Nils M Holm

Lightweight Compiler Techniques

Nils M Holm
Lightweight Compiler Techniques

Print and Distribution: Lulu Press, Inc

This book was typeset using TROFF(1) and friends on a FreeBSD system.

"Lightweight Compiler Techniques" was printed in small quantities and distributed by myself in the 1990's. I always wanted to create a second edition, but never managed to do so. However, a few people still seem to be interested in this work, so I decided to launch TROFF one last time and try to get the original manuscript into a printable form, and here it is.

Note: this is the original, unrevised text from 1996 with a few minor corrections from 2002. The layout may look a bit odd here and there. This is because the original book had a different size. I am sorry, but I currently do not have the time to fix this.

Nils M Holm, June 2006

Table of Contents

1
An Introduction to T3X

T3X is a small, portable, procedural, block-structured, recursive, almost typeless, and to some degree object oriented language. Its syntax is derived from Pascal and BCPL, and its object oriented model is similar to that of Java, but much simpler. The structured approach to programming is well-understood, provides a sufficient degree of abstraction and can be easily translated into native machine code at the same time. The object model eases the development of general and resusable code. T3X is an *imperative* language. This means that a program consists of a set of *instructions* which tell the computer in what way to manipulate the *data* defined by the program. An instruction is also called a *statement*. In structured programming languages, there are four fundamental ways of formulating statements:

• Assignments

• Sequences

• Branches

• Iterations

In a *sequence* – which is basically a list of statements – the statements are processed from the top towards the bottom of the list. Each statement is guarranteed to be completely processed when the next one is interpreted. A *branch* is a statement which is executed only if an associated *condition* applies. *Iteration* is the *repetition* of a statement depending on a condition. In a *block-structured* language, statements may be grouped in *statement blocks* or *compound statements*. Each block may have its own local data which cannot be affected by statements contained in other blocks.

An additional layer of abstraction is added to an imperative, block-structured language by providing user-defined *procedures* or *functions* (in this book, these terms will be used synonymously). A procedure is a statement or a set of statements which is bound to a symbolic name. A procedure can be executed by coding a *call* to that procedure. Most languages provide a mechanism to transport data to a procedure and return a value to the calling program. Some languages (like BCPL and Pascal) make a distinction between procedures and functions, others (like K&R **C**) do not. In languages which make a distinction between procedures and functions, only functions may return values. In T3X, all procedures return values, but the caller is free to ignore them. Therefore, procedures and functions are basically the same.

Another level of abstraction is provided by adding an *object model* to the language. The object model of T3X consists solely of

• Classes

• Objects

• Messages

Classes are used to encapsulate code and data of a program. It may contain any number of data objects and procedures. Only public procedures (so-called *methods*) may be called by other procedures (or methods). *Objects* are used to *instantiate* classes. Each instance of a class has its own private data area. Hence the same class may be used for a template to create independent objects. *Messages* are used to activate methods of specific objects. T3X does not provide inheritance, since it is the source of a whole load of semantic problems. It does not support different protection levels (like public data or 'friend' relationships), either, because these concepts undermine the object oriented model.

T3X is an *almost typeless* language. There exist two different types, so-called *atomic variables* which may hold small data objects, like characters, numbers and references to other data objects, and *vectors* which are used to store logically connected groups of small data objects. Additionally, there are constants, templates for defining structured data objects and classes, and different types of procedure declarations. The T3X compiler does not allow some combinations of operators which do not make sense (like assigning a value to a procedure or sending a message to a vector). Consequently, T3X's type checking is much more strict than for example BCPL's, but much less restrictive than Pascal's. Weakly typed and typeless languages have been exposed to a lot of critique in the past, because they are considered 'insecure', but the degree of simplicity and flexibility which is bought by 'sacrificing' this bit of security is immense.

The type checking mechanisms of the T3X language are limited to the detection of

• wrong argument counts in procedure calls

• assignments to non-variables

• calls of non-procedures

• instantiations of non-classes

• sending messages to non-objects

• sending non-messages to objects

• dependencies on non-classes

BTW: during the development of an early version of T3X, a severe bug occurred in the compiler. After tracking it down, it turned out that was limited to the (type-safe) ANSI **C** version of the translator and did not addect the T3X version. Of course, this was coincidence, but to some degree it weakens the proposition that typeless languages are per se insecure and dangerous.

The remainder of this chapter describes the structure of T3X programs and the single components a program may consist of.

1.1 Programs

The components used to build T3X programs form a strict hierarchy which is displayed in figure 1.

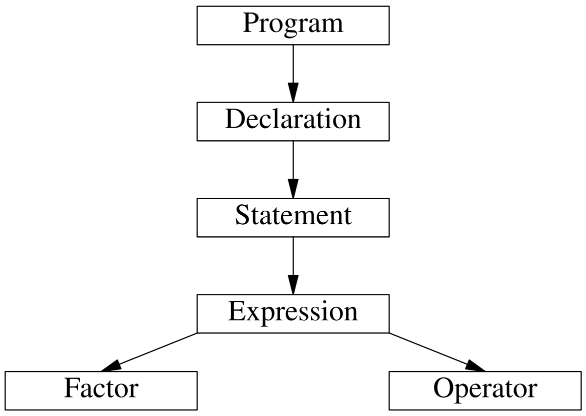


Figure 1: The Program Hierarchy in Imperative Languages

Each T3X program is a set of declarations which may define data objects like variables and vectors, templates like constants, structures. and classes, or procedures. Each procedure definition contains at least one statement. Most statements employ one or more expressions. Expressions, finally, are divided into operators and factors. Each expression consists of at least of one factor. Of course, declarations are statements, too, but they belong to a different class than the statements which are used to form the bodies of a procedures. Statements which 'do something' at run time are called *executable statements* and only those are covered by the word 'statement' in the above picture. *Declaration statements* define data, but have no visible effect at run time. (This is not entirely true, because *local* declarations may have runtime effects very well, but this fact may be ignored for now.)

In the following sections, the components of the hierarchy displayed in figure 1 will be explained basically from the bottom up, because each component is based upon the one below. Declarations are the only exception. Because even factors may require data objects defined in declarations, declaration statements will be discussed first.

Before diving into the details of statements and declarations, however, some even more basic properties of T3X will be explained.

1.1.1 The Input Alphabet

The input alphabet of a language determines the set of characters which will be accepted by its compiler (or interpreter). When the T3X translator finds a character which is not contained in this set in its input program text, it signals an error. The translator will accept the following characters:

• Upper and lower case letter (`A`...`Z`, `a`...`z`).

• Decimal digits (`0`...`9`).

• This set of special characters:
`! " # % & ( ) * + , - . / : ; < = > @ [ ] ^ _ | ~`

• The white space characters ‘blank’, HT (tab), CR, LF, and FF.

Alphabetic characters will be folded to all lower case upon input except for occurances in strings and character literals.

1.1.2 Comments

A *comment* is a part of the input text which does not belong to the program itself and should be invisible to most parts of the translator. All comments will be removed from the input as soon as they are recognized (during *lexical analysis*, technically speaking). Since comment texts do not belong to the program text, they may contain any characters available on the host computer system.

In T3X, a comment is introduced by a single exclamation point (`!`). It may begin between any two *tokens* or at the beginning of the program. Each comment extends up to and including the end of the line it was started in. To the subsequent phases of the translation process, a comment looks like a single space character, and therefore, it may not occur inside of a token. For an exact definition of T3X tokens, see the section about *scanning* in the following chapter. Fow now, it is sufficient to know that all coherent partitions of text, like keywords, operators, symbolic names, strings, numbers, etc are tokens.

```
P(s) DO VAR i;
    i := 0;
    WHILE ( ! collect hex digits
        '0' <= s[i] /\ s[i] <= '9' \/  ! decimal digit
        'a' <= s[i] /\ s[i] <= 'f' \/  ! lowercase hex
        'A' <= s[i] /\ s[i] <= 'F'     ! uppercase hex
    ) DO
        i := i+1;
    END
    RETURN i;
END
```

Example 1: Comments

Example 1 shows some code in which comments have been inserted at different places inside of a single statement.

1.1.3 Naming Conventions

The naming conventions describe what a valid *symbol name* looks like and how the equivalence of two names is determined by the compiler. The T3X naming conventions are similar to **C**'s. A symbol name may consist of

• alphabetic characters
• decimal digits
• the underscore character (_)

The first character must be alphabetic or an underscore. If the first character could be a digit as well, it would be impossible to distinguish names from numbers by their first characters.

Since the translator folds upper case characters to lower case upon input, the case of symolic names is not preserved. Consequently, the words

```
abc abC aBc aBC Abc AbC ABc ABC
```

would all denote the same symbol in T3X. The length of symbol names is itself not restricted and all characters contained in two names ar guaranteed to be compared when checking for the equivalence of two symbols. For example, the names

```
This_is_a_very_very_long_name_ending_with_5
```

and

```
This_is_a_very_very_long_name_ending_with_7
```

are considered different symbols by the translator.

1.1.4 Data Declarations

In T3X, each data object and template must be *declared* before its first use, because otherwise, the translator would not 'know' what kind of object it is. A *data object* is something which can be used to store data in it. There are *atomic* variables for storing small data objects like characters, numbers, and references, and vector objects for storing collections of atomic data objects and compound data objects like strings, structures, tables, and objects. *Templates* are data objects which do not allocate any space, but will be replaced in situ with their associated values, like constants and structure layouts. In the following text, the T3X data declarations will be explained in detail.

Atomic Variables

Each (atomic) T3X variable allocates exactly one machine word. When talking about *variables* in the remainder of this document, the attribute *atomic* will be always implied. Vectors will be explicitly referred to as *vectors* or maybe as *arrays*.

Variables are defined using a `VAR` statement. Any number of names may be defined in a single statement:

```
VAR x_coord, y_coord, depth;
```

Although, it is recommended to define only logically connected variables in a single statement. All types of values may be stored in a variable: numeric values, pointers to strings, pointers to vectors, pointers to structures, or literal characters. The range of numeric values which may be stored in a variable actually depends upon the implementation, but the Tcode engine only uses 16 bits to represent a cell – independently from the underlying platform. Therefore, programs which use values not in the range

```
-32768 ... 65535
```

should be considered machine-dependant.

When a variable is placed in an expression (frequently also called a *righthand side value*), it evaluates to its *value*. When it is placed on the *lefthand side* of an assignment, however, it evaluates to its *address* (which will be immediately dereferenced by the assignment operator).

Constants

Constants are variables which exist only at compile time. Instead of an automatically assigned address, they are initialized with an explicit value when they are declared. Since they are known only at compile time, the values of constants may not change at run time. Any number of constants may be declared in a single `CONST` statement:

```
CONST read=1, write=2, rdwr=read|write;
```

Each constant name must be followed by an equal (=) sign and a *constant expression* which evaluates to the value of the constant. Constant expressions will be explained in a later section. Constants may occur only in righthand side values where they evaluate to their values.

Vectors

Vectors, like constants, are compile time variables. When they are declared, they will be initialized with the address of an array of subsequent machine words. These machine words are called the *members* or *elements* of the vector. The address of a vector is equal to the address of its first member. Any number of vectors may be defined in a single `VAR` statement. Declarations of vectors and atomic variables may be mixed in the same statement:

```
VAR RingBuffer[1000], Head, Tail;
```

Vector declarations differ from atomic variable declarations by the trailing square brackets containing a constant expression which specifies the size of the vector in machine words. The first member of a vector has the *index value* 0 and the last one has the index *vectorsize*-1 (999 in the above example).

Since vector addresses are stored in compile time variables, they may not change at run time. It *is* legal to change the values of *vector members*, though. When occuring in righthand side expressions, vector names evaluate to the addresses of their associated arrays.

Single members of a vector may be addressed using the *subscript operator* '`[]`'. The expression

```
v[5]
```

for example, evaluates to the sixth member of the vector *v*. Subscripted vectors may occur in lefthand side expressions as well. The assignment

```
v[i] := 99;
```

would change the `i`'th member of *v* to 99. Like atomic variables, the members of vectors may be used to store any data type, even pointers to (nested) vectors. See the description of the `[]`-operator for details about nested vectors.

Byte Vectors

A special case of the vector is the *byte vector*. Like 'ordinary' vectors, they are declared in `VAR` statements but using a double colon (`::`) instead of the square brackets:

```
VAR Input::256, Output::256;
```

The only difference between a vector and a byte vector is the computation of the required size. The size value after the `::`-operator specifies the number of *characters* required. The amount of memory actually allocated depends on the size of a machine word on the target machine which is denoted by the class constant *T3X.BPW* (*B*ytes *P*er *W*ord). The reference implementation of T3X, which is based upon the Tcode machine, assumes *BPW*=2. Generally, the size of a byte vector is computed using this formula:

$$\frac{\mathit{vectorsize} + \mathit{BPW} - 1}{\mathit{BPW}}$$

which allocates enough space for *at least vectorsize* characters. No further type information is stored in vector entries. Therefore, it is valid to access byte vectors using '`[]`' and word vectors using '`::`'. However, this technique is deprecated, because the size of byte vectors may vary among different implementations.

Structures

A structure is a composed data object. Only one structure may be defined in a single `STRUCT` statement:

```
STRUCT point = pt_x, pt_y;
```

Note that this statement does not actually create a new data object, but only the 'layout' of a *point* structure. To create some point data objects, an additional `VAR` statement is required.

```
VAR point_a[point], point_b[point];
```

creates two *point* data objects, *point_a* and *point_b*. The members of such a data object can be addressed using the subscript operator: *point_a[pt_x]* and *point_a[pt_y]*.

Structures do not really have an own type. As the declaration and member access syntax already suggests, they are ordinary arrays and the member names are constants. In fact, the statement

```
STRUCT s = a, b, c;
```

is perfectly equal to

```
CONST s=3, a=0, b=1, c=2;
```

The `STRUCT` statement only defines symbolic names for accessing vector members with a fixed position and meaning and another constant which names the entire structure and holds the number of its members.

Classes and objects will be explained in a later section covering the T3X object model.

1.1.5 Factors

This section describes the most basic elements of each T3X program, the *factors* which may be used to form expressions. There are many different kinds of factors: *symbols*, *numeric literals*, *character constants*, *string literals*, *tables*, *procedure calls*, and *messages*. A factor may occur only in expressions and a single factor is the minimum form of an expression. Factors may be prefixed by *unary* operators or they may be combined using *binary* or *ternary* operators. Basically, all sorts of factors are exchangable: where one of them may occur, all others are allowed, too. The only exception is the *symbol* which has some additional properties which make it special. For example, symbols may be subscripted and it is possible to compute their *addresses*. These operations are limited to symbol names. All other operations may be applied to any kind of factor, even if it makes little sense like the multiplication of two strings (which will lead to highly machine- and platform-dependant results):

```
"Hello" * "World"
```

The evaluation of **symbols** depends on their type. Variables and constants evaluate to their values, vectors (including *objects*) evaluate to their addresses. Structure names, *class names*, and structure members are basically constants, too. **Class constants** are public constants which are stored in classes. Like 'ordinary' constants, they evaluate to their values. Names of class constants are glued to the name of the class they are contained in using the dot operator:

```
T3X.BPW
```

Numeric literals are written in decimal notation and represent their own values. A percent sign may be used to negate a number: *%123* = *−123*. The difference between %123 and −123 is that %123 is a factor while −123 is an expression ('minus' applied to a numeric factor). In fact, the percent sign has little meaning in T3X, since the compiler evaluates ordinary minus prefixes in constant expression contexts, too. (In earlier versions of T3X, constant expressions were limited to single factors and therefore, the percent sign was required to define negative constant values. The '`%`' prefix has been kept for compatibility reasons.) An optimizing compiler might turn −*n* into %*n*, if *n* is a constant numeric factor.

Character constants are single characters or *escape sequences* enclosed by single quote characters like

```
'a' '0' '\s' ''' '\\'
```

A character constant evaluates to the ASCII code of the enclosed character. An escape sequence may be used to include certain unprintable or special characters. The backslash character is used to introduce such a sequence. The '`\`' and the following character will be removed and replaced with the associated character. Notice that no escape sequence is required to represent an apostrophy: `'''`. Besides most C-style sequences, the following translations are supported:

```
\e  ESC
\q  "
\s  blank
```

The latter has been included to improve readability: compare `"   "` with `"\s\s\s"`. Unlike **C**, T3X accepts uppercase sequences as well: `'\e'` and `'\E'` both evaluate to ESC. Like in **C**, the escape character may be used to escape itself: `'\\'` evaluates to a single backslash.

String literals are sequences of characters delimited by double quotes (`"`):

```
"Hello, World!\N"
```

Each character either represents itself or is part of an escape sequence as described above. A full machine word will be allocated per character, only the least significant eight bits will be filled with the ASCII code of the character. (In future extensions, the entire space may be used to represent 16-bit *Unicode* characters.) Each string literal is terminated with a NUL character, so *n+1* machine words are required to store a string of the length *n*. Since a string is an array of subsequent machine words, the '`[]`' operator may be used to access its single characters.

When a string is prefixed with the keyword `PACKED`, a byte instead of a full word is used to store each of its characters, and enough NUL characters (but at least one) will be appended to pad the literal to the next word boundary. This way, the proper alignment of subsequent data objects is guaranteed. The byte operator '`::`' may be used to access single characters of packed strings. At runtime, either form of the string literal evaluates to the address of its first character.

BTW: The `#packstrings;` meta command may be used to pack *all* strings in a T3X program. If used, this command should be placed at the beginning of a program.

A more general form of a literal vector is the **table**. A table is an initialized first-class vector. Syntactically, it is a list of *table members* delimited by square brackets:

```
[ 7, "MOD", @modulo ]
```

Each table member occupies exactly one machine word. Vector objects like the string `"MOD"` above are represented by pointers, while the vector object itself is placed outside of the table. Therefore, table members can be accessed using the subscript operator '`[]`'. The expression

```
[ 77,88,99 ] [2]
```

for example, evaluates to *99*. The square brackets have been chosen for delimiting tables because of the connection between vectors and subscript operators.

The type of each table member may be any out of the following list:

• Constant expressions.

• Strings (packed or unpacked).

• Addresses of global data objects.

• Addresses of procedures.

• Tables.

• Embedded dynamic expressions.

Constant expressions include everything which has a value that may be computed at compile time (like numeric literals). The inclusion of strings already has been described above. Addresses of global variables and procedures are represented by a symbol name prefixed with the *address operator* '`@`'

A property that makes tables particulary flexible is the possibility to nest them:

```
[ [ 2, 9, 4 ],
  [ 7, 5, 3 ],
  [ 6, 1, 8 ] ]
```

Like strings, embedded tables are stored outside of the surrounding table and included as pointers. Therefore, if the above table is assigned to the symbol *v*, the following conditions hold:

v[0] = [2, 9, 4]
v[1] = [7, 5, 3]
v[2] = [6, 1, 8]

Since the result of applying a subscript operator to a table containing tables results in a vector again, the subscript operator may be applied one more time, and consequently,

```
v[1][1]
```

would result in 5:

```
v       = [ [2,4,9], [7,5,3], [6,1,8] ]
```

```
v[1]    = [7,5,3]
v[1][1] = 5
```

(Remember that the first element of a vector has an index of 0.)

A table which contains at least one non-constant expression is called a *dynamic table*. Non-constant expressions must be put in parentheses when they are included in a table:

```
v := [ "a * b = ", (a*b) ];
```

Embedded (non-constant) expressions are computed freshly each time the flow of the program passes the table they are contained in. Therefore, the values of table members computed by embedded expressions may be different each time the table is evaluated. This is why such a table is called 'dynamic'. The parentheses tell the compiler that an expression is non-constant and make it generate additional code to fill in the value of the expression whenever the table is encountered. Therefore, static (constant) expressions should never be parenthesized in tables, because inefficient code would be the result.

```
v := [ "5 * 7 = ", (5*7) ];
```

works, but fills in *5*7* each time the table is evaluated. On the other hand, including dynamic expressions in a table without any parentheses will lead to an error:

```
v := [ "a * b = ", a*b ];
```

will not work unless both *a* and *b* are constant.

Like strings, tables may be prefixed with the keyword `PACKED`. Packed tables may contain only byte-values. Therefore, their members are limited to constant expressions with bit patterns where only the least significant 8 bits may contain values other than 0. In numbers, this is the range from −128 to 255.

Strings may be considered a special form of a table. Consequently, each string may be written as a table as well. For example,

```
"T3X"
```

is equal to

```
[ 'T', '3', 'X', 0 ]
```

(Note the trailing zero in the vector literal.) A similar relation exists between packed strings and packed tables. Like packed strings, packed tables will be padded to the next word boundary with zeroes.

The maximum number of members per table may be limited, but at least 128 elements per table are guaranteed to be avaliable. The elements contained in nested tables do not count themselves, but the entire embedded table counts as a single member. The same limit may exist for packed tables and string literals.

Procedure calls are represented by the procedure name followed by a parentheses-enclosed list of zero or more comma-separated arguments:

```
printstring("Hello, World!")
```

Each argument may be any valid *expression*. When a procedure expects zero arguments, the parentheses still must be supplied: *P()*. A procedure call evaluates to the *return value* of the called procedure. If a procedure does not return through a `RETURN` statement, its value defaults to zero.

In T3X, only procedures may be called. Calls to absolute addresses and computed calls are not allowed. There is a mechanism to perform *indirect calls*, though. The `CALL` operator allows to call a procedure whose address is stored in a variable:

```
p := @printpacked;
CALL p(packed "Hello, World!\n");
```

For several resaons, the `CALL` operator should be treated with special care: The argument count in the call will not be checked. Due to the T3X calling conventions, a wrong number of arguments will in normally lead to an undefined result. The result is also undefined, if the variable in the `CALL` statement does not point to a valid procedure. The only way to obtain a valid procedure pointer is the application of the address operator '`@`' to a procedure symbol.

When the symbol following the keyword `CALL` references a procedure rather than a variable, the `CALL` operator will be ignored completely, and a type-checked direct call will be generated instead.

Messages are used to apply methods to objects. Applying a method is basically the same as calling a procedure. The only difference is that in a message, an additional object context is passed to a procedure. Like in references to class constants, this is done using the dot operator. The statement

```
printer.print("some text")
```

would send the message *print* with the argument *"some text"* to the object *printer*.

Methods may be sent to variables containing references to objects. The following statements would send the same message as the above factor (given that *print* is an instance of the class *printer_interface*):

```
o := @print;
SEND (o, printer_interface, print("some text") );
```

The `SEND` operator requires three arguments: a variable *o* containing the address of a valid object, the class of the object *o*, and a procedure call calling a method of the class of *o*. The same caveats apply to the `SEND` and the `CALL` operators. Additionally, the second argument of the `SEND` operator *must* match the class of the object its first argument points to. If this is not the case, the result of the `SEND` operation is undefined.

1.1.6 Expressions

In expressions, operators may used to modify or combine factors in various ways. Most operators may be applied to any kind of operand, even if the resulting operation does not evaluate to any meaningful value. There are different kinds of operators and like procedures, they are classified by the number of their arguments (called *operands* in this context). There are *unary* (prefix) operators, *binary* (infix and postfix) operators, and there is one *ternary* operator and one *variadic* operator.

Another way to classify operands is by their *precedence*. The higher the precedence of an operator is, the stronger it *binds* to its operands. For example, the term operators (multiplication, division, modulo) bind stronger than the sum operators (addition, subtraction). Therefore,

```
a * b + c * d
```

is equal to

```
(a*b) + (c*d)
```

Like in math expressions, parentheses may be used to override these default bindings. These are the complete precedence rules of T3X:

1. Postfix operators bind strongest.

2. The less operands an operator has, the stronger it binds.

3. The ascending order of precedence of the binary operators is as follows: disjunction (1), conjunction (2), equational (3), relational (4), bit (5), sum (6), and term (7) operators.

The precedence rules in 3. are similar to the rules used in the evaluation of algebraic math expressions.

Another property of an operator is its *associativity*. An operator is left-associatve, if a sequence of equal operations is evaluated from the left to the right. It is right-associative, if such a sequence evaluates from the right to the left.

Associativity	**Expression**	**Meaning**
left	a X b X c	(a X b) X c
right	a X b X c	a X (b X c)

Table 1: Associativity

Table 1 illustrates left and right associativity. With the exception of '`::`' and the unary operators, all T3X operators are left-associative. The byte operator and the unary operators associate to the right.

In the remainder of this section, all availabe operators will be explained. The appearance is ordered by descending precedence.

Procedure Calls and Subscripts

The operators '`()`', '`[]`', and '`::`' are the only postfix operators. They are always bound to primary factors in the form of symbols and this asspciation cannot be overridden using parentheses. Subscripts and call operators may be considered *part of* a factor rather then an operator applied to a factor.

The '`()`' operator is the only variadic operator. Given the procedure call

```
P(a1, ..., aN)
```

its *arity* (the number of its operands) is *N+1* (*P* plus *N* arguments). The meaning of the operator is the application of the procedure *P* to the (optional) arguments *a1* through *aN*. If *P* does not have any formal arguments, its syntax is

```
P()
```

The value of the operation depends on the semantics of *P*. The '`()`' operator may be applied only to symbols of the type 'procedure' (this type does include methods). The procedure must have been declared before its first application using either a procedure definition, declaration, or class dependency. The number of arguments to a procedure call will be checked against the arity of the called procedure. If the numbers do not match, an error will be signalled.

Each argument to a procedure call may be any valid expression itself which includes, of course, procedure calls. Given the binary function *P2*, the following expression is perfectly valid:

```
P2( P2(1, 2), P2( 3, P2(4, 5) ) )
```

An *indirect procedure call* may be performed using the `CALL` operator which is in fact an extension to the '`()`' operator:

```
CALL PP(a1, ..., aN)
```

Like in the previous example, this expression evaluates to the result of the application of *PP* to *a1...aN*, but in this case, *PP* is a *procedure pointer* instead of an actual procedure. A procedure pointer is an ordinary variable which has been assigned the address of a procedure using the address operator:

```
PP := @P;
```

In indirect procedure calls, no type checking will be performed. If *PP* is the name of a procedure instead of a variable, the keyword `CALL` will be ignored.

In direct and indirect calls, the calling scheme is *call by value*. This means that the arguments to the call will be evaluated *before* the call actually takes place and the *value* of each parameter expression will be transported to the procedure.

Since vectors evaluate to their addresses, passing a vector by value will actually pass a reference to the array associated with the vector symbol. Therefore, vectors are in fact always passed *by reference*. Instead of passing the entire vector, only the address of its first member is transported to the called procedure. The callee stores this address in an (atomic) parameter variable. Since parameters are always atomic (and therefore evaluate to their values) and vectors evaluate to their addresses, both the original vector and the parameter will reference the same memory location and the parameter may be used in the same way as a vector.

Applications of methods work in exactly the same way as procedure calls. The same operator is used to apply methods to objects, but the procedure call syntax is 'glued' to an object using the (binary) '`.`' operator (the dot operator which will be explained in detail in a later section). Like procedures, messages take any expressions as arguments. Consequently, messages may be nested. The expression

```
o.m( o.m(5, 7), o.m(12, 19) )
```

sends the message *m(5,7)* and *m(12,19)* to *o* and uses their results as arguments to another message to *o*.

In analogy to the `CALL` operator, the `SEND` operator may be used to send a message to a *pointer to an object*. The expression

```
o.m(x,y)
```

is perfectly equal to the statements

```
 po := @o;
SEND (po, C, m(x,y) );
```

where *C* is the class of *o*.

The **subscript operator** '`[]`' may be applied to vectors as well as to atomic variables:

```
symbol [subscript]
```

where *subscript* may be any valid expression. If the symbol is a vector, the subscript operation

```
a[b]
```

evaluates to the *b*'th member of *a*. If *a* is an atomic variable, the operation evaluates to the *b*'th member of the vector *a* points to. This means that both subscripts in the following example would evaluate to the same value:

```
var    v[100], pv;
var    a1, a2;

pv := v;
a1 := v[25];
a2 := pv[25];
```

The reason is simple. Since vectors evaluate to their addresses, the assignment

```
pv := v;
```

stores the address of *v* in *pv*. Atomic variables, however, evaluate to their values and therefore, *pv* evaluates to the address of *v* which has been previously stored in it. Consequently, *v* and *pv* both evaluate to the address of *v* in the above example. Hence, a variable which holds the address of a vector may be used in place of that vactor. This is the reason why the subscript operator may be applied to atomic variables as well.

Since there is no nesting limit for vectors, any number of subscript operators may follow a single symbol. Assuming that the variable *v5* denotes a vector containing five levels of nested vectors, the expression

```
v5[i1][i2][i3][i4][i5]
```

may be used to access single elements at the deepest nesting level. Since the '`[]`' operator is left-associative, such chains of subscripts evaluate from the left to the right.

The **byte subscript operator** '`::`' differs from the ordinary (word) subscript operator in several ways. Firstly, it addresses *bytes* in (byte)vectors and secondly, it associates to the right. The expression

```
a::b
```

evaluates to the *b*'th byte contained in the vector *a*. Therefore, '`::`' is mostly used to access characters in packed strings. Since the results of `::`-operations are always limited to byte-width, they cannot be assumed to return valid addresses. For this reason, byte subscripts are right-associative. If the expression

```
a :: b :: c
```

would evaluate from the left to the right

```
a::b :: c
```

the result of *a::b* would probably not be a valid address, since it is limited to eight bits. In this case, however, the following subscript would possibly reference the position *c* of a non-vector − which is certainly not the desired result. If the expression evaluates from the right to the left

```
a :: b::c
```

however, the subexpression *b::c* is evaluated first and will probably return a valid subscript. This subscript is then applied to the (also valid) vector *a*.

Finally, the '`::`' operator differs from '`[]`' in the point that there is no right delimiter for the subscript expression. Therefore, the right side of '`::`' is always a single factor and expressions like

```
a::b+c
```

actually evaluate to

```
(a::b)+c
```

since '`::`' has the highest precedence. To address the *b+c*'th byte in the array *a*, the subscript must be parenthesized:

```
a::(b+c)
```

Remember that it is impossible to change the associativity of subscript operators using parentheses.

Unary Operators

All unary operators have a high precedence and apply to single factors. Unless explicitly specified using parentheses, they never affect subexpressions containing other operators except for postfix operators which have an even higher precedence. Suffix operators bind stronger than prefix operators, because this order leads to much more sensible semantics. For example

```
-P(a,b)
```

means 'negate the result of applying *P* to *a* and *b*' and

```
~v[j]
```

means ‘evaluate to the inverse value of the *j*’th member of *v*’. If the order of precedence would be reverse, the meaning of the first example would be ‘apply whatever is at the negative address of *P* to *a* and *b*’ and the second one would mean ‘evaluate to the *j*’th member of the vector located at the inverse value of *v*’.

Altogether, there are four prefix operators. The minus sign ‘−’ (which exists as a binary operator, too) evaluates to the negative value of its operand. Like in math, any even number of minus signs has no effect. The unary minus sign is distinguished from the binary ‘−’ by its context. When the sign occurs between two operands, it is binary. If it occurs at the place of a factor, it is unary and the factor itself follows.

The tilde operator ‘`~`’ results in the value of its operand with all bits inverted. Since inverting a bit twice always yields the original value, even numbers of ‘`~`’ operators have no effect, either.

The backslash ‘`\`’ represents the logical NOT (while ‘`~`’ represents the bitwise NOT). This operator evaluates to the normal form of the *true* value (−1), if its operand is *false* (0) and vice versa. Only the zero value is considered ‘false’ in T3X and all non-zero values are considered ‘true’. The normal form of the ‘true’ value is −1. Two (or any even number of) subsequent logical NOT operators may be used to create the normal form of a truth value.

The address operator ‘`@`’ evaluates to the address of its operand. Therefore, it may only be applied to symbol names. The addresses of constants, structures, and classes may not be computed using ‘`@`’, because constants and templates have no addresses, but only compile-time values. Since the subscript operators bind stronger than the address operator, ‘`@`’ can be used to compute addresses of vector and structure members, and even the addresses of members of nested tables:

```
@v[i][j]
```

computes the address of the *j*’th member of the embedded vector *v[i]*. The address operator may be combined with byte subscipts, as well:

```
@s::i
```

yields the address of the *i*’th byte in *s*.

Term Operators

The operation `A*B` evaluates to the product of *A* and *B*. If $A \cdot B$ does not fit in a machine word, the result is undefined.

`A/B` results in the integral part of the quotient of *A* and *B*. The result is undefined, if *B* is zero.

All term operators respect the signs of both of their operands. Two equally signed operands yield a positive result and operands with different signs lead to a negative result. However, T3X also provides some *modified* operators which work on unsigned values. All modified operators are prefixed with a dot. The operation `A.*B` evaluates to the product of the unsigned values *.A* and *.B* (*.X* is used to denote the unsigned value of `X`) and `A./B` results in the integral part of the quotient of *.A* and *.B*.

`A MOD B` evaluates to the difference between *A* and *A./B.*B* where *A./B*. Therefore, `A MOD B` is the division remainder of `A./B`. Like '`/`', `MOD` leads to an undefined result, if *B=0*.

Sum Operators

`A+B` evaluates to the sum of *A* and *B* and `A-B` evaluates to their difference.

Bit Operators

In T3X, all bit operations have the same precedence. The grouping of such operations must be made explicitly using parentheses. Otherwise, evaluation is performed from the left to the right. The operation `A&B` results the bitwise AND of *A* and *B*. Each bit is the result of computing the logical product of one bit in *A* with the bit at the same position in *B*.

`A|B` yields the result of performing a bitwise OR on *A* and *B*. Each bit in the result is a logical sum of a bit in *A* and the bit at the same position in *B*.

`A^B` performs a bitwise exclusive OR (XOR). In this case, the computation of a single bit is done by combining bits at the same positions in *A* and *B* using a logical negative equivalence operation.

A	B	AND (·)	OR (+)	XOR (≠)
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Table 2: Bitwise Logical Operations

See table 2 for the results of applying logical operations to pairs of bits.

`A<<B` evaluates to the value of *A* with all bits shifted to the left by *B* positions. This is the same as an *unsigned* multiplication with the *B*'th power of 2:

$$a << b \;=\; a \cdot 2^b \textit{ where } 0 \le (a << b) < 2^{BPW-1} \wedge a = |a| \wedge b = |b|$$

Naturally, this equivalence is only given, if the process of shifting *A* is guaranteed not to change the sign bit of *A*. Otherwise, the sign of the result must be considered undefined. (This is not relevant, of course, if *A* is used as a *bit field* where each bit represents a binary state.)

`A>>B` yields the result of shifting the bits in *A* to the right by *B* positions. Under certain conditions similar to those described above, this result equals the quotient $\frac{A}{2^B}$.

Technically speaking, one might say that the shift operators in T3X perform *bitwise* instead of *arithmetic* shift operations. This implementation has been chosen, because manipulating bit fields is the main purpose of these operators and it is hard to manipulate bit fields using arithmetic shift operators.

Relational Operators

Relational operators are used to *compare* two operands. The relation between the operands is expressed as a truth value: all these operators return *true*, if their meanings apply to their operands and otherwise *false*. A summary of the relational operators available in T3X is given in table 3. *.X* is used to denote the unsigned value of *X*.

Operator	**Meaning**
`A < B`	A is less than B
`A > B`	A is greater than B
`A <= B`	A is less than or equal to B
`A >= B`	A is greater than or equal to B
`A .< B`	.A is less than .B
`A .> B`	.A is greater than .B
`A .<= B`	.A is less than or equal to .B
`A .>= B`	.A is greater than or equal to .B
`A = B`	A is equal to B
`A \= B`	A is not equal to B

Table 3: Relational Operators

Notice that the operators expressing *equivalence* (‘=’ and ‘\=’) have a lower precedence than operators expressing ordering. Therefore,

```
A < B = C < D
```

is equal to

```
(A<B) = (C<D)
```

Consequently, the equation sign may be interpreted as ‘logical equivalence’ when used between comparisons: the above expression evaluates to true, if either

(A<B) AND (C<D)

or

\*(A<B) AND* \*(C<D)*

applies. Since the inequation operator ‘\=’ has the same precedence as ‘=’, it may be used for a negative logical equivalence operator (aka Exclusive OR):

```
A<0 \= B<0
```

becomes true, if either *A* or *B* is negative (but not both of them). If the truth values of the comparisons *A<0* and *B<0* are equal, the expression yields the result ‘false’.

Notice again, that any value may be considered a *truth value* in a typeless language like T3X. Basically, *everything* but the value *zero* is interpreted as ‘truth’, and only 0 will be taken for a ‘false’ value.

Conjunctions and Disjunctions

The operations `A/\B` and `A\/B` represent the logical conjuction (AND) and disjunction (OR). Generally speaking, the expression

```
A /\ B
```

evaluates to *true*, if and only if *A AND B* evaluate to *true*.

```
A \/ B
```

yields a *true* result if and only if *A OR B* (or both of them) evaluate to *true*.

More specifically, `/\` and `\/` are so-called *short circuit operators*. Since the expression `A/\B` can lead to a ‘true’ result only if *all* its operands are ‘true’, there is no actual need to evaluate *B*, if *A* already has evaluated to a ‘false’ value. Therefore, the second operand of a conjunction will never be evaluated by a T3X program, if the first one is false. The result will be zero in this case. If, on the other hand, the first value is ‘true’, the result of the entire conjunctional expression will be equal to the value of the second operand. Therefore, the result of

```
A /\ B
```

can be specified more precisely as:

0, if A=0
B, if A\=0

In analogy to above case, the expression `A\/B` can never become false, if *A* already has been found out to be true. Therefore, no T3X program will ever evaluate *B* in such a case, and the result of the disjunction

```
A \/ B
```

can be defined more precisely as

A, if A\=0.
B, if A=0.

Like in common algebra, conjunctions bind stronger than disjunctions:

```
A /\ B \/ C /\ D
```

equals

```
(A/\B) \/ (C/\D)
```

In chains of equal logic operations, the order of evaluation is from the left to the right (as for all binary operators). This means that chains of conjunctions will be evaluated up to the first ‘false’ occurance and chains of disjunctions will be processed up to the first ‘true’ occurance. In either case, the result of the entire chain is the value of the last processed operand.

There is a connection between the logical operators and conditional statements (which will be explained later): Because of the short circuit evaluation, logical operators may be used to implement some basic *flow control* within expressions. The expression

```
A /\ B()
```

has almost the same meaning as

```
IF (A) B();
```

The only difference is that the expression yields a value, while the statement only has a *side effect* (defined by the semantics of *B*). Likewise, the expression

```
A \/ B()
```

has a meaning similar to

```
IF (\A) B();
```

Conditional Expressions

The ternary conditional operator has the least precedence. Therefore, it may be used to combine any kind of expressions without using parentheses. The following expression, for example, implements the *minimum* function:

```
a<b -> a : b
```

Since the operator has three operands, it consists of two parts: '`->`' and '`:`'. The meaning of the conditional operator is as follows: Given the expression

```
A-> B: C
```

if the operand *A* (the condition) evaluates to 'true', *B* will be evaluated and otherwise, *C* will be evaluated. If *B* is evaluated, *C* will not be evaluated and vice versa. The result of the expression is equal to the last evaluated operand.

Like the logic operators '`/\`' and '`\/`', the conditional operator has a connection to conditional statements:

```
A-> B(): C()
```

is equivalent to

```
IE (A) B(); ELSE C();
```

except for the fact, of course, that the expression has a value, while the statement only has a side effect. (`IE` is short for If/Else and introduces a conditional statement with an alternative).

1.1.7 Constant Expressions

Constant expressions are used wherever a value must be known at compile time. Only a limited set of operators is allowed in constant expressions and the order of evaluation is *always* from the left to the right. There are no precedence or associativity rules, and so

```
L+1*10
```

evaluates to *(L+1)*10* and not to *L+(1*10)* as it would in ordinary expressions. The resons for this decision were 1) ease of implementation and 2) the fact that most conditional expressions contain only a single operator. Because of the lack of precedence rules, parentheses are not required in constant expressions. Most orders of evaluation can be specified by ordering the operations appropriately.

Note: *The decision to simplify constant expression is the way described above has turned out to be a bad idea, since two sets of rules for evaluating expressions have to be kept in memory (the memory of the programmer, that is). For compatibility reasons, this approach has been kept, though.*

The following operators are recognized inside of constant expressions:

'`+`' is frequently required to increase the lengths of arrays. For example, if a buffer has to be used as a string, an additional character must be appended to hold the delimiting NUL character:

```
VAR buffer::BUFSIZE + 1;
```

'`*`' is required for allocating memory for arrays of structures:

```
VAR Points[point * NUM_OF_POINTS];
```

'`|`' is useful when creating constant bit maps:

```
CONST A = 8, B = 16, AorB = A | B;
```

'`~`' is also useful for creating constant bit maps:

```
CONST SignBit = -32768, ValueMask = ~SignBit;
```

The *unary* '`-`', finally, can be used to negate constants. This is particularly useful when counting down in `FOR` loops where the step width must be constant:

```
FOR (i=99, -1, -STEPWIDTH) p();
```

1.1.8 Statements

Statements are the basic building-stones of T3X programs. While expressions evaluate to a value, statements are used to 'tell the program to do something'. Each program is a list of 'commands' which is executed in sequence. Each command is also called a *statement* in the terminology of imperative programming. There are different kinds of statements in T3X: *assignments*, *procedure calls*, *messages*, *conditional statements*, *loop statements*, *branch statements*, and *compound statements*. The assignment is an essential part of every imperative language. It is even frequently used to characterize the imperative approach. Compound statemements do not have an own meaning, but they are used to group statements to form the *bodies* of loops, conditionals, and procedures. All other statement types serve the control of the program's flow.

In T3X, all statements have to be *terminated* with a semicolon. This means that a semicolon must follow every statement in a program, except for compound statements which are delimited by the keywords `DO` and `END`. In other procedural languages (like BCPL and Pascal), statements are *separated* rather than terminated. In such languages, a delimiter has to be used only, if two statements are written in sequence – there may not be a delimiter after the last statement. T3X uses the simpler form of combining statements: Each non-compound statement has to be terminated.

Assignments

An assignment is used to transfer the value of an expression to a specific storage location. For example, the statement

```
A := B;
```

copies the value of *B* to *A*. After the assignment, both variables have the same value. The previous value of *A* is thereby lost. The righthand side of an assigment may be any valid expression as described in the previous section. The left side, however, is restricted to a subset of expressions which is frequently referred to as *lvalues* (lefthand side values). In T3X, lvalues may be any out of the following list:

- Atomic variables.
- Vector members.
- Byte vector members.
- Structure members.

(Of course, vector members and structure members are basically the same.) Assignments to vector members are in no way limited: addressing elements of multiply nested vectors is perfectly legal. In the section about factors, the evaluation of variables on left and righthand sides of assignments has been explained: On right sides sides of assignment operators, variables evaluate to their *values* and on left sides, they evaluate to their *addresses*. The assignment operator '`:=`' first evaluates the lefthand side value and remembers the resulting address. Then, if evaluates the righthand side expression and stores its value at the memorized address.

A generalization of the evaluation of lvalues is the following: All but the last *reference* in an lvalue evaluates to its value. Only the last reference evaluates to its address. Here are some examples:

```
A := B;
```

The symbol *A* references a specific storage location. Since it is the last reference on the left side, it evaluates to its address. In the statement

```
A[i] := B;
```

however, *A* is not the last reference and hence, it yields its value (which *is* its address, if *A* is a vector). The operation `[i]` references the *i*'th member of *A*. Since it is the last reference on the left side, it evaluates to the address of *A[i]* instead of its value. Consequently, the following assignment operator stores *B* in this address − the *i*'th member of *A*. The same is valid for the access of vector elements at any nesting level. The statement

```
A[i1][i2][i3][i4] := B;
```

for example, stores *B* in the *i4*'th member of `A[i1][i2][i3]`.

Accessing byte vectors works in exactly the same way:

```
A::i := B;
```

stores the least significant eight bits of *B* in the *i*'th byte of *A*. Since '`::`' associates to the right, the last evaluated reference is the leftmost one in expressions like

```
A::B::i := C;
```

Because *B::i* will be evaluated first in this example, it will yield its *value*. Then, the address of *A::(B::i)* is computed since no more references are following after *A::*. Finally, the value of *C & 255* will be stored in the *B::i*'th byte of *A*.

Note: Although the assignment symbol '`:=`' is an operator, it may not be used in expressions. The occurance of this operator automatically turns an expression into a statement.

Procedure Calls

The application of a procedure may be turned into a complete statement by appending a semicolon to it:

```
newline();
```

In this case, the return value of the called procedure will be discarded and only the side effects of the procedure will actually take effect. (The side effect of the above statement, for example, could be the output of a newline sequence.) Each procedure – no matter whether it explicitly returns a value or not – may be used in a standalone procedure call. Like in procedure calls in expression contexts, the arguments may be any valid expressions. The `CALL` operator may be used in standalone statement as well. For details on direct and indirect procedure calls, see the the sections on *factors* and the '`()`' operator in earlier parts of this book.

Messages

Like procedure calls, messages may form complete statements, too:

```
t.memfill(region, 0, 64);
```

The above statement sends the message *memfill(region, 0, 64)* to the object *t*. (If *t* is an instance of the *T3X* class, this message will fill 64 bytes of the vector *region* with zeroes.) The return value of the message is discarded. The `SEND` operator may be used to form a statement as well. See the sections on messages and '`()`' for details.

Conditional Statements

There are two forms of the conditional statement, the first one being the `IF` statement, which is avaliable in most procedural languages. Its general syntax is

```
IF (expression) statement
```

where *expression* may be any expression and *statement* may be any statement. The `IF` statement itself does not have to be terminated with a semicolon, since its body (which is a statement, too), already supplies the terminating semicolon. The statement which forms the body of the `IF` statement will be executed, if and only if *expression* evaluates to a 'true' (non-zero) value. The following statement turns the variable *a* into its absolute value:

```
IF (a < 0) a := -a;
```

If a is less than zero, *then a* will be loaded with $-a$, thereby changing its sign. Since the body

```
a := -a
```

is executed only if $a<0$ applies, this conditional statement always leaves a positive value in *a*. Notice that the semicolon in the above example belongs to the *assignment*.

The second form of the conditional statement has an additional alternative part:

```
IE (expression) statement-T ELSE statement-F
```

Like in `IF` statements, any valid expression and statements may be used in the places of *expression*, *statement-T* and *statement-F*, respectively.

The meaning of the `IE` statement is the same the one of the `IF` statements as long as the *expression* evaluates to 'true'. In this case, the first statement (*statement-T*) will be executed. If the expression evaluates to 'false', however, *statement-F* will be executed. An `IF` statement would not have any effect in this case.

`IE` is an abbreviation for **I**f/**E**lse. In most languages, the **if** statement may or may not have an alternative branch. In T3X, there is a separate keyword for each version. The reason for this design choice is the so-called 'dangling else' problem which cannot arise when separate statement types are provided for the conditional statement with and witout an alternative branch. If no further information is supplied, the program in example 2 (written in a language which allows optional alternatives) would be ambiguous.

```
if (condition1)
    if (condition2)
        statement1
else
    statement2
```

Example 2: Dangling ELSE

The problem is to decide to which **if** the **else** branch belongs: is it the alternative of *IF (condition1)* or *IF (condition2)*? The indentation in this example suggests that it belongs to the first **if**. In fact, however, most languages will bind it to the most recently opened **if** – the second one in this example. In T3X, such an ambiguity does not exist.

Since the `IF` statement cannot have an alternative part, the `ELSE` branch must belong to `IE (condition1)` in example 3.

```
IE (condition1)
    IF (condition2) statement1
ELSE
    statement2
```

Example 3: An Embedded IF in T3X

BTW: The idea of separating `IF` and `IE` is not new. In BCPL, for example, `IE` is called **TEST**.

Loop Statements

There are two kinds of loops: ‘while’ loops and ‘for’ loops which represent two classes of problems: those which are computable by algorithms with a known upper limit of iterations (FOR-computable or primitive recursive functions) and problems which cannot be computed by algorithms with a fixed number of iterations (WHILE-computable or general recursive functions). Since the FOR-computable functions are a subset of the WHILE-computable ones, `FOR` statements may be considered a special case of `WHILE` statements and in fact, it is possible to express a `FOR` loop using `WHILE`, but not vice versa. (In T3X and other procedural languages, it generally *is* possible, but theoretically, it is not.)

There is a third kind of loop in many other languages, the *repeating loop*, but it turns out to be a special case of the `WHILE` loop. Repeating loops are not very frequently needed and if they are, they can easily be simulated using `WHILE` and `IF` in T3X.

The `WHILE` loop has the following general form:

```
WHILE (expression) statement
```

where *expression* may be any expression and *statement* may be any statement. The body consisting of the statement will be repeated *as long as* the test expression in parentheses evaluates to a ‘true’ value. If the expression evaluates to ’false’ before the statement has been executed for the first time, it will never be executed. However, a loop which tests its exit condition at its end may be constructed using `WHILE`, `IF`, and a *compound statement* (which will be explained later in this section). A skeleton for a repeative loop can be found in example 4.

```
WHILE (-1) DO    ! -1 is always true
    statement
    IF (\condition) LEAVE;
END
```

Example 4: A Repeative Loop in T3X

In this example, *statement* will be executed at least once, because the exit condition -1 is a ‘true’ constant. In the following `IF` statement, the loop will be left if *condition* does *not* apply. `LEAVE` is used to branch out of the loop. It will be explained later in this chapter.

The FOR loop exists in two forms: an explicit form and a short form. This is the explicit form:

```
FOR (var=start, limit, step) statement
```

Var is an atomic variable which must have been declared before (other languages may declare index variables implicitly). *Start* and *limit* are expressions and *step* is a constant expression. The FOR loop works as follows: First, *var* is initialized with the value of *start*. Second, *var* is compared against *limit*. If either the condition

```
var>=limit /\ step>0
```

or

```
var<=limit /\ step<0
```

holds, the loop is left and *statement* will not be executed any more. If none of the above conditions holds, the loop statement is executed, *step* is added to *var*, and the loop will be repeated from the point where the exit condition is checked (the second step in this description). In analogy to a WHILE loop, the statement will never be executed, if either of above exit conditions already is true at the first time it is checked.

The following example calls the procdure *P* with the arguments 0 through 9:

```
FOR (i=0, 10, 1) p(i);
```

And this example counts down from 9 to 0:

```
FOR (i=9, -1, -1) p(i);
```

Special attention should be paid to the limits of the FOR loops in these examples. They always specify the first value which will *not* be applied to the statement. Another way to write the second example would be the following one where the FOR loop has been replaced with a WHILE loop:

```
i := 10; WHILE (i>-1) DO
        p(i);
        i := i-1;
END
```

The meaning of this program fragment is completely equal to the above one, but the syntax of the FOR statement is more compact and expresses the purpose of the statement clearer. This is another reason for the inclusion of the FOR statement in the T3X language.

The *step* value is optional in FOR statements. In the short form of the statement, it is omitted. If only two operands are given to FOR, the step width defaults to *one*. Therefore, the statements

```
FOR (j=0, 100, 1) p(i);
```

and

```
FOR (j=0, 100) p(i);
```

have exactly the same meaning.

Branch Statements

A ‘branch’ passes control to a specific point in a program. Typical destinations for branch commands are the beginnings and ends of loops and the ends of procedures and programs. There is no branch command with a freely definable destination like **goto** in C.

The `LEAVE` command, which already has been used in the previous subsection, causes the immediate termination of the innermost `WHILE` or `FOR` loop. There are no operands to `LEAVE`.

The following code compares the first 100 characters of two strings *A* and *B*. It stops at the first position where they differ, but in any case after 100 steps:

```
FOR (i=0, 100) IF (a::i \= b::i) LEAVE;
```

The loop is set up for 100 passes and the conditional `LEAVE` statement makes the loop terminate, if a mismatch has been found.

The `LOOP` command transfers control to the beginning of the innermost loop. Like `LEAVE`, it has no operands. If `LOOP` is used inside of a `FOR` loop, it branches to the increment part where the value of the index variable is modified. In `WHILE` loops, it branches directly to the point where the exit condition is checked.

To leave a procedure, a `RETURN` statement may be used. It has the general forms

```
RETURN expression;
```

and

```
RETURN;
```

The statement evaluates the specified expression, if any, and passes its value back to the calling procedure. In order to return to the caller, it performs a branch to the end of the current procedure, where local storage is released and the procedure is left. The value received by the calling procedure is the value of *expression*. (See example 5).

```
P(x) RETURN x*x;

Q() DO VAR y;
        y := P(5);
END
```

Example 5: A Simple Procedure

In this sample program fragment, 5 is passed as an argument to the procedure *P*. The procedure computes the square of its argument and returns it to *Q* where the result will be stord in *y*.

When the expression part of `RETURN` is omitted, the return value defaults to zero. A return statement without an explicit value may be used to denote that a procedure has no specific return value. At the end of a procedure,

```
RETURN;
```

would be redundant, since every procedure returns zero when the flow of control reaches the end of its body.

All the above branch statements take care of locally allocated storage. If local symbols are defined in the bodies of loops, for example, `LOOP` and `LEAVE` will release this storage before performing their branches. This allows the use of these commands in any loop context, even if local symbols are present. In a similar way, `RETURN` will release all local storage of the current procedure before branching to its end.

The last form of the branch is the `HALT` statement, which has the general forms

```
HALT;
```

and

```
HALT const-expression;
```

It performs a branch to the end of the entire program, thereby terminating it. If necessary, the command cleans up the runtime environment of the program so that it can terminate gracefully. When a constant expression is specified, its value will be returned to the calling process, if supported by the host operating system. When no value is specified, zero is returned.

Compound Statements

A compound statement (sometimes also called a *block statement*) is a group of statements which is treated like a single statement under some aspects: a compound statement may occur at any place where an

```
WHILE (expression) statement
```

In such situations, a compound statement can be used to ‘extend’ the scope of the loop in such a way that it is applied to a group of statements instead of a single statement.

```
v := 0;
c := s::0;
p := 1;
WHILE ('0' <= c /\ c <= '9') DO
    v := v*10 + c - '0';
    c := s::p;
    p := p+1;
END
```

Example 6: Extended Scope of a Loop

In example 6, three assignments take place inside of the loop. (The example computes the value of the decimal number represented by the string `s`.) The keywords `DO` and `END` are used to delimit statement blocks. There is no terminating semicolon after a compound statement. The line

```
DO p(); q(); END ;
```

would be recognized as a compound statement containing the procedure calls *P()* and *Q()* and a separate *empty statement* consisting of a single semicolon.

In T3X, compound statements are ordinary statements and they may occur at *any* place where a statement is expected. Even statements like

```
DO DO DO END END END
```

are perfectly valid. The use of compound statements in sequences becomes clear later in this chapter when the allocation of local storage in compound statements is explained.

Empty Statements

There are two forms of the empty statement (aka *null statement*) in T3X, The first being the single semicolon

```
;
```

and the second one the empty compound statement

```
DO END
```

Both null statements have absolutely no effect. Their only purpose is to fill a gap where a statement is required, but nothing is to do. They are useful to negate the meanings of complex conditions, for example. Instead of negating the condition at the cost of making it harder to understand, one might turn

```
IF (complex-condition) statement
```

into

```
IE (complex-condition)
    ;
ELSE
    statement
```

1.1.9 Local Storage

Besides the grouping of commands, compound statements provide a mechanism for the definition of *local symbols* and the allocation of *dynamic memory*. Declaration statements already have been explained in a previous section. All data objects which can be created in T3X also may be declared *locally* inside of compound statements by placing their declarations at the beginning of a statement block. Any number of declarations will be accepted after the keyword `DO` which introduces the block. The declaration statements themselves do not change in this case. Only the position inside of a statement block makes the declared symbols local to that block. The statement

```
DO VAR i; FOR (i=0, 10) p(i); END
```

for example, applies the procedure *P* to the sequence *0...9*. The index variable is declared inside of the compound statement which also contains the `FOR` loop which generates the sequence. The variable *i* does not exist before the compound statement is entered. It will be created automatically at the point of its declaration and it will cease to exist at the end of the block it has been declared in. Therefore, variables which are local to compound statements are sometimes also called *automatic variables* , but the more common term is *local variables*.

Besides variables and vectors, structures and constants may be declared locally, too. T3X does not support nested procedure definitions, though. In case of atomic variables and vectors, the storage required by the variables is allocated at the point where the symbol name becomes valid and released when the variable is destroyed again. In most environments, automatic storage will be allocated in a stack structure called the *runtime stack*. The main purpose of local symbols is the definition variables local to procedures. For another application of local storage allocation, imagine the situation described in example 7

```
P() DO
    VAR     big_V[LARGE_1], big_W[LARGE_2];

    task1(big_V);
    task2(big_W);
END
```

Example 7: Local Storage Allocation

where two tasks requiring large amounts of storage shall be run sequentially in the same procedure, but not enough memory for *both* arrays is available. One solution – of course – would be the creation of two procedures where each one creates local storage for its own task. Another one would be to share the vector, but both solutions only work at the cost of readablility and maintainability.

```
P() DO
    DO VAR big_V[LARGE_1];
        task1(big_V);
    END ! big_V gets released here

    DO VAR big_W[LARGE_2];
        task2(big_W);
    END ! big_W gets released here
END
```

Example 8: Storage Allocation in Statement Blocks

T3X provides another solution, since the compiler guarantees that local storage is allocated exactly at the point of its declaration and released immediately at the point of the destruction of the associated symbol (see example 8).

Since compound statements may be nested, naming conflicts may occur, like example 9 (in **C**) illustrates.

```
{ int i;
        i = 123;
        { int i;
                i = 456;
        }
        printf("%d\n", i);
}
```

Example 9: Shadowing in C

The variable *i* which is defined in the outer compound statement (delimited by **{** and **}**) is redefined in in the inner block. Inside of the inner block, the variable *i* gets assigned the value 456. Clearly, the assignment *i=123* in the outer block references the variable defined in the outer block, but which one is referenced in the inner scope? **C** – like most other procedural languages – resolves this ambiguity by always giving precedence to the innermost definition. Therefore, the example program fragment would print 123. When this method is used, the symbol *i* defined in the outer scope becomes inaccessible in the embedded scope. This effect is called *shadowing*: The inner definition 'shadows' the outer one which thereby becomes temporarily invisible to the compiler.

T3X uses more strict scoping rules than most other languages: Symbols generally may not be redefined in T3X programs. These rules also apply to *global symbols* (symbols which have been declared at the *top level*, outside of procedure definitions or statement blocks). This way, shadowing can never happen. The flexibility of local symbols remains, though, since names can be reused as soon as a local data object is getting destroyed:

```
F(x,y) DO VAR i, j; ... END

G(x,y) DO VAR i, j; ... END
```

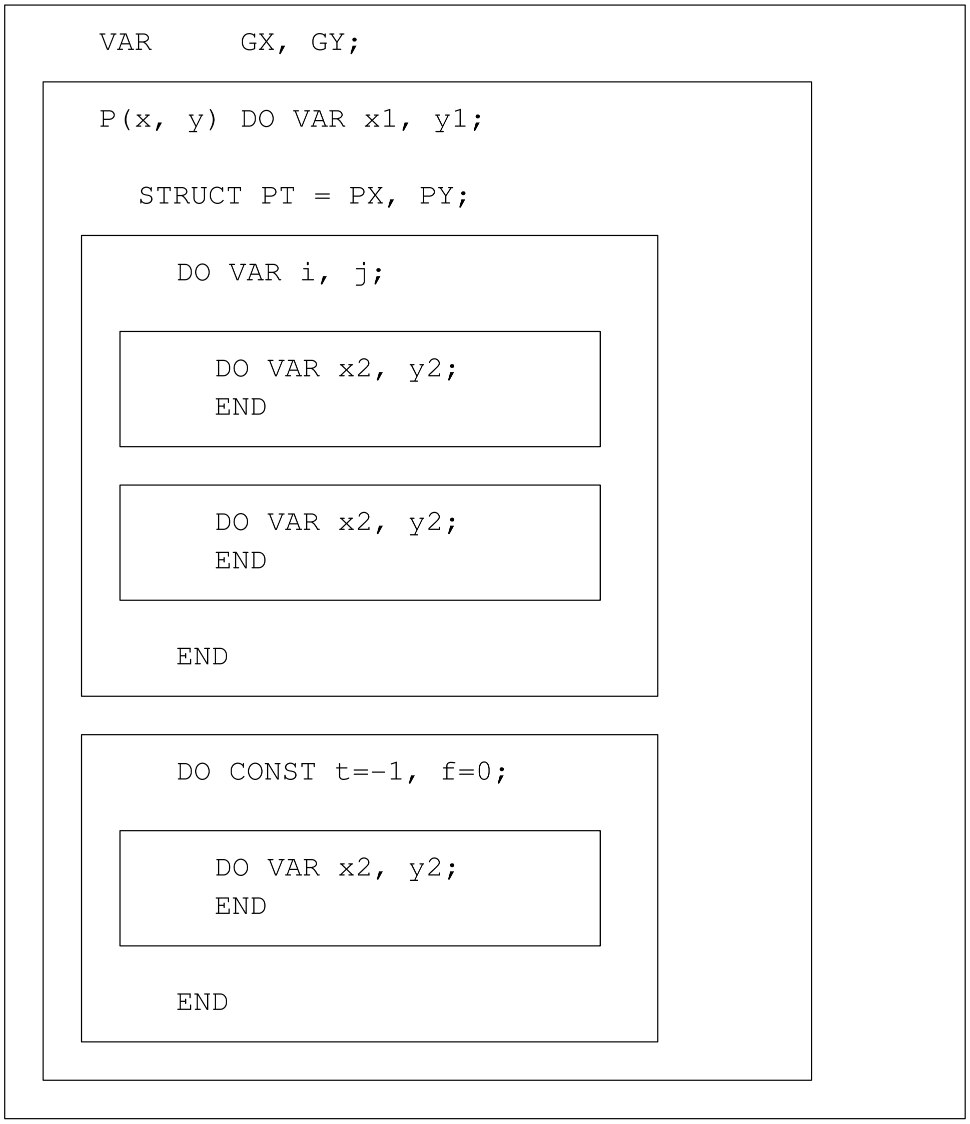
As shown in this example, symbol names may be reused in procedure definitions (formal argument names) as well as in subsequent compound statements. Since the variables *i* and *j* will be destroyed at the end of the compound statement forming the body of *F*, they can be reused in *G*. The same is valid for the argument names *x* and *y*.

Figure 2 shows some local and global symbols and their scopes.

Like all other symbols, the global variables *GX* and *GY* are valid from the point of their declaration, but unlike locally declared names, they remain existant until the end of the program. Their scope is the entire program (beginning at their declaration). The scopes of all symbols in the example are illustrated using boxes. Especially notice that the names *x2* and *y2* which are used in three different scopes denote three different variables. A value stored in *x2* within the first scope, for example, **cannot** be retrieved in the second or the third scope from the symbol *x2*, because the two names reference *different locations*. The variable which is created at the beginning of the first scope containing *x2* is deleted at the end of this scope and the value stored in that variable is lost. Assignments to local variables remain valid only inside the innermost boxes around the respective declarations.

1.1.10 Procedures

Each procedure may be considered a small program on its own. It communicates with other procedures using *parameters* and *return values* or through global variables. Each procedure has access to all global data objects which have been declared *before* it. A typical T3X program is a set of procedures which exchange information through arguments and global storage. Generally, it is considered 'good style' in procedural languages to keep procedures self-contained and use global storage as little as possible, but when a data object has to be referenced by a big number of different procedures, the use of top-level definitions is very common.



```
VAR     GX, GY;

P(x, y) DO VAR x1, y1;

  STRUCT PT = PX, PY;

    DO VAR i, j;

      DO VAR x2, y2;
      END

      DO VAR x2, y2;
      END

    END

    DO CONST t=-1, f=0;

      DO VAR x2, y2;
      END

    END
```

Figure 2: Nested Scopes

The definition of a procedure has only one single form in T3X, while there are different ways to *declare* procedures, depending on their types its locations (more types of declarations will be declared in the section covering the T3X object model). Since there is no support for nested routines, all procedure declarations must occur at the top level – the space *between* the other procedures.

The only form of the definition is

```
P(a1, ..., aN) statement
```

where *P* is the name of the procedure, *a1...aN* are the names of its formal arguments, and *statement* is the body of the procedure – the part which describes its *meaning* and which will be executed when the procedure is called.

The procedure name may be any valid symbol and it is declared in a global context. Therefore, procedure names may never be reused. (One advantage of T3X's strict scoping rules is that procedures cannot get shadowed.) The arguments *a1...aN* are local to the procedure. Their names will cease to exist after the statement forming the body has been accepted. Hence, they may be reused after the procedure definition, but not inside of it. The parentheses around the argument list must always be specified, even if the list itself is empty:

```
Q() statement
```

The number of arguments specified in the definition of a procedure determines the *type* of the procedure. The type of a procedure is notated as a single number which represents the routine's *arity* – the number of its arguments. In T3X, the argument counts of all procedure calls will be checked. The compiler will not allow calls with a wrong number of parameters. This has to be done because of T3X's calling conventions: Parameters are passed in *reverse* order and therefore, each procedure relies on a correct number of arguments. Other languages like **C**, for example, use a totally different approach in which it is easy to compensate for missing or superflous procedure parameters. The only real advantage of this approach, however, is the possibility to define variadic procedures (procedures with a variable number of arguments). This is also possible in T3X, but using a different mechanism. (Variadic procedures will be explained later in this chapter.)

When a procedure is called, it may receive data through its arguments. This works in the following way. Given a procedure

```
P(x, a, b, c) RETURN a*x*x + b*x + c;
```

and a procedure call

```
y := P(n, i, j, k);
```

the caller places the values of the *actual* arguments *n*, *i*, *j*, and *k* in a temporary storage location (usually on the runtime stack), saves the address of the following operation (the assignment in this case) and then transfers control to the procedure *P*. In *P*, the *formal* arguments *x*, *a*, *b*, and *c* reference storage locations which exactly match the temporary locations of the values passed to the routine, so that

`x=n, a=i, b=j,` and `c=k`

apply.

The procedure *P* computes

$ax^2 + bx + c$

and returns its value to the caller. Each procedure returns automatically, if its body has been processed completely or if an explicit `RETURN` statement is executed. In the above example, both happens at the same time. It is not unusual to specify a `RETURN` statement at the end of a procedure, since only `RETURN` may pass an explicit value back to the caller. Procedures which do not return through `RETURN` have an default return value of zero. In the example, however, the value of *P* is explicitly specified. After passing control back to the caller, the assignment takes place, and the result of the procedure call is stored in *y*. *Between* the procedure return and the assignment, the temporary storage where the actual arguments were held is released again.

Unlike most other languages, T3X defines a strict order of evaluation for procedure arguments: argument lists are evaluated from the left to the right. Given a (fictious) procedure *P*, which prints its argument) and an (also fictious) four-argument procedure Q, the following statement is guaranteed to print `ABCD`:

```
Q( P("A"), P("B"), P("C"), P("D") );
```

The most frequently used form of the procedure has a body consisting of a compound statement (see example 10).

```
fib(n) DO
    VAR f, i, j, k;

    f := 1;
    j := 1;
    FOR (i=1, n) DO
        k := f;
        f := j;
        j := j+k;
    END
    RETURN f;
END
```

Example 10: An Iterative Fibonacci Function

Notice that the variables declared at the beginning of the procedure

```
VAR f, i, j, k;
```

belong to the compound statement rather than to the procedure. Like in conditional statements and loops, the statement block is used to extend the scope of the procedure: not only a single statement, but a group of statements forms the body of the routine.

Since the values of actual arguments are copied to a local storage location beforwe they are passed to the procedure called, the values of the actual arguments may never get changed inside of the called procedure.

```
p(x) x := 7;

q() DO VAR x;
    x := 5;
    p(x);
    p(x);
END
```

Example 11: Modifying an argument

In example 11, the assignment *x:=7* in *P* does not affect the variable *x* inside of *Q*. Even though the *x* of *P* shares the same address with the actual argument in *p(x)*, the value transported to *P* is a *copy* of the *x* of *Q*. Therefore the assignment in *P* changes the value of the (invisible) local copy. If this was not the case, how could the following procedure call work?

```
p(5+7)
```

Since the expression *5+7* has no address (is not a valid lvalue), no argument address could be associated with its value, if no local copy of this value was created.

Recursive Procedures

It is perfectly safe for a procedure to call itself. Since the declaration of a procedure takes place while parsing its *head* (consisting of its name and its argument list), the declaration is already valid when the compiler processes the body. Therefore, the procedure may *recurse* into itself:

```
fac(n) RETURN n=0-> 1: fac(n-1)*n;
```

This small example computes *n*!. For the trivial case *n=0*, it simply returns 1. To compute *n!* for *n>0*, it first computes *(n–1)!* and then multiplies it with *n*. To compute the factorial of *n–1*, it applies itself to *n–1*. Since the argument of the recursive call is decremented by one at each level of recursion, it will finally reach 0 and the procedure will start returning.

```
MODULE fac(t3x, string);

OBJECT  t[t3x], str[string];

fac(n) DO VAR f, b::100;
    t.write(T3X.SYSOUT,
        str.format(b, packed"n = %d\n", [(n)]),
        str.length(b));
    f := n = 0 -> 1 : fac(n-1) * n;
    t.write(T3X.SYSOUT,
        str.format(b, packed"fac(%d) = %d\n", [(n), (f)]),
        str.length(b));
    RETURN f;
END

DO fac(5); END
```

Example 12: Visible Recursion

Recursion is safe in T3X, because local variables (which include formal arguments) are created freshly each time the according declaration is passed. Therefore, the symbol *n* in the above example denotes different variables at each level of recursion. To see how recursion works, the modified factorial function in listing 12 is recommended. It is not necessary to understand this example in detail. It is sufficient to know that it prints the value of *n* before re-entering *FAC* and the values of *n* and *f* after returning from the recursive call. Try to predict what this program will print before running it.

Mutually Recursive Procedures

```
A() DO !...
        B();
END

B() DO !...
        A();
END
```

Example 13: Mutual Recursion

Recursive procedures which depend on each other are called 'mutually recursive'. Such a configuration introduces the following problem: Given the procedures *A* and *B* displayed in example 13 which depend on each other, it does not matter which one is declared first – one will always be inaccessible inside of the other. In the above configuration, *B* is undefined in *A* because it will be declared after *A*. When swapping the definitions, *A* would become undefined in *B*.

Some languages solve this problem using *simultaneous definitions* or by by declaring undefined procedures implicitly. T3X, uses a more explicit scheme: each procedure may be *declared* before its definition. A declaration makes a symbol known to the compiler, but does not associate any meaning with the declared symbol. In the case of procedures, the meaning may be 'subsequently delivered' in a later definition. To declare a procedure, the `DECL` statement is used:

```
DECL name(type);
```

Analogous to `VAR` statements, any number of comma-separated declarations may be included in a single `DECL` statement. *Name* is the name of the procedure to declare and *type* is a constant expression specifying the number of formal arguments of that procedure. This value is required to type check forward calls to the procedure. The number of formal arguments in a subsequent definition must exactly match the type specified in the declaration. Otherwise, a redefinition error will be signalled.

`DECL` reserves the given names for later procedure definitions. Therefore, each of these names may be reused *only* in *one* subsequent procedure definition. Declaring a procedure without defining it later is an error, since this may leave forward references to the declared procedure unresolved.

To correct the above program containing the mutually recursive procedures *A* and *B*, the declaration

```
DECL B(0);
```

has to be inserted *before* the definition of *A*. Like procedure definitions, `DECL` statements are allowed only at the top level, but not in local scopes.

Variadic Procedures

All procedures have fixed numbers of arguments in T3X. It is possible, however, to pass a variable number of arguments to a procedure using a vector. The example program 14 computes the average of the values stored in the vector *v*.

```
average(n, v) DO VAR i, x;
    x := 0;
    FOR (i=0, n) x := x+v[i];
    RETURN x/n;
END
```

Example 14: A Simple Variadic Procedure: average()

Since vectors are first-class data objects in T3X, it is possible to inline vectors in procedure calls, thereby forming an elegant way of passing vectors with variable sizes to a procedure:

```
average(5, [ 2, 3, 5, 7, 11 ]);
average(3, [ 123, 456, 789 ]);
```

Using dynamic tables, variables can be passed to variadic procedures, too. Given a T3X implementation of a subset of of the variadic **C** library function **printf()** and the `fib` procedure which has been defined earlier in this chapter, it would be possible to implement the following statement which prints the line

```
fib(n) = m
```

for each $i \in \{1, 2, \ldots, 10\}$ and $m = fib(n)$:

```
FOR (i=1, 11)
    printf("fib(%d) = %d\n", [ (i), (fib(i)) ]);
```

printf() replaces each **%d** with the readable representation of the value of one of the arguments following the string. Each time, a **%d** is processed, the procedure advances to the next argument. The **C** version uses a variable number of arguments while the T3X version uses a vector to transport the arguments.

BTW: **printf()** uses the number of **%d**'s to determine the number of arguments passed to it. The programmer is responsible for supplying a sufficient number of them.

1.1.11 The Main Program

Each program has an initial entry point where the execution begins at run time. In T3X, the entry point is a compound statement at the top level which does not belong to any procedure definition. This compound statement is mandatory and it always must be the last definition in the entire program. Consequently, the minimum valid T3X program is

```
DO END
```

The main procedure is an ordinary compound statement and may declare its own local symbols. Since it has no name, it cannot recurse, however. Also, `RETURN` may not be used inside of it, because there is no procedure to return to. When the flow of control reaches the end of the main program, the program terminates automatically. In this case, a zero return code is delivered to the calling process (if supported by the host operating system).

1.2 The T3X Object Model

Like many popular object oriented languages, T3X is a *hybrid language*. A hybrid language is a language incorporating (at least) two different paradigms. T3X uses the object oriented approach at a rather abstract level and the procedural approach at the lower levels. For example numbers are no objects in T3X and adding numbers is not done by sending messages. In a purely object oriented language, the term

```
5 + 7
```

would be interpreted as *send the message '+' with the argument '7' to the object '5'*. In a procedural language, however, adding numbers is done by combining the factors '5' and '7' using the '+' operator. Interpreting numbers as operators and expressions as messages makes no sense in a procedural language, since numbers and operators are not implemented this way. There are many hybrid languages employing both the procedural and object oriented approach. For example, C++, Java, and ObjectPascal fit in this category. A well-known member of the family of purely object oriented languages is Smalltalk.

1.2.1 Object Oriented Programming

What is the *object oriented programming* (OOP) paradigm anyway? The term 'object oriented' (OO) has become a little fuzzy, since it has been excessively misused in marketing campaigns and blurry language definitions.

An object oriented language encapsulates code and data definitions in templates called *classes*. A class contains data definitions, procedures, and *public procedures* (aka *methods*). Each class can be *instantiated* by declaring an *object* (aka *instance*) of that class. Each object contains the data objects defined inside of its class. (The term *object* is used to relate to instances of classes in this book. Variables and vectors are referred to as *data objects*. Each object is a data object, but not all data objcts are objects.) Only procedures defined in its class may access the data contained in an object. No procedure which is not contained the the class of an object may ever access data of the object. This principle is called *encapsulation*. This is a fundamental property of OO languages.

Each class may have multiple instances (objects). In this case, each object of the class has its own private data area. This is why classes may be reused. Manipulating one object has no effect on other objects of the same class.

Methods of classes are invoked by sending *messages* to objects of that class. A message is similar to a procedure call. It may transport arguments 'into' the object and the object may return a value to the caller. Since the method is part of the class of the object, it is allowed to manipulate data inside of the object. This way, methods provide a clean and abstract interface to the data of the object. The data structure itself is hidden from the user and may change without changing the interface.

Other languages define additional concepts like *inheritance*, *protected* and *public variables*, *virtual methods*, *friend relationships*, *class variables*, etc, but most of these concepts are semantically hard to handle and weaken the object oriented principles. All concepts that are actually necessary to define an object oriented model are

- Classes
- Objects
- Messages

1.2.2 Classes

The general forms of the class declaration are as follows:

```
CLASS classname() declarations END
CLASS classname(required) declarations END
```

Between the class header (consisting of the keyword `CLASS`, the name of the class, and the *depdendency list* in parentheses) and the keyword `END` which delimits the class, there may be any number of declarations. These types of declarations are allowed in class contexts:

- Variables
- Constants
- Structures
- *Public constants*
- *Public structures*
- Procedures
- *Public procedures* (*methods*)
- Forward declarations
- Objects

Nested classes are not defined in the T3X object model.

The general forms of the class declaration are as follows:

```
CLASS classname() declarations END
CLASS classname(required) declarations END
```

Between the class header (consisting of the keyword `CLASS`, the name of the class, and the *depdendency list* in parentheses) and the keyword `END` which delimits the class, there may be any number of declarations. These types of declarations are allowed in class contexts:

- Variables
- Constants

• Structures

• *Public constants*

• *Public structures*

• Procedures

• *Public procedures* (*methods*)

• Forward declarations

• Objects

```
CLASS a()
 VAR  flag;

 flip() flag := \flag;
END ! the scope of class A ends here.

CLASS b()
 VAR flag;

 flop() flag := \flag;
END
```

Example 15: Class-level Scopes

All declarations between `CLASS` and `END` are local to the class. Therefore, classes add an additional level of scoping between the global level and the procedural level. All data objects and procedures declared inside of a class are only visible inside of that class. The names of entities declared at class level may be reused outside of the scope of the class. Hence different classes may define data objects, procedures and even methods with equal names. Example 15 illustrates this principle. This example defines two classes *A* and *B* each defining a variable named *flag*. At the end of the scope of *A*, all declarations of *A* become invisibe (encapsulated) and so the name *flag* may be reused in *B*. Since the procedure *flip* is contained in the same scope as *flag*, it may access the *flag* of *A*. In the same way, the procedure *flop* may access the *flag* of *B*. The both variables named *flag* are different entities, though, since they are contained in different classes.

Like structures, classes are merely templates for data objects. The describe the layout of a data structure plus a set of method which may be used to manipulate the structure. The size of a class is computed in the same way as the size of a structure. It is equal to the sum of the sizes of all class members. In expressions, the name of a class is a constant evaluating to the size of the class. Classes without any instance variables have a size of one machine word.

The only way of changing the state of a class from the outside is to send it a message. T3X supports a simplified form of the methods called *class constants*. Class constants may be though of as 'lightweight' methods returning a constant value. Thet allow to export values and structures without having to send a full message. OO systems which do not allow to change the state of an object without sending a message are said to employ *strict encapsulation*. Strict encapsulation in T3X is illustrated in figure 3.

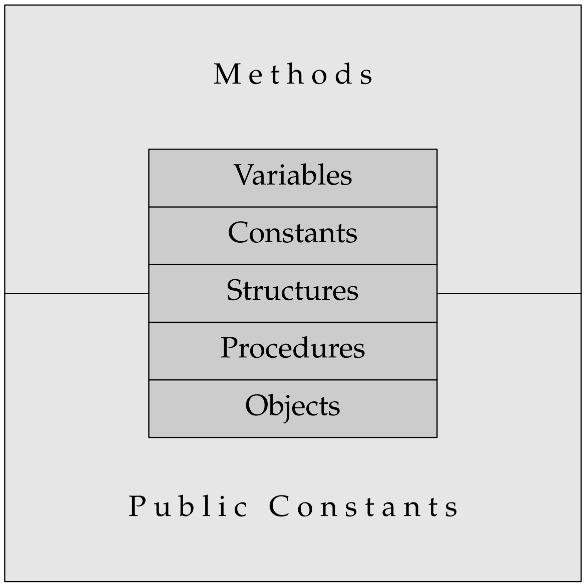


Figure 3: Strict Encapsulation

1.2.3 Objects

1.2.4 Instances

Objects are used to instantiate classes. An `OBJECT` statement is to a `CLASS` definition what a `VAR` statement is to a `STRUCT` definition. While the class defines the layout of an object, the `OBJECT` statement actually creates an object in memory. The general form of the object definitition is as follows:

```
OBJECT an_object[a_class], another_object[another_class];
```

Any number of objects may be defined in a single `OBJECT` statement. Each class may be instantiated any number of times and different classes may be instantiated in the same statememt. The name of the object to define is specified before the square brackets and the class of the object inside of the brackets:

```
OBJECT str[string];
```

creates an object of the class *string* and names it *str*.

An object may be a factor in an expression. It evaluates to the address of its first member. The notations

```
objectname
```

and

```
@objectname
```

are equivalent.

When creating multiple instances of a class, only the data defintions of the class are instantiated. The methods of a class belong to the class rather than the object. They are created when a class is declared.

The only way to alter the state of an object is to send it a message. Therefore, the state of each object is – ideally – completely independant from the states of other objects, even if they belong to the same class. In a hybrid language like T3X, however, the procedures of a class may change data objects defined in the global scope. Changing a global object from within an object changes the state of *all* other objects of the same class (and maybe even others). Therefore, this technique is deprecated. Of course, there are situations where an object *has to* change the global state, for example when performing input/output operations. Classes defining such objects are said to have *side effects*.

1.2.5 Class Dependencies

In order to create an instance of a class *A* inside of a class *B*, the class *B* must *require A*. In this case, *B* is also said to *depend on A*. The most simple scenario contains two classes which are defined inside of the same file:

```
CLASS a()
 ! definitions of A
END

CLASS b(a)
 OBJECT xa[a];
END
```

Example 16: Dependent Classes

Since *B* instantiates *A*, it required *A*. A class is *required* by including its name in the dependency list of the class header of the dependent class. Requiring a class has two effects:

• it embeds information about the required class in the name space of the dependent class

• it allows procedures of the dependent class to send messages to instances of the required class

Modules

Things get a little more complex, if the dependent class and the required class are located in different files. Since class names are contained in the global scope, they are lost as soon as the compiler has finished the translation of the file they are contained in. Hence the required class would be unknown when the compiler translates the file containing the depedent class. To allow classes to be located in different files, an additional level *above* the global scope is added. It is called the *public scope* and it persists even when the compiler finishes.

To add a class to the public scope, two steps are necessary. First, the file must get a *module header* which names the file. A `MODULE` statement has the following general form:

```
MODULE module_name (required, also_required);
```

The module name specified in the module header *must* be the same as the actual name of the file containing the module (but not including the *.t* suffix). If, for example, the class *A* is located in a file named *tools.t*, the module header would look like this:

```
MODULE tools();
```

The parentheses after the module name serve the same purpose as in class definitions: they delimit a dependency list. It may be ignored for now. The module header allows the compiler to locate the definition of a class, even if multiple classes are contained in a single file. (If files were named after classes, only a single class could be included per file.)

The second step required to export a class to the public level is to prefix its class header with the keyword `PUBLIC`:

```
PUBLIC CLASS a()
```

Example 17 shows the contents of two files *file_a* (containing the required class *A*) and *file_b* (containing the dependent class *B*). When compiling first *file_a*, *A* is exported to the public level. When compiling *file_b*, *A* is imported from the public level when it is required by *B*.

```
MODULE file_a();                MODULE file_b();
PUBLIC CLASS a()                CLASS b(a)
 ! definitions                   OBJECT xa[a];
END                             END
```

Example 17: External References

Using Classes in Procedural Programs

Another purpose of the `MODULE` statement is to provide an interface between the procedural and the object oriented parts of T3X. If only classes could require classes, it would be impossible to instantiate a class inside of a T3X program, since the main program is procedural. To instantiate a class in a procedural program, it is added to the dependency list of the module wishing to instantiate the class. It does not matter whether the required class is a public class contained in a different module or a ’private’ class contained in the same module. In the latter case, however, the `MODULE` statement must *follow* the class definition. Example 18 show a procedural program sending a message to a class.

```
CLASS A()
 PUBLIC m() DO
        t.write(T3X.SYSOUT, "A: received m.\n", 15);
 END
END

MODULE main(A);

DO OBJECT xa[A];
        xa.m();
END
```

Example 18: The Procedure/Object Interface

1.2.6 Methods and Messages

Messages are used to alter the state of an object from the outside. Basically, procedural programs are sets of procedures calling each other. An object oriented program is a set of objects sending messages to each other. Sending a message to an object activates a public procedure defined in the class of the object. A method definition looks like a procedure definition with the keyword `PUBLIC` attached:

```
PUBLIC procname(arguments) statement
```

Method definitions are only valid in class-level scopes.

A message is sent to an object using the syntax

```
objectname.methodname(arguments)
```

Messages may be factors in expressions or standalone statements. When used as statements, they must be terminated with a semicolon:

```
objectname.methodname(arguments);
```

The arguments of a method are passed in the same way as the arguments of a procedure and like a procedure, a method returns a value. The difference between an 'ordinary' procedure and a method is that a method changes the *instance context* upon entry. The instance context is the set of data objects defined in a class. It is comparable to local contexts of procedures: when a procedure is entered, it created a new local scope and when it leaves, it restores the caller's context. Unlike a local scope, the instance context is *persistent*, though. Therefore, method do not create new instance context, but just activate an existing context. The caller's context is saved upon entry (usually on the runtime stack) and restored when the method returns. Since instance contexts are persistent, the changes performed by methods are permanent.

Each object has its own instance context, which is divided into the data objects declared in its class. Methods use the instance context to access class-level data. By shifting the instance context upon entry, each object accesses only its own private data. The instance context may be thought of as a multiplexer. The principle is illustrated in figure 4.

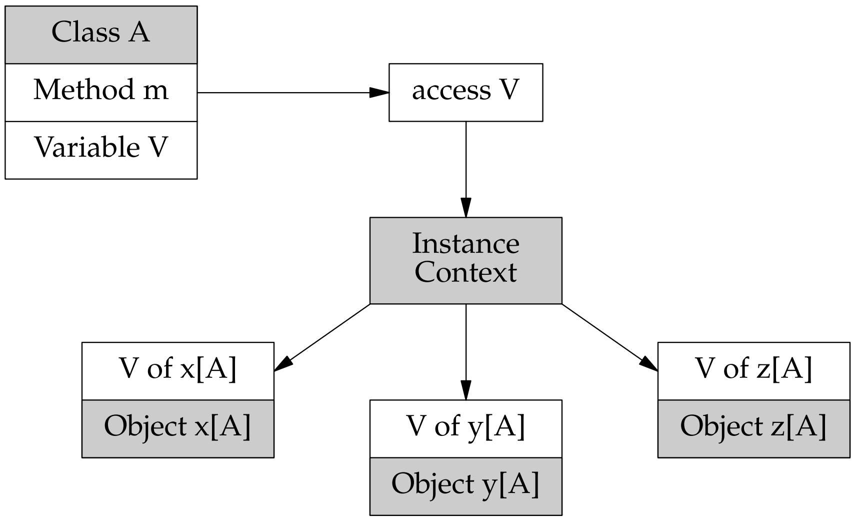


Figure 4: Multiplexed Instance Contexts

In this figure, class *A* defines a method *m* which accesses the instance variable *V* which is also declared in *A*. The instance of *V* accessed by *m* depends on the object the message is sent to. Sending *x.m()*, for example, results in accessing the *V* of *x* and sending *z.m()* results in accessing the *V* of *z*.

In T3X, the *current instance* (the currently active instance context) can be referred to using the symbol *SELF*. *SELF* is a pseudo-variable which always refers the the object owning the current instance context. Therefore, *SELF* may only be used inside of procedures local to classes. Using *SELF*, an object may send a message to itself, as shown in example 19.

```
CLASS math()
 PUBLIC prod(i, j) DO VAR p;
        p := 1;
        FOR (i=i, j+1) p := p*i;
        RETURN p;
 END

 PUBLIC fac(n) RETURN self.prod(1, n);
END
```

Example 19: Sending a Message to the own Instance

In this example, the method *fac* of the class *math* uses the method *prod* of the same class to express the factorial of *n* by sending the message *prod(n)* to itself. Of course, methods may recurse, too, since they are basically procedures. Therefore, *fac* could as well be defined this way:

```
PUBLIC fac(n) RETURN n<1-> 1: self.fac(n-1) * n;
```

Since objects are basically vectors, they may be passed to procedures (or methods) as parameters. By passing an object to a procedure, however, the object loses its type information, since the pointer to the object is stored in a typeless argument variable. To be able to send messages to such objects, the `SEND` operator is introduced. Its gerenal form is

```
SEND(variable, classname, methodname(arguments))
```

This operator sends the message *methodname(arguments)* to the object of the class *classname* pointed to by *variable*. For example, the statement

```
y := m.fac(5);
```

is equal to

```
pm := @m;
y := SEND(pm, math, fac(5));
```

Class Constants

Constants may be defined public as well:

```
PUBLIC CONST symbol = expression;
```

Such constants can be accessed from outside the class by sending a special form of a message to the *class* which defines the constant. Given the constant *MAXLEN* of the class *STRING*:

```
PUBLIC CLASS STRING()
 PUBLIC CONST MAXLEN = 32767;
END
```

the expression

```
STRING.MAXLEN
```

could be used to access the value of *MAXLEN*. So the general form of the class constant access is

```
classname.constname
```

The 'classic' way of exporting such a constant would be to define a method returning the constant:

```
PUBLIC maxlen() RETURN 32767;
```

Class constants have the advantage of saving a procedure call. Since their values ares known at compile time, they can also be used in constant expression contexts.

Structures can be exported in the same way as constants:

```
PUBLIC STRUCT structname = member1, member2;
```

Public structures are useful for objects requiring arguments in structured form or returning values in this form. For example, the *SYSTEM.STAT* function (the function *STAT* of the *SYSTEM* class) returns a structure containing information about a specific file. In addition, the *SYSTEM* class provides a public structure describing the layout of the field returned by the *STAT* function. This structure allows programs using the *SYSTEM.STAT* function to decompose the returned information.

Since class constants cannot be altered, they do not weaken the strict encapsulation principle. Public structures are an interface and an implementation at the same time. Since the same `PUBLIC STRUCT` statement is used to define the same structure internally and externally, the interface changes automatically when the implementation changes.

1.3 The Complete Scoping Model

Scoping rules basically define the contexts in which symbols are valid and under which conditions they may be redefined. In T3X, there are five different contexts:

(1) The **public context** contains all public entities (public classes and their methods and class constants) exported by a set of modules. The public context is persistent. It is not even destroyed when the compilation of a program ends. The public context may contain any number of global contexts. To purge the public context, the file containing its entries must be deleted.

(2) The **global context** covers a complete file and/or module. Its declarations are located in the space between procedures and class definitions. Each global context may contain any number of class contexts and/or procedure contexts.

(3) A **class context** contains all symbols which belong to a specific class. Class contexts are delimited by a class header (`CLASS` ...) and the keyword `END`. Each class context may contain any number of procedure contexts.

(4) A **procedure context** is equal to the argument list of a procedure plus the statement forming the body of the procedure. The argument list is a list of implicit atomic variable declarations. No other data objects may be defined in this context. A single block context (forming the body of the procedure) may be embedded in each procedure context. Procedures include methods.

(5) A **block context** begins with the keyword `DO` and ends with the keyword `END`. Each block context may contain any number of nested block contexts. Notice that this defintion is recursive, so that blocks may be nested to any level. A block context is called a *block*^{n} context, if there are n enclosing blocks.

The following rules apply:

(1) Each public class and the entities defined in its context may be redefined *once* in the module containing the original definition. The happens only, when a module containing a public class is *re-compiled*. In this case, the definition in the public context is silently updated.

(2) Each symbol used in a forward declaration (DECL statement) may be redefined by a one single matching procedure (or method) definition.

(3) Except for forward declarations and public entities, no symbol may be redeclared or shadowed ever:

- Global names may not be reused at class level, procedure level, or at block level.

• Class level names may not be reused at procedure level or at block level.

• Procedure level names may not be reused at block level.

• Block level names may not be reused in embedded blocks.

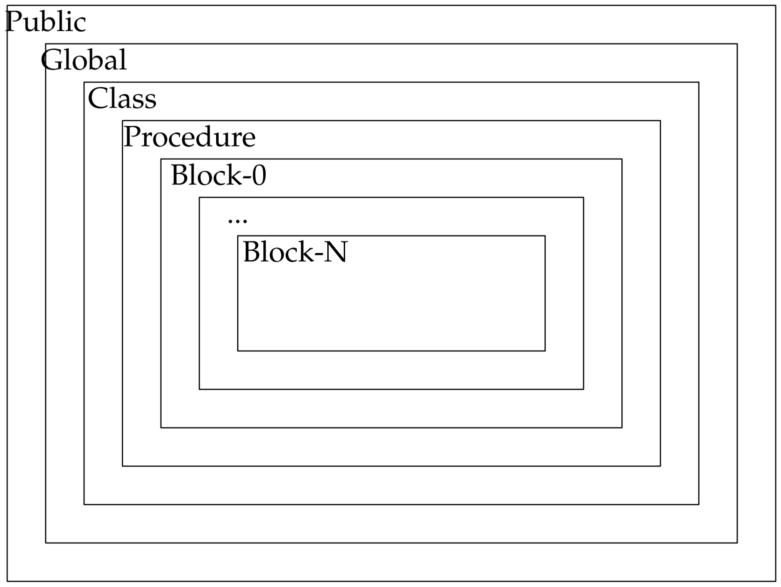


Figure 5: Levels of Scoping

Figure 5 outlines the separate levels of scoping of the T3X language.

1.3.1 Class Conflicts

At the end of a class context, all names contained in that class context *including the classname itself* will be removed from the global context. However, the class name will be memorized at a different location and may never be reused in the same program. This is a workaround for an inconsistency resulting from an interference with the module system. If the class name was not memorized, the situation outlined in example 20 could arise.

```
CLASS A() ... END

VAR A;

CLASS B(A) ...  END
```

Example 20: Bad Dependency I

In this case, class *B* would depend upon the variable *A* which would be semantically incorrect. On the other hand, if the name *A* would persist, the code in example 21 would be correct.

```
CLASS A() ...  END

CLASS B()
 OBJECT XA[A];
END
```

Example 21: Bad Dependency II

In this case, class *B* could instantiate class *A* without being dependent on it, which would also be semantically incorrect, since the module system requires that *B* be dependent on *A* in this case. Therefore, the (admittedly brute force) solution of not permitting the reuse of (deleted) class names has been chosen.

1.4 Meta Commands

A *meta command* is a statement which does not belong to the T3X language itself, but changes aspects of the language or the implementation. Each meta command begins a '#' character, followed by the command name, and some (optional) arguments. Like all other statements, meta commands are terminated using a semicolon. Meta commands may be used at the same places as statements and declararations. It is common, however, to place them at the beginning of a program. Some common meta commands are summarized below.

`#CLASSPATH "path";"`

This command is used to specify the path containing the runtime classes. The path specified takes precedence over the default locations hardwired in the compiler.

`#PACKSTRINGS;`

This command implicitly packs *all* strings contained in the following program text. When `#packstrings` is in effect, the expressions

```
"Hello, World"
```

and

```
packed "Hello, World"
```

are equal. In this case, unpacked string can no longer be expressed using string literals (tables must be used instead).

For a complete summary of all meta commands, see the appendix.

1.5 Runtime Support Classes

The T3X runtime support system consists of a core class containing routines required to interface the host operating system and a set of utility classes containing commonly used procedures for tasks like string manipulation, buffered I/O, dynamic memory management, terminal control, etc.

The runtime classes are normally required at module level. Virtually all programs depend on the *T3X* core class. It is required by including a `MODULE` statement of the form

```
MODULE module_name(t3x);
```

To access the T3X methods, the class has to be instantiated, too:

```
OBJECT t[t3x];
```

The symbol *t* is by convention used to instantiate the *T3X* class. Other runtime classes follow similar conventions.

Sime runtime classes do have global a state, others do not. Some classes do not have a state at all. Some require explicit initialization and or shutdown, others do not. These properties are outlined per class in the following subsections. Classes with global state (like the external memory interface) or no state at all (like the *string* class) are usually instantiated *once* per module. Classes with independant state (like *iostreams*) are often instantiated multiple times in the same program.

Classes with independant state are usually treated like data objects while classes which only export utility procedures may be thought of as libraries. Library classes are always instatiated once per program and do have preferred instance names.

Class	Instance	Init	Fini	State
class name	preferred instance	initialization procedure	shutdown procedure	state

Table 4: Class Property Template

Each class is described by a table at the beginning of its subsection. Table 4 shows a prototype. The **Class** entry contains the name of the class to be required in a `MODULE` statement, the **Instance** entry lists the name which is by convention used to instantiate the class (or '–' for no preferred name), **Init** and **Fini** list the initialization and shutdown procedures (or '–' for none), and **State** gives the state of the class (global, none, local), where local is the same as independent.

The type of a procedure is specified in the form

```
instance.method(arguments) ! arg1,...,argN => value
```

where each *arg* and *value* denote types of data objects. For example, the description

```
T.GETARG(n, buffer, size) ! Num,Str,Num => Num
```

denotes a routine named `GETARG` contained in the instance *t* which has three formal arguments of the types *numeric*, *string*, *numeric* and returns a *numeric* data object. Table 5 lists all types which may occur in procedure descriptions.

Identifier	Type
0	Zero (Num)
−1	Minus 1 (Num)
Bvec	Byte vector (region of bytes)
Char	ASCII character
Fdesc	File descriptor
Fndesc	Function descriptor (of shared libraries)
Num	Integer number
Shldesc	Shared library descriptor
Str	(Packed) string
Ustr	Unpacked string
Vec	Vector (region of machine words)
a | *b*	Type *a* or type *b*

Table 5: Type Identifiers used in Procedure Descriptions

All T3X runtime procedures (except for `T3X.PACK)` operate on *packed* strings. Therefore, the term *string* always denotes a *packed string* in the remainder of this section.

1.5.1 The T3X Core Class

Class	Instance	Init	Fini	State
t3x	t	–	–	none

Table 6: T3X Class Properties

The T3X core class contains methods for accessing services of the underlying operating systems as well as some machine-dependent procedures.

`T.BPW() ! => Num`

Return the number of Bytes Per Word on the host machine. When running a Tcode program, this value will always be 2, regardless of the host environment. When called by a native machine program, the procedure will return the actual machine word size of the target machine.

`T.CLOSE(fdesc) ! Fdesc => Num`

Close the file descriptor *fdesc*. To obtain a valid file descriptor, use `T.OPEN`. `T.CLOSE` returns 0 on success and a negative value in case of an error.

`T.CVALIST(n, bmap, ilist, olist) ! Num,Num,Vec,Vec => 0`

Convert a Tcode argument list into a native argument list. Since the Tcode machine is a 16-bit architecture, argument lists may need to be extended before passing them to machine code procedures in 32-bit environments.

Extending an argument list from 16 to 32 bits (or whatever is appropriate on the host system) is done by zero-extending all values in the argument vector to the size of a generic pointer on the host machine. Additionally, the offset of the Tcode machine's data area will be added to pointer type arguments. The bitmap *bmap* specifies the type of each argument.

Argument lists may not be longer than 16 elements (plus the trailing null).

N specifies the number of elements in the argument list *ilist*. If a trailing null is required, it must be counted, too. *Ilist* is a vector containing the arguments. *Bmap* is a bit field where a bit is set when the argument with the according offset is a pointer: if bit #0 is set, (*bmap* & 1), *ilist[0]* is a pointer, if bit #1 is set (*bmap* & 2), *ilist[1]* is a pointer, and so on. *Olist* will be filled with the extended argument list. It must provide up to 17 times the size of a generic pointer in bytes (which is usually equal to 17 machine words on the host system).

When a negative count is supplied, the effect of `T.CVALIST` is reversed. In this case, each member of *olist* will be copied to *ilist* and truncated to 16-bits. No pointers may be processed this way. The *Bmap* argument is ignored when converting argument lists in this direction.

`T.CVALIST` is used to prepare argument lists for passing them to dynamically loaded procedures in Tcode programs. When a T3X program is run in an environment where the size of a pointer is equal to the size of a T3X machine word, `T.CVALIST` simply copies the argument vector.

`T.SHLCALL` and `SYS.SPAWN` use `T.CVALIST` internally. Most programs will not require its use.

`T.GETARG(n, buffer, size) ! Num,Str,Num => Num`

Retrieve the *n*'th command line argument and store its first *size*−1 characters in *buffer*. If the length *K* of the the requested argument is less than *size*−1, copy only *K* characters. In either case, append a NUL character.

`T.GETARG` returns the number of characters copied. A return code of −1 indicates that a non-existing argument has been requested (*n* is too big).

`T.GETENV(name, buffer, size) ! Str,Str,Num => Num`

Retrieve the value of the environment variable *name* and store up to *size*−1 chararacters of its value in *buffer*. Append a NUL character to the text in *buffer*.

`T.GETENV` returns the number of characters copied. A return code of −1 indicates that a non-existing variable name has been specified.

`T.MEMCOMP(r1, r2, len) ! Bvec,Bvec,Num => Num`

Compare up to *len* bytes of the regions *r1* and *r2*. When a mismatch is found during the comparison, the procedure returns

```
r1::p - r2::p
```

where *p* is the position of the mismatch. When *len* bytes have been compared without encountering a mismatch, zero is returned.

`T.MEMCOPY(dest, src, len) ! Bvec,Bvec,Num => 0`

Copy *len* bytes from region *src* to region *dest*. The regions may overlap.

`T.MEMFILL(region, val, len) ! Bvec,Num,Num => 0`

Fill the first *len* bytes of *region* with the value of the least significant byte of *val*.

`T.MEMSCAN(region, val, len) ! Bvec,Num,Num => Num`

Scan the first *len* bytes of *region* for *val*. If the scanned region contains *val*, return its offset (0... *len*−1) and otherwise return −1.

`T.NEWLINE(s) ! Str => Str`

Write a system-dependent newline sequence to the string *s*. The sequence will move the cursor to the beginning of a new line when sent to terminal screens (in *cooked mode*). The sequence written to *s* will never be longer than three characters (including the terminating NUL).

`T.NEWLINE` returns a pointer to 's'.

`T.OPEN(path, mode) ! Str,Num => Fdesc`

Open the file whose path is specified in *path* in the given *mode*. The exact format of *path* depends on the operating system. The possible modes are outlined in table 7.

Mode constant	**Allow write**	**Allow read**	**If file is nonexistant**	**If already existant**	**Append to file**
`T3X.OREAD`	No	Yes	Fail	Open	No
`T3X.OWRITE`	Yes	No	Create	Overwrite	No
`T3X.ORDWR`	Yes	Yes	Fail	Open	No
`T3X.OAPPND`	Yes	No	Fail	Open	Yes

Table 7: Open Modes

When `T3X.OWRITE` is specified and a file with the given name already exists, it will be deleted first.

`T3X.OAPPND` is like `T3X.ORDWR`, but the file pointer will be positioned at the end of the file so that `T.WRITE` will append its output to the file.

`T.OPEN` returns a file descriptor for accessing *path* on success and a negative number in case of an error.

When a T3X program starts up, there already are some open file descriptors, which are by default connected to the user's terminal. These standard descriptors are summarized in table 8.

`T.PACK(vec1, vec2) ! Ustr, Str => Num`

Copy the least significant byte of each machine word of the vector *vec1* into a byte of *vec2* until a null word is found in *vec1*. This way, the unpacked string contained in *vec1* is converted into a packed string. A terminating NUL character will be placed at the end of the string in *vec2*.

Name	**Descriptor**	**Mode**
`T3X.SYSIN`	standard input	read-only
`T3X.SYSOUT`	standard output	write-only
`T3X.SYSERR`	standard error	write-only

Table 8: Standard Descriptors

Vec1 and *vec2* may be the same vector. In this case, the string will be packed *in situ* and the unpacked string will be overwritten.

`T.PACK` returns the number of machine words required to store the packed string including the terminating NUL byte.

`T.READ(fdesc, buffer, count) ! Fdesc,Vec,Num => Num`

Read up to *count* characters from the file descriptor *fdesc* into *buffer*. Return the number of characters read.

A return value less than zero indicates a severe error. A return value which is less than 'count' usually indicates that the end of the input has been reached.

When reading line oriented devices, such as terminals, a return value below *count* may indicate the end of a line. In this case, a zero value means that the input stream is exhausted.

For a summary of standard descriptors (system input and output), see `T.OPEN`.

`T.REMOVE(path) ! Str => Num`

Remove the directory entry specified in *path*. The exact format of *path* depends on the operating system.

On systems supporting multiple links (names) for a single file, this procedure will only remove the specified link. On such systems, other links to the file may still be used to access the file. Only when the last link is removed, the file will become inaccessible. On other systems, `T.REMOVE` deletes the given file immediately.

`T.REMOVE` returns zero, if the directory entry could be successfully deleted and otherwise a negative value.

`T.RENAME(old, new) ! Str,Str => Num`

Rename the directory entry whose path is specified in *old* to *new*. *Old* and *new* may describe names contained in different pathes. In this case, the directory entry will be *moved* to the directory specified in *new*. The old and the new name of the directory entry must both reside on the same physical device.

`T.RENAME` returns zero upon success and a negative value in case of an error.

`T.SEEK(fdesc, where, origin) ! Fdesc,Num,Num => Num`

Move the file pointer associated with the file descriptor *fdesc* to a new position. *Where* specifies the desired position and *origin* specifies where the motion shall start. The possible origin values are summarized in table 9.

Constant	**Origin**	**Distance**
`T3X.SEEK_SET`	Beginning of the file	+where
`T3X.SEEK_FWD`	Current position	+where
`T3X.SEEK_END`	End of the file	−where
`T3X.SEEK_BCK`	Current position	−where

Table 9: Origin Values of T.SEEK

`T3X.SEEK_SET` and `T3X.SEEK_FWD` move the file pointer forward, `T3X.SEEK_END` and `T3X.SEEK_BCK` move it backward. In either case, *where* is an unsigned value so that offsets may range from 0 to 65535 bytes.

`T.SEEK` returns zero upon success and −1 in case of an error.

The SEEK operation may be undefined on certain devices and pipes.

`T.SHLCALL(sym, args) ! Fndesc,Vec => X`

Call the shared library procedure *sym*, passing the arguments contained in *args* to it. *Sym* is a function descriptor which must be obtained using `T.SHLSYM`. The argument vector *args* contains the members outlined in table 10.

Member	**Content**
`args[0]`	the number of actual arguments
`args[1]`	the argument type bitmap
`args[2]...[N]`	the arguments to be passed to *sym*

Table 10: Argument Vector of `T.SHLCALL`

No more than 16 arguments may be passed to *sym*. The bitmap in *args[1]* works as follows: when transporting vectors to a shared library procedure, vector addresses must be converted from Tcode machine relative to absolute addresses, but numeric value must be passed unchanged. Therefore, a bit in *args[1]* has to be set for each vector argument. The first argument is associated with the least significant bit (2^0), the second argument with 2^1, and so on. For details see `T.CVALIST`.

`T.SHLCALL` returns the return value of the called procedure. When passing an invalid argument vector to it, it returns −1. There is no way to distinguish a code indicating failure of `T.SHLCALL` from a procedure return value.

`T.SHLCLOSE(shldesc) ! Shldesc => Num`

Close the shared library descriptor *shldesc*. Return zero on success and a negative value in case of an error.

`T.SHLOPEN(path) ! Str => Shldesc | -1`

Open the shared library whose location is specified in *path*. The exact format of *path* depends on the operating system. If `T.SHLOPEN` fails to open the specified shared object, it might attempt to open the same file in the directory

`$T3XDIR/lib/`*path*

where *$T3XDIR* is an environment variable containing the location of the T3X base directory (usually `/usr/local/t3x/r6`).

When the libary could be opened successfully, `T.SHLOPEN` returns a shared library descriptor. Otherwise, it returns –1.

`T.SHLSYM(shldesc, name) ! Shldesc,Str => Fndesc | -1`

Lookup the specified *name* in the shared library referenced through *shldesc*. When *name* could be resolved in *shldesc*, return a function descriptor which may be used to call the procedure *name*. If no object with the given name exists in the shared object, return –1.

Notice: only procedures should be looked up in the shared object, because calling non-procedures will lead to unpredictable results.

`T.UNPACK(vec1, vec2) ! Str, Ustr => Num`

Copy each byte of the vector *vec1* into a separate machine word of *vec2* until a null character is found in *vec1*. Each value will be zero-extended when transferred. This way, the packed string contained in *vec1* is converted into an unpacked string. A terminating NUL word will be placed at the end of the string in *vec2*.

Vec1 and *vec2* may *not* be the same vector. In this case, the string would be destroyed.

`T.UNPACK` returns the number of machine words required to store the unpacked string including the terminating NUL word.

`T.WRITE(fdesc, buffer, count) ! Fdesc,Vec,Num => Num`

Write *count* characters from *buffer* to the file descriptor *fdesc*. Return the number of characters actually written.

A return value which is less than *count* indicates a severe error (such as insufficient space left on a device).

For a summary of standard descriptors (system input and output), see `T.OPEN`.

1.5.2 The Char Class

Class	Instance	Init	Fini	State
char	chr	chr.init	–	global

Table 11: Char Class Properties

The *char* class contains methods for classifying and manipulating ASCII characters.

`CHR.INIT() ! => 0`

Initialize the character class by loading an internal pointer with the character type map.

`CHR.ALPHA(c) ! Char => Num`

Return *true*, if *c* is an alphabetic character (in the range 'a'...'z' or 'A'...'Z'). Otherwise return *false*.

`CHR.ASCII(c) ! Char => Num`

Return *true*, if *c* is a valid ASCII value (in the range 0...127). Otherwise return *false*.

`CHR.CNTRL(c) ! Char => Num`

Return *true*, if *c* is a control character (in the range 0...31 or equal to 127). Otherwise return *false*.

`CHR.DIGIT(c) ! Char => Num`

Return *true*, if *c* is a decimal digit (in the range '0'...'9'). Otherwise return *false*.

`CHR.LCASE(c) ! Char => Char`

If the character *c* is an upper case character (see `CHR.UPPER`), convert it to lower case and return it. Otherwise, return it unchanged.

`CHR.LOWER(c) ! Char => Num`

Return *true*, if *c* is a lower case letter (in the range 'a'...'z'). Otherwise return *false*.

`CHR.MAP() ! => Vec`

Return the character description map used internally. This map is a vector of 128 words containing flags for describing each ASCII character. It can be used to implement faster character checks. For example,

```
IF (CHR.LOWER(c)) ...
```

can be written as

```
chrmap := CHR.MAP();
...
IF (chrmap[c] & (CHAR.C_UPPER|CHAR.C_ALPHA) = CHAR.C_ALPHA) ...
```

which saves a procedure call each time a character is tested for being lower case.

Table 12 lists the public constants which are defined in the *char*class and can be used for testing character flags.

Flag	Property
`CHAR.C_ALPHA`	alphabetic
`CHAR.C_UPPER`	upper case
`CHAR.C_DIGIT`	decimal digit
`CHAR.C_SPACE`	white space
`CHAR.C_CNTRL`	control character

Table 12: Character Flags

`CHR.SPACE(c) ! Char => Num`

Return *true*, if *c* is a space character (HT(9), LF(10), VT(11), FF(12), CR(13)). Otherwise return *false*.

`CHR.UCASE(c) ! Char => Char`

If the character *c* is a lower case character (see `CHR.LOWER`), convert it to upper case and return it. Otherwise, return it unchanged.

`CHR.UPPER(c) ! Char => Num`

Return *true*, if *c* is a upper case letter (in the range 'A'...'Z'). Otherwise return *false*.

1.5.3 The IOStream Class

Class	**Instance**	**Init**	**Fini**	**State**
iostream	–	ios.open, ios.create	ios.close	local

Table 13: IOStream Class Properties

The *iostream* class implements fully buffered I/O streams.

I/O streams provide a character-oriented interface to the programmer while performing block-oriented I/O to the file or device associated with a stream. This way, they combine the speed of block-I/O with the flexibility of character-based I/O.

This class contains the I/O stream data structure and procedures for creating, opening, closing, reading, and writing streams.

A separate *iostream* object must be defined for each stream to be used in a program.

`ios.CLOSE() ! => Num`

Shutdown the I/O stream *ios* by first flushing its buffer and then closing the file associated with the stream. Flushing a buffer means to write pending output (if the stream has been written to) and to discard any pending input (if the stream is to be read from).

`IOS.CLOSE` returns zero, if the stream could be closed and otherwise −1. After sucessfully sending `CLOSE`, the receiving stream becomes invalid immediately and should no longer be accessed.

`ios.CREATE(fd, buffer, len, mode) ! Fdesc,Bvec,Num,Num => 0`

Initialize the iostream *ios* with the given parameters. *Fd* is an open file descriptor which will be associated with the stream. *Buffer* will be used for buffering read/write operations on the stream. *Len* specifies the size of *buffer* in characters. *Mode* controls the operations allowed on *ios*. Table 14 lists the flags which may be used to build the mode value.

Mode constant	Read OK	Write OK	LF→CRLF	CRLF→LF
`IOSTREAM.FREAD`	Yes	No	–	–
`IOSTREAM.FWRITE`	No	Yes	–	–
`IOSTREAM.FRDWR`	Yes	Yes	–	–
`IOSTREAM.FKILLCR`	–	–	–	Yes
`IOSTREAM.FADDCR`	–	–	Yes	–
`IOSTREAM.FTRANS`	–	–	Yes	Yes

Table 14: IOStream Mode Constants

CRLF→LF denotes that each CR character found in an input stream will be silently discarded. This is useful when reading DOS-style ASCII text files. LF→CRLF means that a CR character will be added before each LF in the output stream. Since `IOSTREAM.FADDCR` has no effect on input and `IOSTREAM.FKILLCR` has no effect on output, `IOSTREAM.FTRANS` may be used safely on input as well as output streams.

`IOS.CREATE` merely initializes an *iostream* object with some data. It cannot fail and therefore, it returns always 0.

When using `IOS.CREATE` to create a stream for accessing standard file descriptors (such as `T3X.SYSIN` and `T3X.SYSOUT`), these stream should never be closed. `IOS.FLUSH` may be used to synchronize them.

`ios.EOF() ! => Num`

Return a flag indicating whether input has been exhausted on the stream *ios*. When IOS.EOF returns *true*, if no more input can be read from *ios*. This is the case when the end of the associated input file has been reached or when an EOF character has been typed on a terminal.

`ios.FLUSH() ! => Num`

Flush the stream *ios* and return a value indicating whether the operation was successful. Zero means success, -1 means failure.

Flushing an output stream means to write all pending data to the associated file, flushing an input stream means to discard all pending input. The operation performed on a combined input/output stream depends on the type of the last operation performed before (reading or writing).

`ios.MOVE(offset, origin) ! Num,Num => Num`

Move the file pointer of the file descriptor associated with *ios* to a new position. The position is computed using the given *offset* and *origin*. *Offset* is the number of bytes to move and *origin* specifies where the motion shall begin. The origin values available are listed in table 15.

`IOSTREAM.SEEK_SET` and `IOSTREAM.SEEK_FWD` move the file pointer forward, `IOSTREAM.SEEK_END` and `IOSTREAM.SEEK_BCK` move it backward. In either case, *where* is an unsigned value so that offsets may range from 0 to 65535 bytes.

`IOS.MOVE` always flushes the stream buffer before changing the file pointer. It returns zero upon success and −1 in case of an error.

Constant	Origin	Distance
`IOSTREAM.SEEK_SET`	Beginning of the file	+where
`IOSTREAM.SEEK_FWD`	Current position	+where
`IOSTREAM.SEEK_END`	End of the file	−where
`IOSTREAM.SEEK_BCK`	Current position	−where

Table 15: Origin Values of IOS.SEEK

`ios.OPEN(path, buffer, len, mode) ! Str,Vec,Num,Num => Num`

Open the file specified in *path* and initialize *ios* with the resulting file descriptor and the arguments *buffer*, *len*, and *mode*. See `IOS.CREATE` for details. The exact format of *path* depends on the operating system. Table 14 (see `IOS.CREATE`) shows the existing open modes (*mode* values). When creating a file, any existing file with the same name will be deleted.

`IOS.OPEN` returns zero upon success and −1 in case of an error.

`ios.RDCH() ! => Char`

Read a single character from *ios* and return it. When the EOF condition is true on *ios*, return −1 (which cannot be a valid character).

`ios.READ(buffer, len) ! Vec,Num => Num`

Read up to *len* characters from *ios* into *buffer*. Return the number of characters actually read. A return value less than *len* may indicate the end of input or the beginning of a new line a on a terminal device. A return value of zero always indicates the EOF. A value below zero indicates a severe error.

`ios.READS(buffer, len) ! Vec,Num => Num`

Read up to *len*−1 characters from *ios* into *buffer*. Return the number of characters actually read. A return value of zero indicates that the EOF has been reached. A value below zero indicates a severe error.

Unlike `IOS.READ`, `IOS.READS` stops reading when it encounters a line separator (LF).

`ios.RESET() ! => 0`

Reset the error flag of the given iostream *ios*. Resetting the error flag is necessary to access a stream after an error has occurred (for example, after reading beyond the EOF).

`ios.WRCH(c) ! Char => Char|Num`

Write the character *c* to the stream *ios*. If the character could be written, return its ASCII code and otherwise return −1.

`ios.WRITE(buffer, len) ! Vec,Num => Num`

Write *len* characters from *buffer* to *ios*. Return the number of characters actually written. A return value less than *len* indicates a severe error (such as no space left on the target device).

`ios.WRITES(str) ! Str => Num`

Write the packed string *str* to *ios*. Return the number of characters actually written. A return value less than *len* indicates a severe error (such as no space left on the target device).

1.5.4 The Memory Class

Class	Instance	Init	Fini	State
memory	mem	mem.init	–	local

Table 16: Memory Class Properties

The *memory* class implements dynamic memory pools. Each instance of this class manages a memory area called a *pool*. Vectors can be allocated from a pool and released to it again when they are no longer required.

A first-match algorithm is used to allocate memory in a pool. The algorithm is optimized for sequential allocation. The pool is defragmented when releasing memory, but no garbage collection is performed.

Memory objects are ineffective when allocating a large number of small vectors, since the free list is kept inside of the pool.

Multiple memory pools may be defined using the *memory* class.

`mem.ALLOC(size) ! Num => Vec`

Allocate *size* bytes from the memory pool *mem* and return a pointer to the allocated vector. If the request could not be satisfied due to insufficient memory, return 0.

Up to 32765 bytes may be allocated in a single request.

`mem.FREE(vec) ! Vec => 0`

Release the memory occupied by the vector *vec* to *mem* and defragment *mem*. Thereby, the size of *vec* will be added to the amount of free memory in *mem*.

Vec must be the address of a vector which has been previously allocated in *mem*. Otherwise, the calling program may be terminated with an error message of the form

```
mem_free(): bad block
```

Accessing a free'd vector is undefined.

`mem.INIT(pool, size) ! Vec,Num => 0`

Initialize the memory pool *mem* by adding *size* bytes to its internal freelist. *Pool* must have a size of at least *size* bytes. *Size* may not be larger than 32767.

All previously allocated vectors of *mem* will be freed by `MEM.INIT`.

`MEM.INIT` always returns 0.

mem.WALK(vec, sizep, statp) ! Vec,Vec,Vec => Vec

Traverse the list of vectors in *mem*. This list contains both allocated and free vectors. Traversing the list works as follows:

When `MEM.WALK` is called for the first time, *vec* must be zero:

```
v := mem.walk(0, @p, @s);
```

This call will return a pointer to the first vector in *mem*. The returned vector can be passed to `MEM.WALK` in a subsequent call to retrieve a pointer to the next vector:

```
v := mem.walk(v, @p, @s);
```

When the *vec* argument finally points to the last vector in *mem*, `MEM.WALK` will return zero, thereby indicating the end of the list.

The *sizep* argument is a one-word vector which will be filled with the size of the returned vector. *Statp* is also a one-word vector argument which will be filled with the status of the vector (1=free, 0=allocated). If either *statp* or *sizep* is zero, it will be ignored by `MEM.WALK`.

1.5.5 The String Class

Class	Instance	Init	Fini	State
string	str	–	–	none

Table 17: String Class Properties

The *string* class contains procedures for manipulating NUL-terminated ASCII strings.

STR.COMP(a, b) ! Str,Str => Num

Compare each character in *a* with the character at the same position in *b* and return the difference

```
a::i - b::i
```

of the characters at the first mismatching position *i*. When no mismatch is encountered, the difference between the terminating NUL characters (0) is returned. Consequently, the return value of `STR.COMP` can be interpreted as follows:

>0 → 'a' is lexically greater than 'b'
<0 → 'a' is lexially less than 'b'
=0 → 'a' is equal to 'b'

STR.COPY(a, b) ! Str,Str => 0

Copy the string stored at the location *b* to the location *a*. Return zero.

`STR.FIND(a, b) ! Str,Str => Num`

Find the first occurrence of the string *b* in the string *a*. Return the offset of the string found, if any. Return −1, if *a* does not contain *b*.

`STR.FORMAT(buf, tmpl, list) ! => Str,Str,Vec => Str`

Format the arguments contained in *list* according to the template *tmpl* and store the resulting string in *buf*.

Tmpl is a string containing literal characters as well as *format definitions*. A format definition is a substring which begins with a percent sign and ends with one of the characters in `{C,D,S,X,%}`. When *tmpl* does not contain any format definitions, it will be copied to *buf* and *list* will be ignored. When format defintions exist, each definition will be used to format one element of *list*. Instead of the definition itself, the result of formatting the current member of *list* according to the definition will be inserted into *buf*.

A format definition has the following syntax:

```
%[max][:F][U][{LR}]{CDSX%}
```

([x] indicates an optional element, {xyz} indicates *one out of x,y,z*.)

• The percent sign '`%`' starts the definition.

• When a decimal number *max* is specified after '`%`', this number will be the minimum field length of the current argument. If formatting the current argument yields a result shorter than *max*, the field will be filled with blanks.

• '`:F`' denotes that the character F should be used for filling fields instead of blank characters.

• When '`U`' is specifed together with '`D`', an unsigned numeric string will be generated (a signed representation will be generated by default).

• '`L`' instructs `STR.FORMAT` to left-justify the current field, '`R`' instructs it to right-justify it. The default is to left- justify strings and to right-justify numbers.

• The last character specifies the type of the current argument in *list*.

Existing types are given in table 17 (*i* denotes the index of the current argument). The sequence '`%%`' may be used to include a literal percent sign.

Type	**Insert list[i] as**
C	character
D	decimal numeric literal
S	string
X	hexa-decimal numeric literal

Table 18: Format Definition Types

Example 21 lists some sampe format templates. `STR.FORMAT` returns the address of 'buf'.

```
Template                  Argument list   Result
"%D%% of %10:*D = %D"     [10,200,20]     10% of *******200 = 20
"'%C' = 0X%X = %D"        ['A','A','A']   'A' = 0X41 = 65
"%:-9LS%:+9RS"            ["ZZZ","YYY"]   ZZZ------++++++YYY
```

Example 22: STR.FORMAT Examples

STR.LENGTH(a) ! Str => Num

Return the number of characters contained in *a* (excluding the terminating NUL character).

STR.NUMTOSTR(buf, n, radix) ! Str,Num,Num => Str

Convert a number *n* into a string representing that number with respect to a given *radix*. The resulting numeric literal will be strored in *buf*. If both *radix* and *n* are negative, a leading minus sign will be generated. If *radix* is positive, an unsigned literal will be generated.

Buf must provide enough space to hold the resulting literal.

Valid values range from 2 (binary) to 16 (hexa-decimal) and from −2 (signed binary) to −16 (signed hexa-decimal).

STR.NUMTOSTR returns the address of the first character of the resulting literal.

STR.PARSE(source, tmpl, list) ! Str,Str,Vec => Num

Extract patterns described in *tmpl* from *source* and store the extracted objects in the members of *list*. Patterns used in *tmpl* are similar to format descriptions as used by STR.FORMAT. Characters not belonging to patterns are matched literally.

STR.PARSE compares each character contained in *tmpl* with a character in *source* (like STR.COM) unless it finds a '%'-character in *tmpl*. A percent character indicates the beginning of a pattern. Patterns match specific classes of characters. Instead of matching the pattern description, the character class described by the pattern is matched.

Some patterns store the matched substring in an element of *list* and some do not. Each pattern may consists of the following parts:

`%[len][:D]{CDSWX%}`

([x] indicates an optional element, {xyz} indicates *one out of x,y,z*.)

The special form `%[c0...cN]` may be used to match any character in the range 'c0'...'cN'.

• If a length *len* is specified, **up to** *len* characters will be matched. Typically, this option is used together with `%S`.

• ':D' instructs STR.PARSE to recognize the character in the place of D as a delimiter (default = none).

Possible pattern types are listed in table 19.

† These leading prefixes are accepted: `{+-%}`.

‡ When no length is specified, `%S` matches the entire rest of *source*. ':D' or a

Type	Store as	Matches
C	character	any single character
D	number	a signed decimal number †
S	string	a string ‡
W	-	space (any number of '\s' or '\t' characters)
X	number	a signed hexa-decimal number †
%	-	a percent sign

Table 19: Pattern Types

length may be specified to match a substring.

Numbers and characters are stored in *list[i][0]* (where *i'* is the index of the current member of *list*) and strings are copied to the location pointed to by *list[i]*.

Here is an example:

```
VAR name::50, speed, unit::10;    STR.PARSE(packed"HAL9000 @
500 MHz", packed"%:@S@ %D%W%S", [ name, @speed, unit]);
```

will store

`"HAL9000 "` in *name* `500` in *speed* `"MHz"` in *unit*

`STR.PARSE` returns the number of patterns stored.

`STR.RSCAN(s, c) ! Str,Num => Num`

Find the rightmost occurrence of the character *c* in the string *s* and return the offset (position) of the character found. If *c* is not contained in *s*, return −1.

`STR.SCAN(s, c) ! Str,Num => Num`

Find the first occurrence of the character *c* in the string *s* and return the offset (position) of the character found. If *c* is not contained in *s*, return −1.

`STR.STRTONUM(s, radix, lastp) ! Str,Num,Vec => Num`

Compute the value represented by the numeric literal stored in the string *s*. *Radix* specifies the base of the literal in *s*. It may range from 2 (binary) to 16 (hexa-decimal).

`STR.STRTONUM` performs the following steps:

• Leading '\t' and '\s' characters in *s* are skipped.

• A single plus (+) or minus (−,%) sign is recognized.

• Characters belonging to the specified literal class (based upon *radix*) are collected and converted into a numeric value.

The following characters may represent the digits from 0 to 15: `"0123456789ABCDEF"`.

When the argument *lastp* is non-zero, it will be filled with the number of characters processed. Consequently, it points to the first non-numeric character in *s* when `STR.STRTONUM` returns.

`STR.STRTONUM` returns the computed value.

No overflow checking is performed.

`STR.XLATE(s, old, new) ! Str,Num,Num => Str`

Replace each occurrence of the character *old* in the string *s* with *new*. Return the address of *s*.

1.5.6 The Tcode Class

Class	Instance	Init	Fini	State
tcode	–	–	–	none

Table 20: Tcode Class Properties

The *tcode* class contains a set of public constants describing the instruction set of the Tcode machine. There is no need to instantiate this class, since it does not contain any state or methods. To access the opcode of a specific Tcode instruction, use the class constant notation

`TCODE.I`*instruction*

For example, to load the variable *I* with the opcode of the JUMP instruction, use

```
I := tcode.IJUMP;
```

The special constant `TCODE.IENDOFSET` contains a value which is one above the highest value used to form Tcode instructions. To check whether a variable *J* contains a valid instruction, the following code may be used.

```
IF (J < 0  (J & 0x7F) > TCODE.IENDOFSET)
    ; ! Call your illegal instruction handler here
```

1.5.7 The System Class

Class	Instance	Init	Fini	State
system	sys	sys.init	sys.fini	global

Table 21: System Class Properties

The *system* class contains some procedures which form a more or less portable interface to the operating system. Most procedures have the same names and functions as Unix system calls. Some functions may be unavailable when running the virtual Tcode machine on non-Unix systems.

`SYS.INIT() ! => 0`

Initialize the operating system interface. Basically, this routine loads the shared object containing the actual interface procedures. `SYS.INIT` will fail, if the shared object could not be opened. In this case, the calling program will be halted.

`SYS.CHDIR(path) ! Str => Num`

Change the current working directory to *path*. *Path* contains the operating system dependent representation of a path.

`SYS.CHDIR` returns 0 on success and a negative value in case of an error.

`SYS.CLOSEDIR(ddesc) ! Ddesc => Num`

Close the directory descriptor *ddesc*. `SYS.CLOSEDIR` returns 0 on success and a negative value in case of an error.

`SYS.DUP(oldfd) ! Fdesc => Fdesc | -1`

Duplicate the file descriptor *oldfd* and return a new descriptor which will reference the same file. Since the old and the new descriptor reference the same file, all operations performed on one of them will also affect the other.

`SYS.DUP` returns a new descriptor on success and a negative number in case of an error.

`SYS.DUP2(oldfd, newfd) ! Fdesc,Fdesc => Num`

Duplicate the file descriptor *oldfd* and make *newfd* reference the same file. Thereafter, all operations performed on one of them will also affect the other. If *newfd* already references a valid descriptor, it will first be closed using `T.CLOSE`.

`SYS.DUP2` returns zero upon success, and a negative number in case of an error.

`SYS.FINI() ! => 0`

Shutdown the operating system interface and unload the shared object containing the interface routines. After calling `SYS.FINI`, the *system* services become unavailable immediately.

`SYS.FORK() ! => Num`

Duplicate the calling process. The new process – called the child process – will start running exactly at the point where `SYS.FORK` returns. Each process has an own data segment and an own set of file descriptors. Descriptors, which where open when `SYS.FORK` was called, will reference the same files in both processes, though.

After the successful creation of the new process, `SYS.FORK` returns 0 to the child process and the process ID of the child to the parent process.

In case of an error, it returns −1.

`SYS.GETDIR(buf, len) ! Str,Num => Num`

Store the fully qualified path name of the current working directory in *buf*. Do not store more than the first *len*−1 characters. Append a trailing NUL character. *Buf* must be at least *len* characters in length, and it may not be smaller than 65 characters.

If *len* is less than 65 or the function fails, −1 is returned. Upon success, the number of stored characters is returned.

`SYS.KILL(pid, sig) ! Num,Num => Num`

Send a signal to the process with the process ID *pid*. The constants listed in table 22 may be used in the place of *sig* to specify which signal to send to the process.

Constant	Action
`SYSTEM.SIGTEST`	Test process
`SYSTEM.SIGTERM`	Request termination
`SYSTEM.SIGKILL`	Force termination

Table 22: `SYS.KILL` Signal Values

`SYS.KILL` returns zero, if the signal could be delivered sucessfully, and a negative number in case of an error.

Delivering `SYSTEM.SIGTEST` does not have any effect. Therefore, it can be used to check whether the process with a given PID exists.

`SYSTEM.SIGTERM` may be caught by the receiving process to initiate a clean shutdown.

`SYSTEM.SIGKILL` terminates the receiving process immediately.

`SYS.MKDIR(path) ! Str => Num`

Create a directory with the name stored in *path*. *Path* is an operating system dependent path name.

`SYS.MKDIR` returns zero, if the directory could be created and otherwise a negative value.

`SYS.OPENDIR(path) ! Str => Ddesc | -1`

Open the directory specified in *path*. The exact format of *path* depends on the underlying operating system.

Upon success, `SYS.OPENDIR` returns a directory descriptor and in case of an error, it returns −1.

`SYS.PIPE(vec) ! Vec => Num`

Create a pipe (a FIFO structure) and fill the vector *vec* with two descriptors which can be used to access the pipe. Each element of *vec* will be filled with an ordinary file descriptor as returned by `T.OPEN`. Therefore, the usual I/O operations can be used to read and write a pipe.

After the successful creation of a pipe, *vec[0]* will contain the output descriptor (which is read-only) and *vec[1]* will contain the input descriptor (which is write-only).

Data written to *vec[1]* can be read from *vec[0]*. Write requests will block, if the pipe is full and read requests will block, if the pipe is empty. The size of the pipe depends on the operating system.

`SYS.PIPE` returns zero when a pipe could be created and a negative value in case of an error.

`SYS.RDCHK(fdesc) ! Fdesc => Num`

Check whether input is available from the file descriptor *fdesc* (whether a read operation on *fdesc* would NOT block).

`SYS.RDCHK` returns a non-zero value, if the operation would succeed without blocking. If the read request would block, zero is returned.

`SYS.READDIR(ddesc, buffer, lim) ! Ddesc,Str,Num => Num`

Read the next directory entry from the directory descriptor *ddesc* and fill *buffer* with the name of that entry. If the name is longer than *lim*−1 characters, truncate it to *lim*−1 characters. In any case, terminate the name with a NUL character.

`SYS.READDIR` returns the length of the name read upon success, and −1 in case of an error. Reading beyond the end of the directory will also return −1.

`SYS.RMDIR(path) ! Str => Num`

Remove the directory specified in *path*. *Path* is an operating system dependent path name.

`SYS.RMDIR` returns zero, if the directory could be removed and otherwise a negative value.

`SYS.SPAWN(prog, args, mode) ! Str,Vec,Num => Num`

Create a new process by running the program *prog* with the command line options stored in *args*. *Prog* contains the path of the executable in an operating system dependent format. *Args* is a vector of packed strings where each one contains one command line argument. The last element of the vector must be zero. *Mode* controls whether execution of the calling process will be suspended until the spawned process exits. Possible modes are outlined in table 23.

Mode name	**Meaning**
`SYSTEM.SPAWN_NOWAIT`	Execute the new process concurrently.
`SYSTEM.SPAWN_WAIT`	Suspend the caller until the new process terminates.

Table 23: `SYS.SPAWN` Modes

Notice: some operating systems may restrict the space which can be used for passing command line arguments. On non-multitasking systems, `SYSTEM.SPAWN_NOWAIT` may be unimplemented.

`SYS.SPAWN` returns the exit code of the subprocess when called with mode=`SYSTEM.SPAWN_WAIT` and zero when called with mode=`SYSTEM.SPAWN_NOWAIT`. In case of an error, it will return −1.

`SYS.STAT(path, sb) ! Str,Statbuf => Num`

Retrieve some information about the file specified in *path*. The path is in a system dependent format. The retrieved information will be stored in a *statbuf* structure which has the format outlined in table 24.

Member	**Content**
`ST_DEV`	device ID
`ST_INO`	inode number
`ST_MODE`	access bits
`ST_NLINK`	number of links
`ST_UID`	user ID of owner
`ST_GID`	group ID of owner
`ST_RDEV`	device type
`ST_SIZE`	file size in bytes
`ST_EXT64`	file size in 64K blocks
`ST_MTIME`	date of last modification (8 bytes)
	format is CYMDHMSh, see `SYS.TIME`.
`ST_MT_2`	*Buffer for ST_MTIME*
`ST_MT_3`	*Buffer for ST_MTIME*
`ST_MT_4`	*Buffer for ST_MTIME*

Table 24: `STATBUF` Structure Members

Depending on the operating system, some fields will be filled with more or less meaningful standard values. For example, systems not supporting multiple links will fill the `SYSTEM.ST_NLINK` field with 1.

The access field `SYSTEM.ST_MODE` is a bit field consisting of the flags listed in table 25.

Flag	**Description**
`SYSTEM.FM_RDOK`	file is readable
`SYSTEM.FM_WROK`	file is writeable
`SYSTEM.FM_EXOK`	file is executable †
`SYSTEM.FM_ISDIR`	file is a directory

Table 25: SYSTEM.ST_MODE Flags

† DOS-files do not have an executable flag. Therefore,

```
x[SYSTEM.ST_MODE] | SYSTEM.FM_EXOK
```

is always zero on DOS systems.

`SYS.STAT` returns zero upon success and otherwise a negative value.

`SYS.TIME(tbuf) ! Bvec => 0`

Fill the buffer *tbuf* with the current system time. *Tbuf* must provide eight bytes of space which will be filled as described in table 26.

`SYS.TIME` never fails and always returns 0. It might return an incorrect time, though, and on systems without a clock, it may fill *tbuf* with the same values each time it is called.

`SYS.WAIT() ! => Num`

Wait for a subprocess to terminate and return its exit code.

Field	Value	Range
`tbuf[0]`	year / 100	19...
`tbuf[1]`	year mod 100	0...100
`tbuf[2]`	month	1...12
`tbuf[3]`	day	1...31
`tbuf[4]`	hour	0...23
`tbuf[5]`	minute	0...59
`tbuf[6]`	second	0...59
`tbuf[7]`	second/100	0...99

Table 26: `SYS.TIME` Buffer Structure

1.5.8 The TTYCtl Class

Class	Instance	Init	Fini	State
ttycyl	tty	tty.init	tty.fini	global

Table 27: TTYCtl Class Properties

The *ttyctl* class implements a set of routines for controlling character-based video terminals and reading keyboards. Procedures contained in this class include writing to the terminal screen, cursor movement, clearing and srolling screen regions, setting display colors (where available), and decoding keyboard input.

`TTY.INIT() ! => 0`

Initialize the TTY control structures. This routine must be called before any other procedure of this class can be used. It performs the following steps (depending on the used operating system, some of these steps may be skipped):

• Open the shared library containing the low level TTY control routines.

• Determine the user's terminal type (by evaluating the *$TERM* variable).

• Check the terminal's color capability. Assume color, if the terminal name contains the substring "color".

• Check the *$ANSICOLOR* variable which may be set to YES or NO to override the color capability detection.

• Extract some required properties, control strings, and keycodes from the `/etc/termcap` database.

• Link the symbols contained in shared library.

TTY.INIT may fail for any of the following reasons:

• The shared library could not be opened.

• *$TERM* is not defined or its value is not a known TTY type.

• One out of the following termcap entries is undefined: {`co`, `li`, `ce`, `cl`, `cm`, `cs`, `sf`, `sr`, `se`, `so`}

• A required procedure could not be found in the shared library.

In any of the above cases, an appropriate message will be printed and the calling program will be terminated.

`TTY.CLEAR() ! => 0`

Clear the terminal screen using the currently selected color.

`TTY.CLREOL() ! => 0`

Clear all characters from the cursor position to the end of the current line using the currenly selected color.

`TTY.COLOR(color) ! Num => 0`

Select a new foreground and background color. `Color` is created by OR'ing together a foreground and a background color value. The following values exist (`F_` indicates 'foreground' and `B_` indicates 'background'):

`F_BLUE`, `F_GREEN`, `F_CYAN`, `F_RED`, `F_MAGENTA`, `F_YELLOW`, `F_GREY`, `B_BLACK`, `B_BLUE`, `B_GREEN`, `B_CYAN`, `B_RED`, `B_MAGENTA`, `B_YELLOW`, `B_GREY`.

The special value `F_BRIGHT` may be OR'ed in to increase the intensity of the foreground color. For example,

```
tty.color(TTYCTL.F_CYAN | TTYCTL.B_BLUE | TTYCTL.F_BRIGHT)
```

selects bright cyan color on blue background.

On monochrome terminals, only the color values `F_GREY|B_BLACK` and `F_BLACK|B_GREY` should be considered defined.

`TTY.FINI() ! => 0`

Shutdown the *ttyctl* class and unload the shared library module.

`TTY.MODE(rawflag) ! Num => 0`

Switch the terminal to *raw mode*. Some terminals (especially in the Unix world) must be in raw mode to allow to read single characters from them. In non-raw (*cooked*) mode, reading a TTY device returns only when CR (aka ENTER,NL) is pressed on the terminal's keyboard.

```
TTY.MODE(1)
```

selects raw mode and

```
TTY.MODE(0)
```

selects cooked mode.

These calls may have no effect on platforms other than Unix, but when switching a TTY driver to raw mode, it should be switched back to cooked mode before terminating the program. Otherwise, the TTY driver may be left in an undesired state and make the TTY inaccessable.

On some systems, cooked mode may not be implemented. In this case, `TTY.READC` will always return after receiving a single key.

`TTY.MOVE(x, y) ! Num,Num => 0`

Move the cursor to the specified location (column x, row y). If the specified coordinates do not exist on the used TTY, the result is undefined. Coordinates start at (0,0) in the upper/left corner.

`TTY.QUERY() ! => Num`

Check whether there are characters in the keyboard input buffer. If there are any characters, `TTY.READC` would return when called at that moment. Otherwise, it would block.

`TTY.QUERY` returns −1 if there are characters in the buffer and otherwise 0.

`TTY.READC() ! => Num`

Read a single character from the terminal's keyboard and return its keycode. For keys generating ASCII characters, the ASCII code of the key will be returned. 'Special' keys like the arrow keys, PREVIOS PAGE, NEXT PAGE, INSERT, DELETE, and the function keys return values above 255. The symbols provided to match special key codes are listed in table 28.

Keycode	Label or Keys
`K_HOME`	'Home'
`K_LEFT`	Left arrow
`K_RGHT`	Right arrow
`K_END`	'End'
`K_BKSP`	Backspace, <--, <X]
`K_DEL`	'Del', 'Delete', 'Remove'
`K_KILL`	Control + 'U', Control + Backspace
`K_INS`	'Ins', 'Insert'
`K_CR`	'CR', 'Enter', 'Return', <-'
`K_UP`	Up arrow
`K_DOWN`	Down arrow
`K_ESC`	'ESC', 'Escape'
`K_PREV`	'Prev', 'PgUp', 'PageUp'
`K_PGUP`	= K_PREV
`K_NEXT`	'Next', 'PgDn', 'PageDn'
`K_PGDN`	= K_NEXT
`K_F1`	'F1'
`K_F2`	'F2'
`K_F3`	'F3'
`K_F4`	'F4'
`K_F5`	'F5'
`K_F6`	'F6'
`K_F7`	'F7'
`K_F8`	'F8'
`K_F9`	'F9'
`K_F10`	'F10'

Table 28: Key Codes

Some systems may require to switch the TTY driver to raw mode (see `TTY.MODE`) before single characters can be received from a terminal.

`TTY.RSCROLL(top, bottom) ! Num,Num => 0`

Scroll the screen region from *top* to *bottom* down by one line. At the top of the region, a blank line will be inserted using the currently selected color. Line numbers start at 0.

`TTY.SCREENTYPE(scr) ! Vec => Num`

Determine the screen size and color capability of the connected terminal screen. The retrieved information will be stored in a SCRN structure which is defined as follows:

```
struct SCRN =
    SCRN_X,  ! Number of columns per line
    SCRN_Y,  ! Number of lines
    SCRN_C;  ! nonzero = terminal supports ANSI color
```

`TTY.SCREENTYPE` returns zero upon success and a negative value, if the screen type could not be determined.

`TTY.SCROLL(top, bottom) ! Num,Num => 0`

Scroll the screen region from *top* to *bottom* up by one line. At the bottom of the region, a blank line will be inserted using the currently selected color. Line numbers start at 0.

`TTY.WRITEC(c) ! Num => Num`

Write the character *c* to the terminal screen and return its ASCII code. The character will be output at the current cursor position. Writing a character advances the cursor. When the cursor is in the rightmost column when writing a character, the cursor position is undefined after the output operation.

`TTY.WRITES(string) ! Str => 0`

Write a string to the terminal screen as if each character of the string had been written using `TTY.WRITEC`. However, `TTY.WRITES` is usually faster than the character-oriented WRITEC method.

1.5.9 The XMem Class

Class	**Instance**	**Init**	**Fini**	**State**
xmem	xm	xm.init	xm.fini	global

Table 29: XMem Class Properties

The *xmem* class provides access to external memory blocks. An external memory (XM) block is a continous region of memory not contained in the T3X data area. XM blocks are byte addressed. Bytes in XM blocks can only be read and written using the procedures `XM.GET`, `XM.PUT`, and friends, as defined by this class.

Since each external memory block must be completely adressable using Tcode machine words, their sizes may not exceed 65536 bytes.

`XM.INIT() ! => 0`

Initialize the external memory interface. Basically, this routine loads the shared object containing the actual interface procedures. `XM.INIT` will fail, if the shared object could not be opened. In this case, the calling program will be halted.

`XM.ALLOC(length) ! => id | -1`

Allocate a block of external memory with a size of *length* bytes. Upon success, return an identifier which may be used in subsequent XM operations to access the block. In case of an error (out of memory / out of IDs), return −1.

`XM.FREE(id) ! => 0 | -1`

Release a previously allocated external memory block. *Id* is an identifier returned by `XM.ALLOC`.

`XM.GET(id, index) ! => value | -1`

Return the byte stored at address *index* of the external memory block referenced through *id*. *Index* may not exceed *X*−1 where *X* is the size of the block as specified at allocation time. `XM.GET` returns −1, if an invalid ID is passed to it.

`XM.PUT(id, index, value) ! => value | -1`

Replace the value of the byte stored at address *index* of the external memory block referenced through *id* with *value*. *Index* may not exceed *X*−1 where *X* is the size of the block as specified at allocation time. All but the least significant 8 bits of *value* will be discarded. `XM.PUT` returns −1, if an invalid ID is passed to it.

`XM.READ(id, index, buffer, length) ! => 0 | -1`

Copy *length* bytes stored at address *index* of the external memory block referenced through *id* into *buffer*. *Index* may not exceed *X*−1−*length* where *X* is the size of the block as specified at allocation time. `XM.READ` returns −1, if an invalid ID is passed to it.

`XM.WRITE(id, index, buffer, length) ! => 0 | -1`

Copy *length* bytes from *buffer* to the address *index* of the external memory block referenced through *id*. *Index* may not exceed *X*−1−*length* where *X* is the size of the block as specified at allocation time. `XM.WRITE` returns −1, if an invalid ID is passed to it.

2
How a Compiler Works

The task of translating a program written in a formal language (in this context called the *source language*) into a semantically equivalent program in a different, usually more low level language (the *target language*) may be separated into several stages, frequently called *passes* or *phases*. The front end stages of a typical compiler are illustrated in figure 6.

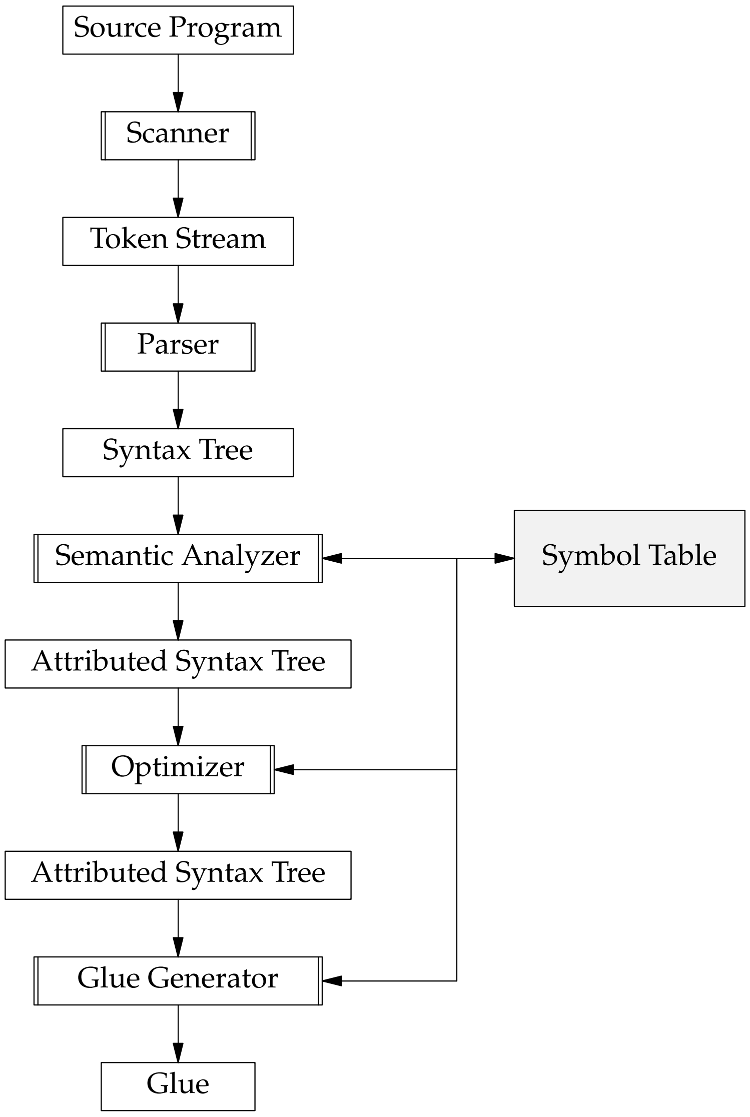


Figure 6: The Stages of Compilation: The Front End

2.1 Scanning

The *scanner* divides the input stream of the compiler (the source program) into small pieces called *tokens*. Each token represents an entity like a keyword, an operator, a punctuation sign, or a *class* of data objects like a symbol, a string, or a numeric literal. (A *class* is a set of entities which have equal properties. In this context the term *class* is not related to onject oriented programming.) The main task of the scanner is to extract tokens from the source program and to remove 'fillers' like white space and comments. In non-case sensitive languages like T3X, it also folds the case of keywords and identifiers.

Scanning is done by a finite state automaton which first skips filling characters and then determines the type of the token which is following in the input stream by examining its first character. This character is associated with a group (or a *class*) of characters that may occur only in a specific type of token. In T3X, for example, symbolic names and keywords may contain letters and underscore characters at the first position. Therefore, the *class 'symbol'* is the class containing exactly these characters (and digits at following positions). The scanner collects characters until it reads one that is *not* in the current class. It rejects this character and returns a token representing the string read.

The statement

```
WHILE (i<10) i:=i+3;
```

for example, contains the following tokens:

```
WHILE ( i < 10 ) i := i + 3 ;
```

The scanner does not require the blanks between the tokens because two neighbouring tokens always consist of characters belonging to different classes. The '<' operator that follows the symbol *i*, for example, is not in the class *'symbol'* and therefore, the scanner will recognize the end of *i* at the '<' sign.

To bring the input program into a more general form, each token is internally represented by a small numeric data object like a single machine word. Also, classes are represented by their class names rather than their values. This means that a number like 10 is not represented by 10 but by a more abstract constant like *NUMBER*. Therefore, the above statement in tokenized form could read something like

```
KW_WHILE LPAREN SYMBOL LESS NUMBER RPAREN SYMBOL
ASSIGN SYMBOL PLUS NUMBER SEMICOLON
```

where each word like *LPAREN* represents one single token of the input statement.

Of course, there is some loss of information in this representation, since the individual values of class tokens are lost (like the names of symbols, the values of numbers and character literals, and the texts of strings). Therefore, class tokens are extended with *attributes* that carry their values. There are two kinds of values:

numeric ones to hold the values of numbers and characters and textual ones that contain the names of identifiers and the texts of string literals. The attributed form of the example statement would look like this:

```
KW_WHILE LPAREN SYMBOL("i") LESS NUMBER(10)
RPAREN SYMBOL("i") ASSIGN SYMBOL("i") PLUS
NUMBER(3) SEMICOLON
```

The recognition of operators requires a different method than used in the extraction of symbols, strings, etc. Since all operators are unique objects and no classes, the scanning automaton recognizes them individually. Tokens like parentheses are absolutely unique. When a left parenthesis has been read, for example, the scanner may instantly return *LPAREN*. The backslash used for logical negation '\' on the other hand may be immediately followed by an equation sign thereby forming the negative equation operator '\='. For the backslash, there is even a third possibility: the logical OR operator '\/'.

To recognize all these operators correctly, the scanner reads the following character and compares it to each sign that may occur at the second position of the operator. If it matches one, the according token is simply returned. If it does not match any of them, the most recently read character is rejected and the token for the single-character operator is returned. The decision tree in figure 7 illustrates this process.

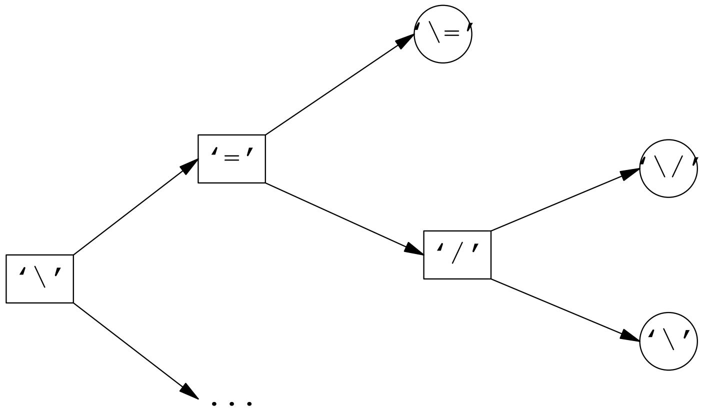


Figure 7: A Sample Operator Scanning Tree

Decision trees (or trees in gerenal) consist of *inner nodes* and *leafs*. An inner node is a node that connects multiple other nodes. A leaf is a node which has only a single connection (or none at all, if it is the only node in a tree). In this book, nodes are represented by boxes or circles and connections are represented by lines or arrows.

In scanning trees, each square box denotes a comparison with the character contained in it. An upward arrow denotes a match. In this case the current character is consumed, if the selected child is not a leaf. A downward arrow means a mismatch. Circles contain the operators whose tokens will be returned at the end of the according path. The recognition of the '\/' operator, for example, works as follows: At the leftmost node, the character '\' is matched and the upper arrow is followed. Since the selected follower is not a leaf, the current character is consumed. By consuming the backslash, '/' becomes the current character. At the next node, '/' is compared to '=' which leads to a mismatch. Following the lower arrow leads to the node containing '/' which is matched again. (The slash will not be consumed, because here the child is a leaf.) The upper arrow leads to the leaf containing '\/'. At this point, the operator '\/' has been recognized.

2.2 Parsing

Naturally, the entities delivered by the scanner may occur in any combination and in any order, but only sequences complying to certain rules form *valid* input programs. The task of the *parser* is to analyze the structure of the input program and to report violations to these rules. The set of rules a program must comply to is called the *syntax* of the source language. Therefore, parsing is also referred to as *syntax analysis*.

To some extent, parsing is much like scanning, but on a higher level, since parsing works on abstract entities – tokens – instead of characters. If the scanner detects a specific input character, it knows the *input that has to follow* in the input stream. When it reads a symbolic character, for example, more symbolic characters (or digits) *may* follow. When it reads a double quote which begins a string literal, zero or more literal characters *may* and another double quote *must* follow. The task of the parser is quite similar: When the keyword `WHILE` has been read, for example, the parser knows that a left parenthesis, an expression, a right parenthesis, and a statement must follow. It will not accept anything else following `WHILE`.

A whole lot of different syntax rules is required to describe a complete programming language, even if it is as small as T3X. Hence, a formal notation has been developed to write down syntax rules. It is called the *Backus Naur Form* or in short *BNF*. A BNF rule looks like this:

```
While_stmt:
        WHILE '(' expression ')' statement
        ;
```

BNF rules are also called *productions*. The left side of the production (on the left side of the colon) reads the production name and the right side describes what the sentence named on the left side looks like. The above production would read in words: A *While_stmt* is the token `WHILE` followed by a left parenthesis, an *expression*, a right parenthesis, and a *statement*. The trailing semicolon terminates the production – it is not part of the described syntax.

Notice that nothing is said about the syntax of expressions and statements in this example. *expression* and *statement* are just names of other productions describing the forms of these syntax objects.

By convention, tokens are written in all upper case (like `WHILE`) in syntax descriptions, while production names are in lower case (like *expression*). The right sides of productions may contain tokens as well as other production names. Productions, like subroutines in programs, are combined to form more complex (or higher level) productions. Because productions can be divided into tokens and other productions, they are called *non-terminals*. This terms is influenced by the fact that embedded productions are *inner nodes* in tree representations of higher level productions, while tokens are *leafes*. Consequently, tokens are called *terminals* in this context. A set of rules describing a complete language is called the *grammar* of that language.

A grammar for a small subset of T3X expressions can be found in example 23.

```
Sum:
        Term OptSumOps
        ;

OptSumOps:
        PLUS Sum
        | MINUS Sum
        |
        ;

Term:
        Factor OptTermOps
        ;

OptTermOps:
        MUL Term
        | DIV Term
        | MOD Term
        |
        ;

Factor:
        MINUS Factor
        | NUMBER
        | SYMBOL
        ;
```

Example 23: A Grammar for Simple Expressions

The vertical bars (|) in the grammar are used to separate right sides of productions with the same name. Its meaning is a logical OR. Generally,

```
a : A | B ;
```

also could be written as

```
a : A ;
a : B ;
```

and it means that *a* may have the form of *either A or B*.

The example grammar is in a form which is suitable for a parsing approach that is called *top-down* analysis. More technically, one would say it is an LL(1) grammar, because it can be reduced from the left to the right where the first (leftmost) token in a rule can be used to select the next production. This is achieved by beginning each production which has more than one right side with a token. Only productions without alternatives may begin with a non-terminal, since at those points, there is no decision to take.

The production *Factor* is quite straight forward. The last three rules mean that a factor may be either a *NUMBER*, a *CHARACTER*, or a *SYMBOL*. The first production

```
Factor :   MINUS Factor ;
```

is selected, if the current token is a minus sign (`MINUS`). The sign is consumed and another factor is expected. This factor may be a *NUMBER*, a *CHARACTER*, a *SYMBOL*, or even another occurance of

```
MINUS Factor
```

Therefore, any number of minus signs may precede a factor. This method can be easily implemented by using *recursion* (a factor may be a minus sign followed by a factor). Hence, this parsing method is also known as *recursive descent parsing*.

The productions *Sum* and *Term* are almost identical in their structures. In words, *Sum* is equal to *Term* followed by some optional sum operations (add, subtract) and *Term* is equal to *Factor* followed by some optional term operations (multiply, divide, modulo). Since a sum may be a single term and a term may be a single factor, the most simple form of a sum is also a single factor. When *Term* is followed by either *PLUS* or *MINUS*, however, there *must* follow another *Sum* (the according rules are in *OptSumOps*).

The last rule of *OptSumOps* is the empty production

```
OptSumOps : ;
```

This is the production which makes the trailing sums optional, since it matches any token. It will be selected only, if the current input token is neither *PLUS* nor *MINUS*.

Notice that there is an indirect recursion in *Sum* and *OptSumOps*. This way, chains of sums can be accepted. The recursion ends when a token other than *PLUS* or *MINUS* occurs in the input stream in *OptSumOps*. Such a token will be matched by the empty production which, of course, does not lead to another recursion. *Term* and *OptTermOps* work exactly in the same way.

BTW: There is a different and more powerful approach to parsing which is called *bottom up* analysis or LR(k) parsing (where *k* is the lookahead depth, like in LL(1)). Bottom-up parsing is much more flexible than top-down parsing. In fact, there exist languages that may be parsed by an LR(1) parser, but not by an LL(1) parser. In many cases, however, such languages can be processed by LL(k) parsers with *k*>1. It is often worth examining the real structure of a language, before switching to an LR(k) parser, since LR parsers are quite complex and usually require a *parser generator* to construct them. Handcoding an LL(k) parser, on the other hand, is pretty straight forward.

While the parser analyzes the tokenized input program, it builds a *tree* representing the syntactical structure of the program. Hence, such a tree is called a *synax tree*. In a syntax tree, different parts of the input program may be accessed conveniently, even if they do no occur subsequently in the input stream. Additionally, the tree structure allows manipulations at a very abstract level (statements, expressions, declarations) as well as on low levels down to single tokens. Therefore, this representation is particularly suitable for the following semantic analysis.

Syntax trees are basically made of nodes carrying tokens. Tokens which are only used as hints for the parser (like commas, semicolons, and such) have no counterparts in the tree, because they have no *meaning* execpt for the separation of syntactic objects. Abstract objects like expressions and statements are recognized by their context in syntax trees. For an illustration, see figure 8. It contains a syntax tree that represents the statement

```
WHILE (i<10) i := i+3;
```

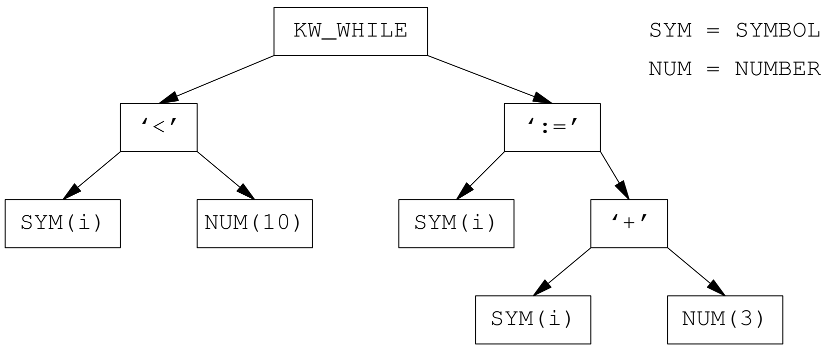


Figure 8: A Sample Syntax Tree

In the syntax tree in figure 8, the left child of `KW_WHILE` is an expression and its right child is a statement (an assignment). The left side of the assignment is an lvalue and its right side is an expression (a single ‘+’ operation in this case). The tree starting at `KW_WHILE` is itself a statement. It could be part of a larger syntax tree, like the statement part of another loop or conditional statement. *Conventions* like ‘the left child of a `KW_WHILE` node is an expression and its right child is a statement’ render ‘syntactic sugar’ like parentheses and semicolons redundant.

The extent of a syntax tree depeneds on the strategy used. It may contain a single statement, whole procedures, or the entire input program. Larger syntax trees allow more sophisticated optimizations but smaller syntax trees, naturally, save memory. On statement level, only very few optimizations can be performed, but only very little memory is required. On the global level, interprocedural optimizations become possible, but the syntax tree of the complete program must be held in core to achieve this. The procedure level is a good compromise. It is abstract enough to optimize the control flow of the program efficiently, but on the other hand it requires only one procedure to be kept in memory at the same time.

Since the parser analyzes the syntax of the input program, it also does the major part of the error reporting, because most (formal) errors made in programming are syntactical errors like missing or superflous operators and misplaced or misspelled keywords or identifiers. When the parser finds a sentence with no matching production, it prints an error message describing the error and then attempts to resynchronize itself with the input stream. When the synchronization fails, it may abort the translation process. The error message created by the parser should be self-explanatory, and it should contain the line number of the offending sentence.

2.3 Semantic Analysis

The first stages have analyzed the lexical and syntactical structure of the input program. Therefore, the program can be assumed to be formally correct at this point. Also, it is in a form which is suitable for effective processing. The next phase, called the *semantic analyzer* examins the *meaning* of the program. Naturally, only a small subset of the semantics of the source program can be covered by this analysis. Understanding the *entire* meaning of a program would allow predictions about its runtime behaviour, including the detection of logical errors and deadlocks. Such an analysis is far beyond the capabilities of a compiler (and computer science in general). The tasks actually performed by the semantic analysis phase are the following:

• The maintenance of the *symbol table*.

• Type checking.

• The detection of context errors.

The maintenance of the symbol table is one of the major tasks of this stage. The symbol table is a structure which is used to store information about symbols declared in the source program. The semantic analyzer examines declaration statements and adds the declared symbols to the table. The information contained in a symbol table entry may include information like the symbol's *name*, its *size*, *value*, *type*, and some *flags* used for symbol management.

The *symbol name* is stored to identify the symbol. When adding a symbol to the table, its name is looked up first, since all names must be unique in most languages. (Languages which do not prevent shadowing may allow equal names in different scopes.) When a name already exists in the symbol table, a redefinition error is reported and the symbol is rejected.

One exception exists in T3X: A symbol which has been *declared* but not yet *defined* may be defined later. There is only one situation where this can happen: the definition of a previously declared procedure. When a procedure is just declared (using `DECL`), a flag in the symbol table is set which indicates that the symbol may be redefined. When the procedure is defined later, the flag will be reset, thereby turning the declaration into a definition which is no longer mutable.

The *size* field may be used for bounds checking, statistics, or local storage management. It can be used to distinguish ordinal variables from vectors, too. When the described object is a procedure, the size field may be used to store the number of arguments, for example.

The *value* field holds the values of constants. Since variables have no value at compile time, this field may be used to store their addresses.

Since the typing system of T3X is pretty simple, a single field is sufficient to store all *type information*.

The *flags* may be used to store information about the initialization and use of variables, for example. The declaration flag described above also will be placed here.

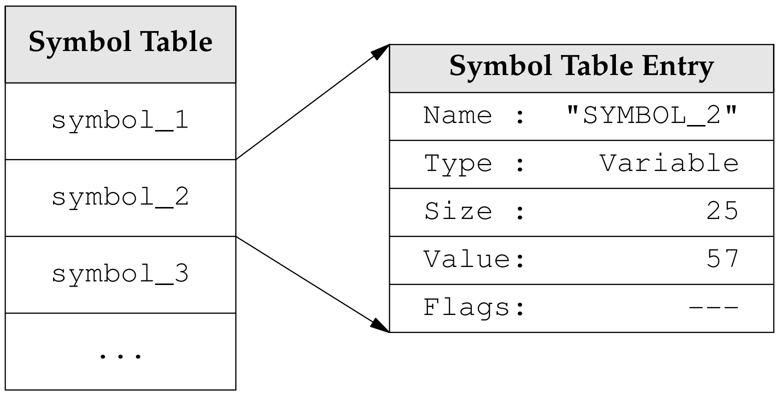


Figure 9: A Sample Symbol Table

Figure 9 shows a sample symbol table with a zoomed out entry in detail. The entry describes the symbol `symbol_2` which has the type *variable*. Its size is 25 machine words, so it is a vector. The address of the vector is a label with the identifier 57. The symbol has no flags set.

Local and global symbols are stored in the same table. When a local scope is left while processing the syntax tree, the semantic analyzer removes the symbols local to that scope from the table.

Another major task of the semantic analyzer is **type checking**. Because the typing system of T3X is quite simple, type checking is also a rather simple task in this case. A type error occurs, when an operator is applied to an operand with a type it cannot process. For example, the statement

```
C := 2307;
```

(which is syntactically correct) would be semantically wrong, if `C` would be a constant, since assignments to the type *constant* are not allowed.

To determine the type of a symbol, the semantic analyzer looks its name up in the symbol table. It may happen, of course, that the requested symbol is not contained in the symbol table, because it has been misspelled or its declaration is missing. In this case, the analyzer should report the undefined symbol. The undefined symbol may be defined thereafter to avoid subsequent error messages regarding the same symbol.

Other possible sources of type errors would be:

- A procedure name with no call operator applied to it
- The attempt to compute the address of a constant
- A subscript operator applied to a constant
- Non-constant values in compile time expressions
- A call operator ('`()`') applied to a non-procedure
- A procedure call with a wrong number of actual arguments
- An assignment to a procedure, constant, or vector
- A dependency on a non-class
- An instantiation of a non-class
- Sending a message to a non-object
- Sending an unknown message to an object

Most other procedural languages have a more strict typing system. They would not allow to build the product of different types like a character and a string, for example. The actions taken when such a type conflict arises depends on the specific language. Some languages (like **C**) have implicit type conversion rules which allow some types to be intermixed. Other languages require explicit conversion of different types, or do not allow to mix them at all.

Some **context errors** may be caught by a clean syntax description as well, but sometimes, it is more efficient to handle them at the semantic level. A context error occurs, when an otherwise correct statement appears at a place where it makes no sense (in a wrong *context*, that is). This happens, for example, when `LOOP` or `LEAVE` is found outside of a loop or `RETURN` is used in the main body of a program.

All these errors can be caught by defining ordinary statements *without* `LOOP`, `LEAVE`, and `RETURN` and allowing loop branches only in loop bodies and `RETURN` only in procedure bodies. Doing so would, of course, lead to a more accurate syntax description, but, on the other hand, it would require an additional rule for each single case. What happens, for example, if a loop is contained in a procedure body? One statement production had to be written for each case: ordinary statements in the main body, ordinary statements in a procedure body, loop statements in the main body, and loop statements in procedure bodies. Naturally, this can be done for T3X, but the number n of additional rules increases exponentially with each new class of context-dependant statements: $n = 2^{\textit{number of classes}}$. Therefore, even for languages with small sets of context-dependant classes, this method may become impractical.

Therefore, it is preferrable to handle context errors at the samantic level. When the semantic analyzer traverses the syntax tree, it remebers the beginning of each loop by defining a *loop tag* when it passes the introducing `WHILE` or `FOR` keyword. When leaving the loop context, it removes the loop tag. When a `LEAVE` or `LOOP` statement is found, the semantic analyzer checks whether there are any active loop tags. If there are none, the statement is in a bad context. Other context dependencies are checked in similar ways.

Another task of the semantic analyzer is the generation of attributes that contain additional information about symbols used in the syntax tree. For example, the 'symbol' nodes in the syntax tree are linked to their symbol table entries. This way, an *attributed syntax tree* is built. Since this tree contains information about the *meanings* of symbols, it is ready for following phases like optimizing and glue (or code) generation.

2.4 Optimizing

Optimizing is the process of transforming a syntax tree (or some other suitable representation of a program) into a semantically equivalent structure which will result in more efficient code than the original structure. Some simple – and portable – techniques to achive this would be

• Removing redundant subexpressions.

• Evaluating constant subexpressions.

• Replacing instructions with cheaper ones.

There are more sophisticated optimizations, but most of them require temporary storage allocation (like register allocation) and therefore, they cannot be performed on syntax trees. More aggressive techniques are usually placed in the back end, because they include machine-specific transformations.

Redundant subexpressions like additions and multiplications with neutral factors can be removed without altering the meaning of the program. In a syntax tree, this is done by unlinking the redundant operation and linking back the non-constant part of the operation.

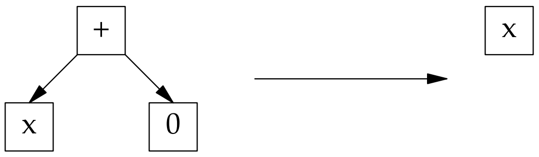


Figure 10: Redundancy Elimination

Figure 10 illustrates this process. The left child of the displayed operation is the non-constant *x* and the right follower is a constant zero. Since an addition with zero has no effect, the operation is replaced with its non-constant part.

The **Evaluation of constant subexpressions** is possible if all operands of an operation are constant. In such a case, the operation can be executed at compile time, thereby saving the time required for the computation at run time. Since the evaluation of a constant subexpressions may create new constant factors, the evaluation process is done during a *depth-first* traversal of the syntax tree: first, the subtrees are optimized and if both children could be evaluated, the node itself can be optimized, too. An expression like

```
100 + 10 * 20
```

is constant, because it contains no *variable* parts and therefore will always evaulate to the same value. Although, its syntax tree contains a non-constant factor in the sum operation. By traversing the tree in *depth-first* order, such expressions can be completely resolved at compile time, however.

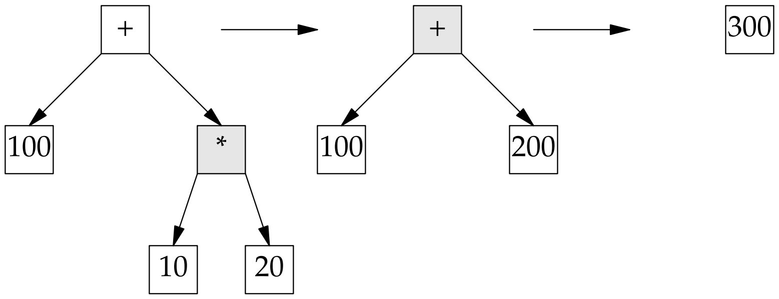


Figure 11: Constant Subexpression Evaluation

In example 11, the optimization process starts at the root node carrying the '+' operator (left figure). First, the left child is visited. Since it is a constant expression already, it is left alone. Next, the right child is processed. Its children are both numeric constants and therefore, they are left unchanged, too. Since the leafes of the '*' node are both constant, the operation can be replaced with its result (middle figure). After returning to the root node, the right subexpression has become constant, too. Hence, the '+' operation can also be evaluated, giving a simple numeric constant.

A table containing templates for possible eliminations and reductions can be found in the appendix.

Another common technique is the **replacement of complex instructions**. For example, an *unsigned multiplication* can be replaced with a *shift left* operation, if one factor is a constant multiple of 2. On many machines, the shift operation requires much less time than a multiply instruction (if there exists one at all). The same optimization can be applied to *unsigned divide operations*, if the second factor (the divisor) is a constant multiple of 2.

Note: The replacement of multiply and divide operations by shift instructions does not work for signed operations, since the sign of the variable part of the operation cannot (easily) be predicted.

Other portable optimizations include *constant condition elimination*, *jump to jump elimination*, and *loop unrolling*. Constant conditions are conditions with constant truth values. Such conditions may be created during constant expression evaluation, for example. When the truth value of an expression is known, either both, the condition and the conditional branch can be removed (if the condition is *false*), or the conditional part can be linked back into the tree without the condition, thereby making it unconditional (if the condition is *true*).

Nested conditional statements or expressions and constant condition elimination frequently lead to jump instructions whose destination is another jump instruction. This way, *chains* of jumps can be constructed. Such chains can be reduced by redirecting the first branch to the destination of the last branch of such a chain, thereby bypassing all its other jumps.

Loops with a known small number of iterations (mostly `FOR` loops) and small loop bodies may be *unrolled*. 'Unrolling' a loop means to turn the loop into a sequence by duplicating the body and removing the loop construct. For example, the statement

```
FOR (i=0, 3) v[i] := w[i];
```

may be unrolled into

```
v[0] := w[0]; v[1] := w[1]; v[2] := w[2]; i := 3;
```

While the first form is more readable and compact, the second one certainly has a better runtime performance, because no additonal instructions are to be execeuted for the loop construct itself. Loop unrolling automates this transformation. **Note**: This technique will in many cases increase the memory consumption of the target program.

2.5 Glue Generation

One might prefer to call this stage *code generation*, since this is what it does, but the major purpose of this phase is to generate an external representation of the input program that is suitable for passing it to a subsequent phase, whatever it may be. The passes which follow the glue generation are part of the compiler *back end* and therefore, the generated 'code' is actually used to *paste* the front end to the back end. Hence the name of this stage.

The glue generator produces an *intermediate language*. This language is something between the abstract, machine-independant source language and the machine-specific target language. In most cases, the glue consists of instructions for a virtual machine like a three-address machine, a register machine, or a stack machine. Therefore, the intermediate language is very low level, but still portable. It will be converted into native machine language in the back end of the compiler.

It is also possible to generate assembly language at this stage. In this case, however, the glue language will not be portable and assembly language source code is not particularly suitable for applying machine-specific optimizations. On the other hand, the code generation scheme used in this stage is pretty simple and therefore, the generated code is likely to be correct.

The most simple form of an intermediate language is code for a virtual *stack machine*. It is very easy to generate, it can be converted into native code on the fly, and it is suitable for the application of more aggressive code generation schemes, since it is pretty easy to reconstruct a tree structure from it. The generation of stack machine code will be explained by the means of some examples in the following paragraphs.

A stack machine is a machine which has a built in data stack on which all operations are performed. The stack machine used in this example has an additional external memory region for storing data objects. The stack itself will only be used to keep temporary values. T3X operators will be used to denote the operations associated with them. Addtionally the following instructions exist (S0 denotes the top of the data stack):

- **NUM #** Load numeric value '#'.
- **LDL #** Load local object from stack frame position '#'.
- **SAVL #** Store S0 in the local object at frame position '#'.
- **CLAB #** Place label '#'.
- **JUMP #** Jump to label '#'.
- **BRF #** Jump to label '#', if S0 is false.

The **BRF** instruction implicitly removes S0.

The stack machine program

```
NUM 100 NUM 10 NUM 20 * +
```

for example, means: "Load 100, 10, and 20 onto the stack. Then, remove the two top members (S0=20, S1=10), multiply them (S1*S0), and leave their product on the stack. Finally, remove S0 (200) and S1 (100), add them (S1+S0), and load their sum onto the stack." After the execution of the stack machine program, the value on the top of the stack would be:

$S_0 = 100 + 10 \cdot 20 = 300.$

The generation of this code from a syntax tree can be done by simply traversing the tree in *left-right-node* order and emitting code when leaving a node.

Figure 12 shows a syntax tree fragment representing the expression

```
100 + 10 * 20
```

To generate stack machine code from this tree, the traversal starts at the root node containing the '+' operator. First, the left branch is followed which leads to the node containing `NUM(100)`. Since this node has no children, it is left and the instruction **NUM 100** is emitted. Next, the right branch of the root node is visited.

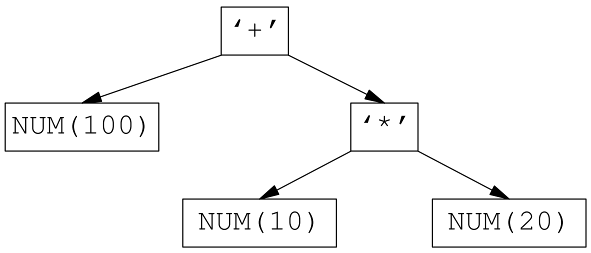


Figure 12: A Sample Expression Tree

The node '*' has two leafes. As in `NUM(100)`, the instructions **NUM 10** and **NUM 20** are generated while visiting the children of '*'. After visiting these branches first the '*' branch is left (emitting the ***** instruction) and then the root node is left (emitting the **+** instruction). Collecting the boldface words in this paragraph, will result in the sentence

```
NUM 100 NUM 10 NUM 20 * +
```

which is equal to the initially shown stack machine program.

Notice that, since the parser respects the precedence of the operators, there is no need to care about '*' having a higher precedence than '+'. The syntax tree is already built in such a way that all precedence rules will be met when stack machine code is generated using left-right-node (aka. *postfix order*) traversal. In postfix notation (which is used to program the stack machine), no precedence rules exist, since all terms are in an unambiguos form.

Generating code for special statement types like loops and conditional statements requires a slightly different technique. Keywords which introduce a specific construct will always form the root node of that statement. When visiting such a node, the postfix traversal will be suspended, a special procedure handling the construct will be invoked, and finally, the node will be left again (without traversing its children in the usual way).

The root node of the syntax tree of the example from the previous subsections

```
WHILE (i<10) i := i+3;
```

contains the token `KW_WHILE` (an illustration of the tree can be found in figure 8). When the glue generator finds the node `KW_WHILE`, it calls a handler which basically does the following:

• Generate two labels *L0* and *L1*

• Emit the label *L0*

• Traverse the condition (the left follower)

• Emit a *branch on false* to *L1*

• Traverse the loop body (the right child)

• Emit a jump to *L0*

• Emit the label *L1*

The resulting stack machine program can be found in example 24. (The example assumes that *i* is a local variable at stack frame position 1).

```
 +----> CLAB 0
 |      LDL 1          !
 |      NUM 10         !  i < 10
 |      <              !
 |      BRF 1 -----+
 |      LDL 1      |   !
 |      NUM 3      |   !  i := i+3
 |      +          |   !
 |      SAVL 1     |   !
 +----- JUMP 0     |
        CLAB 1 <---+
```

Example 24: A Sample Stack Machine Program

The program fragment

```
LDL 1 NUM 10 <
```

is the code generated for the condition *i<10* and the fragment

```
LDL 1 NUM 3 + SAVL 1
```

is the stack machine program which performs the increment of i by 3. The rest of the code in example 24 forms the semantics of the `WHILE` construct itself. The arrows show the destinations of the jump and branch instructions. The left arrow indicates the repetition of the loop, and the right one illustrates the exit branch which will be taken, if the loop condition evaluates to false.

Note: Without the repeating jump, this example program would perform the simple conditional execution of the loop body, like the T3X statement

```
IF (i<10) i := i+3;
```

Code for other flow control constructs, like `FOR`, `IE`, `LEAVE`, etc is generated by similar procedures.

Of course, there is other important information which must be delivered to the back end, as well, like global storage definitions, local storage allocation and deallocation, table and string data, etc. The used intermediate language must provide instructions for transporting all these information.

2.6 A Simplified Model

The separation of the stages described so far in this section is a little artificial. In most actual implementations, the phases would not be strictly separated but interleaved. Since the syntax of the source program controls the translation process, the parser is the controlling part of the compiler. It reads tokens through the scanner and writes the target program through the glue generator (which may contain an additional optimizing phase). Because it is the syntax which controls the compilation, one also speaks of *syntax directed translation*. This method is nowadays the most common compilation technique.

There exists a recursive descent parsing method which is called *predicative parsing*. 'Predicative' means that it is based upon predicates (or *verbs*). In a predicative parser, there is one procedure for each production (or for each class of productions) which describes the syntax of the verb (or verbs) used in that production (like '`*`', '`/`', and `MOD` in a term expression).

Imagine a predicative parser for the grammar given in example 23 parsing an expression like

```
a + -b * 10
```

Such a parser would have three procedures for the classes `Sum`, `Term`, and `Factor`. The procedure `sum()` first descends into `term()`. When `term()` returns, `sum()` checks the current token. If it is a sum operator, it consumes it and descends into `term()` again. Otherwise, it exits – thereby accepting the expression. `Term()` works in a similar way. `Factor()` only accepts *SYMBOLS*, *NUMBERS*, *CHARACTER*, and negated factors. It descends into itself after matching a minus sign to parse the operand of the unary *MINUS*.

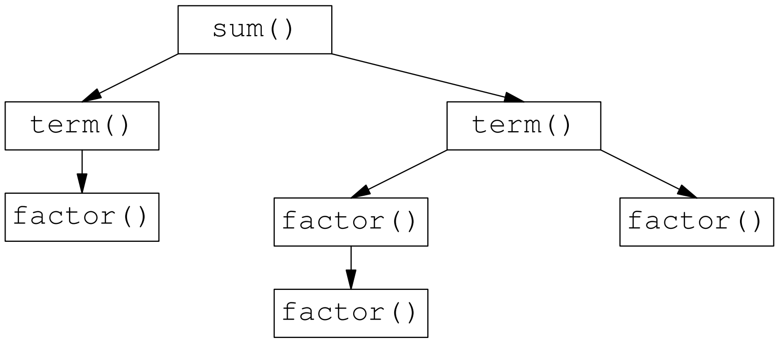


Figure 13: A Predicative Parsing Call Tree

When parsing the expression

```
a + -b * 10
```

the parser would perform the procedure calls displayed in figure 14.

When comparing this call tree to the syntax tree of the expression (which can be found in figure 14), one can observe that the structures of the trees are equal. This is *because* the translation is syntax directed. The predicative parser performs a procedure call for the creation of each branch in the syntax tree. The empty node on the left side of '+' in the syntax tree is the counterpart of the `term()` node in the call tree. Since a term may be a simple factor, `term()` has to be called to match a factor, but it does not add anything to the syntax tree in this case.

The empty node has been added just to elucidate the identity of the tree structures.

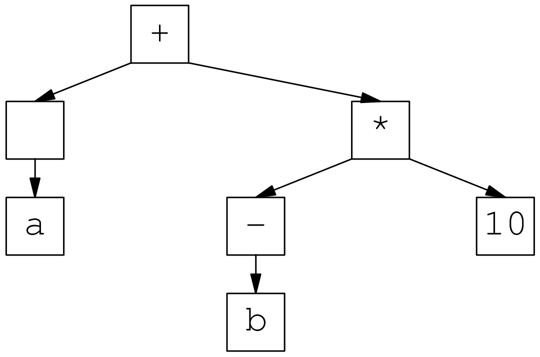


Figure 14: A Syntax Tree created by a Predicative Parser

If an optimizing phase, which would require the presence of a syntax tree as an actual data structure, is not needed, the generation of the syntax tree is not really necessary. Since the call tree of the parser has the same structure as the syntax tree, the traversal of the syntax tree as performed by the glue generator can be done *while* parsing the source program. Thereby, the parser and glue generator can be combined to form a single phase. Data is no longer transported to the glue generator via the syntax tree but through a procedure call interface (ie. by passing arguments to the generator). Naturally, this method is much simpler and more efficient than the creation of an actual data structure.

The semantic analysis takes place *between* parsing and glue generation. Therefore, if parsing and glue generation are combined to a single phase, semantic analysis has to be integrated into this phase, too: when parsing a declaration statement, the routine for adding new symbols can be called directly by the parser and when a symbol is found in an expression, the symbol can be looked up by the parser, too (by calling the appropriate semantic analysis routine). Context and type checking can also be integrated seamlessly by calling the according routines directly from the parser. The integration of the stages leads to a highly simplified model of the compiler front end, as illustrated in figure 15.

One of the major advantages of this model – besides simplicity – is the encapsulation of code generator. Because all relevant information is delivered to the generator through the procedure call interface, it needs no access to the symbol table. Therefore, only a description of the interface and the semantics of the code generating procedures is necessary to write new glue generators.

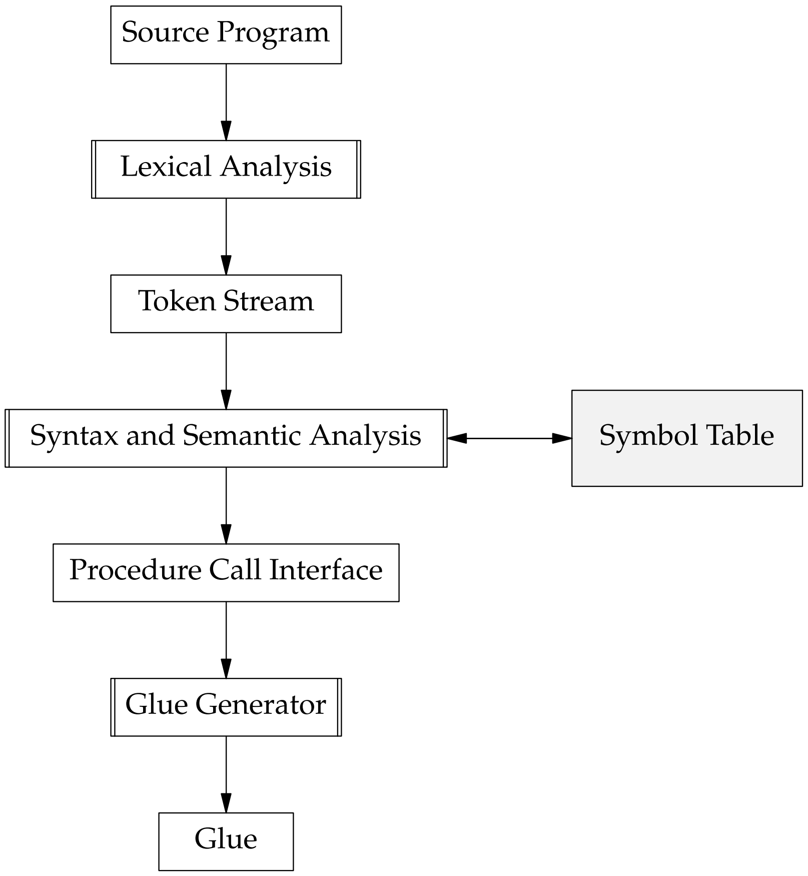


Figure 15: A Simplified Model of the Compiler Front End

The model discussed in this chapter covers all stages of a typical compiler from lexical analysis to code (glue) generation. A symbolic back end has been chosen to keep the description machine-independent. The transformation of stack machine code into code for a real CPU will be covered in a later chapter.

3
The Tcode Machine

The T3X compiler generates code for a very simple stack-based bytecode machine called the *Tcode* machine. The Tcode machine does not exist as a ‘real’ machine – there is no (hardware) processor whose native language is Tcode. Therefore, the target of the T3X compiler is called a *virtual machine* or a *bytecode* machine.

3.1 The Tcode Architecture

Like many real computers, the Tcode machine uses an array of storage cells (its *memory*) to store data and another separate array to store the program to be executed – just like a machine with separate instruction and data space. Instead of a set of registers (like most ‘real’ CPU’s), stack machines use a stack to store the operands of most operations. This approach is very simple and it is particularly easy to generate code for such architectures.

An operation on the stack machine is performed by first transferring the operands from memory to the CPU’s stack. Then, the operation itself is performed on one or more arguments on the top of the stack. Finally, the result is removed from the stack and put back into memory. The process of transferring an object onto the stack is also referred to as *pushing* it, and removing a value from the stack is called *popping*. Since most operations are performed on stack elements rather than on storage cells, no special *addressing modes* are necessary for each operation. Only two operations are required to exchange information with the main memory. The separate modes of these instructions may be implemented as separate instructions, too, thereby making the concept of different addressing modes completely redundant.

The virtual Tcode machine is a *von Neumann* architecture: it has an array of storage cells and a processor which processes the data stored in that array. The storage array is called the *data space* of the machine. The Tcode machine can interpret only one single program at a time. This program must be stored in a separate array called the *instruction space*. Both arrays are byte-addressable. A machine word is 16 bits wide on the Tcode machine. Therefore, neither array may be larger than 2^{16} bytes, because otherwise, the upper part of it (above 64K) would not be addressable by using a cell as a pointer. Machine words are stored at subsequent addresses with the most significant byte at the position with the lower address. This kind of representation is also known as *little endian* byte ordering.

3.1.1 The Registers of the Tcode Machine

Besides the instruction and data array, the machine possesses five *registers* which are required to interpret Tcode programs. These registers are called the *return register* (*RR*), the *instruction pointer* (*IP*), the *stack pointer* (*SP*), the *frame pointer* (*FP*), and the *instance context* (*SELF*).

RR is only used in procedure calls. It is loaded with the value at the top of the stack using the `POP` instruction. Its sole purpose is to transport the return value of a procedure to the caller. The `CLEAN` instruction removes the procedure arguments from the stack and then pushes the return register value back onto the stack.

IP points to the next instruction to interpret. Loading this register with a specific address is equal to performing a *jump* to that location in the instruction space.

SP points to the object which is currently stored at the top of the CPU's stack. An object is pushed onto the stack by first decrementing SP and then storing the value of the object at the address SP points to. The CPU stack is no separate memory location, but SP points into the program's data space. **Note**: SP is a word pointer. To access a storage location it points to, it must be scaled by multiplying it by two. Therefore, only full machine words may be stored on the stack.

FP points to the return address of the most recently called procedure. Like SP, FP is a word pointer which must be scaled to address machines words in memory. Its purpose is to address the base of the context of the currently executing procedure. In Tcode programs, *local addressing* means addressing relative to FP and all instructions which address 'local storage' (like `LDL`) use FP as their base register.

SELF is used to address data local to *objects*. It is saved and adjusted when a method is entered and restored when returning from a method. The value of the instance context is added to the addresses used in `LDI` and `SAVI` (load/save instance variable) instructions.

A special location called the **jump table** is used by the Tcode interpreter to access routines provided by the runtime environment. The table consists of so-called *slots* which contain the addresses of native procedures provided by the Tcode interpreter. The Tcode instruction *EXEC n* calls the procedure whose address is stored in slot *n* of the jump table. The calling conventions for Tcode routines and external procedures are the same.

The registers described here are not directly visible to Tcode programs, but they are modified implicitly by certain instructions. However, the registers are used to explain the semantics of the Tcode machine.

Figure 16 illustrates the architecture of the Tcode machine.

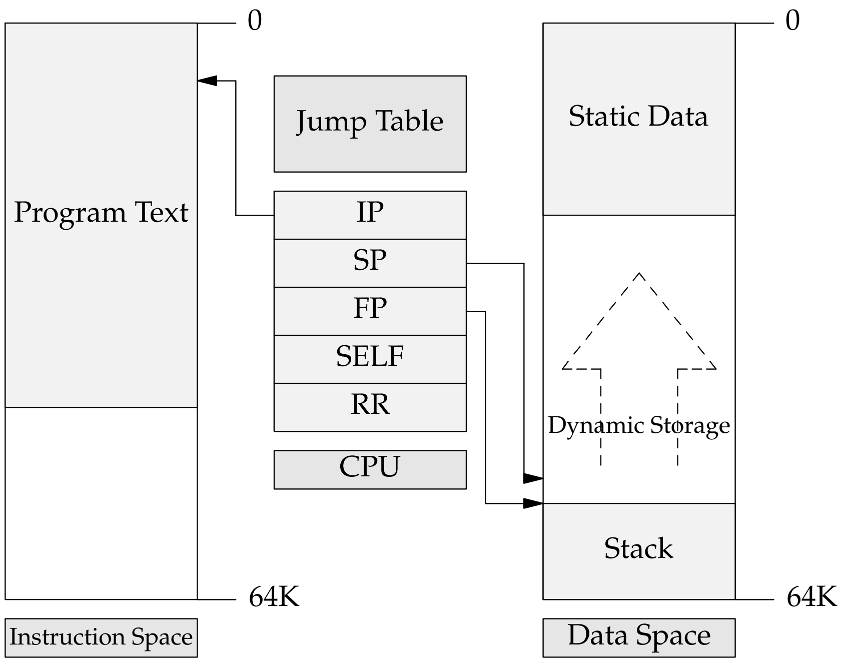


Figure 16: The Tcode Machine

3.2 Fundamental Definitions

This subsection defines some fundamental operations, which are required to formally describe the Tcode machine, in terms of basic mathematics.

T denotes the value of a true proposition.

F denotes the value of a false proposition.

n, *m*, and *e* denote numeric variables.

p and *q* denote truth values.

vn denotes a vector of the length *n*.

vn(i) denotes the *i*'th member of *vn*.

bn denotes a byte vector of the length *n*.

bn(i) denotes the *i*'th member of *vn*.

MEM denotes a vector of 65536 members.

n := *m* denotes that the value of *m* be copied to *n*.

mem(n) denotes the value of the *n*'th member of *MEM* modulo 256 for $n \in \{0, \ldots, 65535\}$.

mem(n) := *m* denotes that the *n*'th member of *MEM* be set to the value *m modulo 256*.

LBL denotes a vector of 32768 members.

lbl(n) denotes the value of the *n*'th member of *LBL* for $n \in \{0, \dots, 32767\}$.

lbl(n) := m denotes that the *n*'th member of *LBL* be set to *m*.

m+n denotes the sum of *m* and *m*.

m−n denotes the difference between *m* and *m*.

m·n denotes the product of *m* and *m*.

m/n denotes the integer part of the quotient of *m* and *n*. The notation $\frac{n}{m}$ is used as a synonym for *n/m*.

m^n denotes *m* raised to the *n*'th power.

wmem(n) denotes $mem(n{\cdot}2) + mem(n{\cdot}2+1){\cdot}256$ for $n \in \{0, \dots, 32767\}$.

wmem(n) := m denotes *mem(n·2) := m modulo 256* and *mem(n·2+1) := m / 256*.

m=n denotes the relation 'equal to'. The result of *m=n* is either *T* or *F*.

m<n denotes the relations 'less than'. The result of *m<n* is either *T* or *F*.

$\neg p$ denotes the logical complement of p (if p then F else T).

$m \neq n$ denotes $\neg(m=n)$.

$m>n$ denotes $n<m$.

$m \geq n$ denotes $\neg(m<n)$.

$m \leq n$ denotes $\neg(m>n)$.

$p \rightarrow q$ denotes 'p implies q' (if p then q).

$(p \rightarrow e1 ; e2)$ denotes $p \rightarrow e1$ and $\neg p \rightarrow e2$.

$p \wedge q$ denotes the logical AND: $(p \rightarrow q ; F)$.

$p \vee q$ denotes the logical OR: $(p \rightarrow T ; q)$.

n MOD m denotes $n - \frac{n}{m} \cdot m$

n OR m denotes the bitwise OR operation ($n, m \in \{0, \dots, 65535\}$) :

$$\sum_{i=0}^{15} \left[((\frac{n}{2^i}\ MOD\ 2) + (\frac{m}{2^i}\ MOD\ 2) > 0) \rightarrow 2^i ; 0 \right].$$

n AND m denotes the bitwise AND operation ($n, m \in \{0, \dots, 65535\}$) :

$$\sum_{i=0}^{15} \left[(\frac{n}{2^i}\ MOD\ 2) \cdot (\frac{m}{2^i}\ MOD\ 2) \cdot 2^i \right].$$

n XOR m denotes bitwise exclusive OR operation:

$$\sum_{i=0}^{15} \left[((\frac{n}{2^i}\ MOD\ 2) = (\frac{m}{2^i}\ MOD\ 2)) \rightarrow 0 ; 2^i \right].$$

s(x) denotes the *sign* of *x*: $(x \geq 32768 \rightarrow T ; F)$.

v(x) denotes the *value* of *x*: $(s(x) \rightarrow (0-(65536-x)) ; x)$.

w(x) denotes *((x<0) → (65536+x) ; x)*. This encodes x in 16-bit two's complement notation.

L denotes the next unused location in the data space. It is initially set to 0.

U_n denotes $wmem(SP+n)$. This is the n'th element of the stack as an *unsigned value*. U_0 is the word most recently pushed to the stack.

S_n denotes $v(U_n)$. This is the same object as U_n but as a *signed value*.

$\perp$ denotes an *undefined* value.

3.3 Notation

The notation

Instruction Arguments ::= { expr , ... }

is used to describe the semantics of each Tcode instruction in the following subsections. The lefthand side of the `::=` sign reads the name of the desribed instruction and its (optional) arguments. The righthand side contains a list of zero of more expressions as described in the previous subsections. The expressions are separated by commas (,) and delimited by braces ({...}). Braces may also be used to indicate that an operation be applied to a set of instructions instead of a single instruction. For example

{ for each i=0...32767 { mem(i) := 0, lbl(i) := 0 } }

would mean to fill the first 32768 members of the two vectors *MEM* and *LBL* with zeroes.

3.4 Declarations

CLAB n ::= { lbl(n) := IP }

DLAB n ::= { lbl(n) := L }

DECL n ::= { (n = 0) → (L := L+2) ; (L := L+n·2) }

DATA n ::= { wmem(L/2) := n , L := L+2 }

CREF n ::= { wmem(L/2) := lbl(n) , L := L+2 }

DREF n ::= { CREF n }

STR n vn ::= { for each i=0...n−1 { wmem(L/2) := vn(i) , L := L+2 } ,
wmem(L/2) := 0 , L := L+2 }

PSTR n bn ::= { for each i=0...n−1 { mem(L) := bn(i) , L := L+1 } ,
mem(L) := 0 , L := L+1 ,
(L MOD 2 ≠ 0) → { mem(L) := 0 , L := L+1 } }

INIT n m ::= {}

PUB m n vn ::= {} †

EXT m n vn ::= $\perp$ †

HINT n ::= {} ‡

LINE n ::= {}

GSYM m n vn ::= {}

LSYM m n vn ::= {}

ISYM m n vn ::= {}

† *PUB* and *EXT* are used for external linking. They ar explained informally later in this chapter.
‡ *HINT* is used to exchange meta information between compiler phases.

3.5 Arithmetic

NOP ::= {}

$NEG ::= \{ U_0 := w(0 - S_0) \}$

$LNOT ::= \{ U_0 := w((S_0 \rightarrow 0 : -1)) \}$

$BNOT ::= \{ U_0 := w(65536 - S_0) \}$

$$\text{MUL} ::= \left\{ U_1 := w(([s(S_1) = s(S_0)] \rightarrow f(U_1 \cdot U_0) \; ; 0 - f(U_1 \cdot U_0))) \; , SP := SP + 1 \right\}$$

where $f(x) := x \; MOD \; 65536$.

$$DIV ::= \left\{ U_1 := w(([s(S_1) = s(S_0)] \rightarrow \frac{U_1}{U_0} \; ; 0 - (\frac{U_1}{U_0}))) \; , SP := SP + 1 \right\}$$

$MOD ::= \{ U_1 := w(U_1 \; MOD \; U_0) \; , SP := SP + 1 \}$

$ADD ::= \{ U_1 := w(U_1 + U_0) \; , SP := SP + 1 \}$

$SUB ::= \{ U_1 := w(U_1 - U_0) \; , SP := SP + 1 \}$

$BAND ::= \{ U_1 := U_1 \; AND \; U_0 \; , SP := SP + 1 \}$

$BOR ::= \{ U_1 := U_1 \; OR \; U_0 \; , SP := SP + 1 \}$

$BXOR ::= \{ U_1 := U_1 \; XOR \; U_0 \; , SP := SP + 1 \}$

$$BSHL ::= \left\{ U_1 := w(U_1 \cdot 2^{U_0} \; MOD \; 65536) \; , SP := SP + 1 \right\}$$

$$BSHR ::= \left\{ U_1 := w(\frac{U_1}{2^{U_0}} \; MOD \; 65536) \; , SP := SP + 1 \right\}$$

$EQU ::= \{ U_1 := ((U_1 = U_0) \rightarrow w(-1); 0) \; , SP := SP + 1 \}$

$NEQU ::= \{ U_1 := ((U_1 \neq U_0) \rightarrow w(-1); 0) \; , SP := SP + 1 \}$

$LESS ::= \{ U_1 := ((S_1 < S_0) \rightarrow w(-1); 0) , SP := SP + 1 \}$

$GRTR ::= \{ U_1 := ((S_1 > S_0) \rightarrow w(-1); 0) , SP := SP + 1 \}$

$LTEQ ::= \{ U_1 := ((S_1 \leq S_0) \rightarrow w(-1); 0) , SP := SP + 1 \}$

$GTEQ ::= \{ U_1 := (S_1 \geq S_0) \rightarrow w(-1); 0) , SP := SP + 1 \}$

$UMUL ::= \{ U_1 := w(U_1 \cdot U_0 \ MOD\ 65536) , SP := SP + 1 \}$

$$UDIV ::= \left\{ U_1 := w(\frac{U_1}{U_0} \ MOD\ 65536) , SP := SP + 1 \right\}$$

$ULESS ::= \{ U_1 := ((U_1 < U_0) \rightarrow w(-1); 0) , SP := SP + 1 \}$

$UGRTR ::= \{ U_1 := ((U_1 > U_0) \rightarrow w(-1); 0) , SP := SP + 1 \}$

$ULTEQ ::= \{ U_1 := ((U_1 \leq U_0) \rightarrow w(-1); 0) , SP := SP + 1 \}$

$UGTEQ ::= \{ U_1 := ((U_1 \geq U_0) \rightarrow w(-1); 0) , SP := SP + 1 \}$

3.6 Memory

$$LDG\ n ::= \left\{ SP := SP - 1 , U_0 := wmem(\frac{n}{2}) \right\}$$

$LDGV\ n ::= \{ SP := SP - 1 , U_0 := n \}$

$LDL\ n ::= \{ SP := SP - 1 , U_0 := wmem(FP - n) \}$

$LDLV\ n ::= \{ SP := SP - 1 , U_0 := (FP - n) \cdot 2 \}$

$$LDI\ n ::= \left\{ SP := SP - 1 , U_0 := wmem(\frac{SELF}{2} + n) \right\}$$

$LDIV\ n ::= \{ SP := SP - 1 , U_0 := SELF + n \cdot 2 \}$

$LDLAB\ n ::= \{ SP := SP - 1 , U_0 := lbl(n) \}$

$NUM\ n ::= \{ LDGV\ n \}$

$SELF ::= \{ SP := SP - 1 , U_0 := SELF \}$

$$SAVG\ n ::= \left\{ wmem(\frac{n}{2}) := U_0 , SP := SP + 1 \right\}$$

$SAVL\ n ::= \{ wmem(FP - n) := U_0 , SP := SP + 1 \}$

$$SAVI\ n ::= \left\{ wmem(\frac{SELF}{2} + n) := U_0 , SP := SP + 1 \right\}$$

$$STORE ::= \left\{ wmem(\frac{U_1}{2}) := U_0 , SP := SP + 2 \right\}$$

$STORB ::= \{ mem(U_1) := U_0 , SP := SP + 2 \}$

$INCG\ n\ m ::= \{ wmem(n) := wmem(n) + m \}$

$INCL\ n\ m ::= \{\ wmem(FP - n) := wmem(FP - n) + m\ \}$

$INCI\ n\ m ::= \left\{\ wmem(\frac{SELF}{2} + n) := wmem(\frac{SELF}{2} + n) + m\ \right\}$

$DEREF ::= \left\{\ U_0 := wmem(\frac{U_0}{2})\ \right\}$

$DREFB ::= \{\ U_0 := mem(U_0)\ \}$

$NORMB ::= \{\ ADD\ \}$

$NORM ::= \{\ U_1 := U_1 + U_0 \cdot 2\ ,\ SP := SP + 1\ \}$

$DUP ::= \{\ SP := SP - 1\ ,\ U_0 := U_1\ \}$

$SWAP ::= \{\ x := U_1\ ,\ U_1 := U_0\ ,\ U_0 := x\ \}$

3.7 Procedures

$HDR ::= \{\ SP := SP - 1\ ,\ U_0 := FP\ ,\ FP := SP\ \}$

$END ::= \{\ FP := U_0\ ,\ IP := U_1\ ,\ SP := SP + 2\ \}$

$MHDR ::= \{\ SP := SP - 2\ ,\ U_1 := FP\ ,\ FP := SP\ ,\ U_0 := SELF\ ,\ SELF := wmem(FP + 2)\ \}$

$ENDM ::= \{\ SELF := U_0\ ,\ FP := U_1\ ,\ IP := U_2\ ,\ SP := SP + 3\ \}$

$STACK\ n ::= \{\ SP := SP - n\ \}$

$CLEAN\ n ::= \{\ STACK\ 0{-}n\ \}$

$CALL\ n ::= \{\ SP := SP - 1\ ,\ U_0 := IP\ ,\ IP := lbl(n)\ \}$

$CALR ::= \{\ x := U_0\ ,\ U_0 := IP\ ,\ IP := x\ \}$

$EXEC\ n ::=$ †

$CALX\ n ::= \perp$ ‡

$POP ::= \{\ RR := U_0\ ,\ SP := SP + 1\ \}$

† The semantics of *EXEC* are explained in a later subsesction.
‡ *CALX* is used for external linkage. It is explained later in this chapter.

3.8 Branches

$BRF\ n ::= \{ (U_0 = 0) \rightarrow (IP := lbl(n))\ ,\ SP := SP + 1 \}$

$BRT\ n ::= \{ (U_0 \neq 0) \rightarrow (IP := lbl(n))\ ,\ SP := SP + 1 \}$

$NBRF\ n ::= \{ (U_0 = 0) \rightarrow (IP := lbl(n)) \}$

$NBRT\ n ::= \{ (U_0 \neq 0) \rightarrow (IP := lbl(n)) \}$

$JUMP\ n ::= \{ IP := lbl(n) \}$

$UNEXT\ n ::= \{ (U_1 \geq U_0) \rightarrow (IP := lbl(n))\ ,\ SP := SP + 2 \}$

$DNEXT\ n ::= \{ (U_1 \leq U_0) \rightarrow (IP := lbl(n))\ ,\ SP := SP + 2 \}$

$HALT ::=$ †

† The semantics of HALT are explained in the following subsesction.

3.9 EXEC and HALT

The Tcode instructions *EXEC* and *HALT* cannot be formally described by means of the terms used in the previous subsections. Therefore, they will be described informally here.

The *HALT* instruction stops the execution of the Tcode program which is currently executing on the machine. Therefore, its result is not in the domain of the Tcode machine. On the other hand, it differs from a program like

```
CLAB 0 JUMP 0
```

(which implements an infinite loop and therefore results in $\perp$), in the point that its execution indicates the (successful) *termination* of a Tcode program.

The *EXEC* instruction provides an interface between the Tcode machine and the environment in which the Tcode machine is running. Some Tcode interpreters may provide some functionality which is not covered by the formal description in this section (for example, the runtime support routines used by T3X programs). To make such additional functions available to Tcode programs, the function is coded in a language which is suitable for this task, and then its address is placed at a fixed location of the *jump table* of the Tcode machine. The instruction *EXEC n* will perform a jump to the routine whose address has been stored at location *n* in the jump table. The function may not take any arguments and its return value will be ignored. To communicate with the Tcode program, the argument stack of the Tcode machine must be used. See the section on *calling conventions* in a later chapter for details.

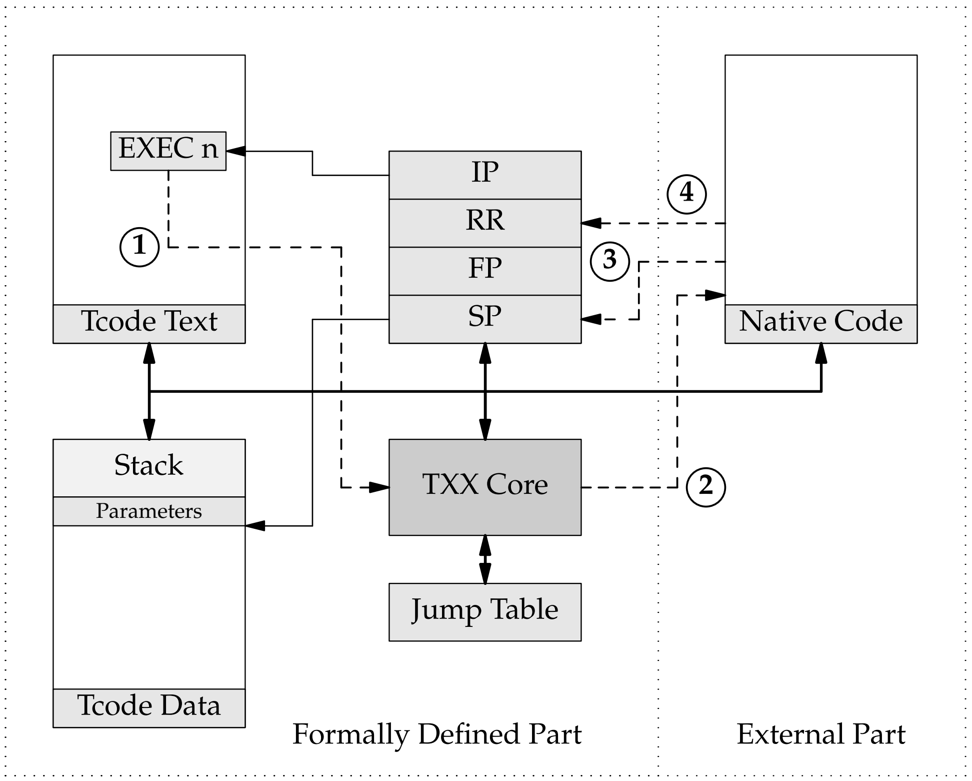


Figure 17: Calling an Extension Procedure

Figure 17 illustrates the linkage between the formally described Tcode machine and an extension procedure which has been implemented as a part of the virtual Tcode machine. When the Tcode interpreter encounters an *EXEC* instruction (1), it takes its numeric argument as an index into its internal jump table and looks up the requested extension procedure. It then performs an indirect call to that procedure (2). No arguments are passed to the procedure and its return value will be ignored. All communiation between the extension procedure and the Tcode program is done via the runtime stack of the Tcode program. Because the extension routine is part of the Tcode machine, it may access all internal registers of the machine. By convention, however, it uses only the registers *SP* and *RR*. When the procedure is entered, *SP* points to the *last* argument passed to the *EXEC*uted procedure (3), *DataSpace[SP+1]* points to the second to last parameter, and so on (assuming that *DataSpace* is a 16-bit word vector containing the data area of the Tcode machine). Because arguments are passed in reverse order, the number of arguments must be known in advance by the called procedure. Before returning, each extension routine should place a meaniningful value in *RR* (4). This value will be placed on the stack by the *CLEAN* instruction which usually follows each *EXEC* command.

3.10 External Linkage

The *EXT*, *PUB*, and *CALX* instructions provide a mechanism for binding separately compiled modules together. External references are limited to procedure calls. This means that a module can call (public) procedures defined in another module, but it cannot access the data of an external module.

In order to call an external procedure, the label which tags the entry point of the routine must be declared public (using a *PUB* instruction) in the module containing *callee*. In the module of the *caller*, it must be declared extern using *EXT*. The extern declaration creates a so-called external label which may be referenced in *CALX* instructions. *CALX* is used to call an external procedure.

The *PUB* instruction provides a symbolic name for a procedure. This symbolic name may be referenced by *EXT* records in a external modules. *CALX* references an *EXT* record in the same module. An external reference is resolved in four steps. The following steps are performed for each *CALX* instruction in a program:

1) The operand of the *CALX* instruction is looked up in the external symbol table (a table holding the *EXT* records).

2) The name contained in the matching *EXT* record is looked up in the public symbol table (a table holding the *PUB* records).

3) The label contained in the matching *PUB* record replaces the external label in the *CALX* instruction.

4) The *CALX* instruction is replaced with *CALL*.

Since labels are represented by integers in Tcode, label collisions *will* occur when binding two (or more) Tcode modules together. Therefore, labels must be renamed: When a module *A* already has been loaded when a module *B* is to be loaded, the highest label ID used in *A* should be added to each *non-external* label in *B*.

Two or more *EXT* records with the same name may exist, because the same symbol may be associated with different external labels in different modules.

The existance of two *PUB* records with the same name is considered an error (redefinition error).

There *must* be a matching *PUB* record for each *EXT* record. Otherwise, an *unresolved external reference* error is to be signalled.

External linkage is illustrated in figure 18.

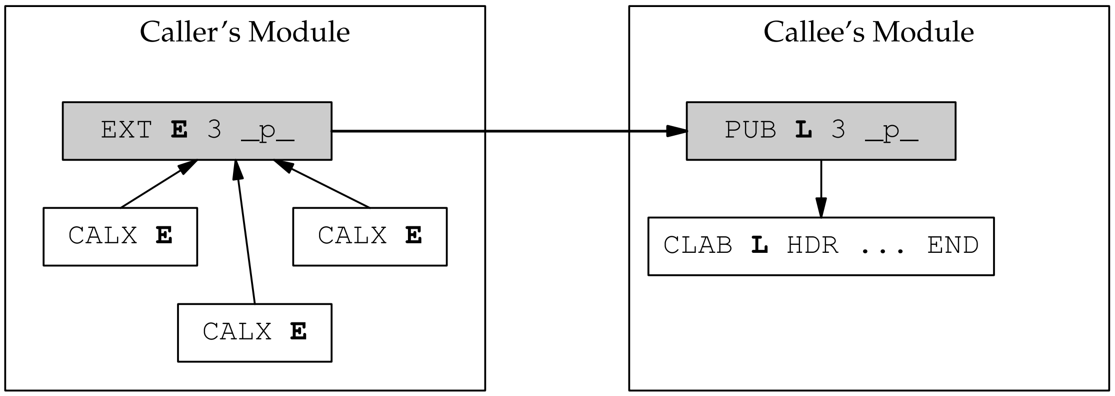


Figure 18: External Linkage

3.11 Programming Conventions

Of course, the Tcode machine is a very flexible tool which can be programmed in many different ways. However, an automatic translator – like the T3X compiler – has to use some well-defined rules the generated code has to comply with. These rules will be defined in this subsection.

The **memory model** used by T3X programs is the one illustrated in figure 18. The low memory cells are filled with the program's static data. The stack grows from the top of the memory towards the static data area. Dynamic storage is allocated on the stack. The Tcode instructions assume that the stack grows *down* (towards lower addresses), so this is not just a convention, but a invariant property of the Tcode machine. Technically, one says that the machine has a *push down stack*, since pushing an object *decreases* SP.

Since the stack grows down, dynamic memory is allocated by *subtracting* the number of required machine words from SP and it is released again by *adding* the same amount to SP. The `STACK` instruction is used to modify the stack pointer directly. See Tcode instruction set for details.

The **calling conventions** of a programming language define the way a program passes arguments to a procedure and how the procedure returns its result (if any). In T3X, arguments are placed on the stack. The called procedure returns its result in the return register RR. The caller is responsible for removing the arguments from the stack after the call returns. The `CLEAN` instruction is used to remove arguments *and* put the content of RR back on the stack so that it is available to subsequent computations.

Storage, local and **parameters** are addressed relative to the FP register in procedures. When a procedure is called, the `CALL` or `CALR` instruction pushes the return address (the address of the instruction following the call) onto the stack so that execution can be resumed later at this place. The first thing a procedure has to do is to save the FP register by pushing it onto the stack. This is done by a `HDR` instruction which introduces each Tcode procedure. Note that this is not optional, since the `END` instruction – which performs a return to the caller – restores both, the frame pointer *and* the instruction pointer. Besides saving the context of the calling routine, `HDR` also creates the new context (aka stack frame) by copying the value of SP into FP.

The stack frame of the procedure fragment

```
P(a,b) DO VAR i[3]; ... END
```

when called through the statement

```
P(X, Y);
```

can be seen in figure 19.

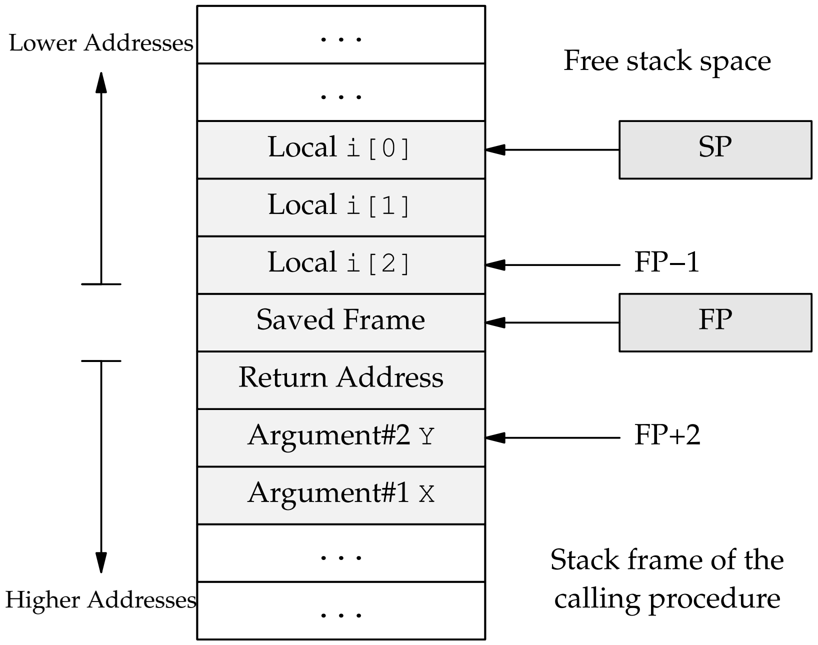


Figure 19: A Stack Frame with Arguments

SP points to the end of the dynamically allocated variables so that operands of the instructions of *P()* will be placed at lower addresses and therefore will not interfere with any objects declared inside the procedure. The stack frame of the procedure *P()* is itself placed 'on top' of the stack frame of the calling procedure. This way, each procedure creates its own private context which cannot interfere with stack frames of calling procedures and not even with contexts of calling instances of the same procedure, as it happens case in recursive algorithms.

The arguments are delivered to procedures in *reverse order*. Therefore, the *last* argument is located at the lowest address 'above' the saved return address. The last argument can be accessed by the procedure using the instruction

```
LDL 2
```

(LoaD Local) which loads the content of the cell pointed to by FP+2. The local variables of a procedure are located 'below' the link information formed by the saved SP and FP registers, because the link information is pushed after computing the argument values, but before allocating local memory. The address of the vector `i` would be FP–3 in the figure. The addresses of local vectors have to be computed on demand at run time, since they are relative to the value of FP. The instruction

```
LDLV -3
```

(LoaD Local Vector address) could be used to compute the absolute address of `i` in the example. For details on the load instructions, see the Tcode instruction set.

The contexts of methods differ from those of 'ordinary' procedures, because in addition to the link information of procedures, methods also save the instance contexts of their callers. Therefore, methods begin with an `MHDR` (method header) instruction and end with an `ENDM` (end method) instruction. The stack frame maintained by these instructions can be found in figure 20.

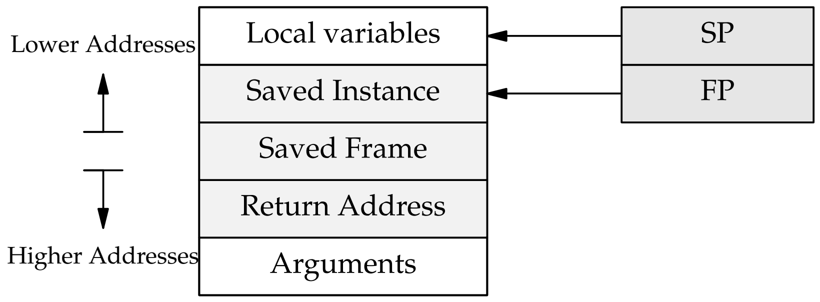


Figure 20: A Method Stack Frame

3.12 Startup Conditions

When the Tcode machine starts up, the following steps always will be performed. Hence, a Tcode program may expect the resulting conditions each time it will be loaded into the machine.

The instruction and data space is allocated. The amount of space allocated for each array depends on the implementation of the machine. The state of the single cells of the arrays is undefined. However, the instruction space cells from zero to the size of the Tcode program will be filled with the desired program and the first cells of the data space will be filled with data defined by that program.

The stack pointer is set to the highest available cell (divided by two, since it is a word pointer), so the stack will grow down from the top of the memory towards the static data area. The frame pointer is set to the value of the stack pointer. Therefore, the main program cannot have any parameters, but it may have local variables.

The value of the return register is undefined at this point.

The value of the instance context is undefined at this point.

The machine finally will be started by loading the instruction pointer with zero and entering the interpretation loop. Tcode programs always have their initial entry point at location zero.

4
The T3X Translator

This chapter contains a detailed description of the T3X translator TXTRN including its full source code with additional comments. TXTRN is the part of the compiler which translates T3X into *Tcode*. It is a one-pass translator based upon the simplified model described at the end of the previous chapter. It is itself written in T3X.

4.1 The Commented TXTRN Listing

```
!       TXTRN -- a T3XR6->Tcode translator
!       Copyright (C) 1997-2001 Nils M Holm. All rights reserved.
!       See the file LICENSE for conditions of use.

module txtrn(t3x, tcode);

object  t[t3x];
```

Table 30 summarizes the tunable parameters of TXTRN. It contains the minimum, maximum, and default value for each symbol as well as a short description. An entry reading 'dynamic' denotes that when increasing this value, other parameters may have to be decreased. By default, the translator parameters have been set up to save space. The settings are exactly sufficient to bootstrap the compiler.

Symbol	**Min**	**Max**	**Dflt**	**Meaning**
`BUFSIZE`	129	2050	1026	Input buffer size
`OBUFL`	129	1024	1024	Output buffer size
`SYMBSPACE`	3072	dynamic	3072	Space for symbol table entries
`CACHESPACE`	2048	dynamic	4096	Space class directory cache
`NBSPACE`	8192	dynamic	8192	Space for symbol names
`TEXTLEN`	129	257	129	Max. length of a token
`MAXTBL`	128	dynamic	128	Max. members per table

Table 30: TXTRN: Tunable Parameters

```
const   BUFSIZE=        1026,   ! input buffer size
        OBUFL=          1024,   ! output buffer size
        SYMBSPACE=      3072,   ! symbol table size
        CACHESPACE=     4096,   ! class directory cache size
        NSPACE=         8192,   ! name pool size
        TEXTLEN=        129,    ! max. token length
        MAXTBL=         128;    ! max. member per table
```

Setting TUNE16 = 1 speeds up some things on true little endian 16-bit machines.

```
const   TUNE16 = 0;
```

This symbol is used to flag escaped characters in strings and character literals.

```
const   META=   256;
```

These constants are used to describe the types of symbol table entries.

```
const   CNST=   1,      ! Constant
        GLOBL=  2,      ! Global symbol
        PROC=   4,      ! Procedure definition
        PROTO=  8,      ! Procedure declaration
        SYS=    16,     ! System procedure
        PUBLC=  32,     ! Public
        EXTRN=  64,     ! External
        CLSS=   128,    ! Class
        OBJCT=  256,    ! Object
        IVAR=   512,    ! Instance variable
        REDIR=  1024,   ! Redirection list (internal)
        LINKD=  2048;   ! Resolved ext. reference (internal)
```

The following constants describe the layout of a symbol table entry (a symbol record). The `SSIZE` field contains the size of vector objects or zero, if the described object is atomic. `SVAL` contains the addresses of variables and the values of constants. For procedure objects, the `SVAL` field holds the address and `SSIZE` contains the procedure's arity. In class entries, `SSIZE` contains the size of the class in machine words and in object entries, it contains a pointer to the class entry.

```
struct SYMREC = SNAME,
                SVAL,   ! CNST: value           other: address
                SSIZE,  ! CLSS: elements        OBJCT: ptr to class
                SFLAGS;
```

This is a byte-order-independent version of SYMREC. It is used to export and import class directories. The public scope consists of records of the type `SR2`.

```
struct SR2 =    SNL, SNH,
                SVL, SVH,
                SZL, SZH,
                SFL, SFH;
```

These are pointers to and buffers for paths to be searched for classes. They are initialized in `initvars`.

```
var     Classpaths,     ! Class paths
        Syscp::TEXTLEN, ! System CP
        Usrcp::TEXTLEN; ! User CP
```

In the following section, all global storage used by the translator will be defined.

```
var     Symbols[SYMBSPACE],             ! Symbol table,
        St;                             ! Pointer to first free entry
var     Libcache[CACHESPACE+1],         ! Class directory cache,
        Lt;                             ! Pointer to end of cache
var     Sym_equ, Sym_mod, Sym_sub,      ! Pointers to frequently used
        Sym_add, Sym_mul, Sym_bar,      ! operators
        Sym_not;
var     Names::NSPACE,                  ! Symbol name pool,
        Nt;                             ! Pointer to free space of pool
var     Line;                           ! Input line number
var     File::TEXTLEN;                  ! Input file name
var     Modname::TEXTLEN,               ! Module name
        Modctx;                         ! Pointer to module context
var     Errcount;                       ! Error counter
var     Token;                          ! Current token
var     Text::TEXTLEN;                  ! Token text
var     Value;                          ! Token value
var     Op;                             ! Current operator
var     Label, Xlabel;                  ! Next label / external label
var     Startlab,                       ! Loop context: start and
        Exitlab;                        ! end labels
var     Endlab;                         ! Procedure context: end label
var     Looplevl;                       ! Locladdr of loop context
var     Locladdr;                       ! Next free local address
var     Buffer::BUFSIZE;                ! Input buffer
var     Cp,                             ! Pointer to current character
        Ep;                             ! Pointer to end of buffer
var     Lowp, Nomore;                   ! Low water mark, EOF flag
var     Obuf::OBUFL, Obp;               ! Output buffer, Pointer to free space
var     Last[9], LL;                    ! Delay queue, length
var     Ops;                            ! Operator table
var     Bcslot;                         ! INTERFACE slot (obsolete)
var     Debug, Packstr;                 ! #debug, #packstrings flags
var     Classctx;                       ! Current class context
var     Classoff;                       ! Next free instance address
var     Selfobj;                        ! Self
var     Method_adjust;                  ! Method adjustment flag (1=in method)
var     K64;                            ! 65536 constant
var     Big;                            ! Enable Big generator
var     Initdone;                       ! Init sequence emitted?
```

The following constants are token values which will be delivered by the scanner.

```
const   ENDFILE=%1;

const   SYMBOL=0, NUMBER=1, STRING=2;

const   BINOP=10, DISOP=11, CONOP=12, UNOP=20, ADDROP=21, LPAREN=30,
        RPAREN=31, LBRACK=40, RBRACK=41, SEMI=50, COMMA=60, ASSIGN=70,
        COND=80, COLON=81, BYTEOP=90, DOT=98, METAOP=99;

const   KCALL=100, KCLASS=101, KCONST=102, KDECL=103, KDO=104, KELSE=105,
        KEND=106, KFOR=107, KHALT=108, KIE=109, KIF=110, KIFACE=111,
        KLEAVE=112, KLOOP=113, KMODULE=114, KOBJECT=115, KPACKED=116,
        KPUBLIC=117, KRETURN=118, KSEND=119, KSTRUCT=120, KVAR=121,
        KWHILE=122;
```

This is the layout of an operator table entry (operator structure).

```
struct  OPER =
        OPREC,  ! Precedence (greater is higher)
        OLEN,   ! Length of operator symbol
        ONAME,  ! Pointer to operator symbol
        OTOK,   ! Token for this operator
        OCODE;  ! Associated Tcode instruction
```

These procedures are involved in mutual recursion.

```
decl    fatal(2), factor(0), expr(0), stmt(0), compound(0), declaration(1);
```

initvars() – $\Rightarrow$ 0
Initialize the global variables.

```
initvars() do var k;
        ! The entries of Classpaths will be searched for classes.
        ! The last two entries are hardwired.
        Classpaths := [
                packed"",       ! Empty slot for #classpath
                packed".",      ! Try current directory
                packed"",       ! Reserved for $T3XDIR/classes
                packed"/usr/local/t3x/r6/classes",
                packed"C:/v/t3xr6/classes",
                0
        ];
        ! Set up the system class path
        k := t.getenv(packed"T3XDIR", Syscp, TEXTLEN-9);
        if (k > 0) do
                t.memcopy(@Syscp::k, packed"/classes", 9);
                Classpaths[2] := Syscp;
        end
        St := 0;        ! Clear symbol table
        Lt := 0;        ! Clear Library cache
        Nt := 0;        ! Clear name pool
        Line := 1;      ! Reset line counter
        File::0 := 0;   ! No input file name (reading stdin)
        Modname::0 := 0;        ! Clear module name
        Modctx := %1;   ! Reset module context
        Errcount := 0;  ! Clear error counter
        Label := 2;     ! First label to be generated
        Xlabel := 1;    ! First external label to be generated
        Startlab := 0;  ! Invalidate loop/procedure context
        Endlab := 0;
        Exitlab := 0;
        Looplevl := 0;
        Locladdr := 1;  ! First local address to use
        Cp := 0;        ! Reset input pointers
        Ep := 0;
        Lowp := 0;
        Nomore := 0;
        Obp := 0;       ! Clear output buffer
        LL := 0;        ! Clear delay queue
        Bcslot := 0;    ! Mark all slots free
        Debug := 0;     ! Disable debugging
        Packstr := 0;   ! Do not pack strings by default
        Classctx := %1; ! Clear class context
        Method_adjust := 0;     ! Reset emthod flag
        K64 := 16384;   ! non-16-bit hack, avoid constant folding
        K64 := K64*4;
        Big := 0;       ! Do not emit 32-bit Tcode
        Initdone := 0;  ! Initialization not yet done
        ! This table contains all operators known to the compiler.
        ! It is used for lexical analysis and falling precedence
        ! parsing.
        ! Do not change the order of this table - see findop()
        Ops := [
                [ 6, 1, packed"+",      BINOP,  tcode.IADD      ],
                [ 7, 1, packed"*",      BINOP,  tcode.IMUL      ],
                [ 0, 1, packed";",      SEMI,   0               ],
                [ 0, 1, packed",",      COMMA,  0               ],
                [ 0, 1, packed"(",      LPAREN, 0               ],
                [ 0, 1, packed")",      RPAREN, 0               ],
                [ 0, 1, packed"[",      LBRACK, 0               ],
                [ 0, 1, packed"]",      RBRACK, 0               ],
                [ 3, 1, packed"=",      BINOP,  tcode.IEQU      ],
                [ 5, 1, packed"&",      BINOP,  tcode.IBAND     ],
                [ 5, 1, packed"|",      BINOP,  tcode.IBOR      ],
                [ 5, 1, packed"^",      BINOP,  tcode.IBXOR     ],
                [ 0, 1, packed"@",      ADDROP, 0               ],
                [ 0, 1, packed"~",      UNOP,   tcode.IBNOT     ],
                [ 0, 1, packed":",      COLON,  0               ],
                [ 0, 2, packed"::",     BYTEOP, 0               ],
                [ 0, 2, packed":=",     ASSIGN, 0               ],
                [ 0, 1, packed"\\",     UNOP,   tcode.ILNOT     ],
                [ 1, 2, packed"\\/",    DISOP,  0               ],
                [ 3, 2, packed"\\=",    BINOP,  tcode.INEQU     ],
                [ 4, 1, packed"<",      BINOP,  tcode.ILESS     ],
                [ 5, 2, packed"<<",     BINOP,  tcode.IBSHL     ],
                [ 4, 2, packed"<=",     BINOP,  tcode.ILTEQ     ],
                [ 4, 1, packed">",      BINOP,  tcode.IGRTR     ],
                [ 5, 2, packed">>",     BINOP,  tcode.IBSHR     ],
                [ 4, 2, packed">=",     BINOP,  tcode.IGTEQ     ],
                [ 6, 1, packed"-",      BINOP,  tcode.ISUB      ],
                [ 0, 2, packed"->",     COND,   0               ],
                [ 7, 1, packed"/",      BINOP,  tcode.IDIV      ],
                [ 2, 2, packed"/\\",    CONOP,  0               ],
                [ 0, 1, packed".",      DOT,    0               ],
                [ 7, 2, packed"./",     BINOP,  tcode.IUDIV     ],
                [ 7, 2, packed".*",     BINOP,  tcode.IUMUL     ],
                [ 4, 2, packed".<",     BINOP,  tcode.IULESS    ],
                [ 4, 3, packed".<=",    BINOP,  tcode.IULTEQ    ],
                [ 4, 2, packed".>",     BINOP,  tcode.IUGRTR    ],
                [ 4, 3, packed".>=",    BINOP,  tcode.IUGTEQ    ],
                [ 0, 1, packed"#",      METAOP, 0               ],
                [ 7, 3, packed"mod",    BINOP,  tcode.IMOD      ],
                [ 0, 0, 0,      0,      0                       ]
        ];
end
```

length(a) – string ⇒ number
Return the number of characters stored in the string **a** excluding the terminating NUL word.

```
length(a) return t.memscan(a, 0, 32767);
```

strequ(a,b) – string,string ⇒ number
Compare the strings pointed to by **a** and **b** and return the difference of the character values at their first differing position. A return value of **0** means that the strings are equal.

```
strequ(a, b) return \t.memcomp(a, b, length(a)+1);
```

strcpy(a,b) – string,string ⇒ number
Copy the string (including the terminating NUL) pointed to by **b** to the location pointed to by **a**. Return the number of characters copied (excluding the NUL).

```
strcpy(a, b) do var k;
        k := length(b);
        t.memcopy(a, b, k+1);
        return k;
end
```

writep(f,s) – fdesc,string ⇒ number
Write a packed string to the file descriptor f **f**.

```
writep(f, s) t.write(f, s, length(s));
```

nl(f) – fdesc ⇒ 0
Write a newline sequence to the file descriptor **f**.

```
nl(f) do var b::3;
        t.write(f, t.newline(b), length(b));
end
```

ntoa(v,w) – number,number ⇒ string
Convert the number **v** to a string. Add leading spaces to make the string **w** characters wide. Return a pointer to a static buffer containing the string.

```
var     ntoa_buf::32;
ntoa(v, w) do var g, i;
        g := 0;
        if (v<0) do
                g := 1;
                v := -v;
        end
        ntoa_buf::31:= 0;
        i := 30;
        while (v \/ i = 30) do
                ntoa_buf::i := v mod 10 + '0'; i := i-1;
                v := v/10;
                w := w-1;
        end
        if (g) do
                ntoa_buf::i := '-'; i := i-1;
                w := w-1;
        end
        while (w>0) do
                ntoa_buf::i := '\s'; i := i-1;
                w := w-1;
        end
        return @ntoa_buf::(i+1);
end
```

dumptbl(tbl,lim) – vector,number ⇒ 0
Internat debug function: dump the first **lim** entries of the symbol table **tbl** on the standard error stream.

```
dumptbl(tbl, lim) do var i, y, err;
        for (i=0, lim, SYMREC) do
                y := @tbl[i];
                err := T3X.SYSERR;
                writep(err, ntoa(y[SVAL], 6));
                writep(err, ntoa(y[SSIZE], 6));
                writep(err, "\s");
                writep(err, y[SNAME]);
                writep(err, packed" (");
                writep(err, y[SFLAGS] & CNST-> packed" CNST": packed"");
                writep(err, y[SFLAGS] & GLOBL-> packed" GLOBL": packed"");
                writep(err, y[SFLAGS] & PROC-> packed" PROC": packed"");
                writep(err, y[SFLAGS] & PROTO-> packed" PROTO": packed"");
                writep(err, y[SFLAGS] & SYS-> packed" SYS": packed"");
                writep(err, y[SFLAGS] & PUBLC-> packed" PUBLC": packed"");
                writep(err, y[SFLAGS] & EXTRN-> packed" EXTRN": packed"");
                writep(err, y[SFLAGS] & CLSS-> packed" CLSS": packed"");
                writep(err, y[SFLAGS] & OBJCT-> packed" OBJCT": packed"");
                writep(err, y[SFLAGS] & IVAR-> packed" IVAR": packed"");
                writep(err, y[SFLAGS] & REDIR-> packed" REDIR": packed"");
                writep(err, packed" )\n");
        end
end
```

error(m,s) – string,string ⇒ 0
Report an error to the standard error stream in the following format:

```
TXTRN: line: [file:] message[: string]
```

File is only printed, if the first character of the input file name is non-zero (when processing an included file) or a module name was specified in a `MODULE` header. The string pointed to by **s** is only printed, if **s** is non-zero. **M** holds the message itself. The error counter is incremented.

```
error(m, s) do var o, col;
        col := packed": ";
        writep(T3X.SYSERR, packed"TXTRN: ");
        ie (File::0) do
                writep(T3X.SYSERR, File); writep(T3X.SYSERR, col);
        end
        else if (Modname::0) do
                writep(T3X.SYSERR, Modname); writep(T3X.SYSERR, col);
        end
        writep(T3X.SYSERR, ntoa(Line, 0));
        writep(T3X.SYSERR, col);
        writep(T3X.SYSERR, m);
        if (s) do
                writep(T3X.SYSERR, col);
                writep(T3X.SYSERR, s);
        end
        nl(T3X.SYSERR);
        Errcount := Errcount+1;
        if (Errcount >= 20) do
                Errcount := 0;
                fatal(packed"too many errors", 0);
        end
end
```

fatal(m,s) – string,string ⇒ halt
Report a fatal error. First print an error message using **error()** (see above) and then stop the translator.

```
fatal(m, s) do
        error(m, s);
        writep(T3X.SYSERR, packed"terminating."); nl(T3X.SYSERR);
        ! Create (empty) error file. Used on systems which
        ! do not evaluate return codes.
        t.close(t.open(packed"T3X.ERR", T3X.OWRITE));
        halt 1;
end
```

fillbuf() – ⇒ 0
Refill the input buffer. The refill algorithm is described in detail later in this chapter.

```
fillbuf() do var i;
        if (Nomore) return 0;    ! EOF reached?
        ! Move remaining input to start of buffer
        for (i=Cp, Ep) Buffer::(i-Cp) := Buffer::i;
        i := Ep-Cp;              ! Number of characterd moved
        Cp := 0;
        ! Read new chunk
        Ep := t.read(T3X.SYSIN, @Buffer::i, BUFSIZE/2-1);
        if (Ep < 1) Nomore := 1;        ! Nothing to read?
        Ep := Ep + i;                   ! Adjust size
        Lowp := Ep-TEXTLEN;             ! and low water mark
end
```

eof() – ⇒ flag
Check whether the end of the input program has been reached.

```
eof() return Nomore /\ Cp >= Ep;
```

getce() – ⇒ character
Read a character from a string or character literal. If the first character read is '\', read another character and attempt to translate it into a special character. Return the special character, if any, and the last read character otherwise.
When returning a quote character, its **META** bit will be set to signal the scanner that it does not terminate a string, but is a part of it.

```
getce() do var c;
        c := Buffer::Cp; Cp := Cp+1;
        if (c \= '\\') return c;        ! Return non-'\' character
        c := Buffer::Cp; Cp := Cp+1;    ! Get character after '\'
        ! and translate it
        if (c = 'a' \/ c = 'A') return '\a';
        if (c = 'b' \/ c = 'B') return '\b';
        if (c = 'e' \/ c = 'E') return '\e';
        if (c = 'f' \/ c = 'F') return '\f';
        if (c = 'n' \/ c = 'N') return '\n';
        if (c = 'q' \/ c = 'Q' \/ c = '"') return '"' | META;
        if (c = 'r' \/ c = 'R') return '\r';
        if (c = 's' \/ c = 'S') return '\s';
        if (c = 't' \/ c = 'T') return '\t';
        if (c = 'v' \/ c = 'V') return '\v';
        return c;       ! No translation, return the character
end
```

findkw(s) – string ⇒ number
Check whether the string pointed to by **s** contains a T3X keyword. If it does, return the associated token and otherwise return zero.

```
findkw(s)
        ! First select the branch associated with the first character
        ! of the symbol. This way, each string is compared only against
        ! keywords starting with the same character.
        ie (s::0 = 'c') do
                if (strequ(s, packed"call")) return KCALL;
                if (strequ(s, packed"class")) return KCLASS;
                if (strequ(s, packed"const")) return KCONST;
                return 0;
        end
        else ie (s::0 = 'd') do
                if (strequ(s, packed"decl")) return KDECL;
                if (strequ(s, packed"do")) return KDO;
                return 0;
        end
        else ie (s::0 = 'e') do
                if (strequ(s, packed"else")) return KELSE;
                if (strequ(s, packed"end")) return KEND;
                return 0;
        end
        else ie (s::0 = 'f') do
                if (strequ(s, packed"for")) return KFOR;
                return 0;
        end
        else ie (s::0 = 'h') do
                if (strequ(s, packed"halt")) return KHALT;
                return 0;
        end
        else ie (s::0 = 'i') do
                if (strequ(s, packed"ie")) return KIE;
                if (strequ(s, packed"if")) return KIF;
                if (strequ(s, packed"interface")) return KIFACE;
                return 0;
        end
        else ie (s::0 = 'l') do
                if (strequ(s, packed"leave")) return KLEAVE;
                if (strequ(s, packed"loop")) return KLOOP;
                return 0;
        end
        else ie (s::0 = 'm') do
                if (strequ(s, packed"mod")) return BINOP;
                if (strequ(s, packed"module")) return KMODULE;
                return 0;
        end
        else ie (s::0 = 'o') do
                if (strequ(s, packed"object")) return KOBJECT;
                return 0;
        end
        else ie (s::0 = 'p') do
                if (strequ(s, packed"packed")) return KPACKED;
                if (strequ(s, packed"public")) return KPUBLIC;
                return 0;
        end
        else ie (s::0 = 'r') do
                if (strequ(s, packed"return")) return KRETURN;
                return 0;
        end
        else ie (s::0 = 's') do
                if (strequ(s, packed"send")) return KSEND;
                if (strequ(s, packed"struct")) return KSTRUCT;
                return 0;
        end
        else ie (s::0 = 'v') do
                if (strequ(s, packed"var")) return KVAR;
                return 0;
        end
        else ie (s::0 = 'w') do
                if (strequ(s, packed"while")) return KWHILE;
                return 0;
        end
        else do
                return 0;
        end
```

findop(c) – ⇒ number
Identify the operator which follows next in the input program. **C** contains the first character of the operator to recognize. The remaining characters will be read on demand from the input file.
Findop() uses a *longest match first* algorithm whose basic concept has been explained in chapter {2}.
This procedure returns either the token associated with the recognized operator or zero, if no operator could be matched.

```
findop(c) do var nmatch, i, pos;
        nmatch := 0;    ! Number of matched chars
        i := 0;         ! Table index
        pos := %1;      ! Index of matched operator, %1=none
        while (nmatch < Ops[i][OLEN]) do
                if (c = Ops[i][ONAME]::nmatch) do       ! Match?
                        c := Buffer::Cp; Cp := Cp+1;
                        pos := i;
                        nmatch := nmatch+1;
                        if (Ops[i][OLEN] > nmatch) do
                                ! More characters to match for this
                                ! operator
                                pos := %1;
                                i := i-1;
                        end
                end
                i := i+1;
        end
        Cp := Cp-1;     ! Reject non-operator character
        Op := pos;
        return pos = %1-> 0: Ops[pos][OTOK];
end
```

findop2(s) – string ⇒ number
Search an operator with the symbolic name specified in **s** and return its index. If the operator cannot be found, force a fatal error (this should never happen).

```
findop2(s) do var i;
        i := 0;
        while (Ops[i][OLEN]) do
                if (strequ(Ops[i][ONAME], s)) return i;
                i := i+1;
        end
        fatal(packed"bad name in findop2()", s);
end
```

scan() – ⇒ number
This function implements the procedural interface of the lexical analysis phase of the compiler, the *scanner*. Its purpose is to skip over white space characters and comments and to extract one token from the input program.
Scan returns the extracted token. Additionally, it fills the global variable `Text` with the literal text of the token, if it is of the type `SYMBOL` or `STRING`. If it has the type `NUMBER` (which is valid for numeric literals and character constants), its fills the variable `Value` with the value (or ASCII value, respectively) of the token. When the recognized token is an operator, it sets the global variable `Op` to the index of the operator table entry describing the operator.
If the current input character is not contained in the T3X input alphabet, the scanner forces a fatal error and prints the hex code of the offending character.

The scanning technique which is applied here already has been explained in chapter {2}.

```
scan() do
        var     c, i, bc::3;
        var     radix, lastv, dv, ovfl;

        if (Cp >= Lowp) fillbuf();      ! Low-water mark reached?
        c := Buffer::Cp; Cp := Cp+1;    ! Read next character
        ! Skip over white space and comments
        while (1) do
                while ( c = '\s' \/ c = '\t' \/ c = '\n' \/
                        c = '\r' \/ c = '\f'
                ) do
                        if (c = '\n') Line := Line+1;
                        c := Buffer::Cp; Cp := Cp+1;
                        ! Check low-water mark again.
                        ! It may be reached when scanning long comments.
                        if (Cp >= Lowp) do
                                if (eof()) return ENDFILE;
                                fillbuf();
                        end
                end
                if (c \= '!') leave;    ! Leave, if no comment is following
                ! Else skip over the comment
                while (c \= '\n') do
                        if (Cp >= Lowp) do
                                if (eof()) return ENDFILE;
                                fillbuf();
                        end
                        c := Buffer::Cp; Cp := Cp+1;
                end
        end
        if (eof()) return ENDFILE;
        if (    'a' <= c /\ c <= 'z' \/
                'A' <= c /\ c <= 'Z' \/
                c = '_'
        ) do                    ! Accept symbol name or keyword
                i := 0;
                while (1) do
                        ie ('A' <= c /\ c <= 'Z')
                                c := c-('A'-'a');
                        else if (\('a' <= c /\ c <= 'z' \/
                                '0' <= c /\ c <= '9' \/
                                c = '_')
                        )
                                leave;
                        if (i >= TEXTLEN-1) fatal(packed"symbol too long", 0);
                        Text::i := c;
                        i := i+1;
                        c := Buffer::Cp; Cp := Cp+1;
                end
                Text::i := 0;
                Cp := Cp-1;             ! Reject non-symbol character
                c := findkw(Text);      ! Check for keyword
                if (c) do
                        ! 'MOD' is in fact an operator, so set 'Op' here
                        if (c = BINOP) Op := Sym_mod;
                        return c;
                end
                return SYMBOL;
        end
        if ('0' <= c /\ c <= '9' \/ c = '%') do ! Accept a numeric literal
                Value := 0;
                radix := 10;
                ovfl := 0;
                i := 0;                 ! Sign flag
                if (c = '%') do         ! Accept sign
                        i := 1;
                        c := Buffer::Cp; Cp := Cp+1;
                end
                if (c = '0') do         ! Accept '0x' prefix
                        c := Buffer::Cp; Cp := Cp+1;
                        if (c = 'x' \/ c = 'X') do
                                c := Buffer::Cp; Cp := Cp+1;
                                radix := 16;
                        end
                end
                ! Collect digits
                while (1) do
                        lastv := Value;
                        ie (radix = 16 /\ 'A' <= c /\ c <= 'F')
                                dv := c-('A'-10);
                        else ie (radix = 16 /\ 'a' <= c /\ c <= 'f')
                                dv := c-('a'-10);
                        else ie ('0' <= c /\ c <= '9')
                                dv := c-'0';
                        else
                                leave;
                        Value := Value * radix + dv;
                        if (    \Big /\ \ovfl /\
                                ((Value - dv) ./ radix \= lastv \/
                                 Value & 0x7fff \= Value /\ radix \= 16)
                        ) do
                                error(packed"number out of range", 0);
                                ovfl := 1;
                        end
                        c := Buffer::Cp; Cp := Cp+1;
                end
                Cp := Cp-1;     ! Reject non-digit
                if (i) Value := -Value;
                return NUMBER;
        end
        if (c = '\'') do        ! Accept character literal
                Value := getce() & ~META;
                c := Buffer::Cp; Cp := Cp+1;
                if (c \= '\'') error(packed"missing '\\''", 0);
                return NUMBER;
        end
        if (c = '"') do         ! Accept string literal
                i := 0;
                c := getce();
                while (c \= '"') do
                        if (i >= TEXTLEN-2) fatal(packed"string too long", 0);
                        Text::i := c & ~META;
                        i := i+1;
                        c := getce();
                        if (eof()) fatal(packed"unexpected EOF", 0);
                end
                Text::i := 0;
                return STRING;
        end
        ! If it was nothing else, it might be an operator
        i := findop(c);
        if (i) return i;
        ! If not, c it is not a valid input character
        ! Convert input char to hex string
        bc::0 := c/16 + (c/16 > 9 -> 'A'-10 : '0');
        bc::1 := c mod 16 + (c mod 16 > 9 -> 'A'-10 : '0');
        bc::2 := 0;
        fatal(packed"bad input character", bc);
end
```

synch(tok) – number ⇒ 0
Resynchronize the translator after catching a syntax error. If **tok** is non-zero consume characters until a token of the type **tok** is found. If the EOF is reached during this process, signal a fatal error.
If **tok** is zero, simply consume a single token.

```
synch(tok) do
        ie (tok) do     ! Find sync token
                while (Token \= tok) do
                        Token := scan();
                        if (Token = ENDFILE)
                                fatal(packed"EOF found in error recovery", 0);
                end
        end
        else do         ! Swallow one token
                Token := scan();
        end
end
```

findivar(tbl,base,lim,name,rpt) – vector,number,number,string,flag ⇒ number
Find an instance variable named **name** in the entries between **base**+SYMREC and **lim** of the symbol table **tbl**. If the symbol cannot be found and **rpt** is set, report an undefined symbol.
If a symbol with the given name exists, return its offset in the symbol table and otherwise, return –1.
Base points to the current class context, **lim** points to the end of the (relevant) symbol table.

```
findivar(tbl, base, lim, name, rpt) do var i, s;
        i := base + SYMREC;
        while (i < lim /\ tbl[i+SFLAGS] & IVAR) do
                s := tbl[i+SNAME];
                if (s::0 = name::0 /\ strequ(name, s)) return i;
                i := i+SYMREC;
        end
        if (rpt) error(packed"undefined symbol", name);
        return %1;
end
```

extend(h,l) – number,number ⇒ number
Adjust negative values in non-16-bit environments. **H** is the high-byte and **l** is the low-byte of the number to adjust. Return the sign-extended number **h***256+**l**.

```
extend(h, l) do var v;
        v := h<<8 | l;
        if (v > 32767)          ! This cannot happen on 16-bit systems
                v := v-K64;
        return v;
end
```

extendcache(cache,len) – vector,number ⇒ 0
Convert a symbol table from portable to native format by sign-extending its entries. Entries are also converted to local byte-ordering.

```
extendcache(cache, len) do var i, y, r;
        ! Reverse order is important. Otherwise, source
        ! fields may be overwritten.
        for (i=len-1, -1, -1) do
                r := @cache::(i*SR2);
                y := @cache[i*SYMREC];
                ! Reverse order, see above
                y[SFLAGS] := extend(r::SFH, r::SFL);
                y[SSIZE] := extend(r::SZH, r::SZL);
                y[SVAL] := extend(r::SVH, r::SVL);
                y[SNAME] := 0;  ! Names are patched later
        end
end
```

rdclsdir(dirname,reqd) – string,flag ⇒ 0

Read a class directory from external storage. **Dirname** contains the fully qualified path of the file containing the class directory. The **required** flag causes an error message to be generated, if the class file could not be found.

```
rdclsdir(dirname, reqd) do
        var     i, j, k, lib;
        var     magic::4;
        var     rf;
        var     n, nbase;
        var     nlt, nlen;
        var     nb::2;

        rf := packed"corrupt CLASSDIR";
        lib := t.open(dirname, T3X.OREAD);
        if (lib = %1) do
                if (\reqd) return 0;
                fatal(packed"class not available", dirname);
        end
        ! Check magic number ("CDIR")
        if (    t.read(lib, magic, 4) \= 4 \/
                t.memcomp(magic, packed"CDIR", 4)
        )
                fatal(rf, packed"#1");
        ! Get number of entries and convert it to local byte-ordering.
        if (t.read(lib, nb, 2) \= 2) fatal(rf, packed"#2");
        n := nb::1 << 8 | nb::0;
        if (Lt + n*SYMREC >= CACHESPACE)
                fatal(packed"class directory cache overflow", 0);
        nlen := n*SYMREC;       ! Length in bytes
        nlt := Lt + nlen;       ! Expected new length
        ! Append table entries
        if (t.read(lib, @Libcache[Lt], nlen*2) \= nlen*2)
                fatal(rf, packed"#3");
        if (\TUNE16) extendcache(@Libcache[Lt], n);
        nbase := Nt;            ! Remember beginning of new name pool
        ! Append symbol names
        Nt := Nt + t.read(lib, @Names::Nt, NSPACE-Nt);
        if (Nt >= NSPACE \/ Nt < nbase)
                fatal(packed"name space overflow (rdclsdir)", 0);
        j := Lt;        ; Beginnig of imported table
        k := nbase;     ; Beginning of imported names
        ! Fix SNAME references in imported symbols table entries
        for (i=nbase, Nt) do
                if (Names::i = 0) do    ! End of current name
                        Libcache[j+SNAME] := @Names::k;
                        k := i+1;
                        j := j+SYMREC;
                end
                if (j > nlt) leave;
        end
        ! Consistency check
        if (j \= nlt) fatal(rf, packed"#4");
        Lt := nlt;
        t.close(lib);
end
```

wrclsdir() – ⇒ 0
Export symbols to the public level (write class directory cache).

```
wrclsdir() do
        const   WBUFLEN = 256;          ! Output buffer size
        var     k, i, j, f, lib;
        var     buf::WBUFLEN, wp;       ! Output buffer, pointer
        var     wf, pub;
        var     nb::2, p, q;

        wf := packed"error writing CLASSDIR";
        k := 0;
        pub := 0;       ! 'In public class' flag
        ! Count items to export and fix flags
        for (i=0, Lt, SYMREC) do
                f := Libcache[i+SFLAGS];
                ! Count only symbols in public classes
                if (f & CLSS) pub := f & PUBLC;
                ! Clear LINKD and set EXTRN,
                ! except in SYS procedures and module entries (f=0)
                if (\(f & SYS))
                        Libcache[i+SFLAGS] := f & ~LINKD | (f->EXTRN: 0);
                if (pub) k := k+1;
        end
        if (k = 0) return 0;    ! Nothing to export
        lib := t.open(packed"CLASSDIR", T3X.OWRITE);
        if (lib = %1) fatal(packed"could not update CLASSDIR", 0);
        nb::0 := k & 255;
        nb::1 := k >> 8;
        ! Write magic ID and number of members
        if (    t.write(lib, packed"CDIR", 4) \= 4 \/
                t.write(lib, nb, 2) \= 2
        )
                fatal(wf, packed"#1");
        pub := 0;
        wp := 0;
        ! Write the symbol table entries
        for (i=0, Lt, SYMREC) do
                ! Export only public classes
                if (Libcache[i+SFLAGS] & CLSS)
                        pub := Libcache[i+SFLAGS] & PUBLC;
                if (pub) do
                        ! Flush buffer if required
                        if (wp + SR2 >= WBUFLEN) do
                                if (t.write(lib, buf, wp) \= wp)
                                        fatal(wf, packed"#2");
                                wp := 0;
                        end
                        ! Convert entry to external byte-ordering and
                        ! append it to the output buffer
                        ie (TUNE16) do
                                t.memcopy(@buf::wp, @Libcache[i], SR2);
                        end
                        else do
                                p := @buf::wp;
                                q := @Libcache[i];
                                p::SNL := 0;
                                p::SNH := 0;
                                p::SVL := q[SVAL] & 255;
                                p::SVH := q[SVAL] >> 8;
                                p::SZL := q[SSIZE] & 255;
                                p::SZH := q[SSIZE] >> 8;
                                p::SFL := q[SFLAGS] & 255;
                                p::SFH := q[SFLAGS] >> 8;
                        end
                        wp := wp+SR2;
                end
        end
        pub := 0;
        ! Write the symbol names
        for (i=0, Lt, SYMREC) do
                if (Libcache[i+SFLAGS] & CLSS)
                        pub := Libcache[i+SFLAGS] & PUBLC;
                if (\pub) loop;
                j := length(Libcache[i+SNAME]);
                if (j + wp >= WBUFLEN) do
                        if (t.write(lib, buf, wp) \= wp)
                                fatal(wf, packed"#3");
                        wp := 0;
                end
                t.memcopy(@buf::wp, Libcache[i+SNAME], j+1);
                wp := wp+j+1;
        end
        ! Flush output buffer
        if (wp /\ t.write(lib, buf, wp) \= wp)
                fatal(wf, packed"#4");
        t.close(lib);
end
```

findintclass(name) – ⇒ number

Find internal class (class in directory cache). **Name** is the name of the class to find. Return the offset of the class entry in the directory cache or %1, if the class could not be found.

```
findintclass(name) do var i, j, s;
        for (i=0, Lt, SYMREC) do
                s := Libcache[i+SNAME];
                if (    Libcache[i+SFLAGS] & CLSS /\
                        s::0 = name::0 /\ strequ(name, s)
                )
                        return i;
        end
        return %1;
end
```

mkclasspath(dest,base,dir) – string,string,string ⇒ 0

Make class path: create a path name consisting of the directory **base** and the file name **dir** with the suffix `".dir"`. Write the path to **dest**. Basically this function emulates

```
str.format(dest, "%S/%S.dir", [(base), (dir)]);
```

```
mkclasspath(dest, base, dir) do var i, j, sfx;
        sfx := packed".dir";
        j := 0;
        i := 0; while (base::i) do
                dest::j := base::i;
                i := i+1; j := j+1;
        end
        dest::j := '/'; j := j+1;
        i := 0; while (dir::i) do
                dest::j := dir::i;
                i := i+1; j := j+1;
        end
        i := 0; while (sfx::i) do
                dest::j := sfx::i;
                i := i+1; j := j+1;
        end
        dest::j := 0;
end
```

findclass(name) – ⇒ number
Find a class and return a pointer to its class dierctory entry. If the class cannot be found, force a fatal error. **Name** holds the name of the class to be found.

```
findclass(name) do
        var     cl, i;
        var     cldir::TEXTLEN+1;

        ! First try to find it in the cache
        cl := findintclass(name);
        if (cl \= %1) return cl;
        ! Else search the class paths
        i := 0;
        while (cl = %1) do
                if (\Classpaths[i]) leave;
                if (Classpaths[i]::0) do
                        ! Read corresponding module
                        mkclasspath(cldir, Classpaths[i], name);
                        rdclsdir(cldir, 0);
                        ! Check cache again
                        cl := findintclass(name);
                end
                i := i+1;
        end
        if (cl = %1) fatal(packed"class not available", name);
        return cl;
end
```

findredclass(ctx,name) – number,string ⇒ number
Find a class in a redirection list (in a dependent class). **Ctx** is the context to search and **name** is class to find. The function returns the index of the class to find or −1 if the class could not be found.

```
findredclass(ctx, name) do var i, cl;
        ! The redirection list is a list of REDIR entries
        ! preceeding the context itself.
        i := ctx-SYMREC;
        while (i >= 0) do
```

```
                if (\(Symbols[i+SFLAGS] & REDIR)) leave;
                cl := Symbols[i+SVAL];
                if (strequ(Libcache[cl+SNAME], name)) return cl;
                i := i - SYMREC;
        end
        return %1;
end
```

findclassref(name,rpt) – string,flag ⇒ number
Find a reference to a class in any active context (class or module redirection list). **Name** is the name of the class to find. The **rpt** causes an error message to be generated, if it is set. The function returns the index of the class to find or −1 if the class could not be found.

```
findclassref(name, rpt) do var k;
        k := Classctx = %1-> %1: findredclass(Classctx, name);
        if (k = %1 /\ Modctx \= %1) k := findredclass(Modctx, name);
        if (rpt /\ k = %1) error(packed"class not available", name);
        return k;
end
```

findsym(name,rpt) – string,flag ⇒ number
Find a symbol with the name specified in **name** If a symbol with the given name is contained in the symbol table, return its offset and otherwise return −1.
If the requested symbol does not exist *and* the parameter **rpt** is non-zero, report the undefined symbol.
The symbol table is searched backward so that local symbols will be found early.

```
findsym(name, rpt) do var i, s;
        i := St-SYMREC;
        while (i >= 0) do
                s := Symbols[i+SNAME];
                if (s::0 = name::0 /\ strequ(name, s))
                        return i;
                i := i-SYMREC;
        end
        if (rpt) error(packed"undefined symbol", name);
        return %1;
end
```

rptdecls() – ⇒ 0
Report all procedure symbols which have been declared but not defined. (*External* symbols will not be reported.)

```
rptdecls(from, to) do var i, s;
        i := from;
        while (i < to) do
                s := @Symbols[i];
                if (s[SFLAGS] & PROTO)
                        error(packed"undefined procedure", s[SNAME]);
                i := i+SYMREC;
        end
end
```

newsym(name,size,val,flags) – string,number,number,number ⇒ number
Create a new symbol with the given parameters. Each argument name an associated field in the `SYMREC` structure defined at the beginning of the translator source code:

```
name  = SNAME,
size  = SSIZE,
val   = SVAL,
flags = AFLAGS.
```

Newsym first checks whether the given symbol name already exists in the table. If it does exist, a redefinition error is reported. However, a symbol with `flags=PROC|PROTO` may be redefined by a symbol with `flags=PROC`. If no redefinition occurs, a new symbol is created and the fields of the symbol table entry are filled with the arguments of **newsym**. The symbol name **name** is copied to the string pool and a pointer to that copy is placed in the `SNAME` entry.
The symbol `"*"` and `"+"` are considered special and may be redefined any number of times. `"*"` is used to separate (nested) local scopes in the symbol table. `"+"` is used to name redirection list (REDIR) entries.
The management of the symbol table is explained in detail later in this chapter.

```
newsym(name, size, val, flags) do var k, sp;
        k := length(name);
        ! Class names never expire (see 'scoping conflicts' in
        ! the language manual)
        if (findintclass(name) \= %1)
                error(packed"duplicate symbol (class conflict)", name);
        sp := findsym(name, 0);
        ! These symbols may be repeated:
        ! * = separator entry
        ! + = redirection list entry
        if (name::0 \= '*' /\ name::0 \= '+' /\ sp \= %1) do
                ! A procedure may redefine a prototype
                if (    flags & (PROC|PROTO) \= PROC \/
                        Symbols[sp+SFLAGS] & (PROC|PROTO) \= (PROC|PROTO)
                ) do
                        error(packed"duplicate symbol", name);
                        return sp;
                end
        end
        if (St >= SYMBSPACE) fatal(packed"too many symbols", name);
        if (Nt + k >= NSPACE) fatal(packed"out of name space", name);
        strcpy(@Names::Nt, name);
        Symbols[St+SNAME] := @Names::Nt;
        Nt := Nt+k+1;
        Symbols[St+SSIZE] := size;
        Symbols[St+SVAL] := val;
        ! If there is a class context, set IVAR
        Symbols[St+SFLAGS] := flags |
                ((Classctx \= %1)-> IVAR: 0);
        St := St+SYMREC;
        return St-SYMREC;
end
```

newlab() – ⇒ number
Create a unique label identifier.

```
newlab() do
        if (\Label) fatal(packed"out of labels", 0);
        Label := Label+1;
        return Label-1;
end
```

flush() – ⇒ 0
Flush the output buffer.

```
flush() do
        t.write(T3X.SYSOUT, Obuf, Obp);
        Obp := 0;
end
```

commit() – ⇒ 0
Flush the delay queue. The use of the delay queue will be explained later in this chapter.

```
commit() do var i;
        for (i=0, LL) do
                if (Obp >= OBUFL) flush();
                Obuf::Obp := Last[i];
                Obp := Obp + 1;
        end
end
```

gen0(instr) – number ⇒ 0
Generate a *type-0* instruction (an instruction without any arguments). See also **commit()**.

```
gen0(instr) do
        commit();
        Last[0] := instr;
        LL := 1;
end
```

gen1(instr,arg) – number,number ⇒ 0
Generate a *type-1* instruction (an instruction with a single argument). See also **commit()**.

```
gen1(instr, arg) do
        commit();
        ie (Big) do
                Last[0] := instr;
                Last[1] := arg;
                Last[2] := arg>>8;
                Last[3] := arg>>16;
                Last[4] := arg>>24;
                LL := 5;
        end
        else do
                Last[0] := instr;
                Last[1] := arg;
                Last[2] := arg>>8;
                LL := 3;
        end
end
```

gen2(instr,a1,a2) – number,number,number ⇒ 0
Generate a *type-2* instruction (an instruction with two arguments). See also **commit()**.

```
gen2(instr, a1, a2) do
        commit();
        ie (Big) do
                Last[0] := instr;
                Last[1] := a1;
                Last[2] := a1>>8;
                Last[3] := a1>>16;
                Last[4] := a1>>24;
                Last[5] := a2;
                Last[6] := a2>>8;
                Last[7] := a2>>16;
                Last[8] := a2>>24;
                LL := 9;
        end
        else do
                Last[0] := instr;
                Last[1] := a1;
                Last[2] := a1>>8;
                Last[3] := a2;
                Last[4] := a2>>8;
                LL := 5;
        end
end
```

stack(n) – number ⇒ 0
Generate a STACK instruction to allocate (**n**>0) or deallocate (**n**<0) **n** machine words on the runtime stack.

```
stack(n) if (n) gen1(tcode.ISTACK, n);
```

mtdadj(n) – number ⇒ number
Adjust addresses of local variables in methods (the saved caller context uses up an additional word).

```
mtdadj(n) return n<0-> n: n+Method_adjust;
```

linkage(type,base,k) – number,number,number ⇒ 0
Generate linkage information for public/external symbols. **type** = public (`TCODE.IPUB`) external (`TCODE.IEXT`)
base = index of class. **k** = index of symbol.

```
linkage(type, base, k) do var i, j, lab, s::TEXTLEN*2+1;
        ! Symbols are imported/exported as: classname_symbolname
        ie (type = tcode.IPUB) do
                i := strcpy(s, Symbols[Classctx+SNAME]);
                s::i := '_'; i := i+1;
                strcpy(@s::i, Symbols[k+SNAME]);
                lab := Symbols[k+SVAL];
        end
        else do
                i := strcpy(s, Libcache[base+SNAME]);
                s::i := '_'; i := i+1;
                strcpy(@s::i, Libcache[k+SNAME]);
                ! assign external label
                if (\Xlabel) fatal(packed"too many external labels", 0);
                lab := Xlabel;
                Xlabel := Xlabel+1;
                Libcache[k+SVAL] := lab;
        end
        j := length(s);
        gen2(type, lab, j);
        for (i=0, j) gen0(s::i);
end
```

loadsym(k) – number ⇒ 0
Generate a Tcode instruction(s) to load the value of the object with the symbol table index **k**. If `@Symbols[k]` describes a procedure, emit nothing (the procedure will be referenced later by a CALL instruction). **Loadsym** generates code to load the *values* of atomic variables and the *addresses* of non-atomic objects.

```
loadsym(k) do var f, z;
        f := Symbols[k+SFLAGS];
        if (f & PROC) return 0;
        ! Get size (atomic objects have z=0)
        ! The size of on object is the size of its class
        z := (f & OBJCT)-> Libcache[Symbols[k+SSIZE]+SSIZE]:
                Symbols[k+SSIZE];
        ! SELF is special
        ie (k = Selfobj) do
                gen0(tcode.ISELF);
        end
        else ie (f & GLOBL) do  ! Global or instance variable
                ie (f & IVAR)
                        gen1(z-> tcode.ILDIV: tcode.ILDI, Symbols[k+SVAL]);
                else
                        gen1(z-> tcode.ILDGV: tcode.ILDG, Symbols[k+SVAL]);
        end
        else do                 ! Local variable
                gen1(z-> tcode.ILDLV: tcode.ILDL, mtdadj(Symbols[k+SVAL]));
        end
end
```

savesym(k) – number ⇒ 0
Generate code to save the value at the top of the stack in an (atomic) local or global object. **K** specifies the symbol table index of the destination object.

```
savesym(k) do var f;
        f := Symbols[k+SFLAGS];
        ie (f & GLOBL)  ! Global or instance variable
                gen1(f & IVAR-> tcode.ISAVI: tcode.ISAVG, Symbols[k+SVAL]);
        else            ! Local variable
                gen1(tcode.ISAVL, mtdadj(Symbols[k+SVAL]));
end
```

tcond(n,flags) – number,number ⇒ 0
Generate code to introduce the *true* branch of a conditional statement or expression or the body of a loop. **N** specifies the destination label to branch to, if the condition described in **flags** does *not* hold. The meaning of the **flags** parameter is as follows:

`flags=0` destructive branch on false
`flags=1` non-destructive branch on true
`flags=2` non-destructive branch on false

`flags=0` is used in `if`, `ie`, and `while` statements and `flags={1,2}` is used in conditional expressions. The generation of flow control code will discussed in detail later in this chapter.

A non-destructive branch is a branch which leaves the condition code on the stack.

```
tcond(n, flags) do
        ie (flags) do
                gen1(flags=1-> tcode.INBRT: tcode.INBRF, n);
                ! Swallow value, if branch failed
                gen0(tcode.IPOP);
        end
        else do
                gen1(tcode.IBRF, n);
        end
end
```

fcond(y,n) – number,number ⇒ 0
Generate code to introduce the *false* branch of a conditional statement or expression. **Y** denotes the label which tags the end of the conditional (where control is passed after processing either branch of the conditional) and **n** denotes the label which marks the beginning of the *false* branch.

```
fcond(y, n) do
        gen1(tcode.IJUMP, y);   ! Jump to end (from true branch)
        gen1(tcode.ICLAB, n);   ! Set label of false branch
end
```

econd(y) – nunber ⇒ 0
Generate a label which marks the end of a conditional statement or expression. **Y** holds the id of the label.

```
econd(y) gen1(tcode.ICLAB, y);  ! Place end label
```

match(t,m) – number,string ⇒ 0
Match the token **t**. If the token could be matched, advance to the next one and otherwise print the error message **m**.

```
match(tok, m) do
        ie (tok = Token)
                Token := scan();
        else
                error(m, 0);
end
```

xsemi() – ⇒ 0
Match a semicolon.

```
xsemi() match(SEMI, packed"missing ';'");
```

xlparen() – ⇒ 0
Match a left parenthesis.

```
xlparen() match(LPAREN, packed"missing '('");
```

xrparen() – ⇒ 0
Match a right parenthesis.

```
xrparen() match(RPAREN, packed"missing ')'");
```

xrbrack() – ⇒ 0
Match a right square bracket.

```
xrbrack() match(RBRACK, packed"missing ']'");
```

xcomma(swallow) – ⇒ 0
Match a comma. If **swallow** is not set, do not consume the token.

```
xcomma(swallow) ie (Token = COMMA)
                if (swallow) Token := scan();
        else
                error(packed"missing ','", 0);
```

equsign() – ⇒ 0
Match an equation sign.

```
equsign() ie (Token \= BINOP \/ Op \= Sym_equ)
                error(packed"missing '='", 0);
        else
                Token := scan();
```

forcesym(msg) – ⇒ flag
Match a symbol and return 0. If the current input token is not a symbol, emit the error message **msg**, skip the rest of the current statement, and return −1. If **msg** is null, use a default message.

This function is used to avoid subsequent error messages after missing symbol names.

```
forcesym(msg) ie (Token = SYMBOL) do
                return 0;
        end
        else do
                error(msg-> msg: packed"symbol expected", 0);
                synch(SEMI); Token := scan();
                return %1;
        end
```

classconst(comp) – flag ⇒ number
Accept a class or class constant:

```
classconst := SYMBOL | SYMBOL '.' SYMBOL
```

If the **comp** flag is set, compile the value of the constant or the size of the class. Return the size or value.

```
classconst(comp) do var k, v;
        ! Find the class
        k := findclassref(Text, 0);
        if (k = %1) error(packed"undefined symbol", Text);
        Token := scan();
        if (k = %1) return %1;
        ie (Token = DOT) do             ! Dot found, get class constant
                Token := scan();
                if (forcesym(0)) return %1;
                ! Find the symbol following the dot
                k := findivar(Libcache, k, Lt, Text, 0);
                ie (k \= %1) do
                        if (\(Libcache[k+SFLAGS] & CNST))
                                error(packed"not a class constant", Text);
                        v := Libcache[k+SVAL];
                        if (comp) gen1(tcode.INUM, v);
                end
                else do
                        error(packed"no such class constant", Text);
                end
                Token := scan();
        end
        else do                         ! No dot found, bad message
                error(packed"missing '.' after class name", 0);
        end
        return v;
end
```

constfac() – ⇒ number
Accept a single factor of a constant expression:

```
constfac := NUMBER
 | SYMBOL
 | classconst
 | '-' constfac
 | '~' constfac
```

Constant factors may be either symbols denoting constants or numeric literals or class constants. There may be any number of leading '–' or '~' signs. **Constfac** returns the value of the accepted factor.

```
constfac() do var k, v;
```

```
        v := 1;
        ie (Token = BINOP /\ Op = Sym_sub) do
                Token := scan();
                return -constfac();
        end
        else ie (Token = UNOP /\ Op = Sym_not) do
                Token := scan();
                return ~constfac();
        end
        else ie (Token = NUMBER) do
                v := Value;
                Token := scan();
        end
        else ie (Token = SYMBOL) do
                k := findsym(Text, 0);
                ie (k \= %1) do
                        ie (Symbols[k+SFLAGS] & CNST)
                                v := Symbols[k+SVAL];
                        else ie (Symbols[k+SFLAGS] & CLSS) do
                                v := Symbols[k+SSIZE];
                        end
                        else do
                                error(packed"not a constant", Text);
                        end
                        Token := scan();
                end
                else do
                        v := classconst(0);
                end
        end
        else do
                error(packed"constant value expected", 0);
        end
        return v;
end
```

constval() – $\Rightarrow$ number

Accept a constant expression. A constant expression is a constant factor as accepted by **constfac** followed by an optional part consisting of an operator and another constant expression:

```
constval := constfac
 | constfac '+' constval
 | constfac '*' constval
 | constfac '|' constval
```

Evaluation is left-to-right. The following operators are accepted by **constval**: '+', '*', '|'. **Constval** returns the value of the accepted constant expression.

```
constval() do var v, o;
        v := constfac();
        while (Token = BINOP) do
                o := Op;
                Token := scan();
                ie (o = Sym_add)
                        v := v + constfac();
                else ie (o = Sym_mul)
                        v := v * constfac();
                else ie (o = Sym_bar)
                        v := v | constfac();
                else
                        leave;
        end
        return v;
end
```

pcall(k,ind) – number,flag ⇒ 0
Generate a procedure call. **K** holds the symbol table index of the procedure object and **ind** is a flag which indicates whether a direct (**ind**=1) or an indirect call has to be generated. If **k**=−1, an error has occured before and the object to call is not a valid procedure.

```
pcall(k, ind) do var n, d[9], dl, i;
        ! Count arguments
        n := 0;
        ie (k = %1) do
                error(packed"call of non-procedure", 0);
        end
        else do
                ie (ind) do
                        ! Ignore CALL before procedure names
                        ie (Symbols[k+SFLAGS] & PROC /\
                                \(Symbols[k+SFLAGS] & PUBLC)
                        )
                                ind := 0;
                        ! Allow only CALLs through atomic variables
                        else if ((Symbols[k+SFLAGS] &
                                 (CNST|CLSS|OBJCT|PUBLC)) \/
                                Symbols[k+SSIZE]
                        )
                                error(packed"bad indirect call",
                                        Symbols[k+SNAME]);
                end
                else do
                        if (    \(Symbols[k+SFLAGS] & PROC) \/
                                Symbols[k+SFLAGS] & PUBLC
                        )
                                error(packed"call of non-procedure",
                                        Symbols[k+SNAME]);
                end
        end
        ! Now match the argument list
        xlparen();
        if (ind) do
                ! Remove reference to procedure pointer from delay
                ! queue and remember it
                for (i=0, 9) d[i] := Last[i];
                dl := LL; LL := 0;
        end
        ! Compile argument list
        if (Token \= RPAREN) while (1) do
                ! Arguments are placed on the stack
                expr();
                n := n+1;
                if (Token \= COMMA) leave;
                Token := scan();
        end
        if (\ind /\ k >= 0 /\ Symbols[k+SSIZE] \= n)
                error(packed"wrong number of arguments", Symbols[k+SNAME]);
        if (k >= 0) do
                ie (ind) do
                        ! Indirect call
                        ! Re-emit reference to procedure pointer
                        commit();
                        for (i=0, 9) Last[i] := d[i];
                        LL := dl;
                        gen0(tcode.ICALR);
                end
                else ie (Symbols[k+SFLAGS] & SYS) do
                        ! System procedure
                        gen1(tcode.ISYS, Symbols[k+SVAL]);
                end
                else ie (Symbols[k+SFLAGS] & EXTRN) do
                        ! Call of external method
                        gen1(tcode.ICALX, Symbols[k+SVAL]);
                end
                else do
                        ! Ordinary procedure call
                        gen1(tcode.ICALL, Symbols[k+SVAL]);
                end
                ! Clean up the stack
                gen1(tcode.ICLEAN, n);
        end
        xrparen();
end
```

strlit(s,ext) – string,flag ⇒ number
Generate a label and a string literal. If **ext**=1, generate an unpacked literal and if **ext**=0, generate a packed literal. Return the label which tags the literal. **S** is a pointer to the string to emit.

```
strlit(s, ext) do var i, l, k;
        if (Packstr) ext := 0;
        k := length(s);
        l := newlab();
        gen1(tcode.IDLAB, l);
        gen1(ext-> tcode.ISTR: tcode.IPSTR, k);
        for (i=0, k) gen0(s::i);
        Token := scan();
        return l;
end
```

bytevec() – ⇒ number
Accept a packed table or string (aka *byte vector*), and emit the appropriate code. Return the label which tags the generated packed object. **Note**: since packed tables are always 'flat', the same code as for packed strings is generated for them.

```
bytevec() do var tbl[MAXTBL], v, i, p;
        Token := scan();
        if (Token = STRING) return strlit(Text, 0);
        if (Token \= LBRACK) do
                error(packed"packed string or vector expected", 0);
                synch(0);
                return 0;
        end
        ! Count members
        p := 0;
        ! Skip left bracket
        Token := scan();
        while (Token = SYMBOL \/ Token = NUMBER) do
                ! Accept member
                tbl[p] := constval();
                if (tbl[p] > 255 \/ tbl[p] < -128)
                        error(packed"vector member too big", 0);
                if (p >= MAXTBL) fatal(packed"table too big", 0);
                p := p+1;
                if (Token \= COMMA) leave;
                Token := scan();
        end
        if (\p) error(packed"empty table", 0);
        v := newlab();
        gen1(tcode.IDLAB, v);
        gen1(tcode.IPSTR, p);
        for (i=0, p) gen0(tbl[i]);
        xrbrack();
        return v;
end
```

table() – ⇒ number

Accept a table and generate the Tcode instructions which are necessary to build the table at run time. Return the id of the label which tags the new table.

The table will be built in two internal buffers (`tv` and `tf`). They are emitted when the table has been completely accepted. This is necessary, because tables may contain nested vector objects. Embedded tables, for example, will be generated by recursing into **table**. In this case, the embedded table is emitted entirely before the surrounding table will get emitted. This way, a pointer to the embedded table can be used to represent it in the outer table.

Additional labels are generated for embedded subexpressions. The embedded subexpressions themselves will be generated immediately (by calling **expr**). Thereafter, a save instruction is generated to place the result of the dynamic expression in the table.

```
table() do
        var     tv[MAXTBL],     ! Member values
                tf::MAXTBL,     ! Member types
                p, v, i, k;

        ! Count members
        p := 0;
        ! Skip left bracket
        Token := scan();
        while ( Token = SYMBOL \/ Token = NUMBER \/
                Token = LBRACK \/ Token = STRING \/
                Token = ADDROP \/ Token = KPACKED \/
                Token = LPAREN \/
                Token = BINOP /\ Op = Sym_sub \/
                Token = UNOP /\ Op = Sym_not
        ) do
                ! Constant expression
                ie (    Token = SYMBOL \/ Token = NUMBER \/
                        Token = BINOP /\ Op = Sym_sub \/
                        Token = UNOP /\ Op = Sym_not
                ) do
                        tv[p] := constval();
                        tf::p := tcode.IDATA;
                end
                ! String literal
                else ie (Token = STRING) do
                        tv[p] := strlit(Text, 1);
                        tf::p := tcode.IDREF;
                end
                ! Packed string or table
                else ie (Token = KPACKED) do
                        tv[p] := bytevec();
                        tf::p := tcode.IDREF;
                end
                ! Address of global object
                else ie (Token = ADDROP) do
                        Token := scan();
                        ie (Token = SYMBOL) do
                                k := findsym(Text, 1);
                        end
                        else do
                                error(packed"symbol expected", 0);
                                k := %1;
                        end
                        if (k >= 0) do
                                ie (Symbols[k+SFLAGS] & (CNST|SYS)) error(
                                        packed"cannot take @const or @syscall",
                                        Text);
                                else ie (Symbols[k+SFLAGS] & PROC)
                                        tf::p := tcode.ICREF;
                                else ie (Symbols[k+SFLAGS] & GLOBL = GLOBL)
                                        tf::p := tcode.IDREF;
                                else
                                        error(packed"unknown address", Text);
                        end
                        tv[p] := Symbols[k+SVAL];
                        Token := scan();
                end
                ! Dynamic expression
                else ie (Token = LPAREN) do
                        tv[p] := newlab();
                        tf::p := tcode.IDLAB;
                        expr();
                        gen1(tcode.ISAVG, tv[p]);
                end
                ! Nested table
                else do
                        tv[p] := table();
                        tf::p := tcode.IDREF;
                end
                if (p >= MAXTBL) fatal(packed"table too big", 0);
                p := p+1;
                if (Token \= COMMA) leave;
                Token := scan();
        end
        v := newlab();
        gen1(tcode.IDLAB, v);
        if (\p) error(packed"empty table", 0);
        ! Emit the table
        for (i=0, p) do
                ie (tf::i = tcode.IDLAB) do
                        ! Generate destination label and empty slot
                        ! for embedded dyamic expression
                        gen1(tcode.IDLAB, tv[i]);
                        gen1(tcode.IDATA, 0);
                end
                else do
                        gen1(tf::i, tv[i]);
                end
        end
        xrbrack();
        return v;
end
```

searchmethod(base,name) – number,string ⇒ number
Find a method in the class directory cache. **Base** is the context of the class to search and **name** is the name of the requested method. If the message has not been referenced before, create an *EXT* record for it. Return the offset of the method or −1, if no method with the given name exists in the given context.

```
searchmethod(base, name) do var m;
        m := findivar(Libcache, base, Lt, name, 0);
        ! If LINKD is not set, this method has not been referenced before.
        ! In this case, generate linkage information and set LINKD.
        if (m \= %1 /\ Libcache[m+SFLAGS] & (EXTRN|LINKD) = EXTRN) do
                linkage(tcode.IEXT, base, m);
                Libcache[m+SFLAGS] := Libcache[m+SFLAGS] | LINKD;
        end
        return m;
end
```

findmethod(k,ptbl) – number,vector ⇒ number
Find an accessible method in the context **k**. **ptbl** is an array to fill with address of the symbol table, the method was found in.

```
findmethod(k, ptbl) do var m, tbl;
        ! Assume no method found.
        m := %1;
        ! If k is the current class context, search the symbol table
        ie (Symbols[k+SSIZE] = Classctx) do
                tbl := Symbols;
                m := findivar(Symbols, Classctx, St, Text, 0);
        end
        ! Else k is an object:
        ! search the context of the class of k in the class directory cache
        else do
                tbl := Libcache;
                m := searchmethod(Symbols[k+SSIZE], Text);
        end
        if (m \= %1 /\ \(tbl[m+SFLAGS] & PROC)) m := %1;
        ptbl[0] := tbl; ! The table in which the method was found
        return m;
end
```

sendmsg(k,cl) – number,number ⇒ 0
Send message to an object. **K** is the index of object and **cl** is the index of its class.

```
sendmsg(k, cl) do
        var     m, n;
        var     d[9], dl, i;
        var     tbl;

        ! Save reference to receiving object
        ! and remove it from the delay queue
        for (i=0, 9) d[i] := Last[i];
        dl := LL; LL := 0;
        ! Skip the dot
        Token := scan();
        ! Is the receiver really an objct?
        if (k = %1 \/ Symbols[k+SFLAGS] & OBJCT \= OBJCT)
                error(packed"invalid object", k=%1-> 0: Symbols[k+SNAME]);
        if (forcesym(packed"method name expected")) return 0;
        ! If an explicit class is given,
        ! search it for the requested message
        ! (used by the SEND operator)
        ie (cl \= %1) do
                m := searchmethod(cl, Text);
                tbl := Libcache;
        end
        ! Else search the message in the class of k
        else do
                m := k=%1-> %1: findmethod(k, @tbl);
        end
        if (m = %1 /\ k \= %1) do
                error(packed"cannot answer message", Text);
        end
        ! Skip method symbol
        Token := scan();
        ! Process the argument list
        if (Token \= LPAREN) do
                synch(SEMI); return 0;
        end
        xlparen();
        ! Count arguments
        n := 0;
        ! Compile argument list
        if (Token \= RPAREN) while (1) do
                ! Arguments are placed on the stack
                expr();
                n := n+1;
                if (Token \= COMMA) leave;
                Token := scan();
        end
        if (m \= %1 /\ tbl[m+SSIZE] \= n)
                error(packed"wrong number of arguments", tbl[m+SNAME]);
        ! Re-emit reference to receiving object
        commit();
        for (i=0, 9) Last[i] := d[i];
        LL := dl;
        ! Emit the call
        if (m >= 0) do
                ! System method
                gen1(tbl[m+SFLAGS] & SYS-> tcode.ISYS:
                        ! Method in external class
                        tbl[m+SFLAGS] & EXTRN-> tcode.ICALX:
                        ! Method in internal class
                        tcode.ICALL, tbl[m+SVAL]);
                ! Clean up the stack
                gen1(tcode.ICLEAN, n+1);
        end
        xrparen();
end
```

address() – number ⇒
Emit code to load the value (or address) of a symbol. First accept a symbol name. If the name denotes a constant, load its value, and otherwise load its address or content using **loadsym**. Then accept any number of (optional) subscripts of the form

```
[ expr ]
```

which follow the symbol name. If a byte subscript of the form

```
:: factor
```

is found, accept no more subscripts of either form.

The grammar processed by this procedure is

```
address := SYMBOL
 | SYMBOL subscripts
subscripts := '[' expr ']'
 | '[' expr ']' subscripts
 | '::' factor
```

If the object which has been accepted by **address** is a *named* object (not a vector member), return its symbol table index, and otherwise return −1.

```
address() do var tok, k;
        if (Token \= SYMBOL) do
                error(packed"symbol expected", 0);
                return %1;
        end
        k := findsym(Text, 0);
        ie (k = %1) do
                ! Not a symbol? Try class constants
                classconst(1);
                return %1;      ! Constants are anonymous
        end
        else do
                ! Skip symbol name
                Token := scan();
                ie (Symbols[k+SFLAGS] & CNST)           ! Constant
                        gen1(tcode.INUM, Symbols[k+SVAL]);
                else ie (Symbols[k+SFLAGS] & CLSS)      ! Class name
                        gen1(tcode.INUM, Symbols[k+SSIZE]);
                else
```

```
                        loadsym(k);     ! Variable or procedure
        end
        ! Process subscripts
        while (Token = LBRACK \/ Token = BYTEOP) do
                if (k >= 0 /\ (Symbols[k+SFLAGS] & (CNST|PROC|OBJCT|CLSS)))
                        error(packed"bad subscript", Symbols[k+SNAME]);
                tok := Token;
                Token := scan();
                ie (tok = BYTEOP) do
                        factor();
                        gen0(tcode.IDREFB);
                        return %1;
                end
                else do
                        expr();
                        gen0(tcode.IDEREF);
                        xrbrack();
                end
                k := %1;        ! Vector elements are anonymous
        end
        return k;
end
```

sendop() – ⇒ 0
Accept a `SEND` operator.

```
sendop() do var p, cl;
        Token := scan();
        xlparen();
        ! Get object pointer
        if (forcesym(packed"missing variable in SEND")) return %1;
        p := findsym(Text, 1);
        if (p < 0) do
                synch(SEMI); return 0;
        end
        if (Symbols[p+SFLAGS] & (CNST|CLSS|OBJCT|PROC)) do
                error(packed"bad pointer in SEND", Symbols[p+SNAME]);
                synch(SEMI); return 0;
        end
        Token := scan();
        xcomma(1);
        ! Get class
        if (forcesym(packed"missing class in SEND")) return %1;
        cl := findclassref(Text, 1);
        if (cl < 0) do
                synch(SEMI); return 0;
        end
        Token := scan();
        xcomma(0);
        ! Accept message
        loadsym(p);
        sendmsg(0, cl);
        xrparen();
end
```

factor() – ⇒ 0
Accept a factor of an expression. This routine recognizes numeric literals (and character constants), (subscripted) symbols, strings, packed tables and strings,

tables, parenthesized expressions, and unary operators. The binary minus (–) operator is interpreted as a unary minus in this context.

```
factor() do var o, k, ind;
        ie (Token = NUMBER) do                  ! Numeric literal
                gen1(tcode.INUM, Value);
                Token := scan();
        end
        else ie (Token = SYMBOL \/ Token = KCALL) do
                ind := 0;
                if (Token = KCALL) do           ! CALL operator
                        Token := scan(); ind := 1;
                end
                k := address();                 ! Symbol
                ie (Token = DOT) do             ! Message
                        if (ind) error(packed"Bad CALL before message", 0);
                        sendmsg(k, %1);
                end
                else ie (Token = LPAREN)        ! Procedure call
                        pcall(k, ind);
                ! () must follow procedure symbol
                else if (k >= 0 /\ Symbols[k+SFLAGS] & PROC \/ ind)
                        error(packed"incomplete procedure call",
                                Symbols[k+SNAME]);
        end
        else ie (Token = KSEND) do              ! SEND operator
                sendop();
        end
        else ie (Token = STRING) do             ! String literal
                gen1(tcode.ILDLAB, strlit(Text, 1));
        end
        else ie (Token = KPACKED) do            ! Packed vector
                gen1(tcode.ILDLAB, bytevec());
        end
        else ie (Token = LBRACK) do             ! Table
                gen1(tcode.ILDLAB, table());
        end
        else ie (Token = ADDROP) do             ! Address operator
                Token := scan();
                k := address();         ! Generate value
                ie (k >= 0) do
                        ! Clear dereferencing code (procedures generate none)
                        if (\(Symbols[k+SFLAGS] & PROC)) LL := 0;
                        ie (Symbols[k+SFLAGS] & (CNST|SYS))
                                error(packed"cannot take @const or @syscall",
                                        Symbols[k+SNAME]);
                        else ie (Symbols[k+SFLAGS] & PROC)
                                gen1(tcode.ILDLAB, Symbols[k+SVAL]);
                        else ie (Symbols[k+SFLAGS] & GLOBL)
                                gen1(Symbols[k+SFLAGS] & IVAR-> tcode.ILDIV:
                                        tcode.ILDGV, Symbols[k+SVAL]);
                        else    ! @local
                                gen1(tcode.ILDLV, mtdadj(Symbols[k+SVAL]));
                end
                else do
                        ! Reduce vector member value to reference
                        ! by removing dereferencing code from the
                        ! delay queue
                        ie (Last[0] = tcode.IDEREF)
                                Last[0] := tcode.INORM;
                        else ie (Last[0] = tcode.IDREFB)
                                Last[0] := tcode.INORMB;
                        else
                                error(packed"bad lvalue", 0);
                end
        end
        else ie (Token = BINOP) do      ! -factor
                if (Op \= Sym_sub) error(packed"bad unary operator", 0);
                Token := scan();
                factor();
                gen0(tcode.INEG);
        end
        else ie (Token = UNOP) do       ! \factor and ~factor
                o := Op;
                Token := scan();
                factor();
                gen0(Ops[o][OCODE]);
        end
        else ie (Token = LPAREN) do     ! ( expr )
                Token := scan();
                expr();
                xrparen();
        end
        else do
                error(packed"bad expression", 0);
                synch(0);
        end
end
```

emitop(stk,sp) – vector,number ⇒ number
Emit a Tcode instruction for a binary operator and remove the emitted operator from the stack represented by **stk** (stack members) and **sp** (stack pointer). The Tcode instruction is taken from the `Ops` table. **emitop()** returns the updated stack pointer **sp**.

```
emitop(stk, sp) do
        sp := sp-1;
        gen0(Ops[stk[sp]][OCODE]);
        return sp;
end
```

binary() – ⇒ 0
This routine accepts all subexpressions which do not contain any flow control operators (‘`\/`’, ’`/\`’, and ’`->  :`’). The algorithm used in this procedure is called *falling precedence parsing*. It depends on the data structure used to build the operator table and is described in detail later in this chapter.

```
binary() do var stk[5], sp;
        sp := 0;
        ! Accept a factor
        factor();
        ! Collect binary operators
        while (Token = BINOP) do
                ! While precedence is not ascending, emit operators
                while (sp /\ Ops[Op][OPREC] <= Ops[stk[sp-1]][OPREC])
                        sp := emitop(stk, sp);
                ! This should never happen:
```

```
                if (sp >= 5) fatal(packed"binary(): stack overflow", 0);
                ! Queue next operator
                stk[sp] := Op; sp := sp+1;
                ! Accept the operator and its righthand side operand
                Token := scan();
                factor();
        end
        ! Emit pending operators
        while (sp) sp := emitop(stk, sp);
end
```

conj() – ⇒ 0
Accept and generate a short circuit AND operation.

```
conj() do var e;
        e := 0;         ! Assume no operation
        binary();
        ! If there is a /\ operator, generate a common end label
        if (Token = CONOP) e := newlab();
        ! Accept a chain of ANDs
        while (Token = CONOP) do
                Token := scan();
                tcond(e, 2);
                binary();
        end
        if (e) econd(e);
end
```

disj() – ⇒ 0
Accept and generate a short circuit OR operation.

```
disj() do var e;
        e := 0;         ! Assume no operation
        conj();
        ! If there is a \/ operator, generate a common end label
        if (Token = DISOP) e := newlab();
        ! Accept a chain of ORs
        while (Token = DISOP) do
                Token := scan();
                tcond(e, 1);
                conj();
        end
        if (e) econd(e);
end
```

expr() – ⇒ 0
Accept an expression. In fact, **expr** itself accepts only the (least precedence) conditional operator '->:'. To accept disjunctions – which form the factors of the conditional operator – it calls **disj**. **Disj** in turn calls **conj** to accept conjunctions and **conj** finally calls **binary** which processes all other precedence levels.

Down to **binary**, the expression parser is an ordinary predicative recursive descent parser. For a discussion of this technique, see the section on *falling precedence parsing* later in this chapter.

```
expr() do var y, n;
        disj();
        if (Token = COND) do
                y := newlab();
                n := newlab();
                Token := scan();
                tcond(n, 0);
                expr();
                fcond(y, n);
                match(COLON, packed"missing ':'");
                expr();
                econd(y);
        end
end
```

ie_stmt(alt) – number ⇒ 0
Translate an `IF` or `IE` statement. If **alt**=0 accept an `IF` statement and otherwise parse an `IE` statement. Flow control statements will be discussed in detail later in this chapter.

```
ie_stmt(alt) do var y, n;
        n := newlab();
        if (alt) y := newlab();
        Token := scan();
        xlparen();
        expr();
        xrparen();
        tcond(n, 0);
        stmt();
        if (alt) do
                fcond(y, n);
                match(KELSE, packed"missing 'ELSE'");
                stmt();
        end
        econd(alt-> y: n);
end
```

while_stmt() – ⇒ 0
Translate a `WHILE` statement. Since loops may be nested to any level, the context of an outer loop is saved in some local variables and restored after translating the loop statement. Flow control statements will be discussed in detail later in this chapter.

```
while_stmt() do var s, e, levl;
        ! Save context of outer loop, if any
        s := Startlab;
        e := Endlab;
        levl := Looplevl;
        ! Create new loop context
        Startlab := newlab();
        Endlab := newlab();
        Looplevl := Locladdr;
        ! Translate the statement
        Token := scan();
        xlparen();
        gen1(tcode.ICLAB, Startlab);
        expr();
        tcond(Endlab, 0);
        xrparen();
        stmt();
        gen1(tcode.IJUMP, Startlab);
        econd(Endlab);
        ! Restore outer loop context
        Startlab := s;
        Endlab := e;
        Looplevl := levl;
end
```

for_stmt() – ⇒ 0
Translate a `FOR` statement. Like **while_stmt** above, this routine saves the context of an outer loop before accepting the statement itself and restores it before returning. Flow control statements will be discussed in detail later in this chapter.

```
for_stmt() do var k, step, r, s, e, levl;
        step := 1;      ! Default step width
        ! Save outer loop context
        s := Startlab;
        e := Endlab;
        levl := Looplevl;
        ! Create new loop context
        Startlab := newlab();
        r := newlab();  ! re-entrance label (\= Startlab in FOR)
        Endlab := newlab();
        Looplevl := Locladdr;
        ! Translate the loop
        Token := scan();
        xlparen();
        ! Initialization part
        ie (Token = SYMBOL) do
                k := findsym(Text, 1);
                if (k \= %1 /\ Symbols[k+SFLAGS] & (CNST|PROC|CLSS|OBJCT))
                        error(packed"bad lvalue in 'FOR'", Text);
                if (k \= %1 /\ Symbols[k+SSIZE])
                        error(packed"index may not be a vector", Text);
        end
        else do
                k := %1;
        end
        match(SYMBOL, packed"symbol name expected");
        equsign();
        expr();
        if (k \= %1) savesym(k);
        xcomma(1);
        ! Limit part
        gen1(tcode.ICLAB, r);
        if (k \= %1) loadsym(k);
        expr();
        ! Accept optional increment part
        if (Token = COMMA) do
                Token := scan();
                step := constval();
        end
        xrparen();
        gen1(step>0-> tcode.IUNEXT: tcode.IDNEXT, Endlab);
        ! Body
        stmt();
        ! Generate increment
        gen1(tcode.ICLAB, Startlab);
        ie (Symbols[k+SFLAGS] & GLOBL)
                ie (Symbols[k+SFLAGS] & IVAR)
                        gen2(tcode.IINCI, Symbols[k+SVAL], step);
                else
                        gen2(tcode.IINCG, Symbols[k+SVAL], step);
        else
                gen2(tcode.IINCL, mtdadj(Symbols[k+SVAL]), step);
        gen1(tcode.IJUMP, r);
        gen1(tcode.ICLAB, Endlab);
        ! Restore outer loop context
        Startlab := s;
        Endlab := e;
        Looplevl := levl;
end
```

asg_or_call() – ⇒ 0
Translate an assignment or a standalone procedure call. For further details on the techniques used in this routine, see the section on *deferred code generation*, later in this chapter.

```
asg_or_call() do var k;
        if (forcesym(packed"bad statement")) return 0;
        k := address();
        ie (Token = ASSIGN) do  ! Assignment
                Token := scan();
                ! Assignment to atomic data object
                ie (k \= %1) do
                        ie (Symbols[k+SFLAGS] & (CNST|PROC|OBJCT|CLSS))
                                error(packed"bad lvalue", Symbols[k+SNAME]);
                        else if (Symbols[k+SSIZE])
                                error(packed"missing subscript",
                                        Symbols[k+SNAME]);
                        LL := 0;
                        expr();
                        savesym(k);
                end
                ! Assignment to vector member
                else do
                        k := Last[0];
                        ie (k = tcode.IDEREF)
                                Last[0] := tcode.INORM;
                        else ie (k = tcode.IDREFB)
                                Last[0] := tcode.INORMB;
                        else
                                error(packed"bad lvalue", 0);
                        expr();
                        gen0(k=tcode.IDEREF-> tcode.ISTORE: tcode.ISTORB);
                end
                xsemi();
        end
        else ie (Token = LPAREN) do     ! Standalone procedure call
                pcall(k, 0);
                gen0(tcode.IPOP);       ! Discard return value
                xsemi();
        end
        else ie (Token = DOT) do        ! Standalone message
                sendmsg(k, %1);
                gen0(tcode.IPOP);       ! Discard return value
                xsemi();
        end
        else do
                error(packed"bad statement", 0);
                synch(SEMI);
                xsemi();
        end
end
```

defr5proc(name,arity) – ⇒ 0
Define an *EXT* record for an R5 compatibility procedure. **Name** is the name of the procedure and *arity* is the number of its arguments.

```
defr5proc(name, arity) do var i, j;
        newsym(name, arity, Xlabel, PROC|EXTRN);
        if (\Xlabel) fatal(packed"too many external labels", 0);
        Xlabel := Xlabel+1;
        j := length(name);
        gen2(tcode.IEXT, Xlabel-1, j);
        for (i=0, j) gen0(name::i);
end
```

initseq() – ⇒ 0
Generate a Tcode header. The header has to be in Tcode4 format, even if it is part of a Tcode5 program.

```
initseq() do var obig;
        ! Save 'big' flag and reset to emit Tcode4
        obig := Big; Big := 0;
        ! Emit header
        gen2(tcode.IINIT, obig->5: 4, 1);
        ! Restore 'big' flag
        Big := obig;
        ! Emit call to main module
        gen1(tcode.ICALL, 1);
        ! Halt program when the main procedure returns
        gen1(tcode.IHALT, 0);
        ! Reserve address 0
        gen1(tcode.IDATA, 0);
        ! Remember that INIT was emitted
        Initdone := 1;
end
```

metacmd() – ⇒ 0
Accept and evaluate a meta command of the form

```
#symbol ... ;
```

```
metacmd() do var badsyn;
        badsyn := 0;    ! Syntax error flag
        ! Skip '#'
        Token := scan();
        ie (Token \= SYMBOL) do
                badsyn := 1;
        end
        else ie (strequ(Text, packed"big")) do
                ! Enable big mode
                ! (only available on machines with >= 32 bits
                if (t.bpw() < 4) fatal(packed"big mode not supported", 0);
                if (Initdone)
                        fatal(packed"#BIG must be the first statement", 0);
                Token := scan();
                Big := 1;
        end
        else ie (strequ(Text, packed"classpath")) do
                ! Set user class path
                Token := scan();
                ie (Token \= STRING) do
                        badsyn := 1;
                end
                else do
                        strcpy(Usrcp, Text);
                        Classpaths[0] := Usrcp;
                        Token := scan();
                end
        end
        else ie (Text::0 = 'l' /\ \Text::2) do
                ! Set input line number and file name
                Token := scan();
                ie (Token \= NUMBER) badsyn := 1;
                else Token := scan();
                Line := Value;
                ie (Token \= STRING) do
                        badsyn := 1;
                end
                else do
                        if (\badsyn) strcpy(File, Text);
                        Token := scan();
                end
        end
        else ie (strequ(Text, packed"debug")) do
                ! Enable output of debug records
                Token := scan();
                Debug := 1;
        end
        else ie (strequ(Text, packed"packstrings")) do
                ! Pack strings implicitly
                Token := scan();
                Packstr := 1;
        end
        else ie (strequ(Text, packed"r5")) do
                ! Declare R5 compatibility procedures
                if (\Initdone) initseq();
                Token := scan();
                defr5proc(packed"open", 2);
                defr5proc(packed"close", 1);
                defr5proc(packed"erase", 1);
                defr5proc(packed"select", 2);
                defr5proc(packed"reads", 2);
                defr5proc(packed"writes", 1);
                defr5proc(packed"newline", 0);
                defr5proc(packed"aton", 1);
                defr5proc(packed"ntoa", 2);
                defr5proc(packed"pack", 2);
                defr5proc(packed"unpack", 2);
                defr5proc(packed"readpacked", 3);
                defr5proc(packed"writepacked", 3);
                defr5proc(packed"reposition", 4);
                defr5proc(packed"rename", 2);
                defr5proc(packed"memcopy", 3);
                defr5proc(packed"memcomp", 3);
        end
        else do
                Token := scan();
                badsyn := 1;
        end
        if (badsyn) error(packed"invalid # syntax", 0);
        xsemi();
end
```

stmt() – ⇒ 0

Translate statements. Basically select the appropriate procedure based upon the current input token. Handle simple statement (like `LEAVE`, `RETURN`, etc) directly. Return a flag indicating whether the accepted statement has an unconditional return value (`RETURN` or a compound statement containing `RETURN` at its end).

```
stmt() do
        ! When debugging, emit a line number record
        if (Debug) gen1(tcode.ILINE, Line);
        ie (Token = KDO) do
                ! This statement has a return value, if the compund
                ! statement has one
                return compound();
        end
        else ie (Token = KIE) do        ! IE statement
                ie_stmt(1);
        end
        else ie (Token = KIF) do        ! IF statement
                ie_stmt(0);
        end
        else ie (Token = KWHILE) do     ! WHILE statement
                while_stmt();
        end
        else ie (Token = KFOR) do       ! FOR statement
                for_stmt();
        end
        else ie (Token = KLEAVE) do     ! LEAVE;
                Token := scan();
                xsemi();
                if (\Endlab) error(packed"'LEAVE' in wrong context", 0);
                ! Release local storage
                stack(-(Locladdr - Looplevl));
                gen1(tcode.IJUMP, Endlab);
        end
        else ie (Token = KLOOP) do      ! LOOP;
                Token := scan();
                xsemi();
                if (\Startlab) error(packed"'LOOP' in wrong context", 0);
                ! Release local storage
                stack(-(Locladdr - Looplevl));
                gen1(tcode.IJUMP, Startlab);
        end
        else ie (Token = KRETURN) do    ! RETURN statement
                Token := scan();
                ! If there is a return value, compile it,
                ! otherwise assume 0.
                ie (Token = SEMI)
                        gen1(tcode.INUM, 0);
                else
                        expr();
                xsemi();
                if (\Exitlab) error(packed"'RETURN' in wrong context", 0);
                gen0(tcode.IPOP);
                ! Release local storage
                stack(-Locladdr+1);
                gen1(tcode.IJUMP, Exitlab);
                return 1;       ! This statement has a return value
        end
        else ie (Token = KHALT) do      ! HALT statement
                Token := scan();
                ! If there is a return value, compile it,
                ! otherwise assume 0.
                ie (Token = SEMI)
                        gen1(tcode.IHALT, 0);
                else
                        gen1(tcode.IHALT, constval());
                xsemi();
        end
        else ie (Token = SEMI) do       ! Empty statement
                Token := scan();
        end
        else ie (Token = KELSE) do      ! ELSE in wrong context
                Token := scan();
                error(packed"'ELSE' with no matching 'IE'", 0);
        end
        else ie (Token = KCALL) do      ! CALL statement
                factor();
                gen0(tcode.IPOP);       ! Discard return value
                xsemi();
        end
        else ie (Token = KSEND) do      ! SEND statement
                sendop();
                gen0(tcode.IPOP);       ! Discard return value
                xsemi();
        end
        else ie (Token = METAOP) do     ! Meta command
                metacmd();
        end
        else do                         ! Assignment or procedure call
                asg_or_call();
        end
        return 0;       ! Most statements do not have a return value
end
```

cleanup() – ⇒ 0
Remove all symbols created in the most recently created scope from the symbol table and release the storage allocated for them.

The entry of the special symbol `*` – which is used to separate scopes – holds the bottom of the stack frame of the outer scope in its `SVAL` field and the address of space used for local symbol names in its `SSIZE` field.

```
cleanup() do var k, la;
        la := Locladdr;
        ! Find innermost local context
        k := findsym(packed"*", 0);
        ! If there is an outer context
        if (k \= %1) do
                ! Restore it
                St := k;
                Nt := Symbols[k+SSIZE];
                Locladdr := Symbols[k+SVAL];
        end
        ! Release local storage
        stack(-(la - Locladdr));
end
```

compound() – ⇒ flag
Translate a compound statement. If the last statement inside of the compound statement is a `RETURN` statement, return *true* and otherwise *false*.

```
compound() do var ot, rv;
        ! Remember top of local storage
        ot := Locladdr;
        ! Create new block context
        newsym(packed"*", Nt, Locladdr, 0);
        Token := scan();
        ! Accept local declarations
        while ( Token = KVAR \/ Token = KCONST \/ Token = KSTRUCT \/
                Token = METAOP \/ Token = KOBJECT
        )
                declaration(1);
        ! Create local storage
        stack(Locladdr - ot);
        ! Assume no return value
        rv := 0;
        ! Accept statements
        while (Token \= KEND) do
                if (eof()) fatal(packed"unexpected EOF", 0);
                rv := stmt();
        end
        ! Skip END
        Token := scan();
        cleanup();
        ! If the last statement in this block had a return value,
        ! this block also has one
        return rv;
end
```

adjust(k,n) – number,number ⇒ 0
Adjust the addresses of local symbols generated in procedure argument lists. Arguments have negative addresses in T3X, but they are generated with positive addresses {2,...}. This is done because arguments are passed in reverse order. Therefore, the first argument has the smallest address –*(N+1)* where *N* is the total

number of arguments. While generating argument symbols, the total number is unknown and so all argument addresses get adjusted after accepting the entire parameter list.

The argument **k** specifies the beginning of the argument scope, and **n** holds the total number of arguments.

The formula to adjust a local address *addr* is $addr_{new} = addr_{old} - (n + 3)$ where *n* is the number of arguments. For efficiency reasons, the *n+3* operation has been extracted and placed before the loop.

```
adjust(k, n) do
        n := n+3;       ! Compensate for return/context information
        while (k < St) do
                Symbols[k+SVAL] := Symbols[k+SVAL]-n;
                k := k+SYMREC;
        end
end
```

dumpsym(k,local) – number,flag ⇒ 0
Emit a debug record for a global or local symbol or an instance variable. The record contains the symbol name, type, and address. **K** is the index of the symbol table entry of the symbol to dump, and **local** indicates whether the symbol is local (**local**=0) or global Instance variables are considered global.

```
dumpsym(k, local) do var i, j, s;
        s := Symbols[k+SNAME];
        j := length(s);
        ! Generate an LSYM, ISYM, or GSYM record, respectively
        ie (local)
                gen1(tcode.ILSYM, mtdadj(Symbols[k+SVAL]));
        else ie (Symbols[k+SFLAGS] & IVAR)
                gen1(tcode.IISYM, Symbols[k+SVAL]);
        else
                gen1(tcode.IGSYM, Symbols[k+SVAL]);
        ! Emit address
        gen0(Symbols[k+SVAL]);
        gen0(Symbols[k+SVAL]>>8);
        ! Emit name (including length)
        gen0(j);
        gen0(j>>8);
        for (i=0, j) gen0(s::i);
end
```

procdef(pub) – flag ⇒ 0
This routine accepts a procedure *definition*. If the **pub** flag is set, the procedure to define is a *method*. In this case, the local addresses will be adjusted to compensate for the additional context argument and **linkage** will be called to generate a public symbol.

```
procdef(pub) do var p, otbl, i, k, pb, lv, n, addr, rv;
        n := 0;                 ! Number of formal argumentss
        addr := pub->1:2;       ! First local address to use
        Method_adjust := pub->1:0;
        p := findsym(Text, 0);  ! Look up potential prototype
        Exitlab := newlab();
```

```
k := newsym(Text, 0, 0, PROC | LINKD | (pub->PUBLC:0));
! Create procedure context, lv = index of local variables
lv := newsym(packed"*", Nt, 1, 0)+SYMREC;
Token := scan();
! Accept argument list
xlparen();
pb := St;       ! Beginnig of procedure context
! Build argument list
if (Token \= RPAREN) while (1) do
        if (Token = SYMBOL) do
                newsym(Text, 0, addr, 0);
                addr := addr+1;
        end
        match(SYMBOL, packed"argument name expected");
        n := n+1;
        if (Token \= COMMA) leave;
        Token := scan();
end
! Save procedure type (arity)
Symbols[k+SSIZE] := n;
! If there is a prototype, make sure the
! redefinition is valid
ie (p \= %1 /\ (Symbols[p+SFLAGS] & PROTO)) do
        if (Symbols[p+SSIZE] \= n)
                error(packed"argument count mismatch",
                        Symbols[k+SNAME]);
        ! Copy prototype label
        Symbols[k+SVAL] := Symbols[p+SVAL];
        ! Make prototype entry invalid by removing the
        ! symbol name and clearing all flags except for IVAR
        Symbols[p+SFLAGS] := Symbols[p+SFLAGS] & IVAR;
        Symbols[p+SNAME] := packed"?";
end
else do
        ! No prototype: assign new label
        Symbols[k+SVAL] := newlab();
end
if (pub /\ Symbols[Classctx+SFLAGS] & PUBLC)
        linkage(tcode.IPUB, 0, k);
gen1(tcode.ICLAB, Symbols[k+SVAL]);
gen0(pub-> tcode.IMHDR: tcode.IHDR);
! Generate HINT with number of arguments
gen1(tcode.IHINT, pub->n+1: n);
adjust(lv, n);
! In debug mode, emit record for arguments
if (Debug) do
        for (i=pb, St, SYMREC) dumpsym(i, 1);
end
xrparen();
rv := stmt();
cleanup();
if (\rv) do
        ! Body has no return value? Generate zero rv.
        gen1(tcode.INUM, 0);
        gen0(tcode.IPOP);
end
gen1(tcode.ICLAB, Exitlab);
gen0(pub-> tcode.IENDM: tcode.IEND);
Exitlab := 0;
Method_adjust := 0;
```

```
end
```

procdecl() – number ⇒ 0
Accept any number of procedure *declarations* in a `DECL` statement.

```
procdecl() do var k;
        Token := scan();
        while (1) do
                if (forcesym(0)) return 0;
                k := newsym(Text, 0, 0, PROC|PROTO);
                Token := scan();
                xlparen();
                Symbols[k+SSIZE] := constval();
                Symbols[k+SVAL] := newlab();
                xrparen();
                if (Token \= COMMA) leave;
                Token := scan();
        end
        xsemi();
end
```

allocvar(k,local) – number,flag ⇒ 0
Allocate space for a local, global, or instance variable. **K** is the index of the symbol table entry of the variable to allocate soace for. The **local** flag is used to indicate that space is to be allocated on theruntime stack.

```
allocvar(k, local)
                ! Local variable
                ie (local) do
                        Symbols[k+SVAL] := Locladdr;
                        ! Local address space grows down,
                        ! addresses of local vectors have
                        ! to be adjusted.
                        if (Symbols[k+SSIZE])
                                Symbols[k+SVAL] :=
                                        Symbols[k+SVAL] +
                                        Symbols[k+SSIZE]-1;
                        ! Allocate space
                        Locladdr := Locladdr +
                                (Symbols[k+SSIZE]->
                                Symbols[k+SSIZE]: 1);
                end
                ! If there is an active class context,
                ! this is an instance variable
                else ie (Classctx \= %1) do
                        ! Allocate space in instance context
                        Symbols[k+SVAL] := Classoff;
                        Classoff := Classoff +
                                (Symbols[k+SSIZE]->
                                Symbols[k+SSIZE]: 1);
                end
                ! Otherwise, this is a global variable
                else do
                        ! Allocate a label
                        Symbols[k+SVAL] := newlab();
                        gen1(tcode.IDLAB, Symbols[k+SVAL]);
                        ! and emit a declaration statement
                        gen1(tcode.IDECL, Symbols[k+SSIZE]);
                end
```

vardecl(pub,local) – flag,flag ⇒ 0
Accept either a CONST or a VAR statement. If **local** is *true*, create local objects, and otherwise create global objects. If **pub** is true, generate instance variables.

```
vardecl(pub, local) do var n, tok, k;
        tok := Token;           ! Remember statement type (VAR or CONST)
        Token := scan();
        while (1) do
                ! Create new symbol without value and with atomic size (0)
                if (forcesym(0)) return 0;
                k := newsym(Text, 0, 0,
                        (tok=KCONST-> CNST: local-> 0 : GLOBL) |
                        (pub-> PUBLC: 0));
                Token := scan();
                ie (tok = KCONST) do
                        ! CONST statement, assign value
                        equsign();
                        Symbols[k+SVAL] := constval();
                end
                else do
                        ! VAR statement
                        if (Token = LBRACK \/ Token = BYTEOP) do
                                ! This is a vector definition
                                tok := Token;   ! Remember type (byte/word)
                                Token := scan();
                                n := constval();        ! Get size
                                ie (tok = BYTEOP)
                                        ! Compute size in Tcode words
                                        n := (n+1) / 2;
                                else
                                        xrbrack();
                                if (n < 1) error(packed"bad vector size",
                                        Symbols[k+SNAME]);
                                Symbols[k+SSIZE] := n;
                        end
                        ! Allocate space for the variable
                        allocvar(k, local);
                        ! When debugging, emit symbol records
                        if (Debug) dumpsym(k, local);
                end
                ! If there is a comma, there must be more symbols
                if (Token \= COMMA) leave;
                Token := scan();
        end
        xsemi();
end
```

defstruct() – ⇒ 0
Accept a structure definition (`STRUCT` statement).

```
defstruct(pub) do var base, count;
        Token := scan();
        if (forcesym(packed"struct name expected")) return 0;
        ! Define structure name
        base := newsym(Text, 0, 0, CNST | (pub-> PUBLC: 0));
        Token := scan();
        equsign();
        count := 0;     ! Count members
        while (1) do
                if (forcesym(packed"member name expected")) return 0;
                ! Add member
                newsym(Text, 0, count, CNST | (pub-> PUBLC: 0));
                Token := scan();
                count := count+1;
                if (Token \= COMMA) leave;
                Token := scan();
        end
        ! Assign size of STRUCT to structure name
        Symbols[base+SVAL] := count;
        xsemi();
end
```

objdef(local) – flag ⇒ 0
Accept any number of object definitions (an `OBJECT` statememt). If **local** is set, generate a local object, otherwise generate a global (or instance level) object.

```
objdef(local) do var k, i, cl;
        Token := scan();
        while (1) do
                if (forcesym(0)) return 0;
                ! Add symbol
                k := newsym(Text, 0, 0, OBJCT | (local->0:GLOBL));
                Token := scan();
                ! Accept class name in square brackets
                match(LBRACK, packed"missing '['");
                if (forcesym(0)) return 0;
                ! Find the class. The class must be
                ! in the directory cache and either in
                ! the current class context or in the
                ! module context.
                cl := findclassref(Text, 1);
                Token := scan();
                ! The size of the object is the size of its class
                Symbols[k+SSIZE] := cl=%1-> %1: Libcache[cl+SSIZE];
                ! Allocate space
                allocvar(k, local);
                Symbols[k+SSIZE] := cl=%1-> %1: cl;
                xrbrack();
                ! Class not found? Delete the object.
                if (cl = %1) St := St-SYMREC;
                if (Token \= COMMA) leave;
                Token := scan();
        end
        xsemi();
end
```

ifacedecl() – ⇒ 0
Accept an `INTERFACE` statement. This statement type is obsolete.

```
ifacedecl() do var k, slot;
        Token := scan();
        while (1) do
                if (forcesym(0)) return 0;
                Token := scan();
                xlparen();
                constval();
                xrparen();
                if (Token = BINOP /\ Op = Sym_equ) do
                        Token := scan();
                        slot := constval();
                        if (slot < Bcslot)
                                error(packed"slot already in use", 0);
                        Bcslot := slot;
                end
                if (Bcslot >= 19)
                        error(packed"extension procedure not available", 0);
                Bcslot := Bcslot+1;
                if (Token \= COMMA) leave;
                Token := scan();
        end
        xsemi();
end
```

accept() – ⇒ 0
Accept any number of declarations and meta commands.

```
accept() while (Token = KVAR \/ Token = KCONST \/ Token = SYMBOL \/
                Token = KDECL \/ Token = KSTRUCT \/ Token = KIFACE \/
                Token = KPUBLIC \/ Token = KCLASS \/ Token = KOBJECT \/
                Token = KMODULE \/ Token = METAOP
        ) do
                ! Initialization has to be done before the
                ! first command, but after the first meta
                ! command, since the #BIG meta command may
                ! change the initialization sequence
                if (\Initdone) do
                        if (Token \= METAOP) initseq();
                end
                declaration(0);
        end
```

redirlist() – ⇒ 0
Accept a dependency list (list of required classes) of a class or module and build a redirection list.

```
redirlist() do var b;
        xlparen();
        while (Token = SYMBOL) do
                ! Find a class. If the class is not in the cache already,
                ! it will be loaded from external storage.
                b := findclass(Text);
                if (b \= %1) newsym(packed"+", 0, b, REDIR);
                Token := scan();
                if (Token \= COMMA) leave;
                Token := scan();
        end
        xrparen();
```

```
end
```

classdecl(pub) – flag ⇒ 0
Accept a class declaration. If **pub** is set, the class is a public class.

```
classdecl(pub) do
        var     cl;
        var     p, i, k;
        var     loc_nt, loc_st;
        var     thiscl::TEXTLEN+1;

        if (pub /\ \Modname::0)
                error(packed"public class in anonymous module", 0);
        if (Classctx \= %1) error(packed"nested class", 0);
        Token := scan();
        ! Get class name
        if (forcesym(0)) return 0;
        ! Remember class name
        strcpy(thiscl, Text);
        Token := scan();
        loc_st := St;   ! Remember beginning of class context
        redirlist();    ! Accept dependency list
        loc_nt := Nt;   ! Remember beginning of class name pool
        ! Now create the class
        ! (The class itself follows the redirection list)
        cl := newsym(thiscl, 0, %1, CLSS | (pub->PUBLC:0));
        ! Establish class context
        Classctx := cl;
        ! Reset class level addresses
        Classoff := 0;
        Symbols[Selfobj+SSIZE] := Classctx;     ! Bind SELF
        accept();       ! Accept declarations at class level
        ! De-activate class context
        Classctx := %1;
        ! Set class size
        Symbols[cl+SSIZE] := Classoff = 0-> 1: Classoff;
        match(KEND, packed"missing END");
        ! Create module entry at end of class
        newsym(Modname::0-> Modname: packed"**", 0, 0, PUBLC);
        ! Unresolved forward references cannot resolved
        ! outside of the class scope
        rptdecls(cl, St);
        ! Copy the class entry and all public items contained
        ! in the class to the class directory cache
        for (p=cl, St, SYMREC) do
                if (p = cl \/ Symbols[p+SFLAGS] & PUBLC = PUBLC) do
                        k := length(Symbols[p+SNAME])+1;
                        t.memcopy(@Names::loc_nt, Symbols[p+SNAME], k);
                        for (i=0, SYMREC) Libcache[Lt+i] := Symbols[p+i];
                        Libcache[Lt+SNAME] := @Names::loc_nt;
                        loc_nt := loc_nt+k;
                        Lt := Lt+SYMREC;
                end
        end
        ! Clear PUBLC flag in module entry
        Libcache[Lt-SYMREC+SFLAGS] := 0;
        ! Remove class from symbol table
        Nt := loc_nt;
        St := loc_st;
```

```
end
```

moddecl() – ⇒ 0
Accept a `MODULE` statement.

```
moddecl() do var k, i, j;
        Token := scan();
        if (forcesym(packed"module name expected")) return 0;
        if (Modname::0) error(packed"module redefinition", Text);
        k := length(Text);
        ! The module name is enclosed in '*'s
        Modname::0 := '*';
        strcpy(@Modname::1, Text);
        Modname::(k+1) := '*';
        Modname::(k+2) := 0;
        Token := scan();
        redirlist();    ! Accept dependency list
        Modctx := St;   ! Remember beginnig of module context
        xsemi();
        ! Delete any classes defined in this module from the
        ! class directory cache.
        ! This is necessary, because this module may re-define
        ! classes which have been imported from the public level.
        for (i=0, Lt, SYMREC) do
                ! Remember beginnig of class
                if (Libcache[i+SFLAGS] & CLSS) k := i;
                ! Module entry (end of class) found
                if (    \Libcache[i+SFLAGS] /\
                        strequ(Libcache[i+SNAME], Modname)
                ) do
                        ! If the module names match, remove the class
                        if (Lt-i-SYMREC)
                                ie (TUNE16) do
                                        t.memcopy(@Libcache[k],
                                                @Libcache[i+SYMREC],
                                                (Lt-i-SYMREC)*2);
                                end
                                else do
                                        for (j=0, (Lt-i-SYMREC))
                                                Libcache[k+j] :=
                                                        Libcache[i+SYMREC+j];
                                end
                        Lt := Lt-(i-k+SYMREC);
                        i := k;
                end
        end
end
```

declaration(local) – flag ⇒ 0
Accept a local or global declaration statement or a meta command. If **local** is *true*, the declaration to accept is contained in a local context, and otherwise it is a global or instance-level declaration.

```
declaration(local) do var pub;
        pub:= 0;        ! Assume non-public declaration
        if (Token = KPUBLIC) do                 ! PUBLIC prefix
                pub := 1;
                Token := scan();
                if (    (Token \= SYMBOL /\ Token \= KSTRUCT /\
                         Token \= KCONST \/ Classctx = %1) /\
                        Token \= KCLASS
                )
                        error(packed"'PUBLIC' in wrong context", 0);
        end
        ! Now select the appropriate handler for
        ! the current declaration.
        ie (Token = KVAR \/ Token = KCONST) do  ! CONST and VAR definitions
                vardecl(pub, local);
        end
        else ie (Token = KDECL) do              ! DECL (procedure declaration)
                procdecl();
        end
        else ie (Token = KSTRUCT) do            ! STRUCT declaration
                defstruct(pub);
        end
        else ie (Token = SYMBOL) do             ! procedure definition
                procdef(pub);
        end
        else ie (Token = KOBJECT) do            ! OBJECT definition
                objdef(local);
        end
        else ie (Token = KCLASS) do             ! CLASS declaration
                classdecl(pub);
        end
        else ie (Token = KIFACE) do             ! INTERFACE (obsolete)
                ifacedecl();
        end
        else ie (Token = KMODULE) do            ! MODULE header
                if (Classctx \= %1)
                        error(packed"'MODULE' in wrong context", 0);
                moddecl();
        end
        else ie (Token = METAOP) do             ! Meta commands
                metacmd();
        end
        else do
                error(packed"bad declaration", 0);
                synch(0);
        end
end
```

initfuncs() – $\Rightarrow$ 0
Second stage of initialization.

```
initfuncs() do
        Selfobj := newsym(packed"self", 0, 0, OBJCT|GLOBL);
        Sym_equ := findop2(packed"=");
        Sym_sub := findop2(packed"-");
        Sym_mod := findop2(packed"mod");
        Sym_add := findop2(packed"+");
        Sym_mul := findop2(packed"*");
        Sym_bar := findop2(packed"|");
        Sym_not := findop2(packed"~");
end
```

TXTRN – ⇒ 0
Initialize the internal structures, generate a header, import and export public level declarations, accept a complete program.

```
do
        ! Initialize variables
        initvars();
        initfuncs();
        ! Import symbols at public level
        rdclsdir(packed"CLASSDIR", 0);
        ! Initialize the parser
        Token := scan();
        ! Accept declarations and meta commands
        accept();
        ! If no initialization sequence was emitted yet, do it now
        if (\Initdone) initseq();
        ! Accept the main program
        if (Token \= KDO) fatal(packed"'DO' or declaration expected", 0);
        ! Create initial label
        gen1(tcode.ICLAB, 1);
        gen0(tcode.IHDR);
        gen1(tcode.IHINT, 0);
        compound();
        ! Report trailing characters, if any
        if (Token \= ENDFILE) error(packed"trailing declarations", 0);
        ! Report unresolved forward references
        rptdecls(0, St);
        ! End of main program
        gen0(tcode.IEND);
        ! Flush the queues and buffers
        commit();
        flush();
        ! Export symbols at public level
        if (\Errcount) wrclsdir();
        ! Report error status to calling process
        if (Errcount) do
                t.close(t.open(packed"T3X.ERR", T3X.OWRITE));
                halt 1;
        end
end
```

4.2 Techniques and Foundations

In the following sections, some algorithms of the T3X translator will be explained in detail. The T3X compiler source code in the above section is consistently structured *bottom up*. Low level routines are always implemented *before* high level routines: Utility functions and error reporting are used by all other parts. Therefore they are implemented first. The scanner, code generator, and symbol table management routines are required by the parser, so their implementations preceed the code of the parser. The main program finally requires all of the above. Consequently, it is located at the end of the code.

The parser is in turn subdivided into a *constant expression parser*, an *expression parser*, a *statement parser*, and a *declaration parser*. This scheme also follows the bottom up paradigm: constant expressions and (general) expressions are used in statements and declarations. Expressions are used inside statements and statements are part of procedure declarations. Programs, finally, consist of declarations. Therefore, a complete source program is accepted by the main body of the translator.

One nice side effect of this implementation strategy is that all procedures are defined before their first use (except, of course, if they are mutually recursive), and so this method results in kind of a 'natural' order of definitions which is particularly easy to read.

4.2.1 Input Buffering

The scanner reads the input program character by character. Since reading files one character at a time is inefficient on most systems, it does some *input buffering*. This means that the scanner reads fixed size *blocks* from the input file and then extracts characters from these blocks. This method is widely used, but integrating it into a scanner requires some care. The biggest problem is to decide when to refill the buffer. Most buffering algorithms simply read the next block when the input from the buffer has been exhausted. For this task, the algorithm maintains two pointers, one pointing to the next character to extract, and one pointing to the end of the buffer. When the current-character pointer reaches the end-of-buffer pointer, the buffer is empty and will be refilled.

This scheme is most flexible and when implemented efficiently, it causes an acceptable overhead. In the *standard I/O library* of the **C** language, for example, the buffered I/O calls for transferring single characters have been implemented as macros. Since macros result in *inlined* code, not even a procedure call overhead is caused when reading a single character (or at least, only when the buffer has to be refilled). On the other hand, each time, a character is extracted, the buffer pointers must be compared like in the following fragment

```
c:=Buffer::Cp; Cp:=Cp+1; if (Cp>=Ep) fillbuf();
```

`Cp` is the current-character pointer, `Ep` is the end-of-buffer pointer, and `fillbuf` is a routine to read the next block and reset the pointers. The variables `Cp` and `Ep` are actually used in the T3X scanner, but not exactly in this way. Since the token length is limited in T3X, the comparison can be saved by a simple improvement. Since the code fragment used for extracting a character is called *for each character* of the input program, it should be as small and fast as possible.

The call of the routine `fillbuf` which reads the next block of input is triggered by a third variable, `Lowp`, which points to a position `TEXTLEN` characters left before `Ep`. When then scanner starts reading, it refills the buffer only, if

```
Cp >= Lowp
```

If `Cp` is less than `Lowp`, there are at least `TEXTLEN` characters between the character pointer and the end of the buffer. Consequently, there must be at least one complete token in the rest of the buffer. Therefore, there is no need to check the refill condition another time while scanning the current token, and the extraction of a character can be reduced to

```
c := Buffer::Cp; Cp := Cp+1;
```

Since `fillbuf` is most frequently called *before* the end of the buffer actually has been reached, it must do some additional work: First, it moves the yet to be processed part of the input to the beginning of the buffer. Then, it appends a new block to the text moved to the beginning, and finally, it adjusts the pointers: `Cp` is moved to the beginning of the buffer, and `Ep` is set to the size of the old block plus the size of the new block.

When `fillbuf` fails to read another block of input, it sets `Nomore` flag. When `Cp` reaches `Ep` and `Nomore` is set, the end of the input program is reached. The `eof` function performs this check.

The input buffering scheme described here is illustrated in figure 21. Part A) shows the organization of the buffer in general. In part B), `Cp` has moved into the 'trigger area' between `Lowp` and `Ep`. If the scanner would be called now, `fillbuf` would move the chunk between `Cp` and `Ep` to the bottom of the buffer, adjust `Cp` (C), and read a new block into the free space after the moved part. Finally, it would set `Ep` to the end of the new block and compute a new trigger value for `Lowp` (D).

The buffering scheme described here has another advantage: Characters up to the beginning of the current token can be rejected at any time during the scan, since the buffer can only be refilled at the beginning of a token. Rejecting an input character is done by simply decrementing `Cp`.

Note: The implementation of the buffering scheme described here has one weak point: In the first loop in `scan`, the refill is triggered before reading white space *and* the following token. Therefore, long sequences of white space and comments could cause a buffer overrun probably break the compiler. To avoid this, the refill trigger is also checked while skipping over white space and comments.

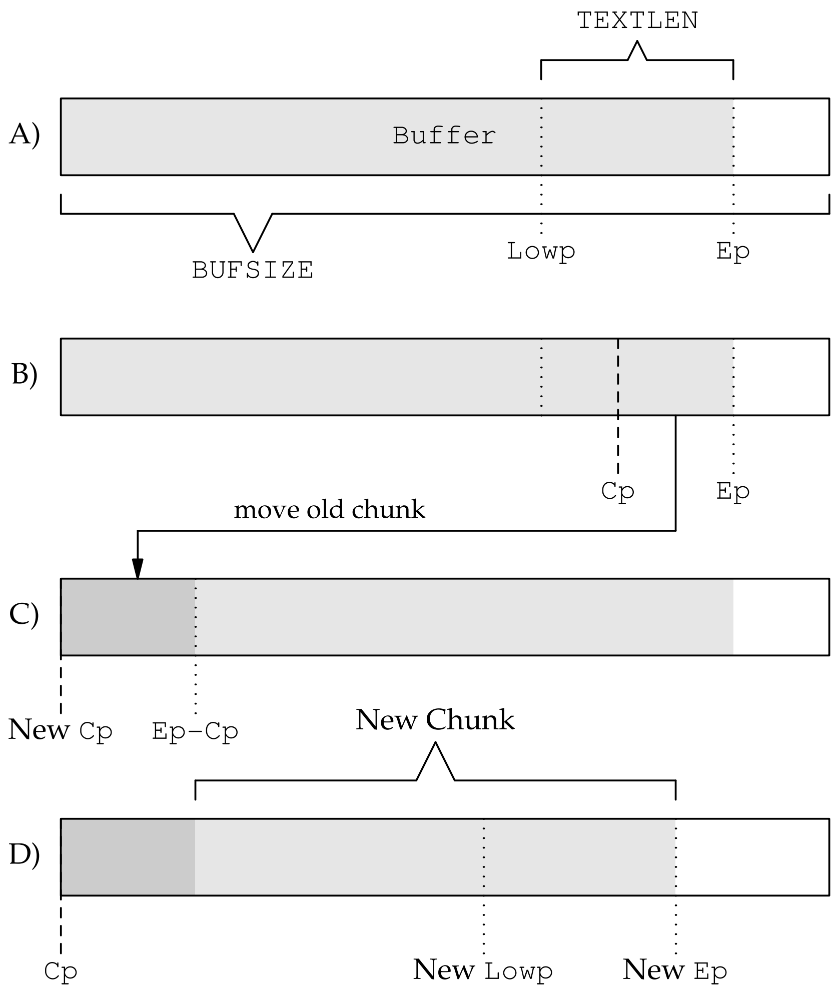


Figure 21: Input Buffering Scheme

4.2.2 Delayed Code Generation

The code generator described here is *very* simple. Basically, it is a buffered output mechanism with a one-instruction *delay queue*. The code generator – also called the *emitter* – is exactly at the end opposite to the scanner. While the scanner *reads* the source program, the emitter *writes* the target program. Like the scanner, it does buffered I/O. The scheme used for output buffering is quite simple: Before queueing an output byte, the emitter checks for a buffer overflow. If an overflow would occur when queuing the byte, the buffer will be flushed, first.

The **delay queue** adds a second level of buffering at instruction level: Before an instruction is written to the output buffer, it will be stored in the delay queue, and when the next instruction arrives, it will be shifted to the output buffer. The purpose of this additional buffer is the option to delete or modify the most recently generated instruction from the buffer before it gets emitted. This operation cannot be performed on the output buffer, since instructions may have different lengths, and therefore, the buffer may get flushed *while* emitting an instruction, thereby making (part of) the instruction inaccessible. The delay queue, though, holds always one complete instruction, no matter what size it has.

The routines `gen0`, `gen1`, and `gen2` are used to queue zero, one, and two-parameter instructions, respectively. Each of these procedures first calls `commit` to move the content of the delay queue out of the way and then queues the new instruction and its arguments, if any. Finally, it loads the variable `LL` with the length of the queued instruction in bytes.

`Commit` moves the instruction currently in the delay queue into the output buffer. If the output buffer is about to overflow, it calls `flush` to write the buffer and reset its pointer.

The deletion or modification of instructions is are required, because the T3X parser is an LL(1) parser. For example, when parsing chains of subscript opertators in lvalues, an indirection command will be generated for each subscript. The code generated for the statement

```
a[i][j][k] := b;
```

up to the euqation sign would look like this:

```
LDG a LDL i DEREF LDL j DEREF LDL k DEREF
```

However, the assignment operator requires the *address* of `a[i][j][k]` and not its value. Since the final `DEREF` instruction generates a value rather than an address, it is replaced with a `NORM` iststruction before generating the remaining instructions of the statement:

```
LDG a LDL i DEREF LDL j DEREF LDL k NORM
```

Using the delay queue, infinite lookahead can be simulated in this case: any number of subscripts may be found between the symbol at the beginning of the statement and the assignment operator, since only the last instruction has to be modified, if an assignment operator is finally encountered in input. The `asg_or_call` routine of the T3X translator performs this step.

4.2.3 Generating Flow Control Statements

The procedures `tcond`, `fcond`, and `econd` are used to build `IF` and `IE` statements, conditional expressions, and loops. `Tcond` compiles the part before the *true* branch or the loop body – basically a jump to the false branch or to the end of a conditional statement or loop. The parameter `n` is the value of the destination label of the conditional jump around the true branch or loop body. The `flags` parameter contains two bits. The bit with the value 1 determines the kind of the branch: 1 means 'branch on true', zero means 'branch on false'. The bit with the value 2 indicates that the branch instruction shall be non-destructive (ie. it may not remove its test value from the stack). Non-destructive branches are used in short circuit expressions (logical AND and OR operations), where the test value must remain on the stack when the a branch is performed. The `BRT` ('branch on true') instruction is never generated in its destructive form by `tcond`.

`Fcond` generates the code between the true and the *false* branch of an `IE` statement or a conditional expression. It consists of a branch to the end of the statement and a label which tags the beginning of the false branch. There are two parameters to `fcond`: `y` is the label tagging the end of the conditional and `n` is the value of the label at the beginning of the false branch.

`Econd` places the 'exit label' at the end of a conditional statement, expression, or loop.

```
Expression:
P /\ Q -> T : F

Generated Tcode:
<P>
NBRF   k |
POP      |  tcond(k,2)    \            \
<Q>                        | P /\ Q    |
CLAB   k |  econd(k)      /            |
BRF    m |  tcond(m,0)                 |
<T>                                    | P/\Q->T:F
JMP    n |                             |
CLAB   m |  fcond(n,m)                 |
<F>                                    |
CLAB   n |  econd(n)                  /
```

Example 25: The Conditional Execution Structure

Example 25 shows the code fragments which will be generated by these procedure to implement a conditional expression with a conjunction in the condition part.

4.2.4 Falling Precedence Parsing

The routines `expr`, `disj`, `conj`, and `binary` together with `factor` reflect the traditional approach to *recursive descent* parsing.

In a predicative RD parser, each routine accepts only operators with a given precedence level. Routines which accept low-precedence operators call procedures which accept the next higher operators to make sure that higher-precedence operations will be completely recognized before accepting operators of a lower level. For example, the routine `expr` which accepts '`->:`' operators first calls `disj` which accepts '`/\`' operators. `Disj` in turn calls `conj` and so on. `Factor` finally accepts the highest precedence operators and returns.

This approach is simple and straight-forward, but it has a clear disadvantage: in the worst case, one procedure call has to be made per precedence level and per factor. Technically speaking, T3X has nine levels of precedence (and this is a rather simple language). So, a purely predicative parser with an own procedure for each level would require at least *levels· factors* = 4·9 = 36 procedure calls to accept this simple function call:

```
P(a,b,c);
```

(Assuming one complete descent per factor). For languages with more precedence levels, this effort would grow accordingly. In **C**, for example, there are 15 different levels of precedence giving 4·15 = 60 procedure calls.

To have an in-depth look at this problem, the sample grammar from example 23 will be used. A straight-forward implementation of a parser which accepts programs described in this grammar can be found in example 26. **Note**: In this example, `scan` returns separate tokens for operators instead of operator classes like in the real T3X parser.

```
! scan() and error() are not shown

const NUMBER=1, SYMBOL=2, PLUS=3,
      MINUS=4, MULTIPLY=5, DIVIDE=6,
      MODULO=7;

var Token;

factor() do
    ie (Token = NUMBER) do
        Token := scan();
    end
    else ie (Token = SYMBOL) do
        Token := scan();
    end
    else ie (Token = MINUS) do
        Token := scan();
        factor();
    end
    else do
        error("bad factor");
    end
end

term() do
    factor();
    while (1) do
        ie (Token = MULTIPLY) do
           Token := scan();
           factor();
        end
        else ie (Token = DIVIDE) do
           Token := scan();
           factor();
        end
        else ie (Token = MODULO) do
           Token := scan();
           factor();
        end
        else do
           leave;
        end
    end
end
```

```
sum() do
    term();
    while (1) do
        ie (Token = PLUS) do
           Token := scan();
           term();
        end
        else ie (Token = MINUS) do
           Token := scan();
           term();
        end
        else do
           leave;
        end
    end
end
```

Example 26: A traditional RD parser

When a sum has to be accepted, `sum` first descends into `term` in *any* case and `term` descends into `factor`. The procedures `term` and `sum` work exactly in the same way: first, they descend into procedure accepting a higher level, then they match an operator with a specific precedence, descend into a higher level again to accept the second operator, and finally recurse into themselves to accept more operations at their own level. The recursion has been replaced with a loop in these routines, since it is a tail-recursion.

The procedures `sum` and `term` are almost identical. Only the tokens accepted in the loop and the the procedures to descend into differ. When code would be generated, the emitted instruction would differ, too, but all the rest would remain the same for *all* routines handling binary operations. This is why a different approach is used in the T3X parser.

```
Expression:
        Factor OptionalOps
        ;

OptionalOps:
        | '*' Expression
        | '/' Expression
        | MOD Expression
        | '+' Expression
        | '-' Expression
        |
        ;

Factor:
        MINUS Factor
        | NUMBER
        | SYMBOL
        ;
```

Example 27: An alternative grammar for simple expressions

The process of parsing binary operations with different precedences belongs to parsing as well as to semantic analysis. In most grammars, precedence is expressed explicitly by specifying a separate set of productions for each level. However, the information expressed by this method belongs to the *semantics* of the language as well, since it describes *properties* of operators. To merely describe the *syntax* of the expressions accepted by the above parser, the grammar in example 27 would suffice very well.

Obviously, this simplified grammar accepts the same set of input programs as the grammar in the earlier example (23), namely all programs consisting of single factors or chains of factors with one single binary operator between each two factors. The only difference is that the second grammar does not express any precedence rules. All operators are treated the same. The advantage of this method is clear: since all binary operators can be treated at the same level (and thereby by the same procedure), a big number of procedure calls can be saved when parsing expressions.

Of course, the precedence of the operators must be preserved, though. To develop a method for handling precedence in a single procedure, it is necessary to examine *at what time* instructions belonging to operators are emitted by a traditional parser. When analyzing the expression

```
A + B * C
```

for example, a traditional parser would run through the call tree displayed in figure 22.

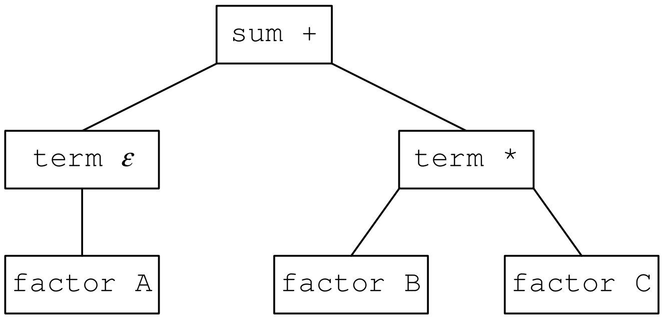


Figure 22: The call tree of an RD parser

Since reverse polish notation – which is used as input for stack machines – is generated by the translator, the Tcode instructions are emitted by the parser procedures *after* processing *both* arguments, leading to the program

```
A B C * +
```

The process of code generation can be simulated by following the left branch of the tree, then following its right branch, and finally emitting an instruction when leaving a node in direction towards its parent.

Using syntax trees, it is easy to show that arithmetic instructions belonging to operators are always generated when the precedence of the last visited instruction is *less* than (or *equal* to) the precedence of the previously visited one when compiling to RPN. In other words this means that an RPN instruction is *generated* when an operator with a lower or the same precedence has been read (or, the end of the expression has been reached). To achieve this effect, however, it is not necessary to implement an extra routine for each precedence level. Instead, a *stack* can be used to defer the code generation for lower-precedence operations.

The procedure `binary` uses the operator table which is created in `initvars` to determine the precedences of the binary operators. The stack and stack pointer are declared locally at the beginning of `binary`, since the procedure may be reentered to process embedded expressions. The size of the stack must be equal to the number of precedence levels to handle.

First, the stack is cleared by setting its pointer to zero and then a single factor is accepted. Thereafter, the routine cycles through a loop as long as subsequent binary operations (indicated by token of the class `BINOP`) are found in the input program. If an operator has been found, another factor must follow. It is matched at the end of the loop. That is all that is required to accept the *syntax* of an expression. The rest of the procedure is dedicated to preserving operator precedence. First, all operators which are currently stored on the stack **and have a higher precedence than the current one**, are emitted. Then, `binary` checks for a stack overflow. (In fact, this is extra safety, since a stack overflow *cannot* happen at this point). Finally, the current operator is placed on the stack in any case. When there are no more operations to accept, the loop is left and all operations still left on the stack are emitted.

Because the core of the algorithm is the fact that instructions will be emitted only, if precedence 'falls' back to (or below) the previous level, I call this method **falling precedence parsing**.

The remainder of this subsection illustrates the process of falling precedence parsing by means of the example

$$a\ |_5\ b\ +_6\ c\ -_6\ d\ =_3\ e\ \hat{}_5\ f\ /_7\ g$$

which contains many operators with different precedences. The levels of the single operators are given as subscripts to their respective symbols.

```
   Input                 Output          Stack
   --------------        ------------    -----
1) a|b+c-d=e^f/g;        a               |
2) a|b+c-d=e^f/g;        ab              | +
3) a|b+c-d=e^f/g;        abc+            | -
4) a|b+c-d=e^f/g;        abc+d-|         =
5) a|b+c-d=e^f/g;        abc+d-|e        = ^
6) a|b+c-d=e^f/g;        abc+d-|ef       = ^ /
7) a|b+c-d=e^f/g;        abc+d-|efg/^=
```

Example 28: Falling Precedence Parsing

The parsing process is displayed in example 28. The left column reads the input expression, the middle one the generated code, and the right one the stack used to delay lower-priority operators. As one can see, the stack depth does not exceed three members during the analysis. Since only the precedence levels 3...7 are handled by `binary`, the stack cannot grow any deeper than five cells and even this case will occur, only if *all* operators in an expression are ordererd by ascending precedence.

In 1), the first factor and the first operator have been scanned. Since there are no operations to emit at this point, the '`|`' operator is simply stacked. In the next cycle, `b` and '+' are read. Since '+' has a higher precedence than '`|`', the binary OR operator is not emitted. '+' is stacked anyway, since no operator is ever emitted immediately, but only if the precedence of the *following* operator is known. This is the case in 3), when `c` and the '–' operator have been read. Since 'plus' and 'minus' have the same precedences, '+' will be emitted before '–' is pushed onto the stack. '`|`' still remains on the stack, since it has a lower precedence than '–'.

In 4), the equation operator '=' has been scanned. It has a very low precedence. Therefore, all operators which have a higher precedence are removed from the stack and emitted before '=' is pushed. In 5), 'ˆ' is pushed onto the stack. No operator is emitted, because 'ˆ' has a higher value than '='. The same happens in 6), where '/' is placed on top of 'ˆ'. In 7), finally, a semicolon is found which is not a valid operator. Therefore, the loop in `binary` is left and all operators still on the stack are flushed. The pending operations on the stack are emitted by the following loop which simply calls `emitop` until the stack is empty.

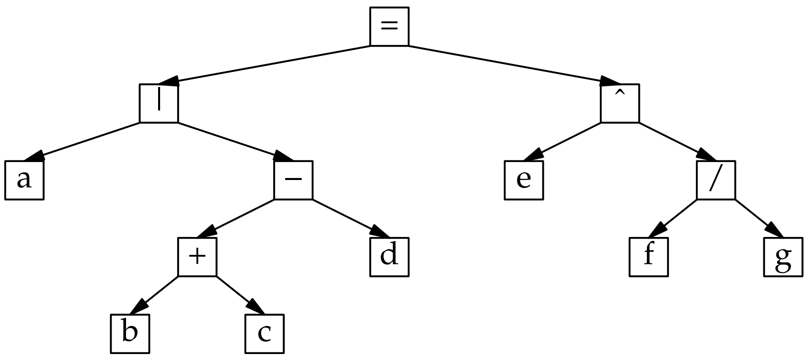


Figure 23: The parse tree of 'a|b+c-d=eˆf/g'

Figure 23 displays the parse tree of the above expression as it could have been generated by a traditional RD parser. Obviously, all precedence rules have been respected while building this tree. When generating RPN from this tree using the previously described left-right-node traversal, the following code would result:

```
a b c + d - | e f g / ˆ =
```

This postfix expression is exactly equal to the expression generated by the falling precedence parser from the same input (example 28).

Since the precedence levels 3...7 are processed in a single loop in the FP parser, the procedure call overhead is reduced significantly compared to the traditioal approach. The simple procedure call

```
P(a,b,c)
```

which has been discussed at the beginning of this subsection would require only 4·4 = 16 procedure calls using the FP parser of the T3X compiler, because five levels can be saved by combining them in the procedure `binary`, while 36 calls would be required by a purely predicative RD parser.

4.2.5 Syntactic Ambiguities

The procedure `asg_or_call` is used to translate both standalone procedure calls and assignments. This is the one and only rule in the entire T3X syntax which cannot be processed by a pure LL(1) parser. A lookahead L=2 is required at this point, since both statements begin with a symbol name, for example:

```
P := fac(7);
```

and

```
fac(7);
```

After reading the name, the statement is unambigous: '`:=`', '`[`', and '`::`' indicate an assignment, '`(`' indicates a procedure call, and anything else would not be accepted at all. Reading an additional lookahead token L_2 is difficult for several reasons: all attributes of the first token L_1 would have to be saved for later processing and there must be a way to *put back* L_2 into the input stream in case it has to be processed by another routine. Imagine the following (simplified) set of rules:

procedure_call : SYMBOL '(' optional_arguments ')' ;
assignment : SYMBOL ':=' expression ;
asg_or_call : procedure_call | assignment ;

Because both paths which can be taken from *asg_or_call* begin with *SYMBOL*, no decision can be made by an LL(1) parser. But since *all* possible paths begin with *SYMBOL*, that token can be simply consumed, thereby *deferring* the decision. However, the rules *procedure_call* and *assignment* expect a *SYMBOL* at the first posititition. Since this token has to be consumed to make the decision possible, it would have to be placed back in the input stream before calling the appropriate procedure. This method would allow a great degree of flexibility, but it would also require substantial changes to the scanner.

Another solution comes into mind when rewriting the above grammar to reflect the deferred decision:

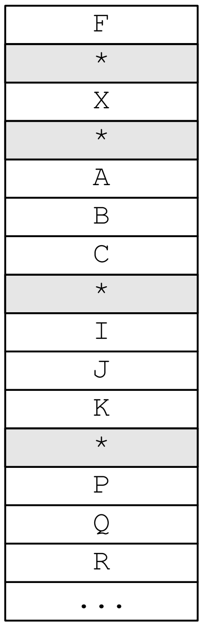
procedure_call : '(' optional_arguments ')' ;
assignment : ':=' expression ;
asg_or_call : SYMBOL procedure_call | SYMBOL assignment ;

The problem with this solution is, of course, that all information about *SYMBOL* is lost when scanning L_2. However, all relevant information about tokens of the class `SYMBOL` are stored in the symbol table. Therefore, it would be sufficient to pass the symbol table index of *SYMBOL* to the procedures handling the *procedure_call* and *assignment* rules. This is exactly the way, `asg_or_call` gets around this LL(1) ambiguity.

4.2.6 Local Contexts

Compound statements are accepted by the procedure `compound`. The routine creates a new block context by adding a special symbol named "`*`" to the symbol table. This symbol is special because it may be duplicated in the symbol table. It is used to delimit nested blocks. The symbol "`*`" cannot interfere with user-defined names, because it is not a valid symbol character.

When thinking of the symbol table as a list where new entries always will be added at the bottom, this model is comparable to a push-down stack. A stack is a data object to efficiently handle recursive structures like nested blocks. The approach is as follows: A new block context is created by 'pushing' "`*`" onto the stack. All symbols which will be pushed thereafter belong to the new context. A context is removed by popping all members up to and including the most recently pushed "`*`".



```
F(x) DO
  DO VAR a,b,c;
    DO VAR i,j,k;
      DO VAR p,q,r;
      END
    END
  END
END
```

Figure 24: Nested block contexts

Figure 24 illustrates the creation of nested scopes in the symbol table. Note that the procedure *F* has its own scope containing the parameter symbol *X*. The procedure itself belongs to the global scope (or a class scope). When the scopes of the procedure (procedure level and block level) are destroyed, the procedure itself remains in the symbol table.

The procedure compound accepts local declarations. No storage is allocated during the declarations. To actually reserve the requested storage, the routine remembers the previous value of Locladdr in the local variable ot and after parsing *all* declarations, it allocates the entire block by passing the difference between the old and the new local address limit to stack. After processing the statements of the block compound calls cleanup to remove the created block context.

The procedure cleanup removes one block context from the symbol table. First, it memorizes the current top of the local storage area and searches the most recently created block context separator ("*"). Then, it resets the symbol table and name pool pointers, thereby releasing the space occupied by the now obsolete symbols. It also resets the local storage limit Locladdr to the value it had before creating the new context. Finally, it generates code to deallocate the local storage by passing the (negated) difference between the old (la) and the new (Locladdr) local storage limit to stack.

4.2.7 Class Level Scopes

At the beginning of a class declaration, the location of the class entry in the symbol table is remember. Symbols subsequently defined inside of the class are added to the global symbol table. Until the class definition terminates, the global level and the class level are the same. When the end of the class declaration is reached, the class declaration including all *public* symbols belonging to the class (class constands and methods) are copied to a location called the *class directory*. After moving the public symbols, the *entire class* is removed from the global symbol table.

The class directory consists of two parts, which are displayed in figure 24. There is a persistent part holding the public class entries and a transient one which holds public *and* non-public entries. The transient part of the directory is called the *class directory cache*. It is initialized with the persistent part when compilation starts (1). During compilation, both public and local classes are added to the cache (2). Both method used to answer messages and class dependencies are looked up in the cache (3). When compilation ends, all public classes contained in the directory cache are written back to the persistent part of the class directory.

The Symbol table is held in a structure called Symbols, the class directory cache is contained in the Libcache structure, and the persistent part of the class directory is stored in a file called CLASSDIR. Class information is stored in a portable format. The routines wrclsdir (write class directory) and rdclsdir (read class directory) are used to import and export the public part of the directory. Another way to import public classes is to *require* a class. Dependent classes are also loaded into Libcache using rdclsdir.

The procedure newsym is used to create all sorts of symbol table entries. It checks for name conflicts in both the symbol table itself and the CLASS entries of the class directory cache. All symbols except for classes, class constants and methods are looked up in Symbols using the findsym procedure. Findsym

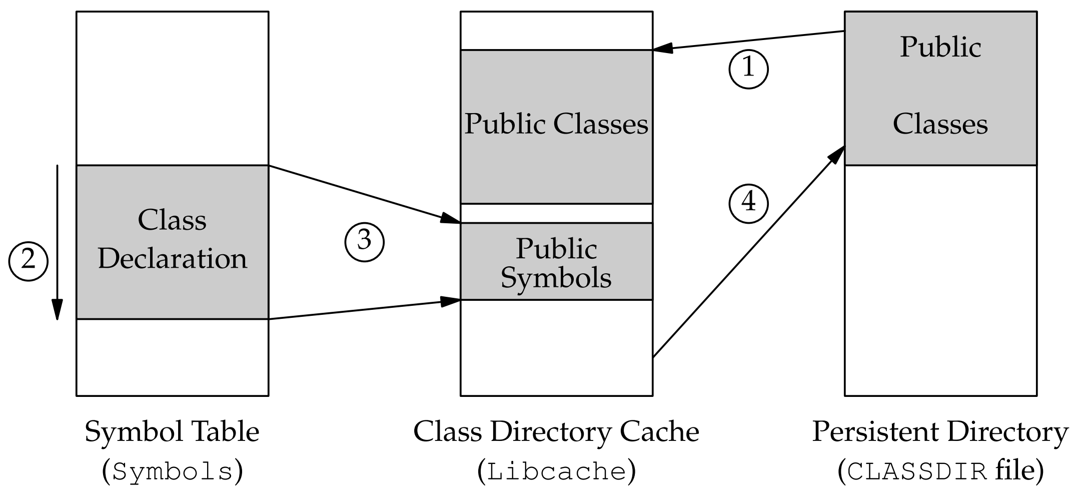


Figure 25: Class Level Scopes

can `not` find classes nor entries contained in closed class scopes, though. (A class scope is closed when the class declaration terminates.) To find classes and symbols in classes, the routines listed in table 31 are used.

Procedure	**In**
`findivar`	Find an instance symbol (constant or method) in a given symbol table (`Symbols` or `Libcache`).
`fintintclass`	Find a class declaration in the cache.
`findclass`	Find a class by first trying the cache and, if the class is not in the cache, loading it from an external class file.
`findredclass`	Find a which is required by a known class.
`findclassref`	Find a class which is required by the currently defined class or the current module.
`findmethod`	Find a method which can answer a given message.

Table 31: Class-level Symbol Management Routines

Each class consists of the code of its methods and a symbol table containing the type information of the methods and the values of the public constants. The symbol table of a class is kept in a file with the same structure as the persistent part of the class directory. Hence class information can be loaded into the class cache from such files using `rdclsdir`.

A method is found by decomposing a message into its object part and its method part. First the class of the object is looked up using the `SSIZE` flag of the object. Since the class object must be known at the time where the object is declared, the class *must* be in the cache. Using the class reference, the method name can be looked up by searching the scope of the class using `findivar`.

`Findclass` is used to load classes which are required by other modules and classes, if they are not in the cache already. `Findredclass` us used whereever a (closed) class declaration has to be found. The class to be found must already be in the cache. This routine is used to find the classes required in `OBJECT` statements and `SEND` operators as well as the values of class constants.

5

Tcode Optimization

This chapter contains the full source to the Tcode optimizer TXOPT. There exist two separate versions of TXOPT, one which processes the entire program in core and one which 'pages thru' the input program. The second version is required to optimize programs which are too big to be held in core in one single piece. The in-core version, however, is smaller and simpler. Because the same optimization techniques are used in both versions, the *in-core version* of TXOPT will be described in detail in the following sections.

5.1 The Commented TXOPT Listing

```
!       TXOPT -- A Tcode Optimizer for T3X.
!       Copyright (C) 1999,2000 Nils M Holm. All rights reserved.
!       See the file LICENSE for conditions of use.
!
!       SYNTAX: TXOPT <INFILE >OUTFILE

module txopt(t3x, tcode);

object  t[t3x];

const   BUFLEN          = 1024; ! I/O buffer size
const   CODESIZE        = 8192; ! max. code size per procedure
const   NLABELS         = 2048; ! max. labels per procedure
const   STACKLEN        = 64;   ! stack size for folding const exprs
const   XSTACKLEN       = 512;  ! expr stack size for tree opts

! Tcode instruction node
struct  TCI = tc_op, tc_arg, tc_left, tc_right;

var     Code::CODESIZE;                 ! code buffer
var     Ctop;                           ! code size
var     Labels[NLABELS], Lbase;         ! Labels, base label
var     Stack[STACKLEN], Sp;            ! folding stack, ptr
var     Inbuf::BUFLEN, Cp, Ep;          ! input buffer, char ptr, end ptr
var     Comtab;                         ! commutation table
var     Xstack[XSTACKLEN*TCI], X0, Xp;  ! expr stack, code ptr, stack ptr
var     Checkmagic;                     ! check magic flag, 1=check
var     K64;                            ! 65536 constant
var     Mode5;                          ! Tcode5 flag
var     W;                              ! Word size
var     Addr;                           ! Input position


init() do
        Cp := 0;
        Ep := 0;
        Checkmagic := 1;
        K64 := 16384;   ! avoid constant expression folding
        K64 := K64 * 4;
        Mode5 := 0;
        W := 2;
        Addr := 0;
        Code::(CODESIZE-1) := tcode.ILDG;
        Comtab := [
        ! P  instruction        replacement     (P = precedence)
        [ 1, tcode.INORM,       0               ],
        [ 1, tcode.INORMB,      tcode.INORMB    ],
        [ 1, tcode.IDEREF,      0               ],
        [ 1, tcode.IDREFB,      0               ],
        [ 2, tcode.IMUL,        tcode.IMUL      ],
        [ 2, tcode.IUMUL,       tcode.IUMUL     ],
        [ 2, tcode.IDIV,        0               ],
        [ 2, tcode.IUDIV,       0               ],
        [ 2, tcode.IMOD,        0               ],
        [ 3, tcode.IADD,        tcode.IADD      ],
        [ 3, tcode.ISUB,        0               ],
        [ 4, tcode.IBAND,       tcode.IBAND     ],
        [ 4, tcode.IBOR,        tcode.IBOR      ],
        [ 4, tcode.IBXOR,       tcode.IBXOR     ],
        [ 4, tcode.IBSHL,       0               ],
        [ 4, tcode.IBSHR,       0               ],
        [ 5, tcode.IEQU,        tcode.IEQU      ],
        [ 5, tcode.INEQU,       tcode.INEQU     ],
        [ 6, tcode.ILESS,       tcode.IGRTR     ],
        [ 6, tcode.IGRTR,       tcode.ILESS     ],
        [ 6, tcode.ILTEQ,       tcode.IGTEQ     ],
        [ 6, tcode.IGTEQ,       tcode.ILTEQ     ],
        [ 6, tcode.IULESS,      tcode.IUGRTR    ],
        [ 6, tcode.IUGRTR,      tcode.IULESS    ],
        [ 6, tcode.IULTEQ,      tcode.IULTEQ    ],
        [ 6, tcode.IUGTEQ,      tcode.IUGTEQ    ],
        [ 0, 0,                 0               ],
        ];
end


length(s) return t.memscan(s, 0, 32767);


writep(f, s) t.write(f, s, length(s));


nl(f) do var b::3;
        writep(f, t.newline(b));
end


! convert number to string
var     ntoa_buf::64;
ntoa(v) do var g, i;
        g := 0;
        if (v<0) do
                g := 1;
                v := -v;
        end
        ntoa_buf::63:= 0;
        i := 62;
        while (v \/ i = 62) do
                ntoa_buf::i := v mod 10 + '0'; i := i-1;
                v := v/10;
        end
        if (g) do
                ntoa_buf::i := '-'; i := i-1;
        end
        return @ntoa_buf::(i+1);
end


fatal(m, n) do
        writep(T3X.SYSERR, packed"TXOPT: ");
        writep(T3X.SYSERR, ntoa(Addr));
        writep(T3X.SYSERR, packed": ");
        writep(T3X.SYSERR, m);
        if (n) do
                writep(T3X.SYSERR, packed": ");
                writep(T3X.SYSERR, n);
        end
        nl(T3X.SYSERR);
        t.close(t.open(packed"T3X.ERR", T3X.OWRITE));
        writep(T3X.SYSERR, packed"terminating."); nl(T3X.SYSERR);
        halt 1;
end


getw(a) do var n;
        ie (Mode5) do
                n := (Code::(a+3) << 24) | (Code::(a+2) << 16) |
                        (Code::(a+1) << 8) | Code::a;
                return n;
        end
        else do
                n := Code::(a+1) << 8 | Code::a;
                return (n > 32767)-> n-K64: n;  ! 32-bit hack !
        end
end


putw(n, v) do
        Code::n := v & 255;
        Code::(n+1) := v >> 8;
        if (Mode5) do
                Code::(n+2) := v >> 16;
                Code::(n+3) := v >> 24;
        end
end


! advance to next Tcode instruction (starting at position p)
next(p) do var i;
        i := Code::p;
        if (i = tcode.IINIT) return p+5;
        p := p+1;
        if (i & 128) do
                p := p+W;
                if (    i = tcode.IINCL \/ i = tcode.IINCG \/
                        i = tcode.IINCI \/
                        i = tcode.IINIT \/
                        i = tcode.IPUB \/ i = tcode.IEXT \/
                        i = tcode.ILSYM \/ i = tcode.IGSYM \/ i = tcode.IISYM
                )
                        p := p+W;
                if (    i = tcode.ISTR \/ i = tcode.IPSTR \/
                        i = tcode.IPUB \/ i = tcode.IEXT \/
                        i = tcode.IGSYM \/ i = tcode.ILSYM \/ i = tcode.IISYM
                )
                        p := p+getw(p-W);
        end
        return p;
end


rdch() do
        if (Cp >= Ep) do
                if (Ep = -1) return -1;
                Ep := t.read(T3X.SYSIN, Inbuf, BUFLEN);
                if (Ep < 1) do
                        Ep := -1;
                        return -1;
                end
                Cp := 0;
        end
        Cp := Cp+1;
        Addr := Addr+1;
        return Inbuf::(Cp-1);
end


readop(p) do var k;
        ie (Mode5) do
                k := rdch() | (rdch()<<8) | (rdch()<<16) | (rdch()<<24);
                p::0 := k;
                p::1 := k>>8;
                p::2 := k>>16;
                p::3 := k>>24;
        end
        else do
                k := rdch() | (rdch()<<8);
                p::0 := k;
                p::1 := k>>8;
        end
        return k;
end


! copy an instruction into memory
! i = first byte of instruction to copy
copyi(i) do var k, j;
        Code::Ctop := i;
        Ctop := Ctop+1;
        if (Ctop >= CODESIZE-6) fatal(packed"procedure too big", 0);
        if (i & 128) do
                k := readop(@Code::Ctop);
                Ctop := Ctop+W;
                ! adjust Lbase if necessary
                if (i = tcode.ICLAB /\ k < Lbase) Lbase := k;
                if (    i = tcode.IINCL \/ i = tcode.IINCG \/
                        i = tcode.IINCI \/
                        i = tcode.IINIT \/
                        i = tcode.IPUB \/ i = tcode.IEXT \/
                        i = tcode.ILSYM \/ i = tcode.IGSYM \/ i = tcode.IISYM
                ) do
```

```
                        if (i = tcode.IINIT) do
                                if (k & 256)
                                        fatal(packed"precompiled module", 0);
                                if (k \= 4 /\ k \= 5)
                                        fatal(packed"
                                                unsupported Tcode version",
                                                ntoa(k));
                        end
                        j := k;
                        k := readop(@Code::Ctop);
                        Ctop := Ctop+W;
                        if (i = tcode.IINIT /\ j = 5) do
                                if (t.bpw() < 4) fatal(packed
                                  "Tcode5 not supported on this platform", 0);
                                Mode5 := 1;
                                W := 4;
                        end
                end
                if (    i = tcode.ISTR \/ i = tcode.IPSTR \/
                        i = tcode.IPUB \/ i = tcode.IEXT \/
                        i = tcode.IGSYM \/ i = tcode.ILSYM \/ i = tcode.IISYM
                ) do
                        if (Ctop + k >= CODESIZE)
                                fatal(packed"procedure too big", 0);
                        for (j=0, k) do
                                Code::(Ctop+j) := rdch();
                        end
                        Ctop := Ctop+k;
                end
        end
        return rdch();
end


! read a procedure and its prefix (data declarations) into memory
readproc() do var op, k;
        Ctop := 0;
        Lbase := 32767; ! will be adjusted by copyi()
        op := rdch();
        if (Checkmagic)
                if (op \= tcode.IINIT)
                        fatal(packed"magic match failed", 0);
        Checkmagic := 0;
        ! read everything up to beginning of next procedure
        while (op \= tcode.IHDR /\ op \= tcode.IMHDR /\ op \= -1)
                op := copyi(op);
        if (op = -1) return -1;
        ! read everything up to end of procedure
        while (op \= tcode.IEND /\ op \= tcode.IENDM /\ op \= -1)
                op := copyi(op);
        if (op = -1) fatal(packed"unexpected end of procedure", 0);
        Code::Ctop := op;
        Ctop := Ctop+1;
end


! 'nop out' a three-byte instruction
nop3(p) do
        ie (Mode5) do
                ! which in fact is a 5-byte instruction in Tcode5
                Code::p := tcode.INOP;
                Code::(p+1) := tcode.INOP;
                Code::(p+2) := tcode.INOP;
                Code::(p+3) := tcode.INOP;
                Code::(p+4) := tcode.INOP;
        end
        else do
                Code::p := tcode.INOP;
                Code::(p+1) := tcode.INOP;
                Code::(p+2) := tcode.INOP;
        end
end


! fold multiply instructions
! p  = position in Code array
! a1 = address of first argument
! a2 = address of second argument
! t1 = type of first argument
! t2 = type of second argument
! v1 = value of first argument
! v2 = value of second argument
fold_mul(p, a1, a2, t1, t2, v1, v2) do
        if (    t1 = tcode.INUM /\ v1 = 0 \/            ! x*0, 0*x --> 0
                t2 = tcode.INUM /\ v2 = 0
        ) do
                nop3(a2);
                Code::a1 := tcode.INUM;
                putw(a1+1, 0);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                return %1;
        end
        if (t1 = tcode.INUM /\ v1 = 1) do               ! x*1 --> x
                nop3(a1);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                Stack[Sp-1] := a2;
                return %1;
        end
        if (t2 = tcode.INUM /\ v2 = 1) do               ! 1*x --> x
                nop3(a2);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                return %1;
        end
        if (t1 = tcode.INUM /\ v1 = %1) do              ! x*-1 --> -x
                nop3(a1);
                Code::p := tcode.INEG;
                Sp := Sp-1;
                Stack[Sp-1] := a2;
                return %1;
        end
        if (t2 = tcode.INUM /\ v2 = %1) do              ! -1*x --> -x
                nop3(a2);
                Code::p := tcode.INEG;
                Sp := Sp-1;
                return %1;
        end
        return 0;
```

```
end


! fold divide instructions
! p  = position in Code array
! a1 = address of first argument
! a2 = address of second argument
! t1 = type of first argument
! t2 = type of second argument
! v1 = value of first argument
! v2 = value of second argument
fold_div(p, a1, a2, t1, t2, v1, v2) do
        if (t1 = tcode.INUM /\ v1 = 0) do               ! 0/x --> 0
                nop3(a2);
                Code::a1 := tcode.INUM;
                putw(a1+1, 0);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                return %1;
        end
        if (t2 = tcode.INUM /\ v2 = 0) do               ! x/0 --> _|_
                fatal(packed"constant divide by zero", 0);
        end
        if (t2 = tcode.INUM /\ v2 = 1) do               ! x/1 --> x
                nop3(a2);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                return %1;
        end
        if (t2 = tcode.INUM /\ v2 = %1) do              ! x/-1 --> -x
                nop3(a2);
                Code::p := tcode.INEG;
                Sp := Sp-1;
                return %1;
        end
        return 0;
end


! fold modulo instructions
! p  = position in Code array
! a1 = address of first argument
! a2 = address of second argument
! t1 = type of first argument
! t2 = type of second argument
! v1 = value of first argument
! v2 = value of second argument
fold_mod(p, a1, a2, t1, t2, v1, v2) do
        if (t1 = tcode.INUM /\ v1 = 0) do               ! 0 mod x --> 0
                nop3(a2);
                Code::a1 := tcode.INUM;
                putw(a1+1, 0);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                return %1;
        end
        if (t2 = tcode.INUM /\ v2 = 0) do               ! x mod 0 --> _|_
                fatal(packed"constant divide by zero", 0);
        end
        if (t2 = tcode.INUM /\ v2 = 1) do               ! x mod 1 --> 0
```

```
                nop3(a2);
                Code::p := tcode.INOP;
                Code::a1 := tcode.INUM;
                putw(a1+1, 0);
                Sp := Sp-1;
                return %1;
        end
        return 0;
end


! fold add instructions
! p  = position in Code array
! a1 = address of first argument
! a2 = address of second argument
! t1 = type of first argument
! t2 = type of second argument
! v1 = value of first argument
! v2 = value of second argument
fold_add(p, a1, a2, t1, t2, v1, v2) do
        if (t1 = tcode.INUM /\ v1 = 0) do               ! 0+x --> x
                nop3(a1);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                Stack[Sp-1] := a2;
                return %1;
        end
        if (t2 = tcode.INUM /\ v2 = 0) do               ! x+0 --> x
                nop3(a2);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                return %1;
        end
        return 0;
end


! fold subtract instructions
! p  = position in Code array
! a1 = address of first argument
! a2 = address of second argument
! t1 = type of first argument
! t2 = type of second argument
! v1 = value of first argument
! v2 = value of second argument
fold_sub(p, a1, a2, t1, t2, v1, v2) do
        if (t1 = tcode.INUM /\ v1 = 0) do               ! 0-x --> -x
                nop3(a1);
                Code::p := tcode.INEG;
                Sp := Sp-1;
                Stack[Sp-1] := a2;
                return %1;
        end
        if (t2 = tcode.INUM /\ v2 = 0) do               ! x-0 --> x
                nop3(a2);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                return %1;
        end
        return 0;
```

```
end


! fold binary AND instructions
! p  = position in Code array
! a1 = address of first argument
! a2 = address of second argument
! t1 = type of first argument
! t2 = type of second argument
! v1 = value of first argument
! v2 = value of second argument
fold_band(p, a1, a2, t1, t2, v1, v2) do
        if (    t1 = tcode.INUM /\ v1 = 0 \/
                t2 = tcode.INUM /\ v2 = 0
        ) do                                            ! 0&x, x&0 --> 0
                nop3(a2);
                Code::a1 := tcode.INUM;
                putw(a1+1, 0);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                return %1;
        end
        if (t1 = tcode.INUM /\ v1 = %1) do              ! %1&x --> x
                nop3(a1);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                Stack[Sp-1] := a2;
                return %1;
        end
        if (t2 = tcode.INUM /\ v2 = %1) do              ! x&%1 --> x
                nop3(a2);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                return %1;
        end
        return 0;
end


! fold binary OR instructions
! p  = position in Code array
! a1 = address of first argument
! a2 = address of second argument
! t1 = type of first argument
! t2 = type of second argument
! v1 = value of first argument
! v2 = value of second argument
fold_bor(p, a1, a2, t1, t2, v1, v2) do
        if (    t1 = tcode.INUM /\ v1 = %1 \/
                t2 = tcode.INUM /\ v2 =%1
        ) do                                            ! %1|x, x|%1 --> %1
                nop3(a2);
                Code::a1 := tcode.INUM;
                putw(a1+1, %1);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                return %1;
        end
        if (t1 = tcode.INUM /\ v1 = 0) do               ! 0|x --> x
                nop3(a1);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                Stack[Sp-1] := a2;
                return %1;
        end
        if (t2 = tcode.INUM /\ v2 = 0) do               ! x|0 --> x
                nop3(a2);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                return %1;
        end
        return 0;
end


! fold binary SHIFT LEFT instructions
! p  = position in Code array
! a1 = address of first argument
! a2 = address of second argument
! t1 = type of first argument
! t2 = type of second argument
! v1 = value of first argument
! v2 = value of second argument
fold_bshl(p, a1, a2, t1, t2, v1, v2) do
        if (t1 = tcode.INUM /\ v1 = 0) do               ! 0<<x --> 0
                nop3(a2);
                Code::a1 := tcode.INUM;
                putw(a1+1, 0);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                return %1;
        end
        if (t2 = tcode.INUM /\ v2 = 0) do               ! x<<0 --> x
                nop3(a2);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                return %1;
        end
        return 0;
end


! fold binary SHIFT RIGHT instructions - see SHIFT LEFT, above
fold_bshr(p, a1, a2, t1, t2, v1, v2)
        return fold_bshl(p, a1, a2, t1, t2, v1, v2);


! fold binary XOR instructions
! p  = position in Code array
! a1 = address of first argument
! a2 = address of second argument
! t1 = type of first argument
! t2 = type of second argument
! v1 = value of first argument
! v2 = value of second argument
fold_bxor(p, a1, a2, t1, t2, v1, v2) do
        if (t1 = tcode.INUM /\ v1 = %1) do              ! x^%1 --> ~x
                nop3(a1);
                Code::p := tcode.IBNOT;
                Sp := Sp-1;
                Stack[Sp-1] := a2;
                return %1;
        end
        if (t2 = tcode.INUM /\ v2 = %1) do              ! x^%1 --> ~x
                nop3(a2);
                Code::p := tcode.IBNOT;
                Sp := Sp-1;
                return %1;
        end
        return 0;
end


! fold NORM instruction (pointer normalization)
! p  = position in Code array
! a1 = address of first argument
! a2 = address of second argument
! t1 = type of first argument
! t2 = type of second argument
! v1 = value of first argument
! v2 = value of second argument
fold_norm(p, a1, a2, t1, t2, v1, v2) do
        ! Fold the address computation of local vector members
        ! at constant offsets
        if (t1 = tcode.ILDLV /\ t2 = tcode.INUM) do     ! @x[N] --> @(x-N)
                putw(a1+1, v1 - v2);
                nop3(a2);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                return %1;
        end
        if (t2 = tcode.INUM /\ v2 = 0) do               ! @x[0] --> @x
                nop3(a2);
                Code::p := tcode.INOP;
                return %1;
        end
        return 0;
end


! fold unsigned multiply instructions
! p  = position in Code array
! a1 = address of first argument
! a2 = address of second argument
! t1 = type of first argument
! t2 = type of second argument
! v1 = value of first argument
! v2 = value of second argument
fold_umul(p, a1, a2, t1, t2, v1, v2) do
        if (    t1 = tcode.INUM /\ v1 = 0 \/            ! x.*0, 0.*x --> 0
                t2 = tcode.INUM /\ v2 = 0
        ) do
                nop3(a2);
                Code::a1 := tcode.INUM;
                putw(a1+1, 0);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                return %1;
        end
        if (t1 = tcode.INUM /\ v1 = 1) do               ! x.*1 --> x
```

```
                nop3(a1);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                Stack[Sp-1] := a2;
                return %1;
        end
        if (t2 = tcode.INUM /\ v2 = 1) do               ! 1.*x --> x
                nop3(a2);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                return %1;
        end
        return 0;
end


! fold unsigned divide instructions
! p  = position in Code array
! a1 = address of first argument
! a2 = address of second argument
! t1 = type of first argument
! t2 = type of second argument
! v1 = value of first argument
! v2 = value of second argument
fold_udiv(p, a1, a2, t1, t2, v1, v2) do
        if (t1 = tcode.INUM /\ v1 = 0) do               ! 0./x --> 0
                nop3(a2);
                Code::a1 := tcode.INUM;
                putw(a1+1, 0);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                return %1;
        end
        if (t2 = tcode.INUM /\ v2 = 0) do               ! x./0 --> _|_
                fatal(packed"constant divide by zero", 0);
        end
        if (t2 = tcode.INUM /\ v2 = 1) do               ! x./1 --> x
                nop3(a2);
                Code::p := tcode.INOP;
                Sp := Sp-1;
                return %1;
        end
        return 0;
end


! fold binary expressions
! p  = position in Code array
fold_binary(p) do var op, t1, t2, a1, a2, v1, v2;
        op := Code::p;
        a1 := Stack[Sp-2];
        a2 := Stack[Sp-1];
        t1 := Code::a1;
        t2 := Code::a2;
        v1 := getw(a1+1);
        v2 := getw(a2+1);
        Sp := Sp-1;
        ! Fold operations with two constant operands
        ! constant OP constant --> constant
        if (t1 = tcode.INUM /\ t2 = tcode.INUM) do
```

```
                if (\v2 /\ (op = tcode.IMOD \/ op = tcode.IDIV))
                        fatal(packed"constant divide by zero", 0);
                ie (op = tcode.IMUL) v1 := v1 * v2;
                else ie (op = tcode.IDIV) v1 := v1 / v2;
                else ie (op = tcode.IMOD) v1 := v1 mod v2;
                else ie (op = tcode.IADD) v1 := v1 + v2;
                else ie (op = tcode.ISUB) v1 := v1 - v2;
                else ie (op = tcode.IBAND) v1 := v1 & v2;
                else ie (op = tcode.IBOR) v1 := v1 | v2;
                else ie (op = tcode.IBXOR) v1 := v1 ^ v2;
                else ie (op = tcode.IBSHL) v1 := v1 << v2;
                else ie (op = tcode.IBSHR) v1 := v1 >> v2;
                else ie (op = tcode.IEQU) v1 := v1 = v2;
                else ie (op = tcode.INEQU) v1 := v1 \= v2;
                else ie (op = tcode.ILESS) v1 := v1 < v2;
                else ie (op = tcode.IGRTR) v1 := v1 > v2;
                else ie (op = tcode.ILTEQ) v1 := v1 <= v2;
                else ie (op = tcode.IGTEQ) v1 := v1 >= v2;
                else ie (op = tcode.IUMUL) v1 := v1 .* v2;
                else ie (op = tcode.IUDIV) v1 := v1 ./ v2;
                else ie (op = tcode.IULESS) v1 := v1 .< v2;
                else ie (op = tcode.IUGRTR) v1 := v1 .> v2;
                else ie (op = tcode.IULTEQ) v1 := v1 .<= v2;
                else if (op = tcode.IUGTEQ) v1 := v1 .>= v2;
                putw(a1+1, v1);
                nop3(a2);
                Code::p := tcode.INOP;
                return 0;
        end
        if (op = tcode.INORM /\ fold_norm(p, a1, a2, t1, t2, v1, v2))
                return 0;
        if (op = tcode.IMUL /\ fold_mul(p, a1, a2, t1, t2, v1, v2)) return 0;
        if (op = tcode.IDIV /\ fold_div(p, a1, a2, t1, t2, v1, v2)) return 0;
        if (op = tcode.IMOD /\ fold_mod(p, a1, a2, t1, t2, v1, v2)) return 0;
        if (op = tcode.IADD /\ fold_add(p, a1, a2, t1, t2, v1, v2)) return 0;
        if (op = tcode.ISUB /\ fold_sub(p, a1, a2, t1, t2, v1, v2)) return 0;
        if (op = tcode.IBAND /\ fold_band(p, a1, a2, t1, t2, v1, v2)) return 0;
        if (op = tcode.IBOR /\ fold_bor(p, a1, a2, t1, t2, v1, v2)) return 0;
        if (op = tcode.IBSHL /\ fold_bshl(p, a1, a2, t1, t2, v1, v2)) return 0;
        if (op = tcode.IBSHR /\ fold_bshl(p, a1, a2, t1, t2, v1, v2)) return 0;
        if (op = tcode.IBXOR /\ fold_bxor(p, a1, a2, t1, t2, v1, v2)) return 0;
        if (op = tcode.IUMUL /\ fold_umul(p, a1, a2, t1, t2, v1, v2)) return 0;
        if (op = tcode.IUDIV /\ fold_udiv(p, a1, a2, t1, t2, v1, v2)) return 0;
        ! otherwise, make sure the result is non-constant
        Stack[Sp-1] := CODESIZE-1;      ! tcode.ILDG
end


! fold increments
! (make advantage of the INCx instructions)
! p     = position of SAVx
! start = beginnig of potential increment (LDx a NUM y ADD SAVx)
fold_inc(p, start) do           ! x := x+N --> inc(x,N)
        var     op, a;          ! x := x-N --> inc(x,-N)
        var     q, q2;

        op := Code::p;
        if (    Code::start = tcode.ILDL /\ op = tcode.ISAVL \/
                Code::start = tcode.ILDG /\ op = tcode.ISAVG \/
                Code::start = tcode.ILDI /\ op = tcode.ISAVI
        ) do
                ! make sure that LD and SAV reference the same address
                ! and the increment is constant
                a := getw(start+1);
                if (a \= getw(p+1)) return 0;
                q := next(start);
                if (Code::q \= tcode.INUM) return 0;
                q2 := next(q);
                if (Code::q2 \= tcode.IADD /\ Code::q2 \= tcode.ISUB) return 0;
                ! replace LDx a NUM y ADD SAVx with INCx a y
                Code::start := Code::start = tcode.ILDL-> tcode.IINCL:
                                Code::start = tcode.ILDI-> tcode.IINCI:
                                        tcode.IINCG;
                putw(start+W+1, Code::q2 = tcode.ISUB-> -getw(start+W+2):
                        getw(start+W+2));
                nop3(start+W*2+1);
                nop3(p);
        end
end


! fold all expressions in the current procedure
foldexpr() do
        var     i, op, a, v;
        var     state;
        var     last[3];

        Sp := 0;
        i := 0;
        last[0] := 0;   ! queue, used to match increments
        last[1] := 0;
        last[2] := 0;
        while (i < Ctop) do
                op := Code::i;
                ie (    op = tcode.ILDG \/ op = tcode.ILDGV \/
                        op = tcode.ILDL \/ op = tcode.ILDLV \/
                        op = tcode.ILDLAB \/ op = tcode.INUM
                ) do
                        if (Sp+1 >= STACKLEN)
                                fatal(packed"FOLDEXPR: stack overflow", 0);
                        Stack[Sp] := i;
                        Sp := Sp+1;
                end
                else ie (Sp /\ (op = tcode.INEG \/ op = tcode.ILNOT \/
                        op = tcode.IBNOT)
                ) do
                        ! unary operations, fold if argument is constant
                        ie (Code::(Stack[Sp-1]) = tcode.INUM) do
                                a := Stack[Sp-1];
                                v := getw(a+1);
                                ie (op = tcode.INEG) v := -v;
                                else ie (op = tcode.ILNOT) v := \v;
                                else v := ~v;
                                putw(a+1, v);
                                Code::i := tcode.INOP;
                        end
                        else do
                                ! make sure it is non-constant
                                Stack[Sp-1] := CODESIZE-1;
                        end
                end
                else ie (Sp>1 /\
                        (op = tcode.IMUL \/
                        op = tcode.IDIV \/ op = tcode.IMOD \/
                        op = tcode.IADD \/ op = tcode.ISUB \/
                        op = tcode.IBAND \/ op = tcode.IBOR \/
                        op = tcode.IBXOR \/ op = tcode.IBSHL \/
                        op = tcode.IBSHR \/
                        op = tcode.IEQU \/ op = tcode.INEQU \/
                        op = tcode.ILESS \/ op = tcode.IGRTR \/
                        op = tcode.ILTEQ \/ op = tcode.IGTEQ \/
                        op = tcode.IUMUL \/ op = tcode.IUDIV \/
                        op = tcode.IULESS \/
                        op = tcode.IUGRTR \/ op = tcode.IULTEQ \/
                        op = tcode.IUGRTR)
                ) do
                        fold_binary(i);
                end
                else ie (op = tcode.IDLAB \/ op = tcode.IDECL \/
                        op = tcode.IDATA \/ op = tcode.ICREF \/
                        op = tcode.IDREF \/ op = tcode.ISTR \/
                        op = tcode.IPSTR \/ op = tcode.INOP \/
                        op = tcode.ILINE \/ op = tcode.IGSYM \/
                        op = tcode.ILSYM \/ op = tcode.IISYM
                ) do
                        ! labels: ignore
                end
                else do
                        Sp := 0;
                        ! SAVx may be the final instruction of an increment
                        if (    (op = tcode.ISAVL \/ op = tcode.ISAVG \/
                                 op = tcode.ISAVI) /\
                                last[0]
                        )
                                fold_inc(i, last[0]);
                end
                last[0] := last[1];
                last[1] := last[2];
                last[2] := i;
                i := next(i);
        end
end


! fold all conditions in the current procedure
foldcond() do var       i, v, op, last;
        i := 0;
        last := 0;      ! last non-NOP instruction
        while (i < Ctop) do
                op := Code::i;
                ie (    (op = tcode.IBRF \/ op = tcode.IBRT) /\
                        Code::last = tcode.INUM
                ) do
                        v := getw(last+1);
                        ! NUM T BRT L --> JUMP L
                        ! NUM F BRT L --> NOP
                        ie (op = tcode.IBRT) do
                                nop3(last);
                                ie (v)
                                        Code::i := tcode.IJUMP;
                                else
                                        nop3(i);
                        end
                        ! NUM T BRF L --> NOP
                        ! NUM F BRF L --> JUMP L
                        else do ! BRF
                                nop3(last);
                                ie (v)
                                        nop3(i);
                                else
                                        Code::i := tcode.IJUMP;
                        end
                end
                else if ((op = tcode.IBRF \/ op = tcode.IBRT) /\
                        Code::last = tcode.ILNOT
                ) do
                        ! NOT BRT  --> BRF
                        ! NOT BRF  --> BRT
                        ! NOT NBRT --> NBRF
                        ! NOT NBRF --> NBRT
                        Code::i := op = tcode.IBRT-> tcode.IBRF:
                                op = tcode.IBRF-> tcode.IBRT:
                                op = tcode.INBRT-> tcode.INBRF:
                                tcode.INBRT;
                        Code::last := tcode.INOP;
                end
                if (Code::i \= tcode.INOP) last := i;
                i := next(i);
        end
end


! remove all NOPs from the current procedure
condense() do var i, j, k, m;
        i := 0;
        j := 0;
        while (i < Ctop) do
                k := next(i) - i;
                if (Code::i \= tcode.INOP) do
                        t.memcopy(@Code::j, @Code::i, k);
                        j := j+k;
                end
                i := i+k;
        end
        k := Ctop;
        Ctop := j;
        return k-Ctop;
end


! XXX WARNING: the is...() procedures work only with Tcode 4

isload(x) return x & 127 >= tcode.ILDG & 127 /\ x & 127 <= tcode.ISELF;


isbinop(x) return x >= tcode.IMUL /\ x <= tcode.IUGTEQ \/ x >= tcode.IDEREF /\
        x <= tcode.INORMB;


isexpr(x) return
```

```
        x = tcode.INEG \/
        x = tcode.ILNOT \/
        x = tcode.IBNOT \/
        isbinop(x) \/
        isload(x) \/
        x = tcode.IDUP \/
        x = tcode.ISWAP;


! Check for a common subscript (like in v[x+y] := v[x+y] + 1;)
common(a, b) do var op;
        if (a = %1) return b = %1;
        if (b = %1) return a = %1;
        if (\common(Xstack[a+tc_left], Xstack[b+tc_left])) return 0;
        if (\common(Xstack[a+tc_right], Xstack[b+tc_right])) return 0;
        op := Xstack[a+tc_op];
        if (op \= Xstack[b+tc_op]) return 0;
        if (Xstack[a+tc_arg] \= Xstack[b+tc_arg]) return 0;
        ! A procedure call in a subscript may have a side-effect!
        if (    op = tcode.ICALL \/ op = tcode.ICALR \/
                op = tcode.ICALX \/ op = tcode.ISYS
        )
                return 0;
        return %1;
end


! elimininate a commom reference EXPR from an expression of the form
! VAR[EXPR] := VAR[EXPR] op EXPR2
! or
! VAR::EXPR := VAR::EXPR op EXPR2
commref(args) do var a0, a1, op, op2, new;
        a0 := args[0];  ! tree repr. lefthand side of :=
        a1 := args[1];  ! tree repr. righthand side of :=
        if (a0 = %1 \/ a1 = %1 \/ Xstack[a1+tc_left] = %1) return;
        op := Xstack[a0+tc_op];
        if (op \= tcode.INORM /\ op \= tcode.INORMB) return;
        op2 := Xstack[Xstack[a1+tc_left]+tc_op];
        if (    \((op = tcode.INORM /\ op2 = tcode.IDEREF) \/
                 (op = tcode.INORMB /\ op2 = tcode.IDREFB))
        )
                return;
        if (    \common(Xstack[a0+tc_left],
                        Xstack[Xstack[a1+tc_left]+tc_left])
        )
                return;
        if (    \common(Xstack[a0+tc_right],
                        Xstack[Xstack[a1+tc_left]+tc_right])
        )
                return;
        ! Replace VAR EXPR DEREF on righthand side
        ! with DUP NUM 0 DEREF,
        ! where DUP duplicates VAR EXPR NORM.
        ! NORM,DEREF may be exchanged with NORMB,DREFB.
        a1 := Xstack[a1+tc_left];
        if (a1 = %1 \/ Xstack[a1+tc_left] = %1 \/ Xstack[a1+tc_right] = %1)
                return;
        new := Xstack[a1+tc_left];
        Xstack[new+tc_op] := tcode.IDUP;
        Xstack[new+tc_left] := %1;
        Xstack[new+tc_right] := %1;
        new := Xstack[a1+tc_right];
        Xstack[new+tc_op] := tcode.INUM;
        Xstack[new+tc_arg] := 0;
        Xstack[new+tc_left] := %1;
        Xstack[new+tc_right] := %1;
end


! Re-order the arguments of certain operations
! to minimize the run time stack space.
! For example,
!       A B C + *
! takes three stack cells, but
!       B C + A *
! only two. Of course, this works only because
!       A*B = B*A
! Note that A<B can be replaced with B>A.
reorder(p) do var op, base, left, right, tmp;
        if (p = %1) return 0;           ! NIL
        reorder(Xstack[p+tc_left]);
        reorder(Xstack[p+tc_right]);
        op := Xstack[p+tc_op];
        if (\isbinop(op)) return 0;
        ! Search op in Comtbl.
        ! IENDOFSET is a resonable limit.
        for (base=0, tcode.IENDOFSET) do
                if (\Comtab[base][0]) return 0; ! op not commutative
                if (op = Comtab[base][1]) do
                        ! No replacement? Give up for now.
                        ! (May also inject SWAP.)
                        if (\Comtab[base][2]) return 0;
                        leave;
                end
        end
        ! find opcodes of arguments
        op := Xstack[Xstack[p+tc_left]+tc_op];
        for (left=0, tcode.IENDOFSET)
                if (\Comtab[left][0] \/ op = Comtab[left][1])
                        leave;
        op := Xstack[Xstack[p+tc_right]+tc_op];
        for (right=0, tcode.IENDOFSET)
                if (\Comtab[right][0] \/ op = Comtab[right][1])
                        leave;
        ! if no information is available on either argument, give up
        if (\Comtab[right][0] \/ Comtab[left][0]) return 0;
        ! if the right operand has higher precedence (lower value)
        ! then the left one, swap them
        if (Comtab[right][0] < Comtab[base][0]) do
                tmp := Xstack[p+tc_left];
                Xstack[p+tc_left] := Xstack[p+tc_right];
                Xstack[p+tc_right] := tmp;
                ! replace the opcode
                Xstack[p+tc_op] := Comtab[base][2];
        end
end


! write back a (modified) tree
writeback(p) do
```

```
        if (p = %1) return 0;           ! NIL
        writeback(Xstack[p+tc_left]);
        writeback(Xstack[p+tc_right]);
        ! glue node? return.
        if (\Xstack[p+tc_op]) return 0;
        Code::X0 := Xstack[p+tc_op];
        X0 := X0+1;
        if (Xstack[p+tc_op] & 128) do
                ie (Mode5) do
                        Code::X0 := Xstack[p+tc_arg];
                        Code::(X0+1) := Xstack[p+tc_arg] >> 8;
                        Code::(X0+2) := Xstack[p+tc_arg] >> 16;
                        Code::(X0+3) := Xstack[p+tc_arg] >> 24;
                        X0 := X0+4;
                end
                else do
                        Code::X0 := Xstack[p+tc_arg];
                        Code::(X0+1) := Xstack[p+tc_arg] >> 8;
                        X0 := X0+2;
                end
        end
end


dumptree(p,d) do var i;
        if (p = %1) return 0;
        if (d>1) for (i=0,d-1) writep(T3X.SYSERR, packed".");
        writep(T3X.SYSERR, ntoa(Xstack[p+tc_op]));
        writep(T3X.SYSERR, packed"\s");
        writep(T3X.SYSERR, ntoa(Xstack[p+tc_arg]));
        nl(T3X.SYSERR);
        if (Xstack[p+tc_left] \= %1) writep(T3X.SYSERR, packed"L");
        dumptree(Xstack[p+tc_left], d+1);
        if (Xstack[p+tc_right] \= %1) writep(T3X.SYSERR, packed"R");
        dumptree(Xstack[p+tc_right], d+1);
end


treeopts() do
        var     i, j, n, p, cxedone;
        var     args[XSTACKLEN], ap;
        var     op;

        i := 0;
        ! X0 holds the address of the code from which the tree was generated.
        X0 := %1;
        cxedone := 0;           ! successful common expr elimination?
        while (i < Ctop) do
                ie (isexpr(Code::i)) do
                        if (X0 = %1) do
                                X0 := i;
                                Xp := 0;
                                ap := 0;
                        end
                        op := Code::i;
                        ! if expr to big, give up
                        if (Xp >= XSTACKLEN*TCI-TCI) X0 := %1;
                        ! push instruction
                        Xstack[Xp+tc_op] := op;
                        ie (op & 128) do
                                ie (Mode5)
                                        Xstack[Xp+tc_arg] :=
                                                (Code::(i+4)<<24) |
                                                (Code::(i+3)<<16) |
                                                (Code::(i+2)<<8) |
                                                Code::(i+1);
                                else
                                        Xstack[Xp+tc_arg] :=
                                                Code::(i+2)<<8 | Code::(i+1);
                        end
                        else do
                                Xstack[Xp+tc_arg] := 0;
                        end
                        ie (isload(op)) do
                                args[ap] := Xp;
                                ap := ap+1;
                                ! load instructions are leafes
                                Xstack[Xp+tc_left] := %1;
                                Xstack[Xp+tc_right] := %1;
                        end
                        else ie (isbinop(op)) do
                                ie (ap < 2) do
                                        X0 := %1;
                                end
                                else do
                                        ! link arguments to top expr
                                        Xstack[Xp+tc_left] := args[ap-2];
                                        Xstack[Xp+tc_right] := args[ap-1];
                                        ! remove right arg
                                        ap := ap-1;
                                        ! replace left arg with complete
                                        ! binary operation
                                        args[ap-1] := Xp;
                                end
                        end
                        else do
                                ! if there's something on the stack,
                                ! it must be a unary operation
                                ie (Xp) do
                                        ! link argument to top expr
                                        Xstack[Xp+tc_left] := Xp-TCI;
                                        Xstack[Xp+tc_right] := %1;
                                        ! replace last arg with complete
                                        ! unary operation
                                        args[ap-1] := Xp;
                                end
                                else do
                                        X0 := %1;
                                end
                        end
                        Xp := Xp+TCI;
                end
                else do
                        ! Current instr is not part of an expression:
                        ! Run the optimizations on the generated tree,
                        ! if any.
                        if (X0 \= %1) do
                                if (ap = 2) commref(args);
                                for (j=0, ap) do
                                        reorder(args[j]);
```

```
                                        writeback(args[j]);
                                end
                                ! X0 < current position after update?
                                ! Fill with NOPs.
                                if (X0 < i) do
                                        while (X0 < i) do
                                                Code::X0 := tcode.INOP;
                                                X0 := X0+1;
                                        end
                                        cxedone := 1;
                                end
                                X0 := %1;       ! delete tree
                        end
                end
                i := next(i);
        end
        if (cxedone) condense();
end


! Redirect jumps to jumps and eliminate zero-length jumps
jumpredir() do var i, j, k, jrm;
        for (i=0, NLABELS) Labels[i] := %1;
        jrm := 0;       ! jumps removed?
        i := 0;
        while (i < Ctop) do
                if (Code::i = tcode.ICLAB) do
                        j := getw(i+1) - Lbase;
                        if (j >= NLABELS) fatal(packed"too many labels", 0);
                        ! make label (i+W+1 = code following CLAB)
                        Labels[j] := i+W+1;
                        ! JUMP x CLAB x --> CLAB x
                        if (    i >= W+1 /\ Code::(i-W-1) = tcode.IJUMP /\
                                getw(i-W) = j+Lbase
                        ) do
                                nop3(i-W-1);
                                jrm := 1;
                        end
                end
                i := next(i);
        end
        i := 0;
        while (i < Ctop) do
                if (Code::i = tcode.IJUMP) do
                        j := i;         ! remember beginning of jump chain
                        while (1) do
                                if (Code::j \= tcode.IJUMP) leave;
                                k := getw(j+1) - Lbase;
                                if (k < 0 \/ k >= NLABELS \/ Labels[k] = %1)
                                        leave;
                                j := Labels[k];
                                if (i = j) leave;       ! cycle detection
                        end
                        if (i \= j) putw(i+1, k+Lbase);
                end
                i := next(i);
        end
        if (jrm) condense();
end
```

```
writeproc() if (Ctop) if (t.write(T3X.SYSOUT, Code, Ctop) \= Ctop)
                fatal(packed"file write error", 0);


do var eof;
        init();
        eof := 0;
        while (\eof) do
                eof := readproc();
                foldexpr();
                foldcond();
                condense();
                treeopts();
                jumpredir();
                writeproc();
        end
end
```

5.2 Optimization Algorithms

In this section, some algorithms used in the TXOPT program will be explained more in detail.

5.2.1 Constant Expression Folding

Major parts of the optimizer handle the evaluation of constant subexpressions. A technique which can be applied to fold constant expressions in trees already has been explained in section 2.4. However, for space reasons, the Tcode optimizer does not build a tree representing the input program, but it simply holds an *image* of the program in its memory. Therefore, a slightly different approach is used here to build a tree-like representation *on-the-fly*. As already described in section 2.5, stack postfix expressions can be generated from trees by travsersing them in left-right-node order. Tcode is generated exactly this way. Therefore, a tree can be easily reconstructed by stacking operands and building each (binary) subtree from the current operator and the two most recently stacked operands. The subtree is then placed back on the stack.

```
Input           Stack           Description
-----           -----           -----------
a b c * +       c               stack a,b,c
    ^           b
                a

a b c * +       a               build tree from
      ^           *             *,b,c
                 / \
                b   c

a b c * +         +             build tree from
        ^        / \            +,a,b*c
                a   *
                   / \
                  b   c
```

Example 29: Building Trees from Expressions

Example 29 demonstrates how a tree is generated from the postfix expression

```
a b c * +
```

When all input has been read, the complete tree is located at the top of the stack. When the input expression is balanced, the stack will always contain one single entry after the conversion. It is easy to show that the tree resulting from the above example can be converted back into the original expression using the algorithm described in 2.5.

Given a tree representation, an algorithm for evaluating constant subexpressions could be implemented very straight-forward, but in fact, it is not even necessary to actually build the tree. The folding algorithm described in an earlier chapter is based upon the fact that leaves are always evaluated before inner nodes. This effect is achieved by performing a *depth-first* traversal. Since Tcode already *is* the result of a depth-first traversal, it is suitable for being directly processed by the optimizer. No temporary tree representation is required. When having another look at example 29, one can see that the algorithm builds the tree *bottom-up*. Subtrees are built first and finally combined to form more complex aggregates. Since trees are built from subtrees, the leaves of an operation already must have been built before the inner node can be constructed. Therefore, leaves are always visited before operators. This is exactly the effect which is required to make the previously described constant expression folding algorithm work.

Instead of building a tree from subtrees, however, the optimizer immediately attempts to evaluate the operation as soon as it detects an operator. When the two most recently stacked operands are both constant, it applies the operator to the operands, stacks the result, and removes the operator. Thereby, the constant expression gets replaced with its value. If the subexpression cannot be folded, the subexpression is simply passed through by emitting first the operands and then the operator. No change takes place in this case.

```
a B C + -                  stack a,B,C
a (BC+) (NOP) (NOP) -      B,C are constant:
                           evaluate B+C
a (BC+) (NOP) (NOP) -      (BC+) is constant,
                           but 'a' is not.
a (BC+) -                  remove (NOP)s.
```

Example 30: Evaluating a Constant Tcode Expression

In example 30, lower case letters denote variables and capital letters denote constants. (X) denotes the value of X and (NOP) is a null operation. Like in the construction of a tree, the operands *a*, *B*, and *C* are first placed on a stack. When the '+' operation is detected, the algorithm checks whether the two operands on the top of the stack are constant. Since both, *B* and *C* are constant, they get replaced with *(BC+)*, the sum of *B* and *C*. This values replaces the load instruction for *B* while the load instruction for *C* and the 'plus' operator itself both are overwritten with *(NOP)*. The value *(BC+)* is also placed back on the stack. When the '−' operator is found, the stack contains the (constant) value *(BC+)* and the variable *a*. Therefore, the sub expression cannot be folded, and the respective code will not be touched. The operands are removed from the stack, though, and replaced with a special value denoting that the expression on the top of the stack is non-constant.

The routine `foldexpr()` implements this technique. In its main loop, it steps through the entire Tcode program searching for foldable expressions. Whenever it finds a *load instruction* (`LDG`, `LDGV`, `LDL`, `LDLV`, `LDLAB`, `NUM`), it stacks the instruction. Because load instructions typically have an argument, the routine does not stack the opcode and the operand value, but just the location of instruction in the code area. This way, the space for storing the parameters and lots of operand shuffling can be saved.

Binary expression folding is performed exactly in the way described above. When a binary operator is found, the routine `fold_binary()` is activated. This routine first checks whether the two subexpressions located on the top of the stack are both constant and if so, it evaluates the expression and replaces the lefthand operand node with the result. Since the stack contains *pointers to instructions* rather than the instructions themselves, the operand node can be replaced in situ. After relacing the left operand, `fold_binary()` deletes the righthand operand and the evaluated instruction by overwriting them with `NOP` instructions.

If at least one operand of a binary operation is non-constant, `fold_binary()` attempts to perform at least some strength reduction. For this purpose, it applies a specialized routine (like `fold_mul()` or `fold_deref()`) to the arguements of the current operation. For a summary of optimization templates used in these routines, see the appendix.

When no constant subexpression evaluation and no strength reduction could be performed, the procedure `fold_binary()` makes sure that the resulting node is non-constant by replacing the point to the result on the top of the stack with a pointer to an `LDG` instruction (which of course always dentes a variable). For this purpose, it uses the address `CODESIZE`−1. The routine `readcode()` initially fills this cell with an `LDG` opcode.

Handling unary operators is done in a way similar to the treatment of binary operations. The only difference to binary expression folding is that only the opcode has to be deleted in this case:

`A` *op*   `(A`*op*`) (NOP)`

where *op* is any unary operator.

The vector `last` is used by `foldexpr()` to find specific patterns. It works like a simple *first-in-first-out queue*. When the tail of the queue (`last[0]`) contains one of the load instructions {`LDG`, `LDL`} and the current operator is one of the save instruction {`SAVG`, `SAVL`},

```
LDL X NUM N SAVL X
```

or

```
LDG X NUM N SAVG X
```

These patterns can be substituted with cheaper `INCL` and `INCG` instructions. To apply this optimization, if the code is suitable for it, the routine `fold_inc()` is called.

5.2.2 Constant Condition Folding

Constant condition folding is trivial. No tree representation is required at all. In a Tcode program, every conditional branch has the form

```
x BRF L
```

or

```
x BRT L
```

(non-destructive branches will not be discussed here). *L* denotes the destination of the branch and is an operand to the branch instruction and *x* denotes an expression. If *x* is constant, the following branch can be modified by either eliminating it (if the jump would never take place), or by replacing it with an unconditional jump (if the jump would always take place). In either case can *x* be removed. The routine `foldcond()` performs this transformation.

Since two different Tcode jump instructions exist for branches on *true* and branches on *false* conditions, negative conditions can be replaced with their positive counterparts, if the meaning of the branch instruction is also reverted. Negative conditions always end with an `NOT` instruction. This reduction can easily be performed by replacing each pattern of the form

`NOT BRT L` with `BRF L`

and

`NOT BRF L` with `BRT L`

Notice that *this* transformation can be applied to non-destructive conditional branches (employing `NBRT` and `NBRF`), too. The procedure `foldcond()` also applies this optimization.

5.2.3 Jump to Jump Redirection

The redirection of the first jump in a chain of unconditional branches to the destination of the last branch instruction is done by simply following each branch in the program and replacing its operand (its destination label) with the label of the last branch in the chain. To follow chains of branches, of course, the address each label is associated with, must be known. Therefore, the first step of this transformation is the creation of a translation table which holds the address of each instruction following a label. The translation table is accessed using direct indexing. The number used to identify a label is at the same time used to access the table. Entries containing no valid address are filled with −1.

The procedure `jumpredir()` performs the jump to jump redirection. The address table is stored in the global vector `Labels`. Basically, the optimization algorithm is simple.

1) When a `JUMP` instruction has been found at a given address *i*, the address of the instruction is saved in a variable *j*.

2) Then, *i* is replaced with the address which the `JUMP` instruction points to (using the vector `Labels`).

3) As long as the instruction pointed to by *i* is in turn a `JUMP` instruction, the steps **2)** and **3)** will be repeated.

4) The operand of the instruction located at *j* will be replaced with *i*.

This way, the operand of the first jump is changed to the destination of the last one in the chain.

This algorithm works fine, but it might never terminate when attempting to optimize an endless loop, as the following example demonstrates:

```
CLAB 1 JUMP 1
```

Since the destination of `JUMP 1` is `CLAB 1`, step **2)** of the above description will be repeated forever.

There exist – at least – two solutions to this problem. An elegant and simple solution is to check whether *i=j* applies after each step **2)**. If this is the case, the origin has been reached again and therefore, a cycle has been detected. In this case, the redirection can be aborted. (One possible optimization at this point would be to redirect the jump to itself, but one might of course argue about the sense of optimized infinite loops.) This method works fine as long as all jump paths are either open or closed cycles with no paths leading *into* the cycle.

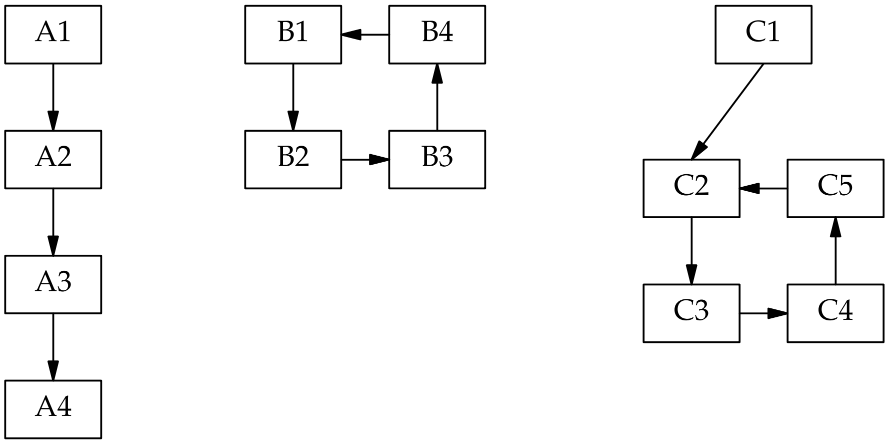


Figure 26: Redirectable and Unredirectable Paths

Figure 26 shows one open path (type **A**), one closed path with no way leading in (type **B),** and one cycle with a path leading into it (type **C**). Type **A** is redirectable using the initially described algorithm. Type **B** can be detected by comparing the origin with each label in the path. No matter where the redirectio starts, since the cycle is closed, the origin will be visited again in this case. Type **C** cannot be detected this way, if the process starts at label **C1**, because the origin is not part of the cycle in this case.

A simple method to handle type **C** paths is to count the number of labels which have been visited and to abort the process after a given maximum number of steps. Another way would be to memorize all visited labels and check the current label against all labels which already have been visited during the currend redirection. The latter method is exhaustive, but requires much more resources than the first one, because at least a flag for each label in the translation table must be held in memory. The first method is easy to implement and yields good results in the most cases. Therefore, it is used in the TXOPT program. The constant `CYCLETRAP` defines the maximim number of labels to follow before assuming a cycle. An in-depth comparison of these two methods would be far beyond the scope of this work.

BTW: I propose that the T3X translator TXTRN can under no circumstances generate type **C** paths. Therefore, an algorithm which just compares the origin which each label would be sufficient. However, no proof to this hypothesis exists so far.

5.2.4 Eliminating Dead Procedures

A procedure is called *dead*, if no references to it exist in a program. Because such a procedure can never be called, it would make sense to removed it entirely from the program. This is particuraly important, because there exists no *loader* for T3X, but only a static interface to a small set of very special extension procedures. Therfore, many standard procedures are included literally (as source code) using the preprocessor. When including sets of standard procedures, of course, it may happen that only few of the included routines are actually used. The *dead procedure elimination* stage of TXOPT removes the unused routines from the program.

The algorithm used is pretty similar to a *garbage collection* algorithm, because they have basically the same purpose: to remove objects from memory which can no longer be addressed by a given program. When talking of garbage collection, the algorithm described in the following paragraohs would be called a *mark and scan* algorithm. It consists of three stages. In the first phase, it *marks* all procedures and during the second one, it unmarks all procedures which can be reached from the main program. Naturally, routines which cannot be reached from within the main program, can never be reached at all when the program executes. In the third step, all procedures which still are marked, will be removed.

The procedure `deadprocs()` performs the elimintation of unused procedures. During the first phase, it scans the entire input program for patterns of the form

```
CLAB L HDR
```

which mark the beginnings of procedures. Each label identifier *L* will be replaced with –*L*. The sign bit of a label id can be savely used as a flag, because the translator TXTRN never generates negative label numbers. Procedures prefixed with negative label identifiers are considered *marked* by the following stages.

The second stage unmarks all procedures which can be referenced by the program. There are two kinds of possible references: procedure calls and computations of procedure addresses. A procedure call of the form

```
P()
```

will always be translated into a Tcode fragment of the form

```
CALL L
```

where *L* denotes the label which marks the procedure *P()*. When the address of a procedure is used in a program, either this address is stored in a table or assigned to a variable. The first case will result in a

```
CREF L
```

instruction and the second one in a

```
LDLAB L
```

instruction, where *L* is a procedure label again.

For each procedure which can be used by a program, one of the following conditions must hold:

1) There exists a chain of `CALL` instructions from the main program to the procedure.

2) There exists a `CREF` or `LDLAB` instruction which references the procedure.

3) There exists a `CREF` or `LDLAB` which references a procedure which in turn references the procedure in question.

3) is of course a special case of **2)**.

The procedure `unmark()` is used to handle case **1)** and part of **2)**. It uses *depth-first* traversal to visit all procedures which can be reached through the procedure whose address (not label) has been passed to it. To unmark the call tree of the entire program, the statement

```
unmark(Labels[0]);
```

is used, since the main program is always located at the label 0.

Figure 27 illustrates a typical procedure call tree with the main program located at the root node. Depth-first traversal works fine in directed anticyclic graphs, but it will run into an infinite loop when traversing cycles (like created by the mutually recrsive procedures *H()* and *K()*). Therefore, `unmark()` only traverses *marked* procedures. When it is applied to an already unmarked procedure, it returns immediately, assuming that that branch of the call tree already has been processed. This way cycles can be processed savely. Before descending deeper into the call tree, the current procedure is unmarked. It is important to do this before visiting the child nodes, *because* the current node (routine) may be a (indirect) descendant of itself.

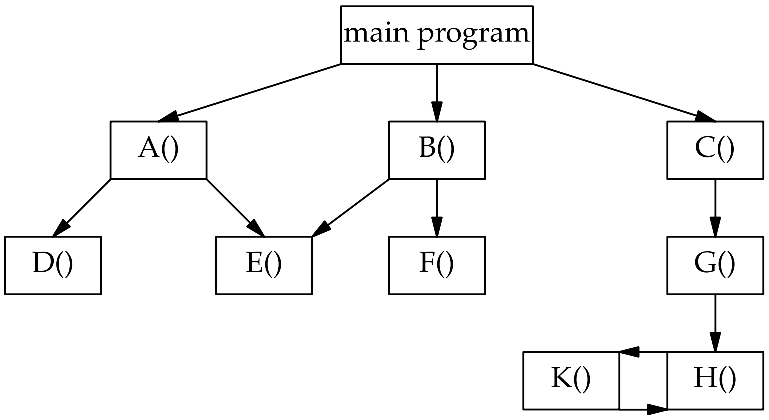


Figure 27: A Procedure Call Tree

To process the body of each procedure, `unmark()` applies itself to the destination of each `CALL` and `LDLAB` instruction in the body of the procedure it has been applied to. The end of the procedure is reached when an `END` instruction is met.

The procedure `unmarkrefs()` processes procedure addresses which are stored in tables. It scans the entire program for `CREF` instructions and applies `unmark()` to each address associated with a label stored in such an instruction.

Finally, `deadprocs()` removes the procedures which have not been visited during the unmark step by overwriting them with `NOP` instructions.

5.2.5 The Order of Optimizations

Some of the optimiztions applied by TXOPT depend on others. For example, the evaluation of constant expressions may lead to constant conditions, but not vice versa. Folding constant conditions can in turn lead to chains of unconditional jumps which will be eliminated in the next step. Between the call of `foldcond()` and `jumpredir()`, the code is condensed to simplify the jump-to-jump elimination algorithm. The dead procedure elimination is performed last. If does not really depend on any of the previously applied transformations, but eventually, it might get replaced with a more general algorithm which removes not only unreferenced procedures but *all* dead code. Such an algorithm could very well make use of a previously performed constant condition folding and jump redirection. The dead procedure elimination routine condenses the input program once more, if some procedures could be removed.

6 Native Code Generation

The T3X compiler whose implementation has been described in the previous chapter is a pure bytecode compiler. A runtime interpreter is required to execute the code generated by it. The resulting code is portable accross a variety platforms, since only the interpreter has to be ported, and the run time of the most programs is acceptable. However, a great improvement of the runtime performance can be achieved by eliminating the overhead caused by the interpretation of the bytecode program by converting the Tcode output of the compiler into native machine code for a specific machine.

Different methods of converting Tcode into native machine code will be discussed in this chapter. The target of the code generator will be a fictous RISC-style CPU with a very basic instruction set. Although, strategies for generating code for single-register machines (*accumulator*-based architectures) will be explained, too. Many real world CPUs have been considered for use in this chapter, but some are not very wide-spread and the widely available ones are too limited for simple register allocation approaches.

6.1 The Target Language

The language described in this section will be the target of the translation schemes discussed in this chapter. The language is a fictous assembly language for a fictous 16-bit machine with a basic instruction set, 16 general purpose registers, a stack pointer, a frame pointer, an instruction pointer, and a byte-addressable memory array containing 2^{16} byte-wide cells. Like in the Tcode machine, cells are stored using little endian byte ordering. There are four addressing modes which may be used in *all* instructions involving memory references. All commands are implicitly applied to full words (16-bit operands). There are separate instructions for loading and storing single bytes. The major purpose of this fictous language is the discussion of code-generation schemes. Hence, its major design goals are *simplicity* and *ease of understanding*. Basic skills in assembly language programming are a big advantage when reading this chapter. Readers who are familiar with *any* kind of assembly language should feel at home soon.

The following assembly language instructions exist:

`DD   n*x`	Define N data words with the value X, N* is defaults to 1.
`DD   (L)`	Define reference to label X.
`PUSH r`	Push R onto the stack.
`POP  r`	Pop R from the stack.
`SWAP r,a`	Swap the values of register and memory.

`NEG`	`r`	R := –R
`NOT`	`r`	R := ˜R
`NOP`		Waste time.
`MUL`	`r,a`	R := R * A
`UDIV`	`r,a`	R := R ./ A
`UMUL`	`r,a`	R := R .* A
`MOD`	`r,a`	R := R MOD A
`SUB`	`r,a`	R := R – A
`ADD`	`r,a`	R := R + A
`AND`	`r,a`	R := R & A
`OR`	`r,a`	R := R | A
`XOR`	`r,a`	R := R ˆ A
`SHL`	`r,a`	R := R << A
`SHR`	`r,a`	R := R >> A
`BEQ`	`r,a,L`	Branch to L, if R=A.
`BNE`	`r,a,L`	Branch to L, if R\=A.
`BGT`	`r,a,L`	Branch to L, if R>A.
`BLT`	`r,a,L`	Branch to L, if R<A.
`BGE`	`r,a,L`	Branch to L, if R>=A.
`BLE`	`r,a,L`	Branch to L, if R<=A.
`BAT`	`r,a,L`	Branch to L, if R.>A.
`BBT`	`r,a,L`	Branch to L, if R.<A.
`BAE`	`r,a,L`	Branch to L, if R.>=A.
`BBE`	`r,a,L`	Branch to L, if R.<=A.
`MOV`	`x,y`	X := Y (x∈{a,r}, y∈({a,r}).
`CALL`	`x`	Push the return address, then branch to X (x∈{r,L}).
`RET`		Return to the caller.
`JMP`	`L`	Jump to label L.
`HALT`		Halt the program.

Each operand *r* denotes a *register* represented by the notation *R0...R15*, *SP*, or *FP* and each operand *a* denotes an *address*. *SP* denotes the stack pointer and *FP* denotes the frame pointer. These registers have the same functions as in the Tcode machine. There are four distinct addressing modes:

```
MOV     R1,#value        ! immediate
MOV     R1,address       ! absolute
MOV     R1,(address)     ! indirect
MOV     R1,FP(address)   ! local
```

Notice that the first operand of a binary instruction is always a register. A *value* may be either a numeric value or the name of a label. An instruction like

```
MOV         R9,#some_label
```

will load *R9* with the address associated with *some_label*. A register/address operation is performed on a register and an address, which may be an absolute address (usually a label tagging a data definition), an indirect address (the value *pointed to* by a data word at an absolute location), a value at a given stack frame position (this is equal to indirect, but *FP* will be added to the specified location), or another register. The operation

```
MOV         R3,(R3)
```

for example, would perform one level of indirection on *R3* (load *R3* with the value pointed to by *R3*).

Each assembly language statement has the general form

```
Label:     Instruction     Destination,Source
```

where the label (and the colon) are optional. There are instructions with zero, one, or two operands. Labels tagging `DD` instructions refer to the data space of the program, all other labels belong to the instruction space.

6.2 The Task of Code Transformation

The generation of native machine code from virtual machine code fills in a gap between translation and optimization. Like a compiler as discussed in the previous chapters, it translates one language into another. On the other hand, the native code generator has to perform only marginal syntax analysis, since its input program is in a rather fixed format and it does not have to do much error checking, since its input is machine generated. Like an optimizer, the code generator attempts to create an output program which is *optimal* under certain aspects, but unlike an optimizer, it does not perform transformations on an existing program, but it creates a program in a different language, thereby attempting to exploit the properties of the target machines for best run time performance of the resulting code.

Of course, the border between translation, optimization, and code generation is fluent. An aggressive optimizer always has to make use of special properties of the target machine and a code generator often implicitly performs some basic optimizations. Frequently, both stages are combined into a single program to achieve the best code quality. In this book, however, a different approach is taken. The code generator is only responsible for translating Tcode programs into assembly language programs, thereby taking advantage of the features of the register-based target machine. The used code generation scheme has a very big impact on the quality of the resulting code – independent from other optimizing transformations.

Since the code generator is a completely separate – and even optional – module in the T3X concept, there is still place for a pure optimizer which can be placed *between* the translator and the code generator. Such an optimizer would perform its transformation on a Tcode program: its input would be Tcode as well as its output. The advantage of this method is that the optimizer can be used independently from the code generator. No matter what code generator is used, the optimizer does not ever has to change. The disadvantage, of course, is the inability to perform machine-specific optimizations. A disadvantage, however, which can be partly compensated in the code generator. A nice side effect of a Tcode optimizer is the ability to generate *optimized Tcode* which makes it useful to improve the performance of interpreted programs, as well.

6.3 VSM Instruction Inlining

A very simple way of creating native code from Tcode is the definition of small code fragments which simulate the semantircs to the respective virtual stack machine (VSM) instructions. By translating each Tcode instruction into the according code fragment, a native machine program with the same meaning as the Tcode source program is built. Since the semantics of the Tcode machine is hidden in the code fragments, no additional interpretation program is required. Because the insertion of small procedure bodies instead of the creation of procedure calls is frequently called *inlining*, I call this method VSM instruction inlining. This is a very simple technique and code generators based upon it can often be retargetted within a few hours. The code generators of the *Small C* compiler, for example, use this method, and so do the generators of the original and compilers. VSM inlining is not very efficient, but it is a great improvement compared to purely interpretative techniques and the code produced by it is sufficient for none-time critical tasks (like compiler bootstrapping).

A typical assembly language fragment for implementing, say, the `SUB` instruction would look like this:

```
! Compute A B -
POP     R1      ! pop B
POP     R0      ! pop A
SUB     R0,R1   ! A - B
PUSH    R0      ! push (A-B)
```

When executed, it removes two entries from the stack, subtracts the S_1 from S_0 (where S_n denotes the n'th objects on the stack, counting from the top), and finally pushes back the result. Code fragments which receive arguments on the stack and place their result on the stack can be combined in any form, just like Tcode instructions. Therefore, Tcode instructions can be translated one by one which is the major reason for the simplicity of this scheme.

VSM inlining has many advantages regarding the implementation:

• It is simple and reliable.

• Only one addressing mode must be implemented per instruction
(except for save and load operations).

• Fragments are easy to translate when retargetting the compiler.

On the other hand, the code generated by this method does not make much use of the register-based architecture of the target machine.

Since expressions make up the major part of most programs, the generation of expression code will be concentrated on in the examples of this chapter. First, the code generated from a more complex expressions by VSM inlining will be examined. The program resulting from the expression

```
a - b * c & ~1
```

can be found in example 31.

```
! Input Program: A B C * - 1 ~ &
!
MOV     R0,a     ! a
PUSH    R0
MOV     R0,b     ! b
PUSH    R0
MOV     R0,c     ! c
PUSH    R0
POP     R1       ! *
POP     R0
MUL     R0,R1
PUSH    R0
POP     R1       ! -
POP     R0
SUB     R0,R1
PUSH    R0
MOV     R0,#1    ! 1
PUSH    R0
POP     R0       ! ~
NOT     R0
PUSH    R0
POP     R1       ! &
POP     R0
AND     R0,R1
PUSH    R0
```

Example 31: VSM Instruction Inlined code for 'a–b*c&~1'

Obviously, the code in the example is pretty inefficient. While saving the overhead of high-level interpretation, it is still very poor quality code compared even to below-average native code compilers. However, there is a simple improvement which can help reduce the size of VSM inlined code significantly.

6.3.1 PUSH/POP Elimination

Since virtually all instructions start with one or more `POP` instructions and end with a `PUSH` instruction, it is an obvious improvement to remove subsequent `PUSH` and `POP` commands which use the same register:

```
PUSH Rx; POP Rx; ⇒ ;
```

Additionally, `PUSH`/`POP` sequences with different registers in the two instructions can be replaced with a – usually cheaper – move operation:

```
PUSH Rx; POP Ry; ⇒ MOV Ry,Rx;
```

For the implementation of this simple optimization, a delay queue as described in the previous chapter is sufficient. Before writing an assembly language instruction, the code generator checks the instruction currently in the delay queue. If the command in the delay queue is `PUSH` and the new one is `POP`, then both can be replaced. If they use the same operand, they can both be deleted. If they use different operands, they are replaced with a `MOV` instruction using the register of the new instruction as destination and the used in the delay queue as source.

```
MOV     R0,a     ! a
PUSH    R0
MOV     R0,b     ! b
PUSH    R0
MOV     R0,c     ! c
MOV     R1,R0    ! *
POP     R0
MUL     R0,R1
MOV     R1,R0    ! \-
POP     R0
SUB     R0,R1
PUSH    R0
MOV     R0,#1    ! 1
NOT     R0       ! ~
MOV     R1,R0    ! &
POP     R0
AND     R0,R1
PUSH    R0
```

Example 32: VSM Inlined code with PUSH/POPs eliminated

As one can see in example 32, PUSH/POP eliminiation has saved five instructions and eight implicit memory references in `PUSH` and `POP` instructions which have been replaced with faster register/register moves. Assuming that each instruction takes one cycle of the processor clock and each memory reference takes another cycle, this means a total of 13 saved cycles. The program generated without PUSH/POP elimination had 23 instructions containing 19 memory references, giving a total running time of 42 cycles. In this context, 13 saved cycles means a speedup of more than 30 percent. While this is not bad for such a simple optimization, it illustrates how inefficient the original code was, on the other hand. To produce code that makes a *real* difference, a completely different approach should be chosen.

6.4 Code Synthesis

A technique which is oriented more towards the actual architecture of the target machine makes use of the fact that Tcode instructions are simplier than real machine instructions. For example, to add two values *a* and *b*, the following stack machine program could be used:

```
LDG     a
LDG     b
ADD
SAVG    a
```

while a programmer of the fictous target machine would use

```
MOV     R0,a
ADD     R0,b
```

```
          MOV      a,R0
```

The `LDG` and `ADD` instructions have been combined in the statement

```
ADD          R0,b
```

A *synthesizing* code generator does the same: it first collects (Tcode) instructions and then combines them to form a more complex native machine instruction, thereby 'synthesizing' a new statement.

The code synthesis algorithm is simple. Basically, it distinguishes instructions which *do something* , and instructions which first must be combined to do something. Instructions which do not have an immediate effect are the *load* instructions: `LDG`, `LDGV`, `LDL`, `LDLV`, `LDLAB`, and `NUM` (which loads an immediate value). When one of these instructions is read, it is not translated, but placed in a queue. When an 'immediate' instruction (all, *but* the load commands) is found, the queue is flushed. How this is done depends on the command found.

When the pending instruction is a binary operator, all members except for the most recently placed one are flushed. When it is a unary operator, all members are flushed. For simplicity, a single-register machine will be used in the following example. The register is assigned a *state* at compile time, which indicates whether the register is loaded (*active*) or *free*. Additionally, a *depth* indicator is used which always holds the current depth of the runtime stack. The members are removed in 'first in first out' (FIFO) order from the queue. For each member, the algorithm performs the following steps:

• If state=free, load the register with the current member and set state=active.

• Otherwise (state=active), push the register, then load it with the current member, and increment *depth*.

After flushing all but the last member from the queue, the actual instruction can be synthesized. In any case, the first operand is in the register at this point and the only instruction in the queue determines the addressing mode of the instruction to generate.

```
Input   Action          Output
------  --------------  -------------
LDG A   queue
LDL -4  queue
ADD     flush LDG A     MOV R0,A
        synthesize ADD  ADD R0,FP(-4)
SAVG A  flush all
        generate SAVG   MOV A,R0
```

Example 33: Code Synthesis for A:=A+B

Example 33 shows the code synthesized from the input statement

```
A:=A+B
```

which compiled to the Tcxode program

```
LDG A LDL -4 ADD SAVG A
```

assuming that *b* is a local variable at stack frame position −4. As one can see in the rightmost column, the output program is equal to the initially shown optimal assembly language version. The reason for this high code quality is the fact that no stack operations had to be used, of course. When synthesizing code from a more complex expression, like the one used in the previous section, however, the limits of the single-register code synthesis become visible.

```
MOV     R0,A
PUSH    R0
MOV     R0,B
MUL     R0,C
POP     R1
SWAP    R0,R1
SUB     R0,R1
PUSH    R0
MOV     R0,#1
NOT     R0
POP     R1
SWAP    R0,R1
AND     R0,R1
```

Example 34: Code Synthesized from 'A B C * – 1 ~ &'

To understand the code shown in example 34, the algorithm used for synthesizing an instruction must be explained. The first operand of a binary instruction or the only operand of a unary instruction is always in a the register *R0*, when the synthesizer is called. Creating code for unary operators is trivial, since they always affect *R0*. To generate a binary operation, the respective assembly language instruction is combined with the addressing indicated by the instruction currently in the queue after the following scheme:

```
LDG A   =>      MOV     R0,A
LDL n   =>      MOV     R0,FP(n)
LDGV A  =>      MOV     R0,#A
LDLV n  =>      MOV     R0,#n
                ADD     R0,FP
LDLAB L =>      MOV     R0,#L
NUM n   =>      MOV     R0,#n
```

When the instruction `LDL -8` is in the queue, for example, and a `MUL` instruction is read, the synthesizer would generate

```
MUL     R0,FP(-8)
```

After generating the instruction, the last member is removed from the queue.

Of course, the synthesizer also has to check for pushed operands. Operands are removed from the runtime stack, if an instruction has to be generated while the queue is already empty. In this case, the synthesizer first produces the sequence

```
POP     R1
SWAP    R0,R1
```

to adjust for the following operations. Otherwise, the operands would be in reverse order, since the top of the stack always holds the *first* operand of a binary

```
• Set state to free and depth to 0.
• For each Tcode instruction I do ...
    • If I is a load instruction, queue it.
    • Otherwise,
        • For each member M, except for the last one do ...
            • If state=free, synthesize MOV with the mode of M and
            set state=active.
            • Otherwise, push R0, synthesize MOV with the mode
            of M, and increment depth.
        • If I is not a binary operation, repeat the above loop once
        more (thereby flushing the entire queue).
        • If I is unary, apply the operation on R0.
        • Otherwise, if I is binary, ...
            • If the queue is empty,
            generate a POP/SWAP sequence,
            apply the operation on R0,R1,
            and decrement depth.
            • Otherwise, synthesize the instruction with the
            mode of the command still in the queue, and then
            empty the queue.
        • Otherwise, compile whatever is appropriate for the
        pending command. (This part is not covered here.)
• Flush the queue.
• At this point, you may assert state=active (result in R0)
and depth=0.
```

Example 35: The Single-Register Code Synthesizing Algorithm

instruction. The following instruction is always performed on R1,R2. The complete synthesizing algorithm is outlined in example 35.

The assembly language program in example 34 consists of 13 statements containing 8 memory references, giving a total of 21 cycles. The code generated by this scheme is pretty good for single register machines (R1 can be easily simulated in memory), but is turns out that it is a special case of a more general scheme which makes much better use of the 16 registers of the fictous target machine.

6.5 Cyclic Register Allocation

Making use of available general purpose registers has a great impact on the quality of the generated code. In the previous section a scheme for generating code for single-register machines has been explained. With a simple modification, this scheme can be extended to use all registers available on the target machine. Note however, that the registers must be *real general purpose registers*. This means that each operation must be applicable to each register or pair of registers. Even some modern and very popular CPUs restrict the use of registers in certain ways, making register allocating code generation a rather hard task.

```
Index points to the next free register.
Depth indicates the depth of the runtime stack.
NREG is the index of the last register.
• For each queue member M, ...
    • If depth \= 0 or index > NREG, ...
        • If Index > NREG, set index to 0.
        • Push the register, index currently points to.
        • Increment depth.
    • Load M into the register, index currently points to.
    • Increment index.
```

Example 36: Cyclic Register Allocation

The difference between the single-register approach and the multi-register version of the synthesizing algorithm is simple. Instead of pushing the content of the only register which is used as an *accumulator*, it keeps the names of all registers in a list, and takes a new name from the list as it is required. On might think of the this register array as a *stack* from which a register is taken when a new value has to be activated (loaded into a register) and to which a register is returned as soon as its value becomes free (is no longer used). Aggressive optimizers use complex algorithms to compute the liveness of values stored in registers. The multi-register code synthesizer, simply uses the register to 'cache' the top of the runtime stack. The algorithm is a very simple extension to the method described in the previous section. The first difference regards flushing the queue. Instead of pushing values on the runtime stack, the values are placed in registers. The allocation algorithm is outlined in example 36 .

When the list of available registers is exhausted, the algorithm resets the pointer to the first entry, thereby cycling through the list again and again. Although, with 16 registers available like in our target machine, chances that the end of the list will ever be reached are pretty low. Like the single-register synthesizing code generator, the allocation algorithm saves the values of reused registers on the runtime stack. In fact, the single-register version of this algorithm can be considered a multi-register algorithm with *NREG*=1. Hence, it always makes sense to implement an algorithm which is able to maintain multiple registers, since it can be scaled down even to handle a single register (using a temporary storage location which may be added to the list as second register).

```
I denotes the currently processed non-load instruction.
Reg[n] denotes the register pointed to by n.
• If I is unary, perform I on Reg[index–1]
(the last active register).
• Otherwise, if I is binary, ...
    • Perform I on Reg[index–1] and Reg[index–2].
    Leave the result in Reg[index–2].
    • If depth is non-zero, ...
        • Pop Reg[index–1].
        • Decrement depth.
    • Decrement index.
• Otherwise, compile whatever is appropriate for I
(this part is not shown here).
```

Example 37: Multi-Register Instruction Synthesis

The synthesizing phase of the algorithm is a generalization of the single-register scheme, too. Instead of popping arguments from the stack, it combines instructions with previously allocated registers. Only if the registers get exhausted, it falls back to using the runtime stack – as already described in the allocation algorithm. The synthesizing scheme can be found in example 37.

Example 38 illutrates the application of multi-register instruction synthesis on the example expression introduced at the beginning of this chapter

```
A - B * C & ~ 1
```

Since registers are used to store temporary results and delayed arguments, now, the number of memory references could be reduced to a minumum of 4. There is no way to reduce this count any farther, since 4 in-memory operands are involved in the expression. Since no `PUSH`, `POP`, and `SWAP` instructions are required, the size of the code could be reduced to seven instructions, making a total of 11 cycles.

6.5.1 Working Around Specialized Registers

Code synthesis only leads to good results, if all registers can be combined with any instruction. As initially mentioned, there are some popular machines which assign specific tasks to specialized registers. The i8086 is one of most problematic CPUs, since virtually *all* registers are specialized in some way, making automatic code generation quite troublesome. For simplicity, the instructions used in the previous sections will be used in this subsection, too. Instead of the registers *R0...R15*, however, the i8086 has eight registers named

```
Input   Action          Output
-----   ------------    -------------
LDG A   queue
LDG B   queue
LDG C   queue
MUL     flush A,B       MOV     R0,A
                        MOV     R1,B
        generate MUL    MUL     R1,C
SUB     flush
        generate SUB    SUB     R0,R1
        release R1
NUM 1   queue
NOT     flush all       MOV     R1,#1
        generate NOT    NOT     R1
AND     flush
        generate AND    AND     R0,R1
        release R1
```

Example 38: Code Synthesis with Register Allocation

AX, *BX*, *CX*, *DX*, *SI*, *DI*, *BP*, and *SP*. *SP* is the stack pointer and *BP* is the base pointer. The remaining registers are kind of multi-purpose registers, but with a whole bunch of restrictions imposed on their use. Table 32 summarizes what instructions may be combined with which registers.

	MUL	**DIV;MOD**	**SHL;SHR**	**MOV R,(R)**
AX	1	1		X
BX				
CX			2	X
DX	i	i		X
SI				
DI				

Table 32: Register/Instruction Combinations on the i8086

A digit means that the denoted register (1st or 2nd) *must* be the respective operand. An *i* means that the register will be implicitly used by the operation, thereby destroying its content. An *X* means that the operation cannot be peformed on that register. The i8086 can perform multiply, divide, and modulo operations only on the *AX* register (the divide operation leaves the modulo in *DX*). Shift operations require the shift count in *CX*. A particular nuisance is that no indirection [MOV R,(R)] can be peformed through the accumulator *AX*. If the indirections could be performed on *AX*, there would be three registers left for caching: *BX*, *SI*, and *DI*. Unfortunately, *none* of the specialized registers can be used to dereference pointers, so one of the free registers has to be reserved for this task, leaving *two*. To complicate things additionally, the i8086 knows faster opcodes for operations using the accumulator. Therefore, the use of, say, *SI* and *DI* for caching prevents the use of these optimized instructions.

There are different ways to circumvent these limitations. The most obvious way would be to use only the registers left, as described above. Of course, two registers are hardly worth the implementation of a allocation algorithm which does not perform a exhaustive liveness analysis. On the other hand, one could simply use *all* of the available registers for caching and save them as they are needed for special purposes. When a shift operations had to be peformed, for example, the following code could be generated for an expression containing the operation `A<<B` while *CX* is already active.

```
MOV     BX,A
PUSH    CX
MOV     CX,B
SHL     BX,CL
POP     CX
```

Since the operations involving special registers have a different complexity, the register list could be *sorted* to allocate frequently-used special register only in very complex expressions. Additionally, the accumulator *AX* should always be allocated first, since this would allow the generation of the faster accumulator operations for expressions like

```
A+B+C−D
```

which would result in the code

```
MOV     AX,A    ; A B + C + D −
ADD     AX,B
ADD     AX,C
SUB     AX,D
```

which makes pretty good use of the *AX* register. Even chains of multiply and divide operations would result in fairly compact code, since these operations *must* be performed on the accumulator. Other operations requiring special registers include *shift operations* and *indirections*. Either *BX*, *SI*, or *DI* is required for indirection, *AX* and *DX* are needed for multiplication and division, and *CX* is needed in shifts. If *BX* would be used for dereferencing pointers, *SI* and *DI* should be allocated immediately after the accumulator. `MUL` and `DIV` are pretty costly instructions on the i8086, anyway. Therefore, *DX* will be activated next. Shifts and indirections are both not very costly, but indirection is usually much more frequently used. Hence, *BX* should be allocated last, or possibly even not at all. These observations lead to the following allocation list:

`AX, SI, DI, DX, CX,` (optionally) `BX`

which gives a total of six free registers, where frequently required special registers are used as little as possible. Even the use ud the *CX* register for caching could be questioned, since saving CX would greatly increase the cost of (otherwise cheap) shift operations when the *CX* register is already active.

Example 39 shows the code generated for the expression used in the previous sections. Four additional instructions have to be generated for the `MUL` instruction: two to save and restore the previous content of the *AX* register, one to load the accumulator with the first factor of the multiplication, and another one to move the result from the accumulator into the destination register. Note that these instructions only had to be emitted, because the `MUL` operations had to be performed on a register other than *AX*. An expression like

```
A*B
```

```
Input   Action          Output
-----   ------------    -------------
LDG A   queue
LDG B   queue
LDG C   queue
MUL     flush A,B       MOV     AX,A
                        MOV     SI,B
        generate MUL    PUSH    AX
                        MOV     AX,SI
                        MUL     C
                        MOV     SI,AX
                        POP     AX
SUB     flush
        generate SUB    SUB     AX,SI
        release SI
NUM 1   queue
NOT     flush all       MOV     SI,#1
        generate NOT    NOT     SI
AND     flush
        generate AND    AND     AX,SI
        release SI
```

Example 39: Register Allocation on the i8086

would result in code like

```
        MOV     AX,A          MUL     B
```

without any additional register-shuffling.

BTW: The successors of the i8086, namely the i386 an later processors, impose much less restrictions on the use of general purpose registers. Indirection may be performed through *AX*, the `MUL` and `DIV` instructions support many additional modes, and constant shift counts may be coded as *immediate* (built-in) values. Variable shift counts, however, still must be placed in *CL* (the least significant eight bits of *CX*).

6.6 Procedure Calls

Since each procedure creates its own context, it is free to use all available registers – unless a global register allocation scheme is used, but this is not the case here. Since procedures may reuse registers already active in the expression performing a procedure call, the active registers must be saved before entering the subprogram and they must be restored after leaving it. The most obvious place to save register values it the runtime stack. As a side effect, this method places the procedure arguments on the stack where they would have to placed anyway, since that is the way, T3X passes parameters.

Since the register allocation scheme *cycles* through the list of available registers, register names *above* the index pointing to the next free register already may be allocated. If they are, the values of reused registers are already on the stack and therefore, the depth indicator will have a non-zero value. So, the first step of generating a procedure call is to save all registers between *Rx* and *NReg*, if *Depth* is non-zero. Then, all active registers (between 0 and *Rx*) are pushed onto the stack, too. At this point, all register values have been saved on the runtime stack and the top *n* values will be used as arguments to an *n*-ary procedure. At this point, the procedure call itself may be generated.

The code used to perform the procedure call depends on the type of the call. Tcode provides three different call instructions: `CALL`, `CALR`, and `EXEC`. The destinations of `CALL` instructions are ordinary labels. Usually, they can be translated directly:

```
CALL n  ⇒  CALL L_n
```

where *L_n* is a label of a form which will be accepted by the target assembler. `CALR` performs an indirect call using the value at the top of the stack as a vector pointing to the called procedure. Instead of popping that vector off the stack, the above routine to save the active registers can be modified, of course, to leave the most recently allocated register active. This minor modification allows to call the indirect procedure directly through the register. Notice that the indirect call consumes the register, leaving it inactive.

The last possible method is the execution of an external procedure using `EXEC`. The code produced for

```
EXEC n
```

is much like an indirect procedure call, but the vector is not already in a register, but it is part of the instruction itself. Also, it must be decoded first, since the address of the external procedure is not known to the T3X compiler. T3X uses a common storage location to store addresses of external procedures. (This is comparable to the BCPL *global vector*). This vector will be initialized with the external procedure addresses by the startup code. Performing a call of the procedure in *slot* **n** requires to load the *n*'th member of the global vector into a register, first:

```
        MOV     R0,(_GV_+n*BPW)
        CALL    R0
```

(*_GV_* is the base address of the global vector and *BPW* is the number of bytes per word on the target machine). The register does not have to be allocated before its use, because it will be released immediately in the following call.

Procedures always return their values in a specific register (*RR* in the Tcode machine). Altough any register could be used for this task, it is common to use *R0*. The return register will be protected by setting *Rx* to one after processing each `CALL`, `CALR`, or `EXEC` command.

Each procedure call is by convention followed by a `CLEAN` instruction in Tcode programs. This instruction advises the code generator to generate some cleanup code which removes the procedure arguments from the stack and – in case of a register allocating method – restores the previous state of the registers unless their values have been consumed by the procedure call.

The argument of `CLEAN` specifies the number of arguments to remove from the stack. This is usually achieved by adjusting the stack pointer of the target machine:

```
ADD     SP,n*BPW
```

where *n* is the number of arguments. No deallocation code has to be generated for `CLEAN` commands with *n*=0.

Besides adjusting the stack pointer, `CLEAN` restores the state of the register allocator (this is the reason why `CLEAN 0` commands cannot be omitted). First, the number of removed arguments is subtracted from *Depth*. Then, a register to hold the return value of the prior procedure call is selected. This is simply the first free register *after* restoring the allocator state. Since restoring the state will possibly destroy the return value in *R0*, however, the return register index must be computed *before* actually reactivating any registers. At this point, *Depth* holds the number of register values to restore, but since register names are allocated in a cycle, *Depth* may be larger than *NReg*. Therefore, the formula

$(Depth+1) \bmod NReg - 1$

is used to compute the index of the first free register, the so-called *Top* register. If a negative result occurs, *NReg* should be added to *Top*. The value in *Top* can be used to save the return value:

```
MOV     R[Top],R0
```

where *R[Top]* is the register pointed to by *Top*. Of course, this instruction only has to be generated, if $Top \neq 0$.

In the following step, all active registers (0...*Top*−*1*) will be reactivated by popping their values off the stack. The free register index *Rx* is set to *Top*+*1* to protect the returned value and *Top* is subtracted from *Depth*. If *Depth* is still non-zero thereafter, registers had been saved on the stack before the procedure call. In this case, the registers above and including *Rx* (*Rx*...*NReg*−1) have to be reactivated, too.

Example 40 illustrates the procedure call code generated in the statement

```
x[0] := 1 | 2 + 3 * x[P()];
```

whose operators are ordererd by ascending precedence so that all operands will be stacked until the procedure call returns. When the `CALL` instruction is reached, *R0* is allocated and holds the address of *x[0]* and four operands are queued. In the following *flush* step, all registers are allocated and *R0* is saved on the stack and reused thereafter (*NReg* has been artifically set to 4 to achieve this effect). In the *save* step, the registers are pushed in the order of their allocation – the registers above *Rx* first. After the call, the stack is adjusted. Since the argument of `CLEAN` is zero, no code will be emitted. Then, the *Top* register is computed. *Depth*=5 and *NReg*=4, so

$Top = (5+1) \bmod 4 - 1 = 1$

Hence, the return value which has been returned in *R0* will be transferred to *R1*.

```
! Input: x[0] := 1 | 2 + 3 * x[P()];
! NReg = 4.
Input   Action          Output          Depth
------- --------------  -------------   -
LDL 1   Queue                           0
NUM 0   Queue                           0
DEREF   Flush all       MOV  R0,FP(2)   0
                        MOV  R1,#0      0
        Generate DEREF  SHL  R2,#1      0
                        ADD  R1,R0      0
NUM 1   Queue                           0
NUM 2   Queue                           0
NUM 3   Queue                           0
LDL 1   Queue                           0
CALL 2  Flush all       MOV  R1,#1      0
                        MOV  R2,#2      0
                        MOV  R3,#3      0
                        PUSH R0         1
                        MOV  R0,FP(2)   1
        Save registers  PUSH R1         2
                        PUSH R2         3
                        PUSH R3         4
                        PUSH R0         5
        Generate CALL   CALL A_2        5
CLEAN 0 Adjust stack                    5
        Save RR         MOV  R1,R0      5
        Restore state   POP  R0         4
                        POP  R3         3
                        POP  R2         2
        Protect R1                      2
```

Example 40: Procedure call generation

Consequently, *R1* will not be reactivated when restoring the state of the register allocator. Since it is the register which has been saved first, all other active registers can be reactivated by simply popping their values in reverse order. (BTW: the algorithm always selects least recently allocated register.) Finally, the return register (and thereby all other active registers) are protected by setting *Rx* to *Top*+1 and adjusting *Depth*.

6.7 Relational Operations and Branches

The relational Tcode operators `EQU`, `NEQU`, `LESS`, `GRTR`, `LTEQ`, and `GTEQ` express the relation of their two operands in a truth value. A common way of automatically coding such operations is the *jump around jump* method. The Tcode program

```
LDG a LDG b EQU
```

for example, would translate to the following assembly language program:

```
        MOV     R0,a
        BEQ     R0,b,T1
        MOV     R0,#0
        JMP     T2
T1:     MOV     R0,#-1
T2:     ...
```

If *a=b,* a jump to *T1* is performed where *R0* is loaded with a true value. Otherwise, *R0* is loaded with *false* and a jump around the true branch is made. **BTW**: Similar code is generated for logical NOT operations (`LNOT`), but the conditional branch is

```
   BEQ     Rx,#0,Ty
```

where *Rx* is the register to test and *Ty* is the label tagging the true branch.

This method works in any case, but it leads to some inefficient code when it is used in actual conditional branches which are expressed using `BRT` (BRanch on True) and `BRF` (BRanch on False) in Tcode programs. Imagine the following function to compute the absolute value of its parameter:

```
a(x) IE (x<0) RETURN -x; ELSE RETURN x;
```

The jump around the true branch of the `IE` statement would be translated as follows:

```
LDL -2 NUM 0 LESS BRF n
```

Since `BRF` branches if the top of the stack (the most recently allocated register) is zero, the following code would work:

```
   BEQ     Rx,#0,Ly
```

where *Ly* is the address of the false branch. When this statement is combined with the previously shown jump around jump scheme, this leads to the code in example 41.

```
Input           Output
------          ----------------------
LDL -2
NUM 0
LESS                    MOV  R0,FP(-2)
                        BEQ  R0,#0,T1
                        MOV  R0,#0
                        JMP  T2
                T1:     MOV  R0,#-1
                T2:
BRF L9                  BEQ  R0,#0,L9
```

Example 41: Conditional execution using a Jump Around Jump

Obviously, the entire jump around jump part could be saved if the first branch would be changed to

```
   BNE     R0,#0,L9
```

In fact, the jump around jump method has to be used only to translate statements like

```
a := a = 0;
```

where the truth value is not immediately consumed by a subsequent branch instruction.

Generating better code for conditional branches is trivial. A simple lookahead suffices to decide whether the resulting truth value will be consumed. If the lookahead token is either a `BRT` or a `BRF` instruction, the flag will be swallowed by that instruction and there is no need to generate an actual truth value. If the token is neither of them, on the other hand, it has to be placed back in the input stream before the jump around jump code is emitted.

If a branch instruction follows the relational operator, the relation and the branch can be combined to form a single branch instruction using scheme outlined in table 33.

Relation	**Branch**	**Generated Code**
`EQU`	`BRT`	`BEQ Rx,y,Lz`
`NEQU`	`BRT`	`BNE Rx,y,Lz`
`LESS`	`BRT`	`BLT Rx,y,Lz`
`GRTR`	`BRT`	`BGT Rx,y,Lz`
`LTEQ`	`BRT`	`BLE Rx,y,Lz`
`GTEQ`	`BRT`	`BGE Rx,y,Lz`
`EQU`	`BRF`	`BNE Rx,y,Lz`
`NEQU`	`BRF`	`BEQ Rx,y,Lz`
`LESS`	`BRF`	`BGE Rx,y,Lz`
`GRTR`	`BRF`	`BLE Rx,y,Lz`
`LTEQ`	`BRF`	`BGT Rx,y,Lz`
`GTEQ`	`BRF`	`BLT Rx,y,Lz`

Table 33: Branch Synthesis

Unlike the branches generated for `BRT` conditions, the relation has to be inverted for `BRF` branches, because the branch takes place, only if the preceding condition did *not* hold.

Note: The scheme described in this section cannot be applied to the non-destructive branch commands `NBRT` and `NBRF`, since these instructions do not consume their truth values.

7 Appendices

7.1 The T3X Grammar

```
Program:
        DeclList CompoundStmt
        ;

DeclList:
        Declaration
        | Declaration DeclList
        ;

Declaration:
        'VAR' VarDeclList ';'
        | 'CONST' ConstDeclList ';'
        | 'DECL' ProtoDeclList ';'
        | 'STRUCT' Symbol '=' StructMemList ';'
        | ProcDecl
        | ClassDecl
        | 'PUBLIC' ClassDecl
        | 'OBJECT' ObjDeclList ';'
        | 'MODULE' Symbol '(' ModList ')' ';'
        | 'MODULE' Symbol '(' ')' ';'
        | 'INTERFACE' IfaceDeclList ';'
        ;

VarDeclList:
        VarDecl
        | VarDeclList ',' VarDecl
        ;

VarDecl:
        Symbol
        | Symbol '[' ConstValue ']'
        | Symbol '::' ConstValue
        ;

ConstDeclList:
        Symbol '=' ConstValue
        | Symbol '=' ConstValue ',' ConstDeclList
        ;

ModList:
        Symbol
        | Symbol ',' ModList
        ;

StructMemList:
        Symbol
        | Symbol ',' StructMemList
        ;

ClassDecl:
        'CLASS' Symbol '(' ModList ')' InstDeclList 'END'
        | 'CLASS' Symbol '(' ')' InstDeclList 'END'
        ;

InstDeclList:
        InstDecl
        | InstDecl InstDeclList
        ;

InstDecl:
        'VAR' VarDeclList ';'
        | 'CONST' ConstDeclList ';'
        | 'DECL' ProtoDeclList ';'
        | 'INTERFACE' IfaceDeclList ';'
        | 'STRUCT' Symbol '=' StructMemList ';'
        | ProcDecl
        | 'OBJECT' ObjDeclList ';'
        | 'PUBLIC' ProcDecl
        | 'PUBLIC' 'CONST' ConstDeclList ';'
        | 'PUBLIC' 'STRUCT' Symbol '=' StructMemList ';'
        ;

ObjDeclList:
        Symbol '[' Symbol ']'
        | Symbol '[' Symbol ']' ',' ObjDeclList
        ;

ProtoDeclList:
        ProtoDecl
        | ProtoDecl ',' ProtoDeclList
        ;

ProtoDecl:
        Symbol '(' ConstValue ')'
        ;

ProcDecl:
        Symbol '(' ArgumentList ')' Statement
        | Symbol '(' ')' Statement
        ;

ArgumentList:
        Symbol
        | Symbol ',' ArgumentList
        ;

IfaceDeclList:
        IfaceDecl
        | IfaceDecl ',' IfaceDeclList
        ;

IfaceDecl:
        ProtoDecl
        | ProtoDecl '=' ConstValue
        ;

Statement:
        CompoundStmt
        | Symbol ':=' Expression ';'
        | Symbol Subscripts ':=' Expression ';'
        | ProcedureCall
        | 'CALL' ProcedureCall ';'
        | Symbol '.' ProcedureCall ';'
        | 'SEND' '(' Symbol ',' Symbol ',' ProcedureCall ')' ';'
        | 'IF' '(' Expression ')' Statement
        | 'IE' '(' Expression ')' Statement 'ELSE' Statement
        | 'WHILE' '(' Expression ')' Statement
        | 'FOR' '(' Symbol '=' Expression ',' Expression ')' Statement
        | 'FOR' '(' Symbol '=' Expression ',' Expression ',' ConstValue ')'
                Statement
        | 'LEAVE' ';'
        | 'LOOP' ';'
        | 'RETURN' ';'
        | 'RETURN' Expression ';'
        | 'HALT' ';'
        | 'HALT' ConstValue ';'
        | ';'
        ;

CompoundStmt:
        'DO' 'END'
        | 'DO' LocalDeclList 'END'
        | 'DO' StatementList 'END'
        | 'DO' LocalDeclList StatementList 'END'
        ;

LocalDeclList:
        LocalDecl
        | LocalDecl LocalDeclList
        ;

LocalDecl:
        'VAR' VarDeclList ';'
        | 'CONST' ConstDeclList ';'
        | 'STRUCT' Symbol '=' StructMemList ';'
        | 'OBJECT' ObjDeclList ';'
        ;

StatementList:
        Statement
        | Statement StatementList
        ;

ExprList:
        Expression
        | Expression ',' ExprList
        ;

Expression:
        Disjunction
        | Disjunction '->' Expression ':' Expression
        ;

Disjunction:
        Conjunction
        | Disjunction '\/' Conjunction
        ;

Conjunction:
        Equation
        | Conjunction '/\' Equation
        ;

Equation:
        Relation
        | Equation '=' Relation
        | Equation '\=' Relation
        ;

Relation:
        BitOperation
        | Relation '<' BitOperation
        | Relation '>' BitOperation
        | Relation '<=' BitOperation
        | Relation '>=' BitOperation
        | Relation '.<' BitOperation
        | Relation '.>' BitOperation
        | Relation '.<=' BitOperation
        | Relation '.>=' BitOperation
        ;

BitOperation:
        Sum
        | BitOperation '&' Sum
        | BitOperation '|' Sum
        | BitOperation '^' Sum
        | BitOperation '<<' Sum
        | BitOperation '>>' Sum
        ;

Sum:
        Term
        | Sum '+' Term
        | Sum '-' Term
        ;

Term:
        Factor
        | Term '*' Factor
        | Term '/' Factor
        | Term '.*' Factor
        | Term './' Factor
        | Term 'MOD' Factor
        ;

Factor:
        Number
        | String
        | Table
        | 'PACKED' String
        | 'PACKED' PackedTable
        | Symbol
        | Symbol Subscripts
        | Symbol '.' Symbol
        | ProcedureCall
        | 'CALL' ProcedureCall
        | Symbol '.' ProcedureCall
        | 'SEND' '(' Symbol ',' Symbol ',' ProcedureCall ')'
        | '@' Symbol
        | '-' Factor
        | '\' Factor
        | '~' Factor
        | '(' Expression ')'
        ;

Subscripts:
        '[' Expression ']'
        | '::' Factor
        | '[' Expression ']' Subscripts
        ;

Table:
        '[' MemberList ']'
        ;

MemberList:
        TableMember
        | TableMember ',' MemberList
        ;

TableMember:
        ConstValue
        | String
        | Table
        | 'PACKED' String
        | 'PACKED' PackedTable
        | '@' Symbol
        | '(' Expression ')'
        ;

PackedTable:
        '[' PackedTableMembers ']'
        ;

PackedTableMembers:
        PackedTableMember
        | PackedTableMember ',' PackedTableMembers
        ;

PackedTableMember:
        Symbol
        | Number
        ;

ProcedureCall:
        Symbol '(' ')'
        | Symbol '(' ExprList ')'
        ;

ConstValue:
        SimpleConst
        | ConstValue '+' SimpleConst
        | ConstValue '*' SimpleConst
        | ConstValue '|' SimpleConst
        ;

SimpleConst:
        Symbol
        | Number
        | '-' SimpleConst
        | '~' SimpleConst
        ;

Number:
        DecimalNumber
        | '0x' # HexNumber
        | '0X' # HexNumber
        | '%' # DecimalNumber
        | '%' # '0x' # HexNumber
        | '%' # '0X' # HexNumber
        | '''' # AnyChar # ''''
        ;

DecimalNumber:
        DecimalDigit
        | DecimalDigit # DecimalNumber
        ;

HexNumber:
        HexDigit
        | HexDigit # HexNumber
        ;

Symbol:
        Letter
        | Letter # SymbolChars
        ;

SymbolChars:
        Letter
        | DecimalDigit
        | Letter # SymbolChars
        | DecimalDigit # SymbolChars
        ;

Letter:
        '_'
        |'A'|'B'|...|'Y'|'Z'
        |'a'|'b'|...|'y'|'z'
        ;

HexDigit:
        DecimalDigit
        |'A'|'B'|'C'|'D'|'E'|'F'
        |'a'|'b'|'c'|'d'|'e'|'f'
        ;

DecimalDigit:
        '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
        ;

! a # b denotes that no space characters are allowed
! between a and b.
String:
        '"' # StringChars # '"'
        ;

StringChars:
        AnyChar
        | AnyChar # StringChars
        ;
```

```
! <character> denotes any character available on
! a given host system.
AnyChar:
        <character>
        | '\' # <character>
        ;
```

7.2 Summaries

7.2.1 Statements

Syntax	**Description**
`VAR v1, v2[c], ... ;`	Declare variables and vectors.
`CONST c1=c, ... ;`	Declare constants.
`STRUCT n=m0,...mN;`	Declare structure N with members M0...MN.
`p(...) s`	Define procedure *p* with statement s.
`DECL p(n), ... ;`	Declare (but do not define) procdures.
`IF (x) s`	Execute s, if x is true.
`IE (x) s1 ELSE s2`	Execute s1, if x is true and s2, if x is false.
`WHILE (x) s`	Execute s, while x is true.
`FOR (v=x1, x2) s`	Iterate v from x1 to x2–1 and execute s for each value of v.
`FOR (v=x1, x2, c) s`	Iterate v from x1 to x2–c and execute s for each value of v. Increment v by c after execution of s.
`LEAVE;`	Leave the innermost FOR/WHILE loop.
`LOOP;`	Reenter the innermost FOR/WHILE loop. In FOR loops, go to the increment part.
`p(...)`	Call the procedure *P*.
`CALL x(...)`	Call the procedure whose address is stored in *X*.
`RETURN x;`	Return x to to the calling procedure.
`HALT;`	Halt program execution.
`l := x;`	Assign x to l. L may be a variable or a variable followed by some subscripts.
`DO ... END`	Block statement with optional local declarations.

Table 34: Statements

7.2.2 Operators

Oper	**Assoc**	**Prec**	**Description**
`(X)`	–,–	0	Expression grouping
`X(...)`	L,V	0	Procedure call
`CALL X(...)`	L,V	0	Indirect procedure call
`X[Y]`	L,B	0	Subscript
`X::Y`	R,B	0	Byte operator
`~X`	R,U	1	Bitwise NOT (Inversion)
`-X`	R,U	1	Negation
`\X`	R,U	1	Logical NOT
`@X`	R,U	1	Address of X
`X*Y`	L,B	2	Signed multiplication †
`X/Y`	L,B	2	Signed division †
`X MOD Y`	L,B	2	Modulo (division remainder)
`X+Y`	L,B	3	Addition
`X-Y`	L,B	3	Subtraction
`X&Y`	L,B	4	Bitwise AND
`X|Y`	L,B	4	Bitwise OR
`X^Y`	L,B	4	Bitwise XOR
`X<<Y`	L,B	4	Bitwise left shift
`X>>Y`	L,B	4	Bitwise unsigned right shift
`X<Y`	L,B	5	Less than (giving %1 or 0) †
`X>Y`	L,B	5	Greater than (") †
`X<=Y`	L,B	5	Less than or equal to (") †
`X>=Y`	L,B	5	Greater than or equal to (") †
`X=Y`	L,B	6	Equal to (giving %1 or 0)
`X\=Y`	L,B	6	Not equal to (")
`X/\Y`	L,B	7	Short circuit logical AND
`X\/Y`	L,B	8	Short circuit logical OR
`X->Y:Z`	L,T	9	Conditional expression

Table 35: Operators

Associativity may be **L**eft, **R**right, or none (–).

Smaller precedence values denote stronger bindings.

† These operators also exist as *unsigned* operators. To turn an operator into an unsigned operator, prefix it with a dot: `.*` `./` `.<` `.>` `.<=` `.>=`

7.2.3 Runtime Support Procedures

Procedure	**Description**
`pack(V,S)`	Pack T-string V into ASCIZ string S.
`unpack(S,V)`	Unpack ASCIZ string S into T-vector V.
`aton(S)`	Parse numeric string S and returns its value.
`ntoa(N,W)`	Create a readable representation of the value N. Pad to W characters, if necessary.
`open(F,M)`	Open file F with Mode M. M=0 = read-only, M=1 = create/write-only, M=2 = read/write, M=3 = append.
`close(D)`	Close file descriptor D.
`select(P, D)`	Select new input (P=0) or output port (P\=0). The new port will be D.
`erase(F)`	Erase the file F.
`reads(S,N)`	Read a string with up to N characters from the current input port. Unpack input into S.
`writes(S)`	Pack S into internal buffer and write the buffer to the current output port.
`newline()`	Write a newline sequence to the current output port.

Table 36: Runtime Support Procedures

Procedure	**Slot**	**Description**
`readpacked(D,V,L)`	11	Read up to L characters from the file descriptor D into the vector V.
`writepacked(D,V,L)`	12	Write L characters from the buffer V to the file descriptor D.
`reposition(D,H,L,O)`	13	Move the read/write pointer of the descriptor D to H*65536+L. O specifies the origin.
`rename(O,N)`	14	Rename the file O to N.
`memcopy(R,S,L)`	15	Copy L characters from location S to R.
`memcomp(R,S,L)`	16	Compare L characters at the locations R and S. Return the lexical difference R-S.

Table 37: Extended Runtime Support Procedures

7.2.4 Escape Sequences

ES	**ASCII**	**Description**
`\a`	BEL	Alarm bell
`\b`	BS	Backspace
`\e`	ESC	Escape
`\f`	FF	Form feed
`\n`	LF	Line feed (newline)
`\r`	CR	Carriage return
`\s`	' '	Blank
`\t`	HT	Horizontal TAB
`\v`	VT	Vertical TAB
`\q`	'"'	Literal double quote
`\\`	'\'	Literal backslash

Table 38: Escape Sequences

7.2.5 Optimization Templates

In	Out
`X 0 MUL, 0 X MUL`	`0`
`X 1 MUL, 1 X MUL`	`X`
`X -1 MUL, -1 X MUL`	`X NEG`
`X 0 UMUL, 0 X UMUL`	`0`
`X 1 UMUL, 1 X UMUL`	`X`
`X 0 DIV`	`Error`
`0 X DIV`	`0`
`X 1 DIV`	`X`
`X -1 DIV`	`X NEG`
`X 0 UDIV`	`Error`
`0 X UDIV`	`0`
`X 1 UDIV`	`X`
`X 1 MOD`	`0`
`X 0 MOD`	`Error`
`0 X MOD`	`0`
`X 0 ADD, 0 X ADD`	`X`
`X 0 SUB`	`X`
`0 X SUB`	`X NEG`
`X 0 BAND, 0 X BAND`	`0`
`X -1 BAND, -1 X BAND`	`X`
`X 0 BOR, 0 X BOR`	`X`
`X -1 BOR, -1 X BOR`	`-1`
`X -1 BXOR, -1 X BXOR`	`X BNOT`
`0 X BSHL, 0 X BSHR`	`0`
`X 0 BSHL, X 0 BSHR`	`X`
`X 0 DEREF IND`	`X IND`
`X 0 DREFB INDB`	`X INDB`
`LDLV A N DEREF`	`LDLV (X-N)`
`N LDL A ADD SAVL A`	`INCL A N`
`LDL A N ADD SAVL A`	
`N LDG A ADD SAVG A`	`INCG A N`
`LDG A N ADD SAVG A`	
`LDL A N SUB SAVL A`	`INCL A -N`
`LDG A N SUB SAVG A`	`INCG A -N`

Table 39: Expression Folding Templates

Symbols:

X denotes a variable expression like `LDL X` or `LDG X`.

N denotes a constant expression like `NUM N`.

A denotes an address.

Constant numbers imply `NUM`: −1 is equal to `NUM -1`.

In	Out
`NUM F BRF L`	`JUMP L`
`NUM T BRT L`	`JUMP L`
`NUM T BRF L`	`NOP`
`NUM F BRT L`	`NOP`
`NOT BRT`	`BRF`
`NOT BRF`	`BRT`
`NOT NBRF`	`NBRT`
`NOT NBRT`	`NBRF`

Symbols:

T denotes a *true* constant.
F denotes a *false* constant.
L denotes a label.

Table 40: Condition Folding Templates

7.3 References

7.3.1 Examples

7.3.2 Pictures

7.3.3 Tables

7.3.4 Index