WRITE YOUR OWN COMPILER

Nils M Holm

Write Your Own Compiler

Nils M Holm, 2017

Print and Distribution:
Lulu Press, Inc.; Raleigh, NC; USA

Put your heart and soul in whatever you do

Contents

Preface **7**
Rules of the Game **9**
The Language **12**
Syntax *12*
Semantics *14*
The Target Architecture **16**
Execution Model *16*
The Compiler **19**
Prelude *19*
Symbol Table *23*
Code Generator *27*
Digression: Execution Model *29*
The VSM Instructions *30*
Marking and Resolving *32*
Function Contexts *33*
Back to the Code *35*
Digression: The Accumulator *39*
Back to the Code *40*
Scanner *44*
The Scanner Code *44*
Parser *56*
Parser Prelude *56*
Declaration Parser *59*
Expression Parser *71*
Statement Parser *89*
Program Parser *100*
Initialization *101*
Main Program *105*

The ABI **107**
Compiling the Compiler **109**
Bootstrapping the T3X9 Compiler *110*
Testing the Compiler *111*
Some Random Facts *112*
Future Projects *113*
Appendix **114**
VSM Code Fragments *115*
T3X9 Summary *121*
Program *121*
Comments *121*
Declarations *121*
Statements *123*
Expressions *127*
Conditions *129*
Function Calls *129*
Literals *130*
Naming Conventions *132*
Shadowing *133*
Variadic Functions *134*
Built-In Functions *134*
386 Assembly Summary *137*
List of Figures **141**
Bibliography **143**
Index **144**
Program Symbols *144*
Definitions *146*

Preface

This text is the most minimal complete introduction to compiler-writing that I can imagine. It covers the entire process from analyzing source code to generating an executable file in about 100 pages of prose. The compiler discussed on the text is entirely self-contained and does not rely on any third-party software, except for an operating system.

The book covers lexical analysis, syntax analysis, and code generation by means of a minimal high-level programming language. The language is powerful enough to implement its own compiler in less than 1600 lines of readable code. The main part of the text is comprised of a tour through that code.

The language used in this book is a subset of T3X, a minimal procedural language that was first described in 1996 in the book "Lightweight Compiler Techniques". Although T3X already is quite minimal, T3X9, the dialect discussed here, is even smaller.

If you are familiar with Pascal, C, Java, or any other procedural language, T3X will be easy to pick up. If you are completely new to the field, there is a brief introduction to T3X9 in the appendix.

The T3X9 compiler runs on FreeBSD-386 and generates code for FreeBSD-386, so you will need that system on your computer if you want to experiment with the compiler. Of course it seems curious to install 2G bytes of software in order to run a 32K-byte executable, but then these are interesting times.

Like the compiler, this book is self-contained. It includes the full compiler, its runtime support code, and enough information to understand both the source language and the target platform.

Welcome to compiler writing, enjoy the tour!

Nils M Holm, April 2017

Rules of the Game

A compiler is a program that reads the source code of a program and outputs an executable form of the same program. The most important aspect of a compiler the generation of *correct* code, i.e. the executable program must perform exactly those actions which the source program describes.

Because a compiler is a program and translates *programs* from source to executable form, it may under some circumstances compile itself. A compiler that compiles itself is called a *self-hosting* compiler. The prerequisite for this to work is that the source language and the implementation language of the compiler are the same.

Generally, three languages are involved when talking about compilers:

- the source language
- the target language
- the implementation language

The source language S is the language which the compiler "understands", i.e. the form of the programs it *reads*. This is typically a *programming language*, such as C, Pascal, or Lisp.

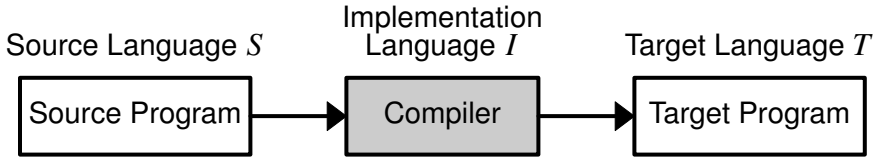


Figure 1: Compilation

The target language T is the machine language that the compiler outputs, for example: machine code for an x86 CPU or bytecode for the Java Virtual Machine (JVM). Executables are typically packaged in some executable file format, like JAR (Java ARchive), COFF (Common Object File Format), or ELF (Executable and Linkable Format).

While the term “compilation” is not necessarily limited to this scenario, we will use it to describe the transformation of a source program to an object program, as illustrated in figure 1.

Executable code is also called *object code*. Many compilers generate linkable object code instead of executable object code. In this case an additional program, a linker or linkage editor, is required to construct an executable program.

This is mostly done in order to support concepts such as separate compilation or runtime libraries. In separate compilation, chunks of a large program are compiled separately and then glued together by the linker.

Runtime libraries are a part of most compiler infrastructures. They provide pre-defined functions that can be used in a source program. Libraries are very common, but in some simple cases, the compiler may generate the pre-defined functions directly instead of referring to a library.

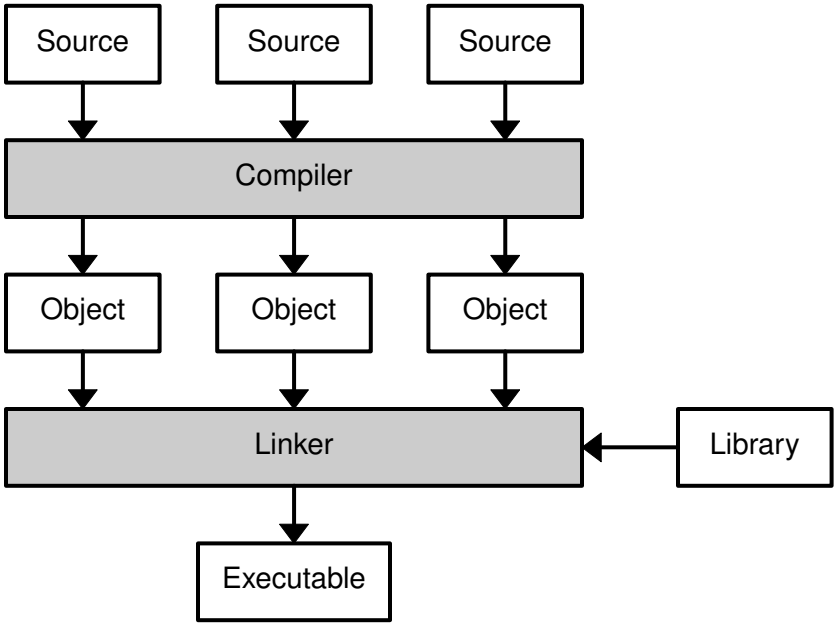


Figure 2: Separate Compilation with Linking

Figure 2 summarizes the process of separate compilation, libraries, and linking.

The compiler described in this text, however, uses the simplified model introduced initially. It reads a single source program and translates it directly to an executable file. All pre-defined functions are generated by the compiler, there is no linker and no support for runtime libraries.

The Language

The source language *and* implementation language used in this book is a subset of an obscure, little, procedural language called T3X. It is a tiny language that once had a tiny community, and it was even used to write some real-life software, like its own compiler and linker, an integrated development system, a database system, and a few simple games. It was also subject of a few college course, most probably because (1) it was reasonably well defined and documented and (2) due to the size of its community, nobody would do your homework assignments for you.

T3X looks like a mixture of C, Pascal, and BCPL. It has untyped data and typed operators, which simplifies the compiler *a lot*, but also leaves all the type checking to the programmer, which is a nightmare from the perspective of a modern software developer.

But this text is not about creating a product and make a shiny web page about it. This is about diving right into the depths of the matter and having some *fun*. And a fun language T3X is. It is interesting to see how little you need to be able to write quite comprehensible and expressive programs.

Syntax

Syntax is what a language looks like. T3X is *block-structured*, *procedural* language, which means that its programs describe *procedures* for manipulating data, i.e. “what to do with data”. It is called *block-structured*, because it is *structured* language that divides programs into blocks. A block is a chunk of source code that describes a part of a procedure.

A *structured* language uses certain constructs to describe the flow of control while a program executes, typically *selection* and *loops* (repetition).

Source code of procedural languages is mostly organized in the form a hierarchy consisting of a programs, procedures or functions, declarations, statements, and expressions. The most

abstract view is the program, the least abstract one the expression. See figure 3 for an illustration.

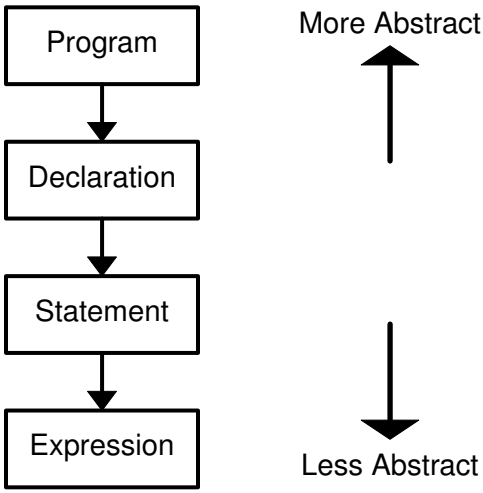


Figure 3: Elements of Block-Structured Languages

In procedural languages:

- programs contain declarations, statements, and expressions
- declarations contain statements and expressions
- statements contain expressions

If you are familiar with C or Pascal or Java, the T3X syntax will look quite familiar. Here is the infamous bubblesort algorithm in T3X:

```
! This is a comment
bubblesort(n, v) do var i, swapped, t;
    swapped := %1;
    while (swapped) do
        swapped := 0;
        for (i=0, n-1) do
            if (v[i] > v[i+1]) do
                t := v[i];
                v[i] := v[i+1];
                v[i+1] := t;
                swapped := %1;
```

```
            end
        end
    end
end
```

bubblesort(n, v) starts the declaration of the procedure *bubblesort* with the formal arguments n and v. The body of the procedure is a block statement (or *compound statement*) enclosed in the keywords **DO** and **END**. The compound statement declares the local variables i, *swapped*, and t.

Assignment is done by the **:=** operator (and equality is expressed with **=**). The statement **FOR (i=0, n-1)** counts from 0 to $n-2$. The i^{th} element of a vector (or array) v is accessed using **v[i]**. Elements are numbered $v[0] \cdots v[n-1]$.

The lexeme **%1** denotes the number -1. You could also write **-1**, but there is a subtle difference: the former is a value, and the latter is an operator applied to a value, which will not work in contexts where a constant is expected.

Furthermore, **/** and **\/** denotes logical (short-circuit) AND and OR, and **X->Y:Z** means "if x then y else z", just like **x?y:z** in C.

IF with an **ELSE** is called **IE** (If/Else).

You will pick up the rest of the T3X syntax as we walk through the compiler source code. If you are interested, there is a brief introduction to T3X in the appendix.

Semantics

Semantics is how the syntax is interpreted. Note that "interpreted" does not imply the use of interpreting software here. Interpretation can be done at various levels, and in the case of the T3X compiler presented here, the code will eventually be interpreted by a 386 (or x86) CPU.

Interpretation in this case is a question of meaning. What does a statement like

```
WHILE (swapped) DO ... END
```

mean? To you, it is probably obvious that it means: "while the value of *swapped* is a 'true' value, repeat everything between **DO** and **END**".

But now we need to know what a "true" value is and what "repetition" means. This is what the semantics of a language describe.

For example, the expressions **v[i]** and **s::i** both denote the i^{th} element of a vector. However, the first variant describes the i^{th} machine word in a vector of machine words, and the second one describes the i^{th} *byte* in a *byte vector*.

In this book, semantics will be specified in two ways: by diagrams describing program flow and by short machine code sequences that resemble the meaning of a language construct.

For instance, the meaning of the **[]** operator in **v[i]** would be specified as

```
shl     $0x2,%eax
pop     %ebx
add     %ebx,%eax
mov     (%eax),%eax
```

assuming that $i = \%eax$ and the address of v is on top of the stack.

(Note that AT&T notation is used here, so **mov a,b** means "move a to b". See the 386 assembly summary in the appendix for further details.)

The exact semantics of the T3X language will be explained informally during the tour through the compiler source code.

The Target Architecture

The target language of the compiler discussed here will be code for the 386 processor family. The code will be packaged in an ELF-format file, and it will use the FreeBSD application binary interface (ABI).

386 machine code can be interpreted by a variety of modern processors, even the latest members of the x86 family. Unlike x86-64 code (64-bit x86 code), it can also be interpreted by older, 32-bit processors.

The ELF format is very popular in modern Unix-based operating systems. For instance, it is used by all modern BSD and Linux systems.

The compiler outputs machine programs that use the FreeBSD ABI to communicate with the operating system, which means that the code generated by the compiler will run on FreeBSD systems.

If you do not have a FreeBSD system, you can either install one in a virtual machine, or change the ABI-specific parts of the runtime support functions of the compiler to use a different ABI. This will be explained later in the text (page 107).

The CPU- and ABI-specific parts of the compiler are all contained in a single procedure in the source code, so porting it to a different operating system or a different 32-bit CPU could be an interesting project once you have finished the tour.

Execution Model

In the ideal case, the execution model of a compiler is exactly the target machine for which code is generated. In fact, CPUs and compilers are designed in such a way that the mapping from source code to machine instructions is as straight-forward as possible.

However, such an ideal mapping is also non-trivial, so in practice, there is often an additional layer between the source language and the target language. We call this additional layer the *execution model* of the compiler.

For instance, an ideal compiler (without an extra execution model) might translate the statement

```
a := a + b * c;
```

to

```
mov     b,%eax
mov     c,%ecx
mul     %ecx
add     %eax,a
```

However, generating such code is far beyond the capabilities of a simple compiler. An easier approach would be to generate chunks of machine code that resemble a more abstract language. A very common abstract machine is the *virtual stack machine* (VSM), where operands are placed on a stack and operators operate on the top stack elements.

In (fictitious) stack machine code, the above program could look like this:

```
load a
load b
load c
mul
add
store a
```

When the program executes, each *load* instruction pushes a value to the stack, *mul* replaces the top two elements by their product, *add* replaces them by their sum, and *store* removes the top element and stores it in memory. Figure 4 shows what the stack would look like during execution.

Stack machine instructions are particularly easily mapped to various CPU types. Figure 5 shows a translation map from above stack machine instructions to 386 code. (We will later see that we can do a little better than this by applying a simple optimization.)

The beauty of this approach is that each virtual stack machine instruction can be translated to a single, static chunk of machine code, so a program can be compiled to native machine code by

first translating it to stack machine code and then outputting the corresponding chunk for each VSM instruction.

Program	Stack
`load a`	`a`
`load b`	`b a`
`load c`	`c b a`
`mul`	`b*c a`
`add`	`a+b*c`
`store a`	

Figure 4: Stack Machine Program Execution

Another beautiful aspect of this approach is that VSM code is a natural by-product of syntax analysis, so the compiler in the next chapter will be able to emit native code *while* performing syntax analysis. Compilers do not get much simpler than that.

Stack Machine	386 CPU
`load a`	`pushl a`

`mul`	`pop %ecx`
	`pop %eax`
	`mul %ecx`
	`push %eax`

`add`	`pop %ecx`
	`pop %eax`
	`add %ecx`
	`push %eax`

`store a`	`popl a`

Figure 5: Stack Machine Instruction Mapping

The Compiler

All code in this book, *but not the prose*, is provided under the Creative Commons Zero license, i.e. it can be used for any purpose, without attribution:
http://creativecommons.org/publicdomain/zero/1.0/

Prelude

```
! T3X9 -> ELF-FreeBSD-386 compiler
! Nils M Holm, 2017, CC0 license
```

We are assuming four bytes per word, which is fair enough for most 32-bit processors and certainly true for the 386.

```
const   BPW = 4;
```

PROG_SIZE is the maximum size of the input program. To keep things simple, we read the whole program into memory in one chunk. When the program is larger than this, we are out of luck.

```
const   PROG_SIZE = 65536;
```

The next two constants specify the maximum size of the text (code) segment and the data segment of the generated executable. Since we are operating in a 32-bit environment, feel free to increase those (and **PROG_SIZE**, above) as you see fit.

Due to the way in which the virtual load addresses in the resulting ELF file will be organized, **TEXT_SIZE** must be a multiple of **PAGE_SIZE**. (See page 27.)

```
const   TEXT_SIZE = 65536;  ! * PAGE_SIZE !
const   DATA_SIZE = 65536;
```

NRELOC specifies the maximum number of relocation entries. A *relocation entry* specifies a location in the generated executable whose content has to be adjusted later, because some part of the program is relocated in memory.

In this particular case, we do not know the final location of the data segment, because we cannot know in advance the size of the text segment.

```
const   NRELOC = 10000;
```

An internal stack is used to keep track of all kinds of things, like the beginnings of functions, loop contexts, etc. This is the maximum number of stack elements.

```
const   STACK_SIZE = 100;
```

SYMTBL_SIZE is the number of *symbol table* entries. Each symbol that is declared in the source program will allocate one slot in the table. **NLIST_SIZE** is the size of the *name list*, which is a memory region holding the names of all declared symbols.

```
const   SYMTBL_SIZE = 1000;
const   NLIST_SIZE = 10000;
```

This is the compile stack and the stack pointer.

```
var     Stack[STACK_SIZE], Sp;
```

The **Line** variable is used to keep track of the current input line number.

```
var     Line;
```

This constant denotes the end of the input file (or, rather, the end of the program in the buffer) internally.

```
const   ENDFILE = %1;
```

T3X9 being a very simple language, there is no function for converting an integer to a string, so we will define one here.

Ntoa(x) returns the address of a buffer holding the ASCII representation of the integer x. The buffer is static and will be overwritten each time **ntoa()** is called. Negative numbers will have a leading **–** sign attached.

```
var     ntoa_buf::100;

ntoa(x) do var i, k;
    if (x = 0) return "0";
    i := 0;
```

```
    k := x<0-> -x: x;
    while (k > 0) do
        i := i+1;
        k := k/10;
    end
    i := i+1;
    if (x < 0) i := i+1;
    ntoa_buf::i := 0;
    k := x<0-> -x: x;
    while (k > 0) do
        i := i-1;
        ntoa_buf::i := '0' + k mod 10;
        k := k/10;
    end
    if (x < 0) do
        i := i-1;
        ntoa_buf::i := '-';
    end
    return @ntoa_buf::i;
end
```

The **str.length()** function returns the length of a string. Since strings are NUL-terminated, this can be done efficiently using **t.memscan()**. The function *will* return wrong results for strings longer than 32766 characters. Feel free to increase the number.

```
str.length(s) return t.memscan(s, 0, 32767);
```

Str.copy(sd,ss) copies the string *ss* (source) to the byte vector *sd* (destination). **Str.append(sd,ss)** appends *ss* to *sd*. *Sd* must provide enough space for the concatenated string.

```
str.copy(sd, ss)
    t.memcopy(ss, sd, str.length(ss)+1);

str.append(sd, ss)
    t.memcopy(ss, @sd::str.length(sd),
              str.length(ss)+1);
```

The following function returns -1, if the strings $s1$ and $s2$ are equal, and 0 otherwise.

```
str.equal(s1, s2)
  return t.memcomp(s1, s2, str.length(s1)+1) = 0;
```

Writes() writes a string to standard output and **log()** writes a string to the standard error stream.

```
writes(s) t.write(1, s, str.length(s));

log(s) t.write(2, s, str.length(s));
```

Aw, something went wrong! So we declare our discontent and leave. M is the message to print and s holds an additional piece of information or 0.

No attempt at error recovery is made, again simplifying the design of the compiler a lot!

```
aw(m, s) do
    log("t3x9: ");
    log(ntoa(Line));
    log(": ");
    log(m);
    if (s \= 0) do
        log(": ");
        log(s);
    end
    log("\n");
    halt 1;
end
```

Oops, we didn't expect this!

```
oops(m, s) do
    log("t3x9: internal error\n");
    aw(m, s);
end
```

The following are operators manipulating the compile stack. **Push(x)** pushes x, **pop()** pops the most recently pushed value, **tos()** retrieves that value without popping it (tos = "top of stack"), and **swap()** exchanges the top two elements.

```
push(x) do
    if (Sp >= STACK_SIZE)
        oops("stack overflow", 0);
    Stack[Sp] := x;
    Sp := Sp+1;
end

tos() return Stack[Sp-1];

pop() do
    if (Sp < 1) oops("stack underflow", 0);
    Sp := Sp-1;
    return Stack[Sp];
end

swap() do var t;
    if (Sp < 2) oops("stack underflow", 0);
    t := Stack[Sp-1];
    Stack[Sp-1] := Stack[Sp-2];
    Stack[Sp-2] := t;
end
```

The following functions define character classes by returning truth when the character c belongs to the corresponding class.

```
numeric(c) return '0' <= c /\ c <= '9';

alphabetic(c) return 'a' <= c /\ c <= 'z' \/
                     'A' <= c /\ c <= 'Z';
```

Symbol Table

A *symbol table entry* consists of three fields. **SNAME** holds the address of the corresponding name in the name list. **SFLAGS** contains the type of the symbol, and **SVALUE** its value (which is the address of the symbol in case of variables, vectors, and functions).

```
struct  SYM = SNAME, SFLAGS, SVALUE;
```

These are **SFLAGS** values: **GLOBF** indicates that a symbol is *global*, i.e. has not been declared inside of a function or statement block. **CNST** indicates a constant, **VECT** a vector or byte vector, **FORW** a forward declaration, and **FUNC** a function.

GLOBF is OR'ed together with the type of a symbol, e.g. **GLOBF|VECT** would describe a global vector.

```
const   GLOBF = 1;
const   CNST  = 2;
const   VECT  = 4;
const   FORW  = 8;
const   FUNC  = 16;
```

Syms holds the symbol table and **Nlist** the name list. The syntax **v[x*y]** allocates a vector of $x \cdot y$ machine words. In this case it allocates a vector of **SYMTBL_SIZE** symbol table structures. The layout of the **Syms** vector is shown in figure 6.

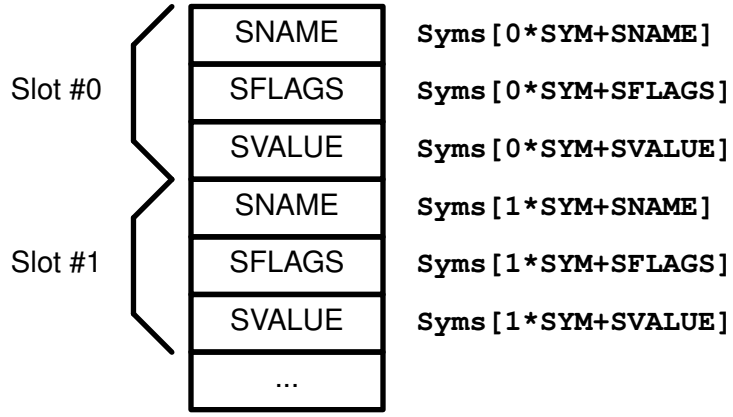


Figure 6: Symbol Table Layout

The syntax **v::n** allocates a vector of n bytes. Note that this is still a vector of machine words, it is just being *allocated* in units of bytes.

Yp and **Np** are the offsets of the free regions of the above vectors. A **SYM** structure is allocated in **Syms** by incrementing **Yp** by **SYM**, and a name is allocated in **Nlist** by adding the length of the name (plus 1 for the trailing NUL) to **Np**.

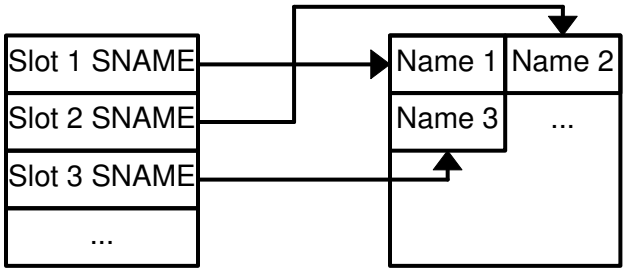


Figure 7: Symbol Table and Name List

The connection between the symbol table and the name list is illustrated in figure 7.

```
var     Syms[SYM*SYMTBL_SIZE];
var     Nlist::NLIST_SIZE;

var     Yp, Np;
```

Find() locates the symbol table entry with the name given in s. It searches the table from the end to the beginning, because local symbols are stored at the end of the table and so the chance of locating them early increases.

If the language would support *shadowing*, where local symbols can have the same names as global ones, this strategy would always return the local symbol in case of a name collision.

```
find(s) do var i;
    i := Yp-SYM;
    while (i >= 0) do
        if (str.equal(Syms[i+SNAME], s))
            return @Syms[i];
        i := i - SYM;
    end
    return 0;
end
```

The **lookup()** function looks up a symbol in the symbol table and reports undefined names. This happens, for instance, when using a variable in an expression without declaring it first.

The function also makes sure that the located symbol has the proper type by checking the symbol flags against f. For instance, to locate a constant, **CNST** would be passed to **loopup()** in the f argument.

```
lookup(s, f) do var y;
    y := find(s);
    if (y = 0) aw("undefined", s);
    if (y[SFLAGS] & f \= f)
        aw("unexpected type", s);
    return y;
end
```

Newname() adds a name to the name list.

```
newname(s) do var k, new;
    k := str.length(s)+1;
    if (Np+k >= NLIST_SIZE)
        aw("too many symbol names", s);
    new := @Nlist::Np;
    t.memcopy(s, new, k);
    Np := Np+k;
    return new;
end
```

Add() adds the symbol s with flags f and value v to the symbol table. Since there is no shadowing in T3X, a name already existing in the table is a redefinition error, except when the existing symbol is a forward declaration (**DECL**) and the new symbol names a function.

```
add(s, f, v) do var y;
    y := find(s);
    if (y \= 0) do
        ie (y[SFLAGS] & FORW /\ f & FUNC)
            return y;
        else
            aw("redefined", s);
    end
    if (Yp+SYM >= SYMTBL_SIZE*SYM)
        aw("too many symbols", 0);
```

```
    y := @Syms[Yp];
    Yp := Yp+SYM;
    y[SNAME] := newname(s);
    y[SFLAGS] := f;
    y[SVALUE] := v;
    return y;
end
```

Code Generator

An ELF module typically has (at least) two *segments*: one holding the *text* (code) and one holding the *data* of the program. The following constants define the virtual memory addresses at which the operating system will load these segments.

The (hexa-decimal) address 8048000_h is the typical load address for the first segment on a 386-based system. [Lev99] The data segment is placed right after the text segment.

Because of the way in which **DATA_VADDR** is computed, **TEXT_SIZE** must be a multiple of **PAGE_SIZE** (all virtual addresses must be page-aligned).

```
const   TEXT_VADDR = 134512640;      ! 08048000h
const   DATA_VADDR = TEXT_VADDR + TEXT_SIZE;
```

HEADER_SIZE is the size of a minimal two-segment ELF header.

```
const   HEADER_SIZE = 116;  ! 74h
```

PAGE_SIZE specifies the size of a virtual memory page in the target operating system. It is also used for segment alignment in the ELF file.

```
const   PAGE_SIZE = 4096;
```

A *relocation entry* consists of an address (**RADDR**) and a segment (**RSEG**). Relocation entries are allocated in the same was as symbol table entries (page 24).

```
struct  RELOC = RADDR, RSEG;

var     Rel[RELOC*NRELOC];
```

These byte vectors hold the two segments and the ELF header.

```
var     Text_seg::TEXT_SIZE;
var     Data_seg::DATA_SIZE;
var     Header::HEADER_SIZE;
```

The following variables indicate:

- **Rp** - the free region in the relocation table
- **Tp** - the next address in the text segment
- **Dp** - the next address in the data segment
- **Lp** - the next address in a local stack frame
- **Hp** - the free region in the ELF header

When emitting code and data, this means that **Tp** always contains the *current address* in the code segment, and **Dp** always contains the current address in the data segment. Because jump instructions are always relative to the current address on the 386, text addresses never need relocation, only data addresses do.

This is why the **RELOC** structure has only one segment field, which indicates the segment *in which* the relocation has to be performed (but no field indicating relative *to which* segment it has to be done).

```
var     Rp, Tp, Dp, Lp, Hp;
```

The **Acc** and **Codetbl** variables as well as the **CG** structure are part of the execution model of the T3X9 compiler. The model will be explained immediately.

```
var     Acc;

var     Codetbl;

struct  CG =
        CG_PUSH, CG_CLEAR,
        CG_LDVAL, CG_LDADDR, CG_LDLREF, CG_LDGLOB,
        CG_LDLOCL,
        CG_STGLOB, CG_STLOCL, CG_STINDR, CG_STINDB,
        CG_INCGLOB, CG_INCLOCL,
        CG_ALLOC, CG_DEALLOC, CG_LOCLVEC, CG_GLOBVEC,
```

```
CG_INDEX, CG_DEREF, CG_INDXB, CG_DREFB,
CG_MARK, CG_RESOLV,
CG_CALL, CG_JUMPFWD, CG_JUMPBACK, CG_JMPFALSE,
CG_JMPTRUE, CG_FOR, CG_FORDOWN,
CG_ENTER, CG_EXIT, CG_HALT,
CG_NEG, CG_INV, CG_LOGNOT, CG_ADD, CG_SUB,
CG_MUL, CG_DIV, CG_MOD, CG_AND, CG_OR, CG_XOR,
CG_SHL, CG_SHR, CG_EQ, CG_NEQ, CG_LT, CG_GT,
CG_LE, CG_GE,
CG_WORD;
```

Digression: Execution Model

The execution model of the T3X9 compiler is a virtual stack machine (VSM) implementing the instructions outlined in this section.

The VSM has four registers: an accumulator, an instruction pointer, a stack pointer, and a frame pointer. The accumulator caches the top of the stack. The instruction pointer points to the next instruction to execute, the stack pointer points to the element most recently placed on the stack, and the frame pointer points to the *context* of the current function. (See page 33.)

In the following,

- S will denote the stack
- I will denote the instruction pointer
- P will denote the stack pointer
- F will denote the frame pointer
- S_0 will denote the element on top of the stack
- S_1 will denote the second element on the stack
- decrementing P will add an element to the stack
- incrementing P will remove an element from the stack
- w, v will indicate machine words
- a will indicate an address (which has to be relocated)

- $[x]$ will indicate the value at address x
- $b[x]$ will indicate the byte at address x

Due to the meaning of P, the stack can be imagined to grow *downwards*, which is what the stack on the 386 CPU actually does.

The VSM Instructions

CG_PUSH	$P := P - 1;\ S_0 := A$
CG_CLEAR	$A := 0$
CG_LDVAL w	$P := P - 1;\ S_0 := A;\ A := w$
CG_LDADDR a	$P := P - 1;\ S_0 := A;\ A := a$
CG_LDLREF w	$P := P - 1;\ S_0 := A;\ A := F + w$
CG_LDGLOB a	$P := P - 1;\ S_0 := A;\ A := [a]$
CG_LDLOCL w	$P := P - 1;\ S_0 := A;\ A := [F + w]$
CG_STGLOB a	$[a] := A;\ A := S_0;\ P := P + 1$
CG_STLOCL w	$[F + w] := A;\ A := S_0;\ P := P + 1$
CG_STINDR	$[S_0] := A;\ P := P + 1$
CG_STINDB	$b[S_0] := A;\ P := P + 1$
CG_INCGLOB a v	$[a] := [a] + v$
CG_INCLOCL w v	$[F + w] := [F + w] + v$
CG_ALLOC w	$P := P - w$
CG_DEALLOC w	$P := P + w$
CG_LOCLVEC	$w := P;\ P := P - 1;\ S_0 := w$
CG_GLOBVEC a	$[a] := P$
CG_INDEX	$A := 4 \cdot A + S_0;\ P := P + 1$
CG_DEREF	$A := [A]$
CG_INDXB	$A := A + S_0;\ P := P + 1$

CG_DREFB $A := b[A]$

CG_MARK see next section

CG_RESOLV see next section

CG_CALL w $P := P - 1;\ S_0 := I;\ I := w$

CG_JUMPFWD w $I := w;$

CG_JUMPBACK w $I := w;$

CG_JMPFALSE w if $S_0 = 0$, then $I := w;$ always: $P := P + 1$

CG_JMPTRUE w if $S_0 \neq 0$, then $I := w;$ always: $P := P + 1$

CG_FOR w if $S_0 \geq A$, then $I := w;$ always: $P := P + 1$

CG_FORDOWN w if $S_0 \leq A$, then $I := w;$ always: $P := P + 1$

CG_ENTER $P := P - 1;\ S_0 := F;\ F := P$

CG_EXIT $F := S_0;\ I := S_1;\ P := P + 2$

CG_HALT w halt program execution, return w

CG_NEG $A := -A$

CG_INV $A :=$ bitwise complement of A

CG_LOGNOT if $A = 0$ then $A := -1$ else $A := 0$

CG_ADD $A := S_0 + A;\ P := P + 1$

CG_SUB $A := S_0 - A;\ P := P + 1$

CG_MUL $A := S_0 \cdot A;\ P := P + 1$

CG_DIV $A := S_0\ div\ A;\ P := P + 1$

CG_MOD $A := S_0\ mod\ A;\ P := P + 1$

($x\ div\ y$ is the integer quotient of x and y and $x\ mod\ y$ is the remainder of the integer division.)

CG_AND $A := S_0\ AND\ A;\ P := P + 1$

CG_OR $A := S_0\ OR\ A;\ P := P + 1$

CG_XOR $A := S_0\ XOR\ A;\ P := P + 1$

(x *AND* y is the logical AND, x *OR* y is the logical OR, and x *XOR* y is the logical exclusive OR of x and y.)

CG_SHL $A := S_0 \cdot 2^A;\ \ P := P + 1$ (left shift)

CG_SHR $A := S_0\ div\ 2^A;\ \ P := P + 1$ (right shift)

CG_EQ if $S_0 = A$ then $A := -1$ else $A := 0$;
always: $P := P + 1$

CG_NEQ if $S_0 \neq A$ then $A := -1$ else $A := 0$;
always: $P := P + 1$

CG_LT if $S_0 < A$ then $A := -1$ else $A := 0$;
always: $P := P + 1$

CG_GT if $S_0 > A$ then $A := -1$ else $A := 0$;
always: $P := P + 1$

CG_LE if $S_0 \leq A$ then $A := -1$ else $A := 0$;
always: $P := P + 1$

CG_GE if $S_0 \geq A$ then $A := -1$ else $A := 0$;
always: $P := P + 1$

Marking and Resolving

The **CG_MARK** instruction is not really a VSM instruction but rather an instruction to the compiler. It tells the compiler to remember the current text segment address (**Tp**) for later reference. A marked address is pushed to the compile stack, so multiple locations can be marked, but only the most recently marked one can be resolved.

CG_RESOLV resolves a marked address by *backpatching*. This is illustrated in figure 8.

Imagine you wanted to generate a jump to some location that follows later in the code. This is called a *forward jump*, because it branches forward in the code. It is also impossible to generate immediately, because the destination is not yet known. So a placeholder is generated instead of the destination address and a mark is dropped. (F8.1)

Then the code between the marked address and the destination is generated. (F8.2)

Finally, when the destination of the jump is reached, the compiler pops the mark off the stack and inserts the current address (**Tp**) in the location pointed to by the mark. (F8.3)

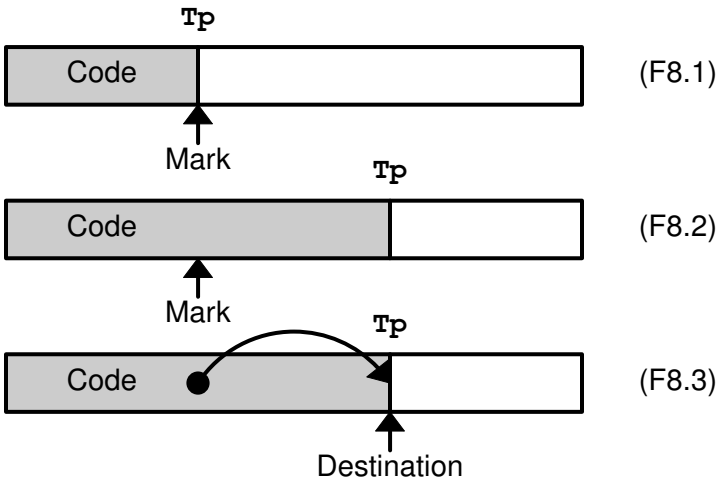


Figure 8: Resolving a Mark by Backpatching

Because this last step modifies code that has been emitted earlier, it is referred to as backpatching.

A mark can also be resolved by just generating a jump *back* to the marked address. In this case, no backpatching is required, because the destination is already known when the jump instruction is generated.

The following VSM instruction generate forward jumps: **CG_JUMPFWD**, **CG_JMPFALSE**, **CG_JMPTRUE**, **CG_FOR**, **CG_FORDOWN**. They place a mark that has to be resolved later.

There is only one instruction involving a backward jump: **CG_JUMPBACK**. It requires a mark to already be in place.

Function Contexts

A *function context* (or *context*) is a region on the stack where a function stores its arguments and local variables. A context is set up by the **CG_CALL** and **CG_ENTER** instructions.

CG_CALL puts the *return address* on the stack *after* the arguments have been placed there by the caller. The return address is the address where program execution will continue when the called function returns.

CG_ENTER saves the current *frame pointer* F on the stack, and then sets F to the value of P, thereby creating a new frame. The names “frame” and “context” are used as synonyms here.

A function context with two arguments and three local variables is shown in figure 9.

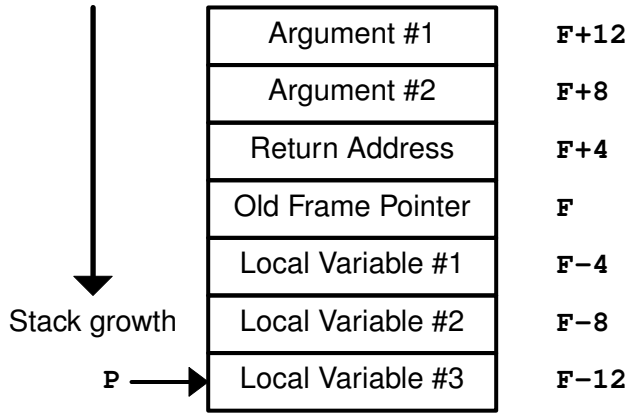


Figure 9: Function Context

The space above argument #1 is allocated by the caller’s frame, which is saved in the “old frame pointer” field of the current frame. Arguments are placed on the stack and removed by the caller when a function returns.

Each function allocates and deallocates its own local variables. Variables are allocated by subtracting their size from the stack pointer P and deallocated by adding their size to P.

All arguments and local variables are addressed relative to the frame pointer F. Their addresses are shown in the right column of Figure 9.

Back to the Code

Emit() writes the next byte to the text segment. If the target program does not fit in memory, we are out of luck. **Emitw()** emits a machine word in little-endian order to the text segment.

```
emit(x) do
    if (Tp >= TEXT_SIZE)
        aw("text segment too big", 0);
    Text_seg::Tp := x;
    Tp := Tp+1;
end

emitw(x) do
    emit(255&x);
    emit(255&(x>>8));
    emit(255&(x>>16));
    emit(255&(x>>24));
end
```

The **tag()** function tags the current location in the text or data segment for relocation. Note that **BPW** is subtracted from the current location, because the address to relocate already has been emitted when **tag()** is called.

```
tag(seg) do
    if (Rp+RELOC >= RELOC*NRELOC)
        oops("relocation buffer overflow", 0);
    Rel[Rp+RADDR] := seg = 't'-> Tp-BPW: Dp-BPW;
    Rel[Rp+RSEG] := seg;
    Rp := Rp+RELOC;
end
```

tpatch() patches the machine word at text segment location a to contain the value x, and **tfetch()** retrieves the machine word at text segment location a. These are used for backpatching and relocation.

```
tpatch(a, x) do
    Text_seg::a := 255&x;
    Text_seg::(a+1) := 255&(x>>8);
```

```
    Text_seg::(a+2) := 255&(x>>16);
    Text_seg::(a+3) := 255&(x>>24);
end

tfetch(a) return  Text_seg::a
        | (Text_seg::(a+1)<<8)
        | (Text_seg::(a+2)<<16)
        | (Text_seg::(a+3)<<24);
```

Data() and **dataw()** emit bytes and words to the data segment in the same way as **emit()** and **emitw()** emit to the text segment.

```
data(x) do
    if (Dp >= DATA_SIZE)
        aw("data segment too big", 0);
    Data_seg::Dp := x;
    Dp := Dp+1;
end

dataw(x) do
    data(255&x);
    data(255&(x>>8));
    data(255&(x>>16));
    data(255&(x>>24));
end
```

Similarly, **dpatch()** and **dfetch()** patch and retrieve values in the data segment.

```
dpatch(a, x) do
    Data_seg::a := 255&x;
    Data_seg::(a+1) := 255&(x>>8);
    Data_seg::(a+2) := 255&(x>>16);
    Data_seg::(a+3) := 255&(x>>24);
end

dfetch(a) return  Data_seg::a
        | (Data_seg::(a+1)<<8)
        | (Data_seg::(a+2)<<16)
        | (Data_seg::(a+3)<<24);
```

The **hex()** function returns the value of the hexadecimal ASCII-digit passed to it.

```
hex(c)
    ie (numeric(c))
        return c-'0';
    else
        return c-'a'+10;
```

The **rgen()** function emits program code to the text segment from an augmented hexdump passed to it in the string s. V is an (optional) value that will be inserted into the hexdump when the dump contains an instruction to do so.

Basically, **rgen()** decodes hexadecimal representations of bytes from s and emits them. However, when it encounters a comma character instead of a hex digit, the following character will be interpreted as follows:

- **w** - emit the value v as a machine word
- **a** - emit the value v as an address (tag for relocation)
- **m** - mark the current text address
- **>** - create a forward jump
- **<** - create a backward jump
- **r** - resolve a forward jump

Note that the 386 CPU performs *relative addressing* in the code segment, so the operand of a jump or call instruction is a *distance*, and not an absolute address. For instance, the instruction

```
0f 84 04 00 00 00   (jz +4)
```

would jump 4 bytes forward from the end of the above instruction. Hence the formula $TP - x - BPW$ calculates the distance for a forward jump and $x - TP - BPW$ computes the distance for a backward jump, where x is the destination address.

```
rgen(s, v) do var x;
    while (s::0) do
        ie (s::0 = ',') do
            ie (s::1 = 'w') do
```

```
                emitw(v);
            end
            else ie (s::1 = 'a') do
                emitw(v);
                tag('t');
            end
            else ie (s::1 = 'm') do
                push(Tp);
            end
            else ie (s::1 = '>') do
                push(Tp);
                emitw(0);
            end
            else ie (s::1 = '<') do
                emitw(pop()-Tp-BPW);
            end
            else ie (s::1 = 'r') do
                x := pop();
                tpatch(x, Tp-x-BPW);
            end
            else do
                oops("bad code", 0);
            end
        end
        else do
            emit(hex(s::0)*16+hex(s::1));
        end
        s := s+2;
    end
end
```

Gen() is like **rgen()**, but generates code for the VSM instruction *id* rather than emitting code from a raw hexdump.

```
gen(id, v) rgen(Codetbl[id][1], v);
```

Digression: The Accumulator

Note that the VSM instructions from the last digression involved a special register, the *accumulator*, which caches the *top of the stack* (TOS). The reasons for adding the accumulator is simple: it saves one memory reference per single-operand operation and three references (two pushes and one pop) per two-operand operation.

VSM	386 w/o accumulator	386 with accmulator
`LDVAL 5` `NEG`	`push $5` `neg (%esp)`	`mov $5,%eax` `neg %eax`

`LDVAL 3`
`LDVAL 2`
 	
`ADD`
	`push $3`
	`push $2`
	
	`pop %ebx`
	`pop %eax`
	`add %ebx,%eax`
	`push %eax`
		`mov $3,%eax`
		`push %eax`
		`mov $2,%eax`
		`pop %ebx`
		`add %ebx,%eax`

Figure 10: Use of an Accumulator

Figure 10 shows some VSM programs and their translations to 386 code with and without the use of $\%eax$ as an accumulator.

Note that in the second program in figure 10, the first **`LDVAL`** instruction of the accumulator-based program just loads the accumulator, while the second **`LDVAL`** first saves the accumulator on the stack and then loads the new value.

This is because the accumulator serves as a *cache* for the TOS. The first value goes to the top of an empty stack. When the second value is pushed, though, the first value has to move from the TOS to position two of the stack first, and position two is on the top of the actual stack.

So we have to keep track of the state of the accumulator. In order to do so, we distinguish two states: *clear* and *active*. Whenever a value is loaded and the accumulator is in *clear* state, we set it to *active* state. When it already is in *active* state, we push its value before loading and leave it in *active* state.

In compiler terminology, we *spill* the register to memory. This is the simplest form of register allocation.

Back to the Code

The following functions perform spilling, query the status, and set the status of the accumulator, which is kept in the **`Acc`** variable.

Note that the **`PUSH`** instruction is automatically generated when needed. It is never emitted explicitly by the other parts of the compiler. (Hence it was not listed in figure 10.)

```
spill()
    ie (Acc)
        gen(CG_PUSH, 0);
    else
        Acc := 1;

active() return Acc;

clear() Acc := 0;

activate() Acc := 1;
```

The **`relocate()`** function resolves relocation entries by fixing the addresses of data objects in the text and data segments. When code is generated, data addresses begin at $Dp = 0$, so a VSM load instruction for the first data object would read **`LDGLOB 0`**.

However, the data segment is not mapped to virtual address 0 at run time, but to **`DATA_VADDR`**. Just adding **`DATA_VADDR`** to each data reference would not suffice, though, due to the way in which an ELF file is mapped to memory. This is illustrated in figure 11.

Because the ELF executable is loaded in chunks of pages, the last page of the text segment will contain the first bytes of the data segment and the first page of the data segment will contain the last bytes of the text segment.

So the overlap between text and data segment causes all data objects in the data segment to move towards the end of the segment by the size of that overlap. The overlap is computed using the formula

$$(HEADER\_SIZE + Tp) \ mod \ PAGE\_SIZE$$

and adding **DATA_VADDR** to that overlap gives exactly the distance of the data segment from 0.

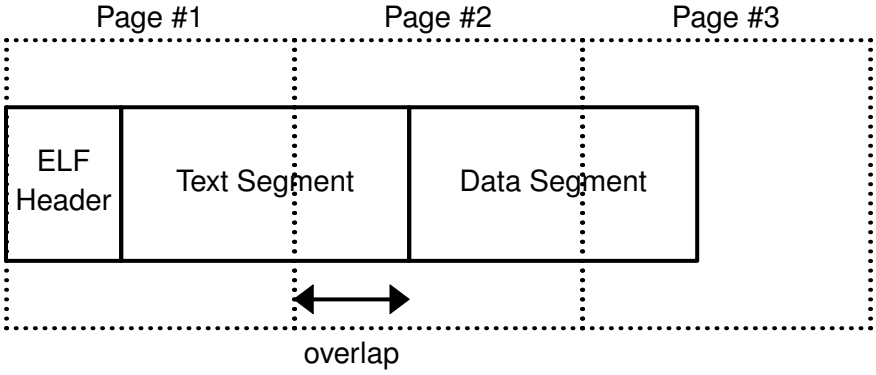


Figure 11: Mapping an ELF File to Memory

```
relocate() do var i, a, dist;
    dist := DATA_VADDR
            + (HEADER_SIZE + Tp) mod PAGE_SIZE;
    for (i=0, Rp, RELOC) do
        ie (Rel[i+RSEG] = 't') do
            a := tfetch(Rel[i+RADDR]);
            a := a + dist;
            tpatch(Rel[i+RADDR], a);
        end
        else do
            a := dfetch(Rel[i+RADDR]);
            a := a + dist;
            dpatch(Rel[i+RADDR], a);
        end
    end
end
```

Builtin() declares a built-in function named *name* with the given *arity* (number of arguments) and the code specified as a hexdump in *code*. The T3X9 compiler does not use a runtime library, but a tiny set of pre-defined functions whose code will be emitted at the beginning of each program.

Builtin() emits the code for these functions and adds their addresses and type information (arity) to the symbol table.

BTW, this is the reason why the empty program **DO END** compiles to a bloated size of 336 bytes instead of 128.

Note the we are jumping over the code of the function instead of just emitting it and then jumping directly into the main body when the program executes. The reason for this will become clear later (see page 62).

```
builtin(name, arity, code) do
    gen(CG_JUMPFWD, 0);
    add(name, GLOBF|FUNC | (arity << 8), Tp);
    rgen(code, 0);
    gen(CG_RESOLV, 0);
end
```

Return x aligned to a multiple of a.

```
align(x, a) return (x+a) & ~(a-1);
```

This function writes a byte to the ELF header.

```
hdwrite(b) do
    if (Hp >= HEADER_SIZE)
        oops("ELF header too long", 0);
    Header::Hp := b;
    Hp := Hp+1;
end
```

The **hexwrite()** function writes a sequence of bytes to the ELF header. The sequence is supplied as a hexdump. **Lewrite()** writes a machine word in little endian byte ordering to the header.

```
hexwrite(b)
    while (b::0) do
        hdwrite(16*hex(b::0)+hex(b::1));
```

```
        b := b+2;
    end

lewrite(x) do
    hdwrite(x & 255);
    hdwrite(x>>8 & 255);
    hdwrite(x>>16 & 255);
    hdwrite(x>>24 & 255);
end
```

The **elfheader()** function writes the complete ELF header to the **Header** vector. See [ELF95] for further details. The FreeBSD ELF loader probably ignores the physical load address.

```
elfheader() do
  hexwrite("7f454c46");    ! magic
  hexwrite("01");          ! 32-bit
  hexwrite("01");          ! little endian
  hexwrite("01");          ! header version
  hexwrite("09");          ! FreeBSD ABI
  hexwrite("0000000000000000");
                           ! padding
  hexwrite("0200");        ! executable
  hexwrite("0300");        ! 386
  lewrite(1);              ! version
  lewrite(TEXT_VADDR+HEADER_SIZE);
                           ! initial entry point
  lewrite(52);             ! program header offset
  lewrite(0);              ! no header segments
  lewrite(0);              ! flags
  hexwrite("3400");        ! header size
  hexwrite("2000");        ! program header size
  hexwrite("0200");        ! number of prog headers
  hexwrite("2800");        ! segment hdr size (unused)
  hexwrite("0000");        ! number of segment headers
  hexwrite("0000");        ! string index (unused)
  ! text segment descrition
  lewrite(1);              ! loadable segment
  lewrite(HEADER_SIZE);    ! offset in file
  lewrite(TEXT_VADDR);     ! virtual load address
```

```
  lewrite(TEXT_VADDR);     ! physical load address
  lewrite(Tp);             ! size in file
  lewrite(Tp);             ! size in memory
  lewrite(5);              ! flags := read, execute
  lewrite(PAGE_SIZE);      ! alignment (page)
  ! data segment descrition
  lewrite(1);              ! loadable segment
  lewrite(HEADER_SIZE+Tp);! offset in file
  lewrite(DATA_VADDR);     ! virtual load address
  lewrite(DATA_VADDR);     ! physical load address
  lewrite(Dp);             ! size in file
  lewrite(Dp);             ! size in memory
  lewrite(6);              ! flags := read, write
  lewrite(PAGE_SIZE);      ! alignment (page)
end
```

Scanner

The *scanner* is the part of the compiler that reads the input program and splits it up into small units called *tokens*. A token consists of multiple elements: a small integer identifying the token (in this text called the *token ID*), and a set of values that describe the token more in detail.

Here is a small sample program:

```
fac(x) RETURN x<1-> 1: x*fac(x-1);
```

Its tokenized form is shown in figure 12.

Each time the scanner is called, it returns the token ID of the next token in the source program and fills in the string, value, and operator ID values, as outlined in figure 12.

The Scanner Code

The **META** constant is used to escape special characters in strings, e.g. the scanner will deliver `'"'|META` when a double quote is to be *included* in a string instead of *delimiting* it.

```
const   META     = 256;
```

TOKEN_LEN is the maximum length for all kinds of tokens, like string literals, numeric literals, and symbol names.

```
const   TOKEN_LEN = 128;
```

The program source code will be read into the **Prog** vector. **Psize** is the length of the source code and **Pp** points to the next character to process.

Token (T)	String (Str)	Value (Val)	Oper. ID (Oid)
SYMBOL	"fac"		
LPAREN	"("		
SYMBOL	"x"		
RPAREN	")"		
KRETURN	"return"		
SYMBOL	"x"		
BINOP	"<"		22
INTEGER	"1"	1	
COND	"->"		
INTEGER	"1"	1	
COLON	":"		
SYMBOL	"x"		
BINOP	"*"		2
SYMBOL	"fac"		
LPAREN	"("		
SYMBOL	"x"		
BINOP	"-"		28
INTEGER	"1"	1	
RPAREN	")"		
SEMI	";"		

Figure 12: Tokenized Program

```
var     Prog::PROG_SIZE;

var     Pp, Psize;
```

When the scanner extracts a token from the source code, it will fill in the following values:

- **T** - the token value itself
- **Str** - the literal text of the token (or the value of a string literal)
- **Val** - the values of integers and characters
- **Oid** - the operator IDs of operators

(These are exactly the variables used in the tokenized program example in figure 12.)

```
var     T;
var     Str::TOKEN_LEN;
var     Val;
var     Oid;
```

Here are the operator IDs of some frequently-used operators.

```
var     Equal_op, Minus_op, Mul_op, Add_op;
```

The **OPER** structure describes a T3X operator. It contains the following fields:

- **OPREC** - the operator precedence
- **OLEN** - the length of the operator symbol
- **ONAME** - the operator symbol
- **OTOK** - the token ID generated for the operator
- **OCODE** - the machine code associated with the operator

The **Ops** variable will be assigned to the operator table in the initialization part of the compiler (see page 101).

```
struct  OPER = OPREC, OLEN, ONAME, OTOK, OCODE;

var     Ops;
```

The **TOKENS** structure contains all token IDs that are used by the compiler. Note that the scanner returns **BINOP** for all binary operators and **UNOP** for all unary operators. In addition it sets the **Oid** variable to the index of the operator in the operator table.

Token IDs starting with a 'K' indicate keywords.

```
struct TOKENS =
    SYMBOL, INTEGER, STRING,
    ADDROF, ASSIGN, BINOP, BYTEOP, COLON, COMMA,
    COND, CONJ, DISJ, LBRACK, LPAREN, RBRACK, RPAREN,
    SEMI, UNOP,
    KCONST, KDECL, KDO, KELSE, KEND, KFOR, KHALT,
    KIE, KIF, KLEAVE, KLOOP, KRETURN, KSTRUCT, KVAR,
    KWHILE;
```

Readprog() reads the complete program into the **Prog** vector. If the program is too big, we are in trouble.

```
readprog() do
    Psize := t.read(0, Prog, PROG_SIZE);
    if (Psize >= PROG_SIZE)
        aw("program too big", 0);
end
```

Readrc() extracts a *raw* character from the input buffer and returns it. **Readc()** does the same, but folds characters to lower case first. It makes the T3X language case-insensitive.

```
readrc() do var c;
    c := Pp >= Psize-> ENDFILE: Prog::Pp;
    Pp := Pp+1;
    return c;
end

readc() do var c;
    c := readrc();
    return 'A' <= c /\ c <= 'Z'-> c-'A'+'a': c;
end
```

Readec() reads an *extended* character, i.e. a (raw) character or an escape sequence. Escape sequences are used to include otherwise unrepresentable characters in string literals or character constants. For instance, **\n** is used to include a newline character, and **\q** is used to include a double quote character. See figure 49 on page 131 for a full list of escape sequences.

```
readec() do var c;
```

```
    c := readrc();
    if (c \= '\\') return c;
    c := readrc();
    if (c = 'a') return '\a';
    if (c = 'b') return '\b';
    if (c = 'e') return '\e';
    if (c = 'f') return '\f';
    if (c = 'n') return '\n';
    if (c = 'q') return '"' | META;
    if (c = 'r') return '\r';
    if (c = 's') return '\s';
    if (c = 't') return '\t';
    if (c = 'v') return '\v';
    return c;
end
```

This function backs up to the previous character in the buffer.

```
reject() Pp := Pp-1;
```

The **skip()** function is the first step when processing the input program. It reads the program character by character and skips over white space and comments. It returns the first character that is neither a space character nor contained in a comment.

Remember: comments start with an exclamation point (**!**) and extend up to (and including) the end of the line.

The function treats **\r** as a space character, so it can also process input programs in DOS-format text files, where **\r\n** is used to separate lines.

```
skip() do var c;
    c := readc();
    while (1) do
        while ( c = ' ' \/ c = '\t' \/
                c = '\n' \/ c = '\r' )
        do
            if (c = '\n') Line := Line+1;
            c := readc();
        end
        if (c \= '!')
```

```
            return c;
        while (c \= '\n' /\ c \= ENDFILE)
            c := readc();
    end
end
```

Findkw() tests whether the text in the argument *s* is a keyword of the T3X language. If it is, it will return the token ID of the keyword and otherwise it will return 0.

The function first tests the first character of *s* in order to minimize the number of string comparisons. Hence it performs an average of 1 string comparison instead of an average of 8 (there are 16 keywords).

Non-optimizing compilers spend a significant part of their time in the scanning phase, so this optimization makes a lot of sense.

Note that **mod** is an operator that looks like a keyword. This will cause a special case later in the code (see page 53).

```
findkw(s) do
    if (s::0 = 'c') do
        if (str.equal(s, "const")) return KCONST;
        return 0;
    end
    if (s::0 = 'd') do
        if (str.equal(s, "do")) return KDO;
        if (str.equal(s, "decl")) return KDECL;
        return 0;
    end
    if (s::0 = 'e') do
        if (str.equal(s, "else")) return KELSE;
        if (str.equal(s, "end")) return KEND;
        return 0;
    end
    if (s::0 = 'f') do
        if (str.equal(s, "for")) return KFOR;
        return 0;
    end
    if (s::0 = 'h') do
```

```
        if (str.equal(s, "halt")) return KHALT;
        return 0;
    end
    if (s::0 = 'i') do
        if (str.equal(s, "if")) return KIF;
        if (str.equal(s, "ie")) return KIE;
        return 0;
    end
    if (s::0 = 'l') do
        if (str.equal(s, "leave")) return KLEAVE;
        if (str.equal(s, "loop")) return KLOOP;
        return 0;
    end
    if (s::0 = 'm') do
        if (str.equal(s, "mod")) return BINOP;
        return 0;
    end
    if (s::0 = 'r') do
        if (str.equal(s, "return")) return KRETURN;
        return 0;
    end
    if (s::0 = 's') do
        if (str.equal(s, "struct")) return KSTRUCT;
        return 0;
    end
    if (s::0 = 'v') do
        if (str.equal(s, "var")) return KVAR;
        return 0;
    end
    if (s::0 = 'w') do
        if (str.equal(s, "while")) return KWHILE;
        return 0;
    end
    return 0;
end
```

Scanop() scans an operator symbol and returns the corresponding token ID. The function uses the operator (**OPER**) structure (page 46) and the **Ops** table (see initialization, pg 101).

Scanning an operator symbol is done as follows:

- One character, c, is read from the source program and compared to the first character of each operator name in the **Ops** table, starting from the beginning of the table.
- When a match is found, another character is read and compared to the second character of each subsequent two-character operator name, if any.
- When a two-character match exists, it is returned.
- When no two-character match exists, but a single-character match was found, the single-character operator is returned.
- When no single-character match is found, an error is reported.

When **scanop()** finds an operator, it sets the **Oid** variable to the index of that operator in the **Ops** table.

Note that the order of the operators in the **Ops** table is import for the **scanop()** algorithm to work.

```
scanop(c) do var i, j;
    i := 0;
    j := 0;
    Oid := %1;
    while (Ops[i][OLEN] > 0) do
        ie (Ops[i][OLEN] > j) do
            if (Ops[i][ONAME]::j = c) do
                Oid := i;
                Str::j := c;
                c := readc();
                j := j+1;
            end
        end
        else do
            leave;
        end
        i := i+1;
    end
    if (Oid = %1) do
        Str::j := c;
```

```
        j := j+1;
        Str::j := 0;
        aw("unknown operator", Str);
    end
    Str::j := 0;
    reject();
    return Ops[Oid][OTOK];
end
```

Findop() locates the operator of the given name in the **Ops** table. Because this function does not process program input, only valid operator symbols are passed to it, and the case of an unknown operator should not happen.

```
findop(s) do var i;
    i := 0;
    while (Ops[i][OLEN] > 0) do
        if (str.equal(s, Ops[i][ONAME])) do
            Oid := i;
            return Oid;
        end
        i := i+1;
    end
    oops("operator not found", s);
end
```

The **symbolic()** functions tests whether the character c is a valid character for *starting* a symbol name. Subsequent characters of a symbol name may also be numeric.

```
symbolic(c)
    return alphabetic(c) \/ c = '_' \/ c = '.';
```

Scan() is the principal input function of the compiler. Each time it is called it extracts a token from the program buffer and returns its token ID. In addition it returns:

- the value of integer and character literals in **Val**
- the value of string literals in **Str**
- the textual representation of other tokens in **Str**
- the offsets of operators in the **Ops** table in **Oid**

When the end of the program buffer is reached, it returns the special **ENDFILE** token ID.

Scan() proceeds as follows: it first skips over white space characters and tests for the end of the input. It then determines the *class* of the following token by looking at its first character. The following tokens classes are recognized:

- symbol names, starting with a symbolic character
- integers, starting with a digit or a % sign
- characters, starting with a ' character
- strings, starting with a " character.
- unary and binary operators in the **Ops** table

Tokens belonging to a token class are distinguished by their *attributes*. For example, the tokens

```
314   %1   0
```

all belong to the class **INTEGER**, but their values in **Val** are different.

Note that not all operators in the **Ops** table belong to the **BINOP** (binary operator) or **UNOP** (unary operator) class. Many tokens have individual token IDs and do not belong to any class at all. Such tokens can be distinguished by their token IDs alone.

These include, for instance, all operators in **Ops** that have an **OTOK** value other then **BINOP** or **UNOP**, as well as punctuation characters, such as ';', '(', ')', etc.

All keywords (like **IF**, **DO**, **STRUCT**, etc), are individual tokens. They are distinguished from symbols by the **findkw()** function. This means that symbols cannot have names that are reserved for keywords.

The **MOD** keyword is a special case, because it represents a binary operator. When **findkw()** returns **BINOP**, the **scan()** function looks up the **Oid** of the operator.

```
scan() do var c, i, k, sgn;
    c := skip();
```

```
if (c = ENDFILE) do
    str.copy(Str, "end of file");
    return ENDFILE;
end
if (symbolic(c)) do
    i := 0;
    while (symbolic(c) \/ numeric(c)) do
        if (i >= TOKEN_LEN-1) do
            Str::i := 0;
            aw("symbol too long", Str);
        end
        Str::i := c;
        i := i+1;
        c := readc();
    end
    Str::i := 0;
    reject();
    k := findkw(Str);
    if (k \= 0) do
        if (k = BINOP) findop(Str);
        return k;
    end
    return SYMBOL;
end
if (numeric(c) \/ c = '%') do
    sgn := 1;
    i := 0;
    if (c = '%') do
        sgn := %1;
        c := readc();
        Str::i := c;
        i := i+1;
        if (\numeric(c))
            aw("missing digits after '%'",0);
    end
    Val := 0;
    while (numeric(c)) do
        if (i >= TOKEN_LEN-1) do
```

```
                Str::i := 0;
                aw("integer too long", Str);
            end
            Str::i := c;
            i := i+1;
            Val := Val * 10 + c - '0';
            c := readc();
        end
        Str::i := 0;
        reject();
        Val := Val * sgn;
        return INTEGER;
    end
    if (c = '\'') do
        Val := readec();
        if (readc() \= '\'')
            aw("missing ''' in character", 0);
        return INTEGER;
    end
    if (c = '"') do
        i := 0;
        c := readec();
        while (c \= '"' /\ c \= ENDFILE) do
            if (i >= TOKEN_LEN-1) do
                Str::i := 0;
                aw("string too long", Str);
            end
            Str::i := c & (META-1);
            i := i+1;
            c := readec();
        end
        Str::i := 0;
        return STRING;
    end
    return scanop(c);
end
```

Parser

The parser is the main component of this compiler. It reads tokens through the scanner and emits machine code though the code generator. The structure of the input program directs the control flow through the parser. This is why this approach is called *syntax-directed translation*.

The parser *analyzes* the structure of the token stream representing the input program. Its makes sure that all sentences of the input program are syntactically correct and reports errors otherwise. A *sentence* is a self-contained unit of a program, such as an *expression*, a *statement*, a *declaration*, etc. These units will be covered more in detail in this section.

The process of parsing is similar to the process of scanning, because the next input unit controls how the subsequent units are processed. For instance, when the scanner finds a double quote character, it will invoke a routine that scans a string literal. When the parser finds an **IF** keywords, it will invoke a routine that parses an **IF** statement.

There is, however, a difference in the level of abstraction. A scanner deals with simple, linear structures, while the parser handles recursive tree structures. For example, a string may not contain a string, but an **IF** statement may contain another **IF** statement, e.g.:

```
IF (a) IF (b) IF (c) DO END
```

Parser Prelude

The **MAXTBL** constant specifies the maximum size of a table (see page 73). **MAXLOOP** is the maximum number of nested loops and/or **LEAVE** and **LOOP** keywords per loop (see below and page 92 and following).

```
const   MAXTBL   = 128;
const   MAXLOOP  = 100;
```

The **Fun** variable indicates whether the parser is currently parsing a function body. It is used to distinguish function bodies from the main program (where **RETURN** is not allowed).

Loop0 indicates whether the parser is currently processing a loop context, i.e. the statement of a loop. It is *not* a flag! A value of -1 means that no loop context currently exists, a value of 0 indicates a **FOR** context, and a positive value indicates a **WHILE** context.

In a **WHILE** context, **Loop0** is the address where the loop begins, so **LOOP** simply generates a jump to that address.

The **Leaves** and **Loops** vectors and their corresponding pointers, **Lvp** and **Llp**, are used to collect addresses of forward jumps in loops. Such jumps are generated by **LEAVE** statements and **LOOP** statements in **FOR** loops. The collected addresses will be used for backpatching after compiling the statement of a loop.

```
var     Fun;
var     Loop0;
var     Leaves[MAXLOOP], Lvp;
var     Loops[MAXLOOP], Llp;
```

On many occasions the parser will *expect* a specific token or a token of a specific class. For example, when reading a **VAR** keyword (**KVAR** token), it will expect a token of the **SYMBOL** class to follow.

The parser keeps the ID of the current token in the variable **T**. The **expect()** function tests whether **T** equals the token ID *tok* and just returns if this test is positive. Otherwise, it prints an error message and aborts compilation.

```
expect(tok, s) do var b::100;
    if (tok = T) return;
    str.copy(b, s);
    str.append(b, " expected");
    aw(b, Str);
end
```

The following functions expect the following tokens, respectively: an equal sign (**=**), a semicolon (**;**), a left parenthesis (**(**), and a right parenthesis (**)**). All of these functions consume the expected

token in case it matches.

```
xeqsign() do
    if (T \= BINOP \/ Oid \= Equal_op)
        expect(BINOP, "'='");
    T := scan();
end

xsemi() do
    expect(SEMI, "';'");
    T := scan();
end

xlparen() do
    expect(LPAREN, "'('");
    T := scan();
end

xrparen() do
    expect(RPAREN, "')'");
    T := scan();
end
```

Xsymbol() expects a **SYMBOL** token, but does not consume it in case of success, because doing so would overwrite its **Str** attribute.

```
xsymbol() expect(SYMBOL, "symbol");
```

In general, the parser presented in this chapter will be a *recursive descent parser* (*RDP*), where each type of sentence is handled by one or multiple functions.

Whenever a function is called, the type of sentence to analyze is already known. For instance, the function analyzing an **IF** statement would only be called when an **IF** keyword was found in a statement context.

Because declarations may contain statements and statements may contain expressions, a parser function analyzing a declaration could eventually *descend* into the functions analyzing statements, which could, in turn, descend into the functions analyzing

expressions. This explains the *descent* part of the term “recursive descent parser”.

Certain sentences may contain instances of themselves, directly or indirectly. For instance, a function declaration contains a statement, which may be a compound statement, which may contain declarations. Or an expression may contain another expression in parentheses. This is why the RDP is called a *recursive* descent parser: the parsing functions may call each other.

Declaration Parser

The **constfac()** and **constval()** functions parse *constant values*, also called *cvalues*.

Constfac() expects a constant factor in the form of an integer or a symbol that as been defined by a **CONST** declaration. Of course, the “defined by **CONST**” part is impossible to assert on a syntactic level, because constants and variables are both **SYMBOL**s. This is why **constfac()** checks the attributes of the symbol by looking at the symbol table.

Constfac() returns the value of a constant factor.

```
constfac() do var v, y;
    if (T = INTEGER) do
        v := Val;
        T := scan();
        return v;
    end
    if (T = SYMBOL) do
        y := lookup(Str, CNST);
        T := scan();
        return y[SVALUE];
    end
    aw("constant value expected", Str);
end
```

Constval() is the principal constant expression parser. It accepts a constant factor $c1$ (by descending into **constfac()**) and when a **+** or ***** operator follows, it consumes it and expects

another constant factor, $c2$. It returns the sum or product of $c1$ and $c2$, respectively.

```
constfac → INTEGER | SYMBOL

constval → constfac
         | constfac * constfac
         | constfac + constfac
```

Figure 13: Constant Value Syntax Rules

The syntax of a sentence, like *constval*, can be described by *syntax rules*, like the ones depicted in figure 13. A valid sentence is created from such rules as follows:

The arrow means “can be written as”. So the first line in figure 13 can be read as “a *constfac* can be written as an **INTEGER** or a **SYMBOL**”, where the vertical bar indicates the logical OR.

In the case of *constval*, there are three valid sentences. Each of them starts with a *constfac*, the first one ends there, the second one adds ***** *constfac*, and the third one adds **+** *constfac*.

When a stream of tokens can be created by a set of syntax rules, the rules *match* the sentence, otherwise a syntax error occurred.

As can be seen in figure 13, sentences can be build on top of sentences: *constval* contains multiple instance of another sentence, *constfac*.

```
constval() do var v;
    v := constfac();
    ie (T = BINOP /\ Oid = Mul_op) do
        T := scan();
        v := v * constfac();
    end
    else if (T = BINOP /\ Oid = Add_op) do
        T := scan();
        v := v + constfac();
    end
    return v;
end
```

The syntax rules for *vardecl*, a declaration of variables, is given in figure 14. The *vardecl* rule says that a variable declaration is a **VAR** keyword, followed by a list of variables (*vars*), followed by a semicolon. To make things more readable, individual tokens will print as text rather than token ID, e.g. **SEMI** (semicolon) will print as **;** and **BYTEOP** will print as **::**.

```
vardecl → VAR vars ;

vars → SYMBOL
     | SYMBOL , vars
     | SYMBOL subscript
     | SYMBOL subscript , vars

subscript → [ constval ]
          | :: constval
```

Figure 14: Variable Declaration Syntax Rules

The *vars* (variables) rule says that *vars* can be a symbol, or a symbol with a subscript attached, or either of the above followed by a comma and more variables. A *subscript* can be a cvalue in square brackets or a byte operator followed by a cvalue.

Vardecl adds each symbol it encounters to the symbol table. Global symbols are assigned the next free data segment address (**Dp**) and local variables are assigned the next free offset in the current frame (**Lp**).

When a declared object is a vector, i.e. it has a form of **SYMBOL[cvalue]** or **SYMBOL::cvalue**, then **vardecl** sets the **VECT** flag for the symbol and remembers the size of the vector.

It then emits code to allocate the vector at run time. Note that *all* vectors are allocated dynamically on the *stack*, even global ones. Doing so allows us to get away without a *BSS* (block started by symbol) segment, which is normally used to store uninitialized data. Storing uninitialized vectors would increase the size of the executable file by sizes of the vectors, because it would store a corresponding sequence of zeros in the data segment. Adding a BSS segment would also complicate relocation.

For both, local and global vectors, a variable is allocated in addition to the vector itself and the address of the vector is stored in that variable, so taking the value of the variable will yield the address of the vector, allowing to write code like this:

```
DO VAR p, v[2];
     p := v;
     v[0] = 1;
     p[1] := v[0];
END
```

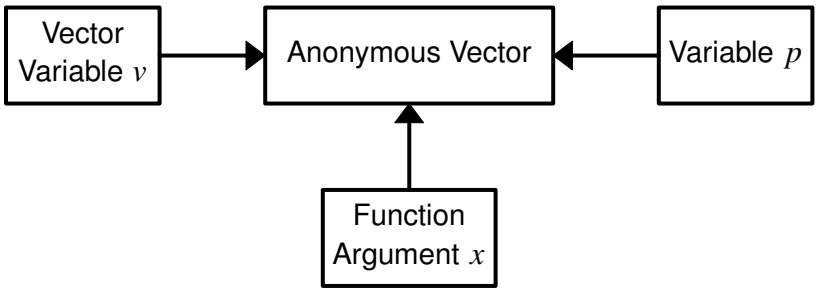


Figure 15: Anonymous Vector and References

Because vectors themselves are anonymous and vector variables are references to them, assigning the value of a vector variable to a variable will create another reference to the vector, so the variable can be used to access and manipulate the vector. This works even in function calls: by passing a vector to a function, the function argument will become a (reference to) a vector. See figure 15.

This approach simplifies vector handling a lot, because variables and vector are not different internally. The only difference between them is that no value can be assigned to vector variables (e.g. `v:=0;` would be illegal in the above program).

This method has been described by Richards in [RWS80], although BCPL did not have vector variables as a separate type, but explicitly assigned anonymous vectors to ordinary variables.

BTW, the allocation of vectors on the stack is the reason why the generated code jumps over functions. See figure 16. The parts of the program that will be run before the main program is entered are shown in gray.

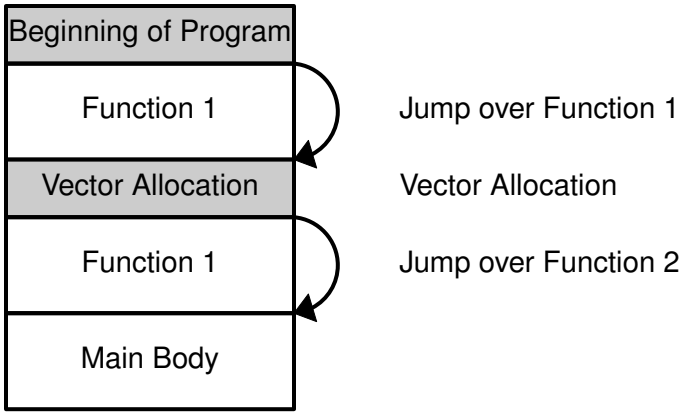


Figure 16: Merging Functions and Vector Allocation

Note that a global vector is a variable in the data segment that points to an anonymous vector on the stack and a local variable is a variable on the stack that points to an anonymous vector right next to itself.

The *glob* argument of **vardecl()** is set to **GLOB** if a variable declaration is taking place in the global context. Otherwise it is zero.

```
vardecl(glob) do var y, size;
    T := scan();
    while (1) do
        xsymbol();
        ie (glob & GLOBF)
            y := add(Str, glob, Dp);
        else
            y := add(Str, 0, Lp);
        T := scan();
        size := 1;
        ie (T = LBRACK) do
            T := scan();
```

```
            size := constval();
            if (size < 1)
                aw("invalid size", 0);
            y[SFLAGS] := y[SFLAGS] | VECT;
            expect(RBRACK, "']'");
            T := scan();
        end
        else if (T = BYTEOP) do
            T := scan();
            size := constval();
            if (size < 1)
                aw("invalid size", 0);
            size := (size + BPW-1) / BPW;
            y[SFLAGS] := y[SFLAGS] | VECT;
        end
        ie (glob & GLOBF) do
            if (y[SFLAGS] & VECT) do
                gen(CG_ALLOC, size*BPW);
                gen(CG_GLOBVEC, Dp);
            end
            dataw(0);
        end
        else do
            gen(CG_ALLOC, size*BPW);
            Lp := Lp - size*BPW;
            if (y[SFLAGS] & VECT) do
                gen(CG_LOCLVEC, 0);
                Lp := Lp - BPW;
            end
            y[SVALUE] := Lp;
        end
        if (T \= COMMA) leave;
        T := scan();
    end
    xsemi();
end
```

The syntax rules of constant declaration are simple, see figure 17.

```
constdecl → CONST consts ;

consts → SYMBOL = constval
       | SYMBOL = constval , consts
```

Figure 17: Constant Declaration Syntax Rules

The **constdecl()** function adds the given symbols to the symbol table and assigns the associated values to them. As in **vardecl()**, the *glob* argument is used to indicate a declaration in the global context.

```
constdecl(glob) do var y;
    T := scan();
    while (1) do
        xsymbol();
        y := add(Str, glob|CNST, 0);
        T := scan();
        xeqsign();
        y[SVALUE] := constval();
        if (T \= COMMA) leave;
        T := scan();
    end
    xsemi();
end
```

The **stcdecl()** function parses **STRUCT** declarations. The syntax rules for these are given in figure 18.

```
stcdecl → STRUCT SYMBOL = members ;

members → SYMBOL
        | SYMBOL , members
```

Figure 18: Constant Declaration Syntax Rules

A structure declaration is similar to a constant declaration, but the *members* of a structure get subsequent values starting at zero. The name of the structure itself will get the number of members as its value.

```
stcdecl(glob) do var y, i;
    T := scan();
    xsymbol();
    y := add(Str, glob|CNST, 0);
    T := scan();
    xeqsign();
    i := 0;
    while (1) do
        xsymbol();
        add(Str, glob|CNST, i);
        i := i+1;
        T := scan();
        if (T \= COMMA) leave;
        T := scan();
    end
    y[SVALUE] := i;
    xsemi();
end
```

A *forward declaration* is similar to a constant declaration, but sets the values of its symbols to zero and expects cvalues in parentheses after each symbol (see figure 19). It stores these cvalues in the *flags* fields of the symbol table entries; they form the types/arities of the declared functions.

```
fwddecl → DECL fwds ;

fwds → SYMBOL ( constval )
     | SYMBOL ( constval ) , fwds
```

Figure 19: Forward Declaration Syntax Rules

```
fwddecl() do var y, n;
    T := scan();
    while (1) do
        xsymbol();
        y := add(Str, GLOBF|FORW, 0);
        T := scan();
        xlparen();
        n := constval();
```

```
            if (n < 0) aw("invalid arity", 0);
            y[SFLAGS] := y[SFLAGS] | (n << 8);
            xrparen();
            if (T \= COMMA) leave;
            T := scan();
        end
        xsemi();
end
```

When a reference to a forward declaration is found, the compiler will generate a function call. Because the address of the function is not yet known, though, it will generate special address instead. When then first reference to a forward declaration appears, this address will be $R0 = 0$.

For all subsequent references to the same forward declaration, it will generate the address of the previous reference, e.g. $R1$ will point to $R0$ and $R2$ to $R1$, etc, leading to the structure shown in figure 20.

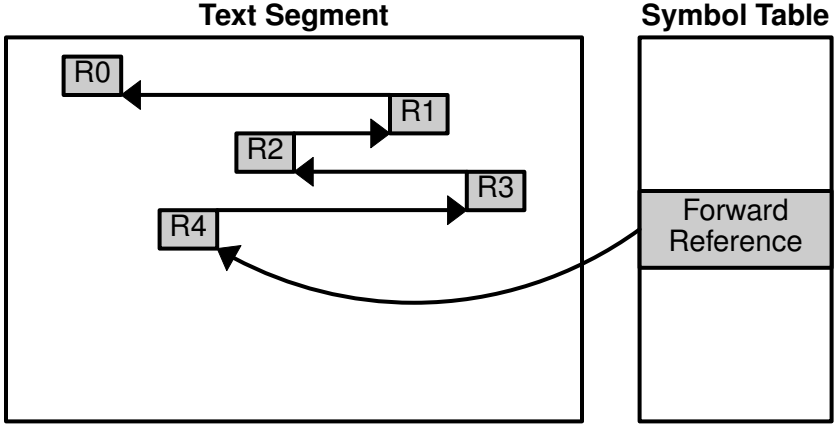


Figure 20: Chain of Forward References

The address of the most recently generated address field ($R4$ in the example) is kept in the *value* field of the symbol table entry of the forward declaration.

When a forward declaration is superseded by a function declaration, the forward references can be resolved by inserting the address of the function in each forward call. This is what

resolv_fwd() does.

The *loc* argument holds the address of the *last* generated forward call and *fn* is the address of the actual function. **Resolv_fwd()** traverses the linked list generated by calling the forward-declared function and replaces the links by the function address.

```
resolve_fwd(loc, fn) do var nloc;
    while (loc \= 0) do
        nloc := tfetch(loc);
        tpatch(loc, fn-loc-BPW);
        loc := nloc;
    end
end
```

Fundecl() parses a function declaration and generates code for the resulting function. It also creates code to jump over the function, as described previously (page 62).

The function is added to the symbol table as a global symbol of the type **FUNC**. Its arguments are added as local variables. Note that arguments are passed in left-to-right order in T3X, so the arguments are placed on the stack “upside down”, i.e. with the first argument buried deepest.

Argument	Allocated Address	Actual Address
1	F+8	F+$(4 \cdot n + 4)$
2	F+12	F+$(4 \cdot (n-1) + 4)$
...	...	...
n	F+$(4 \cdot n + 4)$	F+8

Figure 21: Fixing Argument Addresses

This is why **fundecl()** memorizes the first symbol table slot used by an argument in the *l_base* variable and then computes the actual addresses of the arguments in the loop after collecting the arguments. See figure 21.

When a pre-existing forward declaration with the same name as the current function exists, **fundecl()** makes sure that both have the same type (number of arguments). In this case, it does not

create a new symbol, but removes the **FORW** flag from the existing symbol table entry and sets the **FUNC** flag instead.

Finally, **fundecl()** generates code to create a function context, parses the statement (body) of the function, and emits code to delete the context and return a value of 0.

Before exiting, it also releases the symbol table slots and name list space allocated by function arguments. It does so by memorizing the **Yp** and **Np** pointers at the beginning and resetting them at the end. It also sets **Lp** to zero, because no compound statements may exist outside of a function (except for the main program, but after parsing that, the state of **Lp** does not matter any longer).

```
decl    compound(0), stmt(0);

fundecl() do
    var l_base, l_addr;
    var i, na, oyp, onp;
    var y;

    l_addr := 2*BPW;
    na := 0;
    gen(CG_JUMPFWD, 0);
    y := add(Str, GLOBF|FUNC, Tp);
    T := scan();
    xlparen();
    oyp := Yp;
    onp := Np;
    l_base := Yp;
    while (T = SYMBOL) do
        add(Str, 0, l_addr);
        l_addr := l_addr + BPW;
        na := na+1;
        T := scan();
        if (T \= COMMA) leave;
        T := scan();
    end
    for (i = l_base, Yp, SYM) do
        Syms[i+SVALUE] :=
```

```
            12+na*BPW - Syms[i+SVALUE];
    end
    if (y[SFLAGS] & FORW) do
        resolve_fwd(y[SVALUE], Tp);
        if (na \= y[SFLAGS] >> 8)
            aw("function does not match DECL",
               y[SNAME]);
        y[SFLAGS] := y[SFLAGS] & ~FORW;
        y[SFLAGS] := y[SFLAGS] | FUNC;
        y[SVALUE] := Tp;
    end
    xrparen();
    y[SFLAGS] := y[SFLAGS] | (na << 8);
    gen(CG_ENTER, 0);
    Fun := 1;
    stmt();
    Fun := 0;
    gen(CG_CLEAR, 0);
    gen(CG_EXIT, 0);
    gen(CG_RESOLV, 0);
    Yp := oyp;
    Np := onp;
    Lp := 0;
end
```

The **declaration()** function delegates handling of all kinds of declarations to the above functions. Note that it does not pass the *glob* argument on to **fwddecl()** or **fundecl()**, because these declarations can only appear in the global context. Therefore **declaration()** is never called with $glob \neq GLOB$ while the current token indicates a forward or function declaration.

```
declaration(glob)
    ie (T = KVAR)
        vardecl(glob);
    else ie (T = KCONST)
        constdecl(glob);
    else ie (T = KSTRUCT)
        stcdecl(glob);
    else ie (T = KDECL)
```

```
        fwddecl();
    else
        fundecl();
```

Expression Parser

An *expression* is anything that has a *value*. A value may be a numeric value in case of an integer, or the result of applying an operator to integers, or it may be an *address*, like the location of a string or a vector in memory.

The difference between an expression and a cvalue is that the value of an expression is not known at compile time, but will be computed only when the program containing the expression executes.

This section generates code that computes values at run time.

Note that expressions are inherently recursive structures. For instance, expressions may contain function calls and functions calls may contain expressions (as actual arguments).

```
fncall → SYMBOL ( arguments )
       | SYMBOL ( )

arguments → expr
          | expr , arguments
```

Figure 22: Function Call Syntax Rules

The **`fncall()`** function compiles function calls. The syntax rules for function calls are shown in figure 22. *Expr* denotes an expression, i.e. the totality of rules defined in this section. The *fn* argument of **`fncall()`** is the symbol table entry of the function to call.

`Fncall()` first compiles the expressions forming the actual arguments, pushing their values to the stack. It also type-checks the call, i.e. it makes sure the correct number of arguments is passed to the function. When the accumulator is loaded after compiling arguments, it spills it, making sure that *all* arguments are on the stack.

It then generates either a forward call (see page 67) or a function call, depending on whether *fn* is a function or a forward declaration. **`Fncall()`** also emits code to remove the arguments from the stack after the function returns. Finally, it sets the accumulator state to *active*, because functions return their values in the accumulator.

```
decl    expr(1);

fncall(fn) do var i;
    T := scan();
    if (fn = 0) aw("call of non-function", 0);
    i := 0;
    while (T \= RPAREN) do
        expr(0);
        i := i+1;
        if (T \= COMMA) leave;
        T := scan();
        if (T = RPAREN)
            aw("syntax error", Str);
    end
    if (i \= fn[SFLAGS] >> 8)
        aw("wrong number of arguments",
           fn[SNAME]);
    expect(RPAREN, "')'");
    T := scan();
    if (active())
        spill();
    ie (fn[SFLAGS] & FORW) do
        gen(CG_CALL, fn[SVALUE]);
        fn[SVALUE] := Tp-BPW;
    end
    else do
        gen(CG_CALL, fn[SVALUE]-Tp-5); ! TP-BPW+1
    end
    if (i \= 0) gen(CG_DEALLOC, i*BPW);
    activate();
end
```

Mkstring() places the characters of a string literal int the data segment and returns the (unrelocated) address of the first character of the literal. String literals are padded to occupy a multiple of **BPW** bytes.

```
mkstring(s) do var i, a, k;
    a := Dp;
    k := str.length(s);
    for (i=0, k+1)
        data(s::i);
    while (Dp mod BPW \= 0)
        data(0);
    return a;
end
```

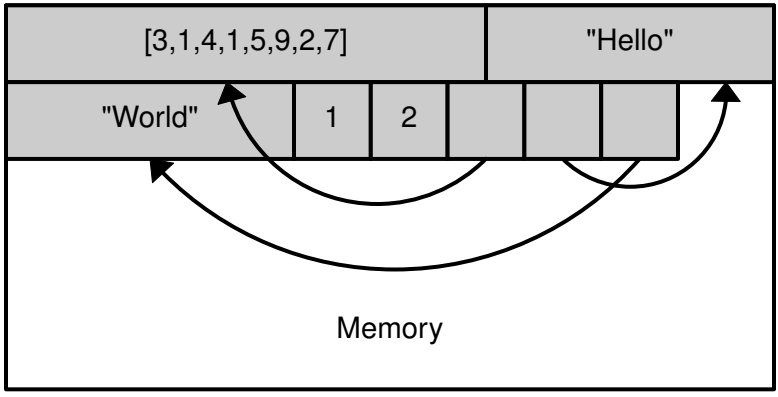
A table is a recursive vector literal, i.e. a set of cvalues, strings, and tables that is delimited by square brackets. See the introduction to T3X in the appendix for examples. The syntax rules of tables are shown in figure 23.

```
table → [ members ]

members → member | member , members

member → constval
       | STRING
       | table
       | ( exprs )

exprs → expr
      | expr , exprs
```

Figure 23: Table Syntax Rules

An element of a table that is itself a vector (like a table or a string) is called an embedded element or embedded member of the table. In order to compile a table, embedded elements are compiled first, and the addresses of the embedded elements are stored in an internal vector, together with the values of “trivial” (non-embedded) elements (cvalues).

The *tbl* vector is used to store values and addresses of table members and the *af* (address flag) vector is used to distinguish embedded elements from trivial elements. The layout of a nested table (a table with embedded elements) in memory is shown in figure 24.



[1, 2, [3,1,4,1,5,9,2,7], "Hello", "World"]

Figure 24: Table Layout in Memory

There is a loop at the end of **`mktable()`** that emits the table elements to the data segment and tags addresses of embedded elements for relocation.

Things are further complicated by the fact that T3X tables may contain *dynamic elements*, which are computed at run time and inserted into the table. For dynamic elements, **`mktable()`** emits code to compute their values and store them in the table. In this case, the corresponding *af* field will contain an address for backpatching the store instruction when the final location of the destination element is known.

(Yes, this means that “address *flag*” is a misnomer.)

`Mktable()` returns the address of the first table element. Because it recurses to compile embedded elements, the value *finally* returned by it will be the address of the first element of the outermost table.

```
mktable() do
    var n, i, a;
```

```
var tbl[MAXTBL], af[MAXTBL];
var dynamic;

T := scan();
dynamic := 0;
n := 0;
while (T \= RBRACK) do
    if (n >= MAXTBL)
        aw("table too big", 0);
    ie (T = LPAREN /\ \dynamic) do
        T := scan();
        dynamic := 1;
        loop;
    end
    else ie (dynamic) do
        expr(1);
        gen(CG_STGLOB, 0);
        tbl[n] := 0;
        af[n] := Tp-BPW;
        n := n+1;
        if (T = RPAREN) do
            T := scan();
            dynamic := 0;
        end
    end
    else ie (T = INTEGER \/ T = SYMBOL) do
        tbl[n] := constval();
        af[n] := 0;
        n := n+1;
    end
    else ie (T = STRING) do
        tbl[n] := mkstring(Str);
        af[n] := 1;
        n := n+1;
        T := scan();
    end
    else ie (T = LBRACK) do
        tbl[n] := mktable();
```

```
            af[n] := 1;
            n := n+1;
        end
        else do
            aw("invalid table element", Str);
        end
        if (T \= COMMA) leave;
        T := scan();
        if (T = RBRACK)
            aw("syntax error", Str);
    end
    if (dynamic)
        aw("missing ')' in dynamic table", 0);
    expect(RBRACK, "']'");
    if (n = 0) aw("empty table", 0);
    T := scan();
    a := Dp;
    for (i=0, n) do
        dataw(tbl[i]);
        ie (af[i] = 1) do
            tag('d');
        end
        else if (af[i] > 1) do
            tpatch(af[i], Dp-4);
        end
    end
    return a;
end
```

Load() and **store()** are shortcuts for loading values into the accumulator and storing values from the accumulator, no matter if the given symbol denotes a local or global variable.

```
load(y)
    ie (y[SFLAGS] & GLOBF)
        gen(CG_LDGLOB, y[SVALUE]);
    else
        gen(CG_LDLOCL, y[SVALUE]);

store(y)
```

```
    ie (y[SFLAGS] & GLOBF)
        gen(CG_STGLOB, y[SVALUE]);
    else
        gen(CG_STLOCL, y[SVALUE]);
```

The meaning of the expression

v[1][2]

depends on its context. On the left side of an assignment, it denotes the address of the second element of the first element of v, while in an expression, it denotes the value stored in the same element. The former is called an *lvalue* (*because* it appears on the left side of an assignment) and the latter is called a *value* (or, rarely, an “rvalue”).

The **address()** function generates code for both, values and lvalues, depending on the context in which the sentence appears. The context is indicated by the lv (lvalue) flag passed to it.

The bp (byte pointer) argument is the address of a variable that will be set when the generated address is the address of a byte. It will be cleared when it is the address of a machine word. It may be considered to be an second return value of **address()**.

The actual return value of the function is the symbol table slot of the addressed symbol, or 0 when the generated code addresses an anonymous vector element.

The syntax rules for addresses are given in figure 25. Note that the right side of a byte operator is a factor and not an expression, so **v::i+1** means **(v::i)+1** and not **v::(i+1)**.

```
address → SYMBOL
        | SYMBOL subscripts

subscripts → [ expr ]
           | [ expr ] subscripts
           | :: factor
```

Figure 25: Address Syntax Rules

`Address()` also does some type checking: it reports attempts to

- take the address of a constant
- take the address of a function (or forward declaration)
- compute a member of a constant (invalid subscript)
- compute a member of a function

Another interesting question is when to spill the accumulator and when to load the value stored in a location. In a value context, the expression **`v[1][2]`** would generate this code:

```
LDGLOB v LDVAL 1 INDEX DEREF LDVAL 2 INDEX DEREF
```

while in an lvalue context, the last **`DEREF`** would be omitted, leaving the address of the vector element in the accumulator instead of its value. Of course, the accumulator has to be spilled after **`LDGLOB v`** or **`LDVAL 1`** would overwrite it.

On the other hand, the expression **`x`** would generate a load instruction in a value context and no code at all in an lvalue context. Subsequently, the accumulator must not be spilled in the second case.

The **`address()`** function takes care of all of these cases. It returns the symbol table entry of an lvalue, if it is known, or zero if an lvalue is anonymous. In the latter case, code will be generated to load the address of the anonymous element into the accumulator.

Note that lv is actually a ternary flag. When $lv = 2$, this indicates the use of the address operator `@`, which may not be applied to functions. Functions may occur in an ordinary lvalue context, though, because the first token of a function call is the same as the first token of an assignment, namely a symbol.

Finally, consider the expression

```
b[1]::2
```

It would generate almost the same code as the expression **`v[1][2]`**, with only two differences: (1) the last generated instruction would be **`INDXB`** or **`DREFB`** (byte index, byte

dereference) instead of **INDEX** or **DEREF**, and, (2) the location pointed to by *bp* would be set to 1, indicating a byte address in the accumulator.

```
decl    factor(0);

address(lv, bp) do var y;
    y := lookup(Str, 0);
    T := scan();
    ie (y[SFLAGS] & CNST) do
        if (lv > 0)
            aw("invalid location", y[SNAME]);
        spill();
        gen(CG_LDVAL, y[SVALUE]);
    end
    else ie (y[SFLAGS] & (FUNC|FORW)) do
        if (lv = 2)
            aw("invalid location", y[SNAME]);
    end
    else if (lv = 0 \/ T = LBRACK \/ T = BYTEOP)
    do
        spill();
        load(y);
    end
    if (T = LBRACK \/ T = BYTEOP)
        if (y[SFLAGS] & (FUNC|FORW|CNST))
            aw("bad subscript", y[SNAME]);
    while (T = LBRACK) do
        T := scan();
        bp[0] := 0;
        expr(0);
        expect(RBRACK, "']'");
        T := scan();
        y := 0;
        gen(CG_INDEX, 0);
        if (lv = 0 \/ T = LBRACK  \/ T = BYTEOP)
            gen(CG_DEREF, 0);
    end
    if (T = BYTEOP) do
```

```
        T := scan();
        bp[0] := 1;
        factor();
        y := 0;
        gen(CG_INDXB, 0);
        if (lv = 0) gen(CG_DREFB, 0);
    end
    return y;
end
```

A *factor* is the smallest component of an expression. A single factor is a complete expression, but most expressions are comprised of multiple factors combined by operators. The syntax rules for factors are listed in figure 26.

```
factor → address
        | SYMBOL ()
        | INTEGER
        | STRING
        | table
        | @ address
        | - factor
        | UNOP factor
        | ( expr )
```

Figure 26: Factor Syntax Rules

The function call case **`SYMBOL()`** is handled below by parsing an address, which theoretically allows constructs like **`x[1]()`**, but this case will be rejected in **`fncall()`**, because it does not generate a valid symbol table entry for the function to call.

The case **`- factor`** is distinguished from the case **`UNOP factor`**, because the **`-`** operator is actually a **`BINOP`**. It just has a special meaning when it appears in a unary operator context.

```
factor() do var y, op, b;
    ie (T = INTEGER) do
        spill();
        gen(CG_LDVAL, Val);
        T := scan();
```

```
end
else ie (T = SYMBOL) do
    y := address(0, @b);
    if (T = LPAREN) fncall(y);
end
else ie (T = STRING) do
    spill();
    gen(CG_LDADDR, mkstring(Str));
    T := scan();
end
else ie (T = LBRACK) do
    spill();
    gen(CG_LDADDR, mktable());
end
else ie (T = ADDROF) do
    T := scan();
    y := address(2, @b);
    ie (y = 0) do
        ;
    end
    else ie (y[SFLAGS] & GLOBF) do
        spill();
        gen(CG_LDADDR, y[SVALUE]);
    end
    else do
        spill();
        gen(CG_LDLREF, y[SVALUE]);
    end
end
else ie (T = BINOP) do
    if (Oid \= Minus_op)
        aw("syntax error", Str);
    T := scan();
    factor();
    gen(CG_NEG, 0);
end
else ie (T = UNOP) do
    op := Oid;
```

```
        T := scan();
        factor();
        gen(Ops[op][OCODE], 0);
    end
    else ie (T = LPAREN) do
        T := scan();
        expr(0);
        xrparen();
    end
    else do
        aw("syntax error", Str);
    end
end
```

The syntax rules for all arithmetic binary operators (**BINOP**s) are simple. They are shown in figure 27.

```
arith → factor
      | factor BINOP arith
```

Figure 27: Arithmetic Operation Syntax Rules

Note that these rules are, strictly speaking, wrong, because they do not consider precedence or associativity.

Associativity is the direction in which equal operations are grouped. For example, the "minus" operation groups to the left in mathematics, so $a-b-c$ equals $(a-b)-c$. The "power" (^) operation, however, associates to the right, so $a\hat{}b\hat{}c$ equals $a\hat{}(b\hat{}c)$. All T3X operators except for `::` associate to the *left*.

The byte operator `::` is a subscript delivering a byte value, so it has to associate to the right: $v::x::y = v::(x::y)$, because $(v::x)::y$ would create a reference to an invalid (byte-sized) address.

Precedence specifies which operations will be computed first in expressions involving *different* operators. An example for precedence would be the "multiplication before addition" rule from mathematics: $a+b\cdot c+d$ equals $a+(b\cdot c)+d$.

The precedence values of the different T3X operators are listed in the **OPREC** field of the **Ops** vector, where a higher value indicates stronger precedence. Operators with stronger precedence will always be computed first.

The **arith()** and **emitop()** functions together implement an algorithm known as *operator precedence parsing* or *falling precedence parsing*. [NMH96] This algorithm handles multiple levels of precedence in a single loop instead of descending into each level as an ordinary RDP parser would do it. Technically speaking, the parser presented here is a composition of an RDP parser and a *bottom-up parser*.

Expression	S/R	Stack	Emitted Code
a*b=c+d*e			a
*b=c+d*e	S	*	a
b=c+d*e		*	a b
=c+d*e	R,S	=	a b *
c+d*e		=	a b * c
+d*e	S	= +	a b * c
d*e		= +	a b * c d
*e	S	= + *	a b * c d
e		= + *	a b * c d e
	R	= +	a b * c d e *
	R	=	a b * c d e * +
	R		a b * c d e * + =

Figure 28: Precedence Parsing

The bottom-up parsing algorithm used by the **arith()** function works as follows:

It collects operands (factors) and operators, where the code for all operands is emitted immediately, while each operator is pushed to a stack (stk). In each iteration, the function checks whether the current operator has a precedence that is lower than or equal to the precedence of the operator on the top of the stack. If so, the

operator is removed from the stack and the corresponding code is emitted.

At the end of the algorithm, when no more **BINOP**s are found in the token stream, all remaining operators in the stack are emitted. A sample operation is depicted in figure 28.

In figure 28, the order of precedence is *****, **+**, **=**. So code is emitted when an equal sign follows the multiplication star. Then all subsequent operators are pushed to the stack, because the precedence of operators grows steadily. Only at the end of the algorithm all remaining operands are flushed from the stack.

The resulting output in the “emitted code” column is essentially a virtual stack machine program that can easily be translated to VSM instructions from page 30.

BTW: Not entirely incidentally, pushing an operator to the stack in **arith()** is equivalent to the *shift* operation of an LR parser and popping an operator off the stack and emitting it is the same as a *reduce* operation. This is indicated by the S/R column of figure 28.

```
emitop(stk, p) do
    gen(Ops[stk[p-1]][OCODE], 0);
    return p-1;
end

arith() do var stk[10], p;
    factor();
    p := 0;
    while (T = BINOP) do
        while (p /\ Ops[Oid][OPREC]
                    <= Ops[stk[p-1]][OPREC])
            p := emitop(stk, p);
        stk[p] := Oid;
        p := p+1;
        T := scan();
        factor();
    end
    while (p > 0)
        p := emitop(stk, p);
```

```
end
```

Conjn() and **disjn()** are essentially the same procedure, which will be explained here by the example of **conjn()**. The syntax rules implemented by these functions are given in figure 29.

```
conjn → arith
      | arith /\ conjn

disjn → conjn
      | conjn \/ disjn
```

Figure 29: Conjunction and Disjunction Syntax Rules

The **conjn()** function implements the logical *conjuction* or *logical AND*. T3X implements the logical AND as a short-circuit operation. That is, in the expression **a/\b**, b will never be computed, if a is false. The implementation of the short-circuit logical AND is shown in figure 30.

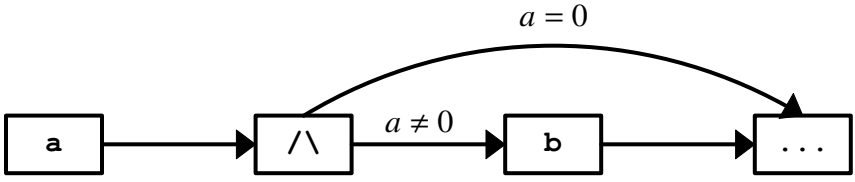


Figure 30: Short-Circuit Logical AND

The conjunction operator performs a jump over its second operand, if its first operand (in the accumulator) is zero. So after computing the operator, the accumulator holds 0 if $a = 0$ and b if $a \neq 0$.

The **conjn()** function uses a little optimization which is illustrated in figure 31. The upper part of the figure shows how a chain of conjunction operators is interpreted. If a is false, a jump to the end of **a/\b** is performed, which is exactly the operator between b and c. Because the accumulator is zero at this point, control is immediately transfered to the end of **b/\c**. This works for chains of any length, but in the worst case, n branches have to be performed for a chain of n conjunction operators.

Because we know that all of these branches will eventually reach the end of the entire chain, we can redirect them to the end of the chain from the beginning. This is illustrated in the lower part of figure 31.

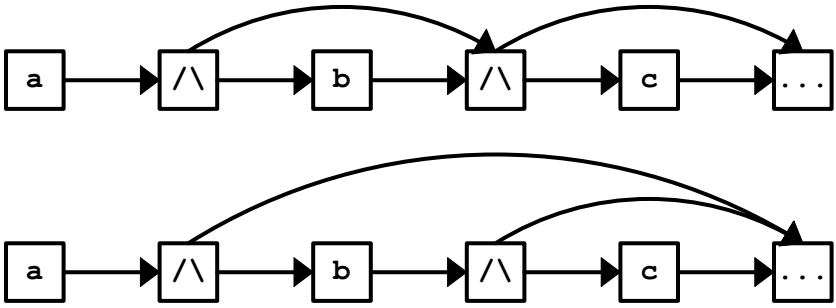


Figure 31: Short-Circuit Optimization

`Conjn()` resolves the n branches of a chain of n operators in the loop at the end of the function.

The **`disjn()`** function is basically the same as **`conjn()`**, only with the condition of the branch reversed: it jumps to the end of the chain, if the accumulator is non-zero (true). If you swap the conditions $a = 0$ and $a \neq 0$ in figure 30, this is how the `\/` operator works.

```
conjn() do var n;
    arith();
    n := 0;
    while (T = CONJ) do
        T := scan();
        gen(CG_JMPFALSE, 0);
        clear();
        arith();
        n := n+1;
    end
    while (n > 0) do
        gen(CG_RESOLV, 0);
        n := n-1;
    end
end
```

```
disjn() do var n;
    conjn();
    n := 0;
    while (T = DISJ) do
        T := scan();
        gen(CG_JMPTRUE, 0);
        clear();
        conjn();
        n := n+1;
    end
    while (n > 0) do
        gen(CG_RESOLV, 0);
        n := n-1;
    end
end
```

The *conditional operator* of T3X is the only ternary operator of the language. It is a generalization of the short-circuit logical operators described above:

```
a /\ b   =   \a -> 0 : b
a \/ b   =    a -> a : b
```

The operator may also be thought of as an in-expression if-then-else construct: if a in **a->b:c** is true, it delivers b and else c. The syntax rules of the conditional operator are specified in figure 32.

```
expr → disjn
     | disjn -> expr : expr
```

Figure 32: Conditional Operator Syntax Rules

The **->** part of the operator can be thought of as a “jump on false” to the c part of **a->b:c** and the **:** part of the operator can be thought of as an unconditional jump to the end of the operation. This implementation leads to the typical *jump around jump* pattern of if-then-else constructs. It is depicted in figure 33.

To generate a jump-around-jump, the **expr()** function first generates a jump-on-false (**JMPFALSE**) and then an unconditional forward jump (**JUMPFWD**), which both leave their addresses on the

stack for backpatching. It then swaps the top stack entries before resolving the jumps, leading to the overlapping branches in figure 33.

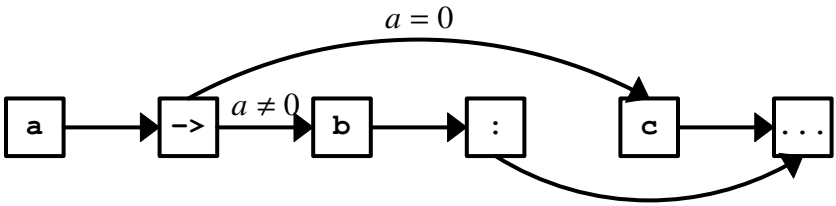


Figure 33: Conditional Operator Control Flow

The clr argument of **expr()** clears the accumulator state before parsing an expression. Sometimes it is necessary to assume that the accumulator is unused when starting to analyze an expression.

For example, when parsing the expression **a\/b**, we have to assume that the accumulator is clear when analyzing b, because otherwise a would be spilled at the beginning of b, which is not the right thing to do here, because b is intended to overwrite the value in the accumulator.

For the same reason **expr()** calls itself with $clr = 1$.

```
expr(clr) do
    if (clr) clear();
    disjn();
    if (T = COND) do
        T := scan();
        gen(CG_JMPFALSE, 0);
        expr(1);
        expect(COLON, "':'");
        T := scan();
        gen(CG_JUMPFWD, 0);
        swap();
        gen(CG_RESOLV, 0);
        expr(1);
        gen(CG_RESOLV, 0);
    end
end
```

Statement Parser

The *statement* is the fundamental building block of a procedural program. While an *expression* has a value, a statement *does something*. For instance, a **RETURN** statement ends interpretation of a function and returns a value, an assignment changes the value of a variable or vector element, and a loop repeats a part of a program.

Many statements operate on other statements. For instance, the **IF** statement runs another statement conditionally or **WHILE** repeats another statement. So statements, like expressions, are inherently recursive structures.

```
halt_stmt → HALT constval ;
```

Figure 34: HALT Statement Syntax Rule

One of the simplest statements is the **HALT** statement, which just has a cvalue as its argument and, like all statements, is terminated by a semicolon. See figure 34 for its syntax rule. It is implemented by **halt_stmt()**, below.

```
halt_stmt() do
    T := scan();
    gen(CG_HALT, constval());
    xsemi();
end
```

The **RETURN** statement has two forms: with and without a value to return. When no value is specified, zero is returned. The value is returned in the accumulator. The syntax rules of **RETURN** are shown in figure 35.

```
return_stmt → RETURN ;
            | RETURN expr ;
```

Figure 35: RETURN Statement Syntax Rules

The **return_stmt()** function, which implements the **RETURN** statement, also makes sure that the statement appears in a function context, because it is not allowed in the main program.

If the function containing the statement has any local variables, **return_stmt()** generates code to deallocate the space used by them before returning.

```
return_stmt() do
    T := scan();
    if (Fun = 0)
        aw("can't return from main body", 0);
    ie (T = SEMI)
        gen(CG_CLEAR, 0);
    else
        expr(1);
    if (Lp \= 0) do
        gen(CG_DEALLOC, -Lp);
    end
    gen(CG_EXIT, 0);
    xsemi();
end
```

The syntax rules for the **IF** and **IE/ELSE** statements can be found in figure 36. Note that no terminating semicolon is included in either rule, because it will be supplied by the *stmt* part of each rule.

```
if_stmt → IF ( expr ) stmt
        | IE ( expr ) stmt ELSE stmt
```

Figure 36: IF Statement Syntax Rules

The **IF** statement **IF (a) b;** is the statement equivalent to the expression **a/\b**, i.e. it executes b only if a is true. Hence the control flow diagram of **IF**, as shown in figure 37, looks similar to the one of the short-circuit AND operator (figure 30). A similar observation can be made regarding the conditional operator (figure 33) and the **IE**/**ELSE** statement (also figure 37).

When compiling **IE**/**ELSE**, the **if_stmt()** function uses the same jump-around-jump method as the **expr()** function implementing the conditional operator.

When **if_stmt()** is called with $alt = 1$, it expects an “alternative” statement, i.e. it compiles **IE**/**ELSE**, otherwise it compiles **IF**.

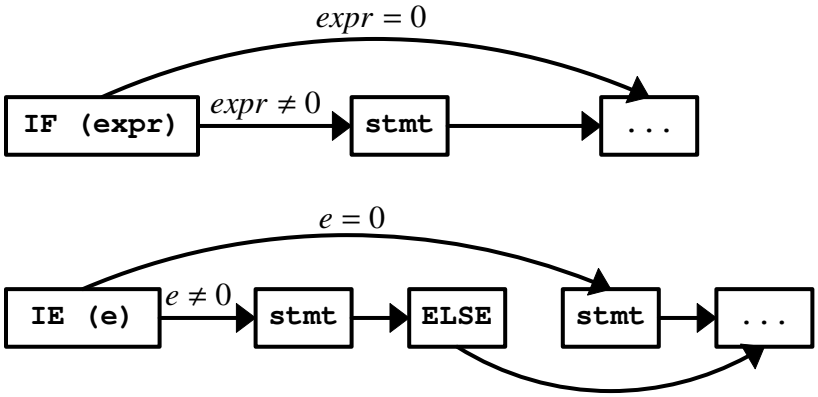


Figure 37: IF/IE Statement Control Flow

```
if_stmt(alt) do
    T := scan();
    xlparen();
    expr(1);
    gen(CG_JMPFALSE, 0);
    xrparen();
    stmt();
    if (alt) do
        gen(CG_JUMPFWD, 0);
        swap();
        gen(CG_RESOLV, 0);
        expect(KELSE, "ELSE");
        T := scan();
        stmt();
    end
    gen(CG_RESOLV, 0);
end
```

The **WHILE** loop, as depicted in figure 38, has structure similar to the **IE** statement. It also uses a jump around jump, but the second jump, after the statement, goes back to the beginning of the loop.

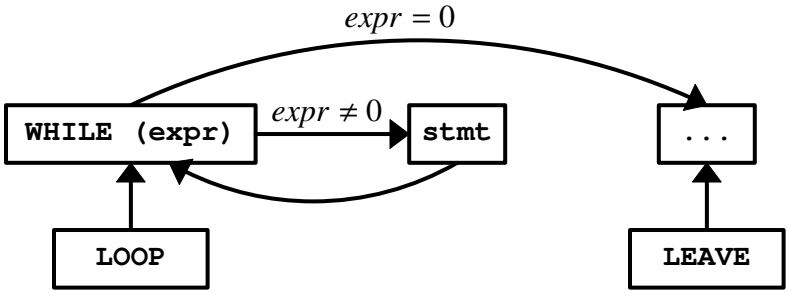


Figure 38: WHILE Statement Control Flow

In a **WHILE** context, the **LOOP** statement jumps to the point where *expr* is tested. This is a backward jump that can be generated immediately. The **LEAVE** statement, though, generates a forward jump, which has to be backpatched at the end of the loop.

While_stmt(), which implements the **WHILE** statement, stores the address for re-entering the loop in **Loop0** and the first **Leaves** slot used by it in *olv*. It restores **Loop0** and **Lvp** at the end, so loops can be nested.

while_stmt → WHILE (expr) stmt

Figure 39: WHILE Statement Syntax Rule

The syntax rule for **WHILE** can be found in figure 39.

```
while_stmt() do var olp, olv;
    T := scan();
    olp := Loop0;
    olv := Lvp;
    gen(CG_MARK, 0);
    Loop0 := tos();
    xlparen();
    expr(1);
    xrparen();
    gen(CG_JMPFALSE, 0);
    stmt();
    swap();
    gen(CG_JUMPBACK, 0);
    gen(CG_RESOLV, 0);
```

```
    while (Lvp > olv) do
        push(Leaves[Lvp-1]);
        gen(CG_RESOLV, 0);
        Lvp := Lvp-1;
    end
    Loop0 := olp;
end
```

Figure 40 shows the syntax rules for the **FOR** statement.

```
for_stmt → FOR ( expr , expr , constval ) stmt
         | FOR ( expr , expr ) stmt
```

Figure 40: FOR Statement Syntax Rules

The compiled structure of the **FOR (v=x1, x2, c) stmt** construct is a bit more complicated, as can be seen in figure 41. The initialization part on the left is run once and then the exit condition is tested. It depends on the sign of the third parameter of **FOR**. When the exit condition does not hold, the statement is run, and *then* the increment part of the statement is executed.

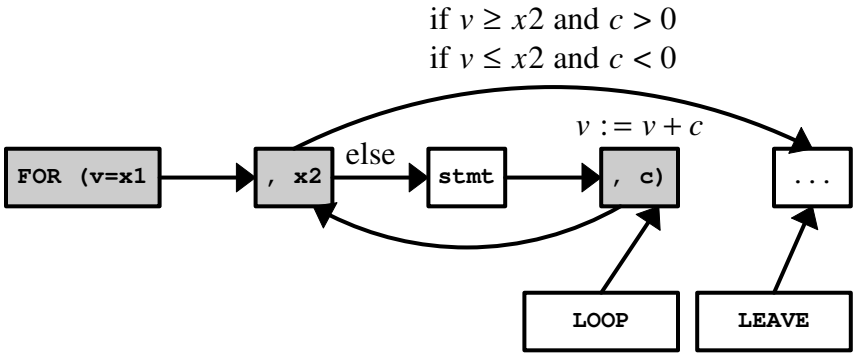


Figure 41: FOR Statement Control Flow

Therefore, the head of the **FOR** construct (in gray) has been spread around the statement in figure 41. After running the statement, the variable v is incremented and then the loop is repeated by jumping back to the test part.

The structure of **FOR** is similar to that of **WHILE**, with the exception of the additional increment part after the statement. This increment part is also the point where the loop is restarted by a **LOOP** statement. Because the statement precedes the increment, **LOOP** is a *forward jump* in **FOR**.

The **for_stmt()** function, which implements the **FOR** loop, keeps the **LEAVE** exit points in the **Leaves** vector and the **LOOP** entry points in the **Loops** vector. Like **while_stmt()**, it memorizes the outer loop context in local variables (*oll*, *olp*, *olv*) and restores it when done.

```
for_stmt() do
    var y;
    var step;
    var oll, olp, olv;
    var test;

    T := scan();
    oll := Llp;
    olv := Lvp;
    olp := Loop0;
    Loop0 := 0;
    xlparen();
    xsymbol();
    y := lookup(Str, 0);
    if (y[SFLAGS] & (CNST|FUNC|FORW))
        aw("unexpected type", y[SNAME]);
    T := scan();
    xeqsign();
    expr(1);
    store(y);
    expect(COMMA, "','");
    T := scan();
    gen(CG_MARK, 0);
    test := tos();
    load(y);
    expr(0);
    ie (T = COMMA) do
        T := scan();
```

```
        step := constval();
    end
    else do
        step := 1;
    end
    gen(step<0-> CG_FORDOWN: CG_FOR, 0);
    xrparen();
    stmt();
    while (Llp > oll) do
        push(Loops[Llp-1]);
        gen(CG_RESOLV, 0);
        Llp := Llp-1;
    end
    ie (y[SFLAGS] & GLOBF)
        gen(CG_INCGLOB, y[SVALUE]);
    else
        gen(CG_INCLOCL, y[SVALUE]);
    gen(CG_WORD, step);
    swap();
    gen(CG_JUMPBACK, 0);
    gen(CG_RESOLV, 0);
    while (Lvp > olv) do
        push(Leaves[Lvp-1]);
        gen(CG_RESOLV, 0);
        Lvp := Lvp-1;
    end
    Loop0 := olp;
end
```

The **LEAVE** and **LOOP** syntax rules are very simple, see figure 42. They are implemented by the **leave_stmt()** and **loop_stmt()** functions. Both of the functions check whether the corresponding keyword appears in a loop context.

```
leave_stmt → LEAVE ;
loop_stmt → LOOP ;
```

Figure 42: LEAVE and LOOP Syntax Rules

Leave_stmt() always creates a forward jump, because its destination cannot be known when the loop exit is generated. **Loop_stmt()** creates a backward jump in **WHILE** and a forward jump to the increment part in **FOR**.

```
leave_stmt() do
    T := scan();
    if (Loop0 < 0)
        aw("LEAVE not in loop context", 0);
    xsemi();
    if (Lvp >= MAXLOOP)
        aw("too many LEAVEs", 0);
    gen(CG_JUMPFWD, 0);
    Leaves[Lvp] := pop();
    Lvp := Lvp+1;
end

loop_stmt() do
    T := scan();
    if (Loop0 < 0)
        aw("LOOP not in loop context", 0);
    xsemi();
    ie (Loop0 > 0) do
        push(Loop0);
        gen(CG_JUMPBACK, 0);
    end
    else do
        if (Llp >= MAXLOOP)
            aw("too many LOOPs", 0);
        gen(CG_JUMPFWD, 0);
        Loops[Llp] := pop();
        Llp := Llp+1;
    end
end
```

The **asg_or_call()** function parses both assignments and function call statements, because they both begin with a symbol. The corresponding syntax rules can be seen in figure 43. A function call statement looks like a function call in an expression (page 80), but with a terminating semicolon attached.

```
assignment → address := expr ;

asg_or_call → assigment
            | fncall ;
```

Figure 43: Assignment and Function Call Syntax Rules

The assignment operator uses the symbol table slot returned by **address()**. When a valid slot is returned, a store instruction is generated for the associated symbol, otherwise the indirect address in the accumulator is used. This happens when referencing anonymous vectors (see page 78).

```
asg_or_call() do var y, b;
    clear();
    y := address(1, @b);
    ie (T = LPAREN) do
        fncall(y);
    end
    else ie (T = ASSIGN) do
        T := scan();
        expr(0);
        ie (y = 0)
            gen(b-> CG_STINDB: CG_STINDR, 0);
        else ie (y[SFLAGS] & (FUNC|FORW|CNST|VECT))
            aw("bad location", y[SNAME]);
        else
            store(y);
    end
    else do
        aw("syntax error", Str);
    end
    xsemi();
end
```

The syntax rules for the *statement* just lists the disjunction of all of the statement parsers described in this section, so the **stmt()** function implementing these rules just delegates analysis to those parsers.

```
stmt()
    ie (T = KFOR)
        for_stmt();
    else ie (T = KHALT)
        halt_stmt();
    else ie (T = KIE)
        if_stmt(1);
    else ie (T = KIF)
        if_stmt(0);
    else if (T = KELSE) do
        aw("ELSE without IE", 0);
    else ie (T = KLEAVE)
        leave_stmt();
    else ie (T = KLOOP)
        loop_stmt();
    else ie (T = KRETURN)
        return_stmt();
    else ie (T = KWHILE)
        while_stmt();
    else ie (T = KDO)
        compound();
    else ie (T = SYMBOL)
        asg_or_call();
    else ie (T = SEMI)
        T := scan();
    else
        expect(%1, "statement");
```

The *compound statement* or *statement block* is a set of statements and optional declarations that is delimited by the **DO** and **END** keywords. Its complete syntax rules can be seen in figure 44. While, formally, both the statements and the declarations are optional, the case with declarations only is very uncommon, because it essentially makes the statement a null statement, i.e. a statement that “does nothing”.

Declarations local to compound statements are limited to variables, constants, and structures; no local forward declarations or nested functions are allowed.

```
local_decl → vardecl | constdecl | stcdecl

local_decls → local_decl
            | local_decl local_decls

statments → stmt
          | stmt statements

compound_stmt → DO END
              | DO local_decls END
              | DO statements END
              | DO local_decls statements END
```

Figure 44: Compound Statement Syntax Rules

The **compound()** function that implements compound statements, emits code to deallocate all local variables of the statement at the end. It also releases memory allocated by symbol table slots and name list entries at compile time, by resetting their free space pointers to the values they had before parsing the compound statement.

```
compound() do var oyp, olp, onp;
    T := scan();
    oyp := Yp;
    onp := Np;
    olp := Lp;
    while (T = KVAR \/ T = KCONST \/ T = KSTRUCT)
        declaration(0);
    while (T \= KEND)
        stmt();
    T := scan();
    if (olp-Lp \= 0)
        gen(CG_DEALLOC, olp-Lp);
    Yp := oyp;
    Np := onp;
    Lp := olp;
end
```

Program Parser

The syntax rules for a complete T3X program are shown in figure 45. The **program()** function, which parses a full program, collects global declarations, expects one compound statement, and also makes sure that no characters are following after the compound statement forming the main body of the program.

```
global_decl → vardecl
            | constdecl
            | stcdecl
            | fwddecl
            | fundecl

global_decls → global_decl
             | global_decl global_decls

program → global_decls compound
        | compound
```

Figure 45: T3X Program Syntax Rules

It also generates code to set up an initial function context and an implied **HALT** statement with a return code of 0.

Finally, it makes sure that there is a function declaration for each forward declaration that has been referenced anywhere in the program. I.e. a **DECL** statement declaring a function that is never used will be fine, but as soon as the forward function is called anywhere in the program, it must also be declared as a (non-forward) function later in the program. This is what the loop at the end of **program()** does.

```
program() do var i;
    T := scan();
    while ( T = KVAR \/ T = KCONST \/
            T = SYMBOL \/ T = KDECL \/
            T = KSTRUCT
    )
        declaration(GLOBF);
```

```
    if (T \= KDO)
        aw("DO or declaration expected", 0);
    gen(CG_ENTER, 0);
    compound();
    if (T = ENDFILE)
        aw("trailing characters", Str);
    gen(CG_HALT, 0);
    for (i=0, Yp, SYM)
        if (Syms[i+SFLAGS] & FORW /\ Syms[i+SVALUE])
            aw("undefined function",
               Syms[i+SNAME]);
end
```

Initialization

The **init()** function initializes the global variables of the compiler, verifies the code table, and generates the built-in functions.

The **Codetbl** variable is assigned a vector holding the code fragments emitted by the compiler. Each fragment is associated with a constant with a **CG_** prefix. These constants have been defined in a structure earlier in the program (see page 28). They are included in the table so that fragments are easier to locate, for making modifications, etc.

The **init()** function checks the consistency of the table by making sure that the **CG_** constants are in ascending order. This is necessary, because fragments will be addressed using the **CG_** constants. For instance, an **LDVAL** instruction will be generated by emitting the fragment

```
Codetbl[CG_LDVAL][1]
```

The local variables *tread*, *twrite*, etc are used to store the machine code for the built-in functions. Note that some strings in the **init()** function have been split across multiple lines. This has been done for typographical reasons and is not actually possible in the T3X language; the fragments have to be contained in single lines or have to be concatenated from smaller strings using **str.append()** or a similar function.

The structure of the **Ops** table, which describes the operators of the language, has been described earlier in this text (page 46). Note again that the order of this table is important for the operator scanner (**scanop()** function, page 51) to work properly.

```
init() do var i, tread, twrite, tcomp, tcopy,
            tfill, tscan;
    Rp := 0;
    Tp := 0;
    Dp := 0;
    Lp := 0;
    Sp := 0;
    Yp := 0;
    Np := 0;
    Pp := 0;
    Hp := 0;
    Line := 1;
    Acc := 0;
    Fun := 0;
    Loop0 := %1;
    Lvp := 0;
    Llp := 0;
    Codetbl := [
       [ CG_PUSH,     "50"                ],
       [ CG_LDVAL,    "b8,w"              ],
       [ CG_LDADDR,   "b8,a"              ],
       [ CG_LDLREF,   "8d85,w"            ],
       [ CG_LDGLOB,   "a1,a"              ],
       [ CG_LDLOCL,   "8b85,w"            ],
       [ CG_CLEAR,    "31c0"              ],
       [ CG_STGLOB,   "a3,a"              ],
       [ CG_STLOCL,   "8985,w"            ],
       [ CG_STINDR,   "5b8903"            ],
       [ CG_STINDB,   "5b8803"            ],
       [ CG_INCGLOB,  "8105,a"            ],
       [ CG_INCLOCL,  "8185,w"            ],
       [ CG_ALLOC,    "81ec,w"            ],
       [ CG_DEALLOC,  "81c4,w"            ],
       [ CG_LOCLVEC,  "89e050"            ],
```

```
[ CG_GLOBVEC,  "8925,a"               ],
[ CG_INDEX,    "c1e0025b01d8"         ],
[ CG_DEREF,    "8b00"                 ],
[ CG_INDXB,    "5b01d8"               ],
[ CG_DREFB,    "89c331c08a03"         ],
[ CG_MARK,     ",m"                   ],
[ CG_RESOLV,   ",r"                   ],
[ CG_CALL,     "e8,w"                 ],
[ CG_JUMPFWD,  "e9,>"                 ],
[ CG_JUMPBACK, "e9,<"                 ],
[ CG_JMPFALSE, "09c00f84,>"           ],
[ CG_JMPTRUE,  "09c00f85,>"           ],
[ CG_FOR,      "5b39c30f8d,>"         ],
[ CG_FORDOWN,  "5b39c30f8e,>"         ],
[ CG_ENTER,    "5589e5"               ],
[ CG_EXIT,     "5dc3"                 ],
[ CG_HALT,     "68,w5031c040cd80"     ],
[ CG_NEG,      "f7d8"                 ],
[ CG_INV,      "f7d0"                 ],
[ CG_LOGNOT,   "f7d819c0f7d0"         ],
[ CG_ADD,      "5b01d8"               ],
[ CG_SUB,      "89c35829d8"           ],
[ CG_MUL,      "5bf7eb"               ],
[ CG_DIV,      "89c35899f7fb"         ],
[ CG_MOD,      "89c35899f7fb89d0"     ],
[ CG_AND,      "5b21d8"               ],
[ CG_OR,       "5b09d8"               ],
[ CG_XOR,      "5b31d8"               ],
[ CG_SHL,      "89c158d3e0"           ],
[ CG_SHR,      "89c158d3e8"           ],
[ CG_EQ,       "5b39c30f95c20fb6c248" ],
[ CG_NEQ,      "5b39c30f94c20fb6c248" ],
[ CG_LT,       "5b39c30f9dc20fb6c248" ],
[ CG_GT,       "5b39c30f9ec20fb6c248" ],
[ CG_LE,       "5b39c30f9fc20fb6c248" ],
[ CG_GE,       "5b39c30f9cc20fb6c248" ],
[ CG_WORD,     ",w"                   ],
[ %1,          ""                     ]
```

```
   ];
tread := "8b4424048744240c89442404b803000000
          cd800f830300000031c048c3";
twrite := "8b4424048744240c89442404b80400000
           0cd800f830300000031c048c3";
tcomp := "8b74240c8b7c24088b4c240441fcf3a609
          c90f850300000031c0c38a46ff2a47ff66
          9898c3";
tcopy := "8b74240c8b7c24088b4c2404fcf3a431c0c3";
tfill := "8b7c240c8b4424088b4c2404fcf3aa31c0c3";
tscan := "8b7c240c8b4424088b4c24044189fafcf2"
         "ae09c90f840600000089f829d048c331c0"
         "48c3";
Ops := [
    [ 7, 1, "mod",  BINOP,  CG_MOD      ],
    [ 6, 1, "+",    BINOP,  CG_ADD      ],
    [ 7, 2, "*",    BINOP,  CG_MUL      ],
    [ 0, 1, ";",    SEMI,   0           ],
    [ 0, 1, ",",    COMMA,  0           ],
    [ 0, 1, "(",    LPAREN, 0           ],
    [ 0, 1, ")",    RPAREN, 0           ],
    [ 0, 1, "[",    LBRACK, 0           ],
    [ 0, 1, "]",    RBRACK, 0           ],
    [ 3, 1, "=",    BINOP,  CG_EQ       ],
    [ 5, 1, "&",    BINOP,  CG_AND      ],
    [ 5, 1, "|",    BINOP,  CG_OR       ],
    [ 5, 1, "^",    BINOP,  CG_XOR      ],
    [ 0, 1, "@",    ADDROF, 0           ],
    [ 0, 1, "~",    UNOP,   CG_INV      ],
    [ 0, 1, ":",    COLON,  0           ],
    [ 0, 2, "::",   BYTEOP, 0           ],
    [ 0, 2, ":=",   ASSIGN, 0           ],
    [ 0, 1, "\\",   UNOP,   CG_LOGNOT   ],
    [ 1, 2, "\\/",  DISJ,   0           ],
    [ 3, 2, "\\=",  BINOP,  CG_NEQ      ],
    [ 4, 1, "<",    BINOP,  CG_LT       ],
    [ 4, 2, "<=",   BINOP,  CG_LE       ],
    [ 5, 2, "<<",   BINOP,  CG_SHL      ],
```

```
        [ 4, 1, ">",    BINOP,  CG_GT       ],
        [ 4, 2, ">=",   BINOP,  CG_GE       ],
        [ 5, 2, ">>",   BINOP,  CG_SHR      ],
        [ 6, 1, "-",    BINOP,  CG_SUB      ],
        [ 0, 2, "->",   COND,   0           ],
        [ 7, 2, "/",    BINOP,  CG_DIV      ],
        [ 2, 2, "/\\",  CONJ,   0           ],
        [ 0, 0, 0,      0,      0           ]
        ];
    Equal_op := findop("=");
    Minus_op := findop("-");
    Mul_op := findop("*");
    Add_op := findop("+");
    i := 0;
    while (Codetbl[i][0] \= %1) do
        if (Codetbl[i][0] \= i)
            oops("bad code table entry",
                 ntoa(i));
        i := i+1;
    end
    builtin("t.read", 3, tread);
    builtin("t.write", 3, twrite);
    builtin("t.memcomp", 3, tcomp);
    builtin("t.memcopy", 3, tcopy);
    builtin("t.memfill", 3, tfill);
    builtin("t.memscan", 3, tscan);
end
```

Main Program

Here comes the main program of the compiler. Note that *program*, *procedure*, and *function* are often used as synonyms, so the *main program* is in this case just a part of a larger program.

The T3X9 compiler main program works as follows: it first initializes its internal state and emits the built-in functions. Then it reads the source program, analyzes it, and relocates it. In the final steps, it creates an ELF header for the resulting binary and writes

the three parts of it (header, text segment, and data segment) to the output.

Binary in this context is a synonym for *executable*, i.e. a program that is ready to be run by the operating system.

The compiler compiles from standard input to standard output. It has no file operations, but they could easily be added as built-ins.

Note: the **`align()`** call aligns the data segment with a 16-byte boundary in the ELF file by filling up the text segment.

```
do
    init();
    readprog();
    program();
    ! align in file
    Tp := align(HEADER_SIZE+Tp, 16)-HEADER_SIZE;
    relocate();
    elfheader();
    t.write(1, Header, Hp);
    t.write(1, Text_seg, Tp);
    t.write(1, Data_seg, Dp);
end
```

The ABI

The T3X9 compiler is in principle capable generating binaries for all 386-based operating system that use the ELF executable file format. The only parts of the code that have to change are those that use the FreeBSD Application Binary Interface (*ABI).*

Operating system services, like reading input or writing output, are requested in *system calls*. The exact structure of a system call is defined by the ABI.

The FreeBSD ABI uses the stack to pass arguments to the operating system. The number of the system call performing a specific service is specified in the *%eax* register and the system call itself is initiated by triggering software interrupt 128. The return value of the call is returned in *%eax* and the carry flag serves as an error flag. See figure 46 and [BSD14].

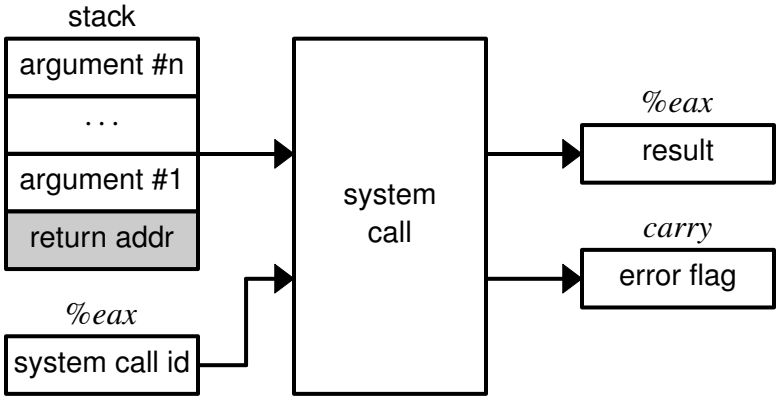


Figure 46: FreeBSD System Call

Note that the system expects arguments to be pushed to the stack in C-style right-to-left order, while T3X uses left-to-right order. This is why functions like **`T.READ()`** re-arrange arguments on the stack before performing a system call.

Also note that FreeBSD expects a return address on the stack (but does not do anything with it). This allows functions to just pass their arguments on to a system call without having to duplicate any

arguments. The T3X9 built-in functions make use of this feature.

The T3X9 compiler uses the following system calls

- *read(2)* in the **`T.READ()`** function
- *write(2)* in the **`T.WRITE()`** function
- *exit(2)* in the **`HALT`** statement

All of these parts need to be modified when porting the compiler to a different operating system.

In addition, the ABI field in the ELF header will need to be adjusted to reflect the new platform. Other fields may also need tweaking.

Compiling the Compiler

Because the T3X9 compiler is *self-hosting* (i.e. it is written in its own source language), it can compile its own source code. However, when a new language is created, how do we get the process started *without* an existing compiler?

This problem is well-known as the *bootstrapping* problem, because it seems to be as hard as pulling oneself out of a swamp by one's own bootstraps. (Bootstraps being the small loops that are attached to some boots to facilitate getting them on.)

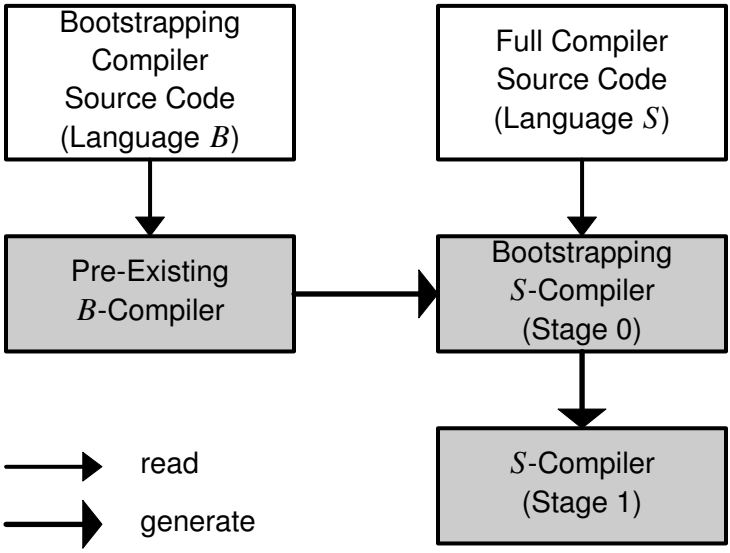


Figure 47: Bootstrapping (gray boxes indicate binaries)

The process of bootstrapping is illustrated in figure 47. The goal of the process is to create a compiler for language S without access to a compiler for language S. The most common approach is to write a *bootstrapping compiler* for S in a different, pre-existing language B and use that compiler to create the stage-0 compiler for S.

The *stage-0 compiler* is the result of bootstrapping a language S. The bootstrapping language which the stage-0 compiles is often a subset S_0 of S that is exactly sufficient to compile the initial full S compiler.

Even if the stage-0 compiler covers the complete language S, it is often very simple. For instance, it may not perform any optimizations, have rudimentary error reporting, or have limited performance. It may even be implemented as a simple interpreter, because it only has to be run for one single time during the entire bootstrapping process.

Once the stage-0 compiler exists, it can immediately be used to compile the full compiler source code again, resulting in the *stage-1 compiler*. If the stage-0 compiler implements a subset S_0, this will be the first compiler implementing the full source language S. At this point, the bootstrapping problem is solved.

Bootstrapping the T3X9 Compiler

The T3X9 compiler package can be found at http://t3x.org. It provides two ways to solve the bootstrapping problem by including the following files:

(1) a bootstrapping compiler written in C;

(2) a pre-compiled FreeBSD-386-ELF binary.

The (stage-3; see below) binary can immediately be used to re-compile the compiler. If you do not trust the binary, run it in a sandbox and/or under a system call tracer, like truss(1). The binary performs exactly five system calls:

(1) reading the source code via read(2),

(2) writing the ELF header via write(2),

(3) writing the text segment via write(2),

(4) writing the data segment via write(2),

(5) terminating the program via exit(2).

Otherwise, a C compiler can be used to create your own stage-0 (and then stage-1, 2, 3) binaries. Any C compiler conforming to the C89 (or later) standard should work fine.

Testing the Compiler

A simple method for verifying that a self-hosting compiler is performing properly is the so-called *triple test*. This test re-compiles the compiler with the stage-1 compiler, resulting in a stage-2 compiler, and then re-iterates the process to generate a stage-3 compiler. The process is depicted in figure 48. It is called "triple test", because it re-compiles the full compiler three times.

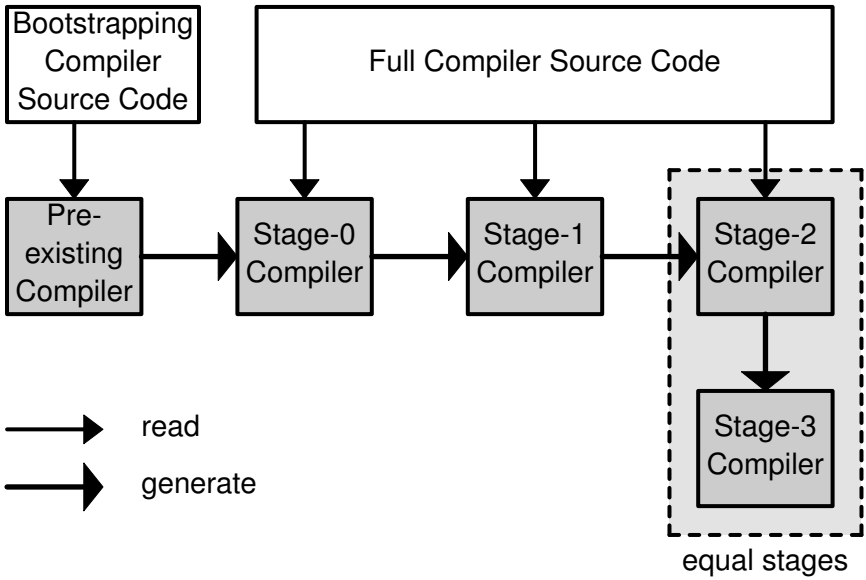


Figure 48: Triple Test (light gray boxes indicate binaries)

As can be seen in the figure, the stage-2 compiler is the first compiler that satisfies the following two conditions:

- it is a full compiler, i.e. it has been compiled from the full compiler sources, so its source language is the full language S and not a subset language S_0
- it has been compiled by a full compiler

Stage 0 is the bootstrapping compiler, which is generated from a completely different source code and probably only accepts a subset language S_0. Stage 1 is a full compiler, but has not been compiled by a full compiler, Stage 2 is the first entirely self-compiled compiler.

It can easily be seen in the diagram that the same conditions hold for the stage-3 compiler. They would also hold for any subsequent stages, so starting at stages 2 and 3, we expect the compilers to be identical, because they have been generated by identical compilers. If this is the case, the triple test has been passed.

Note that the triple test is only a simple test and probably does not cover the entire compiler source code. Passing the triple test is an essential step in the testing of the compiler, though. When this test fails, the compiler cannot be assumed to generate correct code.

Some Random Facts

The T3X9 compiler source code has a total size of 1570 lines or 28K bytes. The parser is the largest part with 750 lines, followed by the scanner with 250, and the emitter with 240 lines. Of course, some of the parts are spread across the code, so these sizes are not exact.

The self-compiled stage-3 executable has a size of 31196 bytes and a virtual set size (size in memory) of 424K bytes. The statically linked stage-0 executable, compiled by Clang, has a size of 517K bytes and a virtual set size of 9172K bytes.

The stage-0 compiler can compile the T3X9 compiler in 0.037 seconds (i.e. 27 times per second) and the stage-3 compiler self-compiles in 0.055 seconds (i.e. 18 times per second), measured on a modest 750MHz notebook.

So the self-compiled T3X9 compiler is about 16 times smaller than the bootstrapping compiler, has a 21 times smaller memory footprint, and achieves about two thirds of the speed of stage 0, which has been compiled by Clang without any optimizations.

The T3X9 compiler package (gzipped tarball) has a size of 33K bytes, including the full compiler source code in T3X, the bootstrapping compiler source code in C, the documentation, *and* a FreeBSD-386-ELF executable.

The compiler has been bootstrapped successfully with Clang, GCC, and PCC on a 10.1-FreeBSD system.

Future Projects

Here are some things you can do with the T3X9 compiler once you have become familiar with its internals. In order of increasing complexity:

(1) Add more built-in functions from the T3X specification, see [NMH96] or the home page at http://t3x.org.

(2) Extend the language. Are you missing a switch statement or nested functions? Go ahead and implement them!

(3) Port the compiler to a different 386-based platform. In the easiest case, you just have to change the ABI (see page 107), in other cases you may have to generate a completely different executable file format.

(4) Look for optimizations. Maybe there are obvious ways to make the compiler generate better code.

(5) Port the compiler to a different 32-bit CPU. Includes everything from (3) plus you have to write your own machine code fragments.

(6) Write a minimal bootstrapping compiler or interpreter. The one coming with the package probably has some features that are not used in the compiler source code.

(7) Port the compiler to a CPU with a word size other than 32 bits or to a virtual machine, like the JVM or the Tcode machine (see [NMH96] or the home page).

Have fun!

Appendix

VSM Code Fragments

VSM Instruction	386 Instructions
`PUSH`	`push %eax`
`CLEAR`	`xor %eax,%eax`
`LDVAL w`	`mov $w,%eax`
`LDADDR a`	`mov $a,%eax`
`LDLREF w`	`lea w(%ebp),%eax`
`LDGLOB a`	`mov a,%eax`
`LDLOCL w`	`mov w(%ebp),%eax`
`STGLOB a`	`mov %eax,a`
`STLOCL w`	`mov %eax,w(%ebp)`
`STINDR`	`pop %ebx` `mov %eax,(%ebx)`
`STINDB`	`pop %ebx` `mov %al,(%ebx)`
`INCGLOB a v`	`add $v,a`
`INCLOCL w v`	`add $v,w(%ebp)`
`ALLOC w`	`sub $w,%esp`
`DEALLOC w`	`add $w,%esp`
`LOCLVEC`	`mov %esp,%eax` `push %eax`
`GLOBVEC a`	`mov %esp,a`

`INDEX`	`shl $0x2,%eax`
	`pop %ebx`
	`add %ebx,%eax`

`DEREF`	`mov (%eax),%eax`
`INDXB`	`pop %ebx` `add %ebx,%eax`

VSM Instruction	386 Instructions

DREFB	mov %eax,%ebx
	xor %eax,%eax
	mov (%ebx),%al

MARK	
RESOLV	
CALL w	call w
JUMPFWD w	jmp w
JUMPBACK w	jmp w
JMPFALSE w	or %eax,%eax je w
JMPTRUE w	or %eax,%eax jne w
FOR w	pop %ebx cmp %eax,%ebx jge w
FORDOWN w	pop %ebx cmp %eax,%ebx jle w
ENTER	push %ebp mov %esp,%ebp
EXIT	pop %ebp ret
HALT w	push $w push %eax xor %eax,%eax inc %eax int $128
NEG	neg %eax
INV	not %eax
LOGNOT	neg %eax sbb %eax,%eax not %eax

VSM Instruction	386 Instructions
ADD	pop %ebx add %ebx,%eax

SUB	mov %eax,%ebx
	pop %eax
	sub %ebx,%eax

MUL	pop %ebx imul %ebx
DIV	mov %eax,%ebx pop %eax cltd idiv %ebx

MOD	mov %eax,%ebx
	pop %eax
	cltd
	idiv %ebx
	mov %edx,%eax

AND	pop %ebx and %ebx,%eax
OR	pop %ebx or %ebx,%eax
XOR	pop %ebx xor %ebx,%eax
SHL	mov %eax,%ecx pop %eax shl %cl,%eax
SHR	mov %eax,%ecx pop %eax shr %cl,%eax
EQ	pop %ebx cmp %eax,%ebx setne %dl movzbl %dl,%eax dec %eax

VSM Instruction	386 Instructions
NEQ	pop %ebx cmp %eax,%ebx sete %dl movzbl %dl,%eax dec %eax
LT	pop %ebx cmp %eax,%ebx setge %dl movzbl %dl,%eax dec %eax
GT	pop %ebx cmp %eax,%ebx setle %dl movzbl %dl,%eax dec %eax
LE	pop %ebx cmp %eax,%ebx setg %dl movzbl %dl,%eax dec %eax
GE	pop %ebx cmp %eax,%ebx setl %dl movzbl %dl,%eax dec %eax

T.READ() Function

```
mov     4(%esp),%eax
xchg    %eax,12(%esp)
mov     %eax,4(%esp)
mov     $3,%eax
int     $128
jnc     1
xor     %eax,%eax
```

```
  dec     %eax
1:ret
```

T.WRITE() Function

```
  mov     4(%esp),%eax
  xchg    %eax,12(%esp)
  mov     %eax,4(%esp)
  mov     $4,%eax
  int     $128
  jnc     1
  xor     %eax,%eax
  dec     %eax
1:ret
```

T.MEMCOMP() Function

```
  mov     12(%esp),%esi
  mov     8(%esp),%edi
  mov     4(%esp),%ecx
  inc     %ecx
  cld
  repz cmpsb
  or      %ecx,%ecx
  jne     1
  xor     %eax,%eax
  ret
1:mov     -1(%esi),%al
  sub     -1(%edi),%al
  cbtw
  cwtl
  ret
```

T.MEMCOPY() Function

```
  mov     12(%esp),%esi
  mov     8(%esp),%edi
  mov     4(%esp),%ecx
```

```
  cld
  rep movsb
  xor    %eax,%eax
  ret
```

T.MEMFILL() Function

```
  mov    12(%esp),%edi
  mov    8(%esp),%eax
  mov    4(%esp),%ecx
  cld
  rep stosb
  xor    %eax,%eax
  ret
```

T.MEMSCAN() Function

```
  mov    12(%esp),%edi
  mov    8(%esp),%eax
  mov    4(%esp),%ecx
  inc    %ecx
  mov    %edi,%edx
  cld
  repnz scasb
  or     %ecx,%ecx
  je     1
  mov    %edi,%eax
  sub    %edx,%eax
  dec    %eax
  ret
1:xor    %eax,%eax
  dec    %eax
  ret
```

T3X9 Summary

Program

A program is a set of declarations followed by a compound statement. Here is the smallest possible T3X program:

```
DO END
```

Comments

A comment is started with an exclamation point (`!`) and extends up to the end of the current line. Example:

```
DO END  ! Do nothing
```

Declarations

```
CONST name = cvalue, ... ;
```

Assign names to constant values.

Example:

```
CONST false = 0, true = %1;
```

```
VAR name, ... ;
VAR name[cvalue], ... ;
VAR name::cvalue, ... ;
```

Define variables, vectors, and byte vectors, respectively. Different definitions may be mixed. Vector elements start at an index of 0.

Example:

```
VAR stack[STACK_LEN], ptr;
```

```
STRUCT name = name_1, ..., name_N;
```

Shorthand for

```
CONST name_1 = 0, ..., name_N = N-1, name = N;
```

Used to impose structure on vectors and byte vectors.

Example:

```
STRUCT POINT = PX, PY, PCOLOR;
VAR p[POINT];
```

```
DECL name(cvalue), ... ;
```

Declare functions whose definitions follow later, where *cvalue* is the number of arguments. Used to implement mutual recursion.

Example:

```
DECL odd(1);
even(x) RETURN x=0-> 1: odd(x-1);
odd(x) RETURN x=1-> 1: even(x-1);
```

```
name(name_1, ...) statement
```

Define function *name* with arguments $name\_1, \cdots$ and a statement as its body. The number of arguments must match any previous **DECL** of the same function.

The arguments of a function are only visible within the statement of the function.

Example:

```
hello(s, x) DO VAR i;
    FOR (i=0, x) DO
        writes(s);
        writes("\n");
    END
END
```

(*Writes*() writes a string; See the compiler source code, page 22, for its implementation.)

Statements

`name := expression;`

Assign the value of an expression to a variable.

Example:

```
DO VAR x; x := 123; END
```

`name[value]... := value;`
`name::value := value;`

Assign the value of an expression to an element of a vector or byte vector. Multiple subscripts may be applied to to a vector:

```
vec[i][j] := i*j;
```

In general, $vec[i][j]$ denotes the j^{th} element of the i^{th} element of vec and $vec::i$ indicates the i^{th} byte of vec.

Subscript and byte subscript operators can be mixed in the same expression, but note that the byte operator `::` associates to the *right*, so $v::x::i$ equals $v::(x::i)$ and therefore,

```
vec[i]::j[k]
```

would denote the $j[k]^{th}$ byte of $vec[i]$.

`name();`
`name(expression_1, ...);`

Call the function with the given name, passing the values of the expressions to the function as arguments. An empty set of parentheses is used to pass zero arguments. The result of the function is discarded.

For further details see the description of function calls in the expression section.

```
IF (condition) statement_1
IE (condition) statement_1 ELSE statement_2
```

Both of these statements run *statement_1*, if the given condition is true.

In addition, **IE**/**ELSE** runs *statement_2*, if the conditions is false. **IF** just passes control to the subsequent statement in this case.

Example:

```
IE (0)
    IF (1) RETURN 1;
ELSE
    RETURN 2;
```

The example always returns 2, because only an **IE** statement can have an **ELSE** branch. There is no “dangling else” problem.

```
WHILE (condition) statement
```

Repeat the statement while the condition is true. When the condition is not true initially, never run the statement.

Example:

```
DO VAR i;    ! Count from 1 to 10
    i := 1;
    WHILE (i < 11)
        i := i+1;
END
```

```
FOR (name=expression_1, expression_2, cvalue)
    statement
FOR (name=expression_1, expression_2)
    statement
```

Assign the value of *expression_1* to *name*, then compare *name* to *expression_2*. If *cvalue* is not negative, run the statement while *name* < *expression_2*. Otherwise run the statement while *name* > *expression_2*. After running the statement, add *cvalue* to *name* and repeat, starting at the comparison. Formally:

```
name := expression_1
WHILE ( cvalue > 0 /\ name < expression \/
        cvalue < 0 /\ name > expression )
DO
    statement;
    name := name + cvalue;
END
```

If *cvalue* is omitted, it defaults to 1.

Example:

```
DO VAR i;
    FOR (i=1, 11);      ! count from 1 to 10
    FOR (i=10, 0, %1); ! count from 10 to 1
END
```

LEAVE;

Leave the innermost **WHILE** or **FOR** loop, passing control to the first statement following the loop (if any).

Example:

```
DO VAR i;  ! Count from 1 to 50
    FOR (i=1, 100)
        IF (i=50) LEAVE;
END
```

LOOP;

Re-enter the innermost **WHILE** or **FOR** loop. **WHILE** loops are re-entered at the point where the condition is tested, and **FOR** loops are re-entered at the point where the counter is incremented.

Example:

```
DO VAR i;    ! This program never prints X
    FOR (i=0, 10) DO
        LOOP;
        T.WRITE(1, "X", 1);
    END
END
```

RETURN expression;

Return a value from a function. For further details see the description of function calls in the expression section.

Example:

```
inc(x) RETURN x+1;
```

HALT cvalue;

Halt program execution and return the given exit code to the operating system.

Example:

```
HALT 1;
```

DO statement ... END
DO declaration ... statement ... END

Compound statements of the form **DO** ... **END** are used to place multiple statements in a context where only a single statement is expected, like selection, loop, and function bodies.

A compound statement may declare its own local variables, constants, and structures (using **VAR**, **CONST**, or **STRUCT**). A local variable of a compound statement is created and allocated at the beginning of the statement ceases to exist at the end of the statement.

Note that the form

DO declaration ... END

also exists, but is essentially an empty statement.

Example:

```
DO var i, x;  ! Compute 10 factorial
    x := 1;
    FOR (i=1, 10)
        x := x*i;
END
```

```
DO END
;
```

These are both empty statements or null statements. They do not do anything when run and may be used as placeholders where a statement would be expected. They are also used to show that nothing is to be done in a specific situation, like in

```
IE (x = 0)
     ;
ELSE IE (x < 0)
     statement;
ELSE
     statement;
```

Example:

```
FOR (i=0, 100000) DO END  ! waste some time
```

Expressions

An expression is an operand of the form of a variable, a literal, or a function call, or a set of operators applied to operands. There are unary, binary, and ternary operators.

Examples:

```
-a       ! negate a
b*c      ! product of b and c
x->y:z   ! if x then y else z
f(x)     ! the value returned by F of X
```

In the following, the symbols X, Y, and Z denote variables or literals.

The operators of the T3X language are listed in figure 49.

The symbol P denotes *precedence*. Higher precedence means that an operator binds stronger to its arguments, e.g. **`-X::Y`** actually means **`-(X::Y)`**.

The symbol A means *associativity*. For any operator "·", left-associativity means $x \cdot y \cdot z = (x \cdot y) \cdot z$ and right-associativity means $x \cdot y \cdot z = x \cdot (y \cdot z)$.

Operator	P	A	Description
`X[Y]`	9	L	the Y'th element of the vector X
`X::Y`	9	R	the Y'th byte of the byte vector X
`-X`	8	-	the negative value of X
`~X`	8	-	the bitwise inverse of X
`\X`	8	-	%1, if X is 0, else 0 (logical NOT)
`@X`	8	-	the address of X
`X*Y`	7	L	the product of X and Y
`Y/Y`	7	L	the integer quotient of X and Y
`X mod Y`	7	L	the division remainder of X and Y
`X+Y`	6	L	the sum of X and Y
`X-Y`	6	L	the difference between X and Y
`X&Y`	5	L	the bitwise AND of X and Y
`X|Y`	5	L	the bitwise OR of X and Y
`X^Y`	5	L	the bitwise XOR of X and Y
`X<<Y`	5	L	X shifted to the left by Y bits
`X>>Y`	5	L	X shifted to the right by Y bits
`X<Y`	4	L	%1, if X is less than Y, else 0
`X>Y`	4	L	%1, if X is less than Y, else 0
`X<=Y`	4	L	%1, if X is less/equal Y, else 0
`X>=Y`	4	L	%1, if X is greater/equal Y, else 0
`X=Y`	3	L	%1, if X equals Y, else 0
`X\=Y`	3	L	%1, if X does not equal Y, else 0
`X/\Y`	2	L	if X then Y else 0 (short-circuit logical AND)
`X\/Y`	1	L	if X then X else Y (short-circuit logical OR)
`X->Y:Z`	0	-	if X then Y else Z

Figure 49: T3X Operators

Conditions

A condition is an expression appearing in a condition context, like the condition of an **IF** or **WHILE** statement or the first operand of the **X->Y:Z** operator.

In an expression context, the value 0 is considered to be "false", and any other value is considered to be true. For example:

X=X is true
1=2 is false
"x" is true
5>7 is false

The canonical truth value, as returned by **1=1**, is **%1**.

Function Calls

When a function call appears in an expression, the result of the function, as returned by **RETURN** is used as an operand.

A function call is performed as follows:

Each actual argument in the call

```
function(argument_1, ...)
```

is passed to the function and stored in the corresponding *formal argument* (“argument”) of the receiving function. The function then runs its statement, which may produce a value via **RETURN**. When no **RETURN** statement exists in the statement, 0 is returned.

Note that the order of argument evaluation is strictly left-to-right, so in

```
f(g(a), g(b))
```

$g(a)$ will always be called before $g(b)$.

Example:

```
pow(x, y) DO VAR a;
    a := 1;
    WHILE (y) DO
        a := a*x;
```

```
        y := y-1;
    END
    RETURN a;
END

DO VAR x;
    x := pow(2,10);
END
```

Literals

Integers

An integer is a decimal number representing its own value. Note that negative numbers have a leading `%` sign rather than a `-` sign. While the latter also works, it is, strictly speaking, the application of the `-` operator to a positive number, so it may not appear in cvalue contexts.

Examples:

```
0
12345
%1
```

Characters

Characters are integers internally. They are represented by single characters enclosed in single quotes. In addition, the same escape sequences as in strings may be used.

Examples:

```
'x'
'\\'
'''
'\e'
```

Strings

A string is a byte vector filled with characters. Strings are delimited by " characters and NUL-terminated internally. All characters between the delimiting double quotes represent themselves. In addition, the escape sequences from figure 50 may be used to include some special characters.

Seq.	Char	Description
\a	BEL	Bell
\b	BS	Backspace
\e	ESC	Escape
\f	FF	Form Feed
\n	LF	Line Feed (newline)
\q	"	Quote
\r	CR	Carriage Return
\s		Space
\t	HT	Horizontal Tabulator
\v	VT	Vertical Tabulator
****	\	Backslash

Figure 50: Escape Sequences

Examples:

```
""
"hello, world!\n"
"\qhi!\q, she said."
```

Tables

A table is a vector literal, i.e. a sequence of values. It is delimited by square brackets and elements are separated by commas. Table elements can be cvalues, strings, and tables.

Examples:

```
[1, 2, 3]
["5 times -7", %35]
[[1,0,0],[0,1,0],[0,0,1]]
```

Dynamic Tables

The dynamic table is a special case of the table in which one or multiple elements are computed at program run time. Dynamic table elements are enclosed in parentheses. E.g. in the table

```
["x times 7", (x*7)]
```

the value of the second element would be computed and filled in when the table is being evaluated. Note that dynamic table elements are being replaced in situ, and remain the same only until they are replaced again.

Multiple dynamic elements may be enclosed by a single pair of parentheses. For instance, the following tables are the same:

```
[(x), (y), (z)]
[(x, y, z)]
```

Cvalues

A cvalue (constant value) is an expression whose value is known at compile time. In full T3X, this is a large subset of full expressions, but in T3X9, it it limited to the following:

- integers
- characters
- constants

as well as (given that X and Y are one of the above):

- `X+Y`
- `X*Y`

Naming Conventions

Symbolic names for variables, constants, structures, and functions are constructed from the following alphabet:

- the characters a-z

- the digits 0-9
- the special characters _ (underscore) and . (dot)

The first character of a name must be non-numeric, the remaining characters may be any of the above.

Upper and lower case is not distinguished, the symbolic names

```
FOO, Foo, foo
```

are all considered to be equal.

By convention,

- **CONST** names are all upper-case
- **STRUCT** names are all upper-case
- global **VAR** names are capitalized
- local **VAR** names are all lower-case
- function names are all lower-case

Keywords, like **VAR**, **IF**, **DO**, etc, sometimes print in upper case in documentation, but are usually in lower case in actual program code.

Shadowing

There is a single name space without any shadowing in T3X:

- all global names must be different
- no local name may have the same name as a global name
- all local names in the same scope must be different

The latter means that local names may be re-used in subsequent scopes, e.g.:

```
f(x) RETURN x;
g(x) RETURN x;
```

would be a valid program. However,

```
f(x) DO VAR x; END  !!! WRONG !!!
```

would not be a valid program, because **`VAR x;`** redefines the argument of F.

Note that function declarations do not shadow **`DECL`** statements, but transform them to function declarations.

Variadic Functions

T3X implements variadic functions (i.e. functions of a variable number of arguments) using dynamic tables. For instance, the following function returns the sum of a vector of arguments:

```
sum(k, v) DO var i, n;
    n := 0;
    FOR (i=0, k)
        n := n+v[i];
    RETURN n;
END
```

Its is an ordinary function returning the sum of a vector. It can be considered to be a variadic function, though, because a dynamic table can be passed to it in the V argument:

```
sum(5, [(a,b,c,d,e)])
```

Built-In Functions

The following built-in functions exist in T3X9. They are generated by the T3X9 compiler and do not have to be declared in any way. The dot in the function names resembles the message operator of the full T3X language, but is an ordinary symbolic character in T3X9.

`T.READ(fd, buf, len)`

Read up to *len* characters from the file descriptor *fd* into the buffer *buf*. Return the number of characters actually read. Return -1 in case of an error.

Example:

```
DO VAR buffer::100;
    t.read(0, buffer, 99);
END
```

T.WRITE(fd, buf, len)

Write *len* characters from the buffer *buf* to the file descriptor *fd*. Return the number of characters actually written. Return -1 in case of an error.

Example:

```
t.write(1, "hello, world!\n", 14);
```

T.MEMCOMP(b1, b2, len)

Compare the first *len* bytes of the byte vectors $b1$ and $b2$. Return the difference of the first pair of mismatching bytes. A return code of 0 means that the compared regions are equal.

Example:

```
t.memcomp("aaa", "aba", 3)  ! gives 'b'-'a' = %1
```

T.MEMCOPY(bs, bd, len)

Copy *len* bytes from the byte vector *bs* (source) to the byte vector *bd* (destination). Return 0.

Example:

```
DO VAR b:100; t.memcopy("hello", b, 6); END
```

T.MEMFILL(bv, b, len)

Fill the first *len* bytes of the byte vector *bv* with the byte value *b*. Return 0.

Example:

```
DO VAR b:100; t.memfill(b, 0, 100); END
```

T.MEMSCAN(bv, b, len)

Locate the first occurrence of the byte value b in the first len bytes of the byte vector bv and return its offset in the vector. When b does not exist in the given region, return -1.

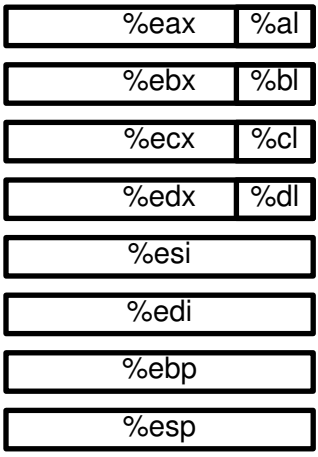
Example:

```
t.memscan("aaab", 'b', 4)  ! returns 3
```

386 Assembly Summary

This is a very terse and incomplete summary of the 386 architecture and assembly language. For a full description, see [Int86].

Registers



All registers are 32-bit registers. The "*l*"-registers (like *%al*) are used to access the lowermost 8 bits of a 32-bit register.

Addressing Modes

Key: *%r* denotes a register, *m* a memory location *n* an offset, and *$n* a constant.

Template	Example	Description
`%r`	`%eax`	content of register
`m`	`8058000`	content of memory address
`(%r)`	`(%eax)`	content of mem. addr. pointed to by *%r*
`n(%r)`	`8(%ebp)`	content of memory address $\%r + n$
`$n`	`$123`	the value *n*

Syntax

This text uses AT&T syntax, which means that operands are in source, destination order, e.g.

```
mov %eax,%ebx
```

means "copy the content of the *%eax* register to *%ebx*".

As a rule of the thumb, all two-operand instructions support all combinations of addressing modes, but only source *or* destination may be a memory address (even indirect), and the destination may not be a constant.

Move Instructions

`lea x,y`	Load address of x into y
`mov x,y`	Load content of x into y
`movzbl x,y`	Like *mov*, but zero-extend byte to long
`pop x`	pop value from stack into x
`push x`	push x to stack
`xchg x,y`	exchange contents of x and y

Arithmetic Instructions

`add x,y`	add x to y
`and x,y`	bitwise AND of x and y, result in y
`dec x`	subtract 1 from x
`idiv x`	divide *%edx:%eax* (double) by x; result in *%eax*, remainder in *%edx*
`imul x`	multiply *%eax* by x; result in *%edx:%eax* (double)
`inc x`	add 1 to x
`neg x`	negate x $(1 - x)$
`not x`	bitwise NOT of x

or x,y	bitwise OR of x and y, result in y
sbb x,y	subtract x from y with borrow, i.e. $y := y - x - c$, where c is the carry flag
shl x,y	shift bits of y to the left by x positions; x must be constant or $\%cl$
shr x,y	shift bits of y to the right by x positions; x must be constant or $\%cl$
sub x,y	subtract x from y
xor x,y	bitwise XOR of x and y, result in y

Function Calls

call x	call function at x, leaving return address on stack
ret	return from function call to address on stack
int x	trigger software interrupt, vector x

Comparison, Flags, and Conditional Jumps

cmp x,y	subtract x from y, but do not store result; only set flags for j? or set? instructions below
je x	jump to x, if $y = x$ in *cmp*
jge x	jump to x, if $y \geq x$ in *cmp*
jle x	jump to x, if $y \leq x$ in *cmp*
jmp x	jump to x (unconditionally)
jnc x	jump to x, if carry flag clear (here used to test results of system calls)
jne x	jump to x, if $y \neq x$ in *cmp*
sete x	set byte in x to 1, if $y = x$ in *cmp*
setg x	set byte in x to 1, if $y > x$ in *cmp*
setge x	set byte in x to 1, if $y \geq x$ in *cmp*
setl x	set byte in x to 1, if $y < x$ in *cmp*
setle x	set byte in x to 1, if $y \leq x$ in *cmp*
setne x	set byte in x to 1, if $y \neq x$ in *cmp*

Conversion

cbtw	convert byte in *%al* to word with sign extension
cwtl	convert 16-bit word in *%eax* to word w/ sign extension

Block Instructions

cld	clear direction flag, all subsequent operations will *increment* source/destination registers
cmpsb	subtract byte at *(%edi)* from byte at *(%esi)*; do not store result, but set flags for *repz*; increment *%esi* and *%edi*
movsb	move byte from *(%esi)* to *(%edi)*; increment *%esi* and *%edi*
rep	repeat following *movsb %ecx* times, i.e. move block of *%ecx* bytes from *%esi* to *%edi* or repeat following *stosb %ecx* times, i.e. fill block of *%ecx* bytes at *%edi* with *%al*
repnz	Repeat following *scasb* until *%al* = (*%edi*), but at most *%ecx* times; i.e. locate *%al* in block of *%ecx* bytes starting at *%edi*; result is *%edi* − 1
repz	Repeat following *cmpsb* while *(%edi)* = (*%esi*), but at most *%ecx* times; i.e. compare blocks of size *%ecx* at *%edi* and *%esi*. *%edi* and *%esi* will point 1 byte past first mismatch.
scasb	subtract byte at *(%edi)* from *%al*; do not store result, but set flags for *repnz*; increment *%edi*
stosb	store *%al* in byte at *(%edi)*; increment *%edi*

List of Figures

1	Compilation	9
2	Separate Compilation with Linking	10
3	Elements of Block-Structured Languages	13
4	Stack Machine Execution Model	18
5	Stack Machine Instruction Mapping	18
6	Symbol Table Layout	24
7	Symbol Table and Name List	25
8	Resolving a Mark by Backpatching	33
9	Function Context	34
10	Use of an Accumulator	39
11	Mapping an ELF File to Memory	41
12	Tokenized Program	45
13	Constant Value Syntax Rules	60
14	Variable Declaration Syntax Rules	61
15	Anonymous Vector and References	62
16	Merging Functions and Vector Allocation	63
17	Constant Declaration Syntax Rules	65
18	Constant Declaration Syntax Rules	65
19	Forward Declaration Syntax Rules	66
20	Chain of Forward References	67
21	Fixing Argument Addresses	68
22	Function Call Syntax Rules	71
23	Table Syntax Rules	73
24	Table Layout in Memory	74
25	Address Syntax Rules	77
26	Factor Syntax Rules	80
27	Arithmetic Operation Syntax Rules	82
28	Precedence Parsing	83
29	Conjunction and Disjunction Syntax Rules	85
30	Short-Circuit Logical AND	85
31	Short-Circuit Optimization	86
32	Conditional Operator Syntax Rules	87
33	Conditional Operator Control Flow	88

34	HALT Statement Syntax Rule	89
35	RETURN Statement Syntax Rules	89
36	IF Statement Syntax Rules	90
37	IF/IE Statement Control Flow	91
38	WHILE Statement Control Flow	92
39	WHILE Statement Syntax Rule	92
40	FOR Statement Syntax Rules	93
41	FOR Statement Control Flow	93
42	LEAVE and LOOP Syntax Rules	95
43	Assignment and Function Call Syntax Rules	97
44	Compound Statement Syntax Rules	99
45	T3X Program Syntax Rules	100
46	FreeBSD System Call	107
47	Bootstrapping	109
48	Triple Test	111
49	T3X Operators	128
50	Escape Sequences	131

Bibliography

[BSD14] The FreeBSD Documentation Project;
“FreeBSD Developers’ Handbook”; 2014

[ELF95] Tool Interface Standard (TIS);
“Executable and Linking Format (ELF) Specification”; 1995

[Int86] Intel Literature Distribution;
“INTEL 80386 Programmer’s Reference Manual”; 1986

[Lev99] John Levine; “Linkers and Loaders”;
Morgan-Kauffman, 1999

[NMH96] Nils M Holm; “Lightweight Compiler Techniques”; 1996;
Lulu Press, 2004

[RWS80] Martin Richards, C. Whitby-Strevens;
“BCPL, the Language and its Compiler”;
Cambridge University Press, 1980

Index

Program Symbols

ACC 28, 39
ACTIVATE() 40
ACTIVE() 40
ADD() 26
ADDRESS() 77
ADD_OP 46
ALIGN() 42
ALPHABETIC() 23
ARITH() 82
AW() 22
BINOP 82
BPW 19
BUILTIN() 41
CLEAR() 40
CNST 23
CODETBL 28, 101
COMPOUND() 98
CONJN() 85
CONST 64, 121
CONSTDECL() 64
CONSTFAC() 59
DATA() 36
DATAW() 36
DATA_SEG 27
DATA_SIZE 19
DATA_VADDR 27
DECL 26, 66, 122
DECLARATION() 70
DFETCH() 36
DISJN() 85
DO 98
DP 28, 61
DPATCH() 36
ELFHEADER() 43
EMIT() 35
EMITOP() 82
EMITW() 35
END 98
ENDFILE 20, 53
EQUAL_OP 46
EXPECT() 57
EXPR() 87
FACTOR() 80
FIND() 25
FINDKW() 49
FINDOP() 52
FNCALL() 71
FOR 124
FORW 23
FUN 56
FUNC 23
FUNDECL() 68
FWDDECL() 66
GEN() 38
GLOBF 23
HALT 126
HALT_STMT() 89
HDWRITE() 42
HEADER 27
HEADER_SIZE 27
HEX() 37
HEXWRITE() 42
HP 28
IE/ELSE 124

IF 124
IF_STMT() 90
INIT() 101
LEAVE 125
LEAVES 56
LEAVE_STMT() 95
LINE 20
LLP 56
LOAD() 76
LOG() 22
LOOKUP() 25
LOOP 125
LOOP0 56, 92
LOOPS 56
LOOP_STMT() 95
LP 28
LVP 56
MAXLOOP 56
MAXTBL 56
META 44
MINUS_OP 46
MKSTRING() 72
MKTABLE() 73
MUL_OP 46
NEWNAME() 26
NLIST 24
NLIST_SIZE 20
NP 24
NRELOC 19
NTOA() 20
NUMERIC() 23
OID 45, 52
OOPS() 22
OPER 46
OPS 46
PAGE_SIZE 27
POP() 22
PP 45
PROG 45
PROGRAM() 100
PROG_SIZE 19
PSIZE 45
PUSH() 22
READC() 47
READEC() 47
READPROG() 47
READRC() 47
REJECT() 48
RELOC 27
RELOCATE() 40
RESOLVE_FWD() 67
RETURN 126
RETURN_STMT() 89
RGEN() 37
RP 28
SCAN() 52
SCANOP() 50
SFLAGS 23
SKIP() 48
SNAME 23
SP 20
SPILL() 40
STACK 20
STACK_SIZE 20
STCDECL() 65
STORE() 76
STR 45, 52
STR.APPEND() 21
STR.COPY() 21
STR.EQUAL() 21
STR.LENGTH() 21
STRUCT 65, 122
SVALUE 23
SWAP() 22

SYM 23
SYMBOLIC() 52
SYMS 24
SYMTBL_SIZE 20
T 45
T.MEMCOMP() 135
T.MEMCOPY() 135
T.MEMFILL() 135
T.MEMSCAN() 135
T.READ() 134
T.WRITE() 135
TAG() 35
TEXT_SEG 27
TEXT_SIZE 19
TEXT_VADDR 27
TFETCH() 35
TOKENS 46
TOKEN_LEN 44
TOS() 22
TP 28
TPATCH() 35
UNOP 80
VAL 45, 52
VAR 61, 121
VARDECL() 61
VECT 23
WHILE 124
WHILE_STMT() 91
WRITES() 22
XEQSIGN() 57
XLPAREN() 57
XRPAREN() 57
XSEMI() 57
XSYMBOL() 58
YP 24

Definitions

386 16
ABI 16, 107
accumulator 39, 72
actual argument 71
address 23, 28, 71, 77
  relative 37
argument
  actual 71
  formal 129
assignment 123
associativity 82, 127
binary 106
binary operator 82
block 12, 98, 126
block-structure 12
bootstrapping 109
bottom-up parser 83
BSS 61
built-in function 41, 134
byte operator 77
byte vector 24
cache 39
case sensitivity 47, 133
character 130
comment 48, 121
compiler 9
  self-hosting 9, 109
compound statement 14, 98, 126
condition 129
conditional operator 87
conjunction 85
constant 64, 121
constant value 59, 132

context 33
counting loop 124
cvalue 59, 132
data segment 27, 61
declaration 70, 121
  forward 26, 66, 122
  function 68, 122
  local 99
dynamic table 74, 132
  element of 74
ELF file format 16, 40
ELF header 43, 108
empty statement 127
end of file 20
executable file format 9
execution model 16, 29
expression 71, 87, 127
factor 80
falling precedence parsing 83
falsity 129
formal argument 129
forward declaration 26, 66, 122
forward jump 32, 93
forward resolution 67
frame 61
frame pointer 34
function 68, 105, 122
  built-in 41, 134
  variadic 134
function call 71, 80, 123, 129
function context 33
function declaration 68, 122
function return 89
global 23
input 106
integer 130
interpretation 14
jump around jump 87, 90
keyword 46, 49, 53, 133
local variable 133
logical AND 85
logical OR 85
loop 12, 124
  counting 124
  unbounded 91
LR parser 84
lvalue 77
machine code 15
main program 105
marking 32
match 60
member (structure) 65
name list 20
null statement 98, 127
object code 10
operand 127
operator 46, 127
  binary 82
  conditional 87
  ternary 87
  unary 80
operator precedence parsing 83
output 106
parser 56
  bottom-up 83
  falling precedence 83
  LR 84
  operator precedence 83
  recursive descent 58
precedence 82, 127
procedure 12, 105
program 9, 100, 105, 121
program termination 89, 126
programming language 9

RDP 58
recursive descent parser 58
relative address 37
relocation 19
relocation entry 27
return address 33
return value 126, 129
runtime library 10
scanner 44
scope 133
segment
  BSS 61
  data 27, 61
  text 27
selection 12, 90, 124
semantics 14
sentence 56
separate compilation 10
shadowing 25, 133
source language 9
spilling 40
stack 20, 61
stack machine 17, 29
stage-0 compiler 109
stage-1 compiler 110
statement 89, 97
  compound 14, 98, 126
  empty 127
  null 98, 127
statement block 98, 126
string 72, 131
structure 65, 122
  member of 65
symbol 59
symbol flags 23
symbol name 132
symbol table 20
symbol table entry 23
syntax 12
syntax analysis 56
syntax rule 60
syntax-directed translation 56
system call 107
table 73, 131
table element 73
  dynamic 74
table, dynamic 132
target language 9
ternary operator 87
testing 111
text segment 27
token 44
token attribute 53
token class 53
top of stack 39
TOS 39
translation 9
  syntax-directed 56
triple test 111
truth 129
unary operator 80
unbounded loop 91
value 71, 77
  constant 59, 132
variable 61, 121
  local 133
variadic function 134
vector 121
virtual stack machine 17, 29
VSM 17, 29