

COMPILER CONSTRUCTION

A Simple Compiler

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University
Spring 2017

NCKU National Cheng Kung University



Chapter 2

Design of a Simple Compiler



Overview

- In this chapter, we will describe:
  - a simple programming language, **adding calculator** (***ac***), &
  - a simple compiler for ***ac***

- The ***ac*** compiler translates the ac program into the corresponding **desk calculator** (***dc***) program
  - which is a **stack-based** calculator
  - Stack-based languages commonly serve as targets of translation
    - They lend themselves to compact representation
    - Examples: Java → Java Virtual Machine, ActionScript → AVM2 for Flash media, printable documents → PostScript



Key Concepts for the *ac* Compiler

- Regular expressions
  - For basic symbols of ac
- Context-free grammar (CFG)
  - For syntax of ac
- Parse tree
- Scanning
- Parsing
- Abstract Syntax Trees
- Semantic Analysis
- Code generation

Token

- Regular Expression for **Token**
  - The actual input characters that correspond to each **terminal symbol** (called **token**) are specified by regular expression, which is covered in Ch.3
- For example:
  - **assign** symbol as a terminal, which appears in the input stream as "=" character
  - The terminal **id (identifier)** could be any alphabetic character
    - Note **f**, **i**, and **p** are reserved for special use in ***ac***
    - It is specified as [a-e] | [g-h]] | [j-o] | [q-z]



Token (Cont'd)

- Recognize tokens via regular expression rules
- Examples:
  - Reserved keywords: f, i, and p
  - '|' is used to specify the union of four sets for **id**
  - One or more decimal digits for **inum**

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

Figure 2.3: Formal definition of ac tokens.



Syntax for *ac*

- A **context-free grammar** (CFG) specifies the **syntax of a language**
  - CFG is used to describe the **acceptable statements** for the language
  - CFG is further described in Ch.4

- Now, we can view a CFG simply as a set of **productions** (or **rewriting rules**)

Order matters!!!

1	Prog	→	Dcls Stmts $
2	Dcls	→	Dcl Dcls
3		|	λ
4	Dcl	→	floatdcl id
5		|	intdcl id
6	Stmts	→	Stmt Stmts
7		|	λ
8	Stmt	→	id assign Val Expr
9		|	print id
10	Expr	→	plus Val Expr
11		|	minus Val Expr
12		|	λ
13	Val	→	id
14		|	inum
15		|	fnum

Figure 2.1: Context-free grammar for ac.



Syntax for *ac*: Stmt Example

Stmt$\rightarrow$ id assign Val Expr (1)

| print id (2)

- Stmt serves the same role in each of the productions separated by ' | '
- These productions indicate that a Stmt can be replaced by one of two strings of symbols

  (1) Stmt is rewritten by symbols that represent **assignment to an identifier**

  (2) Stmt is rewritten by symbols that **print an identifier's value**

Productions

- Two kinds of grammar symbols (in Productions):
  1. A **terminal** cannot be rewritten, a.k.a. **token**
     E.g., id, assign, and $ symbols have no productions
  2. A **non-terminal** can be rewritten by a production rule
     E.g., Val and Expr

- A special non-terminal is **start symbol**
  - which is usually the symbol on the **left-hand side (LHS)** of the grammar's first rule, e.g., **Prog**

- From the start symbol, we proceed to generate a program
  - by replacing (rewriting) a symbol on LHS with the **right-hand side (RHS)** of some production of that symbol



Productions (Cont'd)

- A special symbol **λ** denotes **empty** or **null string**
  - which indicates that there are no symbols on a production's RHS
- The special symbol **$** represents the end of the input stream

1	Prog	→	Dcls Stmts $
2	Dcls	→	Dcl Dcls
3		|	λ
4	Dcl	→	floatdcl id
5		|	intdcl id
6	Stmts	→	Stmt Stmts
7		|	λ
8	Stmt	→	id assign Val Expr
9		|	print id
10	Expr	→	plus Val Expr
11		|	minus Val Expr
12		|	λ
13	Val	→	id
14		|	inum
15		|	fnum

Figure 2.1: Context-free grammar for ac.



Productions (Cont'd)

- For an ac program, we continue **rewriting non-terminals by applying production rules** against CFG until none of non-terminals remains
  - Example shown in the next page
  - **Any string of terminals that can be produced in this manner is considered syntactically valid**
  - Any other string has a **syntax error** and would not be a legal program
- Notice that some productions in a grammar serve to generate an **unbounded list of symbols** from a nonterminal using **recursive rules**
  - E.g., **Stmts→Stmt Stmts** (Rule 6) allows an arbitrary number of Stmt symbols to be produced
  - The recursion is terminated by applying **Stmts**→λ (Rule 7)

Example of the Derivation of One *ac* Program

Program: f b i a   a = 5   b = a + 3.2   p b

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

1	Prog	→	Dcls Stmts $
2	Dcls	→	Dcl Dcls
3		|	λ
4	Dcl	→	floatdcl id
5		|	intdcl id
6	Stmts	→	Stmt Stmts
7		|	λ
8	Stmt	→	id assign Val Expr
9		|	print id
10	Expr	→	plus Val Expr
11		|	minus Val Expr
12		|	λ
13	Val	→	id
14		|	inum
15		|	fnum

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

Parse Tree of the *ac* Program

- The **derivation** shown textually in previous page can be represented as a derivation (or parse) tree shown here

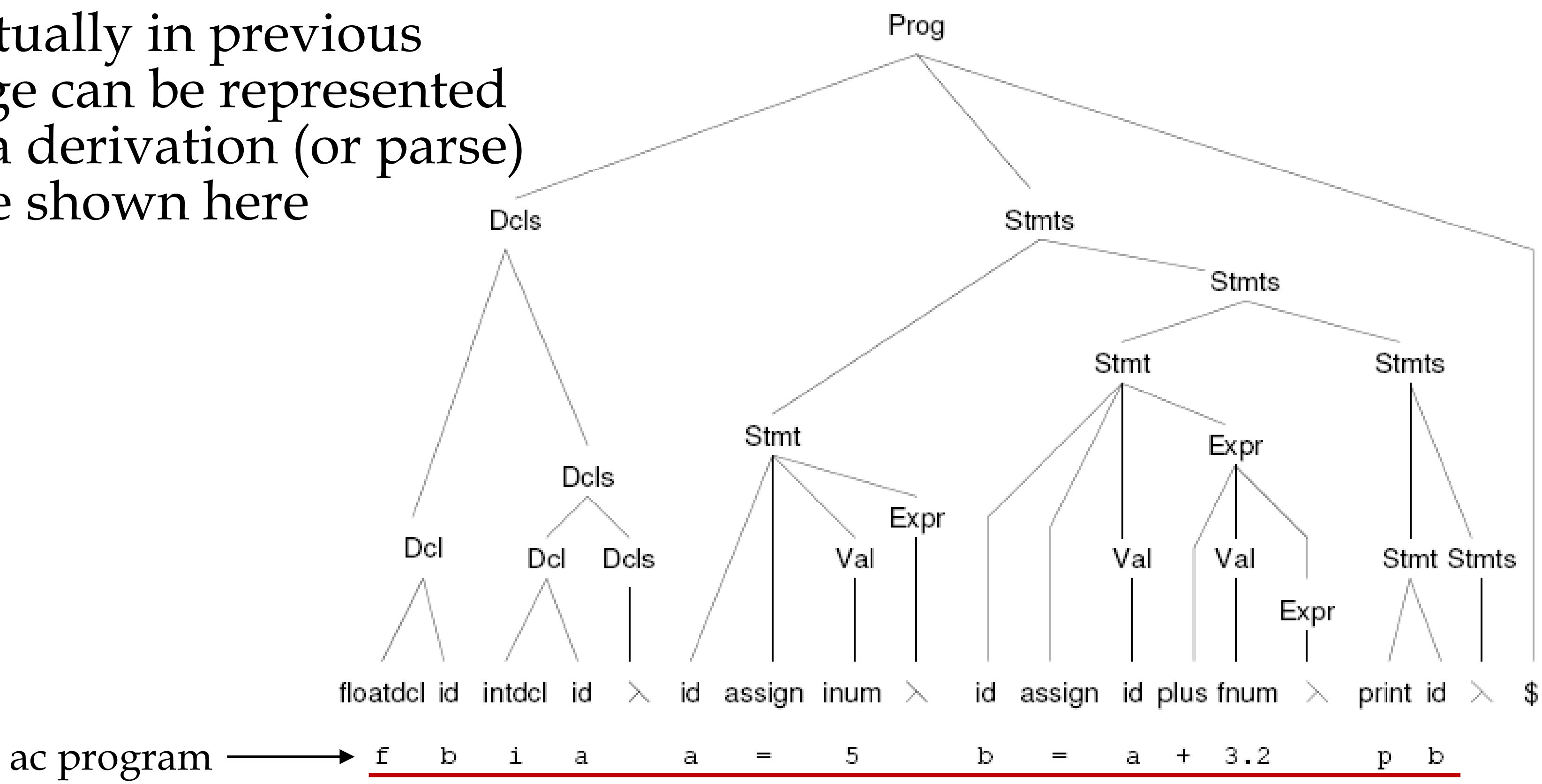

Figure 2.4: An ac program and its parse tree.



Parse Tree of the *ac* Program (Cont'd)

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

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

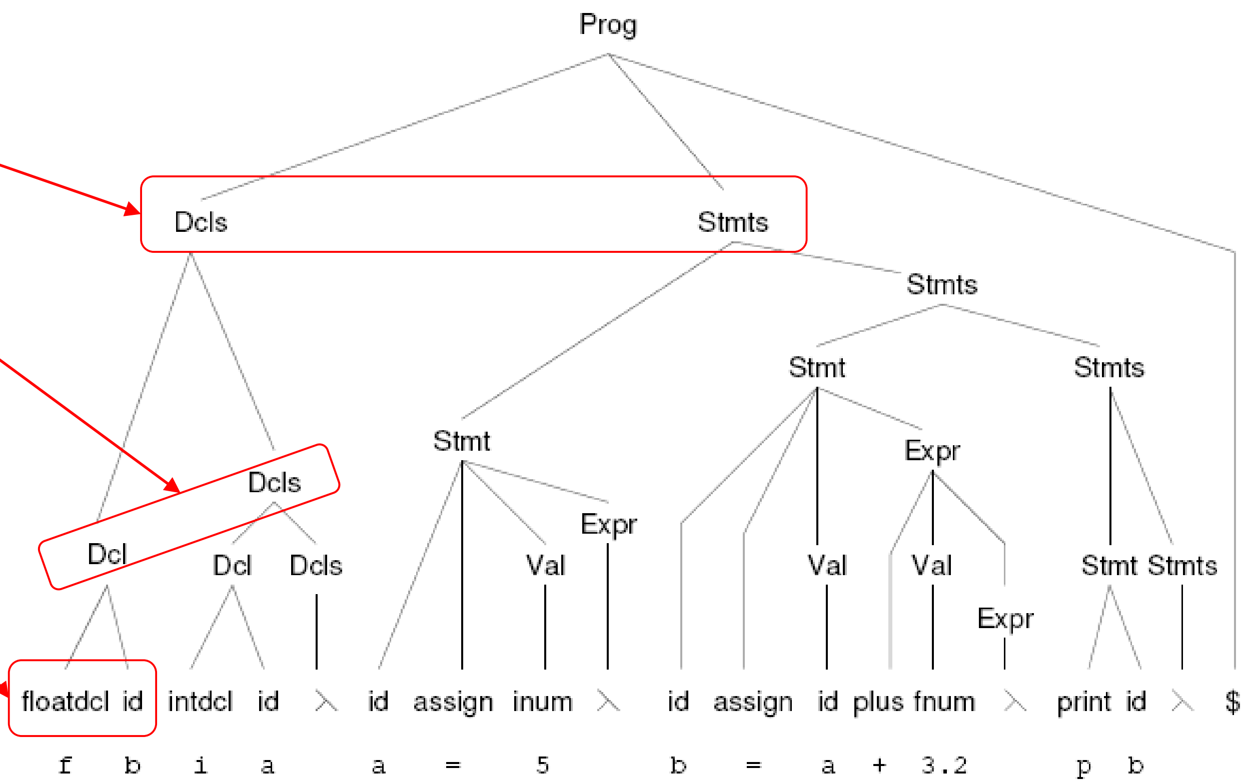

Figure 2.4: An ac program and its parse tree.



Phases of the *ac* Compiler

- Regular expressions
  - for basic symbols of ac
- Context-free grammar (CFG)
  - For syntax of ac
- Parse tree
- Scanning
- Parsing
- Abstract Syntax Trees
- Semantic Analysis
- Code generation



An Informal Definition of *ac*

- **Types**
  - Most programming languages offer a significant number of predefined data types, with the ability to extend existing types or specify new data types
  - In ac, there are only **two data types: integer and float**
  - An **integer type** is a sequence of decimal numerals, as found in most programming languages
  - A **float type** allows five fractional digits after the decimal point
- **Keywords**
  - Most programming languages have a number of **reserved keywords**, such as *if* and *while*, which would otherwise serve as variable names
  - In ac, there are three reserved keywords, each limited for simplicity to a single letter:
    - **f** (declares a float variable), **i** (declares an integer variable), and **p** (prints the value of a variable)
- **Variables**
  - Some programming languages insist that a variable is declared by specifying the variable's type prior to using the variable's name
  - The ac language offers only **23 possible variable names**, drawn from the lowercase Roman alphabet and excluding the three reserved keywords f, i, and p
  - Variables must be declared prior to using them

  

Scanning

- The **scanner** reads a source **ac** program as a text file and produces a stream of tokens

- Each token has the two components:
  1) **Token type** explains the token's category, e.g., **id**
  2) **Token value** provides the string value of the token, e.g., "**b**"

- Methods:
  - PEEK(): a single character of lookahead
  - ADVANCE(): the scanner is moved to the next input character (using advance), which suffices to determine the next token

Scanner for *ac*

- The figure shows a scanner that finds all tokens for *ac*
- Pseudo code for *ac* scanner
  - A big case-switch to choose the type for the character of the input stream
  - Referencing the token definition below

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	$(" \ ")^+$

```
function Scanner( ) returns Token
    while s.peek( ) = blank do call s.advance( )
    if s.EOF( )
    then ans.type ← $
    else
        if s.peek( ) ∈ { 0, 1, . . . , 9 }
        then ans ← ScanDigits( )
        else
            ch ← s.advance( )
            switch (ch)
                case { a, b, . . . , z } – { i, f, 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 LexicalError( )
    return (ans)
end
```

Figure 2.5: Scanner for the ac language. The variable s is an input stream of characters.



Scanner for *ac*

- Given the input stream:
  - i a a = 32 p a
- We show how scanner to get the token **i**

```
function Scanner( ) returns Token
   while s.peek( ) = blank do call s.advance( )
   if s.EOF( )
   then ans.type ← $
   else
      if s.peek( ) ∈ { 0, 1, . . . , 9 }
      then ans ← ScanDigits( )
      else
         ch ← s.advance( )
         switch (ch)
            case { a, b, . . . , z } − { i, f, 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 LexicalError( )
   return (ans)
end
```

Input stream:

i a a = 32 p a

s

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 2.5: Scanner for the ac language. The variable *s* is an input stream of characters.



Scanner for *ac*

- Given the input stream:
  - i a a = 32 p a
- We are at **i**, and we show how scanner
  - skip the **blank** and
  - recognizes **id** (a)

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	$(" \ ")^+$

```
function Scanner( ) returns Token
  while s.peek( ) = blank do call s.advance( )
  if s.EOF( )
  then ans.type ← $
  else
     if s.peek( ) ∈ { 0, 1, . . . , 9 }
     then ans ← ScanDigits( )
     else
        ch ← s.advance( )
        switch (ch)
           case { a, b, . . . , z } − { i, f, 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 LexicalError( )
  return (ans)
end
```

Input stream:

i a a = 32 p a

s s

Figure 2.5: Scanner for the ac language. The variable s is an input stream of characters.

Scanner for *ac* (Numbers)

- The figure shows scanning a number token
- Handle a number
  - **Type**:
    - Integer
    - Float
  - Its **Value**

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	$(" \ ")^+$

```
function ScanDigits( ) returns token
    tok.val ← " "
    while s.peek( ) ∈ { 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( ) ∈ { 0, 1, . . . , 9 } do
            tok.val ← tok.val + s.advance( )
    return (tok)
end
```

Figure 2.6: Finding inum or fnum tokens for the ac language.



Parsing

- **Parser**
  - processes tokens produced by the scanner,
  - determines the syntactic validity of the token stream, &
  - creates an **abstract syntax tree (AST)** for subsequent phases
- In most compilers,
  - the **grammar** serves not only to define the **syntax of a programming language**,
  - but also to guide the **automatic construction of a parser**



Parsing Technique

- **Recursive descent** is one simple parsing technique used in practical compilers
  - The name is taken from the mutually recursive parsing routines that, in effect, descend through a derivation tree
  - In recursive-descent parsing, each **nonterminal** in the grammar has **an associated parsing procedure**
    - that is responsible for determining if the token stream contains a sequence of tokens derivable from that nonterminal

  

Recursive-descent Parsing

- Each parsing procedure examines the next input token to predict which production to apply
- Example:
  The parsing procedure for “Stmt” shown in Fig. 2.7

Stmt → id assign Val Expr

Stmt → print id

```
procedure STMT( )
    if ts.PEEK( ) = id                              1
    then
        call MATCH(ts, id)                          2
        call MATCH(ts, assign)                      3
        call VAL( )                                 4
        call EXPR( )                                5
    else
        if ts.PEEK( ) = print                       6
        then
            call MATCH(ts, print)
            call MATCH(ts, id)
        else
            call ERROR( )                           7
end
```

Figure 2.7: Recursive-descent parsing procedure for Stmt. The variable *ts* is an input stream of tokens.



Recursive-descent Parsing (Cont'd)

- If **id** is the next input token, the parse proceeds with the production:

  Stmt → id assign Val Expr

  and the **predict set** for the production is {id}

- If **print** is the next, the parse proceeds with:

  Stmt → print id

  the **predict set** for the production is {print}

- If the next input token is neither **id** nor **print**, neither rule can be applied; it calls **ERROR**

```
procedure Stmt( )
    if ts.peek( ) = id                              (1)
    then
        call match(ts, id)                          (2)
        call match(ts, assign)                      (3)
        call Val( )                                 (4)
        call Expr( )                                (5)
    else
        if ts.peek( ) = print                       (6)
        then
            call match(ts, print)
            call match(ts, id)
        else
            call error( )                           (7)
end
```

Figure 2.7: Recursive-descent parsing procedure for Stmt. The variable *ts* is an input stream of tokens.



Recursive-descent Parsing (Cont'd)

- Computing the predict sets used in Stmt is relatively easy
  - since each production for Stmt **begins with a distinct terminal symbol: id or print**



Recursive-descent Parsing (Cont'd)

- Consider the productions for **Stmts**:

  Stmts → Stmt Stmts

  Stmts → λ

recursive-descent parser

  - The predict sets for Stmts can be computed by inspecting the following:
  - **Stmts → Stmt Stmts** begins with the non-terminal **Stmt**
    - →Find those symbols that predict **any** rule for Stmt
    - →Check for **id** or **print** as the next token
  - **Stmts → λ** derives no symbols
    - →Look for what symbol(s) could occur **following** such a production
    - →In this case, it is **$**

```
procedure Stmts( )
    if ts.peek( ) = id or ts.peek( ) = print                    8
    then
        call Stmt( )                                            9
        call Stmts( )                                           10
    else
        if ts.peek( ) = $                                       11
        then
            /*    do nothing for λ-production               */  12
        else   call error( )
end
```

Figure 2.8: Recursive-descent parsing procedure for Stmts.

The analysis required to compute predict sets in general is covered in Ch.4 and Ch.5



Recursive-descent Parsing (Cont'd)

- When a terminal such as id is encountered, a call to **MATCH(ts, id)** is placed into the code
  - The **MATCH** procedure simply consumes the expected token **id** if it is indeed the next token in the input stream
  - The next call to **MATCH(ts, assign)** tries to match **assign**
- The last two symbols in **Stmt→id assign Val Expr** are nonterminals
  - Later, calls to **Val()** and **Expr()** are performed
- You may try to think about how Stmts is parsed (Check Sec. 2.5.2)

```
procedure STMT( )
    if ts.PEEK( ) = id                          1
    then
        call MATCH(ts, id)                      2
        call MATCH(ts, assign)                  3
        call VAL( )                             4
        call EXPR( )                            5
    else
        if ts.PEEK( ) = print                   6
        then
            call MATCH(ts, print)
            call MATCH(ts, id)
        else
            call ERROR( )                       7
end
```

Figure 2.7: Recursive-descent parsing procedure for Stmt. The variable *ts* is an input stream of tokens.

```
procedure MATCH(ts, token)
    if ts.PEEK( ) = token
    then call ts.ADVANCE( )
    else call ERROR(Expected token)
end
```

Figure 5.5: Utility for matching tokens in an input stream.



Abstract Syntax Tree (AST)

- The **scanner** and **parser** together
  - accomplish the syntax analysis phase of a compiler
  - ensure that the compiler's input conforms to a language's token and CFG specifications
- **Parse tree**
  - might be considered as the structure that survives syntax analysis and is used for the remaining phases
  - However, such trees can be rather large and unnecessarily detailed, even for very simple grammars and inputs

  

Abstract Syntax Tree (AST) (Cont'd)

- Abstract syntax tree
  - contains the **essential information** from a parse tree,
  - but inessential punctuation and delimiters (braces, semicolons, parentheses, etc.) are not included
- It serves as **a representation of a program** for all phases after syntax analysis
  - It is actually the **data structure kept in memory** to represent the program code during compilation
  - Such phases may make use of information in the AST, decorate the AST with more information, or transform the AST

Abstract Syntax Tree (AST) (Cont'd)

- Consider the expression **a+3.2**
  - 8 nodes for parse tree (Fig. 2.4)
  - 3 nodes for AST (Fig. 2.9)
  - Check Sec.2.6 for rules to translate from parse tree to AST

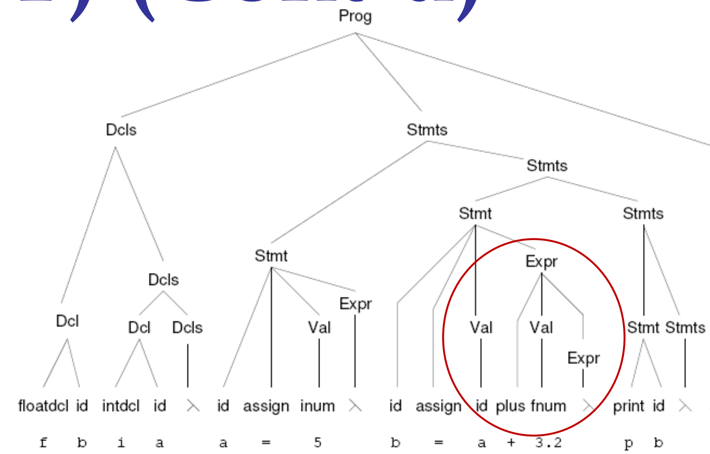

Figure 2.4: An ac program and its parse tree.

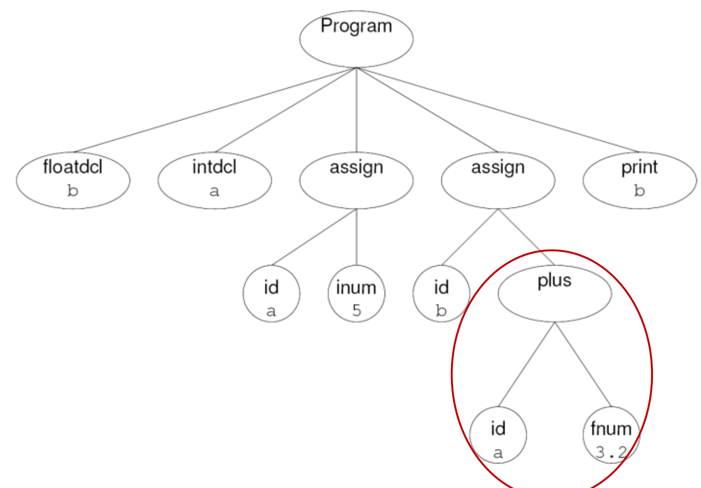

Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.



Semantic Analysis

- A catchall term for any post-parsing processing
  - that enforces aspects of a language's definition **that are not easily accommodated by syntax analysis**
- Examples (Two examples are introduced below):
  1. Declarations and name scopes are processed to construct a **symbol table**, so that declarations and uses of identifiers can be properly coordinated
  2. Language- and user-defined **types** are examined for consistency
  - Operations and storage references are processed so that type-dependent behavior can become explicit in the program representation



Semantic Analysis Example: Symbol Table

- Symbol-table construction is a semantic processing activity
  - that **traverses the AST to record all identifiers and their types** in a symbol table
- In ***ac***, identifiers must be declared prior to use
  - but this requirement is not easily enforced during syntax analysis

→ A separate pass to build the table

```
/*   Visitor methods   */
procedure VISIT( SymDeclaring n )
    if n.GETTYPE( ) = floatdcl
    then  call ENTERSYMBOL(n.GETID( ), float)
    else  call ENTERSYMBOL(n.GETID( ), integer)
end

/*   Symbol table management   */
procedure ENTERSYMBOL(name, type)
    if SymbolTable[name] = null
    then  SymbolTable[name] ← type
    else  call ERROR("duplicate declaration")
end

function LOOKUPSYMBOL(name) returns type
    return (SymbolTable[name])
end
```

Figure 2.10: Symbol table construction for ac.

  

Build Symbol Table for *ac*

- We traverse the AST
  - counting on the presence of a **symbol-declaring node** to trigger appropriate effects on the symbol table
  - Correspondingly, nodes such as **floatdcl** and **intdcl** implement an interface called **SymDeclaring**,
  - which implements a method to return the declared **identifier's type**
- In Fig. 2.10
  - **visit(SymDeclaring n)** shows the code to be applied at nodes that declare symbols
  - **EnterSymbol** checks that the given identifier has not been previously declared

```
/*   Visitor methods   */
procedure VISIT( SymDeclaring n )
    if n.GETTYPE( ) = floatdcl
    then  call ENTERSYMBOL(n.GETID( ), float)
    else  call ENTERSYMBOL(n.GETID( ), integer)
end

/*   Symbol table management   */
procedure ENTERSYMBOL( name, type )
    if SymbolTable[name] = null
    then  SymbolTable[name] ← type
    else  call ERROR( "duplicate declaration" )
end

function LOOKUPSYMBOL( name ) returns type
    return (SymbolTable[name])
end
```

Figure 2.10: Symbol table construction for ac.



Symbol Table for *ac*

- In ***ac***, a program can mention at most 23 distinct identifiers
- The built symbol table for ***ac***
  - with 23 entries
  - Each contains **symbol** and **type**
  - Type: integer, float, or unused (null
- On the contrary
  - most languages have infinite potential identifiers
  - The type information may include other attributes
  - such as, the identifier's **scope** of visibility, storage class, and protection properties

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.



Semantic Analysis Example: Type Checking

- Most programming language specifications include a **type hierarchy**
  - which compares the language's types in terms of their generality
- Example:
  A float type is considered **wider** (i.e., more general) than an integer (Java, C, and C++)
  - Every integer can be represented as a float
  - On the other hand, **narrowing** a float to an integer loses precision for some float values



Type Checking

- Most languages allow **automatic widening of type**
  - →E.g., an integer can be converted to a float without the programmer having to specify this conversion explicitly

```
int a;
float b, c;
c = a+b; //(automatically type casting for a)
```

- On the other hand, a float cannot become an integer in most languages
  - – unless the programmer explicitly calls for this conversion

Type Checking for *ac*

- Two types defined in ***ac***
  - I.e., **integer** and **float**, and
  - all identifiers must be type-declared in a program before they can be used
- Once the symbol table has been constructed,
  - the declared type of each identifier is known, and
  - the executable statements of the program can be **checked for type consistency**

→**Type checking**
Refers to the process that walks the AST **bottom-up**, from its leaves toward its root

Type Analysis for *ac*

- At each AST node, VISIT() is called:
  1. For **constants and symbol** references, the visitor methods simple set the supplied node's type based on the node's contents
  2. For **nodes that compute value**, such as plus and minus, the appropriate type is computed by calling the utility methods
  3. For an **assignment operation**, the visitor makes certain that the value computed by the second child is of the same type as the assigned identifier (the first child)

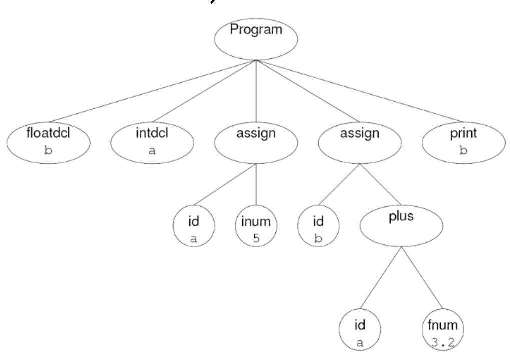

Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.

```
/*    Visitor methods                                              */
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 call error( "Illegal type conversion" )
    else
        if n.type = integer and t = float
        then
            /*    replace node n by convert-to-float of node n         */   (13)
        else   /* nothing needed */
end
```

Figure 2.12: Type analysis for ac.

Type Analysis for *ac*

- CONSISTENT() is responsible for reconciling the type of a pair of AST nodes with the following steps:
  1. The **GENERALIZE()** function determines the least general (i.e., simplest) type that encompasses its supplied pair of types
     For ac, if either type is float, then float is the appropriate type; otherwise, integer will do
  2. The **CONVERT()** procedure checks whether conversion is necessary, possible, or impossible
- An important consequence occurs at **Marker 13** in Figure 2.12
  - If conversion is attempted from integer to float, then the **AST is transformed to represent this type conversion explicitly**
  - Subsequent compiler passes (particularly code generation) can then assume a type-consistent AST in which all operations are explicit

```
/*     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  call ERROR( "Illegal type conversion" )
    else
        if n.type = integer and t = float
        then
            /*    replace node n by convert-to-float of node n           */ 13
        else   /* nothing needed */
end
```

Figure 2.12: Type analysis for ac.

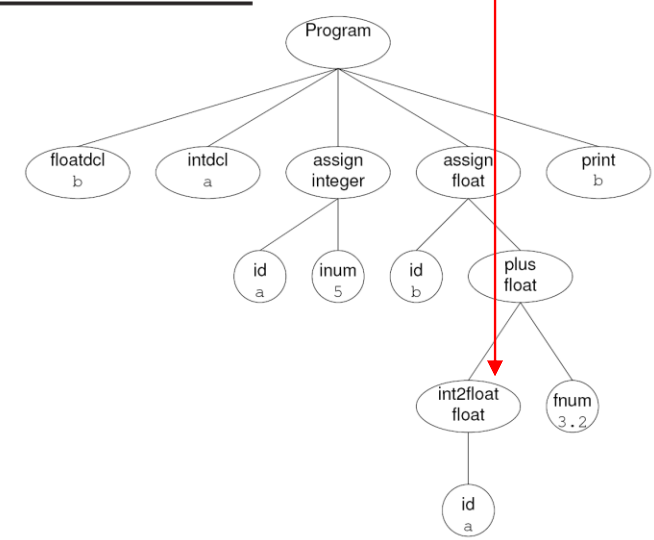

Figure 2.13: AST after semantic analysis.



Code Generation

- The final task undertaken by a compiler

  → The formulation of **target-machine instructions** that faithfully represent the semantics (i.e.,meaning) of the source program

  – Translation exercise of the textbook consists of generating source code that is suitable for the dc program, which is a simple calculator based on a stack machine model

- In a **stack machine**, most instructions receive their input from the contents at or near the **top of an operand stack**
  – The result of most instructions is pushed on the stack
  – Programming languages such as C# and Java are frequently translated into a portable, stack machine representation
  – Check Ch.11 and Ch.13



Code Generation (Cont'd)

- The AST was transformed and decorated with **type information** during semantic analysis
  - Such information is required for **selecting the proper instructions**
- The instruction set on most computers distinguishes between **float** and **integer** data types
  - ARM processors have the instructions
    - **VADD** for Floating-point Add
    - **VDIV** for Floating-point Divide
    - **ADD** for Integer Add
    - **SDIV** for Signed Divide



Generating Code for *ac*

- Traverse the AST
  - starting at its root and working toward its leaves
- The code generator is called recursively
  - to generate code for the left and right subtrees
  - The resulting values will be at top-of-stack

```
procedure VISIT( Assigning n )
    call CODEGEN( n.child2 )
    call EMIT( "s" )
    call EMIT( n.child1.id )
    call EMIT( "0 k" )                                    (14)
end
procedure VISIT( Computing n )
    call CODEGEN( n.child1 )
    call CODEGEN( n.child2 )
    call EMIT( n.operation )                              (15)
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" )                                     (16)
end
procedure VISIT( Converting n )
    call CODEGEN( n.child )
    call EMIT( "5 k" )                                    (17)
end
procedure VISIT( Consting n )
    call EMIT( n.val )
end
```

Figure 2.14: Code generation for ac

- VISIT(Computing n)
  - generates code for **plus** and **minus**
  - The appropriate operator is then emitted (**Marker 15**) to perform the operation



Generating Code for *ac* (Cont'd)

- VISIT(Assigning n)
  - causes the expression to be evaluated
  - Code is then emitted to **store** the value in the appropriate **dc register**
  - The calculator's **precision** is then **reset** to integer by setting the fractional precision to zero **Marker 14**

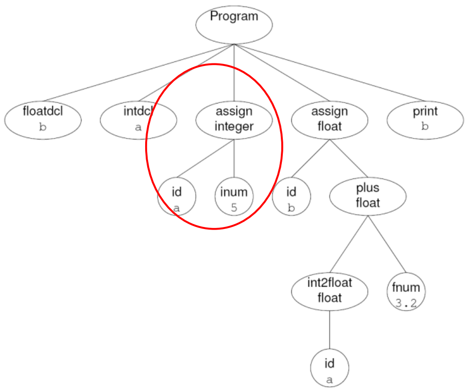

Figure 2.13: AST after semantic analysis.

Code
5
sa
0 k

```
procedure VISIT(Assigning n)
    call CODEGEN(n.child2)      5
    call EMIT("s")              s
    call EMIT(n.child1.id)      a
    call EMIT("0 k")            0k
end
```

Figure 2.14: Code generation for ac

(14)



Generating Code for *ac*

- VISIT(Computing n)
  - generates code for **plus** and **minus**
  - The appropriate operator is then emitted (**Marker 15**) to perform the operation

```
procedure visit( Assigning n )
    call CodeGen( n.child2 )
    call Emit( "s" )
    call Emit( n.child1.id )
    call Emit( "0 k" )                                  14
end
procedure visit( Computing n )
    call CodeGen( n.child1 )
    call CodeGen( n.child2 )
    call Emit( n.operation )                            15
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" )                                   16
end
procedure visit( Converting n )
    call CodeGen( n.child )
    call Emit( "5 k" )                                  17
end
procedure visit( Consting n )
    call Emit( n.val )
end
```

Figure 2.14: Code generation for ac



Generating Code for *ac* (Cont'd)

- **VISIT(SymReferencing n)**
  - causes a value to be retrieved from the appropriate **dc register** and **pushed onto the stack**
  - Push register
    → Load (l) symbol (a) to dc register
    → `**la**'

```
procedure visit( Assigning n )
    call CodeGen( n.child2 )
    call Emit( "s" )
    call Emit( n.child1.id )
    call Emit( "0 k" )                                   (14)
end
procedure visit( Computing n )
    call CodeGen( n.child1 )
    call CodeGen( n.child2 )
    call Emit( n.operation )                              (15)
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" )                                     (16)
end
procedure visit( Converting n )
    call CodeGen( n.child )
    call Emit( "5 k" )                                    (17)
end
procedure visit( Consting n )
    call Emit( n.val )
end
```

Figure 2.14: Code generation for ac

  

Generating Code for *ac* (Cont'd)

- VISIT(Printing n)
  - is tricky because dc does not discard the value on top-of-stack after it is printed
  - The instruction sequence `**si**' is generated at **Marker 16**,
  - thereby popping the stack and storing the value in dc's **i** register
  - Conveniently, the ac language precludes a program from using this register because the **i** token is reserved for spelling the terminal symbol integer

```
procedure VISIT( Assigning n )
    call CODEGEN( n.child2 )
    call EMIT( "s" )
    call EMIT( n.child1.id )
    call EMIT( "0 k" )                                   (14)
end
procedure VISIT( Computing n )
    call CODEGEN( n.child1 )
    call CODEGEN( n.child2 )
    call EMIT( n.operation )                             (15)
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" )                                    (16)
end
procedure VISIT( Converting n )
    call CODEGEN( n.child )
    call EMIT( "5 k" )                                   (17)
end
procedure VISIT( Consting n )
    call EMIT( n.val )
end
```

Figure 2.14: Code generation for ac



Generating Code for *ac* (Cont'd)

- VISIT(Converting n)
  - causes a change of type from integer to float at **Marker 17,**
  - which accomplished by setting dc's precision to five fractional decimal digits → `**5 k**'

```
procedure VISIT( Assigning n )
    call CODEGEN( n.child2 )
    call EMIT( "s" )
    call EMIT( n.child1.id )
    call EMIT( "0 k" )                          (14)
end
procedure VISIT( Computing n )
    call CODEGEN( n.child1 )
    call CODEGEN( n.child2 )
    call EMIT( n.operation )                    (15)
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" )                           (16)
end
procedure VISIT( Converting n )
    call CODEGEN( n.child )
    call EMIT( "5 k" )                          (17)
end
procedure VISIT( Consting n )
    call EMIT( n.val )
end
```

Figure 2.14: Code generation for ac



Generating Code for *ac* (Cont'd)

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.

  

QUESTIONS?