Chapter 2 A Simple Compiler

Wuu Yang

National Chiao-Tung University, Taiwan, R.O.C.

first draft: February 16, 2010

current version: January 25, 2011

©January 25, 2011 by Wuu Yang. All rights reserved.

Chapter outline: A simple compiler

1. An informal definition of the ac language
2. Formal definition of ac
3. Phases of a simple compiler
4. Scanning
5. Parsing
6. Abstract syntax tree
7. Semantic analysis
8. Code generation

§2.1 Informal definition of ac

`ac`: the *adding calculator* language

1. **types**: only integers and floats
2. **keywords**: In ac, there are three reserved keywords: `f` (float), `i` (integer), and `p` (print). Other common keywords in programming languages include `if`, `begin`, `while`, etc.
3. **variables**: ac offers 23 variables: `a` through `z`, except the three keywords. Variables must be declared before use.

Some languages require explicit type `cast` while ac provides automatic (i.e., implicit) conversion from integers to floats.

The target language is `dc` (a *desk calculator*), which is a stack-based calculator that makes use of reverse Polish notation (RPN).

§2.2 Formal definition of ac

We use a context-free grammar to define the syntax of ac. A context-free grammar is a set of production rules for rewriting the symbols. For example, the symbol `Stmt` can be rewritten as one of the two following sequences of symbols:

```
id assign Val Expr
print id
```

This means that there are two kinds of statements in ac: the assignment statement and the print statement.

In a context-free grammar, the left-hand side of a rule is a *nonterminal*. On the right is a sequence of 0 or more *terminals* and nonterminals. In order to avoid confusion, all terminals and nonterminals are distinct. One of the nonterminals is designated as the *start symbol*.

```
Prog ::= Dcls Stmts $
Dcls ::= Dcl Dcls
Dcls ::= λ
Dcl ::= floatdcl id
Dcl ::= intdcl id
Stmts ::= Stmt Stmts
Stmts ::= λ
Stmt ::= id assign Val Expr
Stmt ::= print id
Expr ::= plus Val Expr
Expr ::= minus Val Expr
Expr ::= λ
Val ::= id
Val ::= inum
Val ::= fnum
```

We adopt the convention that nonterminals begin with a capital letter and terminals a small-case letter.

A *derivation* starts from the start symbol, `Prog` in the above example, and continuously replaces a nonterminal with the right-hand side of a production rule.

We usually use the special terminal `$` to denote *end-of-file.* The right-hand side of a rule cold be empty, that is, the null string, which is denoted with λ.

See Figure 2.2 for a derivation.

Step	Sentential Form	Production Number
1	⟨Prog⟩	
2	⟨Dcls⟩ Stmts $	1
3	⟨Dcl⟩ Dcls Stmts $	2
4	floatdcl id ⟨Dcls⟩ Stmts $	4
5	floatdcl id ⟨Dcl⟩ Dcls Stmts $	2
6	floatdcl id intdcl id ⟨Dcls⟩ Stmts $	5
7	floatdcl id intdcl id ⟨Stmts⟩ $	3
8	floatdcl id intdcl id ⟨Stmt⟩ Stmts $	6
9	floatdcl id intdcl id id assign ⟨Val⟩ Expr Stmts $	8
10	floatdcl id intdcl id id assign inum ⟨Expr⟩ Stmts $	14
11	floatdcl id intdcl id id assign inum ⟨Stmts⟩ $	12
12	floatdcl id intdcl id id assign inum ⟨Stmt⟩ Stmts $	6
13	floatdcl id intdcl id id assign inum id assign ⟨Val⟩ Expr Stmts $	8
14	floatdcl id intdcl id id assign inum id assign id ⟨Expr⟩ Stmts $	13
15	floatdcl id intdcl id id assign inum id assign id plus ⟨Val⟩ Expr Stmts $	10
16	floatdcl id intdcl id id assign inum id assign id plus fnum ⟨Expr⟩ Stmts $	15
17	floatdcl id intdcl id id assign inum id assign id plus fnum ⟨Stmts⟩ $	12
18	floatdcl id intdcl id id assign inum id assign id plus fnum ⟨Stmt⟩ Stmts $	6
19	floatdcl id intdcl id id assign inum id assign id plus fnum print id ⟨Stmts⟩ $	9
20	floatdcl id intdcl id id assign inum id assign id plus fnum print id $	7

Figure 2.2: Derivation of an ac program using the grammar in Figure 2.1.

§2.2.2 Token specification

Terminals in a CFG are called *tokens*. A token is a sequence of characters. For example, the `assign` token in the Pascal language consists of two characters "`:=`".

In most common programming languages, an `identifier` token may consist of as many as characters as a programmer wishes. However, in `ac`, for the sake of simplicity, an `identifier` token consists of a single character.

A programming language also includes other tokens not corresponding to terminals in the CFG. For example, comments, white spaces (blanks and tabs), and compilation directives (`pragma`) do not correspond to terminals in the CFG. They are handled in the scanner and will not be passed to the parser.

We usually use *regular expressions* to define tokens. Figure 2.3 shows the token definition for ac.

Terminal	Regular Expression
floatdcl	"f"
intdcl	"i"
print	"p"
id	[a − e] | [g − h] | [j − o] | [q − z]
assign	"="
plus	"+"
minus	"-"
inum	$[0-9]^+$
fnum	$[0-9]^+.[0-9]^+$
blank	$(" \ ")^+$

Figure 2.3: Formal definition of ac tokens.

We may represent a derivation process as a *parse tree*. See Figure 2.4.

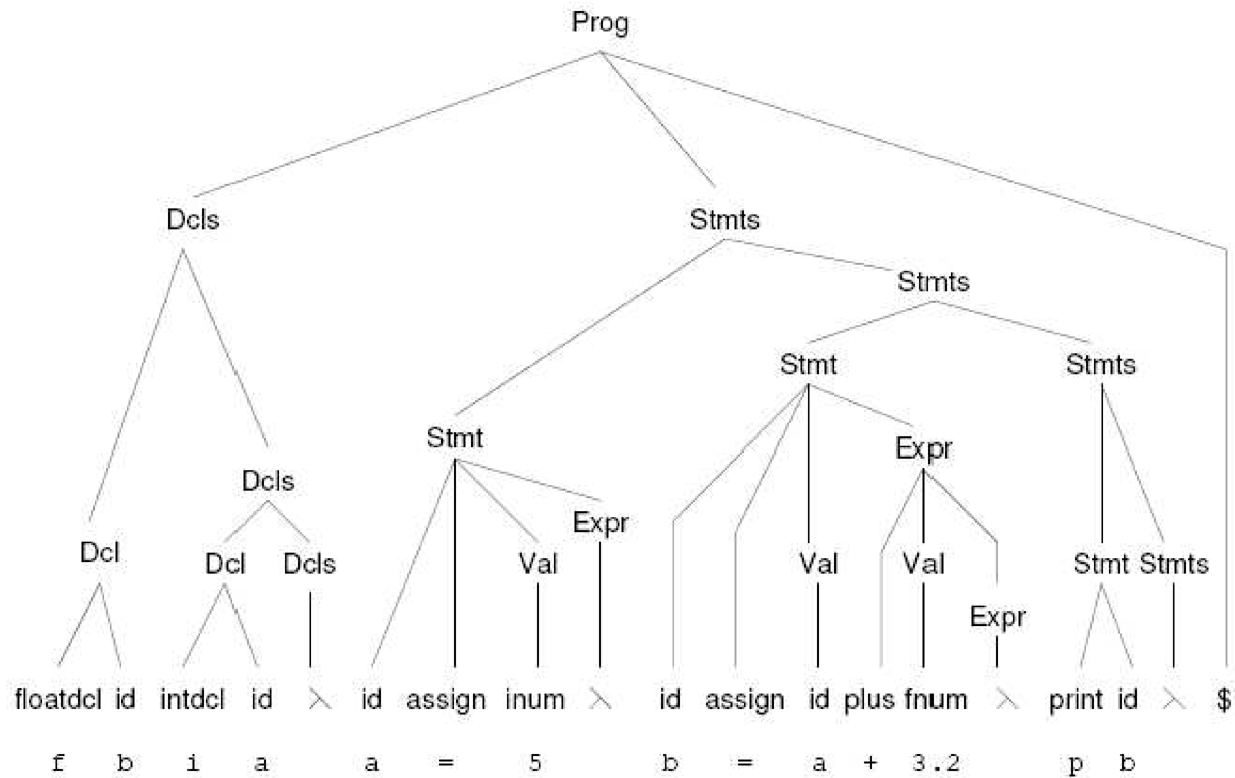


Figure 2.4: An ac program and its parse tree.

Example. Below is the `Micro` language. The start symbol is `<system goal>`.

1.	<program>	→	**begin** <statement list> **end**
2.	<statement list>	→	<statement> {<statement>}
3.	<statement>	→	ID := <expression> ;
4.	<statement>	→	**read** (<id list>) ;
5.	<statement>	→	**write** (<expr list>) ;
6.	<id list>	→	ID {, ID}
7.	<expr list>	→	<expression> {, <expression>}
8.	<expression>	→	<primary> {<add op> <primary>}
9.	<primary>	→	(<expression>)
10.	<primary>	→	ID
11.	<primary>	→	INTLITERAL
12.	<add op>	→	PLUSOP
13.	<add op>	→	MINUSOP
14.	<system goal>	→	<program> SCANEOF

Figure 2.4 Extended CFG Defining Micro

Here is a derivation.

begin A := B - 3 + A; end eof

begin id assign id minus int plus id semi end eof

<goal> ⇒ <program> eof

⇒ begin <stmt list> end eof

⇒ begin <stmt> { <stmt> } end eof

⇒ begin <stmt> end eof

⇒ begin id assign <expr> ; end eof

⇒ begin id assign <pri> { <add> <pri> } ; end eof

⇒ begin id assign id { <add> <pri> } ; end eof

⇒ begin id assign id <add> <pri> { <add> <pri> } ; end eof

⇒ begin id assign id - <pri> { <add> <pri> } ; end eof

⇒ begin id assign id - int { <add> <pri> } ; end eof

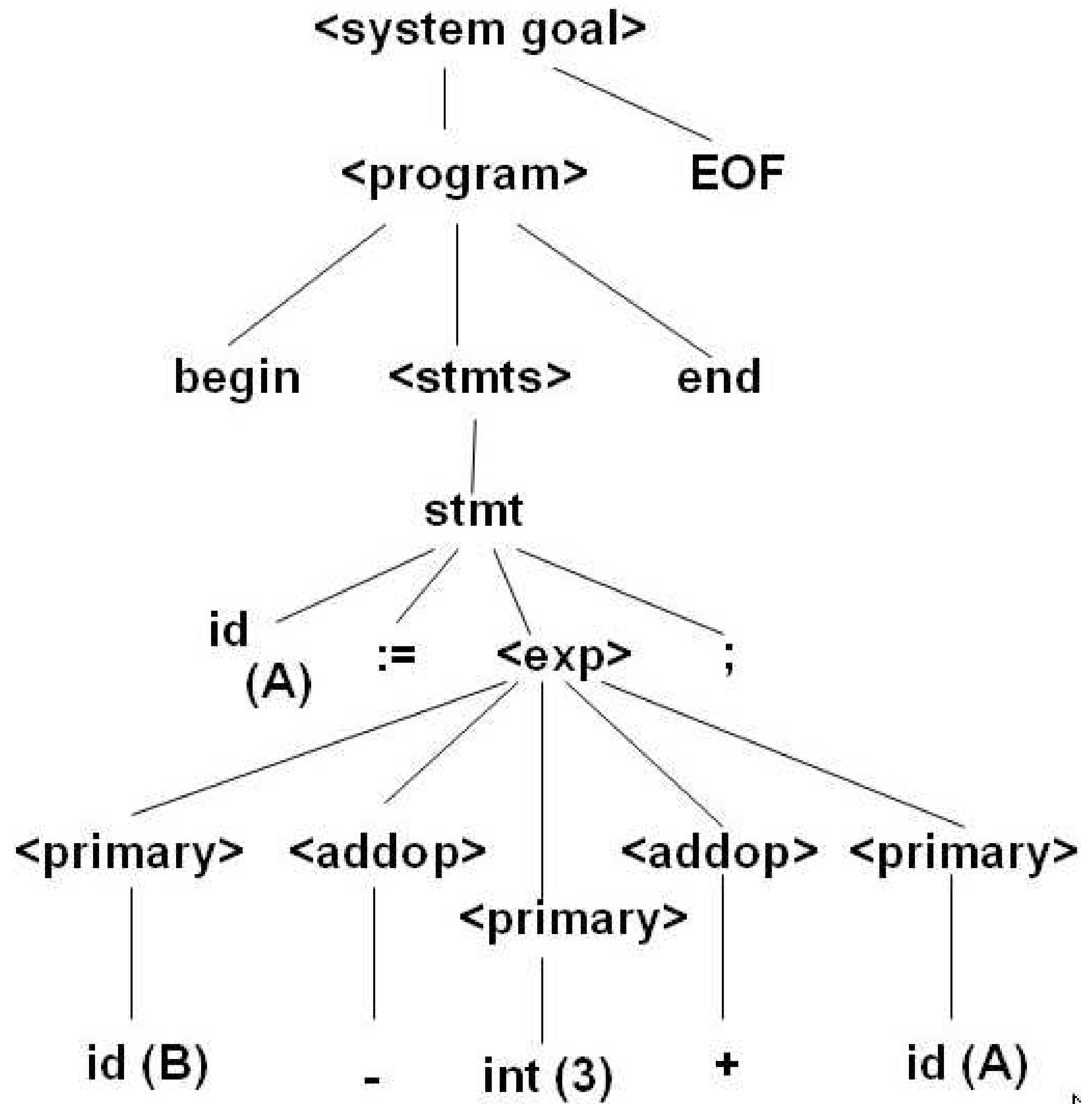
⇒ begin id assign id - int <add> <pri> { <add> <pri> } ; end eof

$\Rightarrow$ begin id assign id - int + <pri> $\{$ <add> <pri> $\}$; end eof

$\Rightarrow$ begin id assign id - int + id $\{$ <add> <pri> $\}$; end eof

$\Rightarrow$ begin id assign id - int + id ; end eof

Here is a syntax tree.



§2.3 Phases of a simple compiler

See slide 1-38 for a structure of a typical compiler.

§2.4 Scanning

The task of a scanner is to divide the stream of input characters into a stream of tokens. Some tokens are handled by the scanner or are deleted. Other tokens are passed to the parser.

Figure 2.5 is a basic scanner.

```
function Scanner() returns Token
   while s.Peek() == blank do call s.Advance()
   if s.EOF() then ans.type := $
   else if s.Peek() belongs to { 0, 1, ..., 9 }
        then ans := ScanDigits()
   else ch := s.Advance()
        switch (ch)
        case { a, b, ..., z } - { f, i, p }:
           ans.type := id
           ans.val  := ch
        case f:
           ans.type := floatdcl
        case i:
           ans.type := intdcl
        case p:
           ans.type := print
```

```
        case =:
           ans.type := assign
        case +:
           ans.type := plus
        case -:
           ans.type := minus
        case default:
           call LexError()
   return(ans)
end
```

Figure 2.5 Scanner for ac. s is the input stream.

```
procedure ScanDigits() returns Token
   tok.val := " "
   while s.Peek() belongs to { 0, 1, ..., 9 } do
      tok.val := tok.val + s.Advance()
   if s.Peek() != "." then tok.type := inum
   else tok.type := fnum
        tok.val  := tok.val + s.Advance()
        while s.Peek() belongs to { 0, 1, ..., 9 } do
           tok.val := tok.val + s.Advance()
   return(tok)
end
```

Figure 2.6 Finding inum and fnum tokens for ac.

A token has a *type*, such as `id`, `inum`, `print`, `plus`, etc.

A token also has a *semantic value*. Some semantic values are trivial. For instance, the *plus* token's semantic value is simply the character "`+`". There is little contents in the *plus* token.

On the other hand, the `inum` token's semantic value is rich. In addition to the type, we also need to the *integer value* of the `inum` token.

Some issues with tokens:

- Some tokens are prefixes of another token. For example, in C, we have both `+` and `++` tokens.
- There is an *escape* mechanism. For example, A string is usually written as `‘‘ ’’`. What if the double quote `‘‘` is part of the string? We use `‘‘abc \‘def’’` to denote such a string.
- Sometimes there are more divisions of a character sequence. For example, `123456`.

In the implementation of a scanner, we may define an enumeration type to represent the token types, such as

```
typedef enum {
    floatdcl, intdcl, print, id, assign, plus, minus, inum,
    fnum, blank
} tokentype;
```

A scanner can be automatically generated from the regular-expression specification of the tokens in the language. A famous scanner generator is the `lex` tool.

§2.5 Parsing

The parser attempts to find a derivation of the input token stream. A derivation can be represented as a syntax tree, which is also called the syntactic structure of the input token stream.

There are many parsing techniques. We will use the *recursive descent* algorithm here.

- Each nonterminal has a parsing procedure.
- The parsing procedures are mutually recursive.
- For symbols on the right-hand side of a production,
  - terminal: match the terminal.
  - nonterminal: call the parsing procedure.
- If a nonterminal has several production rules, there are two alternatives:
  1. Use back-tracking.

2. Compute the FIRST sets. Look ahead the next token(s) to decide which rule should be used.

- Parser is started by invoking procedure corresponding to the start symbol.

For the production rule,

```
Prog ::= Dcls Stmts $
```

We will have the parsing procedure:

```
procedure Prog()
    call Dcls();
    call Stmts();
    call Match(ts, $);
end
```

For each symbole on the right-hand side of a production rule,

- If the symbol is a nonterminal, we call the correspoind parsing procedure.
- If the symbol is a terminal, we call the `Match` procedure.

This works fine if there is exactly one rule for a nonterminal. What if there are multiple rules? For instance,

```
Stmt ::= id assign Val Expr
Stmt ::= print id
```

We may try each rule in turn. This trial-and-error approach is not good.

We may also use the next input terminal to select a rule. For the above example, if the next input terminal is `id`, we will use the first rule. If the next input terminal is `print`, we will use the second rule. If the next input terminal is neither `id` nor `print`, an error has occurred. We will show how to compte the *predict* set of a production rule in the general case.

Figure 2.7 shows the parsing procedure for `Stmt`.

```
procedure Stmt()
   if ts.Peek() == id then
        call Match(ts, id)
        call Match(ts, assign)
        call Val()
        call Expr()
   else if ts.Peek() == print then
        call Match(ts, print)
        call Match(ts, id)
   else call Error()
end
```

Figure 2.7 Parsing procedure for Stmt. ts is the input token stream.

Figure 2.8 shows the parsing procedure for `Stmts`.

```
procedure Stmts()
   if ts.Peek() == id or ts.Peek() == print then
        call Stmt()
        call Stmts()
   else if ts.Peek() == $ then
        // do nothing for the lambda production
   else call Error()
end
```

Figure 2.8 Parsing procedure for Stmts.

§2.6 Abstract syntax tree

A parse tree (also called a *concrete syntax tree*) usually includes information that is useless in the remaining phases of a compiler, for instance, punctuation marks and delimiters. Thus, we may simplify it to obtain an *abstract syntax tree*. Figure 2.9 is an abstract syntax tree.

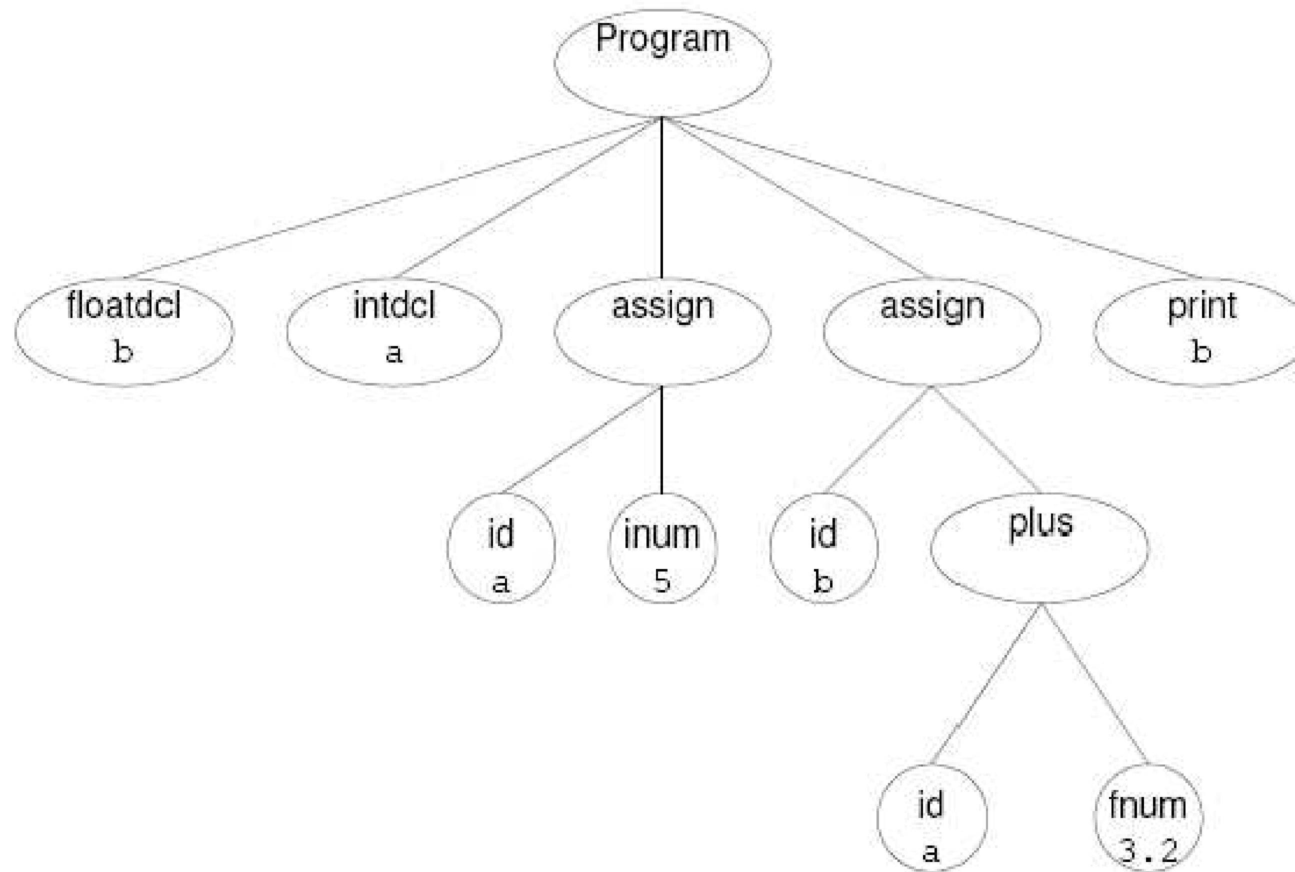


Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.

Issue 1: Syntactic analysis is not powerful enough to cover all aspects of a programming language

The scanner and parser together check the syntax of the input program. However, there are many issues that cannot be easily specified in the context-free grammar and hence cannot be checked by the scanner and the parser. For instance, if variable `a` has the integer type and variable `b` has the boolean type, then the following expression is wrong:

```
a + b
```

We cannot add an integer value to a boolean value . This constraint cannot be specified in the context-free grammar.

Another constraint is that a variable has to be declared before being used. This constraint cannot be specified in the context-free grammar. There are many such constraints.

Similarly, there are expressions that have different meanings in different contexts. For instance, in Java,

```
x.y.z
```

could mean package x, class y, and static field z. It could also mean variable x, field y, and field z within y. The context-free grammar will offer no help in this case.

Most languages also provide *overloading*. For instance, `+` could mean integer addition, string concatenation, and set union. The context-free grammar will be unable to distinguish the differences.

Issue 2: Software engineering discipline dictates an internal representation of programs.

Syntax-directed translation is an efficient compilation method that performs all aspects of program translation during syntax analysis. However, from a software-engineering perspective, it would be more advantageous to partition compilation into *phases*. These phases would share a common internal representation of a program, which is the abstract syntax tree.

The design of the abstract syntax tree must take the requirements and operations of all the compilation phases. All the phases essentially traverse, examine, modify, and transform the abstract syntax tree.

§2.7 Semantic analysis

All the operations done after the AST is constructed are collectively called *semantic analyses*. These include declaration processing, symbol-table construction, type checking, etc.

§2.7.1 Symbol tables

Symbols in a program include type names, function names, variable names, class names, etc. A symbol has many attributes, such as class, type, scope, address, access restrictions, etc.

In Figure 2.10, we use a *visit* procedure to visit nodes in the AST (in some order). *SymDeclaring* denotes a node type that represents a declaration.

```
procedure Visit(SymDeclaring n)
   if n.GetType == floatdcl
```

```
    then call EnterSymbol(n.GetId(), float)
    else call EnterSymbol(n.GetId(), integer)
end

procedure EnterSymbol(name, type)
    if SymbolTable[name] == null
    then SymbolTable[nmae] := type
    else Error("duplicate declaration")
end

procedure LookUpSymbol(name) returns type
    return(SymbolTable[name])
end
```

Figure 2.10 Symbol table construction for ac.

Figure 2.11 is a sample symbol table.

Symbol	Type	Symbol	Type	Symbol	Type
a	integer	k	null	t	null
b	float	l	null	u	null
c	null	m	null	v	null
d	null	n	null	w	null
e	null	o	null	x	null
g	null	q	null	y	null
h	null	r	null	z	null
j	null	s	null		

Figure 2.11: Symbol table for the ac program from Figure 2.4.

2.7.2 Type checking

The ac language's type system is quite simple. For other programming languages, there are a *type hierarchy* with automatic *widening* and *narrowing* operations.

A type checker visits the AST bottom-up. Figure 2.12 shows the type checker for ac.

```
procedure visit(Computing n)
  n.type := Consistent(n.child1, n.child2)
end

procedure visit(Assigning n)
  n.type := Convert(n.child2, n.child1.type)
end

procedure visit(SymReferencing n)
  n.type := LookUpSymbol(n.id)
end

procedure visit(IntConsting n)
  n.type := integer
end
```

```
procedure visit(FloatConsting n)
  n.type := float
end

// type checking utilities
function Consistent(c1, c2) returns type
  m := Generalize(c1.type, c2.type)
  call Convert(c1, m)
  call Convert(c2, m)
  return(m)
end

function Generalize(t1, t2) returns type
  if t1 == float or t2 == float then ans := float
  else ans := integer
  return(ans)
```

```
end

procedure Convert(n, t)
  if n.type == float and t == integer then Error("illegal")
  else if n.type == integer and t == float
       then // replace node n by convert-to-float of node n
end
```

Figure 2.12 Type analysis for ac.

During the visit, a type checker will *decorate* the AST by adding information to the nodes. For instance, we may add type information to the operator nodes and add additional nodes for type conversion and all implicit operations. The information and the additional nodes are needed by the code generator. Figure 2.13 is the AST after semantic analysis.

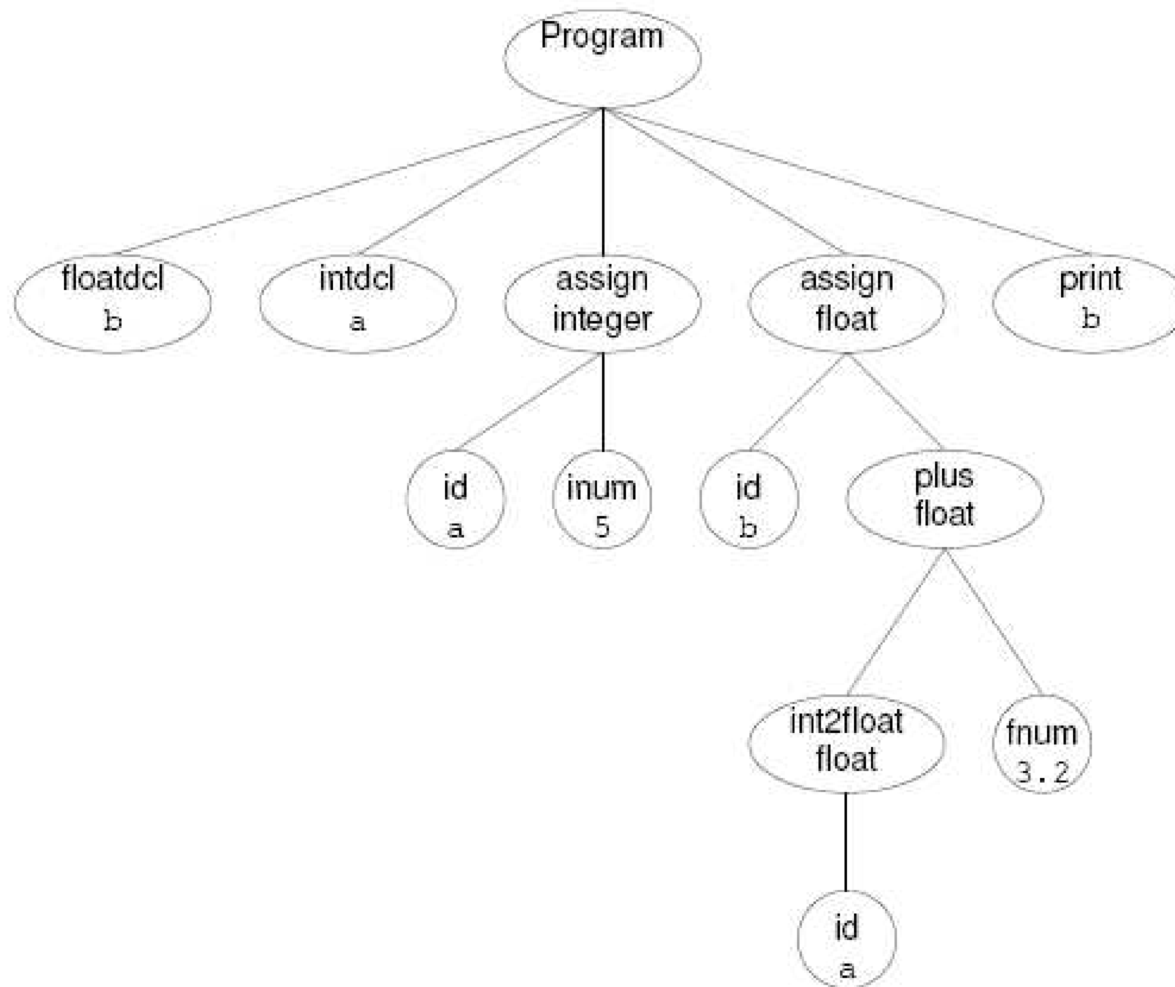


Figure 2.13: AST after semantic analysis.

§2.8 Code generation

Finally we need to generate machine code for the input program. Here we will use a *stack machine* as our target machine. Figure 1 is the generated code. Programming languages such as C# and Java also make use of a stack machine model.

The code generator visits the AST top-down, applying methods based on the nodes' types. Figure **??** shows the code.

Code	Source	Comments
5	a = 5	Push 5 on stack
sa		Pop the stack, storing (s) the popped value in register a
0 k		Reset precision to integer
la	b = a + 3.2	Load (l) register a, pushing its value on stack
5 k		Set precision to float
3.2		Push 3.2 on stack
+		Add: 5 and 3.2 are popped from the stack and their sum is pushed
sb		Pop the stack, storing the result in register b
0 k		Reset precision to integer
lb	p b	Push the value of the b register
p		Print the top-of-stack value
si		Pop the stack by storing into the i register

Figure 2.15: Code generated for the AST shown in Figure 2.9.

Figure 1: Code for a stack machine

```
procedure visit(Assigning n)
  call codegen(n.child2)
  call emit("s")
  call emit(n.child1.id)
  call emit("0 k")
end

procedure visit(Computing n)
  call codegen(n.child1)
  call codegen(n.child2)
  call emit(n.operation)
end
```

```
procedure visit(SymReferencing n)
  call emit("l")
  call emit(n.id)
end

procedure visit(Printing n)
  call emit("l")
  call emit(n.id)
  call emit("p")
  call emit("si")
end

procedure visit(Converting n)
  call codegen(n.child)
  call emit("5 k")
end
```

```
procedure visit(Consting n)
  call emit(n.val)
end
```

Figure 2.14 Code generation for ac

Appendix. A complete recursive descent parser

Here is the complete recursive descent parser for the Micro language on slide 11.

Simple production rules:

```
<goal> ::= <program> eof

 void goal() {
      program();
      match(eof);
 }


<program> ::= begin <stmt list> end

 void program() {
      match(begin);
      stmt_list();
      match(end);
 }
```

For multiple rules:

```
<stmt> ::= ID := <exp> ;
<stmt> ::= read ( <id list> ) ;
<stmt> ::= write ( <exp list> ) ;
```

```
void stmt() {
     tok  := scan();
     switch (tok) {
     case id:    match(id); match(assign); exp(); match(semi);
                 break;
     case read:  match(read); match(lparen); id_list();
                 match(rparen); match(semi); break;
     case write: match(write); match(lparen); exp_list();
                 match(rparen); match(semi); break;
     default:    putback(tok); syntax_error(); }
}
```

For repetition:

<stmt list> ::= <stmt> { <stmt> }

```
void stmt_list() {
     stmt();
     for ( ; ; ) {
       tok := scan();
       switch (tok) {
       case id:
       case read:
       case write: stmt();  break;
       default:    putback(tok); return; }
     }
}
```

For λ-rules:

```
A ::= a B
A ::= c D
A ::= λ
```

```
 void a() {
      tok  := scan();
      switch (tok) {
      case a:   match(a); b(); break;
      case c:   match(c); d(); break;
      default:  putback(tok); return;  /* not syntax error */ }
 }
```

More examples:

`<id list> ::= id { , id }`

```
void id_list() {
     match(id);
     tok := scan();
     while (tok == comma) {
           match(comma); match(id); tok := scan(); }
     putback(tok);
}
```

```
<exp> ::= <primary> { <add_op> <primary> }

 void expr() {
      primary();
      for (t := scan(); t == plus_op || t == minus_op;
           t = scan() ) {
           add_op(); primary(); }
      putback(t);  }
```

```
<expr list> ::= <expr> { , <expr> }
```

```
void expr_list() {
      expr();
      tok := scan();
      while (tok == comma) {
            match(comma);
            expr();
            tok := scan(); }
      putback(tok);
}
```

```
<addop> ::= plus | minus
```

```
void addop() {
      t := scan();
      if (t == plus_op || t == minus_op) match(t);
```

```
     else  syntax_error();
}
```

```
<primary> ::= ( <expr> )
<primary> ::= id
<primary> ::= int
```

```
void primary() {
     t := scan();
     switch(t) {
     case lparen:  match(lparen);
                   expr();
                   match(rparen);
                   break;
     case id:      match(id);
                   break;
     case int:     match(int);
                   break;
     default:      syntax_error(); }
}
```

Missing important issues

1. powerful and efficient parsers
2. symbol table (scope, type, address, etc.)
3. control structures (`if`, `while`, `case`, `function`, etc.)
4. complicated expressions
5. efficient object code
6. optimization (register allocation, transformation, etc.)
7. run-time storage management