An introduction to compilers
DRAFT

D. Vermeir
Dept. of Computer Science
Vrij Universiteit Brussel, VUB
dvermeir@vub.ac.be

February 4, 2009

Contents

Chapter 1

Introduction

1.1 Compilers and languages

A *compiler* is a program that translates a *source language* text into an equivalent *target language* text.

E.g. for a C compiler, the source language is C while the target language may be Sparc assembly language.

Of course, one expects a compiler to do a faithful translation, i.e. the *meaning* of the translated text should be the same as the meaning of the source text.

One would not be pleased to see the C program in Figure 1.1

```
#include        <stdio.h>

int
main(int,char**)
{
int x = 34;
x = x*24;
printf("%d\n",x);
}
```

Figure 1.1: A source text in the C language

translated to an assembler program that, when executed, printed "Goodbye world" on the standard output.

So we want the translation performed by a compiler to be *semantics preserving*. This implies that the compiler is able to "understand" (compute the semantics of) the source text. The compiler must also "understand" the target language in order to be able to generate a semantically equivalent target text.

Thus, in order to develop a compiler, we need a precise definition of both the source and the target language. This means that both source and target language must be *formal.*

A language has two aspects: a *syntax* and a *semantics*. The syntax prescribes which texts are grammatically correct and the semantics specifies how to derive the meaning from a syntactically correct text. For the C language, the syntax specifies e.g. that

> *"the body of a function must be enclosed between matching braces ("{}")".*

The semantics says that the meaning of the second statement in Figure 1.1 is that

> *"the value of the variable x is multiplied by 24 and the result becomes the new value of the variable x"*

It turns out that there exist excellent formalisms and tools to describe the syntax of a formal language. For the description of the semantics, the situation is less clear in that existing semantics specification formalisms are not nearly as simple and easy to use as syntax specifications.

1.2 Applications of compilers

Traditionally, a compiler is thought of as translating a so-called "high level language" such as C[1] or Modula2 into assembly language. Since assembly language cannot be directly executed, a further translation between assembly language and (relocatable) machine language is necessary. Such programs are usually called *assemblers* but it is clear that an assembler is just a special (easier) case of a compiler.

Sometimes, a compiler translates between high level languages. E.g. the first C++ implementations used a compiler called "cfront" which translated C++ code to C code. Such a compiler is often called a "cross-compiler".

On the other hand, a compiler need not target a real assembly (or machine) language. E.g. Java compilers generate code for a virtual machine called the "Java

[1]If you want to call C a high-level language

Virtual Machine" (JVM). The JVM interpreter then interprets JVM instructions without any further translation.

In general, an interpreter needs to understand only the source language. Instead of translating the source text, an interpreter immediately executes the instructions in the source text. Many languages are usually "interpreted", either directly, or after a compilation to some virtual machine code: Lisp, Smalltalk, Prolog, SQL are among those. The advantages of using an interpreter are that is easy to port a language to a new machine: all one has to do is to implement the virtual machine on the new hardware. Also, since instructions are evaluated and examined at run-time, it becomes possible to implement very flexible languages. E.g. for an interpreter it is not a problem to support variables that have a dynamic type, something which is hard to do in a traditional compiler. Interpreters can even construct "programs" at run time and interpret those without difficulties, a capability that is available e.g. for Lisp or Prolog.

Finally, compilers (and interpreters) have wider applications than just translating programming languages. Conceivably any large and complex application might define its own "command language" which can be translated to a virtual machine associated with the application. Using compiler generating tools, defining and implementing such a language need not be difficult. Hence SQL can be regarded as such a language associated with a database management system. Other so-called "little languages" provide a convenient interface to specialized libraries. E.g. the language (n)awk is a language that is very convenient to do powerful pattern matching and extraction operations on large text files.

1.3 Overview of the compilation process

In this section we will illustrate the main phases of the compilation process through a simple compiler for a toy programming language. The source for an implementation of this compiler can be found in Appendix B and on the web site of the course.

```
program          :  declaration_list statement_list
                 ;
declaration_list :  declaration ; declaration_list
                 |  ϵ
                 ;
declaration      :  declare var
                 ;
statement_list   :  statement ; statement_list
                 |  ϵ
                 ;
statement        :  assignment
                 |  read_statement
                 |  write_statement
                 ;
assignment       :  var = expression
                 ;
read_statement   :  read var
                 ;
write_statement  :  write expression
                 ;
expression       :  term
                 |  term  + term
                 |  term  − term
                 ;
term             :  NUMBER
                 |  var
                 |  ( expression  )
                 ;
var              :  NAME
                 ;
```

Figure 1.2: The syntax of the Micro language

1.3.1 Micro

The source language “Micro” is very simple. It is based on the toy language described in [FL91].

The syntax of Micro is described by the rules in Figure 1.2. We will see in Chapter 3 that such rules can be formalized into what is called a *grammar*.

Note that *NUMBER* and *NAME* have not been further defined. The idea is, of course, that *NUMBER* represents a sequence of digits and that *NAME* represents a string of letters and digits, starting with a letter.

A simple Micro program is shown in Figure 1.3

```
{
declare xyz;
xyz = (33+3)-35;
write xyz;
}
```

Figure 1.3: A Micro program

The semantics of Micro should be clear[2]: a Micro program consists of a sequence of read/write or assignment statements. There are integer-valued variables (which need to be declared before they are used) and expressions are restricted to addition and substraction.

1.3.2 x86 code

The target language will be code for the x86 processor family. Figure 1.4 shows part of the output of the compiler for the program of Figure 1.3. The full output can be found in Section B.6.2, page 157.

X86 processors have a number of registers, some of which are special purpose, such as the **`esp`** register which always points to the top of the stack (which grows downwards). More information on x86 assembler programming can be found in the Appendix, Section A, page 139.

line 1 The code is divided into data and text sections where the latter contains the actual instructions.

line 2 This defines a data area of 4 bytes wide which can be referenced using the name *xyz*. This definition is the translation of a Micro *declaration*.

[2]The output of the program in Figure 1.3 is, of course, 1.

```
 1    .section .data
 2    .lcomm xyz, 4
 3    .section .text
    ...
44    .globl main
45    .type main, @function
46    main:
47        pushl %ebp
48        movl %esp, %ebp
49        pushl $33
50        pushl $3
51        popl %eax
52        addl %eax, (%esp)
53        pushl $35
54        popl %eax
55        subl %eax, (%esp)
56        popl xyz
57        pushl xyz
58        call print
59        movl %ebp, %esp
60        popl %ebp
61        ret
            ...
```

Figure 1.4: X86 assembly code generated for the program in Figure 1.3

line 44 This defines *main* as a globally available name. It will be used to refer to the single function (line 45) that contains the instructions corresponding to the Micro program. The function starts at line 46 where the location corresponding to the label '*main*' is defined.

line 47 Together with line 48, this starts off the function according to the C calling conventions: the current top of stack contains the return address. This address has been pushed on the stack by the (function) **`call`** instruction. It will eventually be used (and popped) by a subsequent **`ret`** (return from function call) instruction. Parameters are passed by pushing them on the stack just before the **`call`** instruction. Line 47 saves the caller's "base pointer" in the **`ebp`** register by pushing it on the stack before setting (line 48) the current top of the stack as a new base pointer **`ebp`**. When returning from the function call, the orginal stack is restored by copying the saved value from **`ebp`** to **`esp`** (line 59) and popping the saved base pointer (line 60).

line 49 The evaluation of the subexpression $33 + 3$ is initiated by pushing both constants (indicates by the use of a '$' prefix) on the stack (lines 49, 50) line 51 Once both operands are on the top of the stack, the operation (corresponding to the $33 + 3$ expression) is executed by popping the second argument to the **`eax`** register (line 51) which is then added to the first argument on the top of the stack (line 52). The net result is that the two arguments of the operation are replaced on the top of the stack by the (single) result of the operation.

line 53 To substract 35 from the result, this second argument of the substraction is pushed on the stack (line 53), after which the substraction is executed on the two operands on the stack, replacing them on the top of the stack by the result (lines 54,55).

line 56 The result of evaluating the expression is assigned to the variable *xyz* by popping it from the stack to the appropriate address.

line 57 In order to print the value at address *xyz*, it is first pushed on the stack as a parameter for a subsequent call (line 58) to a *print* function (the code of which can be found in Section B.6.2).

1.3.3 Lexical analysis

The raw input to a compiler consists of a string of bytes or characters. Some of those characters, e.g. the "{" character in Micro, may have a meaning by themselves. Other characters only have meaning as part of a larger unit. E.g. the "*y*" in the example program from Figure 1.3, is just a part of the *NAME* "*xyz*". Still others, such as " ", "\n" serve as separators to distinguish one meaningful string from another.

The first job of a compiler is then to group sequences of raw characters into meaningful *tokens*. The lexical analyzer module is responsible for this. Conceptually, the lexical analyzer (often called *scanner*) transforms a sequence of characters into a sequence of tokens. In addition, a lexical analyzer will typically access the *symbol table* to store and/or retrieve information on certain source language concepts such as variables, functions, types.

For the example program from Figure 1.3, the lexical analyzer will transform the character sequence

```
{ declare xyz; xyz = (33+3)-35; write xyz; }
```

into the token sequence shown in Figure 1.5.

Note that some tokens have "properties", e.g. a $\langle$*NUMBER*$\rangle$ token has a *value* property while a $\langle$*NAME*$\rangle$ token has a symbol table reference as a property.

```
⟨LBRACE⟩
⟨DECLARE symbol_table_ref=0⟩
⟨NAME symbol_table_ref=3⟩
⟨SEMICOLON⟩
⟨NAME symbol_table_ref=3⟩
⟨ASSIGN⟩
⟨LPAREN⟩
⟨NUMBER value=33⟩
⟨PLUS⟩
⟨NUMBER value=3⟩
⟨RPAREN⟩
⟨MINUS⟩
⟨NUMBER value=35⟩
⟨SEMICOLON⟩
⟨WRITE symbol_table_ref=2⟩
⟨NAME symbol_table_ref=3⟩
⟨SEMICOLON⟩
⟨RBRACE⟩
```

Figure 1.5: Result of lexical analysis of program in Figure 1.3

After the scanner finishes, the symbol table in the example could look like

0	"declare"	DECLARE
1	"read"	READ
2	"write"	WRITE
3	"xyz"	NAME

where the third column indicates the type of symbol.

Clearly, the main difficulty in writing a lexical analyzer will be to decide, while reading characters one by one, when a token of which type is finished. We will see in Chapter 2 that regular expressions and finite automata provide a powerful and convenient method to automate this job.

1.3.4 Syntax analysis

Once lexical analysis is finished, the *parser* takes over to check whether the sequence of tokens is grammatically correct, according to the rules that define the syntax of the source language.

Looking at the grammar rules for Micro (Figure 1.2), it seems clear that a program is syntactically correct if the structure of the tokens matches the structure of a ⟨*program*⟩ as defined by these rules.

Such matching can conveniently be represented as a *parse tree*. The parse tree corresponding to the token sequence of Figure 1.5 is shown in Figure 1.6.

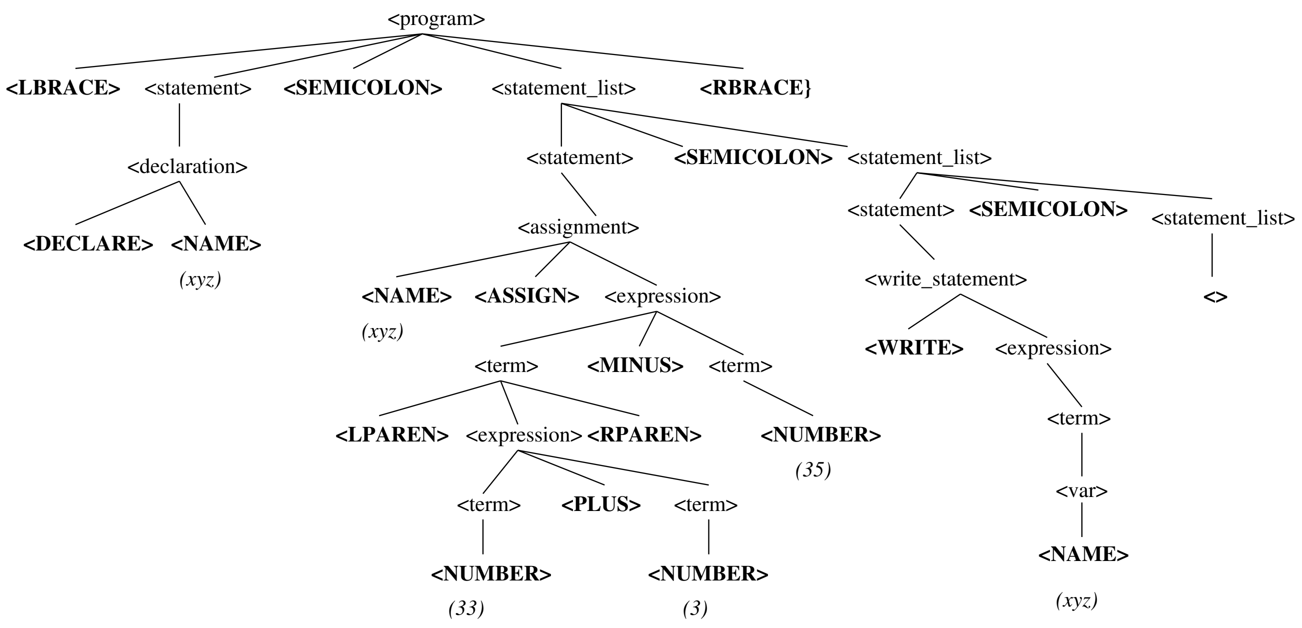


Figure 1.6: Parse tree of program in Figure 1.3

Note that in the parse tree, a node and its children correspond to a rule in the syntax specification of Micro: the parent node corresponds to the left hand side of the rule while the children correspond to the right hand side. Furthermore, the yield[3] of the parse tree is exactly the sequence of tokens that resulted from the lexical analysis of the source text.

Hence the job of the parser is to construct a parse tree that fits, according to the syntax specification, the token sequence that was generated by the lexical analyzer.

In Chapter 3, we'll see how context-free grammars can be used to specify the syntax of a programming language and how it is possible to automatically generate parser programs from such a context-free grammar.

1.3.5 Semantic analysis

Having established that the source text is syntactically correct, the compiler may now perform additional checks such as determining the type of expressions and

[3]The *yield* of a tree is the sequence of leafs of the tree in lexicographical (left-to-right) order

checking that all statements are correct with respect to the typing rules, that variables have been properly declared before they are used, that functions are called with the proper number of parameters etc.

This phase is carried out using information from the parse tree and the symbol table. In our example, very little needs to be checked, due to the extreme simplicity of the language. The only check that is performed verifies that a variable has been declared before it is used.

1.3.6 Intermediate code generation

In this phase, the compiler translates the source text into an simple intermediate language. There are several possible choices for an intermediate language. but in this example we will use the popular "three-address code" format. Essentially, three-address code consists of assignments where the right-hand side must be a single variable or constant or the result of a binary or unary operation. Hence an assignment involves at most three variables (addresses), which explains the name. In addition, three-address code supports primitive control flow statements such as *goto*, *branch-if-positive* etc. Finally, retrieval from and storing into a one-dimensional array is also possible.

The translation process is *syntax-directed*. This means that

- Nodes in the parse tree have a set of *attributes* that contain information pertaining to that node. The set of attributes of a node depends on the kind of syntactical concept it represents. E.g. in Micro, an attribute of an ⟨*expression*⟩ could be the sequence of x86 instructions that leave the result of the evaluation of the expression on the top of the stack. Similarly, both ⟨*var*⟩ and ⟨*expression*⟩ nodes have a *name* attribute holding the name of the variable containing the current value of the ⟨*var*⟩ or ⟨*expression*⟩

  We use $n.a$ to refer to the value of the attribute a for the node n.

- A number of *semantic rules* are associated with each syntactic rule of the grammar. These semantic rules determine the values of the attributes of the nodes in the parse tree (a parent node and its children) that correspond to such a syntactic rule. E.g. in Micro, there is a semantic rule that says that the code associated with an ⟨*assignment*⟩ in the rule

$$\textit{assignment} \;:\; \textit{var} = \textit{expression}$$

consists of the code associated with $\langle$*expression*$\rangle$ followed by a three-address code statement of the form

$$var.name = expression.name$$

More formally, such a semantic rule might be written as

$$assignment.code = expression.code \;\|\; \text{“}\textbf{var.name} = \textbf{expression.name}\text{”}$$

- The translation of the source text then consists of the value of a particular attribute for the root of the parse tree.

Thus intermediate code generation can be performed by computing, using the semantic rules, the attribute values of all nodes in the parse tree. The result is then the value of a specific (e.g. “code”) attribute of the root of the parse tree.

For the example program from Figure 1.3, we could obtain the three-address code in Figure 1.7.

```
T0 = 33 +3
T1 = T0 - 35
XYZ = T1
WRITE XYZ
```

Figure 1.7: three-address code corresponding to the program of Figure 1.3

Note the introduction of several temporary variables, due to the restrictions inherent in three-address code. The last statement before the *WRITE* may seem wasteful but this sort of inefficiency is easily taken care of by the next optimization phase.

1.3.7 Optimization

In this phase, the compiler tries several optimization methods to replace fragments of the intermediate code text with equivalent but faster (and usually also shorter) fragments.

Techniques that can be employed include common subexpression elimination, loop invariant motion, constant folding etc. Most of these techniques need extra information such as a flow graph, live variable status etc.

In our example, the compiler could perform constant folding and code reordering resulting in the optimized code of Figure 1.8.

```
XYZ = 1
WRITE XYZ
```

Figure 1.8: Optimized three-address code corresponding to the program of Figure 1.3

1.3.8 Code generation

The final phase of the compilation consists of the generation of target code from the intermediate code. When the target code corresponds to a register machine, a major problem is the efficient allocation of scarce but fast registers to variables. This problem may be compared with the paging strategy employed by virtual memory management systems. The goal is in both cases to minimize traffic between fast (the registers for a compiler, the page frames for an operating system) and slow (the addresses of variables for a compiler, the pages on disk for an operating system) memory. A significant difference between the two problems is that a compiler has more (but not perfect) knowledge about future references to variables, so more optimization opportunities exist.

Chapter 2

Lexical analysis

2.1 Introduction

As seen in Chapter 1, the lexical analyzer must transform a sequence of “raw” characters into a sequence of *tokens*. Often a token has a structure as in Figure 2.1.

```
#ifndef LEX_H
#define LEX_H
//      %M%(%I%)        %U%     %E%

typedef enum { NAME, NUMBER, LBRACE, RBRACE, LPAREN, RPAREN, ASSIGN,
                SEMICOLON, PLUS, MINUS, ERROR } TOKENT;

typedef struct
        {
        TOKENT  type;
        union {
                int     value;  /* type == NUMBER */
                char    *name;  /* type == NAME */
                } info;
        }       TOKEN;

extern TOKEN    *lex();
#endif  LEX_H
```

Figure 2.1: A declaration for TOKEN and lex()

Actually, the above declaration is not realistic. Usually, more “complex” tokens such as *NAME*s will refer to a symbol table entry rather than simply their string representation.

Clearly, we can split up the scanner using a function *lex()* as in Figure 2.1 which returns the next token from the source text.

It is not impossible[1] to write such a function by hand. A simple implementation of a hand-made scanner for Micro (see Chapter 1 for a definition of “Micro”) is shown below.

```
//      %M%(%I%)        %U%     %E%

#include        <stdio.h>       /* for getchar() and friends */
#include        <ctype.h>       /* for isalpha(), isdigit() and friends */
#include        <stdlib.h>      /* for atoi() */
#include        <string.h>      /* for strdup() */

#include        "lex.h"

static int      state   = 0;

#define MAXBUF  256
static char     buf[MAXBUF];
static char*    pbuf;

static char*    token_name[]    =
                {
                "NAME",         "NUMBER",       "LBRACE",       "RBRACE",
                "LPAREN",       "RPAREN",       "ASSIGN",       "SEMICOLON",
                "PLUS",         "MINUS",        "ERROR"
                };

static TOKEN    token;
/*
*       This code is not robust: no checking on buffer overflow, ...
*       Nor is it complete: keywords are not checked but lumped into
*       the 'NAME' token type, no installation in symbol table, ...
*/
TOKEN*
lex()
{
char    c;

while (1)
  switch(state)
        {
        case 0: /* stands for one of 1,4,6,8,10,13,15,17,19,21,23 */
                pbuf    = buf;
                c = getchar();
                if (isspace(c))
                        state = 11;
```

[1]Some people actually enjoy this.

```
                else if (isdigit(c))
                        {
                        *pbuf++ = c; state = 2;
                        }
                else if (isalpha(c))
                        {
                        *pbuf++ = c; state = 24;
                        }
                else switch(c)
                        {
                        case '{': state = 5; break;
                        case '}': state = 7; break;
                        case '(': state = 9; break;
                        case ')': state = 14; break;
                        case '+': state = 16; break;
                        case '-': state = 18; break;
                        case '=': state = 20; break;
                        case ';': state = 22; break;
                        default:
                                state = 99; break;
                        }
                break;
        case 2:
                c = getchar();
                if (isdigit(c))
                        *pbuf++ = c;
                else
                        state = 3;
                break;
        case 3:
                token.info.value= atoi(buf);
                token.type      = NUMBER;
                ungetc(c,stdin);
                state = 0; return &token;
                break;
        case 5:
                token.type      = LBRACE;
                state = 0; return &token;
                break;
        case 7:
                token.type      = RBRACE;
                state = 0; return &token;
                break;
        case 9:
                token.type      = LPAREN;
                state = 0; return &token;
                break;
        case 11:
                c = getchar();
                if (isspace(c))
                        ;
                else
                        state = 12;
                break;
        case 12:
                ungetc(c,stdin);
                state = 0;
                break;
        case 14:
                token.type      = RPAREN;
                state = 0; return &token;
                break;
        case 16:
                token.type      = PLUS;
                state = 0; return &token;
                break;
        case 18:
                token.type      = MINUS;
                state = 0; return &token;
                break;
        case 20:
                token.type      = ASSIGN;
                state = 0; return &token;
                break;
        case 22:
                token.type      = SEMICOLON;
                state = 0; return &token;
                break;
        case 24:
                c = getchar();
                if (isalpha(c)||isdigit(c))
                        *pbuf++ = c;
                else
                        state = 25;
                break;
        case 25:
                *pbuf   = (char)0;
                token.info.name = strdup(buf);
                token.type      = NAME;
                ungetc(c,stdin);
                state = 0; return &token;
                break;
        case 99:
                if (c==EOF)
                        return 0;
                fprintf(stderr,"Illegal character: \'%c\'\n",c);
                token.type      = ERROR;
                state = 0; return &token;
                break;
        default:
                break;  /* Cannot happen */
        }
}

int
main()
{
TOKEN   *t;

while ((t=lex()))
  {
  printf("%s",token_name[t->type]);
  switch (t->type)
        {
        case NAME:
                printf(": %s\n",t->info.name);
                break;
        case NUMBER:
                printf(": %d\n",t->info.value);
                break;
        default:
                printf("\n");
                break;
        }
  }
return 0;
}
```

The control flow in the above *lex()* procedure can be represented by a combination of so-called transition diagrams which are shown in Figure 2.2.

There is a transition diagram for each token type and another one for white space (blank, tab, newline). The code for lex() simply implements those diagrams. The only complications are

- When starting a new token (i.e., upon entry to lex()), we use a "special" state 0 to represent the fact that we didn't decide yet which diagram to follow. The choice here is made on the basis of the next input character.
- In Figure 2.2, bold circles represent states where we are sure which token has been recognized. Sometimes (e.g. for the *LBRACE* token type) we know this immediately after scanning the last character making up the token. However, for other types, like *NUMBER*, we only know the full extent of the token after reading an extra character that will not belong to the to-

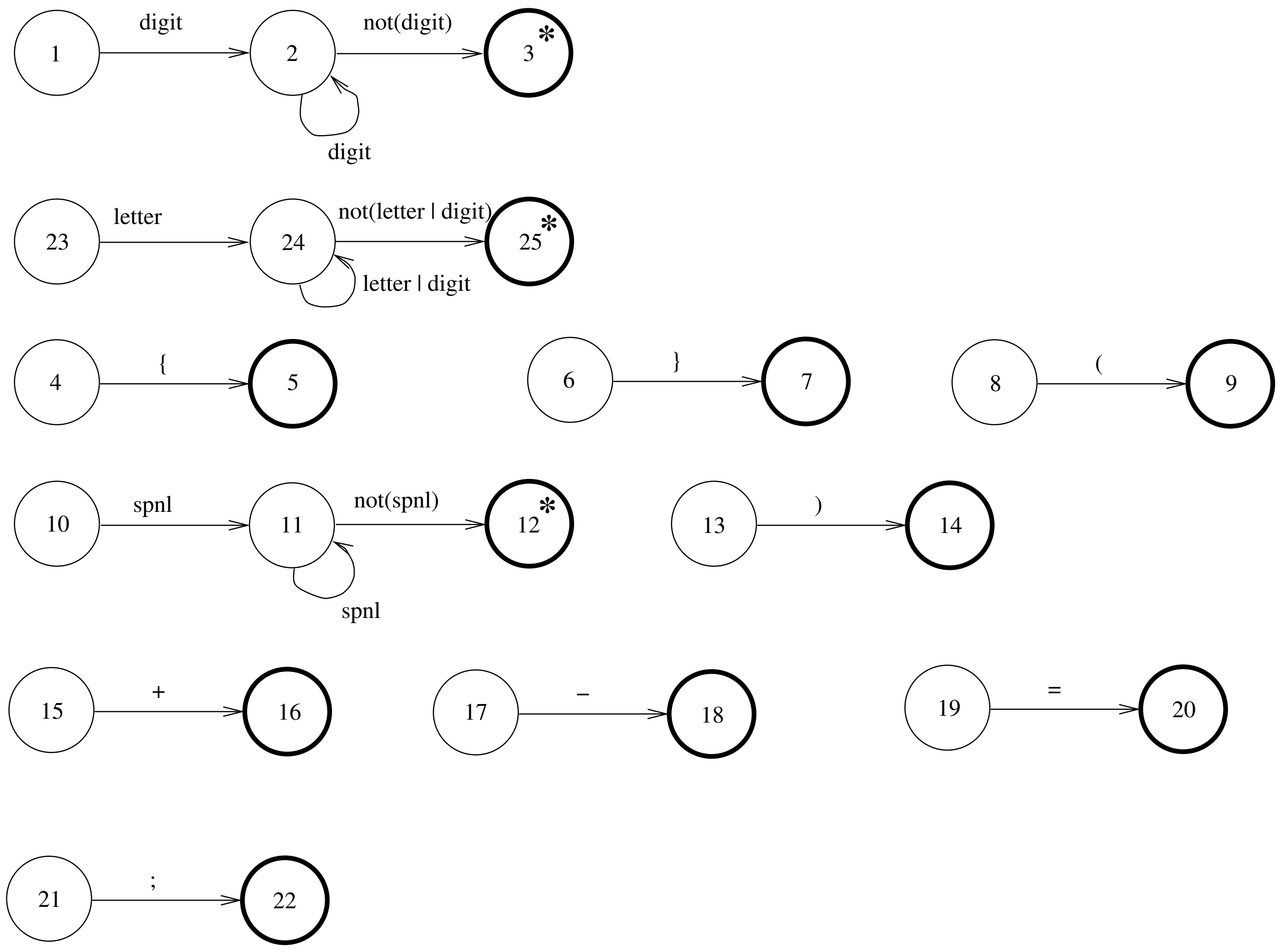


Figure 2.2: The transition diagram for *lex()*

ken. In such a case we must push the extra character back onto the input before returning. Such states have been marked with a * in Figure 2.2.

- If we read a character that doesn't fit any transition diagram, we return a special *ERROR* token type.

Clearly, writing a scanner by hand seems to be easy, once you have a set of transition diagrams such as the ones in Figure 2.2. It is however also boring, and error-prone, especially if there are a large number of states.

Fortunately, the generation of such code can be automated. We will describe how a specification of the various token types can be automatically converted in code that implements a scanner for such token types.

First we will design a suitable formalism to specify token types.

2.2 Regular expressions

In Micro, a *NUMBER* token represents a digit followed by 0 or more digits. A *NAME* consists of a letter followed by 0 or more alphanumeric characters. A *LBRACE* token consists of exactly one “{” character, etc.

Such specifications are easily formalized using *regular expressions*. Before defining regular expressions we should recall the notion of *alphabet* (a finite set of abstract symbols, e.g. ASCII characters), and *(formal) language* (a set of strings containing symbols from some alphabet).

The length of a string w, denoted $|w|$ is defined as the number of symbols occurring in w. The prefix of length l of a string w, denoted $\mathit{pref}_l(w)$ is defined as the longest string x such that $|x| \leq l$ and $w = xy$ for some string y. The *empty string* (of length 0) is denoted ϵ. The *product* $L_1.L_2$ of two languages is the language

$$L_1.L_2 = \{xy \mid x \in L_1 \ \wedge \ y \in L_2\}$$

The *closure* L^* of a language L is defined by

$$L^* = \cup_{i \in \mathbb{N}} L^i$$

(where, of course, $L^0 = \{\epsilon\}$ and $L^{i+1} = L.L^i$).

Definition 1 *The following table, where r and s denote arbitrary regular expressions, recursively defines all regular expressions over a given alphabet Σ, together with the language L_x each expression x represents.*

Regular expression	*Language*
$\emptyset$	$\emptyset$
ϵ	$\{\epsilon\}$
a	$\{a\}$
$(r + s)$	$L_r \cup L_s$
(rs)	$L_r \cdot L_s$
(r^*)	L_r^*

In the table, r and s denote arbitrary regular expressions, and $a \in \Sigma$ is an arbitrary symbol from Σ.

A language L for which there exists a regular r expression such that $L_r = L$ is called a ***regular language***.

We assume that the operators $+$, concatenation and * have increasing precedence, allowing us to drop many parentheses without risking confusion. Thus, $((0(1^*)) + 0)$ may be written as $01^* + 0$.

From Figure 2.2 we can deduce regular expressions for each token type, as shown in Figure 2.1. We assume that

$$\Sigma = \{a, \ldots, z, A, \ldots, Z, 0, \ldots, 9, \text{SP}, \text{NL}, (,), +, =, \{, \}, ;, -\}$$

Token type or abbreviation	Regular expression
letter	$a + \ldots + z + A + \ldots + Z$
digit	$0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$
NUMBER	digit(digit)*
NAME	letter(letter + digit)*
space	(SP + NL)(SP + NL)*
LBRACE	{
..	..

Table 2.1: Regular expressions describing Micro tokens

A full specification, such as the one in Section B.1, page 149, then consists of a set of (extended) regular expressions, plus C code for each expression. The idea is that the generated scanner will

- Process input characters, trying to find a longest string that matches any of the regular expressions[2].
- Execute the code associated with the selected regular expression. This code can, e.g. install something in the symbol table, return a token type or whatever.

In the next section we will see how a regular expression can be converted to a so-called *deterministic finite automaton* that can be regarded as an abstract machine to recognize strings described by regular expressions. Automatic translation of such an automaton to actual code will turn out to be straightforward.

[2]If two expressions match the same longest string, the one that was declared first is chosen.

2.3 Finite state automata

2.3.1 Deterministic finite automata

Definition 2 *A **deterministic finite automaton (DFA)** is a tuple*

$$(Q, \Sigma, \delta, q_0, F)$$

where

- *Q is a finite set of **states**,*
- *Σ is a finite **input alphabet***
- *$\delta : Q \times \Sigma \to Q$ is a (total) transition function*
- *$q_0 \in Q$ is the **initial state***
- *$F \subseteq Q$ is the set of **final states***

Definition 3 *Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. A **configuration** of M is a pair $(q, w) \in Q \times \Sigma^*$. For a configuration (q, aw) (where $a \in \Sigma$), we write*

$$(q, aw) \vdash_M (q', w)$$

just when $\delta(q, a) = q'$ [3]*. The reflexive and transitive closure of the binary relation $\vdash_M$ is denoted as $\vdash_M^*$. A sequence*

$$c_0 \vdash_M c_1 \vdash_M \ldots \vdash_M c_n$$

*is called a **computation** of $n \geq 0$ steps by M.*

*The **language accepted by** M is defined by*

$$L(M) = \{w \mid \exists q \in F \cdot (q_0, w) \vdash_M^* (q, \epsilon)\}$$

We will often write $\delta^*(q, w)$ to denote the unique $q' \in Q$ such that $(q, w) \vdash_M^* (q', \epsilon)$.

[3] We will drop the subscript M in $\vdash_M$ if M is clear from the context.

Example 1 Assuming an alphabet $\Sigma = \{l, d, o\}$ (where "*l*" stands for "letter", "*d*" stands for "digit" and "*o*" stands for "other"), a DFA recognizing Micro *NAME*s can be defined as follows:

$$M = (\{q_0, q_e, q_1\}, \{l, d, o\}, \delta, q_0, \{q_1\})$$

where δ is defined by

$$\begin{array}{rcl}
\delta(q_0, l) & = & q_1 \\
\delta(q_0, d) & = & q_e \\
\delta(q_0, o) & = & q_e \\
\delta(q_1, l) & = & q_1 \\
\delta(q_1, d) & = & q_1 \\
\delta(q_1, o) & = & q_e \\
\delta(q_e, l) & = & q_e \\
\delta(q_e, d) & = & q_e \\
\delta(q_e, o) & = & q_e
\end{array}$$

M is shown in Figure 2.3 (the initial state has a small incoming arrow, final states are in bold):

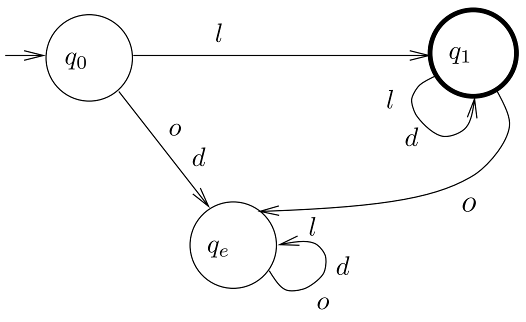


Figure 2.3: A DFA for *NAME*

Clearly, a DFA can be efficiently implemented, e.g. by encoding the states as numbers and using an array to represent the transition function. This is illustrated in Figure 2.4. The `next_state` array can be automatically generated from the DFA description.

What is not clear is how to translate regular expressions to DFA's. To show how this can be done, we need the more general concept of a nondeterministic finite automaton (NFA).

```
typedef int      STATE;
typedef char     SYMBOL;
typedef enum {false,true} BOOL;

STATE   next_state[SYMBOL][STATE];
BOOL    final[STATE];

BOOL
dfa(SYMBOL *input,STATE q)
{
SYMBOl  c;

while (c=*input++)
        q = next_state[c,q];
return final[q];
}
```

Figure 2.4: DFA implementation

2.3.2 Nondeterministic finite automata

A *nondeterministic finite automaton* is much like a deterministic one except that we now allow several possibilities for a transition on the same symbol from a given state. The idea is that the automaton can arbitrarily (nondeterministically) choose one of the possibilities. In addition, we will also allow ϵ**-moves** where the automaton makes a state transition (labeled by ϵ) without reading an input symbol.

Definition 4 *A **nondeterministic finite automaton** (NFA) is a tuple*

$$(Q, \Sigma, \delta, q_0, F)$$

where

- Q *is a finite set of **states**,*
- Σ *is a finite **input alphabet***
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ *is a (total) transition function* [4]
- $q_0 \in Q$ *is the **initial state***
- $F \subseteq Q$ *is the set of **final states***

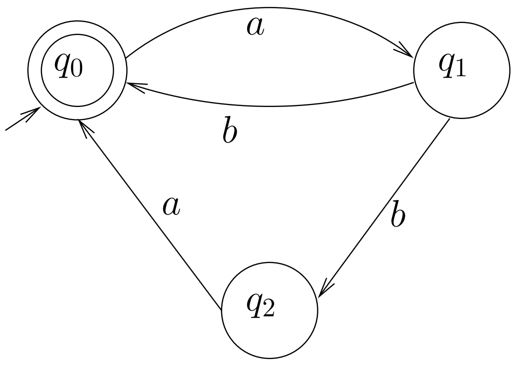


Figure 2.5: M_1

It should be noted that $\emptyset \in 2^Q$ and thus Definition 4 sanctions the possibility of there not being any transition from a state q on a given symbol a.

Example 2 Consider $M_1 = (\{q_0, q_1, q_2\}, \delta_1, q_0, \{q_0\})$ as depicted in Figure 2.5. The table below defines δ_1:

$q \in Q$	$\sigma \in \Sigma$	$\delta_1(q, \sigma)$
q_0	a	$\{q_1\}$
q_0	b	$\emptyset$
q_1	a	$\emptyset$
q_1	b	$\{q_0, q_2\}$
q_2	a	$\{q_0\}$
q_2	b	$\emptyset$

The following definition formalizes our intuition about the behavior of nondeterministic finite automata.

Definition 5 *Let $M = (Q, \Sigma, \delta, q_0, F)$ be a NFA. A **configuration** of M is a pair $(q, w) \in Q \times \Sigma^*$. For a configuration (q, aw) (where $a \in \Sigma \cup \{\epsilon\}$), we write*

$$(q, aw) \vdash_M (q', w)$$

just when $q' \in \delta(q, a)$ [5]*. The reflexive and transitive closure of the binary relation $\vdash_M$ is denoted as $\vdash_M^*$. The **language accepted by** M is defined by*

$$L(M) = \{w \mid \exists q \in F \cdot (q_0, w) \vdash_M^* (q, \epsilon)\}$$

[4]For any set X, we use 2^X to denote its power set, i.e. the set of all subsets of X.

[5]We will drop the subscript M in $\vdash_M$ if M is clear from the context.

Example 3 The following sequence shows how M_1 from Example 2 can accept the string $abaab$:

$$\begin{array}{rl} (q_0, abaab) & \vdash_{M_1} (q_1, baab) \\ & \vdash_{M_1} (q_2, aab) \\ & \vdash_{M_1} (q_0, ab) \\ & \vdash_{M_1} (q_1, b) \\ & \vdash_{M_1} (q_0, \epsilon) \end{array}$$

$$L(M_1) = \{w_0 w_1 \dots w_n \mid n \in \mathbb{N} \land \forall 0 \leq i \leq n \cdot w_i \in \{ab, aba\}\}$$

Although nondeterministic finite automata are more general than deterministic ones, it turns out that they are not more powerful in the sense that any NFA can be simulated by a DFA.

Theorem 1 *Let M be a NFA. There exists a DFA M' such that $L(M') = L(M)$.*

Proof: (sketch) Let $M = (Q, \Sigma, \delta, q_0, F)$ be a NFA. We will construct a DFA M' that simulates M. This is achieved by letting M' be "in all possible states" that M could be in (after reading the same symbols). Note that "all possible states" is always an element of 2^Q, which is finite since Q is.

To deal with ϵ-moves, we note that, if M is in a state q, it could also be in any state q' to which there is an ϵ-transition from q. This motivates the definition of the **ϵ-closure** $\mathcal{C}_\epsilon(S)$ of a set of states S:

$$\mathcal{C}_\epsilon(S) = \{p \in Q \mid \exists q \in S \cdot (q, \epsilon) \vdash^*_M (p, \epsilon)\} \tag{2.1}$$

Now we define

$$M' = (2^Q, \Sigma, \delta', s_0, F')$$

where

- δ' is defined by

$$\forall s \in 2^Q, a \in \Sigma \cdot \delta'(s, a) = \cup_{q \in s} \mathcal{C}_\epsilon(\delta(q, a)) \tag{2.2}$$

- $s_0 = \mathcal{C}_\epsilon(q_0)$, i.e. M' starts in all possible states where M could go to from q_0 without reading any input.

- $F' = \{s \in 2^Q \mid s \cap F \neq \emptyset\}$, i.e. if M *could* end up in a final state, then M' *will* do so.

It can then be shown that $L(M') = L(M)$. □

2.4 Regular expressions vs finite state automata

In this section we show how a regular expression can be translated to a nondeterministic finite automata that defines the same language. Using Theorem 1, we can then translate regular expressions to DFA's and hence to a program that accepts exactly the strings conforming to the regular expression.

Theorem 2 *Let r be a regular expression. Then there exists a NFA M_r such that $L(M_r) = L_r$.*

Proof:

We show by induction on the number of operators used in a regular expression r that L_r is accepted by an NFA

$$M_r = (Q, \Sigma, \delta, q_0, \{q_f\})$$

(where Σ is the alphabet of L_r) which has exactly one final state q_f satisfying

$$\forall a \in \Sigma \cup \{\epsilon\} \cdot \delta(q_f, a) = \emptyset \tag{2.3}$$

Base case

Assume that r does not contain any operator. Then r is one of $\emptyset$, ϵ or $a \in \Sigma$.

We then define $M_\emptyset$, M_ϵ and M_a as shown in Figure 2.6.

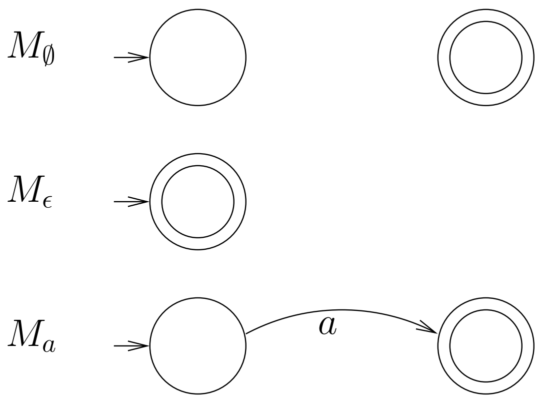


Figure 2.6: $M_\emptyset, M_\epsilon$ and M_a

Induction step

More complex regular expressions must be of one of the forms $r_1 + r_2$, $r_1 r_2$ or r_1^*. In each case, we can construct a NFA $M_{r_1+r_2}$, $M_{r_1 r_2}$ or $M_{r_1^*}$, based on M_{r_1} and M_{r_2}, as shown in Figure 2.7.

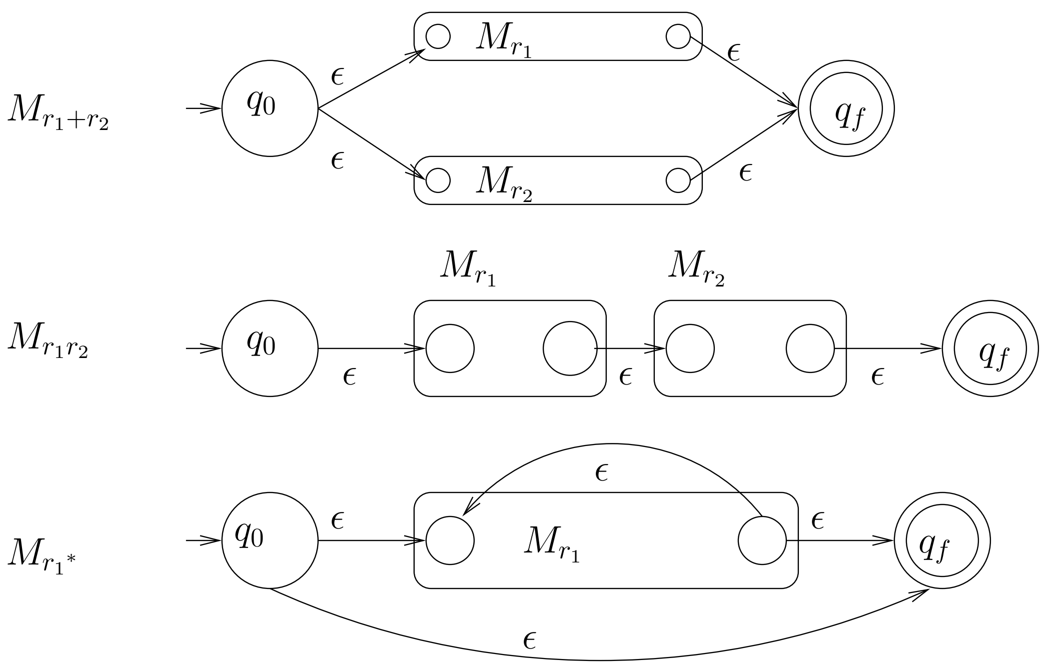


Figure 2.7: $M_{r_1+r_2}, M_{r_1r_2}$ and $M_{r_1^*}$

It can then be shown that

$$\begin{aligned} L(M_{r_1+r_2}) &= L(M_{r_1}) \cup L(M_{r_2}) \\ L(M_{r_1r_2}) &= L(M_{r_1})\, L(M_{r_2}) \\ L(M_{r_1^*}) &= L(M_{r_1})^* \end{aligned}$$

□

2.5 A scanner generator

We can now be more specific on the design and operation of a scanner generator such as *lex(1)* or *flex(1L)*, which was sketched on page 25.

First we introduce the concept of a "dead" state in a DFA.

Definition 6 *Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. A state $q \in Q$ is called **dead** if there does not exist a string $w \in \Sigma^*$ such that $(q, w) \vdash_M^* (q_f, \epsilon)$ for some $q_f \in F$.*

Example 4 The state q_e in Example 1 is dead.

It is easy to determine the set of dead states for a DFA, e.g. using a marking algorithm which initially marks all states as “dead” and then recursively works backwards from the final states, unmarking any states reached.

The generator takes as input a set of regular expressions, $R = \{r_1, \ldots, r_n\}$ each of which is associated with some code c_{r_i} to be executed when a token corresponding to r_i is recognized.

The generator will convert the regular expression

$$r_1 + r_2 + \ldots + r_n$$

to a DFA $M = (Q, \Sigma, \delta, q_0, F)$, as shown in Section 2.4, with one addition: when constructing M, it will remember which final state of the DFA corresponds with which regular expression. This can easily be done by remembering the final states in the NFA’s corresponding to each of the r_i while constructing the combined DFA M. It may be that a final state in the DFA corresponds to several patterns (regular expressions). In this case, we select the one that was defined first.

Thus we have a mapping

$$\textit{pattern} : F \to R$$

which associates the first (in the order of definition) pattern to which a certain final state corresponds. We also compute the set of dead states of M.

The code in Figure 2.8 illustrates the operation of the generated scanner.

The scanner reads input characters, remembering the last final state seen and the associated regular expression, until it hits a dead state from where it is impossible to reach a final state. It then backs up to the last final state and executes the code associated with that pattern. Clearly, this will find the longest possible token on the input.

```
typedef int     STATE;
typedef char    SYMBOL;
typedef enum {false,true} BOOL;

typedef struct { /* what we need to know about a user defined pattern */
        TOKEN*  (*code)(); /* user-defined action */
        BOOL    do_return; /* whether action returns from lex() or not */
        }       PATTERN;

static STATE    next_state[SYMBOL][STATE];
static BOOL     dead[STATE];
static BOOL     final[STATE];
static PATTERN* pattern[STATE]; /* first regexp for this final state */
static SYMBOL   *last_input = 0; /* input pointer at last final state */
static STATE    last_state, q = 0; /* assuming 0 is initial state */
static SYMBOL   *input; /* source text */

TOKEN*
lex()
{
SYMBOl  c;
PATTERN *last_pattern   = 0;

while (c=*input++) {
        q = next_state[c,q];
        if (final[q]) {
                last_pattern = pattern[q];
                last_input   = input;
                last_state   = q;
                }
        if (dead[q]) {
                if (last_pattern) {
                        input   = last_input;
                        q       = 0;
                        if (last_pattern->do_return)
                                return pattern->code();
                        else
                                pattern->code();
                        }
                else /* error */
                        ;
                }
        }

return (TOKEN*)0;
}
```

Figure 2.8: A generated scanner

Chapter 3

Parsing

3.1 Context-free grammars

As mentioned in Section 1.3.1, page 9, the rules (or railroad diagrams) used to specify the syntax of a programming language can be formalized using the concept of context-free grammar.

Definition 7 *A **context-free grammar** (cfg) is a tuple*

$$G = (V, \Sigma, P, S)$$

where

- V *is a finite set of **nonterminal** symbols*
- Σ *is a finite set of **terminal** symbols, disjoint from* V*:* $\Sigma \cap V = \emptyset$.
- P *is a finite set of **productions** of the form* $A \rightarrow \alpha$ *where* $A \in V$ *and* $\alpha \in (V \cup \Sigma)^*$
- $S \in V$ *is a nonterminal **start symbol***

Note that terminal symbols correspond to token types as delivered by the lexical analyzer.

Example 5 The following context-free grammar defines the syntax of simple arithmetic expressions:

$$G_0 = (\{E\}, \{+, \times, (,), \mathbf{id}\}, P, E)$$

where P consists of

$$\begin{array}{lcl} E & \rightarrow & E+E \\ E & \rightarrow & E \times E \\ E & \rightarrow & (E) \\ E & \rightarrow & \mathbf{id} \end{array}$$

We shall often use a shorter notation for a set of productions where several right-hand sides for the same nonterminal are written together, separated by "|". Using this notation, the set of rules of G_0 can be written as

$$E \rightarrow E+E \mid E \times E \mid (E) \mid \mathbf{id}$$

Definition 8 *Let $G = (V, \Sigma, P, S)$ be a context-free grammar. For strings $x, y \in (V \cup \Sigma)^*$, we say that x **derives** y in one step, denoted $x \Longrightarrow_G y$ iff $x = x_1 A x_2$, $y = x_1 \alpha x_2$ and $A \rightarrow \alpha \in P$. Thus $\Longrightarrow_G$ is a binary relation on $(V \cup \Sigma)^*$. The relation$\Longrightarrow_G^*$ is the reflexive and transitive closure of $\Longrightarrow_G$. The **language** $L(G)$ **generated by** G is defined by*

$$L(G) = \{w \in \Sigma^* \mid S \Longrightarrow_G^* w\}$$

*A language is called **context-free** if it is generated by some context-free grammar. A **derivation** in G of w_n from w_0 is any sequence of the form*

$$w_0 \Longrightarrow_G w_1 \Longrightarrow_G \ldots \Longrightarrow_G w_n$$

*where $n \geq 0$ (we say that the derivation has n **steps**) and $\forall 1 \leq i \leq n \cdot w_i \in (V \cup \Sigma)^*$ We write $v \Longrightarrow_G^n w$ ($n \geq 0$) when w can be derived from v in n steps.*

Thus a context-free grammar specifies precisely which sequences of tokens are valid sentences (programs) in the language.

Example 6 Consider the grammar G_0 from Example 5. The following is a derivation in G where at each step, the symbol to be rewritten is underlined.

$$\begin{array}{lll} \underline{S} & \Longrightarrow_{G_0} & \underline{E} \times E \\ & \Longrightarrow_{G_0} & (\underline{E}) \times E \\ & \Longrightarrow_{G_0} & (E + \underline{E}) \times E \\ & \Longrightarrow_{G_0} & (E + \mathbf{id}) \times \underline{E} \\ & \Longrightarrow_{G_0} & (\underline{E} + \mathbf{id}) \times \mathbf{id} \\ & \Longrightarrow_{G_0} & (\mathbf{id} + \mathbf{id}) \times \mathbf{id} \end{array}$$

A derivation in a context-free grammar is conveniently represented by a *parse tree*.

Definition 9 *Let $G = (V, \Sigma, P, S)$ be a context-free grammar. A* ***parse tree*** *corresponding to G is a labeled tree where each node is labeled by a symbol from $V \cup \Sigma$ in such a way that, if A is the label of a node and $A_1 A_2 \ldots A_n$ ($n > 0$) are the labels of its children (in left-to-right order), then*

$$A \longrightarrow A_1 A_1 \ldots A_n$$

is a rule in P. Note that a rule $A \longrightarrow \epsilon$ gives rise to a leaf node labeled ϵ.

As mentioned in Section 1.3.4, it is the job of the parser to convert a string of tokens into a parse tree that has precisely this string as yield. The idea is that the parse tree describes the syntactical structure of the source text.

However, sometimes, there are several parse trees possible for a single string of tokens, as can be seen in Figure 3.1.

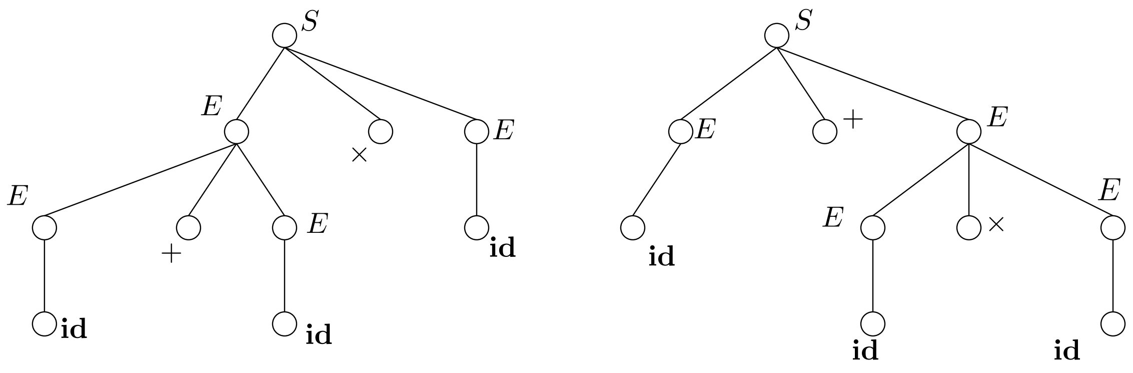


Figure 3.1: Parse trees in the ambiguous context-free grammar from Example 5

Note that the two parse trees intuitively correspond to two evaluation strategies for the expression. Clearly, we do not want a source language that is specified using an ambiguous grammar (that is, a grammar where a legal string of tokens may have different parse trees).

Example 7 Fortunately, we can fix the grammar from Example 5 to avoid such ambiguities.

$$G_1 = (\{E, T, F\}, \{+, \times, (,), \mathbf{id}\}, P', E)$$

where P' consists of

$$\begin{array}{lcl} E & \rightarrow & E+T \mid T \\ T & \rightarrow & T \times F \mid F \\ F & \rightarrow & (E) \mid \textbf{id} \end{array}$$

is an unambiguous grammar generating the same language as the grammar from Example 5.

Still, there are context-free languages such as $\{a^i b^j c^k \mid i = j \vee j = k\}$ for which only ambiguous grammars can be given. Such languages are called *inherently ambiguous*. Worse still, checking whether an arbitrary context-free grammar allows ambiguity is an unsolvable problem[HU69].

3.2 Top-down parsing

3.2.1 Introduction

When using a top-down (also called *predictive*) parsing method, the parser tries to find a *leftmost derivation* (and associated parse tree) of the source text. A leftmost derivation is a derivation where, during each step, the leftmost nonterminal symbol is rewritten.

Definition 10 *Let $G = (V, \Sigma, P, S)$ be a context-free grammar. For strings $x, y \in (V \cup \Sigma)^*$, we say that x derives y in a leftmost fashion and in one step, denoted*

$$x \overset{L}{\Longrightarrow}_G y$$

iff $x = x_1 A x_2$, $y = x_1 \alpha x_2$, $A \rightarrow \alpha$ is a rule in P and $x_1 \in \Sigma^*$ (i.e. the leftmost occurrence of a nonterminal symbol is rewritten).

The relation $\overset{L}{\Longrightarrow}{}^*_G$ is the reflexive and transitive closure of $\overset{L}{\Longrightarrow}_G$. A derivation

$$y_0 \overset{L}{\Longrightarrow}_G y_1 \overset{L}{\Longrightarrow}_G \ldots \overset{L}{\Longrightarrow}_G y_n$$

*is called a **leftmost** derivation. If $y_0 = S$ (the start symbol) then we call each y_i in such a derivation a **left sentential form**.*

Is is not hard to see that restricting to leftmost derivations does not alter the language of a context-free grammar.

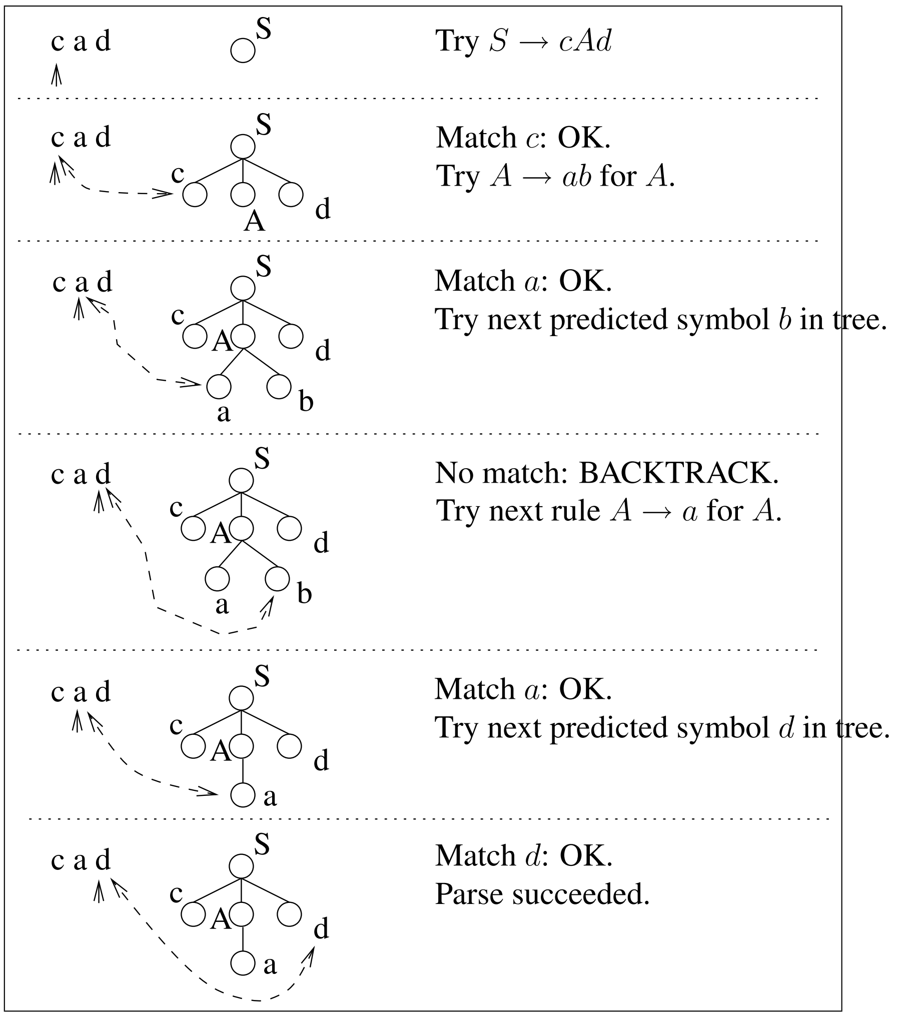


Figure 3.2: A simple top-down parse

Theorem 3 *Let $G = (V, \Sigma, P, S)$ be a context-free grammar. If $A \in V \cup \Sigma$ then*

$$A \Longrightarrow_G^* w \in \Sigma^* \textit{ iff } A \overset{L}{\Longrightarrow}_G^* w \in \Sigma^*$$

Example 8 Consider the trivial grammar

$$G = (\{S, A\}, \{a, b, c, d\}, P, S)$$

where P contains the rules

$$\begin{array}{lcl} S & \to & cAd \\ A & \to & ab \mid a \end{array}$$

Let $w = cad$ be the source text. Figure 3.2 shows how a top-down parse could proceed.

The reasoning in Example 8 can be encoded as shown below.

```
typedef enum {false,true} BOOL;

TOKEN* input; /* output array from scanner */
TOKEN* token; /* current token from input */

BOOL
parse_S() { /* Parse something derived from S */
  /* Try rule S --> c A d */
  if (*token=='c') {
    ++token;
    if (parse_A()) {
      if (*token=='d') {
        ++token;
        return true;
      }
    }
  }
  return false;
}

BOOL
parse_A() { /* Parse stuff derived from A */
  TOKEN* save; /* for backtracking */

  save = token;

  /* Try rule A --> a b */
  if (*token=='a') {
    ++token;
    if (*token=='b') {
      ++token;
      return true;
    }
  }

  token = save; /* didn't work: backtrack */

  /* Try rule A --> a */
  if (*token=='a') {
    ++token;
    return true;
  }

  token = save; /* didn't work: backtrack */

  /* no more rules: give up */
  return false;
}
```

Note that the above strategy may need recursive functions. E.g. if the grammar contains a rule such as

$$E \to (E)$$

the code for *parse_E()* will contain a call to *parse_E()*.

The method illustrated above has two important drawbacks:

- It cannot be applied if the grammar G is *left-recursive*, i.e. $A \overset{L}{\Longrightarrow}{}^*_G Ax$ for some $x \in (V \cup \Sigma)^*$.

  Indeed, even for a "natural" rule such as

  $$E \to E + E$$

  it is clear that the *parse_E()* function would start by calling itself (in an infinite recursion), before reading any input.

  We will see that it is possible to eliminate left recursion from a grammar without changing the generated language.

- Using backtracking is both expensive and difficult. It is difficult because it usually does not suffice to simply restore the input pointer: all actions taken by the compiler since the backtrack point (e.g. symbol table updates) must also be undone.

  The solution will be to apply top-down parsing only for a certain class of restricted grammars for which it can be shown that backtracking will never be necessary.

3.2.2 Eliminating left recursion in a grammar

First we look at the elimination of *immediate* left recursion where we have rules of the form

$$A \to A\alpha \mid \beta$$

where β does not start with A.

The idea is to reorganize the rules in such a way that derivations are simulated as shown in Figure 3.3

This leads to the following algorithm to remove immediate left recursion.

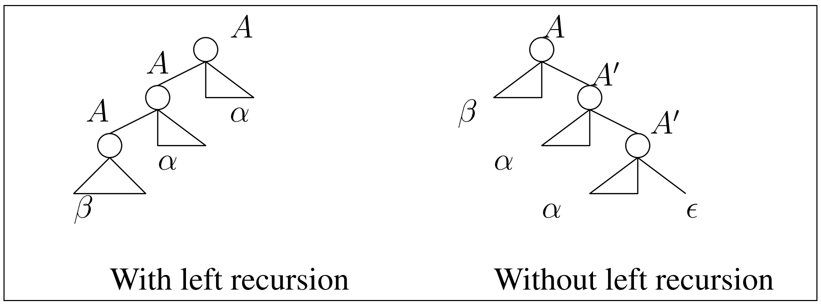


Figure 3.3: Eliminating immediate left recursion

Algorithm 1 [Removing immediate left recursion for a nonterminal A from a grammar $G = (V, \Sigma, P, S)$]
Let

$$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$$

be all the rules with A as the left hand side. Note that $m > 0$ and $n \geq 0$ and $\forall 0 \leq i \leq n \cdot \beta_i \notin A(V \cup \Sigma)^*$

1. *If $n = 0$, no terminal string can ever be derived from A, so we may as well remove all A-rules.*
2. *Otherwise, define a new nonterminal A', and replace the A rules by*

$$\begin{array}{lcl} A & \rightarrow & \beta_1 A' \mid \ldots \mid \beta_n A' \\ A' & \rightarrow & \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \epsilon \end{array}$$

$\square$

Example 9 Consider the grammar G_1 from Example 7. Applying Algorithm 1 results in the grammar $G_2 = (\{E, E', T, T', F\}, \{+, \times, (,), \mathbf{id}\}, P_{G_2}, E)$ where P_{G_2} contains

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & \times FT' \mid \epsilon \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

It is also possible (see e.g. [ASU86], page 177) to eliminate general (also indirect) left recursion from a grammar.

3.2.3 Avoiding backtracking: LL(1) grammars

When inspecting Example 8, it appears that in order to avoid backtracking, one should be able to predict with certainty, given the nonterminal X of the current function *parse_X()* and the next input token a, which production $X \rightarrow x$ was used as a first step to derive a string of tokens that starts with the input token a from X.

The problem is graphically represented in Figure 3.4

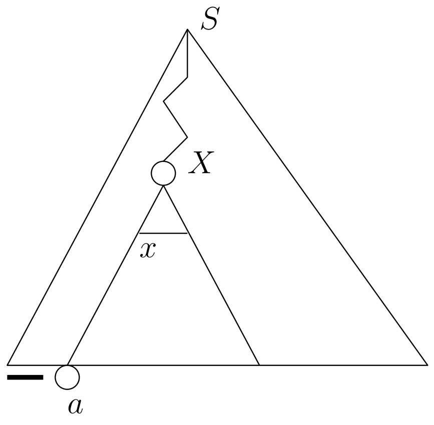


Figure 3.4: The prediction problem

The following definition defines exactly the class of context-free grammars for which this is possible.

Definition 11 *An $LL(1)$* ***grammar*** *is a context-free grammar $G = (V, \Sigma, P, S)$ such that if*

$$S \overset{L}{\Longrightarrow}{}^*_G w_1 X x_3 \overset{L}{\Longrightarrow}_G w_1 x_2 x_3 \overset{L}{\Longrightarrow}{}^*_G w_1 a w_4$$

and

$$S \overset{L}{\Longrightarrow}{}^*_G w_1 X \hat{x_3} \overset{L}{\Longrightarrow}_G w_1 \hat{x_2} \hat{x_3} \overset{L}{\Longrightarrow}{}^*_G w_1 a \hat{w_4}$$

where $a \in \Sigma$, $X \in V$, $w_1 \in \Sigma^*$, $w_4 \in \Sigma^*$, $\hat{w_4} \in \Sigma^*$ then

$$x_2 = \hat{x_2}$$

A language L is called an LL(1) language if there exists a LL(1) grammar G such that $L(G) = L$.

Intuitively, Definition 11 just says that if there are two possible choices for a production, these choices are identical.

Thus for LL(1) grammars[1], we know for sure that we can write functions as we did in Example 8, without having to backtrack. In the next section, we will see how to automatically generate parsers based on the same principle but without the overhead of (possibly recursive) function calls.

3.2.4 Predictive parsers

Predictive parsers use a stack (representing strings in $(V \cup \Sigma)^*$) and a parse table as shown in Figure 3.5.

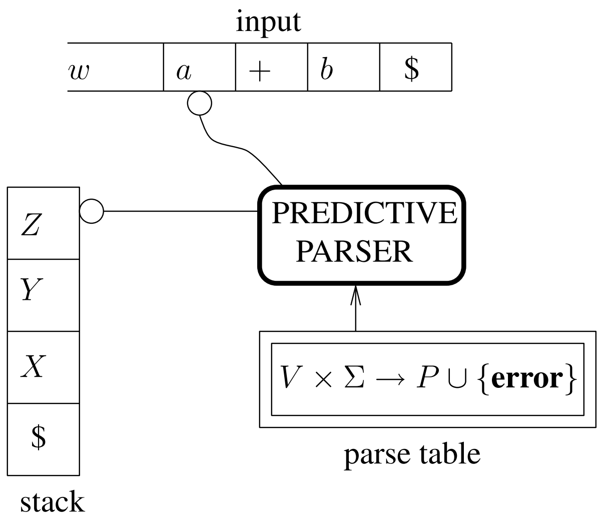


Figure 3.5: A predictive parser

The figure shows the parser simulating a leftmost derivation

$$S \overset{L}{\Longrightarrow}_G \dots \overset{L}{\Longrightarrow}_G \underbrace{wZYX}_{\text{depicted position of parser}} \overset{L}{\Longrightarrow}_G \dots \overset{L}{\Longrightarrow}_G wa+b$$

Note that the string consisting of the input read so far, followed by the contents of the stack (from the top down) constitutes a left sentential form. End markers (depicted as $ symbols) are used to mark both the bottom of the stack and the end of the input.

[1]It is possible to define LL(k) grammars and languages by replacing a in Definition 11 by a terminal string of (up to) k symbols.

The parse table M represents the production to be chosen: when the parser has X on the top of the stack and a as the next input symbol, $M[X, a]$ determines which production was used as a first step to derive a string starting with a from X.

Algorithm 2 [Operation of an LL(1) parser]
The operation of the parser is shown in Figure 3.6. □

Intuitively, a predictive parser simulates a leftmost derivation of its input, using the stack to store the part of the left sentential form that has not yet been processed (this part includes all nonterminals of the left sentential form). It is not difficult to see that if a predictive parser successfully processes an input string, then this string is indeed in the language generated by the grammar consisting of all the rules in the parse table.

To see the reverse, we need to know just how the parse table is constructed.

First we need some auxiliary concepts:

Definition 12 *Let $G = (V, \Sigma, P, S)$ be a context-free grammar.*

- *The function **first** : $(V \cup \Sigma)^* \to 2^{(\Sigma \cup \{\epsilon\})}$ is defined by*

$$\mathit{first}(\alpha) = \{a \mid \alpha \Longrightarrow_G^* aw \in \Sigma^*\} \cup X_\alpha$$

where

$$X_\alpha = \begin{cases} \{\epsilon\} & \text{if } \alpha \Longrightarrow_G^* \epsilon \\ \emptyset & \text{otherwise} \end{cases}$$

- *The function **follow** : $V \to 2^{(\Sigma \cup \{\$\})}$ is defined by*

$$\mathit{follow}(A) = \{a \in \Sigma \mid S \Longrightarrow_G^* \alpha A a \beta\} \cup Y_\alpha$$

where

$$Y_\alpha = \begin{cases} \{\$\} & \text{if } S \Longrightarrow_G^* \alpha A \\ \emptyset & \text{otherwise} \end{cases}$$

Intuitively, $\mathit{first}(\alpha)$ contains the set of terminal symbols that can appear at the start of a (terminal) string derived from α, including ϵ if α can derive the empty string.

On the other hand, $\mathit{follow}(A)$ consists of those terminal symbols that may follow a string derived from A in a terminal string from $L(G)$.

The construction of the LL(1) parse table can then be done using the following algorithm.

```
PRODUCTION*  parse_table[NONTERMINAL,TOKEN];
SYMBOL End, S;  /* marker and start symbol */
SYMBOL* stack;
SYMBOL* top_of_stack;

TOKEN   *input;

BOOL
parse() {  /* LL(1) predictive parser */
  SYMBOL  X;
  push(End); push(S);

  while (*top_of_stack!=End) {
    X = *top_of_stack;
    if (is_terminal(X)) {
      if (X==*input) { /* predicted outcome */
        ++input; /* advance input */
        pop();  /* pop X from stack */
      }
      else
        error("Expected %s, got %s", X, *input);
    }
    else { /* X is nonterminal */
      PRODUCTION* p = parse_table[X,*input];
      if (p) {
        pop(); /* pop X from stack */
        for (i=length_rhs(p)-1;(i>=0);--i)
          push(p->rhs[i]); /* push symbols of rhs, last first */
      }
      else
        error("Unexpected %s", *input);
    }
  }
  if (*input==End)
    return true;
  else
    error("Redundant input: %s", *input);
}
```

Figure 3.6: Predictive parser operation

Algorithm 3 [Construction of LL(1) parse table]
Let $G = (V, \Sigma, P, S)$ be a context-free grammar.

1. *Initialize the table: $\forall A \in V, b \in \Sigma \cdot M[A, b] = \emptyset$*

2. *For each production $A \rightarrow \alpha$ from P*

   (a) For each $b \in$ first$(\alpha) \cap \Sigma$, add $A \rightarrow \alpha$ to $M[A, b]$.

   (b) If $\epsilon \in$ first(α) then

      i. For each $c \in$ follow$(A) \cap \Sigma$, add $A \rightarrow \alpha$ to $M[A, c]$.

      ii. If $\$ \in$ follow(A) then add $A \rightarrow \alpha$ to $M[A, \$]$.

3. *If each entry in M contains at most a single production then return success, else return failure.*

$\square$

It can be shown that, if the parse table for a grammar G was successfully constructed using Algorithm 3, then the parser of Algorithm 2 accepts exactly the strings of $L(G)$. Also, if Algorithm 3 fails, then G is not an LL(1) grammar.

Example 10 Consider grammar G_2 from Example 9. The *first* and *follow* functions are computed in Examples 11 and 12. The results are summarized below.

	E	E'	T	T'	F
first	$(, \textbf{id}$	$+, \epsilon$	$(, \textbf{id}$	$\times, \epsilon$	$(, \textbf{id}$
follow	$\$,)$	$\$,)$	$+, \$,)$	$\$,), +$	$\times, \$,), +$

Applying Algorithm 3 yields the following LL(1) parse table.

	E	E'	T	T'	F
id	$E \rightarrow TE'$		$T \rightarrow FT'$		$F \rightarrow \textbf{id}$
$+$		$E' \rightarrow +TE'$		$T' \rightarrow \epsilon$	
$\times$				$T' \rightarrow \times FT'$	
$($	$E \rightarrow TE'$		$T \rightarrow FT'$		$F \rightarrow (E)$
$)$		$E' \rightarrow \epsilon$		$T' \rightarrow \epsilon$	
$\$$		$E' \rightarrow \epsilon$		$T' \rightarrow \epsilon$	

The operation of a predictive parser using the above table is illustrated below for the input string $\mathbf{id} + \mathbf{id} \times \mathbf{id}$.

stack	input	rule
$\$E$	$\mathbf{id} + \mathbf{id} \times \mathbf{id}\$$	
$\$E'T$	$\mathbf{id} + \mathbf{id} \times \mathbf{id}\$$	$E \rightarrow TE'$
$\$E'T'F$	$\mathbf{id} + \mathbf{id} \times \mathbf{id}\$$	$T \rightarrow FT'$
$\$E'T'\mathbf{id}$	$\mathbf{id} + \mathbf{id} \times \mathbf{id}\$$	$F \rightarrow \mathbf{id}$
$\$E'T'$	$+\mathbf{id} \times \mathbf{id}\$$	
$\$E'$	$+\mathbf{id} \times \mathbf{id}\$$	$T' \rightarrow \epsilon$
$\$E'T+$	$+\mathbf{id} \times \mathbf{id}\$$	$E' \rightarrow +TE'$
$\$E'T$	$\mathbf{id} \times \mathbf{id}\$$	
$\$E'T'F$	$\mathbf{id} \times \mathbf{id}\$$	$T \rightarrow FT'$
$\$E'T'\mathbf{id}$	$\mathbf{id} \times \mathbf{id}\$$	$F \rightarrow \mathbf{id}$
$\$E'T'$	$\times\mathbf{id}\$$	
$\$E'T'F\times$	$\times\mathbf{id}\$$	$T' \rightarrow \times FT'$
$\$E'T'F$	$\mathbf{id}\$$	
$\$E'T'\mathbf{id}$	$\mathbf{id}\$$	$F \rightarrow \mathbf{id}$
$\$E'T'$	$\$$	
$\$E'$	$\$$	$T' \rightarrow \epsilon$
$\$$	$\$$	$E' \rightarrow \epsilon$

3.2.5 Construction of first and follow

Algorithm 4 [Construction of first]
Let $G = (V, \Sigma, P, S)$ be a context-free grammar. We construct an array $F : V \cup \Sigma \rightarrow 2^{(\Sigma \cup \{\epsilon\})}$ and then show a function to compute $first(\alpha)$ for arbitrary $\alpha \in (V \cup \Sigma)^*$.

1. *F is constructed via a fixpoint computation:*
   - *(a) for each $X \in V$, initialize $F[X] \leftarrow \{a \mid X \rightarrow a\alpha, a \in \Sigma\}$*
   - *(b) for each $a \in \Sigma$, initialize $F[a] \leftarrow \{a\}$*
2. *repeat the following steps*
   - *(a) $F' \leftarrow F$*
   - *(b) for each rule $X \rightarrow \epsilon$, add ϵ to $F[X]$.*
   - *(c) for each rule $X \rightarrow Y_1 \dots Y_k$ ($k > 0$) do*
     - *i. for each $1 \leq i \leq k$ such that $Y_1 \dots Y_{i-1} \in V^*$ and $\forall 1 \leq j < i \cdot \epsilon \in F[Y_j]$ do $F[X] \leftarrow F[X] \cup (F[Y_i] \setminus \{\epsilon\})$*
     - *ii. if $\forall 1 \leq j \leq k \cdot \epsilon \in F[Y_i]$ then $F[X] \leftarrow F[X] \cup \{\epsilon\}$*

until $F' = F$

Define

$$\mathit{first}(X_1 \dots X_n) = \begin{cases} \bigcup_{1 \le j \le (k+1)} (F[X_j] \setminus \{\epsilon\}) & \text{if } k(X_1 \dots X_n) < n \\ \bigcup_{1 \le j \le n} F[X_j] \cup \{\epsilon\}) & \text{if } k(X_1 \dots X_n) = n \end{cases}$$

where $k(X_1 \dots X_n)$ *is the largest index* k *such that* $X_1 \dots X_k \Longrightarrow_G^* \epsilon$, *i.e.*

$$k(X_1 \dots X_n) = \max\{1 \le i \le n \mid \forall 1 \le j \le i \cdot \epsilon \in F[X_j]\}$$

□

Example 11 Consider grammar G_2 from Example 9.

The construction of F is illustrated in the following table.

E	E'	T	T'	F	
	$+$		$\times$	$(, \mathbf{id}$	initialization
	ϵ				rule $E' \rightarrow \epsilon$
		$(, \mathbf{id}$			rule $T \rightarrow FT'$
			ϵ		rule $T' \rightarrow \epsilon$
$(, \mathbf{id}$					rule $E \rightarrow TE'$
$(, \mathbf{id}$	$+, \epsilon$	$(, \mathbf{id}$	$\times, \epsilon$	$(, \mathbf{id}$	

Algorithm 5 [Construction of follow]
Let $G = (V, \Sigma, P, S)$ *be a context-free grammar. The* follow *set of all symbols in* V *can be computed as follows:*

1. $\mathit{follow}(S) \leftarrow \$$

2. *Apply the following rules until nothing can be added to* $\mathit{follow}(A)$ *for any* $A \in V$.

   (a) *If there is a production* $A \rightarrow \alpha B \beta$ *in* P *then*

   $$\mathit{follow}(B) \leftarrow \mathit{follow}(B) \cup (\mathit{first}(\beta) \setminus \{\epsilon\})$$

   (b) *If there is a production* $A \rightarrow \alpha B$ *or a production* $A \rightarrow \alpha B \beta$ *where* $\epsilon \in \mathit{first}(\beta)$ *then* $\mathit{follow}(B) \leftarrow \mathit{follow}(B) \cup \mathit{follow}(A)$.

□

Example 12 Consider again grammar G_2 from Example 9.

The construction of *follow* is illustrated in the following table.

E	E'	T	T'	F	
$\$$					initialization
		$+$			rule $E \rightarrow TE'$
$)$					rule $F \rightarrow (E)$
				$\times$	rule $T \rightarrow FT'$
	$\$,)$				rule $E \rightarrow TE'$
		$\$,)$			rule $E \rightarrow TE'$
			$\$,), +$		rule $T \rightarrow FT'$
				$\$,), +$	rule $T \rightarrow FT'$
$\$,)$	$\$,)$	$+, \$,)$	$\$,), +$	$\times, \$,), +$	

3.3 Bottom-up parsing

3.3.1 Shift-reduce parsers

As described in Section 3.2, a top-down parser simulates a leftmost derivation in a top-down fashion. This simulation predicts the next production to be used, based on the nonterminal to be rewritten and the next input symbol.

A bottom-up parser simulates a rightmost derivation in a bottom-up fashion, where a rightmost derivation is defined as follows.

Definition 13 *Let $G = (V, \Sigma, P, S)$ be a context-free grammar. For strings $x, y \in (V \cup \Sigma)^*$, we say that x derives y in a rightmost fashion and in one step, denoted*

$$x \overset{R}{\Longrightarrow}_G y$$

iff $x = x_1 A x_2$, $y = x_1 \alpha x_2$, $A \rightarrow \alpha$ is a rule in P and $x_2 \in \Sigma^*$ (i.e. the rightmost occurrence of a nonterminal symbol is rewritten).

The relation $\overset{R}{\Longrightarrow}^*_G$ is the reflexive and transitive closure of $\overset{R}{\Longrightarrow}_G$. A derivation

$$x_0 \overset{R}{\Longrightarrow}_G x_1 \overset{R}{\Longrightarrow}_G \ldots \overset{R}{\Longrightarrow}_G x_n$$

*is called a **rightmost** derivation. If $x_0 = S$ (the start symbol) then we call each x_i in such a derivation a **right sentential form**.*

A ***phrase*** *of a right sentential form is a sequence of symbols that have been derived from a single nonterminal symbol occurrence (in an earlier right sentential form of the derivation). A* ***simple phrase*** *is a phrase that contains no smaller phrase. The* ***handle*** *of a right sentential form is its leftmost simple phrase (which is unique). A prefix of a right sentential form that does not extend past its handle is called a* ***viable prefix*** *of G.*

Example 13 Consider the grammar

$$G_3 = (\{S, E, T, F\}, \{+, \times, (,), \mathbf{id}\}, P_{G_3}, S)$$

where P_{G_3} consists of

$$\begin{array}{lcl} S & \rightarrow & E \\ E & \rightarrow & E + T \mid T \\ T & \rightarrow & T \times F \mid F \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

G_3 is simply G_1 from Example 7, augmented by a new start symbol whose only use is at the left hand side of a single production that has the old start symbol as its right hand side.

The following table shows a rightmost derivation of $\mathbf{id} + \mathbf{id} \times \mathbf{id}$; where the handle in each right sentential form is underlined.

$$\begin{array}{l} S \\ \underline{E} \\ \underline{E + T} \\ E + \underline{T \times F} \\ E + T \times \underline{\mathbf{id}} \\ E + \underline{F} \times \mathbf{id} \\ E + \underline{\mathbf{id}} \times \mathbf{id} \\ \underline{T} + \mathbf{id} \times \mathbf{id} \\ \underline{F} + \mathbf{id} \times \mathbf{id} \\ \underline{\mathbf{id}} + \mathbf{id} \times \mathbf{id} \end{array}$$

When doing a bottom-up parse, the parser will use a stack to store the right sentential form up to the handle, using **shift** operations to push symbols from the input onto the stack. When the handle is completely on the stack, it will be popped and replaced by the left hand side of the production, using a **reduce** operation.

Example 14 The table below illustrates the operation of a shift-reduce parser to simulate the rightmost derivation from Example 13. Note that the parsing has to be read from the bottom to the top of the figure and that both the stack and the input contain the end marker "$".

derivation	stack	input	operation
S	$\$S$	$\$$	accept
$\underline{E}$	$\$E$	$\$$	reduce $S \rightarrow E$
$\underline{E+T}$	$\$E+T$	$\$$	reduce $E \rightarrow E+T$
$E+\underline{T \times F}$	$\$E+T \times F$	$\$$	reduce $T \rightarrow T \times F$
	$\$E+T \times \mathbf{id}$	$\$$	reduce $F \rightarrow \mathbf{id}$
	$\$E+T\times$	$\mathbf{id}\$$	shift
$E+T \times \underline{\mathbf{id}}$	$\$E+T$	$\times\mathbf{id}\$$	shift
$E+\underline{F} \times \mathbf{id}$	$\$E+F$	$\times\mathbf{id}\$$	reduce $T \rightarrow F$
	$\$E+\mathbf{id}$	$\times\mathbf{id}\$$	reduce $F \rightarrow \mathbf{id}$
	$\$E+$	$\mathbf{id} \times \mathbf{id}\$$	shift
$E+\underline{\mathbf{id}} \times \mathbf{id}$	$\$E$	$+\mathbf{id} \times \mathbf{id}\$$	shift
$\underline{T}+\mathbf{id} \times \mathbf{id}$	$\$T$	$+\mathbf{id} \times \mathbf{id}\$$	reduce $E \rightarrow T$
$\underline{F}+\mathbf{id} \times \mathbf{id}$	$\$F$	$+\mathbf{id} \times \mathbf{id}\$$	reduce $T \rightarrow F$
	$\$\mathbf{id}$	$+\mathbf{id} \times \mathbf{id}\$$	reduce $F \rightarrow \mathbf{id}$
$\underline{\mathbf{id}}+\mathbf{id} \times \mathbf{id}$	$\$$	$\mathbf{id}+\mathbf{id} \times \mathbf{id}\$$	shift

Clearly, the main problem when designing a shift-reduce parser is to decide when to shift and when to reduce. Specifically, the parser should reduce only when the handle is on the top of the stack. Put in another way, the stack should always contain a **viable prefix**.

We will see in Section 3.3.2 that the language of all viable prefixes of a context-free grammar G is regular, and thus there exists a DFA $M_G = (Q, V \cup \Sigma, \delta, q_0, F)$ accepting exactly the viable prefixes of G. The operation of a shift-reduce parser is outlined in Figure 3.7[2].

However, rather than checking each time the contents of the stack (and the next input symbol) vs. the DFA M_G, it is more efficient to store the states together with the symbols on the stack in such a way that for each prefix xX of (the contents of) the stack, we store the state $\delta^*(q_0, xX)$ together with X on the stack.

In this way, we can simply check $\delta_{M_G}(q, a)$ where $q \in Q$ is the state on the top of the stack and a is the current input symbol, to find out whether shifting the input would still yield a viable prefix on the stack. Similarly, upon a $\mathit{reduce}(A \rightarrow \alpha)$

[2]We assume, without losing generality, that the start symbol of G does not occur on the right hand side of any production

```
while (true) {
  if ((the top of the stack contains the start symbol) &&
      (the current input symbol is the endmarker '$'))
    accept;
  else if (the contents of the stack concatenated with the next
           input symbol is a viable prefix)
    shift the input onto the top of the stack;
  else
    if (there is an appropriate producion)
      reduce by this production;
    else
      error;
  }
```

Figure 3.7: Naive shift-reduce parsing procedure

operation, both A and $\delta(q, A)$ are pushed where q is the state on the top of the stack after popping α.

In practice, a shift-reduce parser is driven by two tables:

- An **action** table that associates a pair consisting of a state $q \in Q$ and a symbol $a \in \Sigma$ (the input symbol) with an action of one of the following types:
  - $accept$, indicating that the parse finished successfully
  - $\textit{shift}(q')$, where $q \in Q$ is a final state of M_G. This action tells the parser to push both the input symbol and q on top of the stack. In this case $\delta_{M_G}(q, a) = q'$.
  - $reduce(A \rightarrow \alpha)$ where $A \rightarrow \alpha$ is a production from G. This instructs the parser to pop α (and associated states) from the top of the stack, then pushing A and the state q' given by $q' = goto(q'', A)$ where q'' is the state left on the top of the stack after popping α. Here $goto(q'', A) = \delta_{M_G}(q'', A)$.
  - **error**, indicating that $\delta_{M_G}(q, a)$ yields a dead state.
- A **goto** table:
$$goto : Q \times V \rightarrow Q$$
  which is used during a reduce operation.

The architecture of a shift-reduce parser is shown in Figure 3.8.

Using the *action* and *goto* tables, we can refine the shift-reduce parsing algorithm as follows.

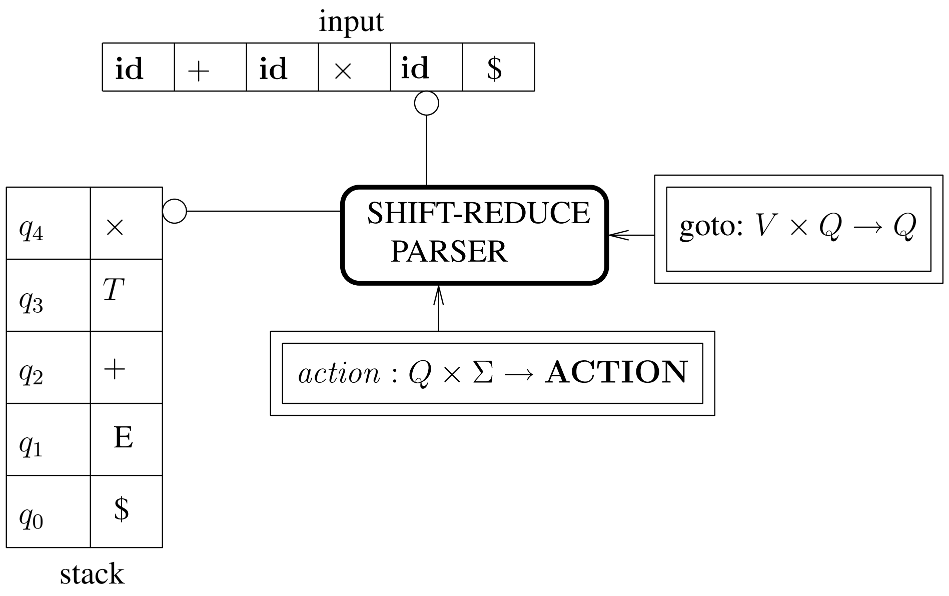

input
id
+
id
×
id
$
SHIFT-REDUCE PARSER
goto: $V \times Q \rightarrow Q$
$action : Q \times \Sigma \rightarrow$ **ACTION**
q_4 ×
q_3 T
q_2 +
q_1 E
q_0 $
stack

Figure 3.8: A shift-reduce parser

Algorithm 6 [Shift-reduce parsing]
See Figure 3.9 on page 55. □

3.3.2 LR(1) parsing

Definition 14 *Let $G = (V, \Sigma, P, S')$ be a context-free grammar such that $S' \rightarrow S$ is the only production for S' and S' does not appear on the right hand side of any production.*

- *An **LR(1) item** is a pair $[A \rightarrow \alpha \bullet \beta, a]$ where $A \rightarrow \alpha\beta$ is a production from P, $\bullet$ is a new symbol and $a \in \Sigma \cup \{\$\}$ is a terminal or end marker symbol. We use $\mathbf{items}_G$ to denote the set of all items of G.*
- *The **LR(1) viable prefix NFA** of G is*

$$N_G = (\mathbf{items}_G, V \cup \Sigma, \delta, [S' \rightarrow \bullet S], \mathbf{items}_G)$$

where δ is defined by

$$\begin{array}{lcll} [A \rightarrow \alpha X \bullet \beta, a] & \in & \delta([A \rightarrow \alpha \bullet X\beta, a], X) & \textit{for all } X \in V \cup \Sigma \\ [X \rightarrow \bullet\gamma, b] & \in & \delta([A \rightarrow \alpha \bullet X\beta, a], \epsilon) & \textit{if } X \rightarrow \gamma \textit{ is in } P \textit{ and } b \in \textit{first}(\beta a) \end{array}$$

```
TOKEN* input; /* array of input tokens */
STATE* states; /* (top of) stack of states */
SYMBOL* symbols; /* (top of) stack of symbols */

BOOL
parse() {
  while (true)
    switch (action[*states,*input]) {
      case shift(s):
        *++symbols = *input++; /* push symbol */
        *++states  = s; /* push state */
        break;
      case reduce(X->x):
        symbols -= length(x); /* pop rsh */
        states  -= length(x); /* also for states */
        STATE  s = goto[*states,X];
        *++states = s; *++symbols = X;
        break;
      case accept:
        return true;
      case error:
        return false;
    }
}
```

Figure 3.9: Shift-reduce parser algorithm

Note that all states in N_G are final. This does not mean that N_G accepts any string since there may be no transitions possible on certain inputs from certain states.

Intuitively, an item $[A \to \alpha \bullet \beta, a]$ can be regarded as describing the state where the parser has α on the top of the stack and next expects to process the result of a derivation from β, followed by an input symbol a.

To see that $L(N_G)$ accepts all viable prefixes. Consider a (partial)[3] rightmost derivation

$$S = X_0 \overset{R}{\Longrightarrow}_G x_0 X_1 y_0 \overset{R}{\Longrightarrow}{}^*_G x_0 x_1 X_2 y_1 \overset{R}{\Longrightarrow}{}^*_G x_0 \ldots x_{n-1} X_n y_n \overset{R}{\Longrightarrow}_G x_0 \ldots x_{n-1} x_n y_n$$

where $y_i \in \Sigma^*$ for all $0 \leq i \leq n$, as depicted in Figure 3.10 where we show only the steps involving the ancestors of the left hand side (X_n) of the last step (note that none of the x_i changes after being produced because we consider a rightmost derivation).

[3]By a partial rightmost derivation we mean a rightmost derivation that can be extended to a successful full rightmost derivation.

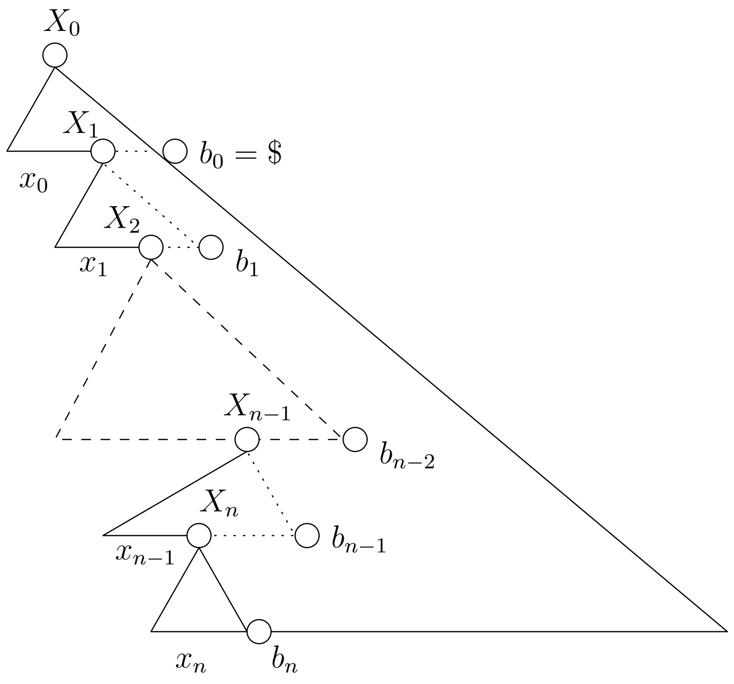


Figure 3.10: A (partial) rightmost derivation

N_G can accept any viable prefix of the right sentential form $x_0 \dots x_{n-1}x_ny_n$, i.e. any prefix of $x_0 \dots x_{n-1}x_n$ as follows:

$[S' \rightarrow \bullet X_0, \$]$	the initial state
$[X_0 \rightarrow \bullet x_0X_1z_0, b_0]$	using an ϵ-move, $b_0 \in \mathit{first}(\epsilon\$) = \{\$\}$
$[X_0 \rightarrow x_0 \bullet X_1z_0, b_0]$	reading x_0
$[X_1 \rightarrow \bullet x_1X_2z_1, b_1]$	using an ϵ-move, $b_1 \in \mathit{first}(z_0b_0)$
$[X_1 \rightarrow x_1 \bullet X_2z_1, b_1]$	reading x_1
...	
$[X_{n-1} \rightarrow \bullet x_{n-1}X_nz_{n-1}, b_{n-1}]$	
$[X_{n-1} \rightarrow x_{n-1} \bullet X_nz_{n-1}, b_{n-1}]$	reading x_{n-1}
$[X_n \rightarrow \bullet x_n, b_n]$	using an ϵ-move, $b_n \in \mathit{first}(z_{n-1}b_{n-1})$
...	any prefix of x_n can be read

Note that all b_i exist since each z_i derives a terminal string.

To see the reverse, it suffices to note that each accepting sequence in M_G is of the form sketched above and thus, a partial rightmost derivation like the one in Figure 3.10 can be inferred from it.

Theorem 4 *Let $G = (V, \Sigma, P, S')$ be a context-free grammar and let N_G be its viable prefix NFA. Then $L(N_G)$ contains exactly the viable prefixes of G.*

Because of Theorem 1, we can convert N_G to an equivalent DFA M_G that can be used to build the *action* and *goto* tables of a shift-reduce parser as follows.

Algorithm 7 [Construction of LR(1) tables]
Let

$$M_G = (2^{\mathbf{items}_G}, V \cup \Sigma, \delta, \mathcal{C}_\epsilon([S' \to \bullet S, \$]), 2^{\mathbf{items}_G} \setminus \{\emptyset\})$$

be the DFA corresponding to the viable prefix automaton N_G of G.

- *Add an action $accept$ to $action[s, \$]$ whenever $[S' \to S\bullet, \$] \in s$.*
- *Add an action $shift(s')$ to $action[s, a]$ whenever $\delta(s, a) = s'$.*
- *Add an action $reduce(A \to \alpha)$ to $action[s, a]$ whenever s contains an item $[A \to \alpha\bullet, a]$ where $A \neq S'$.*
- *For any $X \in V$, s a state of M_G, define $goto(s, X) = \delta(s, X)$.*

After the above rules have been exhaustively applied, add **error** *to all entries in $action$ that remain empty.*

The algorithm succeeds if $action[s, a]$ contains exactly one action for each reachable $s \in 2^{\mathbf{items}_G}$, $a \in \Sigma$. □

Note that it follows from Theorem 4 and Algorithm 7 that the resulting parser will announce an error at the latest possible moment, that is only when shifting would result in a stack that does not represent a viable prefix.

If Algorithm 7 does not succeed, there exists either a **shift-reduce conflict**, when an entry in the action table contains both a shift and a reduce operation, or a **reduce-reduce** conflict, when an entry in the action table contains two or more reduce actions.

Example 15 Consider G_3 from Example 13. The DFA accepting the viable prefixes of G is shown in Figures 3.11 and 3.12 Note that $[A \to \alpha,\ abc]$ is used as shorthand for $\{[A \to \alpha,\ a], [A \to \alpha,\ b], [A \to \alpha,\ c]$

Theorem 5 *Let $G = (V, \Sigma, P, S')$ be a context-free grammar such that Algorithm 7 succeeds. Then Algorithm 6 accepts exactly $L(G)$.*

$State$	$Items$		$State$	$Items$	
0	$S' \rightarrow \bullet S$	$\$$	11	$E \rightarrow E + T \bullet$	$+\$$
	$S \rightarrow \bullet E$	$\$$		$T \rightarrow T \bullet \times F$	$\times + \$$
	$E \rightarrow \bullet E + T$	$+\$$	12	$F \rightarrow (E \bullet)$	$\times + \$$
	$E \rightarrow \bullet T$	$+\$$		$E \rightarrow E \bullet + T$	$)+$
	$T \rightarrow \bullet T \times F$	$\times + \$$	13	$F \rightarrow (E) \bullet$	$\times + \$$
	$T \rightarrow \bullet F$	$\times + \$$	14	$T \rightarrow F \bullet$	$) \times +$
	$F \rightarrow \bullet \mathbf{id}$	$\times + \$$	15	$F \rightarrow (E) \bullet$	$) + \times$
	$F \rightarrow \bullet (E)$	$\times + \$$	16	$F \rightarrow (E \bullet)$	$) + \times$
1	$S \rightarrow E \bullet$	$\$$		$E \rightarrow E \bullet + T$	$)+$
	$E \rightarrow E \bullet + T$	$+\$$	17	$E \rightarrow E + \bullet T$	$)+$
2	$S' \rightarrow S \bullet$	$\$$		$T \rightarrow \bullet T \times F$	$) + \times$
3	$E \rightarrow E + \bullet T$	$+\$$		$T \rightarrow \bullet F$	$) + \times$
	$T \rightarrow \bullet T \times F$	$\times + \$$		$F \rightarrow \bullet \mathbf{id}$	$) + \times$
	$T \rightarrow \bullet F$	$\times + \$$		$F \rightarrow \bullet (E)$	$) + \times$
	$F \rightarrow \bullet \mathbf{id}$	$\times + \$$	18	$F \rightarrow (\bullet E)$	$) + \times$
	$F \rightarrow \bullet (E)$	$\times + \$$		$E \rightarrow \bullet E + T$	$)+$
4	$T \rightarrow F \bullet$	$\times + \$$		$E \rightarrow \bullet T$	$)+$
5	$F \rightarrow \mathbf{id} \bullet$	$\times + \$$		$T \rightarrow \bullet T \times F$	$) + \times$
6	$F \rightarrow (\bullet E)$	$\times + \$$		$T \rightarrow \bullet F$	$) + \times$
	$E \rightarrow \bullet E + T$	$)+$		$F \rightarrow \bullet \mathbf{id}$	$) + \times$
	$E \rightarrow \bullet T$	$)+$		$F \rightarrow \bullet (E)$	$) + \times$
	$T \rightarrow \bullet T \times F$	$) \times +$	19	$E \rightarrow T \bullet$	$)+$
	$T \rightarrow \bullet F$	$) \times +$		$T \rightarrow T \bullet \times F$	$) + \times$
	$F \rightarrow \bullet \mathbf{id}$	$) \times +$	20	$E \rightarrow E + T \bullet$	$)+$
	$F \rightarrow \bullet (E)$	$) \times +$		$T \rightarrow T \bullet \times F$	$) + \times$
7	$E \rightarrow T \bullet$	$+\$$	21	$T \rightarrow T \times \bullet F$	$) + \times$
	$T \rightarrow T \bullet \times F$	$\times + \$$		$F \rightarrow \bullet \mathbf{id}$	$) + \times$
8	$T \rightarrow T \times \bullet F$	$\times + \$$		$F \rightarrow \bullet (E)$	$) + \times$
	$F \rightarrow \bullet \mathbf{id}$	$\times + \$$	22	$T \rightarrow T \times F \bullet$	$) + \times$
	$F \rightarrow \bullet (E)$	$\times + \$$			
9	$T \rightarrow T \times F \bullet$	$\times + \$$			
10	$F \rightarrow \mathbf{id} \bullet$	$) \times +$			

Figure 3.11: States of the viable prefix DFA of G_3

Example 16 The action and goto tables of the LR(1) parser for G_3 are shown in Figure 3.13. The rules of G_3 have been numbered as follows:

1	$S \rightarrow E$	5	$T \rightarrow F$
2	$E \rightarrow E + T$	6	$F \rightarrow \mathbf{id}$
3	$E \rightarrow T$	7	$F \rightarrow (E)$
4	$T \rightarrow T \times F$		

A parse of

$$\mathbf{id} + \mathbf{id} \times \mathbf{id}$$

$\delta_{M_{G_3}}$	S	E	T	F	**id**	(	)	$+$	$\times$
0	2	1	7	4	5	6			
1								3	
2									
3			11	4	5	6			
4									
5									
6			19	14	10	18			
7									8
8				9	5	6			
9									
10									
11									8
12							13	17	
13									
14									
15									
16								17	
17			20	14	10	18			
18			19	14	10	18			
19									21
20									21
21				22	10				
22									

Figure 3.12: Transition function of the viable prefix DFA of G_3

is shown below (states on the stack are between "[]").

stack	input	operation
$\$[0]$	$\mathbf{id} + \mathbf{id} \times \mathbf{id}\$$	shift
$\$[0]\mathbf{id}[5]$	$+\mathbf{id} \times \mathbf{id}\$$	reduce $F \to \mathbf{id}$
$\$[0]F[4]$	$+\mathbf{id} \times \mathbf{id}\$$	reduce $T \to F$
$\$[0]T[7]$	$+\mathbf{id} \times \mathbf{id}\$$	reduce $E \to T$
$\$[0]E[1]$	$+\mathbf{id} \times \mathbf{id}\$$	shift
$\$[0]E[1] + [3]$	$\mathbf{id} \times \mathbf{id}\$$	shift
$\$[0]E[1] + [3]\mathbf{id}[5]$	$\times\mathbf{id}\$$	reduce $F \to \mathbf{id}$
$\$[0]E[1] + [3]F[4]$	$\times\mathbf{id}\$$	reduce $T \to F$
$\$[0]E[1] + [3]T[11]$	$\times\mathbf{id}\$$	shift
$\$[0]E[1] + [3]T[11] \times [8]$	$\mathbf{id}\$$	shift
$\$[0]E[1] + [3]T[11] \times [8]\mathbf{id}[5]$	$\$$	reduce $F \to \mathbf{id}$
$\$[0]E[1] + [3]T[11] \times [8]F[9]$	$\$$	reduce $T \to T \times F$
$\$[0]E[1] + [3]T[11]$	$\$$	reduce $E \to E + T$
$\$[0]E[1]$	$\$$	reduce $S \to E$
$\$[0]S[2]$	$\$$	accept

It will turn out that Algorithm 7 succeeds exactly for so-called LR(1) grammars.

	goto				*action*					
$state$	S	E	T	F	**id**	(	)	$+$	$\times$	$
0	2	1	7	4	$s5$	$s6$				
1								$s3$		$r1$
2										**a**
3			11	4	$s5$	$s6$				
4								$r5$	$r5$	$r5$
5								$r6$	$r6$	$r6$
6			19	14	$s10$	$s18$				
7								$r3$	$s8$	$r3$
8				9	$s5$	$s6$				
9								$r4$	$r4$	$r4$
10							$r6$	$r6$	$r6$	
11								$r2$	$s8$	$r2$
12							$s13$	$s17$		
13								$r7$	$r7$	$r7$
14							$r5$	$r5$	$r5$	
15							$r7$	$r7$	$r7$	
16								$s17$		
17			20	14	$s10$	$s18$				
18			19	14	$s10$	$s18$				
19							$r3$	$r3$	$s21$	
20							$r2$	$r2$	$s21$	
21				22	$s10$					
22							$r4$	$r4$	$r4$	

Figure 3.13: LR(1) tables for G_3

Definition 15 *An $LR(1)$ **grammar** is a context-free grammar $G = (V, \Sigma, \delta, P, S)$ such that S does not appear at the right hand side of any production. Moreover, if*

$$S \overset{R}{\Longrightarrow}^*_G x_1 X w_3 \overset{R}{\Longrightarrow}_G x_1 x_2 w_3$$

and

$$S \overset{R}{\Longrightarrow}^*_G \hat{x_1} \hat{X} \hat{w_3} \overset{R}{\Longrightarrow}_G \hat{x_1} \hat{x_2} \hat{w_3}$$

and

$$\mathit{pref}_{l+1}(x_1 x_2 w_3) = \mathit{pref}_{l+1}(\hat{x_1} \hat{x_2} \hat{w_3})$$

where $X \in V$, $w_3 \hat{w_3} \in \Sigma^*$, $l = |x_1 x_2|$, then

$$x_1 = \hat{x_1} \wedge \hat{X} = X$$

A language L is called LR(1) if there exists an LR(1) context-free grammar G such that $L(G) = L$.

Intuitively, in the above definition, $x_1 x_2$ represents the stack just before a reduce operation in a shift-reduce parser. The definition then says that there is only a single choice for when to reduce, based only on the next input symbol[4].

Theorem 6 *A context-free grammar $G = (V, \Sigma, \delta, P, S)$, where S does not appear on the right hand side of any production, is LR(1) iff Algorithm 7 succeeds.*

One may wonder about the relationship between LL(k) and LR(k) grammars and languages. It turns out (see e.g. [Sal73]) that every LR(k) ($k \geq 0$) language can be generated by an LR(1) grammar. On the other hand, there are context free languages, e.g. the inherently ambiguous language of Section 3.1, page 38, that are not LR(1). In fact, it can be shown that all deterministic context-free languages[5] are LR(1) (and, obviously, also the reverse). As for LL(k) languages, they form a proper hierarchy in that for every $k > 1$ there is an LL(k) language L_k for which no LL($k - 1$) grammar exists. Still, every LL(k) language is LR(1).

Summarizing, as far as parsing power is concerned, bottom-up (LR(1)) parsers are provably superior to top-down (LL(1)) parsers.

[4]One can also define LR(k) grammars and languages for any $k \geq 0$. It suffices to replace $l + 1$ by $l + k$ in Definition 15

[5]A context-free language is deterministic if it can be accepted by a deterministic pushdown automaton.

3.3.3 LALR parsers and yacc/bison

LR(1) parsing tables can become quite large due to the large number of states in the viable prefix DFA. A DFA for a language like Pascal may have thousands of states.

In an LALR parser, one tries to reduce the number of states by merging states that have the same (extended) productions in their items. E.g. in the DFA of Example 15, one may merge, among others, state 3 and state 17.

It is then not hard to see that, due to the way states are constructed using ϵ-closures, if s_1 and s_2 can be merged, then so can $\delta(s_1, X)$ and $\delta(s_2, X)$. This yields a more compact DFA which is "almost" equivalent to the original one in the sense that compaction cannot produce shift-reduce conflicts although new reduce-reduce conflicts are possible.

It can be shown that if the LALR compaction succeeds (i.e. does not lead to new reduce-reduce conflicts), the resulting parser accepts the same language as the original LR(1) parser.

The parser-generator yacc (and bison) generates an LALR parser based on an input grammar. In addition, yacc (and bison) provide facilities for resolving shift-reduce conflicts based on the declared precedence of operators.

Bison will resolve a shift reduce conflict on a state like

$$\begin{array}{lcll} E & \rightarrow & Eo_1E\bullet & ,o_2 \\ E & \rightarrow & E\bullet o_2E & \end{array}$$

by comparing the declared precedences of o_1 and o_2: if o_1 has the higher precedence, then reduce will be selected. If o_2 has a higher precedence than o_1 then shift will be selected. If o_1 and o_2 have the same precedence, associativity information on o_1 and o_2 will be used: if both symbols (operators) are left-associative, reduce will be selected; if both symbols are right-associative, shift will be selected; if both symbols are non-associative an error will be reported[6]. If a symbol (such as `MINUS` in Example 17 below) is used with two precedences, bison allows you to define the precedence of the rule using a `%prec` directive[7]

The default conflict resolution strategy of bison (and yacc) is as follows:

[6]Note that operators with the same precedence must have the same associativity.

[7]The above description is made more concrete in the bison manual:

The first effect of the precedence declarations is to assign precedence levels to the terminal symbols declared. The second effect is to assign precedence levels to certain rules: each rule gets its precedence from the last terminal symbol mentioned in the components (o_1 in the example). (You can also specify explicitly the precedence of a rule.)

- Choose shift when resolving a shift-reduce conflict.
- Choose reduce by the first production (in the order of declaration) when resolving a reduce-reduce conflict.

Example 17 The following is the bison input file for a highly ambiguous grammar, where ambiguities are resolved using the precedence/associativity defining facilities of bison/yacc.

```

%token NAME  NUMBER LPAREN RPAREN EQUAL PLUS MINUS
%token TIMES DIVIDE IF THEN ELSE

/*      associativity and precedence: in order of increasing precedence */

%nonassoc       LOW  /* dummy token to suggest shift on ELSE */
%nonassoc       ELSE /* higher than LOW */

%nonassoc       EQUAL
%left           PLUS    MINUS
%left           TIMES   DIVIDE
%left           UMINUS  /* dummy token to use as precedence marker */

%%

stmt            : IF exp THEN stmt              %prec LOW
                | IF exp THEN stmt ELSE stmt    /* shift will be selected */
                | /* other types of statements would come here */
                ;

exp             : exp PLUS exp
                | exp MINUS exp
                | exp TIMES exp
                | exp DIVIDE exp
                | exp EQUAL exp
                | LPAREN exp RPAREN
                | MINUS exp     %prec UMINUS /* this will force a reduce */
                | NAME
                | NUMBER
```

Finally, the resolution of conflicts works by comparing the precedence of the rule (of o_1 in the example, unless that rule's precedence has been explicitly defined) being considered (for reduce) with that of the look-ahead token (o_2 in the example). If the token's precedence is higher, the choice is to shift. If the rules precedence is higher, the choice is to reduce. If they have equal precedence, the choice is made based on the associativity of that precedence level. The verbose output file made by '-v' (see section Invoking Bison) says how each conflict was resolved.

Not all rules and not all tokens have precedence. If either the rule or the look-ahead token has no precedence, then the default is to shift.

```
                        ;
%%
```

Chapter 4

Checking static semantics

4.1 Attribute grammars and syntax-directed translation

Attribute grammars extend context-free grammars by associating a set of **attributes** with each grammar symbol. Such attributes can be used to present information that cannot easily (if at all) be expressed using only syntactical constructs (e.g. productions). Thus each node in a parse tree has values for each of the attributes associated with the corresponding symbol.

For example, we could associate an attribute *type* with the nonterminal *expression* to represent the type of the expression.

Attributes for nodes are computed using functions that take attribute values of other nodes as input. The computation is specified by *semantic actions* which are associated with each production. Since such actions can only reference the symbols in the production, the value of an attribute of a node depends only on the attribute values of its parent, children and siblings.

Example 18 Consider the grammar

$$G_4 = (\{E\}, \{+, \times, -, /, (,), \mathbf{number}\}, P, E)$$

where P consists of

$$E \rightarrow \mathbf{number} \mid (E) \mid E + E \mid E \times E \mid E/E \mid E - E \mid -E$$

We define an attribute *value* for E and $\mathbf{number}$. The semantic actions are as follow:

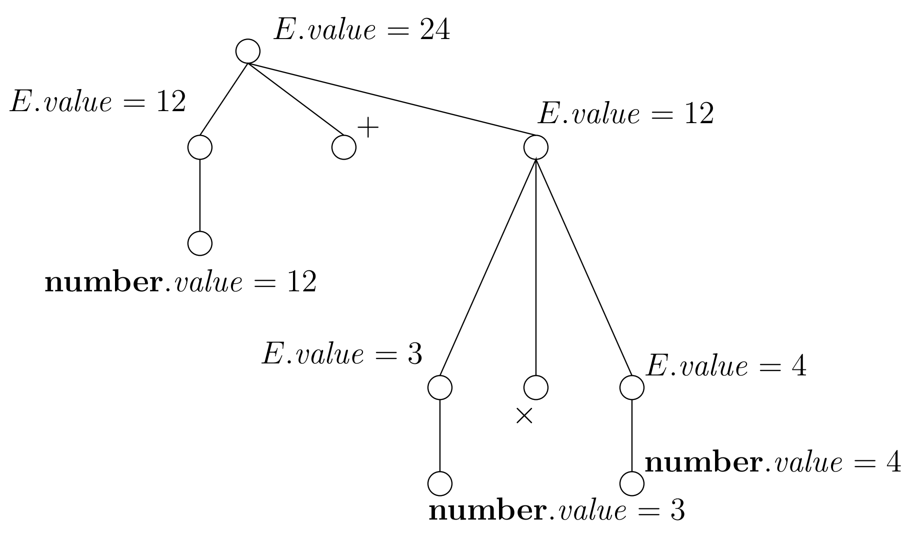

$E.value = 24$
$E.value = 12$
$+$
$E.value = 12$
$\mathbf{number}.value = 12$
$E.value = 3$
$E.value = 4$
$\times$
$\mathbf{number}.value = 4$
$\mathbf{number}.value = 3$

Figure 4.1: Computing synthesized attribute values

E	$\rightarrow$	$\mathbf{number}$	$E_0.value = \mathbf{number}_1.value$
E	$\rightarrow$	(E)	$E_0.value = E_1.value$
E	$\rightarrow$	$E + E$	$E_0.value = E_1.value + E_3.value$
E	$\rightarrow$	$E \times E$	$E_0.value = E_1.value \times E_3.value$
E	$\rightarrow$	$E - E$	$E_0.value = E_1.value - E_3.value$
E	$\rightarrow$	E/E	$E_0.value = E_1.value/E_3.value$
E	$\rightarrow$	$-E$	$E_0.value = -E_1.value$

Note that we use subscripts to refer to a particular occurrence of a symbol in the rule (subscript 0 is used for the left-hand side of the rule).

The attribute values for a parse tree of $12 + 3 * 4$ are shown in Figure 4.1. Note that the $\mathbf{number}.value$ is supposed to have been provided by the lexical analyzer.

The actions in Example 18 all compute an attribute value for a node based on the attribute values of its children. Attributes that are always computed in this way are called *synthesized attributes*. Attributes whose value depends on values of its parent and/or siblings are called *inherited attributes*.

The following example shows a grammar with both synthesized and inherited attributes.

Example 19 Consider the grammar

$$G_5 = (\{D, T, L\}, \{',', ;, \mathbf{id}, \mathbf{int}, \mathbf{double}\}, P, D)$$

Here D represents a C-style declaration consisting of a type followed by a comma-separated list of identifiers. The symbols T, L, and $\mathbf{id}$ have a *type* attribute which is inherited for L and $\mathbf{id}$.

The productions and actions of P are

$$\begin{array}{lcll}
D & \rightarrow & TL; & L_2.type = T_1.type \\
T & \rightarrow & \mathbf{int} & T_0.type = int \\
T & \rightarrow & \mathbf{double} & T_0.type = double \\
L & \rightarrow & L, \mathbf{id} & L_1.type = L_0.type \\
 & & & \mathbf{id}_3.type = L_0.type \\
L & \rightarrow & \mathbf{id} & \mathbf{id}_1.type = L_0.type
\end{array}$$

The attribute values for the parse tree of

int *x,y,z*;

are shown in Figure 4.2. The dependencies between values are shown by dashed arrows.

In general, one can define a **dependency graph** for a parse tree where the nodes are pairs (n, a) where n is a node in the parse tree and a is an attribute for (the symbol of) n. There is an edge from (n_1, a_1) to (n_2, a_2) if the value of $n_2.a_2$ depends on the value of (n_1, a_1), i.e. an action computing $n_2.a_2$ uses (n_1, a_1).

Attribute values can then be computed in any order that corresponds to a topological sort[1] of the nodes of the dependency graph. Clearly, for this to work, the dependency graph may not contain a cycle.

Thus, the syntax-directed translation $G(\alpha)$ of an input string α using an attribute grammar G can be computed by executing the following steps:

1. Parse α, according to the context-free grammar underlying G, yielding a parse tree t_α.

2. Construct the dependency graph g_{t_α}, and compute the attribute values of the nodes of t_α as described above.

[1]A topological sort of a directed (acyclic) graph is any total ordering $x_1 < x_2 < \ldots < x_n < \ldots$ of the nodes of the graph that satisfies $x_i < x_j$ whenever there is an edge from x_i to x_j.

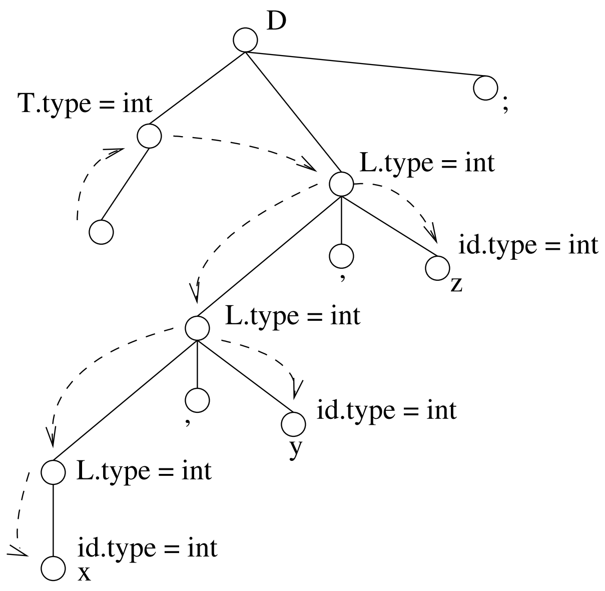


Figure 4.2: Computing inherited attribute values

3. $G(\alpha)$ is the value corresponding to some selected attribute of the root of t_α.

In practice, the full parse tree for an input program is often not explicitly constructed. Rather, attributes are computed and stored on a stack that mirrors the parsing stack (just like the stack containing the states of the viable prefix DFA) and actions are performed upon a reduce operations. This allows computation of synthesized attributes but inherited attributes are not easily supported although in certain cases it is possible to also evaluate inherited attributes (see [ASU86], pages 310-311).

In yacc and bison, each symbol (nonterminal or token) can have only one attribute, not larger than a single word. The type of an attribute is fixed per symbol using `%type` declarations, as can be seen in the example parser in Appendix C.6, page 175. The set of used types must also be declared in a `%union` declaration. The restriction to a single word is not limiting: usually the attribute is declared to be a pointer to a structure.

4.2 Symbol tables

The restriction of shift-reduce parsers to use only synthesized attributes can be overcome by using a global symbol table to share information between various parts of the parse tree. In general, a symbol table maps names (of variables, functions, types, ...) to information on this name which has been accumulated by the parser and semantic actions.

We discuss two aspects of symbol tables:

- The management of names (the domain of the mapping).
- The organization of the symbol table to take into account the scope rules of the language.

4.2.1 String pool

Since a symbol table maps names to information, one could implement a symbol table entry as a structure where the first field is the name and the other fields contain the information associated with the name

```
struct symtab_entry {
        char    name[MAX_NAME_LENGTH];
        INFO    info;
        };
```

However, because modern programming languages do not impose a small limit on the length of a name, it is wasteful or impossible to keep this organization. Hence one typically stores names contiguously in a so-called "string pool" array, declaring a symbol table entry as follows:

```
struct symtab_entry {
        char*   name;   /* points into string pool */
        INFO    info;
        };
```

For fast retrieval, entries may be organized into a hash table, as illustrated in Figure 4.3 where separate chaining is used to link entries that correspond to the same hash value.

4.2.2 Symbol tables and scope rules

Many programming languages impose *scope rules* that determine which definition of a name corresponds to a particular occurrence of that name in a program text. E.g. in Pascal, and in C, scopes may be nested to an arbitrary depth, as is illustrated in the following fragment.

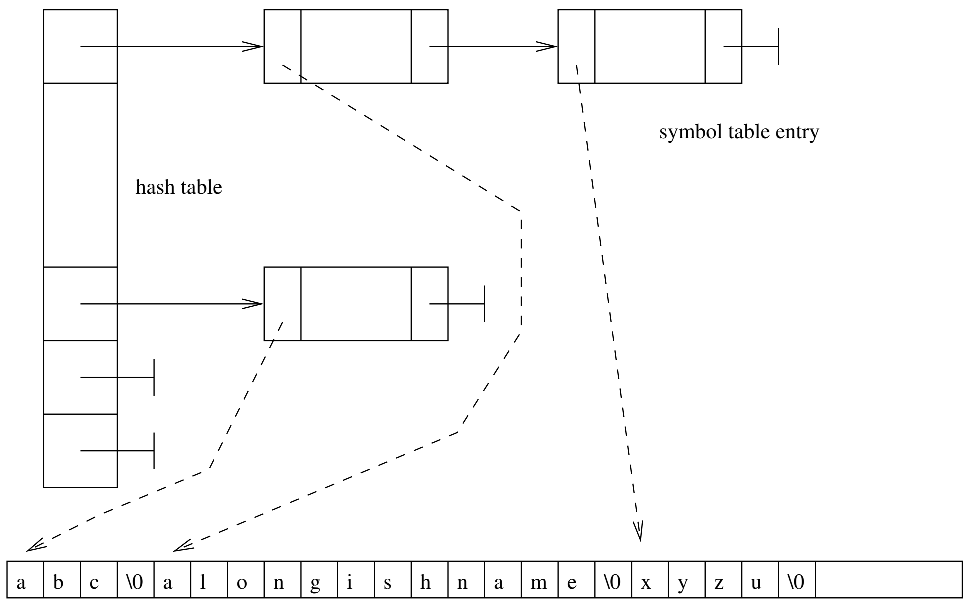


Figure 4.3: A simple symbol table using a string pool and hashing

```
float   x; /* def 1 */

int
f(int a)
{
while (a>0) {
        double  x = a*a,i; /* def 2 */

        for (i=0;(i<a);++i) {
                int x = a+i,b;  /* def 3 */
                b = x+2;        /* refers to def 3 */
                }

        }
}
```

In the above examples, the symbol table will contain three entries (definitions) for x at the time the inner loop is processed. Which definition is meant is determined using the "most closely nested scope" rule.

In a compiler for a language with scope rules, when a semantic action wants to retrieve the information associated with a certain occurrence of a name, it has to know which definition of the name corresponds to the given occurrence. If the language allows only nested scopes for which the "most closely nested scope" rule holds, this can be solved by replacing the symbol table by a stack of simple

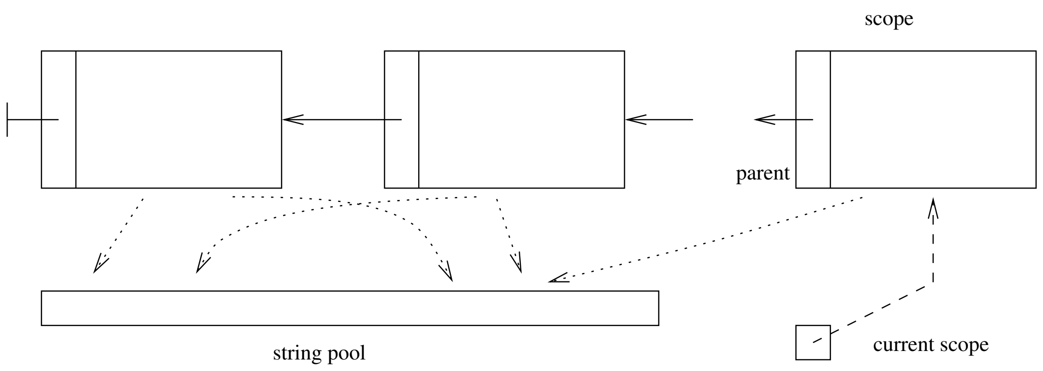


Figure 4.4: A stacked symbol table

symbol tables, one for each scope. Moreover, each simple table refers to its parent in the stack which contains symbol definitions from the enclosing scope. When looking up a name, it suffices to search the simple table at the top of the stack. If the name is not found, the parent table (corresponding to the enclosing scope) is searched and so on.

When the parser finishes a scope it pops the current scope from the stack and continues with the parent scope as the current one. Of course, if the scope is still needed, as is the case e.g. in C++ with scopes that correspond to class declarations, the scope will not be thrown away but a reference to it will be kept, e.g. in the symbol table entry corresponding to the class name.

This organization is depicted in Figure 4.4.

An simple implementation of this scheme, where hashing is replaced by linear search (first in the current scope, then in the parent scope), may be found in Appendix C.3, page 163.

4.3 Type checking

Informally, the so-called *static semantics* of a programming language describes the properties of a program that can be verified without actually executing it. Typically, for statically-types languages, this consists of verifying the "well-typed" property where a program is well-typed if all its constructs (statements, expressions, ...) obey the type rules of the language.

The *type system* of a programming language consists of

- A definition of the set of all possible types.

- Type rules that determine the type of a language construct.

The language from Appendix C, Minic, has two primitive types: integers and floats and a few type constructors, as shown by the following domain[2] definition[3]:

$$
\begin{array}{rcl}
\mathit{type} & :: & \mathbf{int} \\
 & :: & \mathbf{float} \\
 & :: & \mathbf{array}(\mathit{type}) \\
 & :: & \mathbf{struct}(\mathit{tenv}) \\
 & :: & \mathit{type}^* \rightarrow \mathit{type} \\
\mathit{tenv} & :: & \mathit{name} \rightarrow \mathit{type}
\end{array}
$$

Note that the above definition is more general than is actually needed in Minic, which has no higher order functions. E.g. it is not possible to define a function with type

$$\mathbf{int} \rightarrow (\mathbf{int} \rightarrow \mathbf{int})$$

in Minic.

The type rules of Minic can be symbolically represented as follows (we use $x : t$ to denote that x has type t). In the table below we use $+$ to use any of the arithmetic operators $+, \times, /, -$.

$$
\begin{array}{rcll}
t\ x; & \Rightarrow & x : t & \textit{// declaration of } x \\
x : \mathbf{int} \ \wedge \ y : \mathbf{int} & \Rightarrow & (x + y) : \mathbf{int} & \\
x : \mathbf{float} \ \wedge \ y : \mathbf{float} & \Rightarrow & (x + y) : \mathbf{float} & \\
x : t \ \wedge \ y : t & \Rightarrow & (x == y) : \mathbf{int} & \textit{// support = test on any type} \\
x : \mathbf{array}(t) \ \wedge \ i : \mathbf{int} & \Rightarrow & x[i] : t & \\
x : \mathbf{struct}(s) \ \wedge \ n \in \mathit{dom}(s) & \Rightarrow & x.n : s(n) & \\
x = e \ \wedge \ x : t \ \wedge \ e : t' \ \wedge \ t \neq t' & \Rightarrow & \mathbf{error} & \\
\forall 1 \leq i \leq n \cdot x_i : t_i \ \wedge \ f : t_1 t_2 \ldots t_n \rightarrow t & \Rightarrow & f(x_1, \ldots, x_n) : t & \\
\mathit{return}\, x \ \wedge \ x : t' \ \wedge \ f : t_1 t_2 \ldots t_n \rightarrow t \ \wedge \ t' \neq t & \Rightarrow & \mathbf{error} & \textit{// within the body of } f
\end{array}
$$

Here **error** is used to indicate that a certain construct is not well-typed, i.e. no type can be assigned to it without violating the type rules.

[2]See e.g. the "Foundations of Computer Science II" course.

[3]Note that there is no boolean type: like C, Minic uses integers instead of booleans.

Type checking then consists of using the rules to assign a type to each expression, variable etc. If no type, or more than one type, can be assigned to a construct, or if the rules generate **error**, the program is not well-typed.

It should be obvious that with the help of attributes, semantic actions and a symbol table, such type checking is possible. An example of a type-checking parser for Minic can be found in Appendix C. In the example implementation, types are represented by structures where each structure contains a tag corresponding to the type constructor. Furthermore, types are shared, which saves space and facilitates later checking for equality (only pointers must be compared).

Note that there are statically typed languages (e.g. ML or Haskell) that do not require, as Minic does, that each variable is declared to have a specific type. Instead, the type of a variable is inferred from its use. In addition, such languages usually have a more sophisticated type system where types can have variables that may be instantiated. In this way, a language can allow for polymorphism where a function has several (possibly an infinite number of) types.

A similar feature is supported by C++ through the template facility. E.g. a function

```
template <class T>
int     length(list<T>) {
// ...
}
```

has a type $\mathbf{list}(\alpha) \rightarrow int$ where α is a type variable. The compiler will instantiate α to **int** when processing the expression `length(l)` in

```
list<int>       l;

length(l);
```

Chapter 5

Intermediate code generation

While it is possible to construct a compiler that translates the source text directly into the target language, especially if sophisticated optimization is not required, there are important advantages of using an intermediate representation:

- It facilitates retargeting. A compiler employing an intermediate representation can be viewed as consisting of a “front end” (everything up to intermediate code generation[1]) and a “back end” (optimization, code generation). Note that all dependencies on the source language are located in the front end while dependencies on the target language are confined to the back end. Thus, to retarget the compiler (i.e. to generate code for another processor), it suffices to adapt the back end only. Similarly, if the intermediate code is sufficiently general, the back end could be reused in a compiler for another language.
- Many machine-independent optimizations, especially those that require code movement etc., can be done more easily on intermediate code.

Several forms of intermediate code have been proposed (and used):

- Postfix notation.
- Abstract syntax trees.
- Three-address code.

[1]Any machine-independent optimization that can be performed on the intermediate code should also be done in the front end.

5.1 Postfix notation

Postfix notation, also called "postfix Polish" after the originator J. Lukasiewicz, represents expressions and flow of control statements in postfix form, which is free of parentheses, as in the following example:

infix	postfix
$a + b$	$ab+$
$(a + b) \times (c - d)$	$ab + cd - \times$

In general, an expression consisting of a k-ary operator θ, applied on k (sub)expressions $e_1, \ldots, e_k$ is represented as

$$e'_1 e'_2 \ldots e'_k \theta$$

where e'_i, $1 \leq i \leq k$, is the postfix form of e_i.

Evaluation of postfix expressions is trivial, provided one uses an operand stack, as shown below.

```
operand
eval_operator(OPERATOR o,item args[]);

operand
eval_postfix(item code[],int code_len)
{
operand stack[MAX_STACK];
int     tos     = 0; // top of stack

for (int i=0;(i<code_len);++i)
        if (is_operand(code[i]))
                stack[tos++] = code[i]; // push on stack
        else
                { // operator: pop args, eval, push result
                k = arity(code[i]);
                stack[tos-k] = eval_operator(code[i].oprtr, stack-k);
                tos -= (k-1);
                }
return stack[0];
}
```

By introducing labels which refer to a position in the postfix expression, arbitrary control structures can be implemented using instructions like

$$\mathit{label\ jump}$$
$$e_1 e_2\ \mathit{label\ jumplt}$$

In order to interpret labels and jump instructions, the above algorithm must of course be extended to use a program counter (an index in the `code` array),

Postfix notation is attractive because it can be easily generated using syntax-directed translation. Below, we use an synthesized attribute *code* representing the postfix notation of the corresponding symbol occurrence. The statements associated with a rule define the attribute of the left hand side of the rule (denoted $$), based on the attribute values of the symbols in its right hand side ($$i$ refers to the i'th symbol in the rules right hand side).

```
exp     : exp binop exp {
                $$.code = concat($1.code, $3.code, operator($2));
                }
exp     : ( exp) {
                $$.code = $1.code;
                }
exp     : var { $$.code = $1; /* probably a symtab ref */ }
```

This scheme has the so-called *simple postfix* property which means that for a production

$$A \rightarrow A_1 A_2 \ldots A_n$$

the generated code is such that[2]

$$\mathit{code}(A) = \mathit{code}(A_1) + \ldots + \mathit{code}(A_n) + \mathit{tail}$$

which implies that code can be emitted as soon as a reduction occurs.

Postfix code is often used for stack-oriented virtual machines without general purpose registers, such as the Java virtual machine (see e.g. the micro compiler in Appendix B. It is not particularly suitable for optimization (code movement is hard).

5.2 Abstract syntax trees

Abstract syntax trees are "pruned" parse trees where all nodes that contain redundant (e.g. with respect to the tree structure) information have been removed. Indeed, many nodes in the parse tree result from syntactic sugar in the grammar which is needed to ensure that the language can be efficiently parsed. Once we have parsed the text, there is no need to keep these superfluous nodes[3]. E.g. the abstract syntax tree corresponding to the parse tree of Figure 1.6 on page 14 is shown in Figure 5.1.

[2]We use $+$ to denote concatenation.

[3]It should be noted that a formal definition of a language's semantics is often based on a so-called *abstract syntax* which only specifies the structure of the language's constructs, rather than

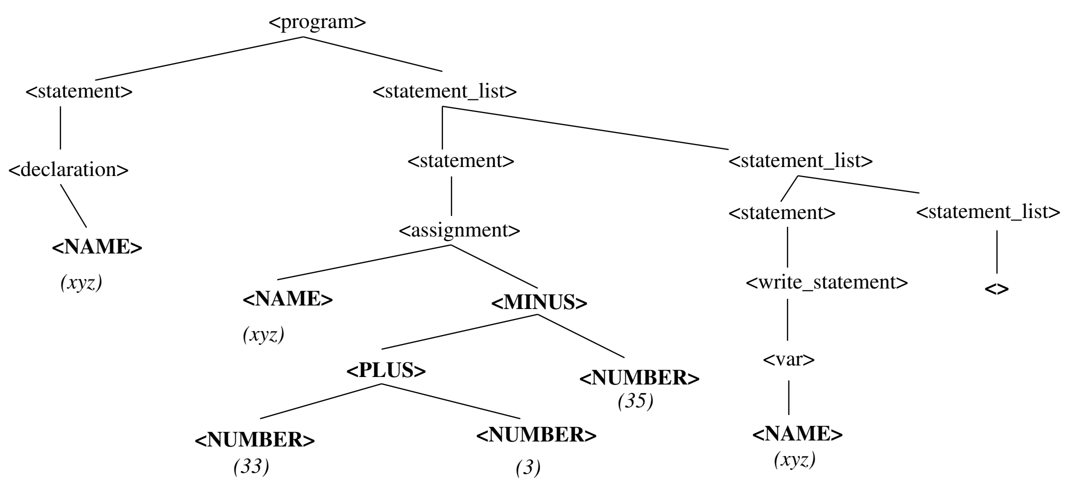


Figure 5.1: Abstract syntax tree corresponding to the program in Figure 1.3

Abstract syntax trees (and directed acyclic graphs), particularly of expressions, are useful for certain optimizations such as common subexpression elimination and optimal register allocation (for expression computation).

An abstract syntax tree can be generated using synthesized attributes.

```
exp      : exp binop exp {
                $$.tree = new_node_2($2,$1.tree,$3,tree);
                }
%*
exp      : ( exp ) {
                $.tree = $2.tree;
                }
%*
exp      : unop exp {
                $$.tree = new_node_1($1,$2.tree);
                }
%*
exp      : var {
                $$.tree = new_node_0($1);
                }
```

Abstract syntax trees can also be constructed for complete source texts. They then

the concrete syntax. E.g. in the abstract syntax specification, one would have a rule

$$assignment \rightarrow var\ expression$$

rather than

$$assignment \rightarrow var\ =\ expression$$

form the input for code generators that are based on pattern matching (in trees) and tree transformations. This strategy is employed by so-called *code-generator generators* (see e.g. Section 9.12 in [ASU86]) which take as input *tree-rewriting rules* of the form

$$\textit{replacement} \leftarrow \textit{template} \; \{\, \text{action} \,\}$$

where the template specifies a tree pattern that must be matched, the replacement is a tree (usually a single node) that is to replace the matched template and the action is a code fragment, as in syntax-directed translation. The code fragment may check additional constraints and generates target instructions. Figure 5.2 illustrates such tree-rewriting rules. The idea is that to retarget the compiler to

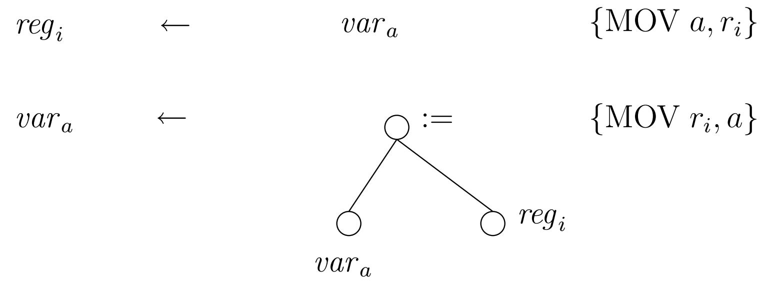


Figure 5.2: Tree-rewriting rules

a processor with different instruction sets (and costs), one simply replaces the tree-rewriting rules.

5.3 Three-address code

Three address code is a convenient and uniform intermediate representation. It consists of a number of simple instructions which involve at most three addresses, hence the name. The addresses refer to user-defined names (variables, functions, ...) or constants, or to temporary variables generated by the compiler.

In this and the following chapter, we will use three-address code as an intermediate representation. The instructions are listed in Figure 5.3.

An important advantage of three-address code is that it can be easily reordered (at least within a function body) for optimization. A convenient representation for three-address code instructions is as a quadruple.

```
typedef enum {A2PLUS, A2TIMES, A2MINUS, A2DIV, A1MINUS,
                 A1FTOI, A1ITOF, A0, GOTO, IFEQ, ..,  PARAM,
```

opcode	instruction	meaning
A2(binop)	A = B binop C	assignment, binop can be $+$, $\times$, ...
A1(unop)	A = unop B	assignment, unop can be $-$, **float2int**, ...
A0	A = B	simple assignment
GOTO	**goto** label	
IF(relop)	**if** A relop B **goto** label	relop can be $<$, $>$, $\leq$, $==$, ...
PARAM	**param** A	push parameter before function call
CALL	**call** F, n	function call, n is number of parameters
AAC	A = B[I]	array access, B contains base address, I is offset in memory units
AAS	A[I] = B	array modification, A contains base address
ADDR	A = **address** B	take address of B
DEREF	A = * B	as in C
DEREFA	*A = B	as in C
RETURN	RETURN A	as in C

Figure 5.3: Three-address code instructions

```
                CALL, AAC, AAS, ADDR, DEREF, DEREFA } OPCODE;
typedef struct {
        OPCODE          op_code;
        SYM_INFO*       args[2];
        SYM_INFO*       result;
        } INSTRUCTION;
```

5.4 Translating assignment statements

The actions below use a simplified grammar for expressions. Also, we assume that all expressions are of type integer.

We use two synthesized attributes:

- *exp.place* represents a reference to the location (variable, symbol table reference) containing the value of the expression *exp*.
- *exp.code* represents the sequence of three-address statements computing the value of *exp*. Code concatenation is written as "$+$".

```
extern INSTRUCTION /* create quadruple representing instruction */
gen3ai(short op_code,SYM_INFO* arg1, SYM_INFO* arg2, SYM_INFO* result);

assign  : var = exp {
                $$.code = $3.code + gen3ai(A0,$3.place,0,$1.place);
                }
exp     : exp + exp {
                $$.place = newvar();
                $$.code = $1.code + $3.code
                                + gen3ai(A2PLUS,$1.place,$3.place,$$.place);
                }
exp     : - exp {
                $$.place = newvar();
                $$.code = $2.code
                                + gen3ai(A1MINUS,$2.place,0,$$.place);
                }
exp     : ( exp ) {
                $$.place = $2.place;
                $$.code = $2.code;
                }
exp     : var {
                $$.place = $1.place;
                $$.code = 0;
                }
```

Note that the actions for the *code* attribute have the "simple postfix" property. Hence we can simplify the actions.

```
extern void
emit(INSTRUCTION i);

assign  : var = exp {
                emit(gen3ai(A0,$3.place,0,$1.place));
                }
exp     : exp + exp {
                $$.place = newvar();
                emit(gen3ai(A2PLUS,$1.place,$3.place,$$.place));
                }
exp     : - exp {
                $$.place = newvar();
                emit(gen3ai(A1MINUS,$2.place,0,$$.place));
                }
exp     : ( exp ) {
                $$.place = $2.place;
                }
exp     : var {
                $$.place = $1.place;
                }
```

As an example of the integration of static (notably type) checking with intermediate code generation, we show below how appropriate conversions can be generated for languages that support them. Note that we use an extra attribute *type* representing the type of an expression, which can be either `t_int` (integer) or `t_float` (floating point). We also assume that there are two instructions for addition: `A2PLUSI` and `A2PLUSF` for integer and floating point addition, respectively.

```
exp     : exp + exp {
                $$.place = newvar();
                if (($1.type==t_int)&&($3.type==t_int)) {
                        $$.type = t_int; symtab_set_type($$.place,t_int);
                        emit(gen3ai(A2PLUSI,$1.place,$3.place,$$.place));
                        }
                else if (($1.type==t_float)&&($3.type==t_float)) {
                        $$.type = t_float; symtab_set_type($$.place,t_float);
                        emit(gen3ai(A2PLUSF,$1.place,$3.place,$$.place));
                        }
                else if (($1.type==t_int)&&($3.type==t_float)) {
                        SYM_INFO* tmpf = newvar(t_float);
                        emit(gen3ai(A1ITOF,$1.place,0,tmpf));
                        $$.type = t_float; symtab_set_type($$.place,t_float);
                        emit(gen3ai(A2PLUSF,$1.place,tmpf,$$.place));
                        }
                else if (($1.type==t_float)&&($3.type==t_int)) {
                        SYM_INFO* tmpf = newvar(t_float);
                        emit(gen3ai(A1ITOF,$3.place,0,tmpf));
                        $$.type = t_float; symtab_set_type($$.place,t_float);
                        emit(gen3ai(A2PLUSF,tmpf,$3.place,$$.place));
                        }
                }
```

5.5 Translating boolean expressions

First we consider the translation of boolean expressions (conditions). An obvious possibility is to translate such expressions *by value*, i.e. by creating, if necessary, a temporary variable to hold its value. In this case, we agree to use 0 to represent “false” and 1 to represent “true”.

```
extern int
next3ai(); /* returns number of next (free) location in code sequence */

bexp    : bexp OR bexp {
                $$.place = newvar(t_bool);
                emit(gen3ai(A2OR,$1.place,$3.place,$$.place));
                }

bexp    : bexp > bexp {
                $$.place = newvar(t_bool);
                emit(gen3ai(IFGT,$1.place,$3.place,next3ai()+3));
                emit(gen3ai(A0C,0,0,$$.place));
```

```
                    emit(gen3ai(GO,next3ai()+2,0,0));
                    emit(gen3ai(A0C,1,0,$$.place));
                    }
```

Note the use of the function `next3ai()` which is used to generate forward jump instructions.

As an example, one can verify that the C expression

```
        a<b || c
```

will be translated to

```
1  if a < b goto 4
2  t1 = 0
3  goto 5
4  t1 = 1
5  t2 = t1 || c
```

Translation by value is appropriate e.g. for code computing the value of a boolean variable. However, for conditions in while- or if-statements, translating by value may be wasteful. Indeed, suppose that $a < b$ in the following statement.

```
if ((a<b)||(c<d))
        x = y + z;
```

When translating by value, the value of $c < d$ will be computed, even though it is already known (because $a < b$) that the then-branch will be taken. A better translation scheme would branch to the code for `x=y+z` as soon as possible, saving on superfluous computations[4].

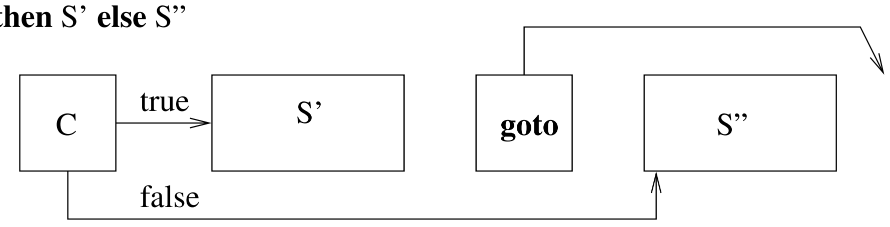
The alternative translation strategy for boolean expressions is *by flow of control*. This scheme is used for conditions in the context of control-flow statements. Intuitively the value of a condition is not represented directly as (the value of) a variable but rather indirectly as the location to which control transfers after evaluating (part of) the condition. This is illustrated in Figure 5.4.

The example statement above should then be translated as

```
1  if a < b goto 4
2  if c < d goto 4
3  goto 6
4  t1 = y + z          /* true exit */
5  x = t1
6                      /* false exit */
```

[4]Note that it depends on the semantics of the source language whether such short-circuit translation of conditions is allowed. E.g. in C, translating `if ((a<b)||((u=z)<d)) x= y+z;` by value will always execute `u=z` while translating “by flow of control” may not.

if C **then** S’ **else** S”



while C S

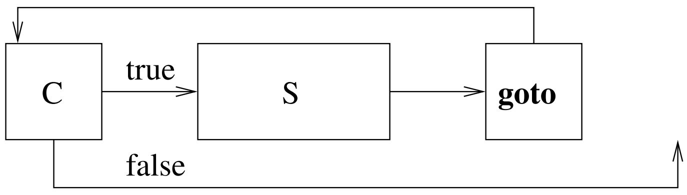


Figure 5.4: Conditional control flow statements

Thus a condition is associated with two locations in the code: a true and false exit. The generated code is such that control will transfer to the true or false exit as soon as possible. However, as can be seen in Figure 5.4, the exits are usually unknown when the condition is parsed, since they refer to locations further in the code array. The solution is to generate a dummy “**goto** ?” instruction wherever a jump to an exit should come. Here “?” is a placeholder for the, as yet unknown, location of the exit. The locations of the dummy jump instructions are kept in two synthesized attributes associated with the condition:

- *bexp.true* contains the set of all “**goto** ?” instructions generated so far where “?” should eventually be replaced by the address of the true exit.
- *bexp.false* contains the set of all “**goto** ?” instructions generated so far where “?” should eventually be replaced by the address of the false exit.

When the exact location *l* of e.g. the true exit becomes known, one can *backpatch* the instructions in *bexp.true*, replacing “?” by *l* in each of them.

The semantics of boolean operations such as disjunction, conjunction and negation can easily be expressed by manipulation of the *true* and *false* sets, as illustrated in Figure 5.5.

The actions are shown below.

```

cond    : cond AND marker cond {
                  $$.true = $4.true;
```

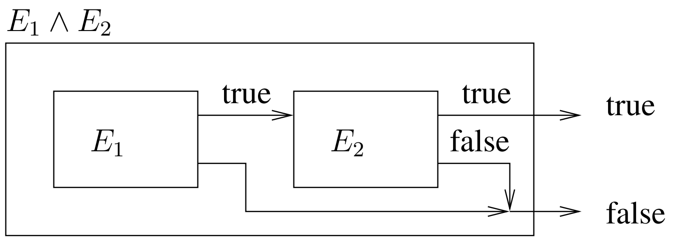


Figure 5.5: Exit sets after “and”

```
                backpatch($1.true,$3.location);
                $$.false = locations_union($1.false,$4.false);
                }
marker  : /* empty right hand side */ {
                $$.location = next3ai();
                }
cond    : NOT cond {
                $$.true = $2.false;
                $$.false = $2.true;
                }
cond    : var {
                $$.true = locations_add(locations_create_set(),next3ai());
                emit(gen3ai(IFNEQ,$1.place,0,0)); /* true, backpatch later */
                $$.false = locations_add(locations_create_set(),next3ai());
                emit(gen3ai(GOTO,0,0,0)); /* false,backpatch later */
                }
cond    : exp < exp {
                $$.true = locations_add(locations_create_set(),next3ai());
                emit(gen3ai(IFLT,$1.place,$3.place,0)); /* true, backpatch later *
                $$.false = locations_add(locations_create_set(),next3ai());
                emit(gen3ai(GOTO,0,0,0)); /* false,backpatch later */
                }
```

Note the dummy $marker \rightarrow \epsilon$ production: since it is reduced just before code for the second *bexp* will be generated, it can be used to store the location of the start of this code in an attribute *marker.location*.

The following example illustrates the translation of

```
        p<q || (r<s && t<u)
```

The generated intermediate code is shown below:

100	**if** p < q **goto** ?t	/* **true** exit */
101	**goto** 102	
102	**if** r < s **goto** 104	
103	**goto** ?f	/* **false** exit */
104	**if** y < u **goto** ?t	/* **true** exit */
105	**goto** ?f	/* **false** exit */

5.6 Translating control flow statements

We will use the following example grammar:

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{if}\ \textit{cond stmt}\ \textbf{else}\ \textit{stmt} \\
 & | & \textbf{while}\ \textit{cond stmt} \\
 & | & \{\ \textit{slist}\ \} \\
 & | & \textit{assign} \\
\textit{slist} & \rightarrow & \textit{slist}\ ;\ \textit{stmt} \\
 & | & \textit{stmt}
\end{array}
$$

Observe that the code corresponding to a *stmt*, e.g. a **while** statement, may contain forward references to a "next" location (meaning, after the code of the present statement) which is unknown at the time code is generated for the current statement. Therefore we use an attribute *stmt.next* which contains the locations containing instructions that need to be backpatched with the location of the first statement *after* the current one. We will also use dummy nonterminals to mark certain locations in the code. The actions for the above grammar are shown below.

```
marker  : /* empty */ {
                $$.location = next3ai();
                }

goend   : /* empty */ {
                $$.next = locations_add(locations_create_set(),next3ai());
                emit(gen3ai(GOTO,0,0,0)); /* goto end,backpatch later */
                }

stmt    : IF cond marker stmt goend ELSE marker stmt  {
                backpatch($2.true,$3.location);
                backpatch($2.false,$7.location);
                $$.next = locations_union(
                        locations_union($4.next,$5.next),
                        $8.next);
                }

```

```
stmt    : WHILE marker cond marker stmt {
                backpatch($3.true,$4.location);
                backpatch($5.next,$2.location);
                $$.next = $3.false;
                emit(gen3ai(GOTO,$2.location,0,0));
                }

stmt    : assign {
                $$.next = locations_create_set();
                }

stmt    : LBRACE slist RBRACE {
                $$.next = $2.next;
                }

slist   : stmt {
                $$.next = $1.next;
                }

slist   : slist SEMICOLON marker stmt {
                backpatch($1.next,$3.location);
                $$.next = $4.next;
                }
```

The following example illustrates the translation of

```
        while (a<b)
                if (c<d)
                        x = y + z;
```

The generated intermediate code is shown below:

```
100   if a < b goto 102
101   goto 107
102   if c < d goto 104
103   goto 100             /* false exit */
104   t = y + z
105   x = t
106   goto 100
```

5.7 Translating procedure calls

We will use the following example grammar:

$$\begin{array}{lcl} \textit{stmt} & \rightarrow & \textbf{id}(\ \textit{arg\_list}\ ) \\ \textit{arg\_list} & \rightarrow & \textit{arg\_list}\ ,\ \textit{exp} \\ & | & \textit{exp} \end{array}$$

Note that, as in C, we assume that all parameters are passed "by value". If, as e.g. in Pascal or C++, the source language also supports passing parameters "by address", extra instructions that take the address of an operand, may need to be inserted before the `PARAM` instruction.

A statement such as

```
f(e1,e2)
```

will be translated as

```
 ..   ..          /* code for e1, place t1 */
 ..   ..          /* code for e2, place t2 */
102   param t1
103   param t2
104   call f
```

The actions are:

```
typedef struct place_list {
        ...
        }       PLACE_LIST;

typedef struct {
        ..
        }       PLACE_LIST_CURSOR;

/*
*       list of places, return new list after successful operation
*/
PLACE_LIST*             place_list_create();
void                    place_list_destroy(PLACE_LIST*);
PLACE_LIST*             place_list_append(PLACE_LIST*,SYM_INFO*);

/*
*       cursor into list of places, return null pointer upon failure.
*/
PLACE_LIST_CURSOR*      place_list_first(PLACE_LIST*);
void                    place_list_cursor_destroy(PLACE_LIST_CURSOR*);
PLACE_LIST_CURSOR*      place_list_next(PLACE_LIST_CURSOR*);
SYM_INFO*               place_list_current(PLACE_LIST_CURSOR*);

stmt    : ID ( arg_list) {
```

```
                PLACE_LIST_CURSOR*      c;
                int                     n = 0; /* number of arguments */

                while (c = place_list.first($3.places)) {
                        ++n;
                        emit(gen3ai(PARAM,place_list_current(c),0,0));
                        c = place_list_next(c);
                        }
                emit(gen3ai(CALL,ID.place,n,0);
                /* clean up */
                place_list_cursor_destroy(c);
                place_list_destroy($3.places);
                }

arg_list: arg_list , exp {
                $$.places = place_list_append($1.places,$3.place);
                }
arg_list: exp {
                $$.places = place_list_append(place_list_create(),$1.place);
                }
```

5.8 Translating array references

Consider the declaration

```
int     a[10,20];
int     i,j;

..
.. a[i,j]
```

We will show how to generate intermediate code for array references such as $a[i, j]$.

We assume that the size of an integer is 4 bytes, that a is the address of the array and that the array is stored in row-major order.

The address of $a[i, j]$ is then

$$a + (20 \times (i - 1) + (j - 1)) \times 4$$

where $20 \times (i - 1)$ is the number of integers in rows appearing before the i'th row and $(j - 1)$ is the number of integers on the same row but before $a[i, j]$.

The address calculation can be rewritten as

$$a - 21 \times 4 + (20 \times i + j) \times 4$$

Note that $a - 21 \times 4$ is a fixed "base" quantity that only depends on the (declaration of) the array. The remainder $(20 \times i + j) \times 4$ is called the "offset".

The corresponding intermediate code is shown below.

```
 ..   ..            /* compute i, place t1 */
 ..   t2 = t1 * 20  /* 20 × i */
 ..   ..            /* compute j, place is t3 */
102   t4 = t2 + t3  /* (20 × i + j) */
103   t5 = addr a
104   t5 = t5 - 84  /* (a − 21 × 4) */
105   t = t5[t4]    /* a[i, j] */
```

In general, we consider an array

$$type \quad a[d_1, \ldots, d_k];$$

where $s = \mathit{sizeof}(\mathit{type})$, d_i is the number of elements corresponding to the i'th dimension ($1 \leq i \leq k$), and a is the address of a.

The address of an element $a[i_1, \ldots, i_k]$ is

$$\begin{array}{ll} \multicolumn{2}{l}{address(a[i_1, \ldots, i_k]) =} \\ & a + [(i_1 - 1) \times d_2 \times \ldots \times d_k + (i_2 - 1) \times d_3 \times \ldots \times d_k + \ldots \\ & +(i_{k-1} - 1) \times d_k + (i_k - 1)] \times s \\ = & a + [(i_1 \times d_2 \times \ldots \times d_k + i_2 \times d_3 \times \ldots \times d_k + \ldots + i_{k-1} \times d_k + i_k \\ & -(d_2 \times \ldots \times d_k + d_3 \times \ldots \times d_k + \ldots + d_k + 1)] \times s \\ = & a - C + [(\ldots(((i_1 \times d_2) + i_2) \times d_3 + i_3)\ldots) \times d_k + i_k] \times s \end{array}$$

where

$$C = (d_2 \times \ldots \times d_k + d_3 \times \ldots \times d_k + \ldots + d_k + 1) \times s$$

depends only on (the declaration of) the array.

We will use the following example grammar (an *lvalue* is an expression that can be assigned to):

$$\begin{array}{lcl} \textit{exp} & \rightarrow & \textit{lvalue} \\ \textit{lvalue} & \rightarrow & \text{ID /* simple variable */} \\ & | & \textit{elist}\ \text{] /* array reference */} \\ \textit{elist} & \rightarrow & \textit{elist}\ ,\ \textit{exp} \\ & | & \text{ID [} \textit{exp} \end{array}$$

For *lvalue*, we use two attributes: *lvalue.place* and *lvalue.offset* representing the base address and the offset, respectively (the offset is only meaningful for array references).

For *elist*, we have the attributes *elist.place*, containing the offset computed so far, *elist.array* containing a reference to the symbol table entry for the array variable and *elist.ndim* representing the number of dimensions seen so far.

The offset will be computed by successively computing (in *elist.place*)

$$\begin{aligned}
o_1 &= i_1 \\
o_2 &= o_1 \times d_2 + i_2 \\
o_3 &= o_2 \times d_3 + i_3 \\
.. &\;..\; .. \\
o_k &= o_{k-1} \times d_k + i_k \\
\mathit{offset} &= o_k \times s
\end{aligned}$$

The actions are shown below.

```
#define NIL     0x0FFFFFFF

int      array_limit(SYM_INFO*,int); /* return size of dimension i of array */
int      array_base(SYM_INFO*); /* return constant C associated with array */
int      array_el_size(SYM_INFO*); /* return size of element of array */

elist    : ID [ exp {
                $$.place = exp.place; /* first index */
                $$.ndim = 1;
                $$.array = $1.place;
                }
elist    : elist , exp {
                int limit  = array_limit($1.array,$1.ndim+1);
                $$.place = newvar();
                /* offset(next) = offset(prev)*limit(prev)+index(next) */
                emit(gen3ai(A2TIMES,$1.place,limit,$$.place);
                emit(gen3ai(A2PLUS,$$.place,$3.place,$$.place);
                $$.array = $1.array;
                $$.ndim = $1.ndim+1;
                }
lvalue   : elist ] {
                $$.place = newvar();
                $$.offset = newvar();
                /* base = addr a - array_base(a) */
                emit(gen3ai(ADDR,$1.array,0,$$.place);
                emit(gen3ai(A2MINUS,$$.place,array_base($1.array),$$.place);
                /* offset = elist.offset * sizeof(element) */
                emit(gen3ai(A2TIMES,$1.place,array_el_size($1.array),$$.offset);
                /* now $$.place[$$.offset] references array element */
                }
lvalue  : ID {
                $$.place = $1.place;
                $$.offset = NIL;
                }
exp     : lvalue {
                if ($1.offset==NIL) { /* note: NIL!=0 */
                        $$.place = $1.place
                        }
                else
                        {
                        $$.place = newvar();
                        emit(gen3ai(AAC,$1.place,$1.offset,$$.place));
                        }
                }
```

Chapter 6

Optimization of intermediate code

6.1 Introduction

In this chapter we look at techniques for improving intermediate (machine-independent) code with respect to two criteria: *time*, i.e. execution time, and *space*, i.e. the space taken up by the intermediate code instructions and variables. Note that optimization is actually a misnomer since there is no guarantee that the resulting code is the best possible.

Usually, improvements in time will also result in more compact code since instructions and variables will be eliminated. This is due to the "technical" nature of the improvements. On a more abstract, e.g. the algorithm, level, the opposite is usually the case in that one can obtain faster algorithms at the cost of more space for data structures. As a simple example, consider the problem of designing a function that takes any ASCII character as input and converts upper case to lower case letters.

A naive "small" algorithm will look something like

```
char
tolower(char c) /* assume 0<=c<=127 */
{
if ((c>='A') && (c<='Z'))
        return c-'A'+'a';
else
        return c;
}
```

By storing the function as a table, hence taking up more space, one can obtain a faster algorithm as shown below.

```
static char TOLOWER[] = {
        '\0',   /* 0 */
        ...
        'a',    /* 65 ('A') */
        'b',    /* 66 ('B') */
        ..
        '~',    /* 126 ('~') */
        '\0177' /* 127 */
        };

char
tolower(char c) /* assume 0 <= c <= 127 */
{
return TOLOWER[c];
}
```

In general, it is worth noting that, if speed is an issue, optimization by the compiler can make a fast algorithm run faster but it cannot make a slow algorithm run fast enough.

In the example below, the running time of a bad algorithm (bubblesort, B) is compared with its optimized version B^* and with a good algorithm (quicksort, Q). The number n is a measure for the size of the problem (i.e. the number of records to be sorted).

	$n = 10^4$	$n = 10^5$
B	1	100
B^*	.1	10
Q	.1	1.5

As the table shows, while compiler optimization can result in a tenfold speedup, as the size of the problem grows, an unoptimized superior algorithm easily catches up with the optimized version of a slow algorithm. Hence, if speed is an issue, there is no excuse for not designing an efficient algorithm before relying on the compiler for further optimization.

Finally, one should be aware that machine-independent code improvement techniques, such as the ones discussed in this chapter, are only one source of optimization that can be performed by the compiler. Indeed, during the code generation phase, there are additional opportunities for the compiler to dramatically improve program running times, e.g. by judicious allocation of variables to registers.

6.2 Local optimization of basic blocks

A sequence of intermediate code instructions is converted to a *flow graph* where the nodes consist of *basic blocks* as follows.

Definition 16 *Let s be a sequence of intermediate code instructions. A* ***leader*** *instruction is any instruction $s[i]$ such that*

- *$i = 0$, i.e. $s[i]$ is the first instruction; or*
- *$s[i]$ is the target of a (conditional or unconditional) jump instruction; or*
- *$s[i-1]$ is a (conditional or unconditional) jump instruction*

A ***basic block*** *is a maximal subsequence $s[i], \ldots, s[i+k]$ such that $s[i]$ is a leader and none of the instructions $s[i+j]$, $i < j \leq k$ is a leader.*

Thus, a basic block is a sequence of consecutive instructions in which flow of control enters at the beginning and leaves at the end.

Definition 17 *Let s be a sequence of intermediate code instructions. The* ***flow graph*** *of s is a directed graph where the nodes are the basic blocks in s and there is an edge from B_1 to B_2 iff*

- *The last instruction of B_1 is a (conditional or unconditional) jump instruction to the first instruction of B_2; or*
- *B_2 follows B_1 in s and the last instruction of B_1 is not an unconditional jump instruction.*

As an example, consider the following fragment which computes the inner product of two vectors.

```
{
product = 0;
for (i=0;i<20;i=i+1)
        product = product + a[i]*b[i];
}
```

The intermediate code is shown below. It contains two basic blocks, B_1 and B_2, comprising the instructions 1-2 and 3-16, respectively. The flow graph is shown in Figure 6.1

```
1  product = 0
2  i = 1
3  t1 = 4*i
4  t2 = addr a
5  t3 = t2 -4
6  t4 = t3[t1]
7  t5 = addr b
8  t6 = t5 - 4
9  t7 = 4*i
10 t8 = t6[t7]
11 t9 = t4 * t8
12 t10 = product + t9
13 product = t10
14 t11 = i+1
15 i = t11
16 if (i<20) goto 3
```

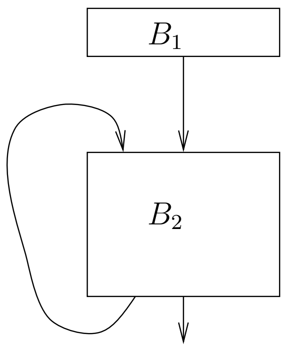


Figure 6.1: Flow graph of example code

6.2.1 DAG representation of basic blocks

A basic block can be represented as a *directed acyclic graph* (dag) which abstracts away from the non-essential ordering of instructions in the block. On the other hand, the dag will be constructed such that expressions are evaluated only once, since they correspond to single nodes in the graph.

Pseudo-code for the algorithm to construct the dag for a basic block is shown below.

Algorithm 8 [Constructing a DAG from a basic block]

```
/*
*  node(SYM_INFO*) and set_node(SYM_INFO*,NODE*) maintain a mapping:
*
*       node:   SYM_INFO* -> NODE*
*
*  which associates a (at most 1) node with a symbol (the node containing
*  the "current value" of the symbol).
*/
extern NODE* node(SYM_INFO*); /* get node associated with symbol */
extern void set_node(SYM_INFO*,NODE*); /* (re)set node associated with symbol */

/*
*  leaf(SYM_INFO*) and set_leaf(SYM_INFO*,NODE*) maintain a mapping:
*
*       leaf: SYM_INFO* -> LEAF*
*
*  which associates a (at most 1) leaf node with a symbol (the node
*  containing the "initial value" of the symbol)
*
*  set_leaf() is called only once for a symbol, by the leaf
*  creating routine new_leaf(SYM_INFO*)
*/
extern NODE* leaf(SYM_INFO*); /* get leaf associated with symbol */
extern void set_leaf(SYM_INFO*,NODE*); /* set leaf associated with symbol */

/*
*  node creation functions. Each node has an opcode and a set
*  of symbols associated with it. A node may have up to 2 children.
*/
extern new_0_node(SYM_INFO*);
extern new_1_node(OPCODE,SYM_INFO*);
extern new_2_node(OPCODE,SYM_INFO*, SYM_INFO*);
extern node_add_sym(SYM_INFO*,NODE*);
/*
*  node finding function: returns node with given opcode and node arguments
*/
extern NODE* match(OPCODE,NODE*,NODE*)

NODE*
new_leaf(SYM_INFO* s) {
  NODE* n;
  n = new_0_node(s);
  set_leaf(s,n);
  set_node(s,n);
  node_add_sym(s,n);
  return n;
}

void
```

```
basic_block_2_dag(INSTRUCTION bb[],int len) {
  for (i=0;(i<len);++i) { /* for each instruction in the basic block */
    switch (type_of(bb[i]) {
      case BINOP: /* A = B op C */
        nb = node(B);
        if (!nb) /* B not yet in dag */
          nb = new_leaf(B);
        nc = node(C);
        if (!nc) /* C not yet in dag */
          nc = new_leaf(C);
        /* find op-node wich children nb,nc */
        n = match(op,nb,nc);
        if (!n)
          n = new_2_node(op,nb,nc);
        node_add_sym(A,n); set_node(A,n);
        break;
      case UNOP: /* A = op B */
        nb = node(B);
        if (!nb) /* B not yet in dag */
          nb = new_leaf(B);
        /* find op-node wich children nb,nc */
        n = match(op,nb,nc);
        if (!n)
          n = new_1_node(op,nb);
        node_add_sym(A,n); set_node(A,n);
        break;
      case ZOP: /* A = B */
        n = node(B);
        if (!n) /* B not yet in dag */
          n = new_leaf(B);
        node_add_sym(A,n); set_node(A,n);
        break;
      case CJUMP: /* if B relop C GOTO L */
        /* this must be the last instruction in the block! */
        nb = node(B);
        if (!nb) /* B not yet in dag */
          nb = new_leaf(B);
        nc = node(C);
        if (!nc) /* C not yet in dag */
          nc = new_leaf(C);
        n = new_2_node(relop,nb,nc);
        /* set_node(L,n); */
        break;
      default:
        break;
    }
  }
}
```

□

The result of executing the algorithm on the basic block B_2 from Figure 6.1 is shown in Figure 6.2.

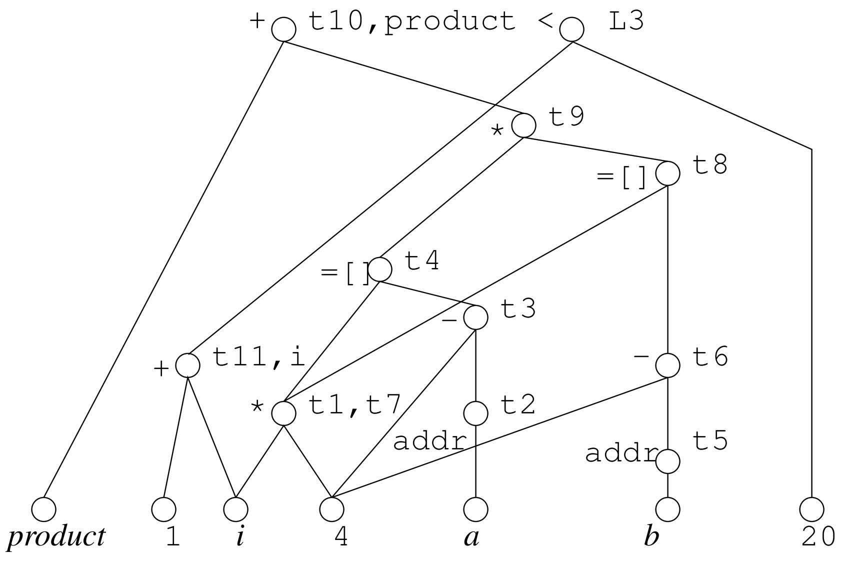


Figure 6.2: Dag of example code

The DAG representation of a basic block provides useful information:

- The range of the *leaf* map is the set of all symbols that are used in the basic block.
- The range of the *node* map contains the set of symbols that are available for use outside the basic block. In the absence of more precise global information, we must assume that they *will* be used and therefore we assume that these variables are *live*[1] at the end of the basic block. However, temporary variables created during intermediate code generation may be assumed not to be live.

Note that a variable available at the end of a basic block is not necessarily "live". Section 6.3.5 (page 110), contains a global analysis that obtains a more precise smaller estimate of which variables are really "live".

[1]See Definition 24, page 110, for a precise definition of "live variable".

6.2.2 Code simplification

Algorithm 9 [Code simplification in a basic block]
The DAG can also be used to perform code simplification *on the basic block. Basically, intermediate code is generated for each node in the block, subject to the following rules and constraints (we assume that the code does not contain array assignments or pointer manipulations):*

- *Code for children should precede the code for the parent. The code for the unique jump instruction should be generated last.*
- *When performing the operation corresponding to a node n we prefer a live target variable from the set $node(n)$.*
- *If $node(n)$ contains several live variables, add additional simple assignments of the form* `A = B` *to ensure that all live variables are assigned to.*
- *If $node(n)$ is empty, use a new temporary variable as the target.*
- *Do not assign to a variable v if its current value is still needed (e.g. because its leaf node still has an unevaluated parent).*

□

The result of applying the algorithm sketched above is shown below. Note that, compared with the original version, both the number of instructions and the number of temporary variables has decreased.

```
 1
 2
 3  t1 = 4*i
 4  t2 = addr a
 5  t3 = t1 -4
 6  t4 = t3[t1]
 7  t5 = addr b
 8  t6 = t5 - 4
 9  t8 = t6[t1]
10  t9 = t4 * t8
11  product = product + t9
12  i = i+1
13  if (i<20) goto 3
```

It should be noted that there are heuristics (see e.g Section 9.8 in [ASU86]) to schedule node evaluation of a dag which tend to have a beneficial effect on the number of registers that will be needed when machine code is generated [2].

[2]For trees there is even an optimal, w.r.t. register usage, algorithm.

6.2.3 Array and pointer assignments

The presence of array references and pointer manipulations complicates the code simplification algorithm.

Consider the intermediate code

```
x = a[i]
a[j] = y
z = a[i]
```

The corresponding dag is shown in Figure 6.3.

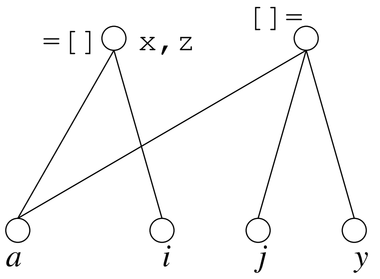


Figure 6.3: Dag of code with array manipulation

A possible simplified instruction sequence would be

```
x = a[i]
z = x
a[j] = y
```

which is clearly not equivalent to the original (consider e.g. what would happen if $i == j$ but $y \neq a[i]$.

The effect of an array assignment (`a[x]=y`) node should be that all nodes referring to an element of a should be *killed* in the sense that no further identifiers can be added to such nodes. Moreover, when generating simplified code, care should be taken that all array reference nodes that existed before the creation of the array assignment node should be evaluated before the assignment node. Similarly, all array reference nodes created after the array assignment node should be evaluated after that node. One way to represent such extra constraints is by adding extra edges to the dag from the "old" array reference nodes to the array assignment node and from that node to the "new" array reference nodes, as shown in Figure 6.4 (the extra constraining edges are shown as dashed arrows).

The effect of a pointer assignment `*a = b` is even more drastic. Since a can point anywhere, the node corresponding to such an instruction kills every node in the dag so far. Moreover, when generating simplified code, all "old" nodes will

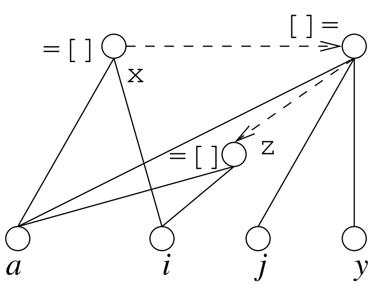


Figure 6.4: Corrected dag of code with array manipulation

have to be evaluated before the pointer assignment and all "new" nodes after it. Similarly, if we assume that a procedure can have arbitrary side effects, the effect of a `CALL` instruction is similar to the one of a pointer assignment.

6.2.4 Algebraic identities

The dag creation algorithm can be made more sophisticated by taking into account certain algebraic identities such as commutativity ($a * b = b * a$) and associativity.

Another possibility when processing a conditional jump instruction on $a > b$ is to check whether there is an identifier x holding $a - b$ and to replace the condition by $x > 0$ which is probably cheaper.

As an example, consider the intermediate code below.

```
t1 = b + c
a = t1
t2 = c + d
t3 = t2 + b
e = t3
```

The corresponding dags (with and without algebraic optimization) are shown in Figure 6.5. In the example, the first version of the dag is optimized by applying associativity to recognize $b + c$ as a common subexpression, eliminating the need for the temporary variable `t2`.

6.3 Global flow graph information

In this section we discuss several kinds of information that can be derived from the flow graph (and the instructions in it). In the next section, we will design code improving transformations that rely on this information.

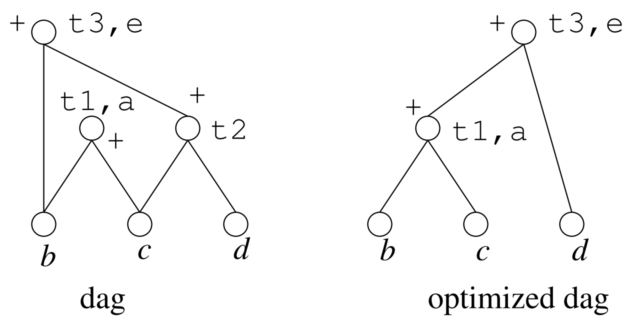


Figure 6.5: Dag using algebraic identities

Note that we assume that a flow graph corresponds to a single procedure. Also, we will not consider problems caused by *aliasing*, where two variables refer to the same location (aliasing may be caused by pointer manipulation and/or reference parameters in procedure calls). The impact of aliasing is briefly discussed in Section 6.5.

In the discussion below we use the following concepts related to flow graphs of basic blocks, which themselves consist of intermediate code instruction sequences.

- A basic block B' is a **successor** of a basic block B, denoted $B < B'$, if there is an edge from B to B' in the flow graph.
- We assume the existence of an **initial basic block** which represents the entry to the procedure.
- There are $n + 1$ **points** in a basic block with n instructions: one between each pair of consecutive instructions, one before the first instruction and one just after the last instruction.
- Consider all points in all basic blocks of a flow graph. A **path** from a point p_0 to a point p_n is a sequence of points $p_0, p_1, \ldots, p_{n-1}, p_n$ such that for each $0 \leq i < n$, either
  - p_i is the point immediately before an instruction s and p_{i+1} is the point immediately after s; or
  - p_i is (after) the end of some block and p_{i+1} is the point before the first instruction in a successor block.

Often, we will confuse an instruction s with the point immediately before it.

6.3.1 Reaching definitions

Definition 18 *A **definition** of a variable is an instruction that assigns, or may assign, a value to it. In the first case, we say that a definition is **unambiguous**, otherwise its is called **ambiguous**. The set of all definitions of a variable a is denoted by Def(a), while UDef(a) represents the set of unambiguous definitions of a*

*A definition d **reaches** a point p if there is a path from the point immediately following d to p such that d is not **killed** by an unambiguous definition of the same variable along that path.*

The phrase "*may* assign" is necessary since, e.g. a call to a procedure with x as a formal (reference) parameter may or may not result in an assignment to x. A similar reasoning holds for an assignment through a pointer or to an array element.

Note that *Def*(a) is easy to compute.

Thus, if a definition d for x reaches a point p, the value of x at p may be given by the value assigned at d.

The following basic block properties will be useful later on:

Definition 19 *Let B be a basic block.*

$$\begin{array}{rcl}
GEN(B) & = & \{d \in B \mid \textit{ d is a definition that reaches the end of } B\} \\
KILL(B) & = & \{d \not\in B \mid d \in \textit{Def}(x) \textit{ and } B \cap \textit{UDef}(x) \neq \emptyset\} \\
IN(B) & = & \{d \mid d \textit{ is a definition and } d \textit{ reaches the start of } B\} \\
OUT(B) & = & \{d \mid d \textit{ is a definition and } d \textit{ reaches past the end of } B\}
\end{array}$$

Observe that $GEN(B)$, which depends only on B, is straightforward to construct using a backward scan of B (after seeing an unambiguous definition for a variable x, no more definitions for x should be added to $GEN(B)$). Similarly, given the set DEF of all definitions in the graph, the set $KILL(B)$ can easily be computed by keeping only those definitions from DEF that define a variable for which an unambiguous definition in B exists.

It is easy to see that $IN(B)$ and $OUT(B)$ must satisfy the following data-flow equations:

$$\begin{array}{rcl}
OUT(B) & = & (IN(B) \setminus KILL(B)) \cup GEN(B) \\
IN(B) & = & \cup_{C<B} OUT(C)
\end{array}$$

If there are n blocks in the graph, we have $2n$ equations with $2n$ unknowns ($GEN(B)$ and $KILL(B)$ can be determined independently). Due to the possible presence of cycles in the flow graph, these equations usually do not have a unique solution.

E.g. consider the graph in Figure 6.6.

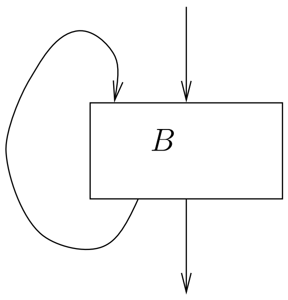


Figure 6.6: Flow graph with cycle

Clearly, if $IN(B) = I_0$ and $OUT(B) = O_0$ form a solution and d is a definition such that $d \notin I_0 \cup O_0 \cup KILL(B)$, then $IN(B) = I_0 \cup \{d\}$ is also a solution. It turns out that the smallest solution is the desired one[3]. It can be easily computed using the following (smallest) fixpoint computation.

Algorithm 10 [Computing definitions available at a basic block]

```
typedef  set<Instruction*>  ISET;
typedef  ..       BLOCK;

ISET  in[BLOCK], out[BLOCK], kill[BLOCK], gen[BLOCK];

void
compute_in_out() {
  for all b in BLOCK {
    in[b] = emptyset;
    out[b] = gen[b];
  }

  bool change = false;

  do {
    change = false;
    foreach b in BLOCK {
```

[3]The solutions are closed under intersection

```
      newin = set_union { out[c] | c < b };
      change =  change || (newin != in[b]);
      in[b] = newin;
      out[b] = (in[b] - kill[b]) set_union gen[b]
    }
  while change;
  }
}
```

□

Definition 20 *Let a be a variable and let u be a point in the flow graph. The set of definitions of a that reach u is defined by*

$$UD(a,u) = \{d \in \mathit{Def}(a) \mid d \text{ reaches } u\}$$

Such a set is often called a ***use-definition*** *chain.*

$UD(a,u)$ can be easily computed using $IN(B)$ where u is the basic block containing u.

Algorithm 11 [Computing use-definition chains]

```
iset
defu(SYM_INFO* a,POINT u) {
  BLOCK b = block_of(u);

  if (Def(a) intersection B != emptyset)
    /* return the singleton containing the last
     * definition in b of a before u
     */
    for (p=u; (p>0); --p)
      if (p in UDEF(a))
        return {p}
  return Def(a) intersect in[b]
}
```

□

6.3.2 Reaching definitions using datalog

An alternative definition uses logic programming (in particular, datalog) to define reaching definitions, see [ALSU07], page 925.

We will use a term `B.N`, $N \geq 0$ to denote the point in block B us before the $N+1$'th statement (in block B). The latter statement is said to be at point `B.N`. We start with the following base predicates.

```
assign(B.N, X, E) % the statement following B.N has the
%  form X = E, E an expression (with at most 2 operands)
use(B.N, X) % the statement at B.N uses the variable X
type(X,T) % type type of X is T
size(B, N) % block B has N statements
succ(B, C) % true iff block B has a successor block C
initial(B) :- not( succ(C,B) ). % initial block has no predecessors
```

We can then define what it means for a statement to (ambiguously or unambigously) define a variable X (here we assume a strongly typed language without casting).

```
% unambig_def(B.N, X) -- statement N in block B unambiguously defines variable X
unambig_def(B.N, X) :- assign(B.N, X, _).

% def(B.N, X) -- statement N in block B may define variable X
define(B.N, X) :- unambig_def(B.N, X).
define(B.N, X) :- assign(B.N, *P, _), type(X, T), type(P, T*).
```

Then the following program computes the reaching definitions.

```
% rd(B.N, C.M, X) -- definition of X at C.M reaches B.N
rd(B.N, B.N, X) :- define(B.N, X).
rd(B.N, C.M, X) :- rd(B.N-1, C.M, X), not(unambig_def(B.N-1, X)).
rd(B.0, C.M, X) :- rd(D.N, C.M, X), succ(D, B), size(D,N).
```

The first rule simply states that a definition reaches itself while the second rule indicates that if a definition reaches a point $N-1$ in a block B and the next statement does not unambiguously (re)define X, then the definition also reaches point N (after the next statement) in B. The third rule says that a definition reaches the beginning of a block B if it reaches the end of a predecessor D of B.

A use-definition predicate be defined as follows, where the use u is represented by a point N in a block B, and d by its block C and point M):

```
ud(A, u(B.N), d(C.M)) :- rd( B.N, C.M, A), use(B.N, A).
```

6.3.3 Available expressions

Definition 21 *An expression $e = x+y$[4] is* ***available*** *at a point p, denoted $AVAIL(e,p)$ if every path from the initial basic block to p evaluates $z = x + y$, and after the*

[4]We use '+' as an example operator. The same definition applies to other binary or unary operators.

last such evaluation prior to p, there are no subsequent definitions for x, y or z. A block ***kills*** *an expression $x + y$ if it defines x or y and does not subsequently (re)compute $x + y$. A block* ***generates*** *an expression $x + y$ is if it unambiguously evaluates $x + y$ and does not subsequently redefine x or y.*

In order to be able to determine the availability of an expression at some point, we define

Definition 22 *Let B be a basic block.*

$$\begin{aligned}
GEN_e(B) &= \{e \mid B \textit{ generates } e\} \\
KILL_e(B) &= \{e \mid B \textit{ kills } e\} \\
IN_e(B) &= \{e \mid e \textit{ is available at the start of } B\} \\
OUT_e(B) &= \{e \mid e \textit{ is available past the end of } B\}
\end{aligned}$$

Observe that $GEN_e(B)$, which depends only on B, is straightforward to construct using a forward scan of B, keeping track of available expressions by appropriately processing definitions. E.g. an instruction $z = x + y$ makes $x + y$ available but kills any available expressions involving z (note that $z = x$ is possible). The set $KILL_e(B)$ consists of all expressions $y + z$ such that either y or z is defined in the block and $y + z$ is not generated by the block.

It is easy to see that $IN_e(B)$ and $OUT_e(B)$ must satisfy the following data-flow equations:

$$\begin{aligned}
OUT_e(B) &= (IN(B) \setminus KILL_e(B)) \cup GEN_e(B) \\
IN_e(B) &= \begin{cases} \cap_{C<B} OUT_e(C) & \text{if } B \text{ is not initial} \\ \emptyset & \text{if } B \text{ is initial} \end{cases}
\end{aligned}$$

Note the similarity of these equations with the ones for reaching definitions. The main difference is the use of $\cap$ instead of $\cup$ in the equation for $IN_e(B)$ ($\cap$ is necessary since e must be available on *all* paths into B in order to be sure that e is available at the start of B).

Consider the flow graph in Figure 6.7 and assume that neither x nor y are defined in block B after d which is itself the final definition of z in block B. Assume furthermore that block C does not contain any definitions for x, y or z. Hence $x + y$ should be available at the beginning of block C, i.e. $x + y \in IN_e(C)$, which would not be found if the minimal solution for the above equations were computed.

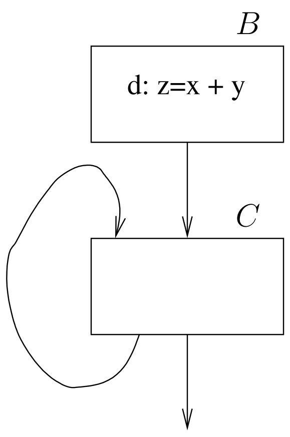


Figure 6.7: Example flow graph for available expressions

Since, clearly, the solutions for $IN_e(C)$ should be closed under union, it turns out that the intended solution is the maximal one. Hence, in the following fixpoint procedure, we compute the greatest fixpoint.

Algorithm 12 [Computing expressions available at a basic block]

```
typedef  .. EXPRESSION;
typedef set<EXPRESSION> ESET;
typedef  .. BLOCK;

extern  BLOCK* initial_block();
extern  ESET all_expressions();

ESET ine[BLOCK], oute[BLOCK], kille[BLOCK], gene[BLOCK];

void
compute_avail_exp() {
  BLOCK b1 = initial_block();

  /* initialization */

  ine[b1] = empyset;  /* never changes */
  out[b1] = gene[b1];  /* never changes */

  for all b in (BLOCK - {b1}) {
    oute[b] = all_expressions() - kille[b]
  }

  bool change = false;

  do {
    change = false;
    for all b in (BLOCK - {b1}) {
      ine[b] = intersection { oute[c] | c < b }
      oldout = oute[b];
      oute[b] = gene[b] set_union (ine[b] - kille[b]);
      change = change || (oute[b]!=oldout);
    }
  }
  while (change);

}
```

□

Note that, based on $IN_e(B)$, it is easy to determine $AVAIL(e = x + y, p)$, for $p \in B$.

Algorithm 13 [Computing expression availability at a point]
Let $p \in B$ be a point where we want to compute $AVAIL(e, p)$.

- *Check whether $e \in IN_e(B)$.*
- *If so, check whether x or y are redefined before p in B. If not, e is available; else not.*
- *If $e \notin IN_e(B)$, check whether e is generated before p in B. If so, e is available; else not.*

□

6.3.4 Available expressions using datalog

In datalog, the formulation is slightly more involved than might be expected because the usual datalog semantics computes minimal solutions. The approach is to define a predicate `unavail_exp/2` asserting that an expression is not available at some point. The availability of an expression at a point then hold is `unavail` does not hold at that point.

```
define_exp(B.N, X+Y) :- assign(B.N, _, X+Y).
% a possibled definition of any operand kills the expression
kill_exp(B.N, X+Y) :- define(B.N, X, _).
kill_exp(B.N, X+Y) :- define(B.N, Y, _).
```

```

unavail_exp(B.N, X+Y) :- kill_exp(B.N
unavail_exp(B.N+1, X+Y) :-
  unavail_exp(B.N, X+Y), not(define_exp(B.N+1, X+Y)).
% if X+Y is unavaible at the end of a predecessor of B,
% it is also unavailable at the beginning of B
unavail_exp(B.0, X+Y) :- succ(C, B), size(C,N), unavail_exp(C.N, X+Y).
% at the start of the initial block, all expressions are unavailable
unavail_exp(B.0, X+Y) :- initial(B).
% an expression is available if it is not unavailable
avail(B.N, X+Y) :- not(unavail_exp(B.N, X+Y)).
```

6.3.5 Live variable analysis

Definition 23 *A variable x is **used** at instruction s if its value may be required by s.*

Definition 24 *Let x be a variable and let p be a point in the flow graph. We say that x is **live** at p is there is some path from p on which the value of x at p can be used. If x is not live at p, we say that it is **dead** at p.*

In datalog this becomes:

```
live(B.N, X) :- assign(B.N, _, X + Y).
live(B.N, X) :- assign(B.N, _, Y + X).
live(B.N-1, X) :- live(B.N, X),
                  not(unambig_def(B.N-1, X).
live(B.N, X) :- size(B, N), succ(B, C), live(C.0, X).
```

In order to be able to determine whether a variable is live at some point, we define

Definition 25 *Let B be a basic block.*

$$\begin{aligned}
KILL_{live}(B) &= \{x \mid x \textit{ is unambiguously defined before used in } B\} \\
GEN_{live}(B) &= \{x \mid x \textit{ may be used in } B \textit{ before it is unambiguously defined in } B\} \\
IN_{live}(B) &= \{x \mid x \textit{ is live at the start of } B\} \\
OUT_{live}(B) &= \{x \mid x \textit{ is live past the end of } B\}
\end{aligned}$$

Note that, in the definition of $\mathit{KILL}_{\mathit{live}}(B)$, it is not necessary that x is actually used in B, as long as there is an unambiguous definition of x before any use. Similarly, in the definition of $\mathit{IN}_{\mathit{live}}(B)$, it may well be that $x \in \mathit{IN}_{\mathit{live}}(B)$ although x is not (unambiguously) defined in B.

Observe that both $\mathit{KILL}_{\mathit{live}}(B)$ and $\mathit{GEN}_{\mathit{live}}(B)$, which depend only on B, are straightforward to construct.

To compute $\mathit{IN}_{\mathit{live}}(B)$ and $\mathit{OUT}_{\mathit{live}}(B)$, we note that they satisfy the following data-flow equations:

$$\begin{aligned} \mathit{IN}_{\mathit{live}}(B) &= \mathit{GEN}_{\mathit{live}}(B) \cup (\mathit{OUT}_{\mathit{live}}(B) - \mathit{KILL}_{\mathit{live}}(B)) \\ \mathit{OUT}_{\mathit{live}}(B) &= \cup_{B<C} \mathit{IN}_{\mathit{live}}(C) \end{aligned}$$

Again we will use a fixpoint computation to determine the smallest solution of the above equations. Note that intuitively, the algorithm works “backwards” in the graph, as can be seen from the use of successors instead of predecessors in the equation for $\mathit{OUT}_{\mathit{live}}(B)$.

Algorithm 14 [Computing live variables for a basic block]

```
typedef set<SYM_INFO*>  VSET;
typedef ..              BLOCK;

VSET    in_live[BLOCK], out_live[BLOCK], use[BLOCK], gen_live[BLOCK];

void
compute_live()
{
for all b in BLOCK
        in_live[b] = emptyset;

bool change = true;

while (change) {
        change = false;
        foreach b in BLOCK {
                oldin = in_live[b];
                out_live[b] = union { in[s] | b < s };
                in_live[b] = use[b] union (out_live[b] - gen_live[b]);
                change = change || (oldin!=in_live[b]);
                }
        }
}
```

□

Note that the information from $OUT_{live}(B)$ can be profitably used to refine Algorithm 9.

6.3.6 Definition-use chaining

Definition 26 *The definition-use chain of a definition* d *is defined as*

$DU(d,x) = \{s \mid s$ *uses* x *and there is a path from* d *to* s *that does not redefine* $x\}$

The following sets, defined for basic blocks, facilitate the computation of $DU(d,x)$.

Definition 27 *Let* B *be a basic block.*

$$\begin{array}{rcl}
\mathit{KILL}_{use}(B) & = & \{(s,x) \mid s \notin B \textit{ uses } x \textit{ and } B \textit{ defines } x\} \\
\mathit{GEN}_{use}(B) & = & \{(s,x) \mid s \in B \textit{ uses } x \textit{ and } x \textit{ is not defined prior to } s \textit{ in } B\} \\
\mathit{IN}_{use}(B) & = & \{(s,x) \mid s \textit{ uses } x \textit{ and } s \textit{ reachable from the beginning of } B\} \\
\mathit{OUT}_{use}(B) & = & \{(s,x) \mid s \notin B \textit{ uses } x \textit{ and } s \textit{ reachable from the end of } B\}
\end{array}$$

Here, "reachable" means without any intervening definitions of x*.*

Observe that both $\mathit{KILL}_{use}(B)$ and $\mathit{GEN}_{use}(B)$ are straightforward to construct. To compute $\mathit{IN}_{use}(B)$ and $\mathit{OUT}_{use}(B)$, we note that they satisfy the following data-flow equations:

$$\begin{array}{rcl}
\mathit{IN}_{use}(B) & = & \mathit{GEN}_{use}(B) \cup (\mathit{OUT}_{use}(B) - \mathit{KILL}_{use}(B)) \\
\mathit{OUT}_{use}(B) & = & \cup_{B<C} \mathit{IN}_{use}(C)
\end{array}$$

Since the above equations are isomorphic to the ones in Section 6.3.5, an algorithm similar to Algorithm 14 can be used to compute $\mathit{OUT}_{use}(B)$. $DU(d,x)$ can then be computed by.

$$DU(d,x) = \begin{cases} \mathit{OUT}_{use}(B) \cup \{(s,x) \mid s \in B \text{ uses } x\} \\ \quad \text{if } B \text{ does not contain a definition of } x \text{ after } d \\ \{(s,x) \mid s \in B \text{ uses } x \text{ and } s \text{ comes before } d'\} \\ \quad \text{if } d' \in B \text{ is the first definition of } x \text{ after } d \end{cases}$$

6.3.7 Application: uninitialized variables

To illustrate the use of global flow information, we show how to detect uninitialized variables.

It suffices to add a dummy assignment d_x:$x = 0$ for each variable x in front of the initial block. Now if for any such d_x, it turns out that $DU(d_x, x) \neq \emptyset$, we can deduce that x is possibly used before it is assigned to.

6.4 Global optimization

6.4.1 Elimination of global common subexpressions

Algorithm 15 [Elimination of global common subexpressions]
For every instruction s:$x = y + z$[5] such that $AVAIL(y + z, p)$ where p is the point before s, do the following (assume $s \in B$):

1. *Find the evaluations of $y + z$ that reach B by searching the graph (and each block) backwards from B, not continuing after a block that evaluates $y + z$.*
2. *Create a new temporary variable u.*
3. *Replace each instruction $w = y + z$ found in step 1 by $u = y + z; w = u$.*
4. *Replace s by $x = u$.*

□

Note that the algorithm will miss the fact that $a * z$ is the same as $c * z$ in

```
a = x + y
b = a * z
...
c = x + y
d = c * z
```

Figure 6.8: Code before first subexpression elimination

Algorithm 15 will produce the code in Figure 6.9. However, repeated application of Algorithm 15, combined with copy propagation (Algorithm 16), will eventually optimize the code further.

[5] Here '$+$' stands for any operator.

```
u = x + y
a = u
b = a * z
...
c = u
d = c * z
```

Figure 6.9: Code after first subexpression elimination

6.4.2 Copy propagation

Copy instructions of the form $x = y$ are generated e.g. by intermediate code generation and by Algorithm 15. Copy propagation can be used to eliminate many such instructions by replacing further uses of x by y.

One can eliminate a copy instruction $s{:}x = y$ and substitute y for x in all uses of x in $s' \in DU(s, x)$ provided that:

- $UD(s', x) = \{s\}$, i.e. s is the only definition of x to reach s'.
- No paths from s to s', including paths that go through s' several times, contains definitions of y.

In datalog:

```
% copy(B.N, X, Y) -- X is copy of Y at B.N
copy(B.N, X, Y) :- assign(B.N, X, Y).
copy(B.N, X, Y) :- copy(B.N-1, X, Y),
        not( def(B.N-1, X) ), not( def(B.N-1, Y) ).
copy(B.0, X, Y) :- not( initial(B) ),
     not (
       succ(C, B), size(C, M), not( copy(C.M, X, Y) )
       ).
```

To solve the latter question, we solve yet another data-flow problem where $IN_{copy}(B)$ ($OUT_{copy}(B)$) is the set of copies $s : x = y$ such that *every* path from the initial node to the beginning (end) of B contains s, and subsequent to the last occurrence of s, there are no definitions of y. We also consider $GEN_{copy}(B)$, the set of copies $s{:}x = y$ generated by B where $s \in B$ and there are no subsequent definitions of x or y within B. $KILL_{copy}(B)$ contains the set of copies $s{:}x = y$ where either x or y is defined in B and $s \notin B$. Note that $IN_{copy}(B)$ can contain only one copy instruction with x on the left since, otherwise, both instructions would kill each other.

Then the following equations hold:

$$\begin{aligned} OUT_{copy}(B) &= GEN_{copy}(B) \cup (IN_{copy}(B) \setminus KILL_{copy}(B)) \\ IN_{copy}(B) &= \begin{cases} \cap_{C<B} OUT_{copy}(C) & \text{if } B \text{ is not initial} \\ \emptyset & \text{if } B \text{ is initial} \end{cases} \end{aligned}$$

Note that these equations are isomorphic to the ones used for Algorithm 12 and therefore we can use a similar algorithm to solve them.

Given $IN_{copy}(B)$, i.e. the set of copies $x = y$ that reach B along every path, with no definition of x or y following the last occurrence of $x = y$ on the path, we can propagate copies as follows.

Algorithm 16 [Copy propagation]
For each copy instruction s:$x = y$ do the following:

1. *Determine $DU(s, x)$, i.e. the set of uses s' that are reached by s.*
2. *Check whether for* every $s' \in DU(s, x)$, $s \in IN_{copy}(B_{s'})$[6], *i.e. s reaches $B_{s'}$ on* every *path and, moreover, no definitions of x or y occur prior to s' in B.*
3. *If the previous step succeeds, remove s and replace x by y in each $s' \in DU(s, x)$.*

□

Applying Algorithm 16 on the code of Figure 6.9 yields

```
u = x + y
b = u * z
...
d = u * z
```

Now, running Algorithm 15 again will result in

```
u = x + y
v = u * z
b = v
...
d = v
```

[6] $B_{s'}$ is the basic block containing s'.

6.4.3 Constant folding and elimination of useless variables

Another transformation concerns the propagation of constants. As an example, consider the code

```
t0 = 33
t1 = 3
t2 = t0 + t1
t3 = t2 - 35
x = t3;
```

Algorithm 17 [Constant folding]
Call a definition s:$x = a + b$[7] "constant" if all its operands a and b are constant.

- *We can then replace s by $x = c$ where $c = a + b$ is a new constant.*
- *For each constant definition s and for each $s' \in DU(x, s)$, if $UD(x, s') = \{s\}$, i.e. s is the only x-definition reaching s', we can replace x in s' by c, possibly making s' constant. If after this, $DU(x, s) = \emptyset$, we can eliminate s.*
- *Repeat the above steps until there are no further changes.*

□

Note that the second step above can be seen as a special simple case of copy propagation since the right hand side c of the copy instruction $x = c$ is constant and therefore there are no c-assignments to worry about.

In the example, application of the algorithm sketched above yields

```
x = 1
```

6.4.4 Loops

It is a well-known piece of computer science folklore that programs tend to spend 90% of their execution time in 10% of their code. One can use a profiling tool such as *gprof* to collect statistics on a program run from which it can be deduced which routines and instructions are most often executed. These "hot spots" will almost always turn out to be loops. So it is natural to concentrate optimization efforts on loops.

Before considering loop optimization, we need to define what constitutes a loop in a flow graph.

[7] Here $+$ may be any operator.

Definition 28 *Let G be a flow graph with an initial basic block node b_0.*

- *A basic block b **dominates** another block b' iff every path from b_0 to b' passes through b.*
- *A **back edge** is an edge $n \rightarrow b$ such that b dominates n.*
- *The **(natural) loop** L of a back edge $n \rightarrow h$ consists of h and all basic blocks x such that there is a path from x to n not going through h; h is called the **header** of the loop.*
- *An **inner loop** is a loop that contains no other loops.*
- *An **exit node** of a loop L is a block $e \in L$ that has a successor outside L.*

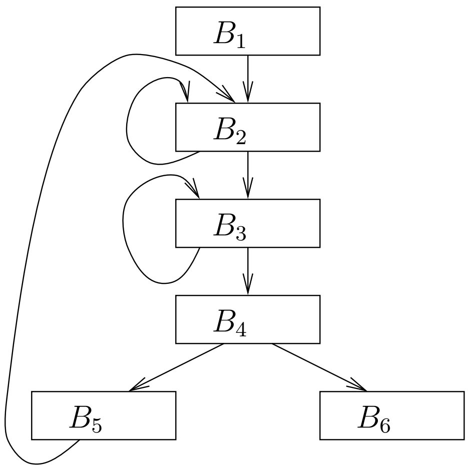


Figure 6.10: Flow graph with loops

In Figure 6.10, there are three loops: $L_0 = \{B_2\}$, $L_1 = \{B_3\}$, $L_2 = \{B_2, B_3, B_4, B_5\}$. B_2 is the header of L_2, while B_4 is L_2's single exit node.

One crude way to compute the dominators of a block b is to take the intersection of all acyclic paths from the initial node to b.

A smarter algorithm relies on the fact that

$$dom(b) = \{b\} \cup \cap_{c<b} dom(c)$$

i.e. a node that dominates all predecessors of b also dominates b.

Algorithm 18 [Computing dominators]

```
typedef set<BLOCK*>     BLOCKS;

BLOCK*          b0; /* initial basic block */
set<BLOCK*>     B; /* set of all blocks */

void
compute_dominates()
{
/* initialize */

dom(b0) = { b0 };
for b in (B - {b0})
        dom(b) = B;

/* compute fixpoint */

do
        {
        change = false;
        for b in (B-{b0})
                {
                old_dom = dom(b);
                dom(b) = {b} union intersection { dom(c) | c < b }
                change = change || (dom(b)!=old_dom);
                }
        }
while (change)
}
```

□

The table below shows the application of Algorithm 19 to the flow graph of Figure 6.10.

block	1	2	3	4	5	6
predecessors		1,2,5	2,3	3	4	4
dominators initialize	1	1,2,3,4,5,6	1,2,3,4,5,6	1,2,3,4,5,6	1,2,3,4,5,6	1,2,3,4,5,6
dominators iteration 1	1	1,2	1,2,3	1,2,3,4	1,2,3,4,5	1,2,3,4,6
dominators iteration 2	1	1,2	1,2,3	1,2,3,4	1,2,3,4,5	1,2,3,4,6

The following algorithm can be used to compute the natural loop corresponding to a back edge. It corresponds to a backward scan from the header.

Algorithm 19 [Computing the natural loop of a back edge]

```
type    set<BLOCK*>     BLOCKS;

typedef struct {
        BLOCKS  nodes;
        BLOCK*  header;
        } LOOP;

BLOCKS
predecessors(BLOCK *b); /* return predecessors of b in flow graph */

LOOP
natural_loop(BLOCK* src,BLOCK* trg)
{
/* compute natural loop corresponding to back edge src-->trg */

BLOCK*  stack[];
BLOCK*  tos = stack;
LOOP    loop = { { trg }, trg }; /* trg is header */

if (trg!=src) {
        loop.nodes = loop.nodes union { src }
        stack[tos++] = src; /* push src */
        }

while (tos!=stack) { /* stack not empty */
        BLOCK*  b = stack[--tos]; /* pop b */
        BLOCKS  P = predecessors(b);

        for each p in P
                if (!(p in loop)) {
                        loop.nodes = loop.nodes union {p};
                        stack[tos++] = p;
                        }

        }
}
```

□

Applying Algorithm 19 to the back edge $B_5 \rightarrow B_2$ in the flow graph of Figure 6.10 is illustrated in the table below.

faze	loop.nodes	stack
initialize	$\{2\}$	
adding src	$\{2,5\}$	5
iteration 1	$\{2,5,4\}$	4
iteration 2	$\{2,5,4,3\}$	3
iteration 3	$\{2,5,4,3\}$	

6.4.5 Moving loop invariants

A *loop invariant* is a instruction of the form $s{:}x = y + z$[8] such that x does not change as long as control stays in the loop. Such instructions can, under certain conditions, be moved outside of the loop so that the value x is computed only once.

Example 20 Consider a C function to compute the scalar product of two vectors (represented as arrays).

```
int
innerproduct(int a[]; int b[], int s) {
// s is size of arrays
int r = 0;
int i = 0;
while (i<s) {
        r = r + a[i]*b[i];
        i = i+1;
        }
return r;
}
```

The flow graph, after code simplification, is shown in Figure 6.11 on page 121. Note that the size of an `int` is assumed to be 4 bytes, as witnessed by the offset computation `t1 = 4* i`.

Applying Algorithm 19 on the graph of Figure 6.11 yields that, among others, C dominates A, B and D and that, consequently, $D \rightarrow C$ is a back edge. It then follows from Algorithm 19 that $L = \{D, C\}$ is an inner loop with exit node C.

Algorithm 20 [Computing loop invariants]
Given a loop L, we find the list of loop invariants from L as follows (note that the order of the list is important, as it reflects the order of the instructions after having been moved):

1. *Mark "invariant" (and append to the list of invariants) all instructions $s{:}x = y + z$ such that each operand y (or z) is either constant or is such that $UD(y, s) \cap L = \emptyset$, i.e. all its reaching definitions are outside L.*

2. *Repeat the following step until no more instructions are marked.*

3. *Mark (and append to the list) "invariant" all instructions $s{:}x = y + z$ such that each operand y (or z) is*

[8]Here $+$ stands for any operator.

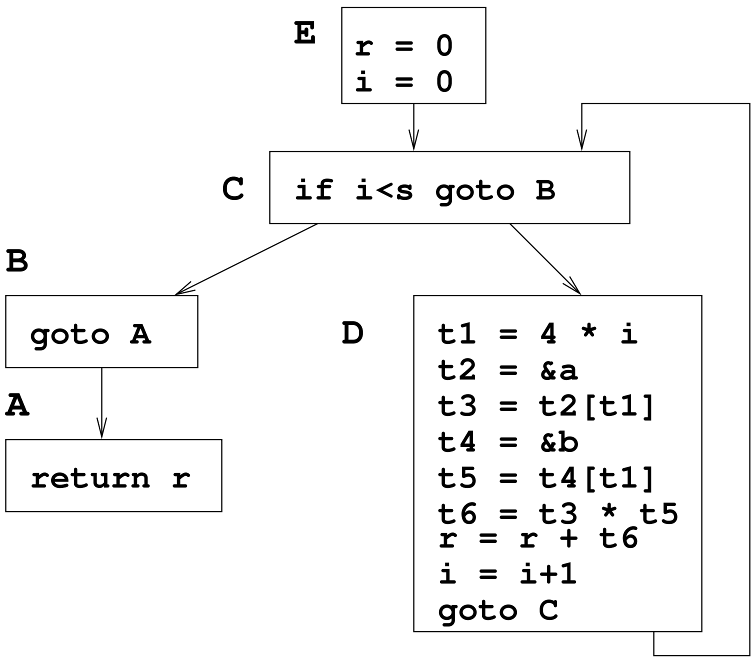


Figure 6.11: Flow graph of Example 20

- *constant; or*
- *all its reaching definitions are outside L; or*
- $UD(y, s) = \{q\}$ *where* $q \in L$ *is marked invariant; i.e. the only reaching definition of* y *is an invariant in the loop.*

□

Applying Algorithm 20 for Example 20 yields two loop invariants: `t2 = &a` and `t4 = &b`, both in block D.

Having found the invariants in a loop, we can move some of them outside the loop. They will be put in a new node, called a *preheader* which has as its single successor the header of the loop. The flow graph is further updated such that all edges that pointed from outside the loop to the header, will be made to point to the new preheader node.

Algorithm 21 [Moving invariants]
Let I be the list of invariants of the loop L.

1. *For each* s:$x = y + z$ *from I, check that:*

   (a) *s is in a block b that dominates all exits of L or $x \notin IN_{live}(e)$ for any successor $e \notin L$ of an exit block of L; and*

   (b) *x is not defined anywhere else in L; and*

   (c) *For each use s' of x in L, $UD(x, s') \cap L = \{s\}$, i.e. all uses of x in L can only be reached by the definition s of x*

2. *Move instructions from I that satisfy the above test, and that do not depend on definitions in I that do not pass the test, to the preheader, in the same order as they appear in I.*

$\square$

Figure 6.12 shows the result of applying Algorithm 21 to Example 20. Note that the block (D) containing `t2 = &a` does not dominate all exits of L. However, since `t2` $\notin IN_{live}(B)$, condition (1a) above is still satisfied.

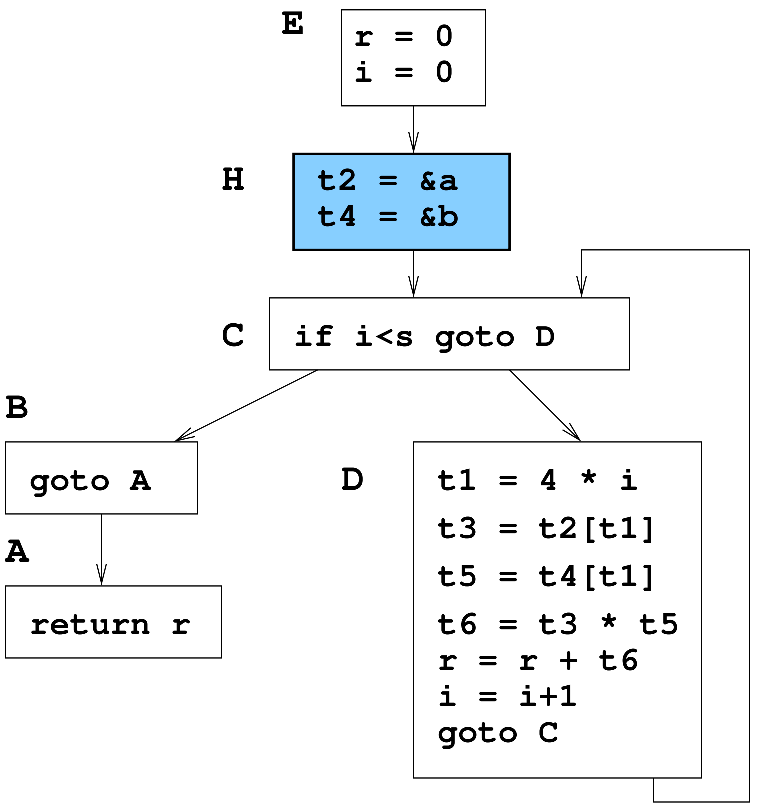


Figure 6.12: Flow graph of Example 20 after moving loop invariants

6.4.6 Loop induction variables

A variable x is called an *induction variable* of a loop L if every time it changes value, it is incremented or decremented by some constant. Examples include array indexes that are incremented on each pass through the loop, counters corresponding to *for* instructions etc.

Often, one induction variable depends on another one.

E.g. in Example 20, both `i` and `t1` are induction variables where `t1` depends on `i`.

Definition 29 *Let L be a loop. A **basic induction variable** is a variable i such that the only assignments to i in L are of the form $i = i \pm c$ where c is a constant. An **induction variable** is a variable j such that j is defined only once in L and its value is a linear function of some basic induction variable i, i.e. $j = a \times i + b$ for some constants a, b. We say that j belongs to the **family** of i, and associate the triple (i, a, b) with j.*

The following algorithm determines the induction variables of a loop.

Algorithm 22 [Finding the induction variables of a loop]
Given a loop L, we proceed as follows:

1. *Find all basic induction variables i. Such variables are associated with the triple $(i, 1, 0)$.*

2. *Find variables k such that Def$(k) \cap L$ is a singleton s which has one of the forms $k = j \times b$, $k = b \times j$, $k = j/b$, $k = b/j$, $k = j \pm b$, or $k = b \pm j$, where b is a constant and j is an induction variable, basic or otherwise.*

   - *If j is basic, then k is in the family of j and we associate k with the appropriate triple. E.g. if s:$k = j \times b$ then the triple of k is $(i, b, 0)$.*
   - *If j is not basic, then j belongs to the family of some basic induction variable i. We first check the following additional conditions:*
     - *There is no definition of i between the (single) definition of j in L and the (single) definition of k; and*
     - *$(UD(j, s) \setminus L) = \emptyset$, i.e. no definition of j outside L reaches s.*

     Provided these conditions are met, we add k to the family of i and associate it with the triple $(i, b \times c, b \times d)$, where we assume that s is of the form $k = j \times b$ and the triple of j is (i, c, d).

□

Applying Algorithm 22 to the loop $L = \{C, D\}$ in Figure 6.12 yields:

- a basic induction variable `i` which is associated with the tuple $(i, 1, 0)$.
- an induction variable `t1` in the family of `i`. `t1` is associated with the tuple $(i, 4, 0)$.

Once we have found suitable induction variables, we can perform strength reduction, where expensive multiplications are replaced by cheaper additions, as in the following algorithm

Algorithm 23 [Strength reduction]
Given a loop L and a set of induction variables, each with associated triple, we consider each basic induction variable i in turn.
For each non-basic induction variable j with associated triple (i, c, d) we do the following:

1. *Create a new variable t (but if two variables j_1 and j_2 have the same triples, create only one variable for both of them).*
2. *Replace the unique assignment to j in L by $j = t$.*
3. *Immediately after each assignment $i = i + a$ (a constant) to the basic induction variable i in L, add an instruction $t = t + c \times a$ and place t in the family of i with triple (i, c, d).*
4. *To properly initialize t, put*

$$\begin{aligned} t &= c \times i \\ t &= t + d \end{aligned}$$

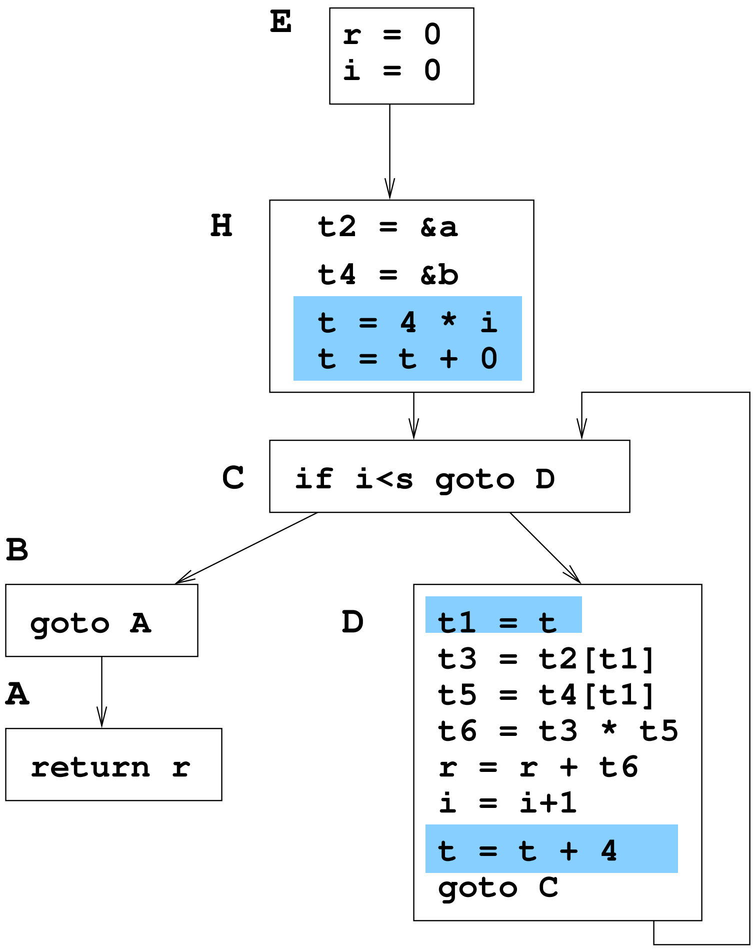
at the end of the preheader of L.

□

The result of applying Algorithm 23 to the loop $L = \{C, D\}$ in Figure 6.12 is shown in Figure 6.13.

Block D can be further simplified using Algorithm 9 (code simplification): because `t1` $\notin OUT_{live}(D)$, `t1` will be eliminated. On the other hand, block B can be simplified using copy propagation and constant folding (Algorithms 16 and 17). The result is shown in Figure 6.14.

After strength reduction, it may be that the only use of some induction variable i is in tests such as



```
r = 0
i = 0
```

```
t2 = &a
t4 = &b
t = 4 * i
t = t + 0
```

```
if i<s goto D
```

```
goto A
```

```
return r
```

```
t1 = t
t3 = t2[t1]
t5 = t4[t1]
t6 = t3 * t5
r = r + t6
i = i+1
t = t + 4
goto C
```

Figure 6.13: Flow graph of Example 20 after strength reduction

```
if i>C goto XXX
```

(and in the computation of other induction variables). In such a case, it may be profitable to replace the variable in the test by another induction variable j from the same family, and modify the test accordingly.

E.g., in our example (Figure 6.14), the triple of t is $(i, 4, 0)$, and we can replace the code in block C by

```
s1 = 4 * s
if t<s1 goto B
```

After this it may become possible to eliminate i since it is only used to compute itself. It may also be that the definition of the new temporary corresponds to a loop invariant which may be moved outside the loop.

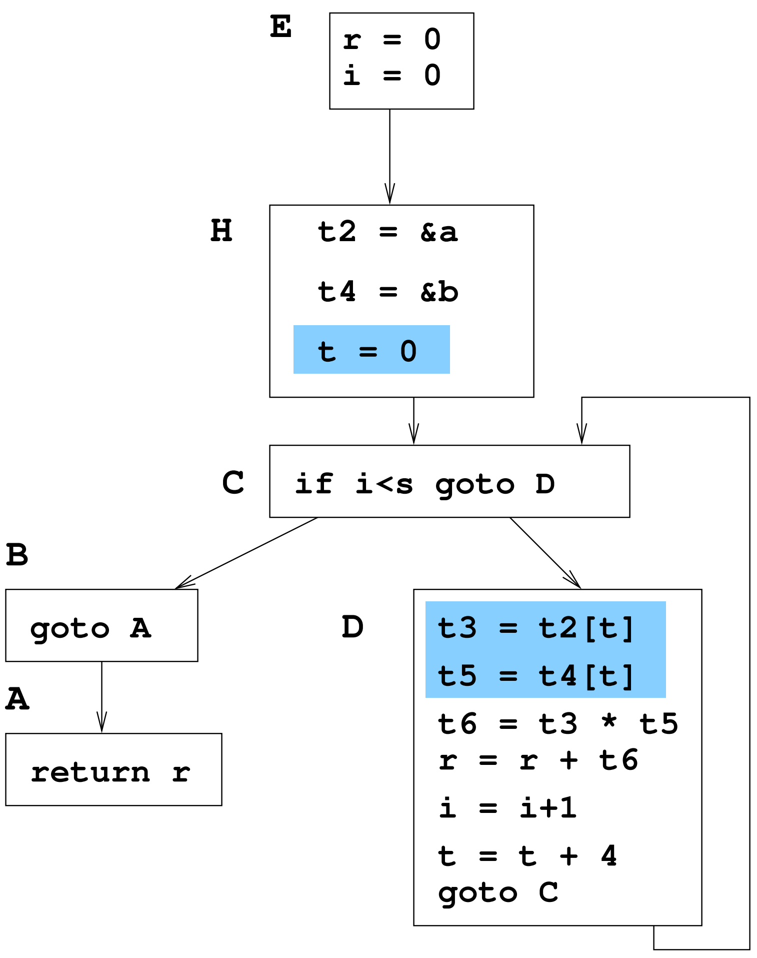


Figure 6.14: Flow graph of Figure 6.13 after copy propagation and constant folding

A more detailed description of this algorithm can be found in [ASU86].

Applying the above transformations to Example 20 results in the final graph depicted in Figure 6.15.

6.5 Aliasing: pointers and procedure calls

Two expressions are *aliases* of each other if they denote the same memory address. Both pointers and procedure calls introduce aliases. Aliases complicate data-flow analysis and code improvement, locally (see e.g. Section 6.2.3, page 100) as well as globally.

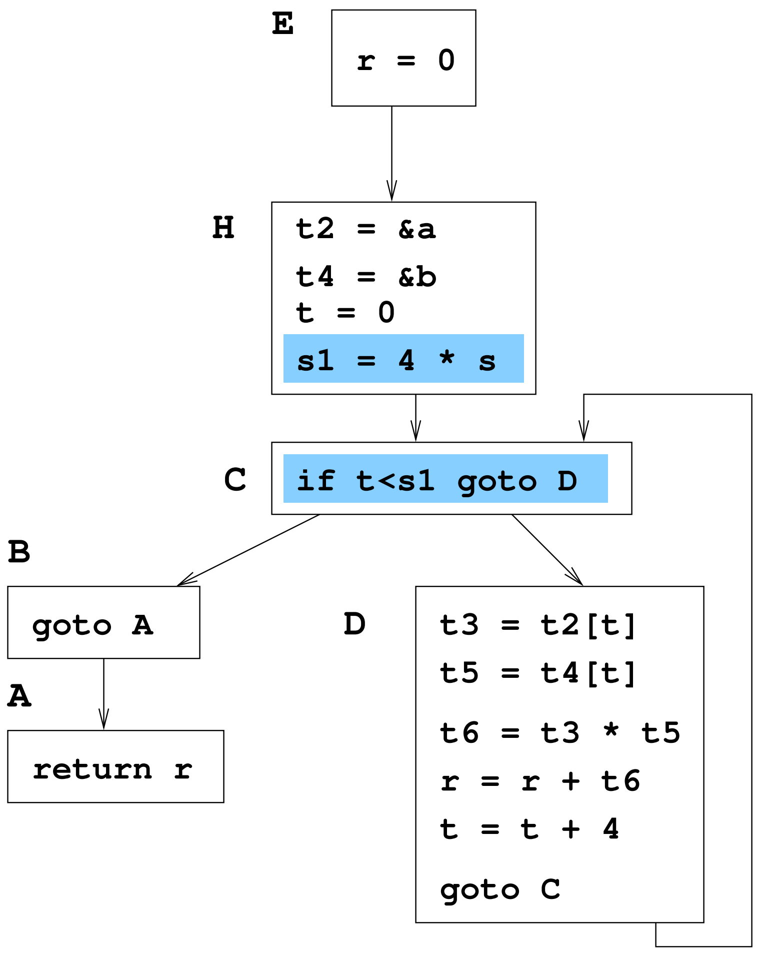


Figure 6.15: optimized flow graph for Example 20

If nothing is known about where a pointer points to, as is e.g. the case in a liberal language such as C, we must assume that any pointer assignment $*p = e$ can potentially define any variable. Likewise, dereferencing a pointer, as in $x = *p$ must be counted as a use of any variable. This obviously increases the number of live variables and reaching definition, while decreasing the number of available expressions. The net result is fewer code improvement opportunities.

A similar reasoning holds for procedure calls: if nothing is known about which variables the procedure may change, we must assume that it potentially defines all variables.

If, e.g. because of the rules of the source language, we know more about the possible usage of pointers and/or procedure calls, we may improve matters.

6.5.1 Pointers

We deal with pointers first. For each basic block B, we define two sets $IN_{ptr}(B)$ and $OUT_{ptr}(B)$ containing pairs (p, a) where p is a pointer variable and a is another variable. Intuitively, $(p, a) \in IN_{ptr}(B)$ means that, on entry to B, p may point to a (note that there may be several a such that $(p, a) \in IN_{ptr}(B)$), while $OUT_{ptr}(B)$ encodes where pointers may point upon exit of B.

The knowledge about pointer rules is encoded in a set of functions $trans_s$, one for each type of instruction s, which transforms a set of pairs (p, a) into another such set which is such that, if p may point to any a such that $(p, a) \in S$, then, after s, p may point to any b such that $(p, b) \in trans_s(S)$. $trans_s$ can be extended to $trans_B$ for any basic block B by composing , in order, the functions corresponding to the instructions of B. $IN_{ptr}(B)$ and $OUT_{ptr}(B)$ then satisfy the data-flow equations.

$$\begin{array}{rcl} OUT_{ptr}(B) & = & trans_B(IN_{ptr}(B)) \\ IN(B) & = & \cup_{C<B} OUT_{ptr}(B) \end{array}$$

Note that these equations are isomorphic to the ones used to compute reaching definitions (Section 6.3.1, page 103, and hence one can use Algorithm 10 (page 104) to compute the smallest fixpoint.

$IN_{ptr}(B)$ can then be used to derive a precise set of variables that may be defined by an instruction such as $*p = e$. Similarly, the set of variables that may be used by an instruction such as $a = *p$ can be derived from $IN_{ptr}(B)$, and the data-flow algorithms can be suitably refined to take this information into account (note that, e.g. for use-definition chains, unless p can point to only one variable, we must not consider $*p = a$ as a definition that may hide earlier ones).

6.5.2 Procedures

If we have knowledge about procedures, we can apply a similar reasoning as for pointers in order to better estimate the effect of `CALL` instructions. In this section, we will use $p(\&x, y)$ to denote a procedure calling sequence in three address code where x is passed by reference while y is not.

As an example, we consider the case of C where procedures may only modify local and global variables. Note that procedures may also pass global variables as reference parameters[9]. Clearly, passing a variable x for a reference formal

[9] In C, reference parameters are simulated by passing a pointer by value.

parameter y makes x and y aliases, denoted $x \equiv y$ (we assume that the names of all local variables, including formal parameters, in all procedures and the names of global variables are all distinct).

The following algorithm computes the set of possible aliases amongst all variables.

Algorithm 24 [Computing aliases]
We show only reference formal parameters for the procedures.

1. *For each procedure* $p(\&x_1, \ldots, \&x_n)$ *and each call* $p(\&y_1, \ldots, \&y_n)$*, add* $x_1 \equiv y_1, \ldots, x_n \equiv y_n$ *to the set of aliases.*
2. *Take the reflexive and transitive closure of the relation* $\equiv$ *resulting from step 1.*

□

Now we can determine the set $change(p)$, where p is a procedure, of global variables and formal parameters[10] that may possibly be changed by a call to p.

Define

- $def(p)$ as the set of formal parameters and globals of p that have definitions in the flow graph of p.
- A_p as the set of global variables or formal parameters a of p such that p contains a call $q(\ldots, \&a, \ldots)$ and the corresponding formal parameter is in $change(q)$.
- G_p as the set of global variables in $change(q)$ where p calls q.

Then

$$change(p) = def(p) \cup A_p \cup G_p$$

and this equation can be solved by a smallest fixpoint computation, starting with $change(p) = def(p)$ as an underestimate.

$change(p)$ can then be used as an estimate of possible definitions of variables, e.g. when computing available expressions (see also Section 10.8 in [ASU86]). A similar analysis can be made for the possible uses represented by a procedure call (for local variables this is easy: they are at most the parameters, for globals, one would need to analyze the call graph of the called procedure).

[10]Note that formal parameters of p are also local variables of p.

Chapter 7

Code generation

Code generation concerns the generation of machine code from (relatively) machine-independent intermediate code.

Actually, there are a few possibilities as to exactly what kind of "machine code" is generated:

- One could generate so-called *absolute* machine code in which addresses are fixed ("absolute") as much as possible. This strategy predates multiprocessing operating systems and was of little use in modern operating systems. It may however be relevant again in the context of so-called "just-in-time" compilers.
- It is also feasible to generated *relocatable* code where most addresses are relative to some unknown base addresses. The code then has to be processed by a linker-loader before it can be executed. Since this option duplicates the work done by an assembler, we prefer the next option.
- We generate *assembler source code* where addresses are symbolic and leave it to a separate program to translate further to relocatable code.

Code generation deals mainly with the following issues:

- *Run-time storage management* is concerned with the mapping of names to addresses. We will look mainly at stack-based allocation of local variables into so-called *activation records*.
- *Instruction selection*. Depending on the characteristics of the hardware and the context, the same instruction may be translated in several ways, each with different costs in time and/or space.

- Closely related to the previous concern is the issue of *register allocation*. Many processors have very efficient instructions that take their operands from a limited set of registers. The aim is to allocate registers to variables in order to optimize global performance.

7.1 Run-time storage management

7.1.1 Global data

The global memory layout for a traditional language such as C is shown in Figure 7.1. Of course, in modern operating systems, these areas may be in different memory segments, with different permissions, e.g. the code segment may be read-only and shared.

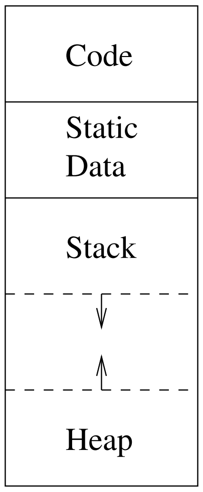


Figure 7.1: Memory layout in C

For static global variables, there is usually no problem: we simply use assembler pseudo-operations like

```
GLOBAL  BYTE    4,0     ; allocate & initialize 4 byte quantity GLOBAL
```

For dynamic global variables (e.g. the **new** operator in C++), memory is (de)allocated at run-time from a so-called *heap*. Efficient algorithms for heap management, including e.g. garbage collection, are outside the scope of this text.

7.1.2 Stack-based local data

If we are dealing with a language such as C, we also have to provide storage for *local variables*. If the language supports recursion, we are naturally lead to a stack-based organization where, upon a function/procedure call, a so-called *activation record* is pushed onto the stack. Upon return from the call, the activation record is popped.

A possible layout for an activation record is shown in Figure 7.2.

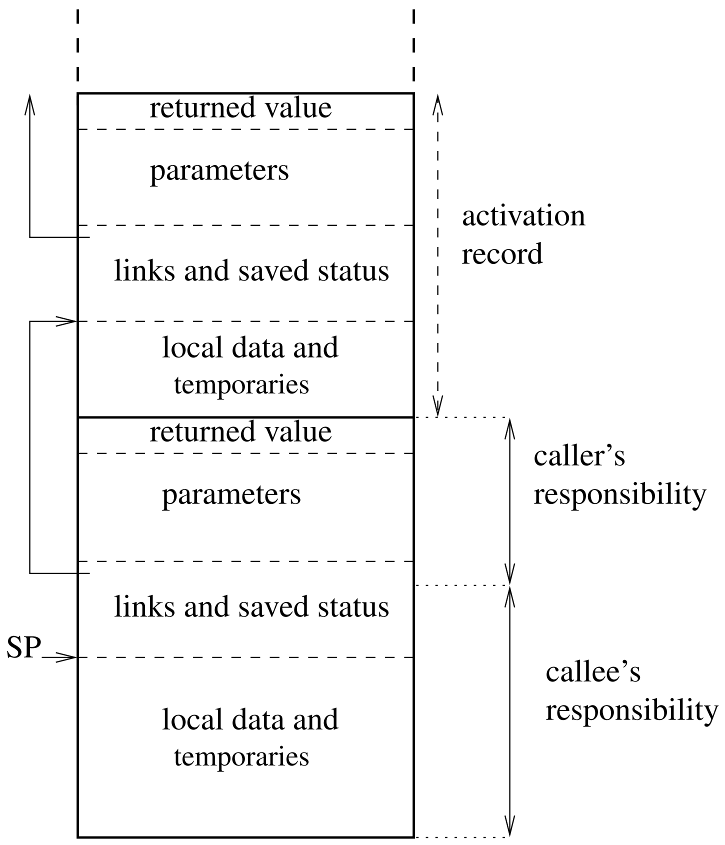


Figure 7.2: Activation records

Such a record contains, in growing stack order,

- Room for the value to be returned by the procedure (if any).
- The actual parameters.
- Links and saved status. The first item here is the stack pointer to be restored upon exit of the procedure. The next one might be the return address, i.e. the

address to which control should be transferred upon exit of the procedure. There may be additional hardware status information that is save here for restoration upon exit of the call.

- Room for local and temporary variables. By local variables, we mean the ones explicitly declared by the programmer. Temporary variables have been generated by the compiler during intermediate code generation and/or optimization.

The above organization is convenient because it allows an easy division of the work between caller and callee. The scenario for a procedure call is:

1. The caller evaluates the actual parameters and pushes them on the stack (note that the caller knows where is the top of the stack, since it knows *SP* and the size of its local and temporary data).
2. The caller pushes the value of *SP*, followed by the return address. It then puts *SP* to its new value and transfers control to the callee, e.g. by a `jump` instruction.
3. The callee saves further status information on the stack (using *SP*). The callee may further update *SP* to point just past the saved information (as in the figure).
4. The callee initializes its local data and goes about its further business.

A possible return (from a call) sequence is:

1. The callee places the return value is the appropriate place (note that it knows how many and which parameters it received, so the return value will be stored at $SP[-c]$ for some known c).
2. The callee restores the saved information, including the *SP* and jumps to the saved return address.
3. The caller has access to the returned value (right at the top of the stack), which it may copy into its local data.

It should be clear that, at run time, a register will be used to store *SP*. Access to local data will then be achieved using indirect addressing through the contents of *SP* and some constant offset which is known at compile time. This offset can easily be computed and stored in the symbol table. An interesting aspect is that local variables or temporaries may share a slot in the activation record provided they are never live at the same time. This occurs, e.g. when compiling the following C code.

```
int f(int x)
{
int a, b; /* slot 0, 1 */

...

if (x>1) {
        int y; /* slot 2 */

        ...
        }
else {
        int z; /* slot 2 */
        }
}
```

In the example, y and z will never be alive at the same time since they are in different (non-nested) scopes. Therefore, both can share slot 2.

Note also that the organization above leaves the possibility for a procedure to have local data whose size is only known at run-time. E.g. variable-sized arrays can be allocated on top of the stack, with a pointer to them reserved in the fixed-size local data area.

7.2 Instruction selection

Even abstracting from the register allocation program (Section 7.3, page 136), there may be several possibilities and constraints involved in the translation of an intermediate code instruction.

As a simple example, consider the intermediate code instruction

```
a = a + 1
```

This may be translated as (note the use of indirect addressing to access local names):

```
load [sp]+a,r0  ; copy a to register 0
add #1
store r0,[sp]+a ; copy register 0 back to a
```

but if the processor supports an *inc* (increment) instruction, it could also be translated as

```
inc [sp]+a      ; increment a
```

Some operations may only be possible with certain registers. E.g. on the IBM/370 architecture, multiplication needs to performed as shown in Figure 7.3, i.e. one operand must be available in the odd register of an even-odd register pair which will be filled with the result.

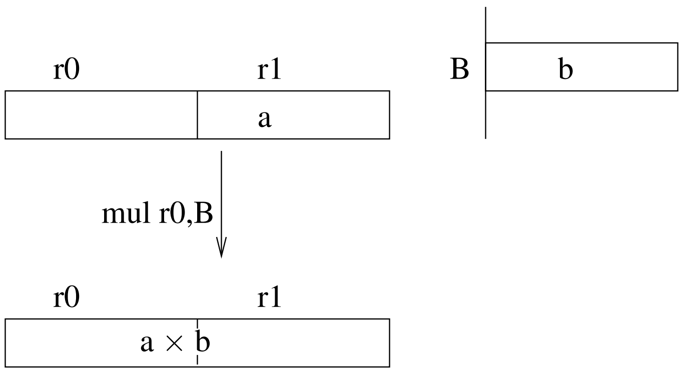


Figure 7.3: Multiplication on IBM/370

Addressing modes also influence the cost of an instruction. As an example, suppose the following table gives addressing modes and (relative) costs.

move r0,r1	1	register-to-register
move [r0],[r1]	1	register-to-register (indirect)
move r5,x	2	register-to-memory
move #1,r3	2	constant-to-register
move [r0]+4,[r1]+5	3	indexed-to-indexed

We can then consider the following alternative code sequences for $a = b + c$ (assuming they are all global):

- We can use a scratch register (total cost 6):

```
move b,r0
add c,r0        ; result in r0
move r0,a
```

- We can perform addition in "memory" (total cost 6):

```
move b,a
add c,a
```

- If the *values* of b and c are in registers $r1$ and $r2$, we can reduce the cost to 3, provided that b is not live after the instruction:

```
add r2,r1
move r1,a
```

As the example illustrates, register allocation can have a dramatic effect on performance.

7.3 Register allocation

We describe how register allocation can be reduced to a graph coloring problem.

The idea is that, for a given flow graph (procedure), every variable corresponds to a *symbolic register* [1]. The problem then is to assign each of a limited number of real registers to symbolic ones without introducing errors.

Clearly, we cannot use the same real register for symbolic registers (variables) r_1 and r_2 if e.g. r_1 is defined at a time when r_2 is live.

Definition 30 *The* ***register inference graph*** *G corresponding to a flow graph is an undirected graph where the nodes are the symbolic registers and there is an edge between r_1 and r_2 is r_1 is defined at a point where r_2 is live.*

If there are k real registers available, every symbolic register can be mapped to a real one without causing errors if G can be colored using k colors (no connected nodes may have the same color). Graph coloring with k colors is known to be NP-complete if $k > 2$ but the following heuristic procedure gives good results in practice.

Algorithm 25 [Register allocation using graph coloring]
Let G be the graph to be colored.

1. *While there is a node n with fewer than k connected edges, remove the n (and remember it).*
   The reason for this is that n can always receive a color, independent of the color chosen for its neighbors.

2. *If we are left with an empty graph, G was colorable and we can assign colors by coloring the nodes in the reverse order (of removal).*

[1]The algorithm can be adapted to hardware peculiarities where, e.g., a special register needs to be used for certain instructions. In this case, we associate a symbolic register with these instructions; the register is defined at the time the instruction is used.

3. *If we are left with a nonempty graph, it is not k-colorable since it has a node with at least k neighbors. We then choose a node n to be* spilled, *remove the node from the graph and start again with step 1. Spilling a node means that the corresponding variable will be loaded (into a register) on each use and stored back on each definition.*

Several heuristics have been proposed to choose a spilled node:

- *Choose a node with a maximal number of attached edges, maximizing the chance of ending up with a k-colorable graph.*
- *Estimate the cost of spilling a node, and choose the one with the lowest cost. A simple measure would count the number of extra load/store instructions, taking care to multiply with a suitable factor any such instructions that appear in a loop*[2].

□

7.4 Peephole optimization

Some machine code optimizations are very easy to perform by scanning the code (using a sliding window of instructions, hence the name) for certain patterns that can be replaced by improved patterns.

Often, it is the case that successful replacements induce opportunities for further replacements; therefore scanning is usually repeated until no further matches are found.

Below we list some example patterns and replacements.

[2]One could also have a higher cost factor for instructions in an inner loop than in an encompassing loop.

Pattern	Replacement	Description
move x,y move y,x	move x,y	redundant load, store
jump(con) z jump y z: (seq1) y: (seq2)	jump(notcon) y z: (seq1) y: (seq2)	jumps over jumps
jump(con) z .. z: jump y	jump(con) y .. z: jump y	jumps to jumps
mul #1,x add #0,x	(delete) (delete)	algebraic simplification
mul #2,x	add x,x	machine specifics
mul #2,x	shiftleft x,1	machine specifics
add #1,x	inc x	machine specifics

Appendix A

A Short Introduction to x86 Assembler Programming under Linux

A.1 Architecture

Memory is byte addressable. There are several general purpose registers: `%eax`, `%ebx`, `%ecx`, `%edx`, `%edi` and `%esi`.

The following registers have a special purpose:

- The "base pointer" `%ebp` is used to point to a stack frame for easy accessing to local variables and parameters, see Section A.4.
- The "stack pointer" `%esp` always points to the top of the stack which grows downwards. E.g. if the contents of `%esp` is 1000, then the top two elements of the stack are at addresses 1000 and 999.
- The instruction counter `%eip` points to the next instruction (in memory) to be executed.
- The register `%eflags` contains the condition codes which are set by several instructions and can be tested/used by others. Table A.1 shows some commonly used flags.

code	name	description
O	overflow	1 if destination cannot hold result
S	sign	sign of last result
Z	zero	1 if last result was 0
P	parity	1 if low byte of result has even number of 1 bits
C	carry	1 if overflow bit

Table A.1: Some x86 Flags

A.2 Instructions

A.2.1 Operands

The operands for instructions may be of the following types:

- *Immediate* (I), i.e. constant integer data. For the gnu assembler (*as*), such data is prefixed with '$', e.g. `$100` (decimal) or `$0x80` (hexadecimal). Such data are encoded using 1, 2 or 4 bytes.
- *Register* (R), referring to a (4 byte) register, e.g. `%eax`.
- *Memory* (M), referring to 4 consecutive bytes of memory.

As an example, the `movl` moves a word (4 bytes) from the first to the second operand. The possibilities are illustrated in the following code fragment.

```
movl $100, %eax /* IR store 100 in register eax */
movl $100, (%eax) /* IM store 100 in memory pointed to by eax */
movl %eax, %ebx /* RR store content of eax in ebx */
movl %ebx, (%eax) /* RM store content of ebx in memory pointed to by eax */
movl (%eax), %ebx /* MR store content of memory pointed to by eax in ebx */
```

Note that single-instruction MM (memory to memory) transfers are impossible. In fact, no instruction can have two M operands.

A.2.2 Addressing Modes

The following *addressing modes* are available for refering to a memory location:

- *Normal* mode, e.g. `(%eax)` refers to the location with address the contents of the `%eax` register. Thus, using $*r$ to denote the contents of register r,

$$(r) \rightarrow [*r]$$

- In *displacement* mode, an offset is specified together with a register. E.g. `-4(%esp)` refers to the (32bit) word just "under" the top of the stack while `8(%esp)` refers to the location at `(%esp)+8`. Thus:

$$d(r) \rightarrow [*r + d]$$

- *Indexed* addressing mode involves a *base* (any) and an *index* (any except `%esp`) register as well as a constant *displacement* (1, 2 or 4) and a *scale* which must be 1,2 4 or 8:

$$d(r_b, r_i, s) \rightarrow [*r_b + s \times *r_i + d]$$

  E.g. `movl 8(%ebx,%edi,4), %eax` would move the word located at address `(%ebx)+4*(%edi)+8` to register `%eax`.

  One of the registers or the scale (default 1) or displacement (default 0) may be missing. E.g. *as* accepts all of the following:

```
        movl $100, (%eax, %ebx)
        movl $100, 4(%eax, %ebx)
        movl $100, (%eax, %ebx,4)
        movl $100, (, %ebx,4)
```

The tables in the following sections are adapted from [Bar03]. Also, o_1 and o_2 are sometimes used to refer to the first and second operand of the instruction.

A.2.3 Moving Data

Data Transfer Instructions			
name	**operands**	**flags set**	**description**
`movl`	IMR,IMR	OSZC	move word from o_1 to o_2
`movb`	IMR,IMR	OSZC	move byte from o_1 to o_2
`leal`	M,IMR	OSZC	store **address** of M in o_2
`popl`	MR	OSZC	pop word from stack into operand
`pushl`	IMR	OSZC	push word on stack
`pushfl`			push `eflags` register on stack
`xchgl`	MR, MR	OSZC	exchange the words in o_1 and o_2

Table A.2: Data Transfer Instructions

Integer Arithmetic Instructions			
name	**operands**	**flags set**	**description**
`adcl`	IMR,MR	OSZPC	add with carry $o_2 \mathrel{+}= (o_1 + C)$
`addl`	IMR,MR	OSZPC	add $o_2 \mathrel{+}= o_1$
`cdq`		OSZPC	convert **`%eax`** to double word in **`%edx:%eax`**, see `idivl`
`cmpl`	IMR,MR	OSZPC	set flags according to $o_2 - o_1$
`decl`	MR	OSZPC	decrement (-1) word
`decb`	MR	OSZPC	decrement (-1) byte
`divl`	MR	OSZPC	unsigned divide **`%edx:%eax`** by o_1, quotient in **`%eax`**, remainder in **`%edx`**
`idivl`	MR	OSZPC	signed division, see `divl`
`imull`	IMR, R	OSZPC	signed multiplication: $o_2 \times= o_1$
`imull`	IMR	OSZPC	signed multiplication: **`%edx:%eax`** = **`%eax`** $\times o_1$
`incl`	MR	OSZPC	increment $(+1)$ word
`incb`	MR	OSZPC	increment $(+1)$ byte
`negl`	MR	OSZPC	negate (2's complement inversion) operand
`sbbl`	IMR, MR	OSZPC	$o_2 \mathrel{-}= o_1$, with borrowing (see adcl)
`subl`	IMR, MR	OSZPC	$o_2 \mathrel{-}= o_1$, no borrowing

Table A.3: Integer Arithmetic Instructions

A.2.4 Integer Arithmetic

See Table A.3 on page 142.

A.2.5 Logical Operations

See Table A.5 on page 145.

A.2.6 Control Flow Instructions

See Table A.6 on page 146.

A.3 Assembler Directives

The gnu assembler *as* supports a number of directives:

- A *section* switch directive is of the form

  ```
  .section name
  ```

  where *name* is usually *.text*, for code, *.data*, for data embedded in the program, or *.bss* for uninitialized data. It indicates which section the following directives and instructions belong to.

- An *equ* directive is of the form

  ```
  .equ label, value
  ```

  and assigns the *value* to *label*.

- Data are defined using an *.ascii*, *.byte* or *.long* directive, as in the following example

```
astring: /* label */
.ascii "Hello world\0"
bytes:
.byte 0, 1, 2, 3, 4, 5
words:
.long 1024, 2048, 4096
```

- Data in the *.bss* section can be reserved using the *.lcomm* directive which has the form

  ```
  .lcomm label, size_in_bytes
  ```

- The *.globl* directive exports a label so that it is accessible from other object files. The *.type* directive tells the linker that a label is a function, as in the following example.

```
.globl ackerman
.type ackerman, @function
ackerman:
  ...
```

  The *.include* directive inserts another file at that point in the text. It has the form

  ```
  .include "pathname"
  ```

- Finally, the *.rept* and *.endr* can be used to repeat everything between the directives a number of times, as in the following example. which reserves 100 data bytes that are initialized to 0.

```
.rept 100
.byte 0
.endr
```

A.4 Calling a function

The C calling convention is illustrated in the following example. Note how `%ebp` is used as a frame pointer.

```
.globl print
.type myfun, @function
# myfun(int a, int b) { int c = 0; int d = 2; ... }
myfun:
        # stack: b a [return-addr]
        pushl %ebp # save base(frame) pointer on stack
        # stack: b a [return-addr] [saved-bp]
        movl %esp, %ebp # base pointer is stack pointer
        .equ a, 8
        .equ b, 4
        # a(%ebp) => a; b(%ebp) => b
        pushl $0 # initialize c
        pushl $2 # initialize d
        # stack: b a [return-addr] [saved-bp] 0 2
        .equ c, -4
        .equ d, -8
        # c(%ebp) => c; d(%ebp) => d

        /* body of function, the return value is stored in %eax */

        movl %ebp, %esp # restore stack pointer
        # stack: b a [returnaddress] [savedebp]
        popl %ebp # restore saved base (frame) pointer
        /* stack: b a [return-addr]
        ret
```

A.5 System calls

To obtain a service from Linux, it suffices to specify the service in register **`eax`**, set up the input parameters in various registers (see Table A.4 below) and then generate an interrupt using the instruction **`int`** **`int`** `$0x80`. The return value and/or error code will be stored in register **`eax`**.

More information can be found in the man pages corresponding to the “function” column in Table A.4 which lists a few I/O related system calls.

%eax	**name**	**%ebx**	**%ecx**	**%edx**	**function**
1	exit	`code`			`exit(code)`
3	read	`fd`	`buf`	`count`	`read(fd, buf, count)`
4	write	`fd`	`buf`	`count`	`write(fd, buf, count)`
5	open	`path`	`flags`	`mode`	`open(path, flags, mode)`
6	close	`fd`			`close(fd)`
19	lseek	`fd`	`offset`	`whence`	`lseek(fd, offset, whence)`

Table A.4: Linux System Calls

Logical Operations			
name	**operands**	**flags set**	**description**
`andl`	IMR,MR	OSZPC	o_2 &= o_1 $C = O = 0$
`orl`	IMR,MR	OSZPC	$o_2 \vert= o_1$ $C = O = 0$
`notl`	MR		$o_1\ = o_1$
`rcll`	I%**cl**,RM	OC	rotate $o_2 + C$ by o_1 bits left
`rcrl`	I%**cl**,RM	OC	rotate $o_2 + C$ by o_1 bits right
`roll`	I%**cl**,RM	OC	rotate o_2 by o_1 bits left
`rorl`	I%**cl**,RM	OC	rotate o_2 by o_1 bits right
`sall`	I%**cl**,RM	C	arithmetic shift o_1 bits left; sign bit to C, 0 added as lsb
`sarl`	I%**cl**,RM	C	arithmetic shift o_1 bits right; lsb bit to C, S to msb
`shll`	I%**cl**,RM	C	shift o_1 bits left; msb bit to C
`shrl`	I%**cl**,RM	C	shift o_1 bits right; lsb bit to C
`testl`	IRM, RM	OSZPC	set flags from o_1&o_2
`xorl`	IRM, RM	OSZPC	$o_2 = o_2$ **`xor`** o_1, $O = C = 0$

Table A.5: Logical Operations

Control Flow Instructions			
name	**operands**	**flags set**	**description**
`call`	address	OSZC	push `%eip`, jump to address
`call`	*R	OSZC	call function a (R), e.g. **`call *%eax`**
`int`	I	OSZC	cause interrupt number I, $I = 0x080$ for Linux kernel call
`jmp`	address	OSZC	unconditional jump
`jmp`	*R	OSZC	goto (R), e.g. **`jmp *%eax`**
`ret`		OSZC	return: `popl %eip;` *`jmp *%eip`*
`jecxz`	address	OSZC	jump if `%`**`ecx`** is 0
`j[n][spoc]`	address	OSZC	jump if [$SPOC$] is (not) set
`j[n][gl][e]`	address	OSZC	signed comparison: jump if (not) $>$ (g), $<$ (l), $\geq$ (ge) or $\leq$ (le)
`j[n]z`	address	OSZC	jump if (not) 0
`j[n]e`	address	OSZC	jump if (not) equal
`j[n][ab][e]`	address	OSZC	unsigned comparison jump if (not) $>$ (a), $<$ (b), $\geq$ (ae) or $\leq$ (be)

Table A.6: Control Flow Instructions

A.6 Example

Example 21 The listing below shows a function that prints its single parameter (an integer) on stdout, followed by `'\n'`.

```
# print an integer
.section .text
.globl print_int
.type print_int, @function
# print_it(int value) -- print value followed by \n to stdout.
print_int:
        pushl %ebp
        movl %esp, %ebp
        .equ value, 8 # parameter offset
        # initialize local variables:
        .equ base, -4
        pushl $10 # base = 10
        .equ bufsize, -8
        pushl $1 # bufsize = 1 ('\n')
        .equ negative, -12
        pushl $0 # negative = 0
        # stack: .. value return-addr saved-ebp base bufsize
        pushl $10 # push newline to be printed last
        movl value(%ebp), %eax
        jge .L1 # if value >= 0
        # value < 0: remember
        movl $1, negative(%ebp)
        negl %eax # value = -value
.L1:
        cdq # setup %edx:%eax for division
        # aex = value/base, edx = value % base
        divl base(%ebp)
        # push remainder digit
        pushl %edx
        addl $48, (%esp)
        incl bufsize(%ebp) # ++bufsize
        cmpl $0, %eax
        jne .L1 # loop if value != 0
        # put sign if negative
        cmpl $0, negative(%ebp)
        je .L2
        pushl $45 # '-'
        incl bufsize(%ebp)
.L2:
        # write(2): eax = 4, ebx = fd, ecx = buffer start, edx = buffer size
        movl $4, %eax # code for write syscall
        movl $1, %ebx # fd stdout = 1
        movl %esp, %ecx # buffer start = top of stack
        movl $4, %edx # bufsize * 4 bytes
        imul bufsize(%ebp), %edx
        int $0x80 # syscall
        movl %ebp, %esp
        popl %ebp # restore saved frame pointer
        ret
```

Example 22 The function from Example 21 is used in the following program.

```
# print an integer
.section .text
.globl _start
_start:
        call main
        jmp exit
.include "print_int.s"
.globl main
.type main, @function
main:
        pushl %ebp # save frame pointer
        movl %esp, %ebp # set frame pointer
        pushl $-100 # number to print
        call print_int
        movl %ebp, %esp
        popl %ebp
        ret
.type exit, @function
exit:
        movl $0, %ebx
        movl $1, %eax
        int $0x80
```

It can be assembled and linked using the following command which translates *tst.s* into an object file *tst.o* which is then linked to an executable file *tst*

```
        as tst.s -o tst.o && ld tst.o -o tst
```

Appendix B

Mc: the Micro-x86 Compiler

B.1 Lexical analyzer

```
%{
/* $Id: lex.l,v 1.1 2008/07/09 13:06:42 dvermeir Exp $
*
* Lexical analyzer for the toy language ``Micro''
*/
#include <string.h> /* for strcmp, strdup & friends */
#include <stdlib.h> /* for atoi() */

#include "micro.tab.h" /* token type definitions from .y file */
#include "symbol.h" /* symbol table management */

extern int lineno;  /* defined in micro.y */

void
lex_init() {
  /* Initialize data structures etc. for scanner */
  symbol_insert("declare",DECLARE); /*Insert keywords in symbol table */
  symbol_insert("read",READ); /*Insert keywords in symbol table */
  symbol_insert("write",WRITE);
}

/*
* The macro below will be called automatically when the generated scanner
* initializes itself.
*/
#define YY_USER_INIT lex_init();

%}

```

```
alpha   [A-Za-z]
digit   [0-9]
alphanum  [A-Za-z0-9]

%%
[ \t]   break;  /* ignore white space */
"\n"    ++lineno;

{alpha}{alphanum}*      {
                        yylval.idx = symbol_find(yytext);

                        if (yylval.idx<0) { /* new symbol: insert it */
                          yylval.idx =symbol_insert(yytext, NAME);
                          return NAME;
                        }
                        else
                          return symbol_type(yylval.idx);
                        }

{digit}+                {
                        yylval.value = atoi(yytext);
                        return NUMBER;
                        }

"("                     return LPAREN;
")"                     return RPAREN;
"{"                     return LBRACE;
"}"                     return RBRACE;
"="                     return ASSIGN;
";"                     return SEMICOLON;
"+"                     return PLUS;
"-"                     return MINUS;

.                       {
                        fprintf(stderr,
                                "Illegal character \'%s\' on line #%d\n",
                                yytext, lineno);
                        exit(1);
                        }

%%

int
yywrap() {
  return 1; /* tell scanner no further files need to be processed */
}
```

B.2 Symbol table management

```c
/* $Id: symbol.c,v 1.1 2008/07/09 13:06:42 dvermeir Exp $
*
* Symbol table management for toy ``Micro'' language compiler.
* This is a trivial example: the only information kept
* is the name, the token type: READ, WRITE or NAME and, for NAMEs
* whether they have been declared in the JVM code.
*/
#include <stdio.h> /* for (f)printf(), std{out,int} */
#include <stdlib.h> /* for exit */
#include <string.h> /* for strcmp, strdup & friends */
#include "micro.tab.h" /* token type definitions from .y file */

#include "symbol.h"

typedef struct {
  char *name;
  int type;  /* READ, WRITE, or NAME */
  int declared; /* NAME only: 1 iff already declared in JVM code, 0 else */
} ENTRY;

#define  MAX_SYMBOLS 100
static ENTRY symbol_table[MAX_SYMBOLS]; /* not visible from outside */
static int n_symbol_table = 0; /* number of entries in symbol table */

int
symbol_find(char* name) {
  /* Find index of symbol table entry, -1 if not found */
  int i;

  for (i=0; (i<n_symbol_table); ++i)
    if (strcmp(symbol_table[i].name, name)==0)
      return i;
  return -1;
}

int
symbol_insert(char* name,int type) {
  /* Insert new symbol with a given type into symbol table,
   * returns index new value */
  if (n_symbol_table>=MAX_SYMBOLS) {
    fprintf(stderr, "Symbol table overflow (%d) entries\n", n_symbol_table);
    exit(1);
  }
  symbol_table[n_symbol_table].name = strdup(name);
  symbol_table[n_symbol_table].type = type;
  symbol_table[n_symbol_table].declared = 0;
  return n_symbol_table++;
}

int
symbol_type(int i) {
  /* Return type of symbol at position i in table. */
  /* ASSERT ((0<=i)&&(i<n_symbol_table)) */
  return symbol_table[i].type;
}

void
symbol_declare(int i) {
  /* Mark a symbol in the table as declared */
  /* ASSERT ((0<=i)&&(i<n_symbol_table)&&(symbol_table[i].type==NAME)) */
  symbol_table[i].declared = 1;
}

int
symbol_declared(int i) {
  /* Return declared property of symbol */
  /* ASSERT ((0<=i)&&(i<n_symbol_table)&&(symbol_table[i].type==NAME)) */
  return symbol_table[i].declared;
}

char*
symbol_name(int i) {
  /* Return name of symbol */
  /* ASSERT ((0<=i)&&(i<n_symbol_table)) */
  return symbol_table[i].name;
}
```

B.3 Parser

```
%{
/* $Id: micro.y,v 1.9 2008/07/11 18:56:43 dvermeir Exp $
*
* Parser specification for Micro
*/
#include <stdio.h> /* for (f)printf() */
#include <stdlib.h> /* for exit() */

#include "symbol.h"

int  lineno = 1; /* number of current source line */
extern int yylex(); /* lexical analyzer generated from lex.l */
extern char *yytext; /* last token, defined in lex.l  */

void
yyerror(char *s) {
  fprintf(stderr, "Syntax error on line #%d: %s\n", lineno, s);
  fprintf(stderr, "Last token was \"%s\"\n", yytext);
  exit(1);
}

#define PROLOGUE "\
.section .text\n\
.globl _start\n\
\n\
_start:\n\
        call main \n\
        jmp exit\n\
.include \"../x86asm/print_int.s\"\n\
.globl main\n\
.type main, @function\n\
main:\n\
        pushl %ebp /* save base(frame) pointer on stack */\n\
        movl %esp, %ebp /* base pointer is stack pointer */\n\
"

#define EPILOGUE "\
        movl %ebp, %esp\n\
        popl %ebp /* restore old frame pointer */\n\
        ret\n\
.type exit, @function\n\
exit:\n\
        movl $0, %ebx\n\
        movl $1, %eax\n\
        int $0x80\n\
"

%}

%union {
 int idx;
 int value;
 }

%token NAME
%token NUMBER
%token LPAREN
%token RPAREN
%token LBRACE
%token RBRACE
%token ASSIGN
%token SEMICOLON
%token PLUS
%token MINUS
```

```
%token DECLARE
%token WRITE
%token READ

%type <idx> NAME var
%type <value> NUMBER

%%
program         : LBRACE
                  { puts(".section .data\n"); } declaration_list
                  { puts(PROLOGUE); } statement_list
                  RBRACE { puts(EPILOGUE); }
                ;
declaration_list  : declaration SEMICOLON declaration_list
                | /* empty */
                ;

statement_list  : statement SEMICOLON statement_list
                | /* empty */
                ;

statement       : assignment
                | read_statement
                | write_statement
                ;

declaration     : DECLARE NAME
                  {
                     if (symbol_declared($2)) {
                       fprintf(stderr,
                               "Variable \"%s\" already declared (line %d)\n",
                               symbol_name($2), lineno);
                       exit(1);
                     }
                     else {
                       printf(".lcomm %s, 4\n", symbol_name($2));
                       symbol_declare($2);
                     }
                  }
                ;

assignment      : var ASSIGN expression
                  {
                    /* we assume that the expresion value is (%esp) */
                    printf("\tpopl %s\n", symbol_name($1));
                  }
                ;

read_statement  : READ var { /* not implemented */}
```

```
                  ;

write_statement : WRITE expression { puts("\tcall print_int\n"); }
                  ;

expression      : term
                | term PLUS term { puts("\tpopl %eax\n\taddl %eax, (%esp)\n"); }
                | term MINUS term { puts("\tpopl %eax\n\tsubl %eax, (%esp)\n"); }
                ;

term            : NUMBER { printf("\tpushl $%d\n", $1); }
                | var { printf("\tpushl %s\n", symbol_name($1)); }
                | LPAREN expression RPAREN
                ;

var             : NAME
                  {
                    if (!symbol_declared($1)) {
                      fprintf(stderr,
                              "Variable \"%s\" not declared (line %d)\n",
                              symbol_name($1), lineno);
                      exit(1);
                    }
                    $$ = $1;
                  }
                ;
%%

int
main(int argc,char *argv[]) {
  return yyparse();
}
```

B.4 Driver script

```
#!/bin/sh
#
#       $Id: microc.sh,v 1.3 2008/07/10 16:44:14 dvermeir Exp $
#
#       Usage:  microc basename.mi
#
#       e.g. "microc tst.mi"  to compile tst.mi, resulting in
#
#       tst.s   assembler source code
#       tst.o   object file
#       tst     executable file
#
# determine basename
```

```
base=`basename $1 .mi`
# this checks whether $1 has .mi suffix
[ ${base} = $1 ] && { echo "Usage: microc basename.mi"; exit 1; }
# make sure source file exists
[ -f "$1" ] || { echo "Cannot open \"$1\""; exit 1; }
# compile to assembly code
./micro <$1 >${base}.s || { echo "Errors in compilation of $1.mi"; exit 1; }
# assemble to object file: the --gdwarf2 option generates info for gdb
as --gdwarf2 ${base}.s -o ${base}.o  || { echo "Errors assembling $1.s"; exit 1; }
# link
ld ${base}.o -o ${base} || { echo "Errors linking $1.o"; exit 1; }
```

B.5 Makefile

```
#       $Id: Makefile,v 1.8 2008/07/10 16:44:14 dvermeir Exp $
#
CFLAGS=         -Wall -g
CC=             gcc
#
SOURCE=         microc.sh micro.y lex.l symbol.c symbol.h Makefile *.mi
#
all:            micro microc demo
micro:          micro.tab.o lex.yy.o symbol.o
                gcc $(CFLAGS) -o $@ $^

lex.yy.c:       lex.l micro.tab.h
                flex lex.l
#
#       Bison options:
#
#       -v      Generate micro.output showing states of LALR parser
#       -d      Generate micro.tab.h containing token type definitions
#
micro.tab.h\
micro.tab.c:    micro.y
                bison -v -d $^
##
demo:           microc micro demo.mi
                ./microc demo.mi
#
CFILES= $(filter %.c, $(SOURCE)) micro.tab.c lex.yy.c
HFILES= $(filter %.h, $(SOURCE)) micro.tab.h
include make.depend
make.depend:    $(CFILES) $(HFILES)
                gcc -M $(CFILES) >$@

clean:
                rm -f lex.yy.c micro.tab.[hc] *.o microc micro *.jasm *.class micr
```

```
#
tar:
		tar cvf micro.tar $(SOURCE)
```

B.6 Example

B.6.1 Source program

```
{
declare xyz;
xyz = (33+3)-35;
write xyz;
}
```

B.6.2 Assembly language program

```
.section .data

.lcomm xyz, 4
.section .text
.globl _start

_start:
        call main
        jmp exit
.include "../x86asm/print_int.s"
.globl main
.type main, @function
main:
        pushl %ebp /* save base(frame) pointer on stack */
        movl %esp, %ebp /* base pointer is stack pointer */

        pushl $33
        pushl $3
        popl %eax
        addl %eax, (%esp)

        pushl $35
        popl %eax
        subl %eax, (%esp)

        popl xyz
        pushl xyz
        call print_int

        movl %ebp, %esp
        popl %ebp /* restore old frame pointer */
        ret
.type exit, @function
exit:
        movl $0, %ebx
        movl $1, %eax
        int $0x80
```

Appendix C

Minic parser and type checker

C.1 Lexical analyzer

```
%{
/*      lex.l(1.6)      17:46:22        97/12/10
*
*       Lexical analyzer for the toy language ``minic''
*/
#include        <string.h>      /* for strcmp, strdup & friends */
#include        <stdio.h>       /* for strcmp, strdup & friends */
#include        <stdlib.h>      /* for atoi() */

#include        "symtab.h"      /* symbol table management */
#include        "types.h"       /* symbol table management */
#include        "minic.tab.h"   /* token type definitions from .y file */
#include        "names.h"       /* string pool management */

extern int      lineno;         /* defined in minic.y */
extern SYM_TAB  *scope;         /* defined in minic.y */

void
lex_init()
/*      Initialize data structures etc. for scanner */
{
scope   = symtab_open(0);       /* open topmost scope */
}

/*
*       The macro below will be called automatically when the generated scanner
*       initializes itself.
*/
#define YY_USER_INIT    lex_init();
```

```

%}

alpha                   [A-Za-z]
digit                   [0-9]
alphanum                [A-Za-z0-9]

%%
[ \t]                   break;          /* ignore white space */
"\n"                    ++lineno;


int                     return INT;     /* Keywords come before NAMEs */
if                      return IF;
else                    return ELSE;
return                  return RETURN;
float                   return FLOAT;
struct                  return STRUCT;

{alpha}{alphanum}*      {
                        yylval.name     = names_find_or_add(yytext);
                        return NAME;
                        }

{digit}+                {
                        yylval.value    = atoi(yytext);
                        return NUMBER;
                        }

"("                     return LPAR;
")"                     return RPAR;
"{"                     return LBRACE;
"}"                     return RBRACE;
"["                     return LBRACK;
"]"                     return RBRACK;
"=="                    return EQUAL;
"="                     return ASSIGN;
";"                     return SEMICOLON;
","                     return COMMA;
"."                     return DOT;
"+"                     return PLUS;
"-"                     return MINUS;
"*"                     return TIMES;
"/"                     return DIVIDE;

.                       {
                        fprintf(stderr,
                                "Illegal character \'%s\' on line #%d\n",
                                yytext,lineno);
                        exit(1);
                        }

%%

int
yywrap()
{
return 1; /* tell scanner no further files need to be processed */
}
```

C.2 String pool management

```
#ifndef NAMES_H
#define NAMES_H
/*      names.h(1.1)    09:25:08        97/12/08
*
*       String pool management.
*/
char*   names_insert(char*);
char*   names_find(char*);
char*   names_find_or_add(char*);

#endif
```

```
/*      names.c(1.1)    09:25:08        97/12/08
*
*       String pool management.
*/
#include        <stdio.h>       /* for fprintf and friends */
#include        <stdlib.h>      /* for malloc and friends */
#include        <assert.h>      /* for assert(invariant) */
#include        <string.h>      /* for strcmp(), strcpy() */

#include        "names.h"

#define INCREMENT_SIZE          1000

static  char*   buf             = 0;
static  int     buf_size        = 0;    /* total size of buffer */
static  size_t  buf_avail       = 0;    /* free bytes in buf */
static  char*   next_name       = 0;    /* address available for next string */

#define INVARIANT       assert(buf+buf_size==next_name+buf_avail); \
                        assert(buf_size>=0); \
                        assert(buf_avail>=0); \
                        assert(next_name>=buf);
static void
```

```
names_grow(size_t size)
{
size_t  next_offset     = next_name - buf;

{ INVARIANT }

buf     = realloc(buf,buf_size+size); /* like malloc() if buf==0 */
if (!buf) {
        fprintf(stderr,"Cannot expand name space (%d bytes)",buf_size+size);
        exit(1);
        }
buf_avail       += size;
buf_size        += size;
next_name       = buf+next_offset;

{ INVARIANT }
}

char*
names_insert(char* s)
{
char*   ps;
size_t  l       = strlen(s)+1; /* include trailing '\0' */

{ INVARIANT }

while (l>buf_avail)
        names_grow(INCREMENT_SIZE);

ps              = strcpy(next_name,s);

buf_avail       -= l;
next_name       += l;

{ INVARIANT }

return ps;
}

char*
names_find(char *s)
{
char    *pc;

for (pc=buf;(pc!=next_name);pc += strlen(pc)+1)
        if (!strcmp(pc,s))
                return pc;
return 0;
}
```

```

char*
names_find_or_add(char *s)
{
char    *pc = names_find(s);

if (!pc)
        pc = names_insert(s);
return pc;
}
```

C.3 Symbol table management

```
#ifndef SYMBOL_H
#define SYMBOL_H
/*      symtab.h(1.5)   10:07:41        97/12/10
*
*       Symbol table management
*/

typedef struct syminfo {
        char                    *name;
        struct type_info        *type;
        } SYM_INFO;


typedef struct symcell {
        SYM_INFO        *info;
        struct symcell  *next;
        } SYM_LIST;

typedef struct symtab {
        struct symtab   *parent;
        SYM_INFO        *function; /* enclosing this scope */
        SYM_LIST        *list;
        } SYM_TAB;

SYM_TAB*        symtab_open(SYM_TAB* enclosing_scope);
SYM_INFO*       symtab_find(SYM_TAB*,char*);
SYM_INFO*       symtab_insert(SYM_TAB*,char*,struct type_info*);
int             symtab_list_equal(SYM_LIST*,SYM_LIST*);
SYM_LIST*       symtab_list_insert(SYM_LIST*,SYM_INFO*);
SYM_INFO*       symtab_list_find(SYM_LIST*,char*);
void            symtab_list_release(SYM_LIST*);
SYM_INFO*       symtab_info_new(char*,struct type_info*);

void            symtab_print(FILE*,SYM_TAB*);
void            symtab_info_print(FILE*,SYM_INFO*);
void            symtab_list_print(FILE*,SYM_LIST*,char* separator);
#endif
```

```
/*      symtab.c(1.5)   10:07:40        97/12/10
*
*       Symbol table management
*/
#include        <stdlib.h>      /* for malloc() and friends */
#include        <stdio.h>       /* for fprintf() and friends */

#include        "util.h"
#include        "symtab.h"
#include        "types.h"

SYM_TAB*
symtab_open(SYM_TAB * enclosing_scope)
{
SYM_TAB *st     = fmalloc(sizeof(SYM_TAB));

st->parent      = enclosing_scope;
st->list        = 0;
if (enclosing_scope)
        st->function    = enclosing_scope->function;
return st;
}

SYM_INFO*
symtab_find(SYM_TAB *st,char *name)
{
SYM_INFO        *i;

for ( ; (st); st = st->parent)
        if ((i=symtab_list_find(st->list,name)))
                return i;
return 0;
}

SYM_INFO*
symtab_insert(SYM_TAB *st,char *name,T_INFO* t)
{
SYM_INFO* i     = symtab_info_new(name,t);
st->list        = symtab_list_insert(st->list,i);

return i;
}

static int
symtab_info_equal(SYM_INFO* i1,SYM_INFO* i2)
{
```

```
/* rely on names and types being stored in a pool ''without duplicates'' */
return ((i1->name==i2->name)&&(i1->type==i2->type));
}

int
symtab_list_equal(SYM_LIST* l1,SYM_LIST* l2)
{
if (l1==l2)
        return 1;

while (l1&&l2)
        if (symtab_info_equal(l1->info,l2->info))
                {
                l1 = l1->next;
                l2 = l2->next;
                }
        else
                return 0;
if (l1) /* l2 == 0 */
        return 0;
else
        return (l1==l2);
}

SYM_LIST*
symtab_list_insert(SYM_LIST* l,SYM_INFO* i)
{
SYM_LIST* nl    = fmalloc(sizeof(SYM_LIST));
nl->info        = i;
nl->next        = l;
return nl;
}

SYM_INFO*
symtab_list_find(SYM_LIST* l,char* name)
{
for (; (l); l = l->next)
        if (l->info->name==name) /* this works if all names in string pool */
                return l->info;
return 0;
}

void
symtab_list_release(SYM_LIST* l)
{
if (l)
        {
        symtab_list_release(l->next);
        free(l);
        }
}

SYM_INFO*
symtab_info_new(char* name,T_INFO* t)
{
SYM_INFO* i     = fmalloc(sizeof(SYM_INFO));
i->name = name;
i->type = t;
return i;
}

void
symtab_info_print(FILE* f,SYM_INFO* info)
{
types_print(f,info->type);
fprintf(f," %s",info->name);
}

void
symtab_list_print(FILE* f,SYM_LIST* l,char* separator)
{
while (l)
        {
        symtab_info_print(f,l->info);
        if (l->next)
                fprintf(f,"%s",separator);
        l = l->next;
        }
}

void
symtab_print(FILE *f,SYM_TAB *tab)
{
if (!tab)
        fprintf(f,"<null symtab>");
else
        while (tab)
                {
                symtab_list_print(f,tab->list,"\n");
                if (tab->parent)
                        fprintf(f,"--------");
                tab = tab->parent;
                }
}
```

C.4 Types library

```
#ifndef TYPES_H
#define TYPES_H
/*
*       types.h(1.4)    17:11:16        97/12/08
*
*       Type pool management
*/
#include        "symtab.h"

typedef enum { int_t, float_t, fun_t, record_t, array_t } T_CONS; /* type construc

typedef struct {
        SYM_LIST                *fields;
        } T_RECORD;

typedef struct {
        struct type_info        *target;
        struct types_list       *source;
        } T_FUN;

typedef struct array_type {
        struct type_info        *base;
        } T_ARRAY;


typedef struct type_info {
        T_CONS  cons;
        union {
                T_FUN           fun;
                T_ARRAY         array;
                T_RECORD        record;
                } info;
        } T_INFO;

typedef struct types_list {
        T_INFO                  *type;
        struct types_list       *next;
        } T_LIST;

T_INFO* types_simple(T_CONS c);
T_INFO* types_fun(T_INFO* target,T_LIST *source);
T_INFO* types_record(SYM_LIST *fields);
T_INFO* types_array(T_INFO *base);
T_LIST* types_list_insert(T_LIST*,T_INFO*);
int     types_list_equal(T_LIST*,T_LIST*);
void    types_print(FILE*,T_INFO*);
void    types_list_print(FILE*,T_LIST*,char*);
void    types_list_release(T_LIST*);
void    types_print_all(FILE*);
```

```

#endif
```

```
/*	types.c(1.5)	17:11:14	97/12/08
*
*	Type pool management
*/
#include	<stdlib.h>	/* for malloc() and friends */
#include	<stdio.h>	/* for fprintf() and friends */
#include	<assert.h>	/* for assert(condition) */

#include	"util.h"
#include	"symtab.h"
#include	"types.h"

static T_LIST	*types	= 0;

static int
types_equal(T_INFO *t1, T_INFO *t2)
{
if (t1==t2)
	return 1;

if (t1->cons==t2->cons)
	switch (t1->cons)
		{
		case int_t:
		case float_t:
			return 1;
		case fun_t:
			if (!types_equal(t1->info.fun.target, t2->info.fun.target))
				return 0;
			else
				return types_list_equal(t1->info.fun.source, t2->in
		case array_t:
			return types_equal(t1->info.array.base, t2->info.array.base
		case record_t:
			return symtab_list_equal(t1->info.record.fields, t2->info.r
		default:	assert(0);
		}
return 0;
}


void
types_list_release(T_LIST* l)
{
if (l) /* free memory held by list nodes, not by types in the nodes */
	{
        types_list_release(l->next);
        free(l);
        }
}

int
types_list_equal(T_LIST* t1,T_LIST *t2)
{
if (t1==t2)
        return 1;

while (t1&&t2)
        if (types_equal(t1->type,t2->type))
                {
                t1      = t1->next;
                t2      = t2->next;
                }
        else
                return 0;

if (t1) /* t2 == 0 */
        return 0;
else    /* t1 == 0 */
        return (t1==t2);
}

static T_INFO*
types_find(T_INFO *t)
{
T_LIST  *tl;

for (tl=types;(tl);tl=tl->next)
        if (types_equal(t,tl->type))
                return tl->type;
return 0;
}

static T_INFO*
types_new(T_CONS c)
{
T_INFO  *t      = fmalloc(sizeof(T_INFO));
T_LIST  *tl     = fmalloc(sizeof(T_LIST));
t->cons         = c;
tl->type        = t;
tl->next        = types;
types           = tl;
return t;
}

```

```
T_INFO*
types_simple(T_CONS c)
{
T_INFO  t,*pt;
t.cons  = c;

if ((pt=types_find(&t)))
        return pt;
else
        return types_new(c);
}

T_INFO*
types_fun(T_INFO* target,T_LIST *source)
{
T_INFO  t,*pt;

t.cons                  = fun_t;
t.info.fun.target       = target;
t.info.fun.source       = source;

if (!(pt=types_find(&t)))
        {
        pt = types_new(fun_t);
        pt->info.fun.source     = source;
        pt->info.fun.target     = target;
        }
else
        types_list_release(source);
return pt;
}

T_INFO*
types_record(SYM_LIST *fields)
{
T_INFO  t,*pt;

t.cons                  = record_t;
t.info.record.fields    = fields;

if (!(pt=types_find(&t)))
        {
        pt = types_new(record_t);
        pt->info.record.fields  = fields;
        }
else
        symtab_list_release(fields);
return pt;
}
```

```

T_INFO*
types_array(T_INFO *base)
{
T_INFO  t,*pt;
t.cons                  = array_t;
t.info.array.base       = base;

if (!(pt=types_find(&t)))
        {
        pt = types_new(array_t);
        pt->info.array.base     = base;
        }
return pt;
}

T_LIST*
types_list_insert(T_LIST* l,T_INFO *t)
{
T_LIST  *nl     = fmalloc(sizeof(T_LIST));
nl->type        = t;
nl->next        = l;
return nl;
}

void
types_print(FILE* f,T_INFO *t)
{
if (!t)
        fprintf(f,"<null_type>");
else
        switch (t->cons) {
                case int_t:
                        fprintf(f,"int");
                        break;
                case float_t:
                        fprintf(f,"float");
                        break;
                case fun_t:
                        types_print(f,t->info.fun.target);
                        fprintf(f,"(");
                        types_list_print(f,t->info.fun.source,",");
                        fprintf(f,")");
                        break;
                case record_t:
                        fprintf(f,"struct {");
                        symtab_list_print(f,t->info.record.fields,";");
                        fprintf(f,"}");
                        break;
                case array_t:
                        types_print(f,t->info.array.base);
                        fprintf(f,"*");
                        break;
                default:
                        assert(0);
                }
/* fprintf(f,"[%x]",(unsigned int)t); */
}

void
types_list_print(FILE* f,T_LIST* tl,char* separator)
{
/* fprintf(f,"{%x}",(unsigned int)tl); */
while (tl)
        {
        types_print(f,tl->type);
        if (tl->next)
                fprintf(f,"%s",separator);
        tl = tl->next;
        }
}

void
types_print_all(FILE* f)
{
types_list_print(f,types,"\n");
}
```

C.5 Type checking routines

```
#ifndef CHECK_H
#define CHECK_H
/*
*       check.h(1.3)    10:31:17        97/12/10
*
*       Semantic checks.
*/
#include        "types.h"
#include        "symtab.h"

void            check_assignment(T_INFO*,T_INFO*);
T_INFO*         check_record_access(T_INFO* t,char* field);
T_INFO*         check_array_access(T_INFO* ta,T_INFO* ti);
T_INFO*         check_arith_op(int token,T_INFO* t1,T_INFO* t2);
T_INFO*         check_relop(int token,T_INFO* t1,T_INFO* t2);
T_INFO*         check_fun_call(SYM_TAB*,char*,T_LIST**);
SYM_INFO*       check_symbol(SYM_TAB* scope,char* name);
```

```

#endif
```

```
/*
*       check.c(1.4)    17:46:20        97/12/10
*/
#include        <stdio.h>               /* for fprintf() and friends */
#include        "check.h"

#include        "minic.tab.h"           /* for tokens */

extern int      lineno;                 /* defined in minic.y */

static void
error(char *s1,char *s2,T_INFO* t1,char* s3,char* s4,T_INFO* t2)
{
fprintf(stderr,"type error on line %d: ",lineno);
if (s1) fprintf(stderr,"%s",s1);
if (s2) fprintf(stderr,"%s",s2);
if (t1) types_print(stderr,t1);
if (s3) fprintf(stderr,"%s",s3);
if (s4) fprintf(stderr,"%s",s4);
if (t2) types_print(stderr,t2);
fprintf(stderr,"\n");
exit(1);
}

void
check_assignment(T_INFO* tlexp,T_INFO* texp)
{
if (tlexp!=texp)
        error("cannot assign ",0,texp," to ",0,tlexp);
}

T_INFO*
check_record_access(T_INFO* t,char* field)
{
SYM_INFO        *i;

if (t->cons!=record_t)
        error("not a record: ",0,t," for field ",field,0);
if ((i=symtab_list_find(t->info.record.fields,field)))
        return i->type;
error("record type ",0,t," has no field ",field,0);
return 0;
}

T_INFO*
check_array_access(T_INFO* ta,T_INFO* ti)
```

```
{
if (ta->cons!=array_t)
        error("not an array type: ",0,ta,0,0,0);
if (ti->cons!=int_t)
        error("index for ",0,ta," must be integer, not ",0,ti);
return ta->info.array.base;
}

T_INFO*
check_arith_op(int token,T_INFO* t1,T_INFO* t2)
{
if (t1!=t2)
        error("type ",0,t1," does not match ",0,t2);
if ((t1->cons!=int_t)&&(t2->cons!=float_t))
        error("type ",0,t1," is not numeric",0,0);
return t1;
}

T_INFO*
check_relop(int token,T_INFO* t1,T_INFO* t2)
{
if (t1!=t2)
        error("type ",0,t1," does not match ",0,t2);
return types_simple(int_t);
}

SYM_INFO*
check_symbol(SYM_TAB* scope,char* name)
{
SYM_INFO* i = symtab_find(scope,name);

if (!i)
        error("undeclared variable \"",name,0,"\"",0,0);
return i;
}

T_INFO*
check_fun_call(SYM_TAB* scope,char* name,T_LIST** args)
{
SYM_INFO*       i = symtab_find(scope,name);
T_INFO*         ft;

if (!i)
        error("undeclared function \"",name,0,"\"",0,0);

ft = i->type;

if (ft->cons!=fun_t)
        error(name," is not a function",0,0,0,0);

if (types_list_equal(ft->info.fun.source,(args?*args:0)))
        {
        /* release type_list from args, replace by equal list
        *  from function type
        */
        if (args) {
                types_list_release(*args);
                *args   = ft->info.fun.source;
                }
        return ft->info.fun.target;
        }
error("bad type of arguments for ",name,0,0,0,0);
return 0;
}
```

C.6 Parser with semantic actions

```
%{
/*      minic.y(1.9)    17:46:21        97/12/10
*
*       Parser demo of simple symbol table management and type checking.
*/
#include        <stdio.h>       /* for (f)printf() */

#include        "symtab.h"
#include        "types.h"
#include        "check.h"

int             lineno  = 1;    /* number of current source line */
extern int      yylex();        /* lexical analyzer generated from lex.l */
extern char     *yytext;        /* last token, defined in lex.l
*/
SYM_TAB         *scope;         /* current symbol table, initialized in lex.l */
char            *base;          /* basename of command line argument */

void
yyerror(char *s)
{
fprintf(stderr,"Syntax error on line #%d: %s\n",lineno,s);
fprintf(stderr,"Last token was \"%s\"\n",yytext);
exit(1);
}

%}

%union  {
        char*           name;
        int             value;
        T_LIST*         tlist;
        T_INFO*         type;
        SYM_INFO*       sym;
        SYM_LIST*       slist;
        }

%token  INT FLOAT NAME STRUCT IF ELSE RETURN NUMBER LPAR RPAR LBRACE RBRACE
%token  LBRACK RBRACK ASSIGN SEMICOLON COMMA DOT PLUS MINUS TIMES DIVIDE EQUAL

%type   <name>  NAME
%type   <value> NUMBER
%type   <type>  type parameter exp lexp
%type   <tlist> parameters more_parameters exps
%type   <sym>   field var
%type   <slist> fields

/*      associativity and precedence: in order of increasing precedence */

%nonassoc       LOW  /* dummy token to suggest shift on ELSE */
%nonassoc       ELSE /* higher than LOW */

%nonassoc       EQUAL
%left           PLUS    MINUS
%left           TIMES   DIVIDE
%left           UMINUS  /* dummy token to use as precedence marker */
%left           DOT     LBRACK  /* C compatible precedence rules */

%%
program         : declarations
                ;

declarations    : declaration declarations
                | /* empty */
                ;

declaration     : fun_declaration
                | var_declaration
                ;

fun_declaration : type NAME {   /* this is $3 */
                        $<sym>$ = symtab_insert(scope,$2,0);
                        scope = symtab_open(scope); /* open new scope */
                        scope->function = $<sym>$; /* attach to this function */
                        }
                  LPAR parameters RPAR {        /* this is $7 */
                        $<sym>3->type = types_fun($1,$5);
                        }
                  block { scope = scope->parent; }
                ;

parameters      : more_parameters       { $$ = $1; }
                |                       { $$ = 0; }
                ;

more_parameters : parameter COMMA more_parameters
                                        { $$ = types_list_insert($3,$1); }
                | parameter             { $$ = types_list_insert(0,$1); }
                ;

parameter       : type NAME {
                        symtab_insert(scope,$2,$1); /* insert in symbol table */
                        $$ = $1; /* remember type info */
                        }
                ;

block           : LBRACE                {  scope = symtab_open(scope); }
                  var_declarations statements RBRACE
                                        { scope = scope->parent; /* close scope */
                ;

var_declarations: var_declaration var_declarations
                |
                ;

var_declaration : type NAME SEMICOLON   { symtab_insert(scope,$2,$1); }
                ;

type            : INT                   { $$ = types_simple(int_t); }
                | FLOAT                 { $$ = types_simple(float_t); }
                | type TIMES            { $$ = types_array($1); }
                | STRUCT LBRACE fields RBRACE /* record */
                                        { $$ = types_record($3); }
                ;

fields          : field fields          { $$ = symtab_list_insert($2,$1); }
                |                       { $$ = 0; }
                ;

field           : type NAME SEMICOLON   { $$ = symtab_info_new($2,$1); }
                ;

statements      : statement SEMICOLON statements
                | /* empty */
                ;

statement       : IF LPAR exp RPAR statement            %prec LOW
                | IF LPAR exp RPAR statement ELSE statement     /* shift */
                | lexp ASSIGN exp       { check_assignment($1,$3); }
                | RETURN exp /* statements always in scope with function */
                        { check_assignment(scope->function->type->info.fun.target,
                | block
                ;

lexp            : var                   { $$ = $1->type; }
                | lexp LBRACK exp RBRACK{ $$ = check_array_access($1,$3); }
                | lexp DOT NAME         { $$ = check_record_access($1,$3); }
                ;

exp             : exp DOT NAME          { $$ = check_record_access($1,$3); }
                | exp LBRACK exp RBRACK { $$ = check_array_access($1,$3); }
                | exp PLUS exp          { $$ = check_arith_op(PLUS,$1,$3); }
                | exp MINUS exp         { $$ = check_arith_op(MINUS,$1,$3); }
                | exp TIMES exp         { $$ = check_arith_op(TIMES,$1,$3); }
                | exp DIVIDE exp        { $$ = check_arith_op(DIVIDE,$1,$3); }
                | exp EQUAL exp         { $$ = check_relop(EQUAL,$1,$3); }
                | LPAR exp RPAR { $$ = $2; }
                | MINUS exp     %prec UMINUS /* this will force a reduce */
                                        { $$ = check_arith_op(UMINUS,$2,0); }
                | var                   { $$ = $1->type; }
                | NUMBER                { $$ = types_simple(int_t); }
                | NAME LPAR RPAR        { $$ = check_fun_call(scope,$1,0); }
                | NAME LPAR exps RPAR   { $$ = check_fun_call(scope,$1,&$3); }
                ;

exps            : exp                   { $$ = types_list_insert(0,$1); }
                | exp COMMA exps        { $$ = types_list_insert($3,$1); }
                ;

var             : NAME                  { $$ = check_symbol(scope,$1); }
%%

int
main(int argc,char *argv[])
{
if (argc!=2) {
        fprintf(stderr,"Usage: %s base_file_name",argv[0]);
        exit(1);
        }
base = argv[1];
return yyparse();
}
```

C.7 Utilities

```
#ifndef UTIL_H
```

```
#define UTIL_H
/*
*       util.h(1.1)     11:24:32        97/12/08
*
*       General utility functions
*/
#include        <stdlib.h>

void    *fmalloc(size_t);       /* malloc() version that aborts on failure */

#endif
```

```
/*
*       util.c(1.2)     10:13:01        97/12/10
*
*       Utility functions.
*/

#include        <stdio.h>       /* for fprintf() and friends */

#include        "util.h"

/* aborts on failure */
void    *fmalloc(size_t s)
{
void    *ptr = malloc(s);
if (!ptr) {
        fprintf(stderr,"Out of memory in fmalloc(%d)\n",s);
        exit(1);
        }
return ptr;
}
```

C.8 Driver script

```
#!/bin/sh
#
#       Usage:  mct basename
#
#       e.g. "mct tst"  to compile tst.c,
#
#       No output except errors on stderr.
#
#       First make sure source file exists
#
if [ ! -f "$1.c" ]
then
        echo "Cannot open \"$1.c\""
        exit 1
fi
#
minic $1 <$1.c
```

C.9 Makefile

```
#       %M%(%I%)        %U%     %E%
#
CFLAGS=         -Wall -g
CC=             gcc
#
HH_FILES=       util.h names.h symtab.h types.h check.h
CC_FILES=       util.c names.c symtab.c types.c check.c
SOURCE=         Makefile $(CC_FILES) $(HH_FILES) minic.y lex.l mct.sh
#
#
all:            minic mct
minic:          minic.tab.o lex.yy.o $(CC_FILES:%.c=%.o)
                gcc $(CFLAGS) -o $@ $^

lex.yy.c:       lex.l minic.tab.h
                flex lex.l
#
include         make_dependencies
#
#       Bison options:
#
#       -v      Generate minic.output showing states of LALR parser
#       -d      Generate minic.tab.h containing token type definitions
#
minic.tab.h\
minic.tab.c:    minic.y
                bison -v -d $^
#
clean:
                rm -f lex.yy.c minic.tab.[hc] *.o mct minic *.jasm *.class minic.ou
#
tar:
                tar cvf minic.tar $(SOURCE)
source:
                @echo $(SOURCE)
#
####### creating dependencies
#
dep:            lex.yy.c minic.tab.c $(CC_FILES)
                $(CC) -M $(CC_FILES) lex.yy.c minic.tab.c >make_dependencies
#
```

```
####### sccs rules
#
delta:
		@echo "Comment: \c"; read inp; \
		for f in $(SOURCE); do\
			[ -f SCCS/p.$$f ] && \
			{ echo "#$$f:"; sccs delget -y"\"$$inp\"" $$f; } done;\
			true
edit:
		@for f in $(SOURCE); do\
			[ ! -f SCCS/p.$$f ] && \
			{ echo "#$$f:"; sccs edit $$f; } done; true
create:
		@for f in $(SOURCE); do\
			[ ! -f SCCS/s.$$f ] && \
			{ echo "#$$f:"; sccs create $$f; } done; rm -f ,*; true
```

Index

List of Figures

List of Tables

Bibliography

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, techniques and tools, second edition*. Pearson,Addison Wesley, 2007.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, techniques and tools*. Addison Wesley, 1986.

[Bar03] Jonathan Bartlett. *Programming from the Ground Up*. `http://www.bartlettpublishing.com/` and `http://savannah.nongnu.org/projects/pgubook/`, 2003.

[FL91] Charles N. Fischer and Richard J. LeBlanc. *Crafting a compiler with C*. Addison-Wesley, 1991.

[HU69] John E. Hopcroft and Jeffrey D. Ullman. *Formal languages and their relation to automata*. Addison Wesley, 1969.

[Sal73] A. Salomaa. *Formal languages*. ACM Monograph Series. Academic Press, 1973.