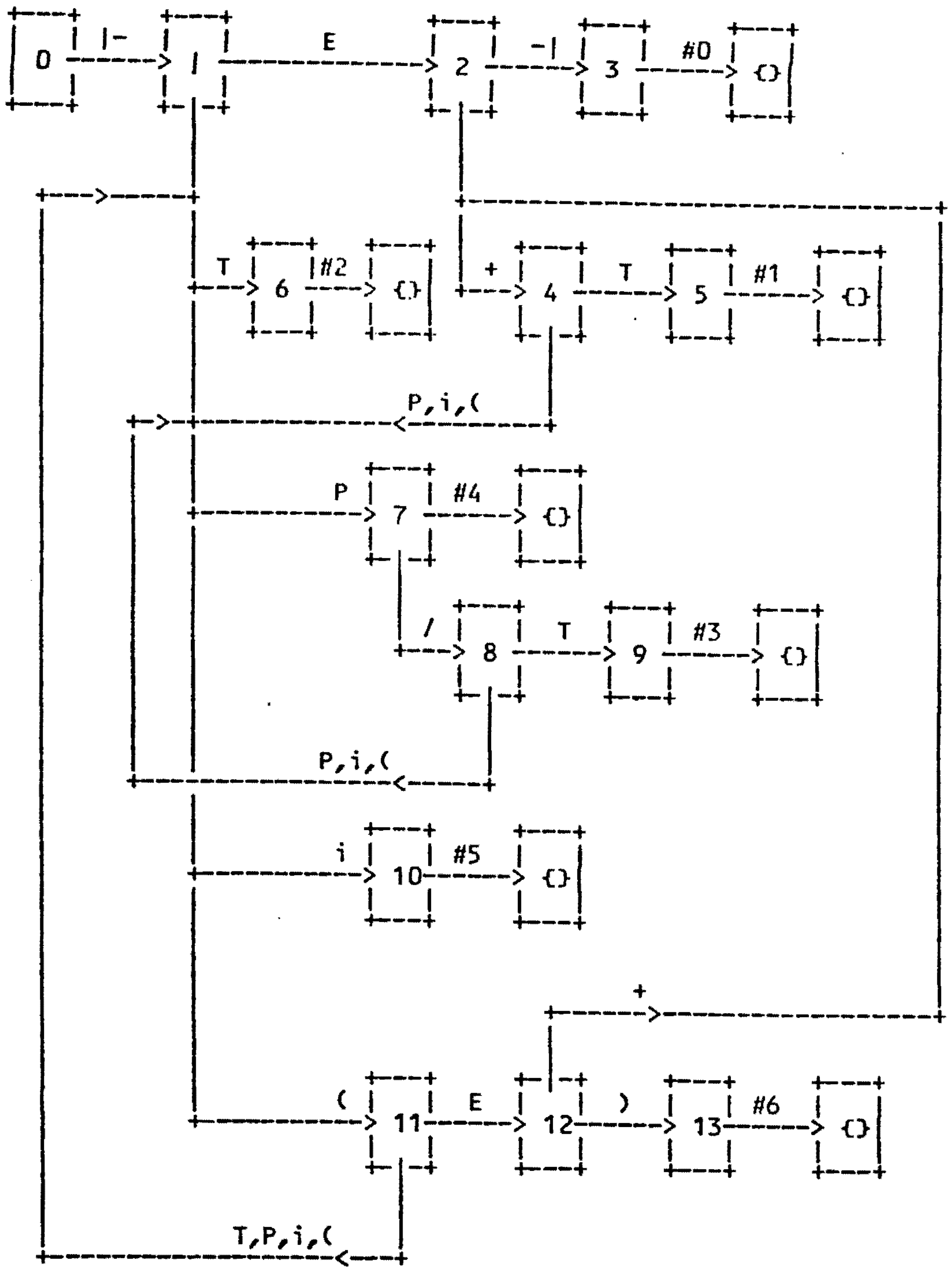
2. Reduce state: there is exactly one #p transition.

3. Inadequate state: any state which is neither a reduce state nor a shift state. It will have either one #p transition and one or more transitions under vocabulary symbols (shift-reduce conflict) or more than one #p transition (reduce-reduce conflict).

A grammar is LR(0) iff its CFSM has no inadequate states, in which case the following parsing algorithm, in which the stack is used to remember left contexts, so the parser can 'restart' after a reduction, can be used.

```
   Set current state = initial state
 $(
   If current state is a shift state then read the next symbol from
   the input, and push it onto the stack.

   Select the successor state according to the CFSM transitions.

   Push the successor state, and set current state to successor
   state.

   Otherwise

   If current state is a reduce state, pop the appropriate number of
   items from the stack (2 * number of symbols in right part of the
   production being applied).

   Prefix the subject symbol of the production to the input.

   If subject of the production is the sentence symbol, then accept.
```

Figure 4.2 CFSM for the example grammar.



Set current state to state on top of the stack.

$) Repeat steps between $(and $).

Note that in fact it is not strictly necessary to push the symbol in a shift state, but this provides a convenient way of remembering semantic information associated with it.

In general, the grammar for a practical language will not be LR(0), and so lookahead is used to resolve parsing conflicts in inadequate states. There are several algorithms for computing appropriate lookahead sets - in Setl-s the most straightforward of these is used to compute the simple 1-lookahead sets associated with each transition from an inadequate state. For a transition under a vocabulary symbol s, the set is {s}, for a transition under #p, where p is a production A -> w, the set used is:

$$\{s \text{ in } V_T \mid S \Longrightarrow^* \alpha A s \beta, \text{ for some strings } \alpha, \beta\}$$

i.e. the set of all terminal symbols which may follow A in any sentential form. If all the lookahead sets for transitions from any inadequate state are disjoint, the grammar is SLR(1). (In general, the simple 1-lookahead set contains symbols which could not possibly be read from this particular state, hence the SLR(1) algorithm cannot produce parsers for some languages which are nevertheless LR(1) or LALR(1), see [AJ74].) The CFSM is modified by replacing each inadequate state N by a lookahead state N', such that, for each transition to a state M from N under s with associated lookahead set L, there is a transition from N' under L to a state M' which has exactly one transition, that under s to M. Here 'transition under s' includes the #p transitions to the empty state. The modification to replace state 7 in the example CFSM is shown in figure 4.3.

The parsing algorithm has to be augmented with:

> If current state is a lookahead state then investigate (but do not read) the next symbol from the input, and change state according to the transitions of the CFSM.

4.1.2 Setl-s Parser Representation

Any table-driven parser resembles an interpreter, the tables or their equivalent being used to direct the execution of the parse. This suggests that the interpretive scheme used in the Setl-s runtime system could profitably be adapted for use in a parser. This has been done successfully in Setl-s where the CFSM is represented by a directed graph. Nodes in this graph are represented by blocks: SSBLKs for shift states, RSBLKs for reduce states and LSBLKs for lookahead states. The block formats are shown in figure 4.4. The first word of each block points to the entry point to one of the routines which perform the parsing actions shift, reduce or lookahead, thereby embedding these actions in the data structure. Parsing is accomplished by interpreting the data structure in a manner resembling that in which the ITC is interpreted at runtime.

The successor to a shift or lookahead state is selected by comparing the next symbol from the input stream with each of the symbol entries in the state block. When a match is found, the pointer in the following word is loaded and stacked, and an indirect branch is

Figure 4.3 Replacement of inadequate state

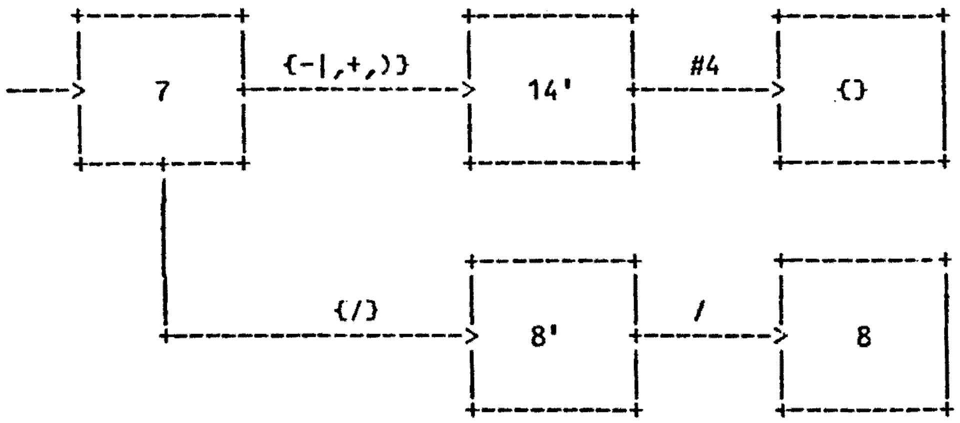


Figure 4.4 Parser Block Formats

a) SSBLK - shift state

```
+---------+
| SSACT   |      Pointer to shift routine
+---------+
| SSLEN   |      Block length
+---------+
/         /
/ SSACT   /      Symbols and successors
/         /
+---------+
```

b) RSBLK - reduce state

```
+---------+
| RSACT   |      Pointer to reduce routine
+---------+
| RSSYM   |      Subject symbol of production
+---------+
| RSLEN   |      No of symbols to pop
+---------+
| RSNUM   |      Production number
+---------+
```

c) LSBLK - lookahead state

```
+---------+
| LSACT   |      Pointer to lookahead routine
+---------+
| LSLEN   |      Block length
+---------+
/         /
/ LSSSS   /      Lookahead sets and successors
/         /
+---------+
```

d) LABLK - lookahead set with multiple entries

```
+---------+
| LATYP   |      Dummy type pointer
+---------+
| LALEN   |      Block length
+---------+
/         /
/ LAMEM   /      Members
/         /
+---------+
```

taken through the first word of this successor state block. Thus the routine to perform the next parsing action is entered with register XR pointing to the state block for the current state. The entries in an SSBLK are simply pairs of grammar symbols and pointers to successors. The entries to in LSBLKs are slightly different, since the lookahead sets corresponding to #p transitions will have multiple entries. These are therefore gathered into a block; the entry in the LSBLK points to this, and the routine to perform the lookahead action searches the block appropriately. The lookahead itself is handled by having a global re-scan switch, which, if set, causes the scanner to return the same result as on the previous call. A lookahead action sets this switch, a shift action does not, and thus absorbs the token as required.

If no match for a token is found in a shift or lookahead state, an error is reported.

An RSBLK contains the information required to perform the reduction: the number of items to pop off the stack, the number of the production being applied (this is used to direct semantic processing, see 4.3.1) and the subject symbol of the production which will be prefixed to the input stream. Another global switch is used to indicate to the scanner that a non-terminal is available.

Figure 4.5 gives BCPL routines to perform the parsing actions - this form of presentation permits the elision of irrelevant details in the Minimal code, for example the saving of registers. In fact, the organisation of the code in the actual system is organised slightly differently to accomodate error recovery, but it is not much longer.

Figure 4.5 Parsing Action Routines

```
LET shift() BE
$(
    LET symbol = nextsymbol()
    push(symbol)
    xr := ssmatch(xr, symbol)
    IF xr = error DO error_recovery()
    push(xr)
    BRI !xr
$)

AND lookahead() BE
$(
    LET symbol = investigatesymbol()
    xr := lamatch(xr, symbol)
    IF xr = error DO error_recovery()
    BRI !xr
$)

AND reduce() BE
$(
    LET symbol = r.symbol!xr
    AND len, prod_number = r.length!xr, r.pnum!xr
    semantic_action(prodno)
    xs -:= len
    prefix_to_input(symbol)
    xr := !xs
    BRI !xr
$)
```

Notes:
    BRI is a fictional command performing an indirect branch.
    On entry to each function, xr points to the state block.
    xs is a stack pointer.
    The functions lamatch and ssmatch search an LSBLK and SSBLK
    respectively, returning the successor pointer, or error.

4.1.3 Parser Generation

The parser is produced by a system known as Slrgen, which consists of two programs: Slrgen itself and Slropt. The first of these is a straightforward implementation of the constructor algoritnm given in 4.1.1. It is written in BCPL and is intended to be portable; it has run successfully on both the DEC-10 and Amdahl 470/V7 machines at Leeds University. No particular effort was made to produce great efficiency in Slrgen, but on the Amdahl it can process the Setl-s grammar (containing about 240 productions) in about 30 seconds of cpu time, which is acceptable, especially as it is to be hoped that the parser will not have to be re-generated often.

Slrgen accepts as input a BNF description of the syntax; no extensions such as the use of regular expressions in productions are accepted, and a set of lexical conventions must be observed. Appendix 2 is an example of the input required, being the grammar for a very simple programming language to be described in Chapter 6. Slrgen computes the parser states from the grammar and diagnoses grammars which are not SLR(1). The output is in a readable form and can be edited to resolve ambiguities in the grammar or to make it possible to parse languages described by non-SLR(1) grammars. In fact, the Setl-s grammar is not SLR(1), because of the over-use of the symbol IN both as an operator and as a connective in the syntax of iterators. Two inadequate states for which the parsing conflicts cannot be resolved by one symbol lookahead arise from this, but it is easy to resolve the conflict by removing IN from lookahead sets. This makes the construction {x IN S} illegal, but as its meaning is unclear this does not seem unreasonable.

The output from the Slrgen program forms the input to Slropt. This program performs three major optimisations: it merges states which are identical, it merges identical lookahead sets and it removes reductions by 'single productions' [AJ74]. These are productions of the form A -> a where |a|=1. Reductions by such productions are merely wasteful and can slow down the parser considerably. If the productions have no semantic actions associated with them, as is generally the case when the right part is a single non-terminal, they can be removed from the parser. The states corresponding to these reductions need never be built, so there is an additional saving of space. Section 5.3 contains some statistics on the efficacy of these optimisations in Setl-s.

The output from Slropt is a linear representation of the CFSM graph known as PCX which consists of a series of symbols, separated by newlines. The symbols are grouped in states as follows:

```
 state number
 state type
{length}    some types only
 members
```

The state number is used to identify the state. The state type is a 2 character code beginning with @; the possible types are:

@S	shift state *
@R	reduce state
@L	lookahead state *
@A	lookahead set *
@D	duplicate state
@B	duplicate lookahead set

State types marked * have a length field, which is the total length of the state group. The entries for shift and lookahead states come in pairs consisting of a symbol followed by its successor. The symbol is either a terminal, represented by itself, a non-terminal represented by a code number, or, in the case of a lookahead state, a lookahead set. The successors are simply state numbers.

A reduce set has three members: the number of symbols to be popped on reduction, the subject symbol of the production, and the production number.

A lookahead set is just a list of symbols.

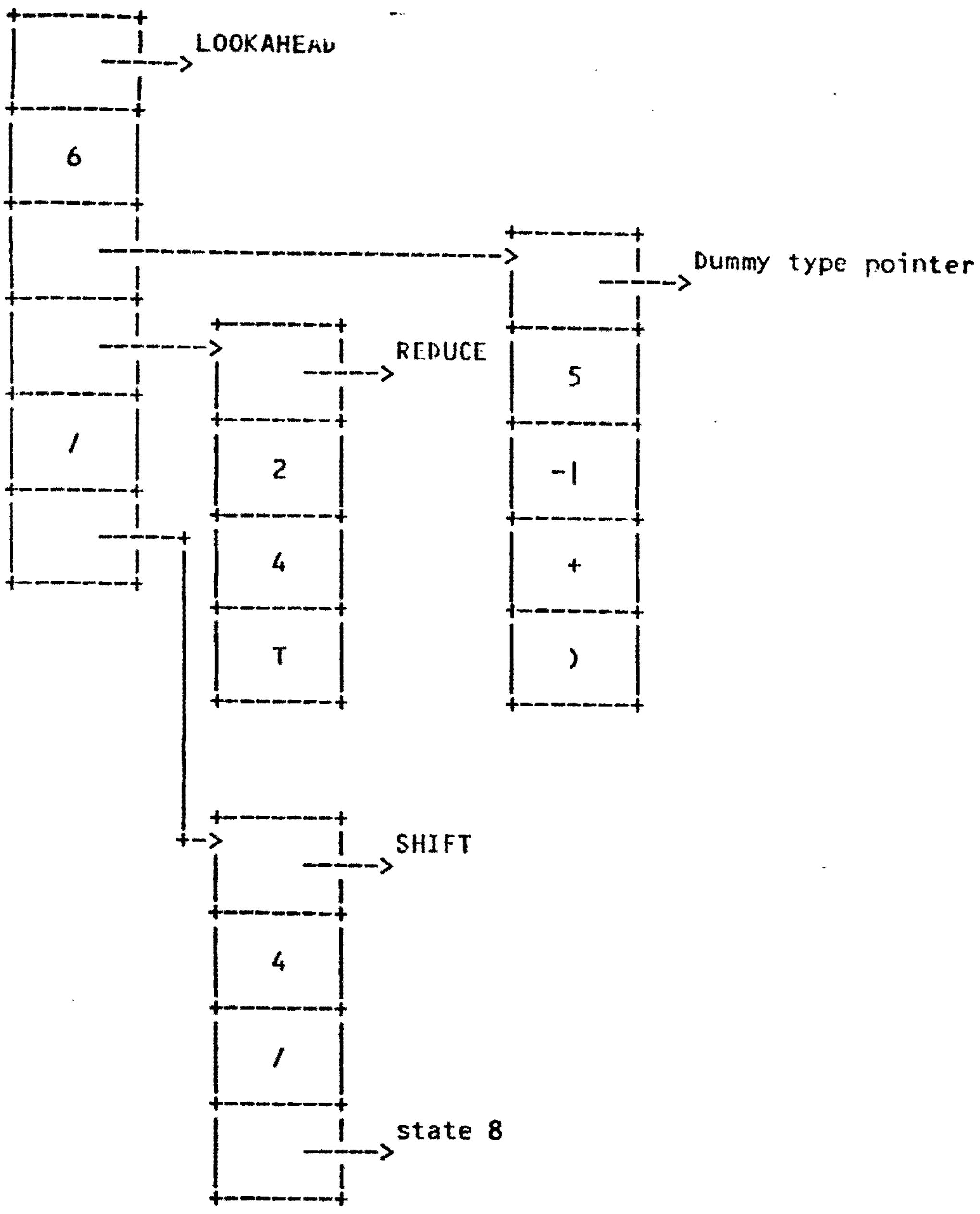
A duplicate state has only one member, the number of the original state of which it is the duplicate. Similarly, a duplicate lookahead set has a pair of members, the state containing the original, and the offset within it to the required set.

To illustrate this novel compiler structure, Figure 4.6 shows the PCX and data structure corresponding to the CFSM fragment of Figure 4.3.

4.1.4 Initialisation

The CFSM graph is built in dynamic memory during the initialisation phase of Setl-s. The PCX produced by Slrgen is read in, and used to build the blocks. Initially, the blocks are built with successor fields holding state numbers as read from the PCX; as each state block is built, an entry corresponding to its state number

Figure 4.6 Data structure and PCX fragments



```
    7                                      16
   @L                                      @R
    6                                       2
   @A                                       4
    5                                     101     (code for T)
   -|
    +                                      17
    )                                      @S
   16     (i.e. 14')                        4
    /                                       /
   17     (i.e. 8')                         8
```

is made in a state table - this entry is a pointer to the block itself. When all blocks have been built, a pass is made through them, and the successor entries are replaced by the appropriate pointers. The result of this process is a data-structure which is garbage collectable. After parsing has been completed, the pointer to the root of the parser is cleared, and the garbage collector is called to reclaim the space formerly occupied by the CFSM graph, thereby providing a form of overlay for the system's workspace. If the operating system interface provides the capability, this space can be returned to the system, leaving a small executing program. For Setl-s the space recovered is nearly 10,000 words; the amount of code left, including error recovery, parser setup and parsing action routines is less than 500 words.

This setting up of the parser is, however, a time-consuming process, typically taking 12 to 13 seconds on the DEC-10. Incurring this overhead on every run would be quite unacceptable in most environments in which Setl-s is envisaged as being used. It should properly be regarded as an extra phase in the building of the system following translation and loading, which should only be repeated when a new version of the grammar or interpreter is introduced. In the context of Minimal it is possible to de-couple the building of the CFSM graph from compilation of users' programs. The operating system interface specification includes a procedure SYSXI, which, when called, releases i-o associations and halts execution, permitting the user to save a core image. (It was originally called for to provide the exit function for macro-Spitbol - see [MHD76].) So, after the CFSM graph has been set up, the garbage collector is called to remove garbage created by the scanner during setup, and then SYSXI is called. The core image saved after this call is what the Setl programmer sees as Setl-s. All initialisation is complete, and parsing commences on entry.

4.1.5 Syntactic Error Handling.

It has become a truism of compiler construction that the diagnosis of and recovery from syntax errors is of major importance. In the current Setl-s system, however, the treatment of errors has not received a great deal of attention, since it is not a primary aim of the project to build a production compiler. The mechanism which is used is similar to that used in Yacc [JOH78]: the grammar is augmented by productions for 'major' non-terminals which include the special terminal symbol $error. When an error is detected, the current input symbol is replaced by $error and the stack is popped until a state is found which has a parsing action on $error. This state is then entered, and parsing continues until a reduction is made by one of the error productions, at which point an error message is given and recovery is attempted by discarding input symbols until one is found on which a legal parsing action is possible - further error messages are suppressed until a specified number of successful shifts has been made, in an attempt to prevent an avalanche of messages.

Erroneous statements are flagged in the listing, with a pointer below the symbol at which the error was detected, hence giving the user as much information as possible from the parser's early error-detecting capability. The message produced is generally rather vague ('syntax error in expression', for example).

It appears that very much better error handling is possible in LR parsers [GHJ79], and it should be a fairly straightforward job to make use of more powerful techniques in a production Setl-s compiler. This development is not entirely trivial, however. The main idea is to attempt several repairs to the input and then to allow the parser to make a forward move, and evaluate the success of the repair. This necessitates buffering input tokens and copying the stack, as well as requiring the parser to operate in an error mode in which no irrevocable actions are performed. This work does not introduce any new ideas into the system and so, although it is important to a production compiler, it has not presently been pursued.

4.2 LEXICAL ANALYSIS

4.2.1 The Scanner

Whenever the parser requires a symbol from the input, it calls the routine SCANE, which returns a pair of values: a token type code in register XL and a token 'value' in XR. Most of the tokens are conventional, corresponding to basic symbols such as + and 'micro-syntactic' constructs such as integers. Each of these has a unique type code, and where appropriate, the value returned in XR is the semantic information associated with the particular token just scanned. For example, if the element scanned were 1356 then XL will contain the type code for number, and XR will contain a pointer to an ICBLK built to hold its value. In the case of operators, the token type designates the priority class of the operator and the value is a pointer to its OPBLK.

During initialisation, SCANE is also used to read the PCX. Since terminal symbols have their normal character representation, SCANE naturally returns their internal token form to the block-building routines, which use these as the symbol entries in SSBLKs and LSBLKs so they can be directly compared with the token types returned from the Setl source during parsing. Because non-terminals are represented by numbers, a condition is imposed that the token types for terminals must be distinguishable from these numbers (state numbers can be identified from their position in the PCX). The highest non-terminal value is known (it is passed as a parameter to Slrgen) so this condition is satisfied by ensuring that all token types exceed this value. In the grammar, classes such as name, number or class 3 operator appear as terminal symbols: $name, $number and $op3. When the grammar is read during initialisation, SCANE recognises names beginning with $ as classes and returns the type code for the class. This means that during initialisation the $ symbol must be treated differently, as must some other symbols, but the overhead of this is negligible and is rarely incurred, since $ otherwise only introduces a comment and the other symbols concerned are illegal in Setl.

During parsing, it is the type code which is used to select successor states in the CFSM, while the token value is put on the stack for use by semantic routines.

4.2.2 The Symbol Table And Reserved Words.

The symbol table used by Setl-s is organised as a hash table, with collisions being handled by chaining entries together. Entries for identifiers consist of the VRBLKs built for them, chained together via their VRNXT fields (see fig 3.4h). Reserved words are entered into the table as special WDBLKs. The format of these is identical to that of VRBLKs, as regards the position of length field and relocatable fields, but the VRVAL and VRSTO fields are used to hold the type code and value for SCANE - the VRGET field is set to a special type, so that it can be recognised as a reserved word and treated accordingly. Identifiers declared as the names of functions also have special table entries: the type word of the block is set to indicate that the variable is, in fact, a function, and since the VRSTO field is redundant it is used to chain together all the entries for functions, to make them easily accessible.

In Setl programs, upper- and lower-case letters are treated as being identical, so all alphabetic characters are converted to lower-case before a table lookup is attempted. On the source listing which can be produced by the system all identifiers are printed in lower-case and all reserved words and names of system functions in upper. As well as producing a more readable listing, this has a useful side-effect, in that any reserved word inadvertantly used by the programmer as an identifier will be upper-cased on the listing. This will permit an easier identification of the source of the error than the resulting syntax error message might. Such accidental use of reserved words is a source of particularly obscure syntax errors, and this listing convention provides an economical aid to their pin-pointing.

The scope rules for identifiers in Setl are especially simple. There is no block structure, and a variable is purely local to the procedure in which it is declared (explicitly or implicitly). The only exception to this is that variables explicitly declared at the head of the main program block are global, only those used without declaration are local to the block (sic). These rules mean that a variable can be uniquely identified by its name and the procedure in which it is declared. This is done by assigning numbers to procedures as they are declared, and logically OR-ing this number into the left hand end of the VRNML field of the VRBLK. Because local variables need not be declared explicitly, the table lookup routine must first look for an entry for a global symbol, by masking out the procedure number, and, failing this, a local symbol. If this search fails a VRBLK for the local is built and entered into the symbol table.

Global declarations cause a minor problem. If, as seems natural, the procedure number were initialised to zero and variables were entered into the symbol table as globals until the reduction for declarations had been performed, all would be well unless there were no globals. In this case, lookahead would be required before the reduction could be made, and this might cause the creation of a global symbol table entry for a name which was genuinely local to the main block. To avoid this, globals are entered into the symbol table as locals of the main program block, and the symbol table entries are amended when a reduction associated with the declaration is made.

4.3 SEMANTIC ACTIONS AND CODE GENERATION

4.3.1 Building The Parse Tree

Whenever the parser performs a reduction, it calls a routine to perform any semantic action which may be appropriate. In nearly all cases, the appropriate action is to build a node of the parse tree. There is very little attribute processing to be done, because of the dynamic typing; the only semantic actions other than building tree nodes relate to dealing with symbol table entries for procedures, fixing the scope of globals and processing the declarations of initialised variables and constants.

The parse tree building is directed by the grammar. That is, whenever a production is applied, the descendants of the node being built will be related only to the grammar symbols in the right part of the production. Single terminals on the right give rise to leaves in the tree; more complicated right parts have their essential structure abstracted to produce interior nodes. The values corresponding to the grammar symbols of the right part will have been placed on the stack during earlier shift moves. The position of each on the stack can be found from the position of the occurrence of the corresponding symbol in the production. After construction of the node, a global variable is set to point to it; when SCANE is next called it will return the non-terminal which was the subject of the production in XL (see 4.2.1) and the pointer to the tree node as the value in XR. This means that sub-trees corresponding to non-terminals get placed on the stack, to become available to the tree-builder.

4.3.2 Generation Of ITC.

A code generator is called for each procedure, and for the main program block, to flatten the tree and produce appropriate codewords. The approach taken is a simple one: the parse tree is walked recursively, with stereotyped code sequences being produced for the various constructs in Setl. No optimisation of special cases is attempted.

The final size of each codeblock is not known until it is complete, but it is necessary for the partly-built block to be protected from the garbage-collector. The scheme used for building CDBLKs is inherited from Spitbol - a code construction block (CCBLK) is allocated, and the codewords are generated in it. The garbage collector knows that only certain fields are in use, and processes the CCBLK accordingly. When a codeblock is complete, it is cut off as a CDBLK from the CCBLK with the remaining words being re-set to form a new reduced CCBLK. The size of block allocated for code construction greatly exceeds the size of typical codeblocks, so it is only rarely that there is not sufficient room to generate codewords. However, when this does happen, a fresh CCBLK has to be allocated, and the codewords generated so far have to be copied into it.

This causes complications in the setting of pointers within the code, to handle jumps. The overall strategy adopted is simple. The compiler generates label numbers for jumps inside loops and for conditionals, and uses a label table to keep track of them, with forward references being chained together from the table entries. Since there can be no pointers into the middle of collectable blocks, the chain entry for a forward reference consists of a pointer to the base of the codeblock in which the reference occurs, and an offset to the codeword which will ultimately hold the pointer to the destination of the jump. A label is set at the head of a codeblock, but the ultimate address of the codeblock will not be known until it is finally cut off from the CCBLK, since the latter may run out. Hence, resolution of forward references must be deferred until the codeblock is complete. Similar complications arise in connection with forward references occurring within the current block, since the base of the descriptor for the forward reference cannot be set. Dealing with these involves, essentially, keeping a chain of 'pending' forward references, in which the base field is used to hold the label number. When the block is complete, these are turned into normal table entries, and then the references to the head of the block can finally be resolved. (Care must be taken to treat a block which jumps back to the head of itself correctly.)

Since compiler generated labels are used for constructs which are defined in a nested manner, they can be issued and de-allocated in a nested fashion, hence the size of the compiler's label table can be kept small without imposing undue limitations on programs.

CHAPTER 5

DISCUSSION OF SETL-S

5.1 ASSESSMENT OF THE SYSTEM

5.1.1 Performance

The listings reproduced in Appendix 1 show some typical performance figures for Setl-s; some explanation of their significance is required. The programs were run on the Leeds University DECsystem-10, which has a KI10 processor and 256K words of memory. Although the timing system on this machine is notoriously inaccurate, the times given provide an indication of the execution speed of Setl-s. (The runs of these programs have been repeated several times and the figures in the appendix are about average.) The actual terminal response time of the system depends on the load on the DEC-10, but it is consistently acceptable, comparing well with other language processors available on the machine.

The figures for 'store used' and 'store left' give the number of occupied words and free words respectively in the dynamic area; the initial size of the dynamic area can be set by the user, the default being 15K (K=1024) words. If during a run the garbage collector cannot reclaim more than a specified number of words the dynamic area is increased, if possible, by claiming more memory from the operating system; this was not necessary on these tests. Under these circumstances, the system consisted of a 36 page impure 'low segment' (a page on the DEC-10 is 512 words) and a potentially shareable 'high segment' of 20 pages. The fact that even for the first small program a garbage collection was required during compilation suggests that the default value for the size of dynamic memory may be too small.

The figures for 'statements executed' should be largely ignored, since they do not correspond to source statement executions in the expected way, although with the 'mcsec/statement' figures they do provide a comparative measure between the programs. (The problem of getting the statement counts to correspond has not been pursued, because it is not considered very important and because it overlaps with other work on performance measurement which lies outside the scope of the present project.)

Each of the programs in the appendix has some noteworthy features. The first program, which computes the prime numbers up to 1000, printing the last few, is an example of the conciseness obtainable in the Setl language and the way in which Setl constructs resemble those of orthodox mathematics. When the implementation is considered, it also illustrates some shortcomings of this approach, since the set primes has to be allowed to grow dynamically, being copied several times in the process (note the two storage regenerations) and is represented internally as a linear hash table, which is not the most appropriate data structure for the application (to put it mildly). The 'representation sub-language' of full Setl addresses itself to problems such as these but its efficacy is still not proven.

Another point should be mentioned in connection with the primes program. According to the Setl definition, the expression controlling a loop is evaluated once before the loop starts, so that any modification of its constituents inside the loop has no effect on the number of times it is executed. (Consider, for example:
(For x in t) t with:= x End) This is merely an extension of the value semantics of assignment. A naive approach to this problem, whereby the routine which sets up the iterator for a loop first copied the value controlling it was at one point implemented; since a quantified test conceals a loop, this copying operation was performed every time the Notexists test was executed. The result of this was that the primes program ran over 100 times slower. A more appropriate solution would be to perform a check at code generation time to determine whether the loop expression had one of the forms described in 3.2.4, and insert an explicit copy instruction into the code, but, in this particular case, even this would have had the same effect. Consequently, the attitude adopted at present is that if the expression controlling a loop is modified inside that loop the effect of the loop is 'undefined'.

The remaining programs in Appendix 1 have been adapted from the Setl test library developed at NYU to test the full Setl system. The first is an implementation of the O(nlogn) sorting algorithm known as Heapsort. It is notable that the total execution time is significantly greater than the time taken actually to sort the items.

The extra time is spent printing the two sequences, or rather building strings out of the tuples, preparatory to printing them. The algorithm used to build strings out of tuples is very crude (a concatenation of strings produced recursively from each component in turn) and the figures here suggest it should be improved. Finally, the third test is an implementation of a linear-time median finding algorithm. A brief examination of some figures obtained from the program showed its performance to be practically linear, so that the Setl-s system has not introduced any gross non-linearities into the behaviour.

Table 5.1 summarises these performance figures and compares them with the results obtained running the same programs on the full Setl system on the same machine. The first example was recoded in BCPL, using the same algorithm, and figures for this program are included in the table. These figures give a clear indication of the superiority in speed of execution, speed of compilation and system size of Setl-s over NYU Setl. Without delving deeply into the details of NYU Setl it is not possible to give authoritative reasons for the performance discrepancy, but several factors can be suggested.

1. NYU Setl is written in a relatively high level language and, moreover, one whose abstract machine model (based on arbitrary length bit strings) cannot be mapped comfortably onto real machine architecture.

2. The full Setl language is significantly more complex than the subset used in Setl-s. Consequently, compilation is more costly and the runtime system more complicated. It appears

Figure 5.1 Performance Comparisons

a) Speed

Program	Compile time / s			Execution time / s		
	NYU	Setl-s	BCPL	NYU	Setl-s	BCPL
primes	5.18	0.44	1.10	38.16	7.26	1.90
heapsort	10.84	1.62	-	18.2	6.52	-
median	11.70	1.82	-	4.32	1.14	-

b) System Size

System	Executing program / pages (lo + hi)	Source files / disk blocks
NYU	76 + 117	2980
Setl-s	36 + 20	780

that the overhead of this added complexity is incurred even by programs which do not use the extra features. (Setl-s implements roughly 75% of full Setl.)

3. There appears to be a real efficiency gain from the ITC interpretive scheme.

Supporters of NYU Setl might argue that this comparison is not entirely fair, since NYU Setl is intended to have a global optimiser and a feature whereby 'nubbins' of hard code are generated in-line for some constructs, thereby speeding up execution (at the expense of portability). Neither of these features is currently available outside NYU, however, and neither of them would prevent the system's being intolerably large, even for a medium-sized machine such as the DEC-10. On the contrary, both would increase the size still further.

5.1.2 Portability

Setl-s is only currently running on the Leeds DEC-10 on which it was originally implemented, so its portability can only be extrapolated from experience with macro Spitbol. This has proved to be a highly portable system, being implemented on the following machines: CDC 6000 series, CII Iris, DEC PDP11, DEC-10/20, DEC VAX, Honeywell 6000, ICL 1900 and 2900 series, Interdata 7/32, Modcomp IV, Odra, Xerox Alto. The production of a macro Spitbol implementation for a new machine is now quite routine, typically taking between 3 and 6 months. Minimal translators and operating system interfaces are available for the machines just listed, and in theory all that is necessary to move Setl-s is to translate the Minimal source into the target machine's assembly language, assemble the resulting program and load it with the interface routines. In practice, various problems might be expected to arise. Firstly, since the DEC-10 is a word-addressed machine all the Minimal features which depend on the difference between word- and byte-addresses have not been exercised and so bugs in the Setl-s source might show up in this area. Secondly, in practice, it appears that some Minimal translators have been written with the single aim of translating macro Spitbol, hence setting translator-defined symbols (see 2.3.1) may necessitate modifications to the translator itself. Finally, the necessity to save a core image of the compiler (see 4.1.4) may cause problems, since this is difficult to provide on some machines (e.g. IBM360/370).

The use of translator-defined symbols for all character codes and conditional assembly directives to cope with the possibility that certain characters such as { may not be available on a particular target machine should, it is hoped, eliminate the character code problems which are often encountered by portable software. Naturally, since Setl-s is interpretive, the different machine architectures will not present problems, except indirectly via the Minimal translator. Experience will be required to determine whether addressing space restrictions, as well as physical memory size, will have a severe effect.

One problem related to portability which has not received much attention in Setl-s concerns input and output, and the association of files. In Spitbol, the meaning to be ascribed to the arguments of INPUT and OUTPUT has been a source of constant dispute among implementors. The underlying model of file organisation varies so widely between operating systems that producing some meaningful, system-independent way of representing an external file inside a program seems almost impossible. This seems to be a problem best tackled at the language design level, and in Setl no clear definition has yet emerged. Similar questions arise from the interface between Setl-s and the host system's JCL (or equivalent) and at other points where operating system concepts interact with the system. Although a set of operating system interface procedures is defined, implementors' experiences with Spitbol have shown that these definitions are not always appropriate, and do not make sense on all systems. There is little to indicate that if Setl-s becomes widely available there will be fewer such problems for this system. Already several minor changes have been required to the DEC-10 operating system interface to accomodate Setl-s.

A major practical obstacle in moving any system to any machine lies in getting material on and off magnetic tape. This is a problem largely created by the manufaturers, and one with no immediate prospect of a solution.

5.1.3 Construction Of The Program.

The actual writing of the Minimal code of the present version of Setl-s took about nine months. Some operators remain unimplemented and both the compile-time and runtime error handling are incomplete; nevertheless, the system is capable of executing Setl programs and can be regarded as an essentially complete prototype. The source for this version is approximately 16,500 lines long. Of these, 2,600 consist of symbol definitions with substantial blocks of comment giving data structure formats and so on; the constant and working-storage sections comprise another 2,500 lines, which include the operator OPBLKs and the WDBLKs for reserved words. The parsing routines occupy a total of 150 lines, with the parser setup code occupying 275. The remainder is the code for the runtime system, including space allocation and input-output, the scanner and some system initialisation and cleanup code. Out of the total, some 4,500 lines contain code derived from Spitbol. The biggest pieces are the routines for converting between strings and numbers, the scanner and the garbage collector. The last two required significant modifications for inclusion in Setl-s. This extraction and adaptation of Spitbol code was mostly carried out by A. P. McCann. The rest of the system was written by myself.

That such a relatively complex piece of software could be written in an assembly language in what is considered to be a short space of time for such a project seems largely attributable to two factors. First is the quantity of code that has been taken from an existing system, and second is the structure of the system. Compilers lend themselves to a modular arrangement, with lexical analysis, syntactic analysis and code generation being treated separately. The use of a systematic parsing algorithm has contributed further to the ease with which the compiler could be constructed. The interpretive scheme also imposes a structure on the runtime routines which only interact in tightly controlled ways. The system is composed of small modules with well-defined interfaces, so each part can be considered in isolation. During development of the system, for example, the code generation scheme has twice been extensively revised, and these revisions have been carried out without upsetting any other part of the system. Features have been added to the runtime system without the need to worry that existing features will cease to work. This is in no way remarkable, but the amount of recent writing on programming methodology suggests that there is a prevailing misunderstanding of the way in which systems can be structured to assist development. ([NEE76] is a typical example of this sort of writing.) Top-down design and the use of high level languages cannot claim a monopoly in this area.

As an implementation language, Minimal has its drawbacks. The most important of these is the difficulty of doing input and output, which makes it hard to add certain kinds of diagnostic information. This can be offset, in a suitable environment, by the fact that the target code produced is closely related to the source, so that full advantage can be taken of interactive debugging systems, such as DEC's DDT. (It is also possible, albeit inadvisable, to patch small bugs without incurring the considerable cost of translating the entire system.)

In the early stages of the system's development, models of parts of it were written in BCPL; in particular, a complete parser was written to debug the Setl-s grammar. These early models were intended to be disposable, and the different criteria applying to them implied that BCPL was more suitable than Minimal.

The main pay-off from using a low-level implementation language is, of course, increased efficiency. This is particularly important in the construction of an interpreter, owing to the amount of time spent in the runtime interpretive routines. Also, the degree to which the flow of control can be specified permits the ITC interpreting cycle to be efficiently implemented, using the branch indirect instruction. Such a control construct is not supplied in high-level languages, although the effect can be simulated in various ways (most conveniently if procedure or label variable are permitted) with a loss of efficiency.

The preceding remarks would apply equally to any low-level implementation language; the following are specific to Minimal. Firstly, the simplicity of the underlying virtual machine makes writing code simple and means that the programmer is freed from a concern with bit-level tricks to improve efficiency; this in turn makes the code more comprehensible and more stable. On the other hand, Minimal is unduly restrictive about statement formats, use of literals and program form. Some of the effects of these restrictions were alleviated by the use of special-purpose editors for the typing in of the source. A more serious defect arises from the mechanism supplied for handling errors: the opcode ERR and ERB cause a transfer of control to the <u>error section</u> with an error code in WA. This mechanism is superficially attractive, allowing the error handling to be grouped in one place, and it also supports a way of providing a file of error messages which the translator produces from the Minimal source, but the only information supplied to the error section is the error number - the action to be taken has to be deduced from this alone. There is no indication of where the error occurred, so control cannot be returned directly there. The net effect resembles an uncontrolled jump, the inadvisability of which is now well-known. An elaborate and obscure (even to Minimal implementors) mechanism for saving and restoring subroutine linkage information had to be developed when Minimal was designed, to permit cleaning up after an error. The result of all this in Setl-s is that the handling of errors is crude and complicated, relying on the setting of global flags and a strict division of error numbers corresponding to different types of error. At this stage it is obvious that a wiser course would have been to ignore the ERR/ERB mechanism (except, perhaps, as a panic response to system errors) and use ordinary procedures to deal with error situations.

5.2 THE INTERPRETER

5.2.1 The Interpretive Scheme.

In Chapter 1, implementation strategies for high level languages were discussed in general terms; the arguments presented there can now be related to the Setl-s system, by considering the alternatives to the use of indirect threaded code.

Consider first the use of hard code. The main advantage of this is the increased execution speed obtainable by using instructions wired into the machine's hardware. To perform the set operations found in Setl programs, long complex sequences of the sort of instruction presently available would be required, leading to unacceptably bulky programs. This would be compounded by the need to generate type-checking code. Hard code is inherently machine-dependent and some form of bootstrap would be required to move a Setl compiler between machines. Also, in order to produce acceptable code making the best possible use of available hardware features, sophisticated optimisation techniques would be required, which would slow down compilation. In an environment where production programs were run many times, this would be offset by the ability to preserve compiled programs in an executable binary form. However, the Setl language seems best suited to applications such as algorithm development, where much time would be spent modifying and re-compiling programs.

To reduce the compiled code to an acceptable size various compromises between hard code and interpretation might be attempted. The most obvious of these involves providing a library of runtime routines to perform, for example, the set operations. The compiler would then generate hard code inline for simple operations, such as integer arithmetic, and subroutine calls to these system routines for the more complex ones. Thus, the code would be more compact, but an additional overhead would be introduced by the subroutine call and return. Bell's threaded code scheme [BEL73] uses a different form of control flow to achieve a similar effect, with a reduced overhead.

Although the name 'indirect threaded code' is derived from this scheme, the differences between the two are greater than their similarities. In threaded code, actual sequences of target code, linked by a threading mechanism are generated by the compiler. The equivalent of a codeword points directly to the entry point of a routine, but the routine does not receive an argument analogous to the pointer in XR, so separate routines have to be generated, for example, to load each variable onto the stack. The same routine can be used every time a particular variable is loaded, though, so there may be a considerable saving of space over the use of a compiler producing inline hard code. Nevertheless, threaded code is more bulky than ITC's single copies of system routines, despite the extra indirect pointer required by ITC, and again is not machine-independent.

For producing a portable Setl system, especially if it is desired to run on small machines, the arguments in favour of using some form of interpretive code are very great. It is not feasible to interpret programs written in Setl directly from the source, so a compiler of some sort is necessary to produce a lower level representation of the program to be interpreted.

Of the several other interpretive schemes described in 1.3, the use of a virtual machine with a conventional register architecture can be dismissed for this application, because Setl operations are not suitable for evaluation using registers and because such schemes use a code format in which certain fields of a word have special significance, which presents problems if a system is to be implemented on a variety of machines. Translation of Setl operations to sequences of lower level virtual machine code is not feasible, because the decoding overhead imposed in the interpreter will be unacceptable.

It would seem that a reverse Polish interpretive code of some description is the only practical choice for an efficient implementation of Setl. Any encoding of the reverse Polish which depends on word size is ruled out, since portability is a main design objective. By insisting that operators can be distinguished from operands, by a rule such as that imposing a threshold on pointer values which is already present in Setl-s (see 2.3.3) the elements of a Polish string can be fitted into address-sized objects. The advantage of ITC over such a scheme comes from the low decoding overhead. A more conventional interpreter will have a main interpretive loop, which repeatedly fetches the next instruction, performs a switch on its value and then calls the corresponding system routine. By embedding pointers to the routines in the code, an ITC based system short-circuits this process, thereby gaining greater efficiency. Coupled with the flexibility of the interpretive approach and the compact, portable nature of the interpretive code, this makes ITC an attractive method of implementation for this application.

ITC would seem to be equally attractive for implementing any language with high level data types which can conveniently be represented as blocks on a heap. In order to obtain the maximum efficiency it is necessary that the interpreter be coded in a low-level language, so the necessary indirect branches can be made, but the flexibility of the format, resulting from the way in which the threading cycle passes arguments to the system routines,can be obtained at any level.

5.2.2 Miscellaneous Topics

This section brings together several noteworthy features of the interpreter.

The way in which codeblocks are generated in the dynamic area of memory so that they can be garbage collected once the code has become unreachable is a pleasing side-effect of the code format. Related schemes, such as throwaway compiling [BRO76] as well as requiring a quite elaborate mechanism for jumps must allow for the possibility that code may have to be re-compiled. This means that the compiler has to be available at runtime. On the other hand, the effectiveness of the Setl-s approach as a space-saving mechanism is dependent on stylistic features of the Setl program being executed: a tree-structured program will benefit the most from the effect, a program such as the following, though, will gain no benefit at all.

```
Program lupe;
Init done := FALSE;
Loop
    Until done
Do

  {long sequence of calculations}

End Loop;
End;
```

If function-valued variables were allowed in Setl, the effect would be much less useful, since some symbol table entry would always point to the code for a function, which could never be recovered. At present, though, this feature promises to be of more use than it was in

Spitbol, where the absence of control structures leads to the use of many labels, each of which permanently anchors part of the code.

The implementation of type checking for polymorphic operators is a further demonstration of the flexibility of ITC; of particular note is the way in which two more or less separate operator application schemes (APPLn and APPXn) can co-exist, with the code generation being unaffected. The more unusual version of the type checking is the one which uses the APPLn mechanism. This is only economically feasible for heavily overloaded operators, but for them it provides totally secure type checking. There are resemblances between this scheme and the capability approach to protection in operating systems [NW74]. Every object of a particular type possesses a 'capability' in the shape of the entry point identification associated with the block action routine which defines its type. It is only through the EPI that the object can, as it were, gain access to the operator routines associated with that type. Such a mechanism might have a further application in an extensible language system based on the notion of abstract data type, since operator overloading is one way of providing an adequate syntax for specifying the operators on user-defined types. In Setl-s, the capabilities are associated with R-values and the type checking is performed at runtime. However, they could just as easily be associated, by declarations, with L-values (in practical terms, this would mean setting appropriate routines into VRBLKs) and the selection of appropriate operations could be done by the compiler, by using a routine resembling APPLn to select the correct codeword to generate or to detect a type incompatibility.

Perhaps surprisingly, there is little to be said about the data structures and algorithms chosen to implement Setl's high level data types. Since set-like operations underlie a wide variety of algorithms, techniques for their implementation are well known (see, for example [AHU74]). Similarly well-known are the techniques for associative data storage needed for maps. The details of the Setl-s data structures are, to a large extent, determined by the garbage collector as described in Chapter 3. Undoubtedly, some of the fine details could have been subject to more analysis which might have altered some design decisions, but any such analysis would have depended on knowledge of the frequency of the particular set operations in typical Setl programs, and on the answers to such questions as: how frequent are deletions from sets? are multi-valued maps more or less common than single-valued ones? are set operations often performed on maps? Since no Setl implementation is widely available there is no sample of Setl programs from which to derive such knowledge. Consequently, strategies have been adopted which, experience shows, are most likely to be generally acceptable compromises between the various factors likely to affect the system's performance.

The problem of copying on assignment deserves a final mention. The example of the primes program in 5.1.1 demonstrates that always copying values is not a feasible approach, since the copying imposes a considerable overhead which should be avoided wherever possible. In general, it is a very difficult problem to determine at compile time whether a value is modified. The usual alternative approach is to use reference counts. The simplest version of this involves having a flag, which is clear when the object is created and becomes set once there is a pointer to it. If an object to be assigned or added to a set has this flag set, then it has to be copied. Given the restrictions imposed by Minimal and by the garbage collector such a flag would have to occupy a whole word, which is a significant space overhead. (An attempt to modify the block action field to provide a variation on such a scheme, described in [CM79] was abandoned, because it proved more useful to have this field as a unique type identifier.) A more sophisticated approach allows more than one pointer to a block, with the reference count actually being used to count them. In this way, a copy is not made until absolutely necessary, but in practice, keeping the counts correct turned out to be terribly complicated and error-prone, so the present compromise solution was adopted. Although it can lead to the production of blocks which are strictly unnecessary, this solution is safe.

Since the interpretive routines produce new atomic values as the result of operations on such values, there is no harm in having shared pointers to them. This fact is already exploited in the way in which the copying routines do not, in fact, copy atomic blocks. A further space-saving economy could be made, to take advantage of the fact that multiple copies of integers and strings might arise in the course of execution of a program. It should be possible to add an extra pass to the garbage collector, in which all pointers to a specific atomic value would be reset to point to just one block, producing an 'ultimate' compacting garbage collector.

5.3 THE COMPILER

5.3.1 Implementation Of The Parser.

The LR(1) parsers described in, for example, [AJ74] differ from Setl-s in the parsing actions they perform and the tabular representation of the parser employed, even though the parsing algorithm is the same. In such parsers, the parsing action lookahead is not distinguished, instead the lookahead token is used in every state to determine the action to be performed. The shift action absorbs the lookahead token and selects a successor state by consulting the parser tables; the successor state and symbol are stacked and the successor is entered. The reduce action does not absorb the lookahead token. The stack is popped the requisite number of times, and a successor state is determined according to the combination of subject symbol for the production being applied and the state now on top of the stack. Thus, the effort of prefixing the non-terminal to the input stream is avoided. Often, the parser is represented by a pair of tables: a 'parsing action' table has entries giving the parsing action (e.g shift and enter state 111, reduce by production 94) for each combination of current state and lookahead token; a 'goto' table gives the successor state for each combination of current state and non-terminal and is consulted after each reduction. If the states are allocated numbers and these tables are kept as matrices, a very fast lookup is possible. In practice, the size of the tables rules out this representation. (Assuming each table entry can be condensed into a single machine word, the space requirement would be S*(t+N), which is over 160,000 for Setl-s.)

Consequently, sparse matrix techniques are employed, which leads to a representation similar to that used in Setl-s (see [AJ74] for details).

By changing the format of RSBLKs to include pairs of entries giving the successor state to each current state exposed by the reduction, it would be possible to remove all the non-terminal transitions from the CFSM (because this is a deterministic machine [DER69]) and re-define the reduce action in Setl-s to be the same as that in the tabular LR(1) parsers. This would entail extra work in the parser generator, and would affect the number of identical states which could be merged by the optimiser (see 5.3.2) so that the space occupied might actually increase even though the non-terminals would no longer appear explicitly in the RSBLK. Any possible increase in speed does not seem to warrant the effort; after all, although a great deal of attention has been devoted to it in the past, parsing only occupies a small amount of the time spent in compilation. Furthermore, the shift of the non-terminal provides a very neat way for the semantic routines to pass partly-built trees around and avoids the need to use a second stack for this purpose, such as Yacc, for example, requires.

A more significant advantage of the tabular representation comes from its not duplicating the information in lookahead states, as Setl-s inevitably does. However, the only way to avoid this is to abandon the idea of having a unique parsing action in every state; if the parsing action is to be identified by an indirect pointer the overhead of having such a pointer for every table entry would be unacceptable, and any compromise scheme whereby special case were identified would introduce complications and inelegance. The only viable way of using this approach would be to use a totally conventional parse table representation, with the actions encoded into bits in the table entries. Portability considerations forbid packing extra bits into a word which already holds an address-sized object, so it would be necessary to give numbers to the states and decode the entries. Considerations similar to those given in 5.2.1 demonstrate the advantages of the present scheme for this application. This is not surprising, since a table-driven parser can usefully be looked at as a simple interpreter.

The main advantage, and much of the novelty, of the indirect pointer representation of the parser comes from being able to call the garbage collector to remove the parsing data-structure thereby providing a form of overlay within the context of the normal space allocation scheme. In Setl-s, this enables the system to reclaim about 10,000 words of storage; when this space is returned to the operating system, the primes program, obtaining its own minimal workspace, executes in 17+20 pages (instead of 36+20). The necessity of setting up the parser first is nicely dealt with by the SYSXI call, although as explained this may cause portability problems.

5.3.2 Semantic Actions And Code Generation.

An attractive idea which has been pursued by many workers is to automate the production of the translation phase of the compiler as well as the parser. It would seem that interpretive systems are particularly amenable to this sort of treatment, since the code to be produced is closely related to the source language, and the translation does not have to take account of peculiarities of machine architecture. The most attractive method is that of 'syntax-directed transduction', first investigated by Lewis and Stearns [LS68], in which 'transduction elements' indicating the translator output to be produced are appended to the productions of the grammar. Each production has two right parts then, and provided that each occurrence of a non-terminal in the right part of the original production has a corresponding occurrence in the transduction element, it is possible to perform 'parallel derivations' for the two right parts, rewriting corresponding non-terminals at each step. A <u>simple Polish syntax-directed transduction</u> of a grammar is one in which the non-terminals of a production appear in the same order in both right parts, with the non-terminals to the left of any terminals in the transduction element. Lewis and Stearns show that: 'Any translation performed by a (deterministic) pushdown [automaton] can be effectively described as a simple Polish SD translation on an LR(k) grammar.' Clearly, the translation from infix to postfix notation can be so described.

It is open to question whether a DPDA is sufficient to perform the translation from Setl into the Polish ITC. In order to satisfy the simple Polish condition for the transduction, substantial manipulation of the Setl-s grammar would be required, which tends to upset the SLR(1) condition required by the parser generator. An early attempt to use such a scheme in Setl-s was, therefore, rapidly abandoned in favour of the present approach of building a parse-tree and generating code from it by a recursive tree walk. The translation into the tree is so trivial that more work would be involved in automating it than in writing and modifying it by hand.

Alternative schemes for embodying translations and semantic actions into the grammar for processing by the parser-generator involve the specification of actions to be performed at each reduction. These can be specified either in some pseudo programming language which is translated by the parser-generator, or else in the parser implementation language, in which case the actions are incorporated into the parser with the parser-generating system taking care of certain book-keeping operations. This approach leads to a critical interdependence between the parser and the parser-generator, which undermines the usefulness of both. For example, in Yacc [JOH78], the semantic actions are written in the programming language C and the parser produced is a C program. Thus, any compiler which uses Yacc must be written in C.

5.3.3 Parser Generation.

LR parsing has become widely accepted, owing to its speed and early error detecting capability and the ease with which parsers can be generated automatically. Its main advantage over the LL(k) methods is that no restrictions, such as forbidding left recursion or null right parts, are imposed on the grammars from which the parsers are generated. It was thought desirable to use some parser generating method for Setl-s because during the system's early development the definition of the Setl language was changing constantly, and re-generating parsers from a modified grammar was thought preferable to continually modifying a hand-built parser.

The SLR(1), LALR(1) and LR(1) methods define increasingly large classes of _grammars_ (using values of k>1 does not significantly increase the class [HOR76]), but all produce recognisers for the same class of _languages_: the _deterministic_ _languages_. The difference between the parser generating algorithms is in the way in which they compute lookahead sets. Since no parser generator was available at Leeds it was necessary to construct one in a short time, so the SLR(1) algorithm was chosen, as it is significantly simpler than the others. It has turned out that a certain amount of re-writing of the grammar has been required leading to some awkward-looking productions, before Setl-s could be made to satisfy the SLR(1) condition, and finally some purely syntactic restrictions have had to be imposed semantically. By examining the output from Slrgen, it is possible to see that some of this awkwardness could have been avoided if a more powerful constructor algorithm had been employed. The objections to an awkwardness in the grammar are not merely aesthetic, since the productions are used to direct the tree building and a corresponding awkwardness results in the tree building routine. To a large extent, the design of the Setl syntax is at fault, with too many features being packed in with no regard for the overall pattern, but Setl is by no means unique in this respect.

Further improvements could be made to Slrgen to make it into a more useful tool. It is possible to modify the constructor algorithm so that it will accept regular expressions in the input grammar, which would be more convenient and could lead to smaller parsers. The Slropt phase could also be improved: as a consequence of the parser representation chosen, the 'single reductions' which are successors to successors to lookahead states (i.e. those which are originally successors to inadequate states) cannot be removed without a substantial computational effort, which so far has not been considered worthwhile, since over 96% of the single reductions do get eliminated. Unfortunately, the remaining few arise from the grammar of expressions, and consequently, as a trace of the parse shows, the parser spends much of its time on them.

Figure 5.2 gives a breakdown of the states of the parser, and demonstrates the effect of the optimisations. The figures for the single reductions refer to the number of transitions into single reduction states. Although no duplicate lookahead sets are found with this grammar, in previous versions this optimisation has had some effect. (Slropt only makes a single pass through the parser merging duplicates; this and subsequent operations might produce further duplicates, so the process should be iterated.) The figures are for the grammar used during system development, without any error productions; the production grammar will probably be smaller, since certain features in the original design have been dropped.

Figure 5.2 Parser states and optimisations

Grammar	
no of productions	242
no of terminals	69
no of non-terminals	119
total no of symbols	188
Parser States	
shift	545
reduce	257
lookahead	72
total	874
(2 inadequate states resolved by hand)	
Optimisations	
duplicate states merged	90
duplicate sets merged	0
single reductions eliminated	1262
single reductions left	48

Notice that it is possible to alter the generating algorithm and add these extra optimisations and, as long as the PCX format is retained, the parser itself need not be altered. Equally, the PCX can be manipulated to produce tables for any LR parser. In cases where it is not possible to save a core image and memory is freely available it would be possible to generate Minimal declarations so that a non-collectable parser could be built in the working storage section. On the other hand a BCPL program or Fortran data statements might be generated to suit the requirements of completely different compilers. The processing required to transform the PCX in this way is relatively straightforward.

5.4 FURTHER WORK

5.4.1 Improvement

Setl-s could be used as a working system with the addition of some simple operators, the completion of the error recovery scheme and a certain amount of tidying up, to prevent, for example, the use of an unimplemented feature causing chaos. I would estimate that, under favourable conditions, this work could be done in less than a fortnight. Significant improvements to the system could, however, be made, most of which have already been mentioned in passing. Perhaps most important is the syntax error handling. On the one hand, a careful design of error productions to be added to the grammar could be carried out, in an effort to establish the right balance between specificity and accuracy of error messages. If too many error productions are used to produce highly specific error messages the grammar will inevitably become ambiguous, and arbitrary parsing decisions will have to made (equivalent to guessing what was meant) which might produce misleading error messages. On the other hand, a scheme to make use of local context information to catch such simple but elusive errors as omitted semi-colons could be added.

The other area for improvement is optimisation. First some of the interpreter's code could be optimised, an example being the string building operation mentioned in 5.1.1. Secondly, some of the code sequences generated could be improved. Finally, a more far-reaching project would be to make use of the flow information in the code to perform more sophisticated optimisations.

5.4.2 Additions.

The most pressing additional requirement is for a proper system of input and output. At present, these can only be performed to standard channels, established at compile-time, unless a crude ad hoc method of re-defining file associations which is DEC-10 dependent is used. Read is still improperly implemented. There should be no particular difficulty in adapting Spitbol's i-o routines, although some modifications will be required to deal with the particular external representations used in Setl.

Another feature which could be added from Spitbol is the Trace facility (described in [DM77]). The ITC format permits easy trace association of variables, and Snobol4 programmers have found the tracing facilities of considerable value. No facilities exist in Setl for specifying tracing, so it would be necessary to design the semantics of this. It might also be helpful to provide a symbolic dumping facility.

Several improvements to the parser generating system were suggested in 5.3.4. To produce a significant improvement in this system, the present version which was simply designed as an implementation tool for Setl-s could be re-written to incorporate the recently published efficient LALR(1) constructor algorithm [DP79]. It might also be worth investigating the use of precedence and associativity declarations, as used in Yacc, to produce a smaller parser from an ambiguous grammar.

CHAPTER 6

INDIRECT THREADED CODE AND SEMANTICS

6.1 PROGRAMMING LANGUAGE SEMANTICS

6.1.1 Mathematical Semantics

An interpreter is concerned with the semantic aspects of a programming language. The way in which a particular interpreter implements a language can only be described precisely if there is a precise means of describing the semantics of the language. The most appropriate description for this purpose is provided by the method known as denotational semantics, or mathematical semantics, which is associated primarily with the work of Scott and Strachey. Axiomatic semantics, although suitable for use in proofs about the behaviour of programs, cannot easily be related to implementations. Operational methods are equally unsuitable for the present purpose; by defining a language in terms of an abstract interpreter, operational descriptions do not provide a basis for the description of another interpreter. At best, a similar description of the second interpreter could be compared with the first, but methods of interpretation can vary considerably, so that such comparisons become difficult and obscure, and by concentrating on the details of the interpreters obscure the role of the language. Furthermore, neither interpreter can claim to be canonical. More insight can be gained by defining the interpreter in terms of the abstract meaning of the language, as given by its denotational description. The operational description obtained in this way will be directly related to the semantics of the language, and can be used to illuminate the internal workings of the implementation and possibly to prove its correctness.

A denotational description of a language comprises a set of valuations, which are functions mapping constructs of the language into their meaning in appropriate mathematical domains. A useful introduction to the denotational approach to semantic description is given in [TEN76] which summarises the work described in earlier papers [SS71, STR73, SW74]. The mathematical basis for the use of higher order functions and reflexive domains is given in [SCO76]. The description of implementation techniques in terms of valuations which map program texts into transformations on stacks and stores was first presented in [MIL74]; the most comprehensive description of the subject appears in the book by Milne and Strachey [MS76].

The presentation of semantic descriptions in this chapter uses conventions which are commonly adopted, although the restricted character set available for printing has necessitated some departures from the most usual notations. A set of syntactic domains is defined, which group together syntactic features of a language with common semantic properties; an example of this is the domain Com of commands. Each syntactic domain has an associated meta-variable, designated by an upper-case Roman letter. The syntax is presented in an idealised (or abstract) form as a set of BNF-like productions involving the syntactic meta-variables, possibly with subscripts or primes. The syntax is usually ambiguous and informal (for example, the ellipsis ... is often used instead of recursion) being intended only to convey the essential structural features of the language, and not to provide the detailed information required for parsing. The idealised syntax may be thought of as describing nodes of the parse tree produced after a conventional syntax analysis.

The _semantic domains_ are designated by a single upper-case letter. They are built up by combining some basic domains using the operators + (sum) x (product) and -> (which forms the space of continuous functions between its two operand domains). The references just cited should be consulted for the precise technical meaning of these operators. The valuations take their first argument from some syntactic domain and may take other, semantic, arguments. Each valuation is specified by a set of equations, one for each clause in the syntactic definition for the relevant syntactic domain. Valuations are designated by capital letters with underlines, since the more usual script letters are not available. The syntactic arguments appear on the left hand side; the right hand side is written in λ-notation, using standard conventions for associativity and the added convention that $\lambda x.\lambda y.F$ is abbreviated to $\lambda xy.F$. The arguments in these λ-expressions are lower-case Greek letters, each of which is associated with a particular semantic domain (the associations are largely arbitrary, depending mainly on the availability of particular Greek letters). It is therefore possible to deduce immediately from the equations the domains of their arguments. The notation $\delta : D$ indicates that δ is a proper member of

D.

A member of a product domain $D_0 \times D_1$ is written $\langle\delta_0, \delta_1\rangle$. Extended products $D_0 \times D_1 \times \ldots D_n$ are permitted, giving rise to sequences $\langle\delta_0, \delta_1, \ldots \delta_n\rangle$. Three operators are defined for manipulating sequences:

$\langle\delta_0, \delta_1, \ldots \delta_n\rangle \downarrow m = \delta_{m-1}$

$\langle\delta_0, \delta_1, \ldots \delta_n\rangle \dagger m = \langle\delta_{m+1} \ldots \delta_n\rangle$

$\langle\delta_0, \delta_1, \ldots \delta_m\rangle \S \langle\delta_{m+1} \ldots . \delta_n\rangle = \langle\delta_0 \ldots \delta_n\rangle$

providing the arguments are all in range. # gives the number of elements in a sequence: $\#\langle\delta_0 \ldots . \delta_n\rangle = n+1$. The domain of sequences of finite length taken from a domain D is written

$D^* = \{\langle\rangle\} + D + D \times D + D \times D \times D + \ldots$

6.1.2 Standard Semantics Of X10

Figure 6.1 gives the standard semantics of a simple language known as X10, which has been designed to illustrate the ideas of this chapter. The meaning of X10 programs is here built up from abstract objects which are independent of the particular representations which might be used inside a computer, and which therefore give no information about how the language is to be implemented. The standard semantics is thus an appropriate standard with which to compare an actual implementation.

The intention of most of the clauses in the grammar should be apparent. Expressions, which return a value, are distinguished in X10 from statements, which do not. Expressions can be formed using

Figure 6.1 Standard Semantics of X10

Syntactic Domains

B : Bas	bases
O : Mon	monadic operators
W : Dya	dyadic operators
I : Ide	identifiers
E : Exp	expressions
C : Com	commands

Abstract Syntax

$$E ::= O\ E \mid E_0\ W\ E_1 \mid B \mid I$$

$$C ::= C_0 C_1 \mid I := E \mid E_0 \to C_0\ E_1 \to C_1 \ldots E_n \to C_n$$
$$\mid \text{While } E \text{ Do } C$$

Semantic Domains

$\alpha : L$	locations
$T = \{tt, ff\}$	
B	basic values
$\beta : V = B + T + \text{undef}$	stored values
$\varepsilon : E = B + T + L$	expressed values
$\rho : U = \text{Ide} \to L$	environments
A	answers
$\theta : C = S \to A$	command continuations
$\gamma : K = E \to C$	expression continuations
$\sigma : S = L \to V$	stores

Valuations

$\underline{B} : \text{Bas} \to K \to C$
$\underline{O} : \text{Mon} \to K \to E \to C$
$\underline{W} : \text{Dya} \to K \to E \to E \to C$
$\underline{E} : \text{Exp} \to U \to K \to C$
$\underline{L} : \text{Ide} \to U \to K \to C$
$\underline{C} : \text{Com} \to U \to C \to C$

$\underline{E}[\![O\ E]\!] = \lambda\rho\gamma.\underline{E}[\![E]\!]\ \rho(\underline{O}[\![O]\!]\ \gamma)$

$\underline{E}[\![E_0\ W\ E_1]\!] = \lambda\rho\gamma.\underline{E}[\![E_0]\!]\ \rho(\lambda\varepsilon'.\underline{E}[\![E_1]\!]\ \rho(\lambda\varepsilon'\varepsilon''.\underline{W}[\![W]\!]\ \gamma\varepsilon'\varepsilon''))$

$\underline{E}[\![B]\!] = \lambda\rho\gamma.\underline{B}[\![B]\!]\ \gamma$

$\underline{E}[\![I]\!] = \lambda\rho\gamma.(\rho[\![I]\!])(rv\gamma)$

$\underline{L}[\![I]\!] = \lambda\rho\gamma.\gamma(\rho[\![I]\!])$

$\underline{C}[\![C_0C_1]\!] = \lambda\rho\theta.\underline{C}[\![C_0]\!]\ \rho(\underline{C}[\![C_1]\!]\ \rho\theta)$

$\underline{C}[\![I := E]\!] = \lambda\rho\theta.\underline{E}[\![E]\!]\ \rho(\lambda\varepsilon'.\underline{L}[\![I]\!]\ \rho(\lambda\varepsilon''.assign\theta\varepsilon'\varepsilon''))$

$\underline{C}[\![E_0 \to C_0\quad E_1 \to C_1 \ldots E_n \to C_n]\!] =$

$\qquad \lambda\rho\theta.\underline{E}[\![E_0]\!]\ \rho test<\underline{C}[\![C_0]\!]\ \rho\theta,\ \underline{E}[\![E_1]\!]\ \rho test<\underline{C}[\![C_1]\!]\ \rho\theta,$

$\qquad\qquad \underline{E}[\![E_n]\!]\ \rho test<\underline{C}[\![C_n]\!]\ \rho\theta,\ \theta>\ldots>$

$\underline{C}[\![\text{While } E \text{ Do } C]\!] =$

$\qquad \lambda\rho\theta.fix(\lambda\theta'.\underline{E}[\![E]\!]\ \rho test<\underline{C}[\![C]\!]\ \rho\theta',\ \theta>)$

Auxiliary Functions

$test = \lambda<\theta', \theta''>\beta.\beta ET \to (\beta|T \to \theta', \theta''), wrong$

$assign = \lambda\theta\varepsilon'\varepsilon''\sigma.\theta(update(\varepsilon'|L)\varepsilon''\sigma)$

$update = \lambda\alpha\beta\sigma.(\lambda\alpha'.(\alpha' = \alpha) \to \beta, \sigma\alpha')$

$rv = \lambda\gamma\varepsilon\sigma.\varepsilon EL \to (\sigma(\varepsilon|L) = undef \to wrong\sigma,\ \gamma(\sigma(\varepsilon|L))\sigma,\ \gamma\varepsilon\sigma$

infixed dyadic operators and monadic operators from identifiers and an unspecified set of bases, which might include the numerals and some representation of the truth values true and false. Typing is dynamic and there are no declarations; nor are there any procedures, as these would introduce complications which are tangential to the present discussion. The language is imperative, and it includes a simple form of assignment. Conditional flow of control is provided by a multi-armed conditional, shown as the third clause of the definition of C, in which each of the E_i is evaluated in turn until one of them yields true, when the corresponding C_i is executed. The while loop behaves in the conventional fashion. Additional syntactic details are contained in the grammar in Appendix 2, which was used by Slrgen to generate an X10 parser.

The semantic domains include a domain of basic values B which corresponds to the syntactic bases, and includes the objects manipulated by the program; its structure will not be examined. The truth values have been distinguished from the basic values because of the way in which they are used in tests. The meaning of an identifier will be taken to be a location, and the environment ρ : U provides the mapping from identifiers to the locations they denote. In addition, a function from locations to their contents is required; this is the store σ. There is no reason why S should not be taken as Ide->V in this example, as locations cannot be shared and there is no way in which the location denoted by an identifier can be altered, but it is more customary to make the store and environment separate, and later on the presence of locations will be useful. The domain of stored values includes an undefined value **undef** which is the initial contents of all locations. The values of all expressions will always be coerced into R-values, hence no valuation $\underline{R}$ is required.

The remaining domains are the continuation domains, the use of which was first described in [SW74]. Continuations are required to provide a satisfactory account of transfers of control, resulting from jumps or the occurrence of certain sorts of error. It is possible to describe a language which does not permit unrestricted jumps without using continuations, but this makes the handling of errors very awkward. Later, when the ITC implementation of X10 is described, continuations will play an important part, so it is worthwhile reviewing their use.

Early versions of mathematical semantics gave the meaning of a command C, $\underline{C}[\![C]\!]$, as a transformation from states (or stores) to states. Thus, if C were executed in an environment ρ and state σ, the resulting state would be $\sigma' = \underline{C}[\![C]\!]\rho\sigma$. To express the meaning of C_0C_1, that is of executing first C_0 then C_1, one would therefore write $\underline{C}[\![C_0C_1]\!]\rho\sigma = \underline{C}[\![C_1]\!]\rho(\underline{C}[\![C_0]\!]\rho\sigma)$ which signifies the transformation corresponding to C_1 being applied to the store resulting from the execution of C_0 in the original store. Unfortunately, this is not adequate, since the execution of C_0 may result in an error trap or may never terminate, in which case $\underline{C}[\![C_0C_1]\!]\rho\sigma$ should be equal to $\underline{C}[\![C_0]\!]\rho\sigma$. The solution to this consists in defining a domain of 'answers' A, which contains, among other things, the results of erroneous computations. The meaning of a command is now a transformation from S to A; such a transformation is called a continuation, and the domain of continuations C = S->A, with θ being a typical member. In order to accomodate sequencing, the valuation for a command is given a continuation as an extra argument. This continuation models the execution of the rest of the program (and presumably yields a final outcome). In the normal course of events, the continuation is applied to the store resulting from the execution of the command, to give the final outcome. If, however, a jump or an error occurs the continuation is thrown away. As is customary, the arguments are supplied to the valuation one at a time, so that $\underline{C}$: Com->U->C->S->A, which is Com->U->C->C, and the equation for the sequential execution of two command becomes $\underline{C}[\![C_0 C_1]\!] \rho\theta = \underline{C}[\![C_0]\!] \rho(\underline{C}[\![C_1]\!] \rho\theta)$. A pleasing side-effect of this notation is that when semantic equations of this sort are read from the left to right, the components are read in their order of execution.

The execution of an expression may also fail to terminate or may result in an error, so continuations must be supplied to the valuations for expressions as well. An expression yields a result as well as a possibly altered store (expressions in general being allowed to have side-effects), so that the functionality of expression continuations has to be E->S->A which is E->C. The domain of expression continuations is signified by K, with γ as a typical member, and the valuation $\underline{E}$ is taken in Exp->U->K->C. It should be clear how continuations are used in the other valuations in figure 6.1; if it is not, the reader is referred to section 1.5 of [MS76]. One particular continuation of practical importance is wrong : S->A, which is applied when an error occurs.

6.2 INDIRECT THREADED CODE

6.2.1 The Semantic Domains

The ITC produced from an X10 program in a realistic implementation will consist of a series of codewords, each of which points to a block whose first word points to the entry point of a system routine. An abstract description of this code need not be concerned with these pointers, which are only necessary to accomodate the organisation of memory in a computer. Instead, the code can be considered as a sequence of blocks, each of which is a pair whose first component is a routine and whose second component is drawn from a suitable domain of block values. This domain of block values, H, will include all the stored values from the standard semantics and also locations, since one sort of block must resemble a VRBLK. In addition, H must include sequences of codewords, since the domain of blocks includes codeblocks. This leads to the following collection of domains:

$\beta : V$	storable values
$\mu : M = X \times H$	blocks
$\omega : W = M^*$	code sequences
$\eta : H = V+L+W$	block values

The fact that M^* is embedded in M may look suspicious at first, but the use of such self-referencing domain equations can be justified on technical grounds, provided that certain restrictions on the domain structure are observed ([MS76] gives the full details).

The functions in X (of which ξ is a typical member) take a member of H as an argument and produce a state transformation, so it might seem that X = H->S->S. However, since the execution of ξ might cause an error or never terminate, considerations similar to those of 6.1.2 suggest that the second argument to ξ should be some sort of continuation, rather than a store. In order to eliminate the distinction between expressions and commands, and to make explicit more of the implementation, a domain of stacks Y is introduced, where $Y = V^*$, that is, a stack is a sequence of storable values. The domain of command continuations C is defined to be S->A as before. The continuations appropriate to codewords receive a stack as an additional argument, much as expression continuations received an expressed value. Therefore, a domain of 'pure continuations' is defined: Z = Y->C. It is then appropriate to write X = H->Z->Z.

6.2.2 Block Action Routines

A typical member of X is the routine to load an integer value onto the stack, which could be defined as:

$$\text{icget} = \lambda\eta\delta\tau\ \delta(\langle\eta\,|\,V\rangle \S \tau)$$

The structure of the domains which have been described shows the important feature of ITC: the state transformations are embedded in the blocks, along with the values being operated on. Thus, although the code is interpretive, no interpreter need be defined, except for the mechanism to apply the block action routines which form the first components of the blocks to the values which form their second components.

What is required is a function lcw (load codeword) which takes as its operands a code sequence (in W) and a code pointer (offset), and applies the function part of the corresponding block to its value; it must also supply it with a continuation which will execute the next codeword so that the indirect threading cycle will continue. A recursive definition is:

$$\mathrm{lcw} = \lambda\omega\pi.\omega{\downarrow}\pi{\downarrow}1(\omega{\downarrow}\pi{\downarrow}2)(\mathrm{lcw}\,\omega(\pi+1))$$

Technically, it is the least fixed point of this equation which is needed:

$$\mathrm{lcw} = \lambda\omega\pi.\mathrm{fix}\,\lambda\phi.(\omega{\downarrow}\pi{\downarrow}1(\omega{\downarrow}\pi{\downarrow}2)(\phi\omega(\pi+1)))$$

The appropriate function to use as the first component of a codeblock is thus:

$$\mathrm{go} = \lambda\eta\delta\tau.\mathrm{lcw}(\eta|W)1\tau$$

The continuation δ is discarded and control passes through the codewords of the codeblock, in sequence via a threading process. The continuation $\mathrm{lcw}\omega\pi$ 'remembers' the current code pointer and codeblock, so that these need not appear explicitly in the continuation domains. This does necessitate the adoption of a different implementation of conditional jumping from that used in Setl-s. A conditional jump consists of a block whose body is the destination of the jump, and whose first component is the routine jt, defined by:

$$\mathrm{jt} = \lambda\eta\delta\tau.(\tau{\downarrow}1\mathrm{E}T\rightarrow(\tau{\downarrow}1\rightarrow\mathrm{lcw}(\eta|W)1(\tau{\dagger}1),\delta(\tau{\dagger}1)),\mathbf{wrong})$$

If the top stack item is a truth value and is true the continuation δ (carrying on with the old codeblock) is discarded and execution of the body of the jump begins, otherwise the old code block is continued. This precludes the possibility of merging back after the conditional part has been executed but, as will be seen, the abstract model will never require this.

There remains the implementation of operators to be considered. It will be recalled that in macro Spitbol an operator is represented in the generated code by an indirect pointer to the routine to perform the operation. The corresponding abstract representation would be a block with a null second component. An example of the operator routines in such a scheme is given by the following routine to perform integer addition:

$$\mathrm{intplus} = \lambda\eta\delta\tau.(\lambda\beta_1.(\beta_1|\mathrm{BEN} \wedge \beta_2|\mathrm{BEN} \rightarrow \delta(\langle\beta_1+\beta_2\rangle\S\tau\uparrow 2),\ \mathrm{wrong}))\ (\tau\downarrow 1)\tau\downarrow 2)$$

No use is made of the argument η.

In Setl-s, however, another implementation of operators has been adopted, to cater for the additional polymorphism. A typical dyadic operator would be a pair <appl2, ø> where $ø : P^*$ was a sequence of functions in Z->Z which would perform the indicated operations on the top two stack items, and

$$\mathrm{appl2} = \lambda\eta\delta\tau.(\eta\downarrow(\mathrm{type}(\tau\downarrow 2)))\delta\tau$$

where type : V->N will return an integer code for the value, corresponding to its type. The continuation δ has to be passed on to the evaluating routine as this may fail to terminate properly, owing to overflow for example.

6.2.3 A Translator For X10

To model the operation of an X10 interpreter, X10 programs have to be translated into code sequences of the sort just described.

Figure 6.2 shows a suitable translator. The equations of figure 6.2 bear a close resemblance to those of figure 6.1. They may be interpreted (in the non-technical sense) in either of two ways. The code sequences produced might legitimately be considered as the meaning of the program as they are composed from sequences of state transformations. In this interpretation, the equations in figure 6.2 provide an alternative definition of the semantics of the language, albeit not a very useful one. (However, see 3.5.4 of [MIL74], in which a domain of answers consisting of state transformations is suggested as a possible means of supplying distinct meanings to separate unending computations.) Alternatively, the code sequences can be thought of as an interpreter for the program, which can be sent through the state transformations to yield a final outcome. The second view is more suited to providing an understanding of ITC as an interpretive method.

The essential difference between ITC and more conventional interpretive schemes, both practical and abstract, is that the translation only involves semantic domains: no new syntactic entities such as abstract machine instructions have been introduced, so there is no need to add an extra level of definition to describe the interpreter itself. In an actual implementation, this feature of the code is responsible for increased efficiency, as no decoding corresponding to this extra level of definition is required.

The valuations provide a straightforward translation into a stack-based implementation. The extra argument supplied to $\underline{C}$ resembles a command continuation in that it represents the rest of the

Figure 6.2 The X10 Translator

Syntactic Domains

B : Bas	bases
O : Mon	monadic operators
W : Dya	dyadic operators
I : Ide	identifiers
E : Exp	expressions
C : Com	commands

Abstract Syntax

$E ::= O\ E \mid E_0\ W\ E_1 \mid B \mid I$

$C ::= C_0 C_1 \mid I := E \mid E_0 \rightarrow C_0\ \ E_1 \rightarrow C_1 \ldots E_n \rightarrow C_n$

$\mid \text{While } E \text{ Do } C$

Semantic Domains

$\alpha : L$	Locations
$T = \{tt, ff\}$	
B	basic values
$\beta : V = B + T + \text{undef}$	stored values
$\varepsilon : E = B + T + L$	expressed values
$\rho : U = \text{Ide} \rightarrow L$	environments
A	answers
$\theta : C = S \rightarrow A$	command continuations
$\gamma : K = E \rightarrow C$	expression continuations
$\sigma : S = L \rightarrow V$	stores
$\tau : Y = V^*$	stacks
$\delta : Z = Y \rightarrow C$	pure continuations
$\mu : M = X \times H$	blocks
$\omega : W = M^*$	code sequences
$\eta : H = V+L+W$	block values
$\xi : X = H \rightarrow Z \rightarrow Z$	block action routines

Valuations

$\underline{B} : \text{Bas} \rightarrow K \rightarrow C$

$\underline{O} : \text{Mon} \rightarrow K \rightarrow E \rightarrow C$

$\underline{W} : \text{Dya} \rightarrow K \rightarrow E \rightarrow E \rightarrow C$

$\underline{E} : \text{Exp} \rightarrow U \rightarrow K \rightarrow C$

$\underline{L} : \text{Ide} \rightarrow U \rightarrow K \rightarrow C$

$\underline{C} : \text{Com} \rightarrow U \rightarrow C \rightarrow C$

$\underline{E}[\![\, O \; E]\!] = \lambda\rho . \underline{E}[\![E]\!] \, \rho \, \S \, \underline{O}[\![O]\!]$

$\underline{E}[\![E_0 \; W \; E_1]\!] = \lambda\rho . \underline{E}[\![E_0]\!] \, \rho \, \S \, \underline{E}[\![E_1]\!] \, \rho \, \S \, \underline{W}[\![W]\!]$

$\underline{E}[\![B]\!] = \lambda\rho . \underline{B}[\![B]\!]$

$\underline{E}[\![I]\!] = \lambda\rho . \langle \rho[\![I]\!] \downarrow 1, \; \rho[\![i]\!] \downarrow 3 \rangle$

$\underline{L}[\![I]\!] = \lambda\rho . \langle \rho[\![I]\!] \downarrow 2, \; \rho[\![I]\!] \downarrow 3 \rangle$

$\underline{C}[\![C_0 \; C_1]\!] = \lambda\rho\omega . \underline{C}[\![C_0]\!] \, \rho \, \S \, \underline{C}[\![C_1]\!] \, \rho \, \S \, \omega$

$\underline{C}[\![I := E]\!] = \lambda\rho\omega . \underline{E}[\![E]\!] \, \rho \, \S \, \underline{L}[\![I]\!] \, \rho \, \S \, \omega$

$\underline{C}[\![E_0 \texttt{->} C_0 \quad E_1 \texttt{->} C_1 \quad E_n \texttt{->} C_n]\!] =$

$\qquad \lambda\rho\omega . \underline{E}[\![E_0]\!] \, \rho \, \S$

$\qquad\quad \langle jt, \; \langle \underline{C}[\![C_0]\!] \, \rho\omega \rangle\rangle \S \underline{E}[\![E_1]\!] \, \rho \, \S \langle jt, \; \langle \ldots$

$\qquad\qquad ..\underline{E}[\![E_n]\!] \, \S \langle jt, \; \underline{C}[\![C_n]\!] \, \rho\omega \rangle\rangle \ldots \rangle \S \omega$

$\underline{C}[\![\text{While } E \text{ do } C]\!] =$

$\qquad \lambda\rho\omega . \mathrm{fix}(\lambda\omega' . \underline{E}[\![E]\!] \, \rho \, \S \langle jt, \; \langle \underline{C}[\![C]\!] \, \rho\omega' \rangle\rangle \S \omega)$

program; it is necessary in order to translate the jumps resulting from conditionals and loops. There is no need to supply any analogue of an expression continuation to $\underline{E}$ since an expression in X10 cannot involve jumps, and the possibility of an improper termination is handled by the continuations in Z embedded in the blocks. The denotations of identifiers are taken in XxXxV, the members of which resemble VRBLKs in having two routines in them. $\underline{L}$ and $\underline{E}$ build blocks in M from these. This method seems rather contrived, and it might be better to have an assignment operator, which would generalise more readily to more complex forms of assignment. However, the use of something resembling VRBLKs helps to emphasise the links between this abstract interpreter and the actual implementation.

6.2.4 Relation To Standard Semantics

It may appear that the description of indirect threaded code just given differs from an implementation in a high level language merely in the notation used, so it is worth emphasising that the entities involved in the description belong to a class of mathematical objects with well understood properties, which makes it possible to prove theorems about their behaviour. In particular, it is possible to prove that an abstract ITC-based interpreter in fact implements the language which is described by the standard semantics. In practice, such a proof would be long and tedious, so an outline of the reasons for believing it possible is all that will be given here.

In essence, the ITC description makes explicit the series of state transformations which the execution of the X10 program undergoes, by keeping the block actions in X separate from their arguments in H. By applying the functions to their arguments a function in Z->Z results. Thus, there is a straightforward relationship between the members of W(=[XxH]*) and members of [Z->Z]*. Equally, the sequences in this domain can be composed to produce values in the domain [Z->Z]. That is to say, members of W can be mapped into [Z->Z] by successive application of the two functions:

$$A = \lambda\omega . \lambda\langle\xi, \eta\rangle . (\#\omega = 1 \rightarrow \langle\xi\eta\rangle, \langle\xi\eta\rangle \S A(\omega \dagger 1))(\omega \downarrow 1)$$

$$S = \lambda\phi\delta . \#\phi = 1 \rightarrow (\phi\downarrow 1)\delta, S(\phi\dagger 1)(\phi\downarrow 1)\delta$$

where ϕ : [Z->Z]*.

(Actually, A and S are the least fixed points of these equations.) But [Z->Z] is essentially the domain of 'pure code' belonging to store semantics and stack semantics, which are two well-established sorts of semantics, developed in [MIL74] for describing implementations and implementation-dependent language features. (since the environment in X10 is a constant the difference between the two is immaterial). It is therefore possible to set up predicates relating the ITC valuations for X10 to its stack semantics, which express the requirement that the 'answer' embedded in the ITC be the same as that in the stack semantics. The equivalence of the two sorts of semantics can be shown by a structural induction on the clauses of the valuations, although since A and S depend on the length of the sequences some induction on their length will also be required. An appeal can then be made to the proof in [MS76] of the congruence between stack and store semantics and standard semantics.

Since the proof contemplated here rests on the properties of the retracts associated with the semantic domains, it leads into an area of mathematics which is a substantial subject of study in its own right. Furthermore, it necessitates the setting up of a stack semantics for X10 as the intermediate step in the proof, which adds nothing to the understanding of ITC. Therefore an appeal will be made to the reader's intuition (or good will) and the relationship between the theoretical descriptions of this chapter and the nature of actual implementations will be taken up instead.

6.3 REALISATION OF THE X10 SYSTEM

6.3.1 The Introduction Of Pointers

The equations of Figure 6.2 embody the interpretive scheme of indirect threaded code, but they are a long way removed from any actual implementation. The codewords in the abstract description are members of a domain built out of abstract function spaces; the codewords generated on a computer are bit patterns representing machine addresses. The model of the store which has been adopted makes it possible to incorporate this feature in the abstract model by making the code sequences be in L^* instead of M^*, and adding the appropriate indirection to the definition of lcw. This modification also permits the removal of fix from the equation for the While-loop, because, as is intuitively obvious, the effect of taking the fixed point of the recursive definition is achieved by building a code sequence which includes a pointer to the head of itself. This introduces the need for the function go introduced in 6.3.2 as unconditional control transfers are now required. In order to model the translation into such a form of code, the valuations would require a mechanism for performing operations analogous to the setting and referencing of labels; this is probably best achieved by incorporating an element into the environment to record the address of the head of the loop.

The locations pointed at by the codewords cannot contain the pair $<\xi, \eta>$, since the former is a member of an abstract space and the latter may be a structured object or may not fit into a machine word. The blocks then must be represented by two locations, one containing a pointer to a piece of code, the other containing a pointer to the value. The simple optimisation of placing the locations holding the value next to the pointer to the code (assuming that store is arranged contiguously) leads to the representation used in Setl-s and macro-Spitbol for values. A similar approach leads to the VRBLK representation of variables.

In a suitable programming language, it is possible to make use of higher order functions in the implementation language to provide such functions as lcw, if the parameter passing mechanism is appropriate. Few languages provide this facility so a further modification is required before the semantic equations can be used as the basis for an implementation. This modification is a desirable optimisation anyway, to prevent recursion reaching an unacceptable depth; it consists of some form of recursion removal, which, depending on the properties of the implementation language may best be done by replacing calls of lcw by jumps or by placing a single-level call inside a loop.

There is clearly a lot of work involved in developing such a description of an implementation, proving its correctness, writing a program based on it and proving that correct. It would seem that, with the technology of formal proof methods in its present state, it is better to use a description of the level of that in figure 6.2 as a basis for an implementation and to rely on informal methods to show the correctness of the program. It is to be hoped, however, that general results might be proved to show the validity of the optimisations proposed in the preceding paragraph.

6.3.2 Interpretive Routines

The members of the domain X can be implemented as pieces of pure code receiving an argument drawn from H. The recursion removal required for lcw can also be applied to these routines, so that they are entered not by an ordinary procedure call, but by some other mechanism. Clearly, the indirect branch used in Setl-s provides a particularly appropriate mechanism and passes the argument automatically via the pointer left in XR. The continuation argument is here sealed in to the code by ending all the routines with a jump to a piece of code to perform the lcw operation. (Perhaps this would be more clearly related to the abstract description had X been taken as Z->H->Z instead of H->Z->Z.)

The description of an ITC-based system given so far is lacking in one important respect: the interpretive routines themselves. The only routines which feature in the description of X10 are jt and go (the latter only when pointers have been introduced) which are concerned with control transfers; any other such routines will arise from the valuations $\underline{B}$, $\underline{O}$ and $\underline{W}$ which have been left undefined. Between them, these three valuations define what can be talked about in the language, whereas the others are concerned with the order in which the operations are performed. In [WIL68], Wilkes pointed out a distinction between what he called 'the inner and outer syntax' of programming languages; in his terms, the outer syntax is concerned with the organisation of the flow of control and the inner syntax is concerned with the operations performed on data objects. Wilkes' discussion was presented in terms of syntax, and as will be seen in 7.1, for practical purposes the syntax of a language must be considered when such a distinction is made. However, it is really a semantic distinction, and, following Wilkes, I will refer to the inner and outer semantics of a programming language. In the simple case of X10, the outer semantics comprises $\underline{E}$, $\underline{L}$ and $\underline{C}$ and the inner $\underline{B}$, $\underline{O}$ and $\underline{W}$; the valuations of Figure 6.1 provide a full definition of the outer semantics, without giving any indication what it is that programs in the language are about, which might be integers, personnel records, polynomials or even sets. There is a parallel between the inner semantics and the mathematical concept of an interpretation of a system. It would be tempting to identify the inner and outer semantics with the information extracted by an interpreter and a compiler, mentioned in 1.1, but, as described in 1.3, the level of the interpretive code can be varied and different amounts of information can be extracted by each component of the system. In a system where the compiler is concerned exclusively with extracting outer semantic information, however, a clean separation of concerns is permitted and experience shows that this provides a suitable framework for structuring system development. In addition, it raises the possibility of plugging different inner semantics into the same syntax and outer semantics, and using the same compiler as the basis for the implementation of a family of specialised languages.

CHAPTER 7

THEORY AND PRACTICE

7.1 SETL-S

The previous chapter showed that it is possible to describe an ITC-based interpretive system in a way which is related to the language being implemented. This suggests that an examination of some features of the semantics of Setl will provide an additional understanding of the workings of the Setl-s system. First, it is necessary to indicate briefly how some aspects of Setl which are not present in X10 can be accomodated.

The most serious shortcoming of X10 is its lack of any mechanism for defining and calling procedures; the main reason for their omission is that the issues of parameter passing and free variable binding require additional complications in the mathematical models, which are not directly relevant to the description of ITC. These issues are discussed at length in [MS76]. In standard semantics, the value of a procedure is a member of the domain $F = E^{*}\text{->}K\text{->}C$ (Setl procedures always return results). In an implementation, the expression continuation here passed as an argument corresponds roughly to the return link placed on the stack; the function itself can be represented by an object resembling a closure [STR67] consisting of a pointer to the code of the procedure, the bound variables and the environment (in Setl this last component is restricted to the global variables of the program). In ITC, the value part of a procedure block would be this closure, with the block action routine being a function to apply it to the arguments supplied by a call.

Since declarations do not play an important part in Setl they will not be considered.

Naturally, the most important aspect of the Setl language is its set-theoretic data types, and the associated iterative constructs. Because sets can be formed out of other expressions, it is not sufficient to treat them as bases to be evaluated by $\underline{B}$, as numerals for example might be. A domain of sets is required which will resemble theoretical sets: a predicate in will test for membership, and the operations of union, intersection and so on will be defined to have their normal meanings. Expressions such as {1, 3.142, 'hello'} suggest that sets can be formed from sequences of expressed values by a function formset, which will be in $K \to E^* \to C$ (thus modelling the evaluation of the complete list before the formation of the set); a similar function performs the analogous operation for tuples. The internal structure of sets need not be examined, but the fact that t(i) can appear on the left of an assignment means that the domain of tuples Tp must be equal to L^*, with a suitable function selecting the appropriate location. Notice, therefore, that to preserve the semantics of assignment the function rv must take the R-value of all the components of its argument if this is in L^*.

To accomodate expressions such as {x: x IN s| x>43}, a syntactic domain of formers Fmr is required, with a corresponding valuation $\underline{F}$. To allow for the possibility of errors occurring during the evaluation of a former a new sort of continuation, a former continuation, is required, which is similar to an expression continuation except that its first argument is the partial result of evaluating the former, so this domain is equal to E^*->C. The value of a former will be a sequence of expressed values, enabling one to write:

$$\underline{E}[\![\{F\}]\!] = \lambda\rho\gamma . \underline{F}[\![F]\!] (\mathrm{formset}_\gamma)$$

So far, this is straightforward and is obviously related to the way in which sets and tuples are formed by Setl-s. Difficulties arise however when the question of maps is considered. One would like to consider maps as members of E->L, but the language definition insists that maps are just special cases of sets. This means that functions operating on maps must first check that all members of their set operands are in Tp and that they are of length 2; in addition, to evaluate f(x) it is necessary to search the set for the pair whose first component's R-value is equal to x. This is the source of the practical difficulties discussed in 3.2.3. What the present discussion demonstrates is that these problems arise from the language design and not from the particular representations chosen for sets in Setl-s.

If it is desired to keep maps as sets, a mechanism is required for the treatment of some types as sub-types of others. Shamir and Wadge [SW77] have proposed such a mechanism, but it is not clear how this could be incorporated into the framework of standard semantics, nor how it could be mapped into an implementation.

Loops controlled by iterators such as (For x in s) can be described easily by providing a function to select and remove an arbitrary element from a set and a predicate to test whether a set is empty. In Setl-s these two have been combined into the JNEXT operation (see 3.4.1) but there is no particular virtue in this. The valuation F would make use of the valuations for loops to build its sequences. The arithmetic formers can be similarly defined in terms of an arithmetic loop. (This raises the interesting possibility of proving the legitimacy of the optimisation of arithmetic loops.) For both kinds of loop, a component in the environment is required to hold the temporary variables used. A further component will be required to hold the continuations to be applied when the commands Quit and Continue are obeyed.

A complete definition of the semantics of Setl would be extremely lengthy and uninteresting, so none will be presented.

7.2 APPLICATIONS OF THE THEORY

It might be argued, quite rightly I think, that analysing an implementation in terms of the semantics of the implemented language, after the system has been built is doing things the wrong way round. Unfortunately, this situation, resulting from the circumstances of the project's inception, is typical of the state of affairs in this particular branch of software engineering at present, and indicates the prevalent lack of understanding of semantic matters. This is in marked contrast to the subject of syntax, which is thoroughly understood. As the Setl-s compiler illustrates, this understanding is sufficiently complete to form the basis of an automatic parser generating system.

Although there are still some unsatisfactory aspects to the theory, denotational semantics has been shown to be a powerful tool for describing the semantics of programming languages, and has been successfully applied to languages as diverse as Snobol4[TEN73], Gedanken[TEN76], Algol60[MOS74a], a superset of Pal and some features of Algol68[MIL74]; in addition, in [MIL74,MS76] it has been shown how the method can be used to describe features such as co-routines and parallel processing, type checking and coercion as well as methods of language implementation. All these applications are firmly based on the theoretical work originating with Scott, which both provides a clear understanding of the properties of semantic valuations and connects them with computability theory in a way which ensures that they provide a reasonable model of computation.

One particularly useful property of denotational semantic descriptions is that obscure or undesirable language features require lengthy and convoluted valuations; because of the connection between the semantics and their implementation the semantic description will also show up language features which are difficult to implement, as the brief discussion of Setl-s in the preceding section will hopefully have illustrated. It is therefore to be hoped that, in the future, formal semantic definitions will play a more prominent role in language design and implementation.

The connection between standard semantics and indirect threaded code outlined in chapter 6 raises the possibility of a more direct use of the semantic descriptions in the building of interpreters. Mosses [MOS74,MOS76] has shown how to formalise the meta-language used to define valuations so that they can be viewed as mappings converting programs written in the defined language into programs written in the defining language. By combining the the valuations with an implementation of the meta-language an implementation of the defined language can be produced. Pagan [PAG79] advocates the use of a conventional high level language as meta-language, which has an obvious appeal, but does not guarantee the correctness of the implementation in the way which a language such as Mosses' MSL, which can be defined in terms of Lambda [SCO76] does. Such implementations may be expected to be very inefficient; this inefficiency can be overcome to a certain extent by the use of optimisation techniques, such as recursion removal, in the implementation of the meta-language.

It has been shown that a relationship exists between the standard semantics of a language and an ITC-based semantics. Since the latter only involves semantic information, it can, in principle at least, be derived from the standard semantics and an account of the domains relevant to the two sorts of semantics. By concentrating on what I have called the outer semantics, a standard semantics could be transformed into an ITC-based semantics which, when implemented, would provide a code generator, producing codewords from programs written in the source language. This could then be combined with a set of interpretive routines implementing the inner semantics to provide an interpreter. To produce a complete implementation, a compiler would be required, since the abstract syntax which is used in the semantic descriptions neglects many syntactic features and can best be regarded as a description of the parse trees produced by a compiler. Efficiency considerations would seem to indicate that the interpretive routines would have to be hand written.

The ideas sketched in the preceding paragraphs adumbrate a whole new research project; the practical difficulties encountered in the building of Sell-s give some indication of the range of problems such a project must attempt to deal with.

Appendix 1: Sample Setl-s Listings

```
SETL-S VERSION 1.6(1) - Leeds University
DSK:PRIMES.STL[10177,13]   21-Sep-1980   19:31:21


                                                                  PAGE 1

1     1      PROG prime ;
2     1          primes := {} ;
3     1          (FOR p IN [2...1000])
4     1              IF NOTEXISTS pp IN primes | p REM pp = 0
5     1                 THEN primes WITH:= p ;
6     3                      IF p >= 990 THEN PRINT(p); END ;
7     4              END IF ;
8     6            END ;
9     6      END PROG ;


STORE USED      11895
STORE LEFT      3352
COMP ERRORS     0
REGENERATIONS   1
COMP TIME-MSEC  440


991
997


NORMAL END
IN STATEMENT    6
RUN TIME-MSEC   7260
STMTS EXECUTED  22873
MCSEC / STMT    317
REGENERATIONS   2
```

```
SETL-S VERSION 1.6..) - Leeds University
DSK:HEAPS.STL[1017~ 13]   21-Sep-1980   19:35:04

                                                        PAGE 1

PROGRAM heapsort_test ;
PROC heap_sort ;

CONST
    seqlen = 50, nstodo = 10 ;
VAR
    testseq, sortseq, timeon ;

    testseq := [ 01, 78, 56, 23, 17, 88, 05, 85, 65, 43,
                 43, 32, 78, 90, 31, 16, 10, 54, 99, 32,
                 38, 55, 99, 02, 25, 07, 54, 88, 77, 66,
                 55, 44, 57, 78, 83, 06, 16, 12, 18, 92,
                 93, 54, 33, 10, 19, 20, 21, 23, 13, 10 ] ;
    PRINT('start of heapsort test')  ;  PRINT ;
    timeon := TIME ;

    (FOR i IN [1...nstodo])
        sortseq := heap_sort(testseq, 1, seqlen) ;
    END ;

    PRINT ;  PRINT ;

    PRINT('sorted ', seqlen, 'items ',
           nstodo, 'times in ', TIME-timeon, 'ms') ;
    PRINT('unsorted sequence = ', testseq) ;
    PRINT('sorted sequence   = ', sortseq) ;
    PRINT ;
    PRINT('end of heap sort test') ;

 PROC heap_sort(tseq, lo, hi) ;
    seq := tseq ;
    (FOR i IN [lo+1...hi])
        LOOP
        INIT m := i ;
        WHILE
            m > lo /\ seq(m DIV 2) < seq(m)
        DO
            mm := m DIV 2 ;
            temp := seq(m)  ;
            seq(m) := seq(mm) ; seq(mm) := temp ;
            m := mm ;
        END LOOP ;
    END ;

    (FOR seqtop IN [hi, hi-1...lo+1])
        temp := seq(lo)  ;
        seq(lo) := seq(seqtop)  ;  seq(seqtop):= temp ;
        LOOP
        INIT m := lo ;
        DOING
```

```
             IF (m*2+1) < seqtop /\ seq(m*2) < seq(m*2 + 1)
                       THEN m*2 + 1
                       ELSE m*2
                    END ;
        WHILE
            (m * 2) < seqtop /\ seq(m) < seq(targ)
        DO
            temp := seq(m)  ;
            seq(m) := seq(targ)  ;  seq(targ) := temp ;
            m := targ ;
        END LOOP ;
    END ;

    RETURN seq ;

END PROC ;

END PROG ;
```

```
STORE USED     13755
STORE LEFT     1492
COMP ERRORS    0
REGENERATIONS  1
COMP TIME-MSEC 1620



start of heapsort test



sorted 50items 10times in 6140ms
unsorted sequence = [1 , 78 , 56 , 23 , 17 , 88 , 5 , 85 , 65 , 43 , 43
, 32 , 78 , 90 , 31 , 16 , 10 , 54 , 99 , 32 , 38 , 55 , 99 , 2 , 25 , 7
 , 54 , 88 , 77 , 66 , 55 , 44 , 57 , 78 , 83 , 6 , 16 , 12 , 18 , 92 ,
93 , 54 , 33 , 10 , 19 , 20 , 21 , 23 , 13 , 10]
sorted sequence   = [1 , 2 , 5 , 6 , 7 , 10 , 10 , 10 , 12 , 13 , 16 , 1
6 , 17 , 18 , 19 , 20 , 21 , 23 , 23 , 25 , 31 , 32 , 32 , 33 , 38 , 43
, 43 , 44 , 54 , 54 , 54 , 55 , 55 , 56 , 57 , 65 , 66 , 77 , 78 , 78 ,
78 , 83 , 85 , 88 , 88 , 90 , 92 , 93 , 99 , 99]

end of heap sort test



NORMAL END
IN STATEMENT   13
RUN TIME-MSEC  6520
STMTS EXECUTED 33354
MCSEC / STMT   195
REGENERATIONS  2
```

```
SETL-S VERSION 1.6(1) - Leeds University
DSK:MEDIAN.STL[10177,13]   21-Sep-1980   19:35:48

                                                         PAGE 1

PROGRAM medianfinder ;
      PROC kthone, min2, max2 ;

$ Based on the median finder from the NYU Setl test library,
$ coded by Dave Shields from a SETLA program by H. Warren.

INIT kthonebln := 0 ;
PRINT('median test') ;
cases := [3, 20, 50] ;

(FOR i IN [1...£cases])
   tim := TIME ;
   testset := {1...cases(i)} ;
   PRINT ; PRINT ;
   PRINT('case number ', i, ' test set is:') ;
   PRINT(testset) ;
   median := kthone((£testset+1) DIV 2, testset) ;
   PRINT('the median of the test set is ', median) ;
   PRINT('time taken = ', TIME - tim, ' ms') ;
END ;

PROC kthone(kparam, setparam) ;
   IF setparam = {} THEN RETURN OM ; END IF ;
   k := kparam ;  sett := setparam ;
   kthonebln +:= 1 ;  kthonebl := '' ;
   (FOR i IN [0...kthonebln]) kthonebl +:= '   ' ; END ;

   LOOP
      WHILE £sett >= 3
   DO
      i := 2 ;
      midpts := {} ;
      (FOR x IN sett)
         i := (i + 1) REM 3 ;
         IF i = 0 THEN u := x ;
         ELSEIF i = 1 THEN v := x ;
         ELSEIF i = 2
         THEN
            IF x < v THEN cas := 1 ;
                     ELSE cas := 0 ; END ;
            IF u < x THEN cas +:= 2 ; END ;
            IF v < u THEN cas := 3 - cas ; END ;
            midpts WITH:= [u, v, x](cas) ;
         END IF ;
      END ;

      PRINT(kthonebl, (£sett DIV 3) * 3) ;

      median := kthone((£midpts+1) DIV 2, midpts) ;
```

```
        (FOR x IN sett)
            IF x <= median
            THEN smallpile WITH:= x ;
            ELSE bigpile WITH:= x ;
            END IF ;
        END ;

        PRINT(kthonebl, £sett) ;

        IF k <= £smallpile
        THEN sett := smallpile ;
        ELSE
            sett := bigpile ;
            k -:= £smallpile ;
        END IF ;

    END LOOP ;

    kthonebln := IF kthonebln > 0
                 THEN kthonebln - 1 ELSE 0 END ;
    IF £sett = 1
    THEN
        IF k = 1 THEN RETURN ARB sett ;
                 ELSE RETURN OM ;
        END ;
    ELSE
        IF k = 1
        THEN RETURN min2(sett) ;
        ELSEIF k = 2
        THEN RETURN max2(sett) ;
        ELSE RETURN OM ;
        END IF ;
    END IF ;

END PROC ;

PROC min2(s) ;
    ss := s ;
    p FROM ss ; q FROM ss ;
    RETURN IF p < q THEN p ELSE q END ;
END ;

PROC max2(s) ;
    ss := s ;
    p FROM ss ; q FROM ss ;
    RETURN IF p > q THEN p ELSE q END ;
END ;

END PROG ;
```

```
STORE USED      14988
STORE LEFT      259
COMP ERRORS     0
REGENERATIONS   1
COMP TIME-MSEC  1820
```

```
median test


case number 1 test set is:
{1 , 2 , 3}
     3
     3
the median of the test set is 2
time taken = 60 ms


case number 2 test set is:
{1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 , 16 ,
17 , 18 , 19 , 20}
     18
        6
        6
        3
        3
     19
     9
 X9 , 20 , 21 , 22 , 23 , 24 , 25 , 26 , 27 , 28 , 29 , 30 , 31
 , 32 , 33 , 34 , 35 , 36 , 37 , 38 , 39 , 40 , 41 , 42 , 43 , 44 , 45 ,
 46 , 47 , 48 , 49 , 50}
     48
        15
           3
           5
           3
           3
        16
        6
        8
        3
        4
     50
     27
        9
           3
           3
        9
        3
        5
```

```
      27
      12
         3
         4
      13
      3
      5
the median of the test set is 25
time taken = 740 ms



NORMAL END
IN STATEMENT   12
RUN TIME-MSEC  1140
STMTS EXECUTED 3991
MCSEC / STMT   285
REGENERATIONS  0
```

Appendix 2: An SLR(1) Grammar for X10

```
<p>                   =
                      % <program> @                              ;    *0*
<program>             =
                      <block>                                    ;    *1*
<block>               =
                      $( <statements> $)                         |    *2*
                      <statement>                                ;    *3*
<statements>          =
                      <statements> <statement>                   |    *4*
                      <statement>                                ;    *5*
<statement>           =
                      <assignment>                               |    *6*
                      <ifset>                                    |    *7*
                      <while>                                    ;    *8*

<assignment>          =
                      $name $assop <exp>                         ;    *9*
<ifset>               =
                      IF $( <choices> $)                         ;    *10*
<choices>             =
                      <choices> <choice>                         |    *11*
                      <choice>                                   ;    *12*
<choice>              =
                      ( <exp> ) -> <block>                       ;    *13*
<while>               =
                      WHILE <exp> DO <block>                     ;    *14*

<exp>                 =
                      <exp> $op0 <exp1>                          |    *15*
                      <exp1>                                     ;    *16*
<exp1>                =
                      <exp1> $op1 <exp2>                         |    *17*
                      <exp2>                                     ;    *18*
<exp2>                =
                      <exp2> $op2 <exp3>                         |    *19*
                      <exp3>                                     ;    *20*
<exp3>                =
                      <exp3> $op3 <exp4>                         |    *21*
                      <exp4>                                     ;    *22*
<exp4>                =
                      $op2 <bop>                                 |    *23*
                      $uop <bop>                                 |    *24*
                      <bop>                                      ;    *25*
<bop>                 =
                      $name                                      |    *26*
                      <exp> )                                    |    *27*
                      $number                                    |    *28*
                      $string                                    |    *29*
                      TRUE                                       |    *30*
                      FALSE                                      ;    *31*
```

Bibliography

[AHU74] Aho, A.V., Hopcroft, J.E. and Ullman, J.D., The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

[AJ74] Aho, A.V. and Johnson, S.C., 'LR Parsing' in ACM Computing Surveys, vi, no.2, June 1974, pp.99-124.

[AU77] Aho, A.V. and Ullman, J.D., Principles of Compiler Design, Addison-Wesley, 1977.

[BEL73] Bell, J.D., 'Threaded Code' in Communications of the ACM, xvi, no.6, June 1973, pp.370-372.

[BRO76] Brown, P.J., 'Throw-away Compiling' in Software-Practice and Experience, vi, no.3, 1976, pp.423-435.

[BRO79] Brown, P.J., Writing Interactive Compilers and Interpreters, Wiley, 1979.

[CM79] Chapman,N. and McCann, A.P., Setl-s, a Proposed Implementation of the Programming Language Setl, University of Leeds Department of Computer Studies Technical Report no.124, Leeds, 1979.

[DER69] DeRemer, F.L., Practical Translators for LR(k) Languages, Ph.D. Thesis, MIT, 1969

[DER71] DeRemer, F.L., 'Simple LR(k) Grammars' in Communications of the ACM, xiv, no 7, July 1971, pp.453-460

[DP79] DeRemer, F.L., and Pennello, T.J., 'Efficient Computation of LALR(1) Lookahead Sets' in Proceedings of the Sigplan Symposium on Compiler Construction, August 1979, pp.176-187

[DEW78] Dewar, R.B.K., et al., 'Setl as a Tool for Generation of Quality Software' in Hibberd and Schumann, Constructing Quality Software, North-Holland, 1978, pp.353-366

[DEW79] Dewar, R.B.K., The Setl Programming Language, Manuscript,

[DM77] Dewar, R.B.K. and McCann, A.P., 'Macro Spitbol - a Snobol4 Compiler' in Software - Practice and Experience, vii, pp.95-113

[GHJ79] Graham, S.L., Haley, C.B. and Joy, W.N., 'Practical LR Error Recovery' in Proceedings of the Sigplan Symposium on Compiler Construction, August 1979, pp.168-175

[HOP69] Hopgood, F.R.A., Compiling Techniques, McDonald, 1969

[HOR76] Horning, J.J., 'LR Grammars and Analysers' in Bauer and Eickel (eds) Compiler Construction - an Advanced Course, Springer-Verlag, 1976, pp.85-108

[JOH78] Johnson, S.C., Yacc: Yet Another Compiler-Compiler, Bell Laboratories Computing Science Technical Report no 32, Bell Laboratories, 1978

[LAN64] Landin, P.J., 'The Mechanical Evaluation of Expressions' in Computer Journal, vi, 1964, pp.308-320

[LS68] Lewis, P.M. and Stearns, R.E., 'Syntax-directed Transduction' in Journal of the ACM, xv, 1968, pp.465-488

[LW71] Lucas, P. and Walk, K., 'On the Formal Description of PL/I' in Annual Review in Automatic Programming, no 6, 1971, pp.105-182

[MHD76] McCann, A.P., Holden, S.C., and Dewar, R.B.K., Macro Spitbol - DECSystem 10 Version, University of Leeds Department of Computer Studies Technical Report no 94, 1976

[McC66] McCarthy, J., 'A Formal Description of a Subset of Algol' in T.B. Steel (ed), Formal Language Description Languages for Computer Programming, 1966, pp.1 - 12

[MIL74] Milne, R.E., The Formal Semantics of Computer Languages and Their Implementations, Ph.D. Thesis, University of

[MS76] Milne, R.E., and Strachey, C., _A Theory of Programming Language Semantics_, Chapman and Hall, 1976

[MOS74] Mosses, P.D., 'The Semantics of Semantic Equations' in _Mathematical Foundations of Computer Science 1974_, Springer-Verlag, 1974, pp.409-422

[MOS74a] Mosses, P.D., _The Mathematical Semantics of Algol60_, Technical Monograph PRG-12, Oxford University Computing Laboratory, 1974

[MOS76] 'Compiler Generation Using Denotational Semantics' in _Mathematical Foundations of Computer Science 1976_, Springer-Verlag, 1976, pp.436-441

[NW74] Needham, R.M., and Wilkes, M.V., 'Domains of Protection and the Management of Processes' in _Computer Journal_, xvii, no 2, 1974, pp.117-120

[NEE76] Neeley, P.M., 'A New Programming Discipline' in _Software - Practice and Experience_, vi, no 1, 1976, pp.7-27

[PAG79] Pagan, F.G., 'Algol68 as a Metalanguage for Denotational Semantics' in _Computer Journal_, xxii, no 1, 1979, pp.63-66

[POO78] Poole, P.C., 'Towards Improved Reliability and Efficiency Through Hybrids' in Hibberd and Schuman (eds) _Constructing Quality Software_, North-Holland, 1978, pp.63-74

[REY72] 'Definitional Interpreters for Higher Order Programming Languages' in _Proceedings of the 25th ACM National Conference_, 1972, pp.717-740

[RIC72] Richards, M., _Intcode - An Interpretive Machine Code for BCPL_, Cambridge University Computing Laboratory, 1972

[SCO76] Scott, D.S., 'Data Types as Lattices' in _SIAM Journal on Computing_, v, 1976, pp.522-587

[SCH76] Schwartz, J.T., On Programming - An Interim Report on the Setl Project, New York University, 1976

[SS71] Scott, D., and Strachey, C., Toward a Mathematical Semantics for Computer Languages, Technical Monograph PRG-6, Oxford University Computing Laboratory, 1971

[SW77] Shamir, A., and Wadge, W.W., 'Data Types as Objects' in Automata, Languages and Programming, 4th Colloquium, Springer-Verlag, 1977, pp.465-479

[STR67] Strachey, C., Fundamental Concepts in Programming Languages, International Summer School in Computer Programming, 1967, Unpublished Manuscript

[STR73] Strachey, C., The Varieties of Programming Language, Technical Monograph PRG-10, Oxford University Computing Laboratory, 1973

[SW74] Strachey, C., and Wadsworth, C.P., Continuations - a Mathematical Semantics for Handling Full Jumps, Technical Monograph PRG-11, Oxford University Computing Laboratory, 1974

[TEN73] Tennent, R.D., 'Mathematical Semantics of Snobol4' in Conference Record of the ACM Symposium on Principles of Programming Languages, 1973

[TEN76] Tennent, R.D., 'The Denotational Semantics of Programming Languages' in Communications of the ACM, xix, no 8, 1976, pp.437-453

[WIL68] Wilkes, M.V., 'The Inner and Outer Syntax of Programing Languages' in Computer Journal, xi, 1968, pp.260-263