AN EXPRESSIVE LANGUAGE

by

PAUL WILLIAM WHALEY

B.Sc., University of British Columbia

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the Department

of

Computer Science

We accept this thesis as conforming to the

required standard

The University of British Columbia

March, 1975
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver 8, Canada

Date April 8, 1975
Abstract

A programming language system which enables a programmer to create his own constructs for the description of an algorithm is presented. The syntactic and semantic design and some details of the implementation are discussed and the language itself is described. For the introduction of new constructs by the programmer, the language system (called "Tove") provides a parsing language for syntactic description and a set of basic procedures for semantic description. The combination of these two enable the programmer to define new procedures, control structures, and operators. Finally, some problems in the design and implementation are discussed, together with some thoughts about future work on such systems.
# TABLE OF CONTENTS

## CHAPTER 1. Introduction ................................................................. 1
  1.1. Syntactic Design ......................................................................... 1
     1.1.1. Syntax Of Tove ......................................................... 3
     1.1.2. Traditional Syntactic Methods .................................. 4
        1.1.2.1. FORTRAN ....................................................... 5
        1.1.2.2. Algol60 .......................................................... 7
        1.1.2.3. LISP .............................................................. 7
        1.1.2.4. Algol68 .......................................................... 8
        1.1.2.5. Pascal And Sue ............................................. 9
  1.2. Semantic Design .......................................................................... 9
     1.2.1. Procedural Organization ........................................... 10
     1.2.2. Scope Conventions ................................................... 11
     1.2.3. Values And Data Types ........................................... 13
     1.2.4. Control Structures .................................................. 14

## CHAPTER 2. User's Guide ................................................................. 15
  2.1. Overview Of Tove ...................................................................... 15
  2.2. The Scanner ............................................................................... 17
  2.3. The Parsing Language ............................................................. 19
  2.4. The Parsing Of Expressions .................................................... 26
  2.5. Basic Procedures ....................................................................... 28
     2.5.1. Arithmetic Procedures ........................................... 28
     2.5.2. Logical Procedures .................................................. 29
     2.5.3. String Procedures ................................................... 29
     2.5.4. Cell Procedures ....................................................... 31
Acknowledgements

Several people have contributed towards the design and implementation of this system and towards the preparation of this thesis. Specifically, my supervisors Alan Ballard and Ray Reiter must be thanked for moral support and for identification of problems and suggestions for improvements in both the system and the thesis. Alan Ballard and Mark DuMont also have worked very hard improving the SUE System Language used in the implementation, and also have provided many engaging discussions, some of which were relevant to the work presented here. I would also like to thank my parents for their support and encouragement and the National Research Council of Canada for their financial support.
CHAPTER ONE

INTRODUCTION

This chapter presents the motivation and design of the syntax and semantics of the Tove system, and some of the previous work it is based on.

1.1. SYNTACTIC DESIGN

The purpose of any syntax in a programming language (or in any language) is to provide a framework for communication between the writer and the readers of the language. A programming language must satisfy the needs of three groups: the programmers writing in the language, the people reading the programs and the compilers reading and translating the language. That programmers have to be able to read the language is becoming more appreciated, especially as the size of programs grow. A very large program cannot be contemplated as a single whole in all its intricacy, so that a programmer attempting to modify such a large program must be able to read and comprehend parts of it, as it is expressed in the programming language. Even during the initial development of a program modifications are made to it, so all successful programmers must be able to read and comprehend both their own and other programmers' programs.

The necessity for programs to be read and understood by compilers (or interpreters) restrains programming languages,
both syntactically and semantically. Specifically, very flexible and complex syntax in a language will force the compiler to be large and probably also slow. Thus the economic limitations on the compiler for a programming language force the syntax of the language to be vastly simpler than the syntax of a natural language. Since the writer, readers, and compiler of a language must be able to decide and agree exactly what the program will do, a programming language must be much more precise than a natural language. This also necessitates a simpler, less ambiguous syntax than that of a natural language.

Viewed in this manner the problem of designing a syntax for a programming language involves a trade-off between the comfort of the three groups who make demands on the syntax of a language. When the programmer is writing in the language he would like a concise, short syntax rather than a verbose and redundant one which will cause him to do a lot more writing or say everything twice. The programmer reading the language wants a clear syntax with enough redundancy so that he will be able to understand the program with less effort. The compiler's needs are a formal and precise syntax that is easily analysed by a parser which is small and fast.

1.1. SYNTACTIC DESIGN
1.1.1. Syntax Of Tove

The major goal of the syntactic design of Tove is the provision of facilities which will produce a more readable and writable language. The facilities are provided by a scheme which allows programmer specification of the syntax of his program. He is able to choose a natural syntax which he and (hopefully) other readers will be able to understand. Like any other freedom, if this syntactic freedom is not accepted responsibly then chaos can result. Specifically it is possible, and indeed easier, to write unreadable programs. Thus the writer of a Tove program must bear in mind the responsibility he has to readers of the program, including himself. The scheme is sufficiently restricted so that the compiler will be able to parse programs without undue difficulty.

The syntactic specification scheme consists of a parsing language, which is used by the programmer to describe the syntax he wishes to use to the system and to readers of his programs. This parsing language is converted to an internal form which is interpreted by the parser, causing the parser to parse the programmer's constructs. The actual parsing method used is a recursive descent parser patterned after the top down operator precedence methods of V. R. Pratt. [Prat] Included in the parsing language are the capabilities to specify that a procedure has an arbitrary number of arguments, that some arguments may be omitted, and that particular delimiters (such as "else") are to be used between arguments. The ability to specify priorities of operators has not been included, although
it is believed that it could have been without major changes. The exclusion is due to the lack of time for implementation and also because there are few operators that have standard, well-agreed priorities.

1.1.2. Traditional Syntactic Methods

The design of programming languages has been much studied in the past twenty years and many techniques have been developed to reconcile the conflicting demands. Often methods have been developed to allow two of the three groups to be accommodated at a small expense to the third. A good example of this is the introduction of paragraphers. A paragrapher is a part of the software system supporting the programming language which produces a listing of the program with a standard indentation and format. This provides very good redundancy for a reader of the program since he can rely on the visual cues of the indenting to guide him in understanding the program. It also does not cost the writer any effort. In fact since paragraphing is of such importance to readability that it must be done, the provision of automatic paragraphers frees the writer from this task. Like everything in life it is not without its cost, and in this case the software system grows larger and a bit slower. The saving in effort to the writer and readers, who are people, now almost certainly outways the costs of an automatic paragrapher so that any complete programming system should now provide automatic paragraphing.

The inclusion of a comment facility in a programming
language provides a medium for the writer to speak directly to his readers in natural language, without also speaking to the compiler. As such, comment facilities are very useful and it is inconceivable that any modern programming language should be without a good comment facility. However, if the language is such that it cannot be understood by a reader without extensive comments, then the writer will be forced to write the program twice, once for the compiler and again, in the comments, for his readers. Thus comments are necessary for a readable language, but they are not sufficient in themselves to make a language readable.

1.1.2.1. FORTRAN

Early in the history of language design, the balance lay to the side of making the compilers small and fast at some expense to both writers and readers of the languages. This was because programs were small, both machine time and space were expensive, few of the current techniques used for constructing compilers were known, programming costs were low, and the increase in programmer productivity which could be gained by the use of more convenient languages for the programmers was greatly underestimated. These factors lead to the development of assemblers and languages like FORTRAN. [FORT]

The FORTRAN programming language uses a card-oriented syntax. It is unusual in that it ignores all blanks in statements. Each input card is divided into three fields: columns 1 to 5 for statement numbers, column 6 for indication of
continuation cards, and columns 7 through 72 for the statements. Statements are recognized by an add-hoc parser. The language is not suited to parsing by any other methods since the lack of keywords can require a great deal of look-ahead before it can be determined whether a symbol is used as a variable or as a pseudo-keyword. For example, it is possible to have an array called FORMAT or IF. The syntax contains several anomalies such as the impossibility of a compiler detecting a missing comma in a DO statement. Since variables can be used without declaring them, a misspelt variable results in the allocation of a new variable, not in a complaint from the compiler. The language seems not to have enough redundancy to enable the compiler to detect common writing errors. Further to these problems, the restriction of labels to only numbers, the restriction of the length of identifiers to only six characters, and the lack of reasonable control structures lead very quickly to unreadable programs.

The major advances of FORTRAN over assemblers are the introductions of infix notation for the built-in arithmetic operators and subroutines. The infix notation gives FORTRAN its name, which stands for FORMula TRANslator. Functions and subroutines which are defined by the user are called by a prefix notation consisting of the name identifying the procedure, followed by the parameters of the procedure enclosed in parentheses.

1.1.2.1. FORTRAN
1.1.2.2. Algol60

With the development of Algol60 [AL60], several techniques for efficiently implementing languages which are more programmer-oriented were introduced and publicized. These techniques include the formal syntactic descriptions and their subsequent incorporation into compiler generators.

Algol60 represents a great syntactic advance over languages such as FORTRAN. The syntax of Algol60 is described in a uniform, formalized manner and so the language is suited to machine generation of parsers. This leads immediately to the lack of serious anomalies, with the exception of the "dangling else" problem. Keywords are recognized by typographic conventions and so cannot be confused with identifiers. Labels can be character identifiers and identifiers are no longer restricted to short lengths. The built-in operations have reasonable syntax, including infix operators and some reasonable control structures. Procedures defined by the programmer are called with the same style of syntax as that used by FORTRAN.

1.1.2.3. LISP

LISP [LISP] is syntactically very different from either FORTRAN or ALGOL60 since it has no infix syntax. All constructions are represented by prefix syntax which consists of a left parenthesis, the atom identifying the procedure to be called, and the arguments of the procedure enclosed by a right parenthesis. This syntactic convention is very simple, easily parsed, and can be used for both programs and data since it
corresponds closely with the internal representation that it is
translated into. Its main drawbacks are its many parentheses
and its cumbersomeness when unary or binary procedures are used
extensively (as in arithmetic calculations). Unless a LISP
program is paragrapged by machine and carefully read, a
misplaced parenthesis can be difficult to detect, sometimes not
showing up until the program is very thoroughly tested. Despite
these problems the syntax of LISP remains elegant, simple, and
uniform.

1.1.2.4. Algol68

With the large decreases in computing machine costs and the
simultaneous rises in programmer costs, recent languages have
been designed even more for ease of writing and reading and less
for ease of compilation. Algol68 [AL68] is an example of such a
language. It is to be hoped that these newer languages are much
easier to read and write while only a little more costly to
compile.

The syntax of Algol68 is patterned closely after that of
Algol60, in that it has a formal description and uses
typographic conventions to distinguish programmer-defined
identifiers from the terminals of its syntax. Algol68, however,
provides an operator declaration mechanism whereby the user can
define infix procedures, in addition to the usual prefix
procedures.

Algol68 is certainly a very complex language to parse.
Ramer [Rame] describes the essential technique used to process
such syntactic constructions as operator declarations. Before the Algol68 program can be completely parsed, the compiler must first scan the entire text of the program and identify and process all operator declarations and all occurrences of the declared operators. Once this has been done, the text can be parsed by formal syntactic methods. The language requires a multiple-pass parser because the declaration of an operator may follow its use in the program text.

1.1.2.5. Pascal And Sue

Pascal [Pasc] and especially its derivative, the Sue System Language [SUE] are good examples of recent languages which were designed specifically for ease of reading. They are very easy to read and understand and they are also fairly easy to compile. However they are relatively difficult to write, chiefly because of the strong compile-time type-checking which requires large amounts of information in the specification of types.

1.2. SEMANTIC DESIGN

Semantically the Tove language is very unsophisticated, and most closely resembles LISP. The major goal of the semantic design was simplicity, both in the facilities provided and in their implementation. Much of this simplicity is a necessary result of the limited time available for implementation. Tove has only the very simplest of data types and only elementary operations on them. The structures it has are very similar to LISP CONS cells and atoms. Its control structures are again
simple but the unrestrained jump ("go to") is not included. These control structures are patterned largely after those of the Sue System Language, which is itself patterned after Pascal. The scope conventions of Tove and the resulting run-time environmental mechanisms are in large part borrowed from BCPL. [BCPL]

1.2.1. **Procedural Organization**

Included in the language is the ability of a programmer to define control procedures which permit the control of the evaluation of their arguments. In essence, the programmer can write his own control structures. This is another feature of the language that the program writer must take care with lest he render the program incomprehensible. This inclusion enables the language to be completely procedured, meaning that any program in the language is just a collection of procedure calls, some being calls to pre-defined basic procedures and others to procedures defined in the program. Thus the semantics of the language can be described by the semantics of these basic procedures and procedure invocation and return. This is essentially the organisation of languages such as LISP. The language is also like LISP in its expression orientation, that is every construct is an expression and returns a value.
1.2.2. Scope Conventions

The scope conventions of a language determine the modularity of procedures written in that language. With good scope rules a procedure will only have access to a small set of variables and the programmer will not have to worry about other uses of a variable name outside that set of variables. Scope is such an important aid to modularity, and therefore also to readability and writability, that its inclusion was considered essential. However it was decided that the overhead of maintaining complete environment chains or displays as in the usual Algol implementations [ALim] should be avoided by using the BCPL approach. Accordingly there are two kinds of variables: permanent and transient. Permanent variables are statically allocated and have the usual Algol scope rules (but of course not the usual lifetimes). Procedures behave as permanent variables and therefore can be declared inside other procedures, giving nested environments as in Algol. Transient variables are allocated on the same stack that is used to maintain procedure invocation information (so that recursion is possible). They are therefore allocated dynamically at procedure invocation time. They can be referenced only inside the body of the procedure that they are declared in. Parameter variables are transient variables. The important simplification is the limitation of the references to transient variables which means that the environment of a procedure is only its own stack frame, which is established whenever the procedure is called.

Related to scope is the question of whether the language is
compiled or interpreted. The correct managing of references to variables requires that the internal representation of the program contains references to the variables themselves, and not to the names of the variables, since a name may represent different variables in different parts of the program. Thus the identifying of distinct variables and references to them must be done during the process of converting the program to an internal representation which can be interpreted.

Traditionally a language processor has been called a compiler if it produced machine code, that is, if it produced an internal representation of the program which could be executed by a machine. Recent usage has extended the term compilation to include translation into a representation which can be executed by a hypothetical machine which is implemented in software. The Tove translator produces an internal representation which is tree-structured and so is unlike the common machine instruction stream. During translation it processes variable and procedure declarations and resolves the references to names of variables to particular variables. The internal representation is interpreted by a part of the language system. It is left to the reader to decide whether the Tove processor is an interpreter, a compiler, or both.

1.2.2. SCOPE CONVENTIONS
One of the important aspects of modern programming language development has been the development of data type schemes that allow a great deal of type checking to be done at compile time. It was felt that time did not permit the incorporation of such schemes into the language, so a run time type checking scheme has been used. Every value is represented at run time not only by an encoding of its value but also by an encoding of its type, so the appropriateness of operations on the value can be checked.

The simple data types managed by the language are numbers and strings. For simplicity of implementation, the data structuring methods of LISP were used to provide compound data types. There is one important difference between the LISP CONS cells and Tove cells: in LISP a cell consists of two pointers, each pointing to another cell or an atom, while in Tove a cell consists of a value (a number, string, cell or atom), and a pointer to another cell. Thus Tove has true number values and not numeric atoms as LISP. Likewise an atom (which is used synonymously with variable) has a value and not a pointer to a value.

One of the large advantages of using cell structures as data items is that they are also used for the internal representation of programs, therefore requiring only one set of management routines for both functions. The same is true of the character string variables, since their management routines are also used for the variable (atom) names.
1.2.4. **Control Structures**

The basic control structures provided by the language were chosen because of their modularity and ease of implementation. Included are the return from a procedure, the repeat or exit of a labelled block, the if-then-else, and the ability to step through a list of arguments. The replacement of the "goto" by the exit and repeat leads to a much simpler implementation, since the transfer involved either restarts or completes the evaluation of a block which must be active (currently being evaluated) at the time of the transfer. The ability to step through a list of arguments is provided since the syntax provides a means whereby a procedure may have an arbitrary number of arguments. The stepping is much like a LISP MAP function in that the same expression is evaluated for each part of the argument list.
CHAPTER TWO
USER'S GUIDE

This section contains a description of the language. It is hoped that the description presented is sufficiently detailed to enable the reader to use the language. A run of the Tove system is appended to this thesis, providing a large example of the use of the system.

2.1. OVERVIEW OF TOVE

Tove is an expression oriented and run time typed language. It is expression oriented in that every construct returns some value, much like LISP or ALGOL68. In fact, like LISP, every construct is a procedure call. In Tove the fundamental components are numbers, strings, atoms, cells, and procedures. A value is either some number, string, atom or cell, or the undefined value. Numbers and strings are simply that: a number is an integer (within the range -32768 to 32767 currently) and a string is a series of zero or more characters (currently up to a maximum of 255).

An atom is an entity which has some name and which has a value. The name is a symbol, as described in the section on the scanner, and is used to refer to the atom. Atoms are like the variables of most algorithmic languages. Atoms are created and given a name during compilation by declaring them. They must be declared textually before they are referred to. There are three
kinds of atoms: permanent, transient and procedure atoms. Transient atoms can be referred to only inside the procedure that they are declared in. They are allocated via a stack discipline so that they are undefined at the beginning of evaluation of their procedure and may have a different value for each active invocation of the procedure. Permanent atoms are allocated when they are declared so that they have only one value and retain it regardless of procedure calls. They can be referred to only inside the procedure in which they are declared and any procedures whose declarations are enclosed in that procedure, as in the usual ALGOL scope convention. There is a special permanent atom called the nil atom which indicates the absence of an atom. Procedure atoms are like permanent atoms in their scope convention, but they are distinguished by having a procedure body as a value. Note that two atoms may have the same name, but only if they are declared in different procedures.

In Tove there are two kinds of procedure: value procedures and control procedures. A value procedure evaluates its arguments before the evaluation of the procedure body whereas a control procedure evaluates its arguments during the evaluation of the procedure body, when the corresponding formal parameters are evaluated. In either case evaluation of arguments is done in the environment of the procedure call and never in the environment of the procedure body. Value procedures generally perform some computations on the values of their arguments and return some value. Control procedures are usually used to
govern the order of evaluation of their arguments. For example, among the basic procedures addition (+) is a value procedure while 'if' is a control procedure.

Procedures are created and given a name during compilation by declarations which describe the syntax of the procedure's call (invocation) and the semantics of the procedure's evaluation. The syntactic part is specified by a parsing language statement. These statements are discussed in the section on the parsing language. The semantic meaning of the procedure is an expression which is the body of the procedure.

A cell is an entity which is composed of two parts. The first part is a value; the second is another cell. This second part is often called the "next cell" of the cell. Cells are created during evaluation by a basic procedure and are reclaimed by the language system when they can no longer be accessed. There is a special cell called the nil cell which indicates the absence of a cell.

2.2. THE SCANNER

The scanner is that part of the input routines which divides the character stream input into tokens. A token is a symbol, a number, a string, or a comment. No token can extend over more than one line of the input stream. Numbers are a series of one or more digits. They are therefore always positive integers. A string is any series of characters excluding the quotation mark ("') surrounded by quotation marks. While inside a string, two quotation marks are interpreted by
the scanner as a single quotation mark, not the end of the string. If there is no closing quotation mark before the end of the input line, then an error is announced and the string is stopped. A comment is the percent character (%), not inside a string, followed by any characters up to the end of the input line containing the percent.

Number tokens and string tokens are converted to number values and string values while comment tokens are ignored.

A symbol can be either any alphabetic character followed by a series of zero or more alphabetic characters or digits, or it can be a series of one or more special characters. The alphabet is A through Z plus the underscore (_). A special character is any character which is not a digit, not in the alphabet, and not a quotation mark nor a blank.

For example:

**Numbers:**
1 0 31415

**Strings:**
"""" "A""""B" "ABNCN%J(<>%+"

**Symbols:**
a a6 a_b _6 +- ( ) : ->

**Comments:**
% everything past the % is a comment
% this is all comment *%!L plus junk

The scanner interprets:

\[(1+a_6)* (-3)||"boo""

as the tokens:

( symbol
1 number

2.2. THE SCANNER
2.3. THE PARSING LANGUAGE

The parsing language is that part of Tove which is used to present descriptions of syntax to the system. Each statement in the parsing language describes the syntax of a call of a procedure and the formal parameters of the procedure. The syntax of a parsing language statement is described in the following, using ALGOL68 syntactic notations. Any terminal written in capital letters indicates that the terminal must be that symbol (e.g. the terminal EXP must be the symbol EXP).

In this description of the parsing language the words "argument" and "parameter" are not used synonymously. An argument is a particular actual argument of a call of a procedure whereas a parameter is the formal parameter used in the definition of the procedure. The purpose of the syntax is to determine the association (or binding) between the actual arguments of a call and the formal parameters of the procedure called.
Parsing statement: left syntax, defined symbol, right syntax.

The defined symbol of the parsing statement is the name of the procedure whose syntax is being described.

Left syntax: EXPR, formal symbol ;.

The left syntax of a procedure describes the left argument of the procedure. If it is empty, then the procedure has no left argument. If it is non-empty then the left argument must be an expression and the procedure is either binary or postfix. The formal symbol becomes the name of a transient atom which is the formal parameter corresponding to the actual left argument of a call of the procedure.

For example:

- foo is a procedure with no parameters
  parsing statement: foo
  possible call: foo

- fooey is a procedure with one left parameter
  parsing statement: EXPR a fooey
  possible call: 3 fooey
  parameter a is associated with argument 3

Right syntax: right syntax element, right syntax ;.

The right syntax is a possibly empty series of right syntax elements. It describes the syntax of the right arguments of the procedure. The right arguments are a
concatenation of the arguments described by the right syntax elements, in the same order as these elements.

Right syntax element: `EXPR`, formal symbol;

The corresponding argument is an expression. The formal symbol becomes the name of a transient atom which is the formal parameter corresponding to the argument. For example:

Foo is a procedure with one right parameter parsing statement: `foo EXPR b`
possible call: `foo "boo"`
parameter b is associated with argument "boo"
Fun is a binary procedure parsing statement: `EXPR a fun EXPR b`
possible call: `7 fun "too"`
a is associated with 7 and b with "too"

Right syntax element: `PERMANENT_ATOM`, formal symbol;

The argument is a symbol which becomes the name of a permanent atom. The compilation of the procedure call causes the creation of the permanent atom. The formal symbol becomes the name of a transient atom which is the formal parameter corresponding to the argument. The formal parameter will have as value the permanent atom created. This syntax element is included mainly for the description of basic procedures.
For example:

parsing statement: foo PERMANENT_ATOM bar
possible call: foo x27

x27 is made the name of a permanent atom and bar is associated with this permanent atom

Right syntax element: TRANSIENT_ATOM, formal symbol;

This element is just like the PERMANENT_ATOM case, except that the argument becomes the name of a transient atom rather than a permanent one. Again, this syntax element is included mainly for the description of basic procedures.

For example:

parsing statement: fido TRANSIENT_ATOM bif
possible call: fido len

len is made the name of a transient atom and bif is associated with this permanent atom

Right syntax element: PROCEDURE_DEFINITION, formal symbol, stop symbol;

This syntax element is intended only for describing the basic procedures which are used to declare procedures, and should not normally be used by the user. It indicates that the argument will be a declaration of a procedure. The actual argument must be a parsing statement, followed by the symbol IS, then an expression and the stop symbol of

2.3. THE PARSING LANGUAGE
this syntax element. The parsing statement describes the syntax of a call of the procedure and the expression is the body of the procedure. The formal symbol becomes the name of a transient atom which is the formal parameter corresponding to the argument. This atom has as value the procedure atom created by the declaration.

For example:

parsing statement: proc PROCEDURE_DEFINITION proc_atom endproc
possible call: proc EXPR a foo IS a - 2 endproc

foo is made the name of a procedure atom, with syntax EXPR a foo and body a - 2. Proc_atom is associated with this procedure atom.

Right syntax element: SEXPR , formal symbol , stop symbol ;

This syntax element provides a convenient way of inputing particular cell structures. It should be noted that these structures may not correspond to valid internal representations of expressions and so generally the s-expression argument should not be evaluated. See the description of the period (.) quoting procedure.

The argument is an s-expression. An s-expression is a number, string, or symbol, or it is a series of zero or more s-expressions enclosed in parentheses. If an s-expression is a symbol which is the name of an atom then the s-expression is internally represented by that atom. If the symbol is not the name of an atom then it becomes
the name of a (newly-created) permanent atom. If the s-expression is a series of s-expressions then the s-expression is internally the cell list of those s-expressions. If the s-expression is entirely empty then it is internally the nil atom. The stop symbol is used to indicate the end of the s-expression.

Examples:

parsing statement: print_tree SEXPR tree endprint_tree
possible call: print_tree (a b (+ foo 19) "string") endprint_tree
Tree's value is a cell list of atom a, atom b, a cell list of atom +, atom foo, and number 19, and string "string".
possible call: print_tree endprint_tree
Tree's value is the nil atom.
possible call: print_tree ( ) endprint_tree
Tree's value is the nil cell.
possible call: print_tree boo endprint_tree
Tree's value is the atom boo.

Right syntax element: OPTIONAL, optional symbol, right syntax, ENDOPTIONAL;

This construction indicates that the argument may be omitted from a call of the procedure. If the optional symbol of the syntax element is present in a call then the arguments described in the right syntax enclosed by the ENDOPTIONAL must also be present. The omission of the
optional symbol indicates the omission of those arguments. Formal parameters of omitted arguments have as value the undefined value.

For example:

parsing statement: do EXPR index OPTIONAL by EXPR step_size ENDOPTIONAL EXPR loop enddo possible call: do i by 1 inside_loop enddo
Index is bound to i, step_size to 1, and loop to inside_loop.

possible call: do i inside_loop enddo
Index is bound to i, step_size to undefined, and loop to inside_loop.

Right syntax element : LIST, formal symbol, right syntax, ENDLIST, separator symbol, stop symbol;

This syntax element provides a method of defining a procedure which is to have an arbitrary number of arguments.

The argument corresponding to this syntax element is a list of one or more parts. The parts all have the same syntax and this syntax is described by the right syntax enclosed by the ENDLIST. The separator symbol must be between parts of the argument list. The stop symbol indicates the end of the list and so must be the last symbol of the argument. The formal symbol becomes the name of a transient atom which is the formal parameter corresponding to the list. The 'eachpart' routine, which
will be described in the semantic section, is used to associate each part of the argument list with the corresponding formal parameters of the right syntax enclosed by ENDLIST.

For example:

parsing statement: select LIST cases EXPR casepart ENDLIST, endselect
possible call: select case1, case2, case3 endselect
Cases is bound to a internal representation of the list case1, case2 and case3. Casepart is bound to the undefined value. See the description of the eachpart routine for a way to bind casepart to each of case1, case2 and case3.

Right syntax element : delimiter symbol .

The argument is the delimiter symbol.

2.4. THE PARSING OF EXPRESSIONS

Tove allows the user to define infix operators, so there must be some way of determining which operators have precedence. The problem can be stated as: given the phrase operator1 operand operator2, where operator1 has a right operand and operator2 has a left operand, how do we decide whether to associate the operand with operator1 or operator2? In other words, is the phrase grouped as (operator1 operand) operator2 or is it operator1 (operand operator2)? In such cases of conflict in

2.4. THE PARSING OF EXPRESSIONS
Tove the operand always associates with the left operator (the first grouping), assuming that the operand is as short a string of the input as possible. This leads to a left to right evaluation of expressions with no precedence. For example given a syntax of `EXPR a + EXPR b`, `EXPR c * EXPR d`, and `(EXPR e)` we have:

- `4 + 2 * 3` is the same as `(4 + 2) * 3` and evaluates to 18
- `4 * 2 + 3` is the same as `(4 * 2) + 3` and evaluates to 11
- `4 + (2 * 3)` evaluates to 10
- `4 * (2 + 3)` evaluates to 20

Note that when an expression is an interior argument, (i.e. the corresponding syntax element is not the first or last element in the parsing statement) then that argument is the longest valid expression that it can be. It will generally continue until a delimiter (such as "") or an atom which has no left argument. If an argument's corresponding syntax element is the last element of the parsing statement then the argument is as short as it can be. Thus in the first example the argument associated with `b` is only 2 and not `2 * 3` since this is a rightmost argument. In the third example the argument associated with `e` is `2 * 3` and not 2 since this argument's corresponding syntax element is not the last in the parsing statement (the syntax element ")" follows it). Note also that in this third example the argument associated with `b` is `(2 * 3)` since this is the shortest valid possibility.
2.5. BASIC PROCEDURES

The basic procedures described in this part of the user's guide are built-in procedures which provide the fundamental components of the semantics of the language. The meaning of any procedure is ultimately resolvable to some combination of these basic procedures. The description of the procedures is divided into sections based on a classification of the procedures.

2.5.1. Arithmetic Procedures

Tove provides the five common binary operations: addition, subtraction, multiplication, division, and modulo (remainder or residue). These are represented by the symbols +, -, *, /, and mod respectively. All of these are syntactically binary procedures. The unary negation is provided by the prefix procedure represented by the symbol underscore (_). All arithmetic is performed on sixteen bit numbers. If the operands are not numbers or if the results overflow the sixteen bits (32767 to -32768) then an error is reported.

Examples:

4 + 2 * 3 evaluates to 18
4 - 2 * 3 evaluates to 6
10 / 4 evaluates to 2
11 mod 4 evaluates to 3
_ 10 evaluates to -10
2.5.2. Logical Procedures

The basic logical procedures provided are or, and, exclusive or and logical negation. These are represented by the symbols | , &, xor, and - respectively. The first three are binary procedures while the latter is postfix. All these procedures operate bit-wise on the sixteen bits of their arguments, which must be numbers, and produce numbers.

Examples:

\[ 5 \& 3 \text{ is } 1 \]
\[ -7 | 3 \text{ is } -5 \]
\[ 3 \& 1 | 5 - \text{ is } -6 \]

2.5.3. String Procedures

For manipulating strings, basic procedures are provided which combine two strings, divide a string taking the first or last part, extract a character from a string, and give the length of a string. For the description of some of these procedures, we define the index of a character in a string to be a number giving the position of the character in the string, starting with zero for the first character.

EXPR a || EXPR b

The concatenation procedure returns a string which is the characters of the value of the first argument followed by the characters of the value of the second argument.

For example:
"hi" || "there" is "hithere"

"" || "empty" is "empty"

EXPR a <| EXPR b

This procedure returns a string which is the characters of the value of the first argument which are before the character whose index is the value of the second argument. The length of the string returned is therefore always the value of the second argument.

For example:

"abcdef" <| 3 is "abc"
"abcdef <| 6 is "abcdef"
"abc" <| 0 is ""
"abc" <| 4 is an error

EXPR a >| EXPR b

This procedure returns a string which is the characters of the value of the first argument which include and follow the character whose index is the value of the second argument. When combined with the <| procedure any substring can be extracted.

Examples:

"abcdef" >| 3 is "def"
"abcdef" >| 6 is ""
"abc" >| 0 is "abc"
"abcdef" >| 3 <| 2 is "de"
"abcdef" <| 3 >| 2 is "c"
"ab" >| 3 is an error
EXPR a char EXPR b

This procedure returns a string which is the character of the value of the first argument whose index is the value of the second argument.

For example:

"abc" char 2 is "c"
"ab" char 0 is "a"
"ab" char 2 is an error

len EXPR a

The length procedure returns the length of the value of its argument.

For example:

len "abc" is 3
len "" is 0
len 3 is an error

2.5.4. Cell Procedures

EXPR a create EXPR b

This procedure returns a new cell whose value is the value of the first argument and whose next cell is that given by the value of the second argument.

EXPR a <@

This procedure returns the value of the cell which is the value of a.
EXPR a  @}  
This procedure returns the next cell part of the cell which is the value of a.

EXPR a =< @}  EXPR b  
The value part of the cell which is the value of b is modified to the value of a. The modified cell is returned.

EXPR a =@>}  EXPR b  
The next cell part of the cell which is the value of b is modified to the cell which is the value of a. The modified cell is returned.

(| LIST a EXPR b ENDLIST , |)  
This procedure produces a list of cells. There is one cell for each part of the list a. Each cell has as value part the value of the corresponding part of the list a, and has as next part the cell corresponding to the next part of list a. The next cell part of the last cell is the nil cell. The procedure returns the first cell.

Examples: (assume that the atom nil has as value the nil cell.)
1 create nil creates a cell with value 1 and next cell nil.
1 create nil < @}  is 1
1 create nil @}>  is the nil cell
2 =< @} (1 create nil) < @}  is 2
2 =< @} (1 create nil) @}>  is the nil cell
1 create nil =⇒ (2 create nil) =⇒
    is a cell with value 1 and next cell nil
(| 1, 2, "ab", x |)
    is a list of cells with values
1, 2, "ab", and the value of x
(| "a", 2, 5, |) @⇒ @⇒ <@ is 5

2.5.5. Atom Procedures

EXPR a −⇒ EXPR b

The only operation available specifically for atoms is the
assignment of a value to an atom. This assignment procedure
assigns the value of a to the atom b. Note that b, the second
argument, is not evaluated and must be an atom. The assignment
is therefore left to right. The assignment procedure returns
the value of a which was assigned.
For example:
3 −⇒ x is 3 and the value of x is 3
3 −⇒ x −⇒ y is 3 and the values of x and y are 3

2.5.6. Conversion Procedures

EXPR a toString

This procedure converts the value of a into a string. If
the value of a is a string, that string is returned. If it is a
number, the string returned is the appropriate series of digits,
possibly with a leading minus sign (-). If it is an atom, the string returned is the symbol which is the name of the atom. If it is a pointer then an error is announced.

For example:

3 tostring is "3"
-15 tostring is "-15"
"a" tostring is "a"
'x' tostring is "x"

(see the description of the quote (') procedure)

EXPR a tonumber

The value of a is converted into a number. If it is a string, then it must be of the form: zero or more blanks, possibly a minus sign (-), zero or more blanks, one or more digits, the end of the string or some character which is not a digit (0 to 9).

For example:

"3270" tonumber is 3270
"-3" tonumber is -3
"15a" tonumber is 15

2.5.6. CONVERSION PROCEDURES
2.5.7. Comparison Procedure

EXPR a :: EXPR b

The comparison procedure is the main way to compare two values of the same type. If the value of a is not of the same type as the value of b, then an error is announced. If the value of a is greater than the value of b, the number four is returned. If equal, then two is returned and if less then one is returned. Numbers are compared in the usual arithmetic manner. Strings are compared by the usual alphabetic ordering: first compare the two strings for the length of the shorter, if they are equal then the shorter string is less than the longer. Atoms and cells have no ordering and therefore they are either equal or unequal. If they are equal, two is returned, if not then five is returned.

Examples:

1 :: 3 is 1
4 :: -2 is 4
"a" :: "ab" is 1
"b" :: "ab" is 4
"" :: " is 1
x :: x is 2
1 create nil :: (1 create nil) is 5
2.5.8. Input-Output Procedures

EXPR a print

The value of a is converted to a string (unless it is a cell) and appended to the current output buffer. The buffer is not written until the newline procedure is called, or the buffer is full (255 characters). The original (unconverted) value of a is returned.

newline

This procedure causes the current output buffer to be written and returns the empty string.

parser

This procedure reads the shortest possible expression from the input and returns its internal representation.

read_sexpr

An s-expression is read from the input and its internal representation is returned. For the definition of an s-expression see the description of s-expr arguments in the section on the parsing language. If the reading causes a new input line to be read, and it is read from a terminal, the prefix character is changed to a period (.). The s-expression must end with a period.

read
The next line of the input is returned, as a string. This procedure always causes the reading of a line. If the line is read from a terminal, the prefix character is changed to a minus sign (-).

EXPR a write

The value of a must be a string. It is written as the next line of output.

readfrom EXPR a

After calling this procedure, all input is read from the input file specified by the value of a. When the end of the input file is reached, input will once again be read from the default input file (which is SCARDS). Any lines read from the file will be echoed to the user (unless the corresponding status switch indicates otherwise – see the description of the status procedure). If the value of a is a string, then that is the name of the file to be read from. If the value of a is a number between 0 and 19, then it is the MTS logical unit number to be read from. If the value of a is -1, then the input will be read from SCARDS. Any other value of a is an error. The value of a is returned.

writeto EXPR a

After calling this procedure, lines written using the write procedure are written to the write file specified by the value of a. Error messages and output produced by print and newline
are always written to SPRINT. If the value of a is a string, then that is the name of the file to be written to. If the value of a is a number between 0 and 19, then it is the MTS logical unit number to be written to. If the value of a is -1 or -2, then lines will be written to SPRINT and SPUNCH, respectively. Any other value of a is an error. The value of a is returned.

Examples:

input: (1 print; " boo" print; 2 print;
   newline ; 3 print)

output: 1 boo2
   : 3

input: read
   : abc"def % hi

the value read is the string of characters: abc"def % hi

input: (read write)

the next input line will be echoed

2.5.9. Quoting Procedures

These two procedures return their arguments without evaluating them.

' EXPR a '

This procedure returns the internal representation of the expression which is the argument corresponding to a. Note that
the argument must be a valid expression, but is not evaluated by this procedure. This routine should be used to introduce values which are subsequently to be evaluated. (See the description of the eval routine.)

For example:

' newline ' is the atom named newline
' 2 ' is the number 2
' 1 + 2 ' is the internal representation of 1 + 2
' x ' is a syntax error if x is not the name of an atom

(SEXPR a).

This procedure returns the internal representation of the s-expression which is the argument corresponding to a. This should be the usual way of introducing particular cell structures or atoms which are to be used in cell structures. For example:

. newline . is the atom newline
. x . is the atom x. If x is not an atom then it is made a permanent one
. ( 1 + 2 ) . is a cell list of the number 1, atom +, and the number 2
2.5.10. Sequencing Procedures

These procedures affect the order of evaluation of expressions.

stop

This procedure stops the Tove system and returns the user to MTS.

EXPR a return

The value of a is returned as the value of the procedure body in which the call of return is made.

For example, if procedure dumb has as body the expression: (1 return; x) then the value of procedure dumb is always 1 and x is never evaluated in the procedure body.

if EXPR a then EXPR b else EXPR c endif

The if procedure evaluates a and based on that value it returns the value of either b or c. Only one of b or c is ever evaluated. The third argument, c, is evaluated when the value of a is one of: the number zero (0), the empty string (""), the nil cell, the nil atom, or the undefined value. In all other cases the second argument, b, is evaluated.

For example:

if 0 then 1+2 else 4 endif is 4
if 1 then 3+2 else 2+2 endif is 5

(: TRANSIENT_ATOM a : EXPR b : EXPR c :)

2.5.10. SEQUENCING PROCEDURES
This procedure defines a labelled block. The transient atom associated with \( a \) becomes the name of the block. The argument \( c \) must be the same atom as \( a \). The second argument, \( b \), is called the body of the block. The value of \( b \) is returned.

\[
\text{EXPR } a \text{ exit EXPR } b
\]

This procedure returns the value of \( a \) as the value of the block labelled by the atom which is the argument \( b \). The labelled block must still be active: that is, this procedure must only be called during the evaluation of the body of the block.

\[
\text{repeat EXPR } b
\]

This procedure restarts the evaluation of the body of the block labelled by the atom which is the argument associated with \( b \).

Examples:

\[
(\text{: block1 : 1 exit block1; 2 : block1 :) is 1}
\]

\[
(\text{: block2 : 1 print; repeat block2 : block2 :})
\]

is an infinite loop, printing 1 at each repetition

\[
(\text{LIST a EXPR b ENDLIST ;})
\]

Each of the arguments in the LIST are evaluated in turn (left to right) and the value of the last is returned. In the present implementation of the parser, the parentheses may be omitted from an enclosed argument.

2.5.10. SEQUENCING PROCEDURES
For example:

\[ (:x: \text{ first; second } :x:) \]

is the same as \[ (:x: (\text{ first; second }) :x:) \]

EXPR a eachpart EXPR b endeachpart

The eachpart procedure is used in procedures with LIST parameters. The argument a must be a LIST parameter atom of a procedure. For each part of the LIST argument corresponding to a, the arguments in that part of the list are bound to their corresponding formal parameters and the argument b (of eachpart) is evaluated. The last value of b is returned.

For example:

If put is defined as:

\[
\text{put LIST putlist EXPR putone EXPR puttwo ENDLIST , endput}
\]

then the body of put might be:

\[
\text{putlist eachpart putone print; puttwo print endeachpart}
\]

If put is called as:\n
\[
\text{put 1 "boo", "hi" 2 endput}
\]

then the output would be:\n
\[1\text{boo}hi2\]

For each part of the LIST the arguments in that part (1 and "boo" in the first part, "hi" and 2 in the second) are bound to putone and puttwo (respectively) and putone print; puttwo print is evaluated.
2.5.11. **Declaration Procedures**

**tran** TRANSIENT_ATOM a

This procedure declares its argument (which must be a symbol) to be a transient atom. It returns the atom. For example:

* tran x makes x a transient atom and returns it

**perm** PERMANENT_ATOM a

This procedure declares its argument (which must be a symbol) to be a permanent atom. It returns the atom. For example:

* perm y makes y a permanent atom and returns it

**proc** PROCEDURE_DEFINITION a endproc

This procedure defines a value procedure. It returns the procedure atom defined. For example:

* proc EXPR a ++ EXPR b is a + b * 2 endproc
  defines the procedure atom ++ to have syntax
  EXPR a ++ EXPR b and body a + b * 2.
  It returns the atom ++.

**nproc** PROCEDURE_DEFINITION a endnproc

This procedure is the same as the proc procedure except that it defines a control procedure. Control procedures differ from value procedures in that they evaluate their arguments when
the corresponding formal parameters are evaluated. Value procedures evaluate all their arguments before their body is evaluated. In either case evaluation of arguments is done in the environment of the call of the procedure, never in the environment of the body of the procedure.

For example:

If put is defined as:

```plaintext
proc put EXPR a EXPR b endput
    is a print; b print endproc
```

then: put 1 newline endput ; newline

produces as output a blank line (assuming the output buffer is initially empty) followed by a line containing the digit 1.

But if put is defined as:

```plaintext
nproc put EXPR a EXPR b endput
    is a print; b print endnproc
```

then: put 1 newline endput ; newline

produces as output a line containing the digit 1 (assuming the output buffer is initially empty) then a blank line.

2.5.12. Evaluation Procedures

These two procedures provide the user access to the interpreter.

EXPR a eval

The eval procedure allows the user to evaluate an internal
representation of a program. The value of a is treated as an internal representation of a program and is evaluated. The resulting value is returned.

For example:

1. \((+ 1 4)\). eval is 5
2. '1 + 4'. eval is 5

EXPR a applyto EXPR b

The value of a must be an atom. The value of that atom is returned. If the atom is a procedure atom, the arguments for the evaluation of the procedure body are given by the value of b. This value of b must be a cell list of the arguments. The arguments are evaluated as they would be if the procedure were called normally.

For example:

1. +. applyto. (2 3). is 5

2.5.13. Error Procedure

EXPR a error

The value of a is printed as an error message and error mode is entered. Refer to the section describing error processing. The string printed is returned if the evaluation is restarted.
2.5.14. **System Procedures**

These four procedures provide the user some access to the internal information maintained by Tove.

EXPR a status EXPR b

The status procedure provides a means for the user to affect the information printed by Tove during its processing. Specifically the user can test or set status switches. The value of b gives the status switch to be accessed and the value of a is what the switch is to be set to. The former value of the status switch is returned. Currently the status switches, the meaning of their values and their initial values are:

status 0:

0 - don't print the value of top level expressions
1 - print the value of top level forms

initially 1

status 1:

0 - don't echo lines read during a readfrom
1 - echo lines read during a readfrom

initially 1

EXPR a type

Type returns a number which is an encoding of the type of the value of a. Specifically:

0 - undefined value
1 - number
2 - string
3 - cell
4 - transient atom
5 - parameter atom
6 - permanent atom
7 - value procedure atom
8 - control procedure atom

EXPR a parse_table EXPR b

This procedure returns information about parsing. The value of b must be a number which determines the kind of information returned, and what the value of a is used for:

<table>
<thead>
<tr>
<th>value of b</th>
<th>value of a</th>
<th>information returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>an atom</td>
<td>one if the atom has a left parameter, else zero</td>
</tr>
<tr>
<td>2</td>
<td>an atom</td>
<td>the parse index (a number) of further parsing information for the atom. Zero if there is no further information</td>
</tr>
<tr>
<td>3</td>
<td>an atom</td>
<td>one if the atom has a left parameter, else zero</td>
</tr>
<tr>
<td>4</td>
<td>a parse index</td>
<td>an internal code number representing the parse kind</td>
</tr>
<tr>
<td>5</td>
<td>a parse index</td>
<td>a number or a string giving</td>
</tr>
</tbody>
</table>
further parse information

EXPR a internal

Internal returns an internal representation of the value of a. If it is a procedure atom, then the internal representation of the body of the procedure is returned. If it is a formal parameter of a control procedure, then the corresponding argument (unevaluated) is returned. In all other cases the value of a is returned.

For example:

If peek is defined as:

\[ nproc\ peek\ EXPR\ a\ is\ body\ endnproc \]

and peek is called as:

\[ peek\ some\_atom \]

then in the body of peek the expression:

\[ 'a'\ internal\ is\ the\ atom\ some\_atom \]

but

\[ a\ is\ the\ value\ of\ the\ atom\ some\_atom \]

2.6. ERROR PROCESSING

When an error is detected by the Tove system, an appropriate error message is printed. If the user is reading input from a "readfrom" file, (see the description of the readfrom procedure) then Tove attempts to continue execution. In all other cases the user is placed in error mode. The error mode processor changes the prefix character to a plus sign (+),
and reads and interprets simple commands from the user. A command consists only of the symbol which is the name of the command and may be abbreviated. Any input after the command name and before the end of the line is ignored. A description of the presently implemented commands follows.

Parse

This command causes the last complete parse to be printed, in a representation of the internal format used for expressions.

Symbols

The symbol table is printed.

Atoms

The atom table is printed.

Stack

The stack used by the interpreter for evaluation is printed.

Dump

Some internal tables are written to unit zero, which must be assigned to a sequential file.

Restart

Attempt to recover from the error. This causes a return to the computation which reported the error. If recovery from an
error is attempted during parsing, then in general the user must re-enter the last line from the point of error. If recovery is attempted during evaluation a default value for the offending computation is usually assumed.

Top

A return is made to the top level parse evaluate loop, abandoning the current parse or evaluation.

Quit

A return is made to MTS, in such a way that the Tove system can be restarted.
CHAPTER THREE
SOME DETAILS OF THE IMPLEMENTATION

In this chapter some of the details of the present implementation of the language system are presented. Particular emphasis is given to those parts of the implementation which are thought to be of interest.

The system is implemented in the SUE System Language by several major modules. The scanner converts the input characters into a token stream. The parser and translator are combined in a single module which converts the token stream into an internal representation reflecting the structure of the expressions, manages scope entry and exit, and processes declarations. The interpreter is responsible for the evaluation of the internal representations and includes routines for each of the basic procedures. In addition to these modules there are a number of miscellaneous routines such as the error processor and management routines for all of the various internal tables. There are several of these tables and areas, each of which is associated with some component of the language. These include the symbol table, atom table, parse table, eval stack, string area, and the cell area.
3.1. THE SCANNER

The scanner is divided into two parts. The first-level scanner reads the character input and converts it to a stream of numbers, strings, symbols, and comments. This part of the scanner uses well-known techniques and will not be further described. The second-level scanner is responsible for skipping comments and for converting symbols into corresponding atoms. This conversion of symbols is done using a link in the symbol table which indicates the atom in the current scope which has the symbol as name. These links are managed by the parser-translator. The second-level scanner therefore produces a stream of tokens consisting of numbers, strings, and atoms. In the case of atoms, the scanner also returns the symbol which represented the atom. This is done since the translator will have to know the symbol which was found if it was in the context of a declaration or a delimiter.

3.2. THE PARSER AND TRANSLATOR

3.2.1. Parsing Method

The parser is implemented as a recursive parsing interpreter. The parsing code which it interprets is an internal representation of the parsing statements. Each atom has associated with it some parsing information which is the representation of its syntax.

When the parser is called it is passed a goal which
indicates whether the parser should parse a simple expression or a complex one. A simple expression is one which is the shortest valid expression starting at the current token while a complex expression is the longest such valid expression. Simple expressions are rightmost arguments while complex expressions are interior arguments, as explained in the user's guide section on the parsing of expressions.

If the parser is parsing a simple expression, then it parses the current token and returns the internal representation obtained. The parsing of the current token proceeds by first examining the token. If the token is a number, a string, or a nilary atom (one with no parameters), then its internal representation is returned. If it is an atom with more complex syntax, then the parser interprets the parsing code for that syntax, which will parse all tokens involved in the syntax of the atom. During this parsing the internal representation of the call of the atom is constructed, and atom and procedure declarations are processed.

For an EXPR argument of the atom, the parser is called recursively to parse the expression. For PERMANENT_ATOM and TRANSIENT_ATOM arguments the next symbol is read and is made the name of a new atom. The symbol table entry for this symbol is adjusted to indicate that it is the name of the atom. For PROCEDURE_DEFINITION arguments, the parsing statement which follows the current token is converted to parsing code and associated with the procedure atom being declared. This procedure atom is declared at the current scope, then a new

3.2.1. PARSING METHOD
scope corresponding to the new procedure is entered. The formal parameters of the new procedure are declared in the new scope, then the parser is called to parse the body of the procedure. Lastly the new scope is closed and the original scope re-activated.

SEXPR arguments are parsed by a simple recursive routine which returns the internal representation of the argument.

For OPTIONAL arguments the next token is examined to see if it is the "optional symbol" specified in the syntax element for the argument. If it is then the parsing interpretation continues sequentially in the parsing code. Otherwise all arguments up to the end of the optional part are set to the undefined value and parsing continues after the end of the optional part.

LIST arguments require a looping structure in the parsing code. Since a LIST argument must consist of at least one occurrence of the sub-arguments corresponding to the "right syntax" portion of the LIST syntax element, the parser parses that section first. At the end of the parsing code for this there is code which causes the parser to check for the "separator symbol" and the "stop symbol". If the "separator symbol" is the next token then the scanner is called for the following token and the parser backs up in the parsing code to again check for the "separator symbol". If the next token is not the "separator symbol" then it is compared with the "stop symbol". If it is the "stop symbol" then the parsing continues with the syntax element following the LIST one. If it is not
the "stop symbol" and at least one "separator symbol" was found, then the parser branches back to the parsing code for the "right syntax" of the LIST syntax element. If no separator symbol was found an error is announced.

For delimiter syntax elements the current token is checked to see if it is the symbol specified. If it is not an error is announced.

The parsing of complex expressions requires that the parser continue after parsing the current token. In this case the parser examines the next token (after the parsing of the current token has read all tokens associated with the call of the current token) to see if it is an atom which has a left operand. If it is, then the current parse is the left argument for this atom and the process repeats by the parsing of this atom. Otherwise the parser returns the current parse.

3.2.2. Internal Representation Of Programs

The translator which is part of the parser module produces a prefix cell list representation of the expressions which comprise programs. This prefix representation is very similar to that used by LISP. Any expression is represented by a list of a cell for the procedure atom which is called followed by a cell for each of its formal parameters. /** needs mention of number and string expressions */ If an argument is an expression which is a call to a procedure with parameters and so will not fit in a single cell, it is represented by another cell list and the formal parameter's cell's value part is the first cell of
that list. If an argument is a LIST argument then it is represented by a cell list which begins with a cell containing a special atom indicating the LIST argument and a cell containing the number of sub-arguments in each part of the LIST. The rest of the cell list is a cell for each sub-argument in the entire LIST argument. The cells in the argument list of the called procedure which correspond to formal parameters of sub-arguments of a LIST argument all contain the undefined value.

For example:

gonsing language statement:

```
select LIST cases EXPR casepart ENDLIST , endselect
```

possible call:

```
select 2 + 3, "b" || "", cell_value <@ endselect
```

internal representation (in s-expression form):

```
(select (_list_ 1 (+ 2 3) ( || "b" "") ( <@ cell_value )
    _undef_ )
```

gonsing language statement:

```
select LIST cases EXPR selectpart then EXPR casepart ENDLIST , else EXPR default endselect
```

possible call:

```
select a < b then a + b,
    a > b then a
    else b
endselect
```

internal representation:

```
(select (_list_ 2
```
( < a b )
( + a b )
( > a b )

a )
_undef_

_undef_

b )

3.3. THE INTERPRETER

The interpreter is a simple stack-oriented recursive evaluator. It is given a value of the language and returns its evaluation. Numbers and strings evaluate to themselves, and so if the interpreter is passed a number or a string it merely returns it. If the interpreter is passed an atom, it is examined to see if it is a permanent atom (including procedure atoms) or a transient atom (including formal parameter atoms).

If the atom is permanent then its value part is evaluated. If this value part is a value of the language, (that is a number, string, atom, or cell) then it is returned as the value of the atom. This value part can also be an indication of one of the basic procedures or a programmer-defined procedure. In either of these cases the interpreter evaluates the indicated procedure with no arguments. If the procedure has parameters then an error is announced. Basic procedures are evaluated by calling the indicated built-in routine. Programmer-defined procedures are evaluated by first setting the current
environment pointer and saving its old value on the stack, allocating space on the stack for the transient atoms of the procedure, and then evaluating the cell which is the body of the procedure. After this the environment and stack are reset and the value found for the body is returned.

If the interpreter is passed a transient atom or a formal parameter atom, then the atom contains, in place of its value part, a number which is its offset on the stack from the current environment pointer. Accordingly the value part of the atom is obtained by combining the offset and the current environment pointer and retrieving the value from the stack. This value is then evaluated in the same manner as the value part of a permanent atom.

To evaluate a cell the interpreter requires that the cell be a valid internal representation of an expression, as outlined in the section on internal representations of programs. If the value part of the cell is a number or a string, then that number or string is returned. If the value part of the cell is another cell, then an error is announced as this cannot occur in a valid internal representation of a program. If the value part is an atom, then that atom's value part is obtained and evaluated as outlined for the evaluation of atoms. In this case, however, if the atom's value part indicates a procedure, then the procedure's arguments are the value parts of the cells which are the next part of the original cell.

If the procedure is a value procedure, these value parts are evaluated and their values placed on the stack so that they
are the initial values of the formal parameter atoms. This is done before the environment pointer is changed when a programmer-defined procedure is evaluated, or before the built-in routine is called when a basic procedure is evaluated.

If the procedure to be invoked is a control procedure, then the arguments are not evaluated. To ensure that the correct environment will be used when an argument is evaluated in a programmer-defined control procedure, those arguments that are atoms or cells are marked as control procedures. The evaluation of numbers, string, and the undefined value cannot be affected by the environment. Whenever the value part of an atom indicates that it is a control argument, the evaluation of that part causes the current environment pointer to be stacked and the previous environment to become the current one. The atom or cell which is the argument is then evaluated, the current environment is restored, and the value found for the argument is returned.

3.3.1. Block Exit And Repeat

The control structures consisting of block exit and block repeat are much easier to implement than the unrestricted "go to". When a labelled block is entered, the label atom of the block is given a value which is a stack pointer indicating the block routine activation entry. An exit which is given the label as its argument pops the stack back to this activation entry and causes the interpreter to return from the call of the block. A repeat also pops the stack back to this activation entry.
entry but the causes the interpreter to call the block again. Thus the structured exit and repeat require labels to be only stack pointers, not program counters as required by the go to. An interesting consequence of the use of exit and repeat in place of the go to is that the interpreter has no need of the concept of a program counter, since only the go to requires this concept.

3.4. TABLE MANAGEMENT

The internal tables of the system are managed by several routines, some of which are specific to individual tables. All the tables can be dynamically expanded by the system, except the symbol table. Whenever a table or area is expanded, error mode is first entered with an error message indicating the table being expanded. This allows the user to abandon the current computation and the expansion, or to continue. See the user's guide section on error mode processing. Each table is accessed by a pointer to the beginning of the table and the index of the entry desired. Therefore table expansion can be accomplished by obtaining a space larger than the current size of the table from the operating system, copying the information from the old space to the new one, then changing the base pointer to indicate the new space. This also allows the table indices to be halfwords rather than fullwords, resulting in some economy of storage space.

The symbol table is not dynamically expanded because it is accessed by a hashing routine to look up symbols as they are
read by the scanner. It could be expanded dynamically, but this would require the re-hashing of all entries, rather than copying, from the old space to a new one. The atom and parse tables are not garbage-collected. When one of these runs out of space, it is expanded. The eval stack is also expanded when it overflows.

The string area is where the actual characters of strings are stored, as well as the character representations of symbols (and therefore the names of atoms). In the tables and areas of the system a character string is represented by a length (between zero and 255) and the index into the string area of the first character of the string. When the string area runs out of space, a new string area the same size as the old one is obtained from the operating system and only active strings are copied from the old area to the new one. The active strings are found by a garbage collector (actually a non-garbage collector) which scans all the tables which have string area indices in them. The indices are changed to indicate the positions of the character strings in the new string area. The substringing procedures of the language do not make copies of the character strings they operate on, so that they return a string length and an index which points into the same character string as the string index from which the substring was taken. This implies that the copying from the old string area to the new may increase the size of the active area, so provision must be made to expand the new area during the copy. Fortunately this is not difficult and only requires the obtaining of a "new new" string.
area and the direct copy of the new area into the new new area, since everything in the new area is active. Another solution to this problem is that chosen by XPL [XPL], which also does substrings without copying. When the XPL string area overflows, all the active descriptors are sorted by the order of their indices into the string area. The compactification routine can then copy an active string only once and adjust all indices into it correctly.

The cell area is maintained by a mark-sweep garbage collector like that of most LISP implementations, using a free list and no compaction. However, since the parser routines keep some cell indices on the SUE stack that the garbage collector cannot access, it is not always safe to garbage collect. Accordingly, when the routine responsible for allocating cells realizes that there are very few free cells left, it requests a garbage collection. The collection is not done until it is safe to do so. If the cell area is completely exhausted and another cell is to be allocated, then the cell area is expanded (and not garbage collected). Of course the cell area is also expanded if the garbage collector does not find enough free cells.
CHAPTER FOUR
CONCLUSIONS

This chapter presents some conclusions about the design and implementation of the Tove system: its successes and failures, how it can be improved, and some thoughts for future work.

4.1. PROBLEMS AND POSSIBLE SOLUTIONS

It is to be hoped that the identification of some of the mistakes of this system which follows will enable others to avoid these pitfalls.

4.1.1. The Scanner

The most obvious flaw in Tove and the most annoying to the user is the crude methods used by the scanner to break character strings into tokens. Specifically the rules regarding the formation of symbols are particularly simple-minded, often requiring the user to insert blanks in places he may not expect, especially if he is an experienced programmer. Essentially the rules used do not correspond to common practice, leading to confusions and a lack of readability. One might argue that the programmer could learn to live with these restrictions and accept them as a price to be paid for flexibility, but such arguments sound too much like those of the bad tailor described by Weinberg. The tailor produced mis-fitting suits but was careful to instruct his clients on how they could hunch over and

4.1.1. THE SCANNER
limp to make the suits look as though they were correctly tailored.

The difficulty of the scanning problem is that more sophisticated rules for syllabication (breaking the character stream into tokens) that would be more natural, would also be more complicated. Probably the best solution is to divide the special symbols into different classes and state the syllabication rules in terms of these classes. For example, the parentheses should always be treated as separate symbols since this corresponds to common usage.

4.1.2. The Parser And Translator

Another problem which causes severe difficulties in reading the language is the necessity to process syntactic declarations (parsing statements) while parsing the program. This must be done by both the parser of the system and anyone attempting to read the language. Without this one cannot tell delimiters from binary, prefix, suffix, or nilary operators. If unconventional syntax is used in a program, the program can be incomprehensible to a reader. Essentially the problem is that too much of the translator's work must be performed during the parsing.

A possible solution to this problem which should also aid the syllabication problem, is to use typographic conventions to distinguish symbols in the various syntactic categories. For example, a convention could distinguish symbols which are delimiters from those that are operators by insisting that delimiters be a series of zero or more alphabetic characters.
followed by a colon, comma, or semicolon. Parentheses could always be used as corral delimiters, that is, they could be used to enclose any right syntax which is more complex than a simple expression. Similarly, one could require that symbols composed of special characters be used for binary or unary operators and alphabetic symbols be used only for nilary operators. Essentially, these requirements are that the name itself of an object describes the syntactic properties of that object.

Indeed, it is felt that with some such typographic conventions it should be possible to produce a parser which is capable of parsing programs without concurrently processing syntactic declarations. This is highly desirable because it would imply better readability and also a cleaner structure for the system. It would also mean that a multi-pass system would be possible, in which the declaration of a procedure need not precede its use. This type of system is considered to be very useful and convenient for top-down programming techniques and would also enable the implementation of a program development facility such as that presented by DuMont. [DuMo]

Another weakness of the current parser is its dismal error detection and recovery. It will announce a syntactic error without reading beyond the first symbol in error, but it may return from some recursive invocations after looking at the symbol in error and before announcing the error. This situation is similar to that of some bottom-up parsers which do some reductions before announcing a syntax error. It is caused, in part, by the incomplete knowledge of valid lookahead symbols.
which is in turn directly attributable to the incremental nature of the processing of the parsing language — each parsing language statement is translated and interpreted independently of other parsing language statements. In interpreting a rightmost expression, the parser does not know if a lookahead symbol is invalid or part of some other parsing statement. In this case the parser must return to its caller and, possibly after other returns, the lookahead symbol will be accepted or rejected. The difficulty is that if it is rejected, one would like to restart the parsing with a new lookahead symbol, but in the incarnation of the parser that first saw the errant symbol, so that the parsing will continue as though the symbol in error was never there.

One way to achieve this is to maintain the position of the top of the parsing stack at the time of the last read of a symbol. If an error is found in the lookahead symbol, then it should be possible to restore the parsing stack and hence the state of the parser to that of the time of the last read of a symbol, thus reverting to the desired incarnation of the parser. Clearly there is a need for more work to be done on syntactic error recovery.

4.1.2. THE PARSER AND TRANSLATOR
4.1.3. The Interpreter

Largely due to the considerable reliance on previous semantic designs, as outlined in the introduction, the interpreter has very few problems. One of the remaining problems is an oversight relating to the ability to assign values to atoms. As currently constituted, the system is not able to correctly assign a value to a transient atom which is passed into a procedure, since the assignment procedure does not perform the assignment in the correct environment. This could be corrected by making the assignment procedure check the value it is about to change. If that value is a control procedure argument which is an atom, then the assignment should proceed to that atom's value in the previous environment, and continue in this manner until it reached a value which was not an atomic control argument. This would enable the writing of such control structures as for-loops which increment or change a variable that they are passed.

4.2. SOME SUCCESSES

Despite the problems in the implementation of the current system, the use of a parsing language to incrementally describe the syntax of the language is felt to be quite successful. Much profitable work has been done on the study of syntax described by the BNF notations and it is felt that future work in the study of other syntax description languages will prove profitable.

The semantic design is considered to be largely successful,
particularly the decision to use block exit and repeat and the control procedures. The simplicity of implementation of the block exit and repeat, combined with their encouragement of clearer programs are powerful arguments for their replacement of the "go to". The inclusion of control procedures is thought to be a very successful method for incorporating control structures into the procedural organisation of the system.

On the subject of complete compilation into conventional machine code, it is felt that only the evaluation procedures (eval and apply), the system procedures, and some of the input procedures rely on the interpretive nature of the implementation. There are some worries about the ability to efficiently compile control procedures into more conventional machine code, but if the control procedures are non-recursive and small, it should be possible to very efficiently compile them "in-line", that is into the same machine code unit as their calls, with one copy of the control procedure code for each call of the control procedure. This is the same method that is used in conventional algorithmic languages to compile their built-in control structures. If a control procedure does not meet these criteria, methods similar to those used for Algol "call-by-name" parameters can be used to compile the procedure. Certainly there is nothing in the syntactic design of the language that precludes an incorporation of declarations of the types of variables and procedures and compile-time type-checking. This implies that a similar language would be amicable to complete compilation into machine code.

4.2. SOME SUCCESSES
BIBLIOGRAPHY


APPENDIX I

SAMPLE RUN OF TOVE

The following is a sample run of Tove, demonstrating some of its features.

```
# R COMPILER T=2 PAR=HIGH_WATER STACK=3P
* READFROM "EXAMPLES"
* 
* STR "EXAMPLES"
* % SAMPLE PROCEDURES
* 
* % SOME COMPARISON PROCEDURES
* 
* PROC EXPR A > EXPR B IS
*  A :: B & 4 ENDPROC
* 
* ATOM  61 >
* 
* PROC EXPR A < EXPR B IS
*  A :: B & 1 ENDPROC
* 
* ATOM  64 <
* 
* PROC EXPR A = EXPR B IS
*  A :: B & 2 ENDPROC
* 
* ATOM  67 =
* 
* PROC EXPR A >= EXPR B IS
*  A :: B & 6 ENDPROC
* 
* ATOM  70 >=
* 
* PROC EXPR A <= EXPR B IS
* 
* ATOM  73 <=
* 
* % FACTORIAL PROCEDURE
* % USES RECURSION.
* 
* PROC ! EXPR N IS
*  IF N <= 0
*   THEN 1
*   ELSE N * ! (N-1)
*  ENDIF
* ENDPROC
```

I. SAMPLE RUN OF TOVE
* ATOM 76!

* % TOWERS OF HANOI PROCEDURE
* PROC HANOI (EXPR N, EXPR S, EXPR I, EXPR D) IS
* IF N <= 0 THEN 0 RETURN
* ELSE HANOI(N-1,S,D,I);
* "MOVE "||(N TOSTRING)||" FROM "||S||" TO "||D PRINT;
* NEWLINE;
* HANOI(N-1,I,S,D)
* ENDIF
* ENDPROC

* ATOM 78 HANOI

* % S-EXPRESSION PRINTER.
* % -- TAKES ANY VALUE AND PRINTS IT IN S-EXPRESSION FORM.
* PROC EXPR VAL PUT_SEXPR IS
* PERM CELL_TYPE;
* 3 -> CELL_TYPE; % CODE INDICATING VALUE IS A CELL
* % PROCEDURE TO PUT BLANKS INTO OUTPUT BUFFER
* % USED TO INDENT
* PROC PUT_BLANKS EXPR HOW_MANY IS
* (:PUT_LOOP:
* IF HOW_MANY > 0
* THEN " " PRINT;
* HOW_MANY - 1 -> HOW_MANY;
* REPEAT PUT_LOOP
* ELSE ""
* ENDF
* :PUT_LOOP:)
* ENDPROC;

* % PROCEDURE TO DETERMINE IF A CELL LIST
* % HAS A SUBLIST
* % RETURNS 1 IF SO, OTHERWISE 0
* PROC HAS_SUBLIST EXPR VAL IS
* % NOTE THAT VAL IN THIS PROCEDURE IS
* % IS A DIFFERENT ATOM THAN THE VAL
* % IN PUT_SEXPR WHICH IS AGAIN DIFFERENT
* % FROM THE VAL IN SEXPR_OUT
* (: LIST_LOOP :)
* IF VAL % IS NOT NIL CELL
* THEN IF VAL <@ TYPE = CELL_TYPE
* THEN 1
* ELSE VAL @> -> VAL;
* REPEAT LIST_LOOP

I. SAMPLE RUN OF TOVE
* \* ENDIF
* ELSE 0
* ENDIF
* :LIST_LOOP:
* ENDPROC;
* *
* % AMOUNT TO INDENT FOR EACH LEVEL
* *
* PERM INDENT_INCREMENT;
* 4 \rightarrow INDENT_INCREMENT;
* *
* % RECURSIVE PROCEDURE TO PRINT THE VALUE
* *
* PROC EXPR VAL SEXPR_OUT EXPR INDENT IS
* EXPANDING CELL AREA
* IF VAL TYPE = CELL_TYPE
* THEN "(" PRINT;
* TRAN COMPLEX;
* HAS_SUBLIST VAL \rightarrow COMPLEX;
* (:OUT CELLS:
* IF VAL \% IS NOT THE NIL CELL
* THEN VAL <@ SEXPR_OUT (INDENT + INDENT_INCREMENT) ;
* % PRINT THE VALUE OF THE CELL
* VAL @> \rightarrow VAL; \% STEP TO NEXT CELL
* IF VAL \% IS NOT THE NIL CELL
* THEN IF COMPLEX
* THEN NEWLINE;
* PUT_BLANKS INDENT;
* ELSE "" PRINT
* ENDIF;
* REPEAT OUT CELLS
* ELSE "" \% DO NOTHING
* ENDIF
* ELSE "" \% DO NOTHING
* ENDIF
* :OUT CELLS:)
* "")" PRINT;
* ELSE VAL PRINT \% PRINT THE SIMPLE VALUE
* ENDIF
* ENDPROC;
* *
* % DO THE PRINTING
* *
* NEWLINE;
* VAL SEXPR_OUT 4;
* NEWLINE
* *
* ENDPROC
* *
* ATOM 83 PUT_SEXPR
* *
* % CONTROL PROCEDURE FOR LISP-STYLE COND
* *
* NPROC COND LIST CONDPARTS EXPR TEST THEN EXPR BODY

I. SAMPLE RUN OF TOVE
* ENDLIST, ELSE Expr DEFAULT ENDCOND
* IS
* (: COND_BLOCK :
* CONDPARTS EACHPART
* IF TEST
* THEN BODY EXIT COND_BLOCK
* ELSE 0
* ENDIF
* ENDEACHPART ;
* DEFAULT
* : COND_BLOCK ;)
* ENDNPROC
* * ATOM 98 COND
* *
* % CONTROL PROCEDURE FOR IF
* % WITH POSSIBLY OMITTED ELSE PART
* *
* NPROC WHEN Expr TEST
* THEN Expr THEN_PART
* EXPANDING PARSE TABLE
* OPTIONAL ELSE Expr ELSE_PART ENDOPTIONAL
* ENDWHEN
* IS
* IF TEST
* THEN THEN_PART
* ELSE ELSE_PART
* ENDIF
* ENDNPROC
* *
* ATOM 104 WHEN
* *
* WHEN 1 THEN "TRUE" ENDWHEN
* *
* STR "TRUE"
* *
* % SAMPLE CALLS OF FACTORIAL
* !1
* *
* INT 1
* !2
* *
* INT 2
* !4
* *
* INT 24
* !7
* *
* INT 5040
* !8
* *
* RESULT OF * IS GREATER THAN 32767
* RES
* *
I. SAMPLE RUN OF TOVE
* INT 32767
* % NOTE DEFAULT VALUE ASSUMED AFTER OVERFLOW
* 
* % SAMPLE CALLS OF HANOI
* HANOI (2, "S", "I", "D")
* MOVE 1 FROM S TO I
* MOVE 2 FROM S TO D
* MOVE 1 FROM I TO D
* 
* INT 0
* HANOI (3, "SOURCE", "INTERMEDIATE", "DESTINATION")
* MOVE 1 FROM SOURCE TO DESTINATION
* MOVE 2 FROM SOURCE TO INTERMEDIATE
* MOVE 1 FROM DESTINATION TO INTERMEDIATE
* MOVE 3 FROM SOURCE TO DESTINATION
* MOVE 1 FROM INTERMEDIATE TO SOURCE
* MOVE 2 FROM INTERMEDIATE TO DESTINATION
* MOVE 1 FROM SOURCE TO DESTINATION
* 
* INT 0
* 
* % SAMPLE CALLS OF THE S-EXPRESSION PRINTER
* ( . ( A B C ) . PUT_SEXPR)
* 
* (A B C)
* 
* STR ""
* ( . ( A ( B ) C ) . PUT_SEXPR)
* 
* (A
* (B)
* C)
* 
* STR ""
* ( . HANOI . INTERNAL 3> PUT_SEXPR )
* % TO PRINT THE BODY OF HANOI
* 
* (IF
* (<= N 0)
* (RETURN 0)
* ((
* (HANOI
* (- N 1)
* S
* D
* I)
* (PRINT
* (|
* (|
* (|
* MOVE
* (TOSTRING N))
* 
* I. SAMPLE RUN OF TOVE
* S) FROM )
* D}) TO )
* NEWLINE (HANOI
* (- N 1)
* I
* S
* D})
* STR ""
* % SAMPLE CALLS OF COND
* COND
* 1 THEN "ONE" PRINT,
* 0 PRINT THEN "TWO" PRINT
* ELSE "ELSE" PRINT
* ENDCOND
* ONE
* STR "ONE"
* COND
* "" THEN "ONE" PRINT,
* 1 PRINT THEN "TWO" PRINT
* ELSE "ELSE" PRINT
* ENDCOND
* ITWO
* STR "TWO"
* COND
* "" THEN "ONE" PRINT,
* 0 PRINT THEN "TWO" PRINT, % NOTE EXTRA COMMA
* ELSE "ELSE" PRINT
* ENDCOND
* OELSE
* STR "ELSE"
* % CALLS TO ILLUSTRATE THE INDEPENDENCE OF ATOMS...
* PERM DEFAULT % SAME NAME AS LAST PARAMETER OF COND
* *
* ATOM 111 DEFAULT
* (1-> DEFAULT) % SET THE ATOM
* *
* INT 1
* COND
* DEFAULT THEN "ONE"
* ELSE 0
* ENDCOND
* *
* STR "ONE"
* STOP
* *
* STACK HIGH WATER MARK - ALLOCATED: 12288 USED: 11402