COMPILER CONSTRUCTION

Principles and Practice

Kenneth C. Louden

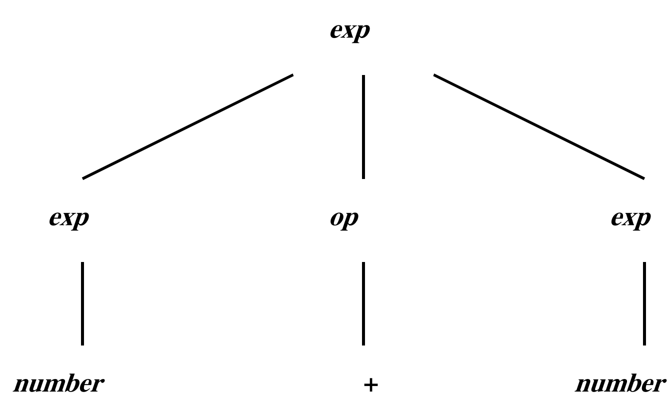
4. Top-Down Parsing

PART ONE

The outline of this chapter

Concept of Top-Down Parsing(1)

- It parses an input string of tokens by *tracing out the steps* in a leftmost derivation.
  - And the implied traversal of the parse tree is a preorder traversal and, thus, occurs from the root to the leaves.
- The example:
  - number + number, and corresponds to the parse tree



Concept of Top-Down Parsing(2)

The example: number + number, and corresponds to the parse tree

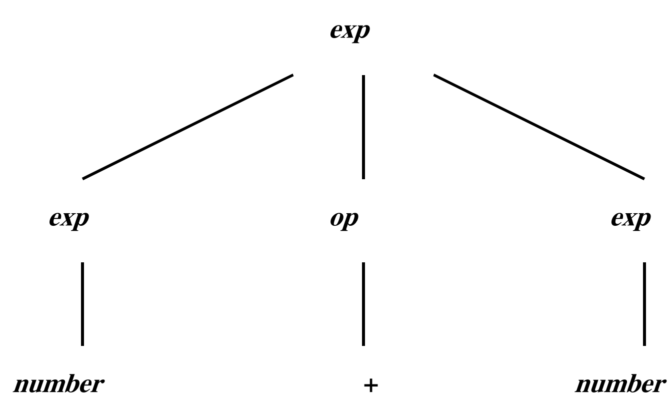
- The above parse tree is corresponds to the leftmost derivations:

(1) *exp* => *exp op exp*

(2) => ***number*** *op exp*

(3) => ***number*** + *exp*

(4) => ***number*** + ***number***



Two forms of Top-Down Parsers

- *Predictive parsers*:
  - attempts to predict the next construction in the input string using one or more look-ahead tokens
- *Backtracking parsers*:
  - try different possibilities for a parse of the input, backing up an arbitrary amount in the input if one possibility fails.
  - It is more powerful but much slower, unsuitable for practical compilers.

Two kinds of Top-Down parsing algorithms

- *Recursive-descent parsing*:
  - is quite versatile and suitable for a handwritten parser.
- *LL(1) parsing*:
  - The first “L” refers to the fact that it processes the input from left to right;
  - The second “L” refers to the fact that it traces out a leftmost derivation for the input string;
  - The number “1” means that it uses only one symbol of input to predict the direction of the parse.

Other Contents

- *Look-Ahead Sets*
  - *First and Follow sets*: are required by both recursive-descent parsing and LL(1) parsing.
- *A TINY Parser*
  - It is constructed by recursive-descent parsing algorithm.
- *Error recovery methods*
  - The error recovery methods used in Top-Down parsing will be described.

Contents

4.1 Top-Down Parsing by Recursive-Descent

4.1.1 The Basic Method of Recursive-Descent

The idea of Recursive-Descent Parsing

- Viewing the grammar rule for a non-terminal A as a definition for a procedure to recognize an A
- The right-hand side of the grammar for A specifies the structure of the code for this procedure
- The Expression Grammar:

$$exp \rightarrow exp\ addop\ term \mid term$$

$$addop \rightarrow + \mid -$$

$$term \rightarrow term\ mulop\ factor \mid factor$$

$$mulop \rightarrow *$$

$$factor \rightarrow (exp) \mid \textbf{number}$$

A recursive-descent procedure that recognizes a *factor*

procedure *factor*
begin
  case token of
  (**:** match(**(**);
    exp;
    match(**)**);
  number:
    match (**number**);
  else error;
  end case;
end factor

- **The token keeps the current next token in the input (one symbol of look-ahead)**
- **The Match procedure matches the current next token with its parameters, advances the input if it succeeds, and declares error if it does not**

Match Procedure

- The Match procedure matches the current next token with its parameters,
  - advances the input if it succeeds, and declares error if it does not

procedure match(expectedToken);
begin
  if token = expectedToken then
    getToken;
  else
    error;
  end if;
end match

Requiring the Use of EBNF

- The corresponding EBNF is

  $exp \rightarrow term\ \{\ addop\ term\ \}$

  $addop \rightarrow$ **+ | -**

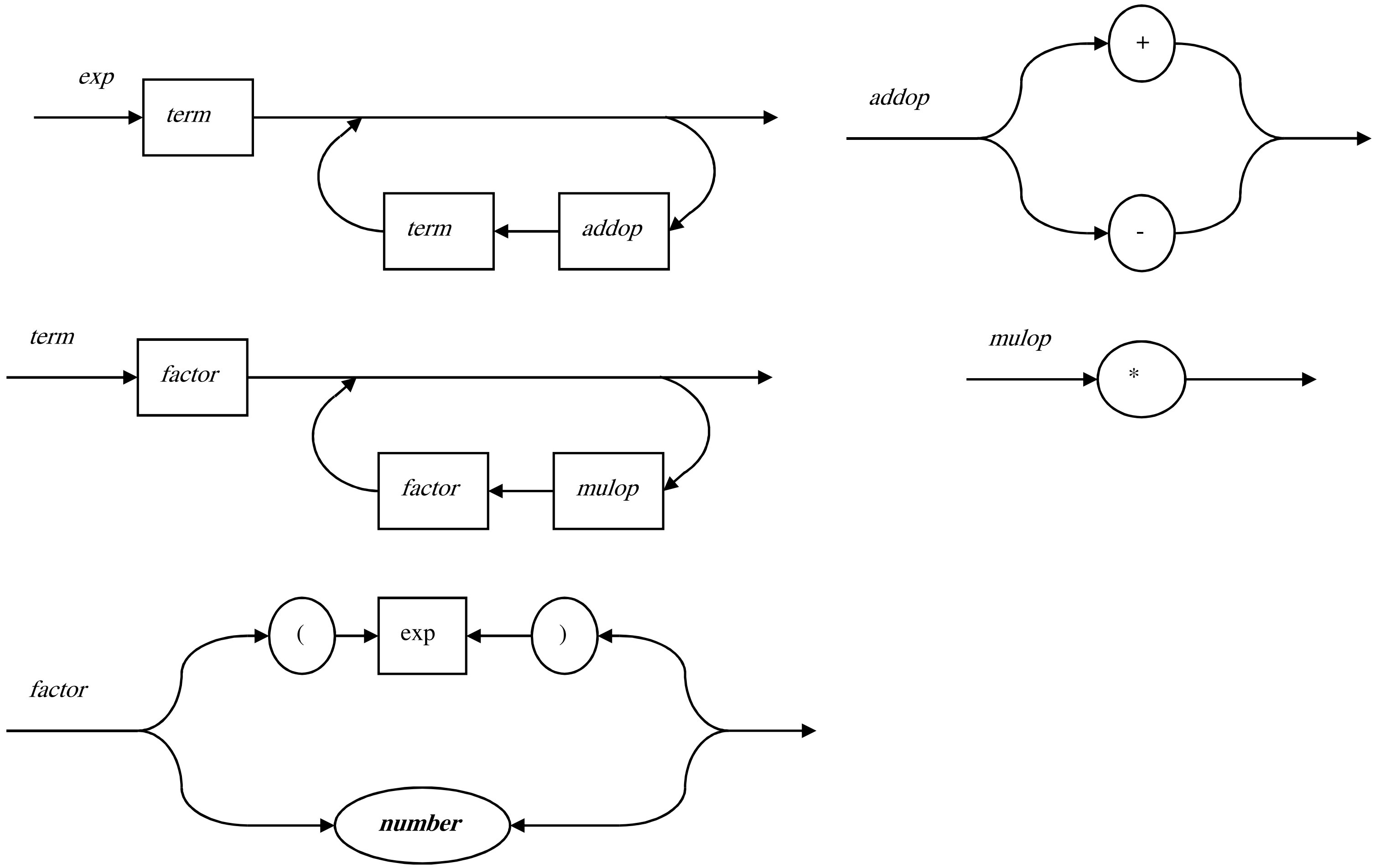
  $term \rightarrow\ factor\ \{\ mulop\ factor\ \}$

  $mulop \rightarrow\ *$

  $factor \rightarrow\ (\ exp\ )$ | ***number***

- Writing recursive-decent procedure for the remaining rules in the expression grammar is not as easy for factor

The corresponding syntax diagrams



4.1.2 Repetition and Choice: Using EBNF

An Example

```
procedure ifstmt;
    begin
      match( if );
      match( ( );
      exp;
      match( ) );
      statement;
      if token = else then
        match (else);
        statement;
      end if;
    end ifstmt;
```

- The grammar rule for an if-statement:

  If-stmt → if (*exp*) *statement*

  | if (*exp*) *statement* else *statement*

- Could not immediately distinguish the two choices because the both start with the token ***if***
- Put off the decision until we see the token **else** in the input

The EBNF of the if-statement

- If-stmt → if (*exp*) *statement* [else *statement*]
  - Square brackets of the EBNF are translated into a test in the code for *ifstmt*.
  - **if** token = else then
  - match (else);
  - statement;
  - **endif**;
- Notes
  - EBNF notation is designed to mirror closely the actual code of a recursive-descent parser,
  - So a grammar should always be translated into EBNF if recursive-descent is to be used.
- It is natural to write a parser that matches each else token as soon as it is encountered in the input

EBNF for Simple Arithmetic Grammar(1)

- The EBNF rule for $exp \rightarrow exp$ addop $term | term$
  - $exp \rightarrow term$ {*addop term*}
  - Where, the curly bracket expressing repetition can be translated into the code for a loop:

```
procedure exp;
begin
  term;
  while token = + or token = - do
    match(token);
    term;
  end while;
end exp;
```

EBNF for Simple Arithmetic Grammar(2)

- The EBNF rule for term:
  - *term → factor {mulop factor}*

Becomes the code

```
procedure term;
begin
  factor;
  while token = * do
      match(token);
      factor;
  end while;
end exp;
```

Left associatively implied by the curly bracket

- The left associatively implied by the curly bracket (and explicit in the original BNF) can still be maintained within this code

```
function exp: integer;
    var temp: integer;
    begin
    temp:=term;
    while token=+ or token = - do
case token of
+ : match(+);
   temp:=temp+term;
-:match(-);
   temp:=temp-term;
end case;
               end while;
               return temp;
end exp;
```

A working simple calculator in C code(1)

```
/*Simple integer arithmetic calculator according to the EBNF;
  <exp> → <term> { <addop> <term>}
  <addop> → + | -
  <term>→   <factor>  { <mulop> <factor> }
  <mulop> →  *
  <factor> →  ( <exp> ) | Number
  inputs a line of text from stdin
  outputs “error” or the result.
*/
```

A working simple calculator in C code(2)

```
#include <stdio.h>
#include <stdio.h>
char token; /* global token variable */
/*function prototype for recursive calls*/
int exp(void);
int term(void);
int factor(void);
void error(void)
{fprint(stderr, “error\n”);
 exit(1);
}
```

A working simple calculator in C code(3)

```
void match(char expectedToken)
{if (token==expectedToken) token=getchar();
 else error();
}
main()
{    int result;
     token=getchar();/*load token with first character for lookahead*/
result=exp();
if (token==’\n’) /*check for end of line*/
          printf(“Result = %d\n”, result);
else error(); /*extraneous chars on line*/
return 0;
}
```

A working simple calculator in C code(4)

```
int exp(void)
{ int temp =term();
  while ((token=='+') || token=='-'))
switch (token) {
case '+': match ('+');
        temp+=term();
        break;
case '-': match ('-');
        temp-=term();
        break;
}
return temp;
}
```

A working simple calculator in C code(5)

```
int term(void)
{int temp=factor();
 while (token==’*’){
     match(‘*’);
     temp*=factor();
     }
 return temp;
}
```

A working simple calculator in C code(5)

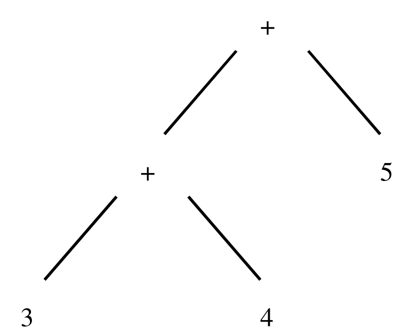
```
int factor(void)
{ int temp;
  if (token==’(‘) {
    match (‘(‘);
    temp = exp();
    match(‘)’);
    }
  else if (isdigit(token)){
    ungetc(token,stdin);
    scanf(“%d”,&temp);
    token = getchar();
    }
   else error();
   return temp;
  }
```

Some Notes

- The method of turning grammar rule in EBNF into code is quite powerful.
- There are a few pitfalls, and care must be taken in scheduling the actions within the code.
- In the previous pseudo-code for exp:
  - (1) The match of operation should be before repeated calls to term;
  - (2) The global token variable must be set before the parse begins;
  - (3) The getToken must be called just after a successful test of a token

Construction of the syntax tree

- The expression: 3+4+5



The pseudo-code for constructing the syntax tree(1)

```
function exp : syntaxTree;
    var temp, newtemp: syntaxTree;
    begin
      temp:=term;
      while token=+ or token = - do
            case token of
            + : match(+);
            newtemp:=makeOpNode(+);
            leftChild(newtemp):=temp;
            rightChild(newtemp):=term;
            temp=newtemp;
```

The pseudo-code for constructing the syntax tree(2)

```
        -:match(-);
        newtemp:=makeOpNode(-);
        leftChild(newtemp):=temp;
        rightChild(newtemp):=term;
        temp=newtemp;
end case;
            end while;
            return temp;
end exp;
```

A simpler one

```
function exp : syntaxTree;
        var temp, newtemp: syntaxTree;
        begin
           temp:=term;
           while token=+ or token = - do
                         newtemp:=makeOpNode(token);
                         match(token);
                         leftChild(newtemp):=temp;
                         rightChild(newtemp):=term;
                         temp=newtemp;
                end while;
                return temp;
end exp;
```

The pseudo-code for the if-statement procedure (1)

```
function ifstatement: syntaxTree;
  var temp:syntaxTree;
  begin
    match(if);
    match(();
    temp:= makeStmtNode(if);
    testChild(temp):=exp;
    match());
    thenChild(temp):=statement;
```

The pseudo-code for the if-statement procedure (2)

```
if token= else then
  match(else);
  elseChild(temp):=statement;
else
  ElseChild(temp):=nil;
end if;
end ifstatement
```

4.1.3 Further Decision Problems

More formal methods to deal with complex situation

(1) It may be difficult to convert a grammar in BNF into EBNF form;

(2) It is difficult to decide when to use the choice A $\rightarrow \alpha$ and the choice A $\rightarrow \beta$;

if both α and β begin with non-terminals. Such a decision problem requires the computation of the **First Sets**.

More formal methods to deal with complex situation

(3) It may be necessary to know what token legally coming from the non-terminal A, in writing the code for an ε-production: A→ε. Such tokens indicate A may disappear at this point in the parse. This set is called the **Follow Set** of A.

(4) It requires computing the First and Follow sets in order to detect the errors as early as possible. Such as ")3-2)", the parse will descend from exp to term to factor before an error is reported.

4.2 LL(1) PARSING

4.2.1 The Basic Method of LL(1) Parsing

Main idea

- LL(1) Parsing uses an explicit stack rather than recursive calls to perform a parse
- An example:
  - a simple grammar for the strings of balanced parentheses:

    $S \to (S)\ S \mid \varepsilon$
- The following table shows the actions of a top-down parser given this grammar and the string ( )

Table of Actions

Steps	Parsing Stack	Input	Action
1	$S	( ) $	$S \rightarrow (S)\ S$
2	$S)S(	( ) $	match
3	$S)S	)$	$S \rightarrow \varepsilon$
4	$S)	)$	match
5	$S	$	$S \rightarrow \varepsilon$
6	$	$	accept

General Schematic

- A top-down parser begins by pushing the start symbol onto the stack
- It accepts an input string if, after a series of actions, the stack and the input become empty
- A general schematic for a successful top-down parse:

$ StartSymbol   Inputstring$

…   …   //one of the two actions

…   …   //one of the two actions

$ $ accept

Two Actions

- **The two actions**
  - Generate: Replace a non-terminal A at the top of the stack by a string α(in reverse) using a grammar rule A →α, and
  - Match: Match a token on top of the stack with the next input token.
- The list of generating actions in the above table:

$$
\begin{aligned}
S &\Rightarrow (S)S \quad [S\to(S)\ S]\\
&\Rightarrow (\ )S \quad [S\to\varepsilon]\\
&\Rightarrow (\ ) \quad [S\to\varepsilon]
\end{aligned}
$$

- Which corresponds precisely to the steps in a leftmost derivation of string ( ).
- This is the characteristic of top-down parsing.

4.2.2 The LL(1) Parsing Table and Algorithm

4.2.3 Left Recursion Removal and Left Factoring

4.2.4 Syntax Tree Construction in LL(1) Parsing

End of Part One

THANKS