COMPILER CONSTRUCTION

Principles and Practice

Kenneth C. Louden

4. Top-Down Parsing

PART TWO

Contents

4.1 Top-Down Parsing by Recursive-Descent

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:

    $S \rightarrow (S)\ S \mid \varepsilon$
- 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→(S) S
2	$S)S(	( ) $	match
3	$S)S	)$	S→ ε
4	$S)	)$	match
5	$S	$	S→ ε
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 $A \rightarrow \alpha$, and
  - Match: Match a token on top of the stack with the next input token.
- The list of generating actions in the above table:

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

- 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

Purpose and Example of LL(1) Table

- **Purpose of the LL(1) Parsing Table:**
  - **To express the possible rule choices for a non-terminal A when the A is at the top of parsing stack based on the current input token (the look-ahead).**
- **The LL(1) Parsing table for the following simple grammar:**

$$S \to (S)\ S \mid \varepsilon$$

M[N,T]	(	)	$
S	$S \to (S)\ S$	$S \to \varepsilon$	$S \to \varepsilon$

The General Definition of Table

- **The table is a two-dimensional array indexed by non-terminals and terminals**
- **Containing production choices to use at the appropriate parsing step called M[N,T]**
  - **N is the set of non-terminals of the grammar**
  - **T is the set of terminals or tokens (including $)**
- **Any entrances remaining empty**
  - **Representing potential errors**

Table-Constructing Rule

- The table-constructing rule
  - If $A \rightarrow \alpha$ is a production choice, and there is a derivation $\alpha \Rightarrow^* a\beta$, where a is a token, then add $A \rightarrow \alpha$ to the table entry M[A,a];
  - If $A \rightarrow \alpha$ is a production choice, and there are derivations $\alpha \Rightarrow^* \varepsilon$ and $S\$ \Rightarrow^* \beta A a \gamma$, where S is the start symbol and a is a token (or \$), then add $A \rightarrow \alpha$ to the table entry M[A,a];

A Table-Constructing Case

- **The constructing-process of the following table**
  - **For the production : S→(S)S, α=(S)S, where a=(, this choice will be added to the entry M[S, (].**
  - **Since: S=>(S)S$， rule 2 applied with A = S, *a* =), so add the choice S→ε to M[S,)]**
  - **Since S$=>* S$, S→ε is also added to M[S, $].**

M[N,T]	(	)	$
S	S→(S) S	S→ ε	S→ ε

Properties of LL(1) Grammar

- **Definition of LL(1) Grammar**
  - **A grammar is an LL(1) grammar if the associated LL(1) parsing table has at most on production in each table entry**
- **An LL(1) grammar cannot be ambiguous**

A Parsing Algorithm Using the LL(1) Parsing Table

(* assumes $ marks the bottom of the stack and the end of the input *)

push the start symbol onto the top the parsing stack;

while the top of the parsing stack ≠ $ and
the next input token ≠ $ do

if *the top of the parsing stack is terminal a and the next input token = a*
then (* match *)
pop the parsing stack;
advance the input;

A Parsing Algorithm Using the LL(1) Parsing Table

else if *the top of the parsing stack is non-terminal A*
and *the next input token is terminal a*
and *parsing table entry M[A,a] contains production A→*

X1X2…Xn
then (* generate *)
pop the parsing stack;
for i:=n downto 1 do
push Xi onto the parsing stack;
else error;
if *the top of the parsing stack = $*
and the next input token = $
then accept
else error.

Example: If-Statements

- **The LL(1) parsing table for simplified grammar of if-statements:**

Statement → if-stmt | other

If-stmt → if (exp) statement else-part

Else-part → else statement | ε

Exp → 0 | 1

M[N,T]	If	Other	Else	0	1	$
Statement	Statement → if-stmt	Statement → other				
If-stmt	If-stmt → if (exp) statement else-part					
Else-part			Else-part → else statement Else-part → ε			Else-part → ε
Exp				Exp → 0	Exp → 1	

Notice for Example: If-Statement

- **The entry M[else-part, else] contains two entries, i.e. *the dangling else ambiguity.***
- **Disambiguating rule: *always prefer the rule that generates the current look-ahead token over any other, and thus the production***

  Else-part → else statement

  ove

  Else-part →ε

- **With this modification, the above table will become unambiguous**
  - **The grammar can be parsed as if it were an LL(1) grammar**

The parsing based LL(1) Table

- The parsing actions for the string:

  If (0) if (1) other else other

- (for conciseness, statement= S, if-stmt=I, else-part=L, exp=E, if=I, else=e, other=o)

Steps	Parsing Stack	Input	Action
1	$S	i(0)i(1)oeo$	S→I
2	$I	i(0)i(1)oeo$	I→i(E)SL
3	$LS)E(i	i(0)i(1)oeo$	Match
4	$ LS)E(	(0)i(1)oeo $	Match
5	$ LS)E	0)i(1)oeo $	E→o
			Match
			Match
			S→I
			I→i(E)SL
			Match
			Match
			E→1
			Match
			match
			S→o
			match
			L→eS
			Match
			S→o
			match
			L→ ε
22	$	$	accept

4.2.3 Left Recursion Removal and Left Factoring

Repetition and Choice Problem

- **Repetition and choice in LL(1) parsing suffer from similar problems** to be those that occur in recursive-descent parsing
  - and for that reason we have not yet been able to give an LL(1) parsing table for the simple arithmetic expression grammar of previous sections.
- Solve these problems for **recursive-descent by using EBNF notation**
  - We cannot apply the same ideas to LL(1) parsing;
  - instead, we must rewrite the grammar within the BNF notation into a form that the LL(1) parsing algorithm can accept.

Two standard techniques for Repetition and Choice

- **Left Recursion removal**

  exp → exp addop term | term

  (in recursive-descent parsing, EBNF: exp→ term {addop term})

- **Left Factoring**

  If-stmt → if (exp) statement

  | if (exp) statement else statement

  (in recursive-descent parsing, EBNF:

  if-stmt→ if (exp) statement [else statement])

Left Recursion Removal

- **Left recursion is commonly used to make operations left associative, as in the simple expression grammar, where**

  exp → exp addop term | term

- **Immediate left recursion:**

  The left recursion occurs only within the production of a single non-terminal.

  exp → exp + term | exp - term |term

- **Indirect left recursion:**

  Never occur in actual programming language grammars, but be included for completeness.

  A → Bb |…

  B → Aa |…

CASE 1: Simple Immediate Left Recursion

- **$A \rightarrow A\alpha \mid \beta$**

  Where, α and β are strings of terminals and non-terminals;

  β does not begin with A.

- **The grammar will generate the strings of the form.** $\beta\alpha^n$

- ***We rewrite this grammar rule into two rules:***

  $A \rightarrow \beta A'$

  To generate β first;

  $A' \rightarrow \alpha A' \mid \varepsilon$

  To generate the repetitions of α, using right recursion.

Example

- **exp → exp addop term | term**
- **To rewrite this grammar to remove left recursion, we obtain**

  exp → term exp'

  exp' → addop term exp' | ε

CASE2: General Immediate Left Recursion

$A \rightarrow A\alpha 1 | A\alpha 2 | \ldots | A\alpha n | \beta 1 | \beta 2 | \ldots | \beta m$

Where none of $\beta 1, \ldots, \beta m$ begin with A.

The solution is similar to the simple case:

$$A \rightarrow \beta 1 A' | \beta 2 A' | \ldots | \beta m A'$$

$$A' \rightarrow \alpha 1 A' | \alpha 2 A' | \ldots | \alpha n A' | \varepsilon$$

Example

- **exp → exp + term | exp - term |term**

- **Remove the left recursion as follows:**

  exp → term exp'

  exp' → + term exp' | - term exp' |ε

CASE3: General Left Recursion

- **Grammars with no ε-productions and no cycles**

  (1) A cycle is a derivation of at least one step that begins and ends with same non-terminal:

  $$A => \alpha => A$$

  (2) Programming language grammars do have ε-productions, but usually in very restricted forms.

Algorithm for General Left Recursion Removal

For i:=1 to m do

For j:=1 to i-1 do

Replace each grammar rule choice of the form

$A_i \rightarrow A_j\beta$ by the rule

$A_i \rightarrow \alpha_1\beta | \alpha_2\beta | \ldots | \alpha_k\beta$,

where $A_j \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_k$ is the current rule for A_j.

Explanation:

(1) Picking an arbitrary order for all non-terminals, say, $A_1, \ldots, A_m$;

(2) Eliminates all rules of the form $A_i \rightarrow A_j\gamma$ with $j \leq i$;

(3) Every step in such a loop would only increase the index, and thus the original index cannot be reached again.

Example

Consider the following grammar:

A→Ba| Aa| c

B→Bb| Ab| d

Where, A1=A, A2=B and m=2

(1) When i=1, the inner loop does not execute, So only to remove the immediate left recursion of A

A→BaA'| c A'

A'→aA'| ε

B→Bb| Ab| d

Example

(2) when i=2, the inner loop execute once, with j=1;To eliminate the rule B→Ab by replacing A with it choices

A→BaA'| c A'

A'→aA'| ε

B→Bb| BaA'b|cAb| d

(3) We remove the immediate left recursion of B to obtain

A→BaA'| c A'

A'→aA'| ε

B→|cA'bB'| dB'

B→bB' |aA'bB'|ε

Now, the grammar has no left recursion.

Notice

- Left recursion removal not changes the language, but
  - Change the grammar and the parse tree
- This change causes a complication for the parser

Example

Simple arithmetic expression grammar

expr → expr addop term|term

addop → +|-

term → term mulop factor | factor

mulop →*

factor →(expr) | number

After removal of the left recursion

exp → term exp'

exp'→ addop term exp'|ε

addop → + -

term → factor term'

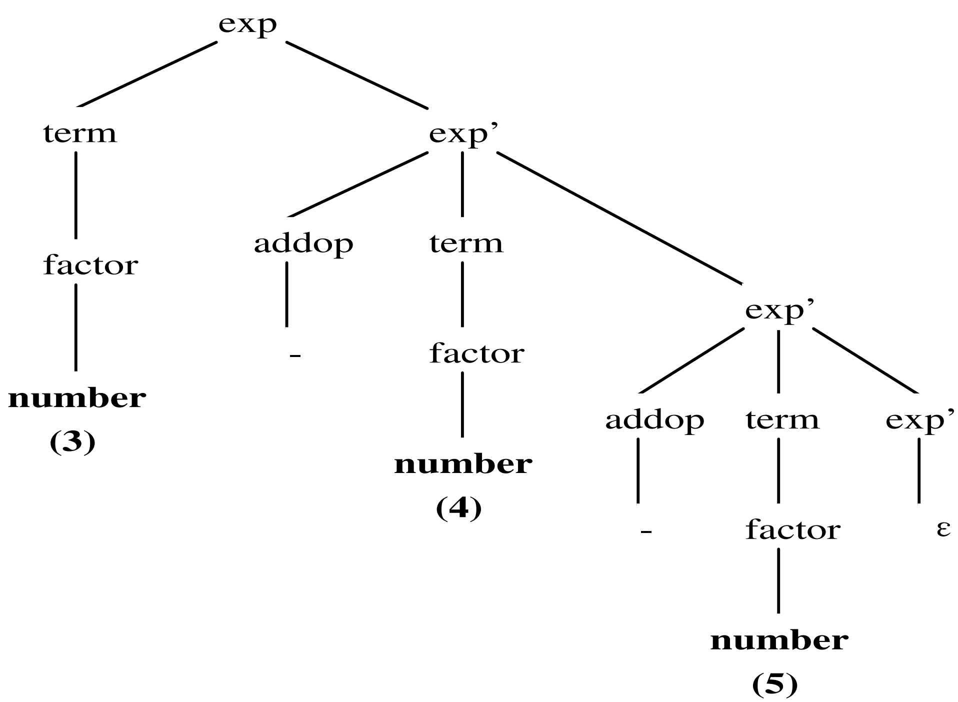
term' → mulop factor term'|ε

mulop →*

factor →(expr) | number

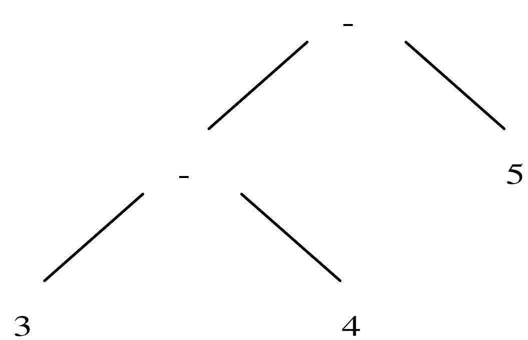
Parsing Tree

- **The parse tree for the expression 3-4-5**
  - **Not express the left associativity of subtraction.**



Syntax Tree

- **Nevertheless, a parse should still construct the appropriate left associative syntax tree**



- **From the given parse tree, we can see how the value of 3-4-5 is computed.**

Left-Recursion Removed Grammar and its Procedures

- **The grammar with its left recursion removed, exp and exp' as follows:**

exp → term exp'

exp'→ addop term exp'|ε

```
Procedure exp
      Begin
        Term;
        Exp';
      End exp;
```

```
Procedure exp'
      Begin
        Case token of
         +: match(+);
           term;
           exp';
         -: match(-);
           term;
           exp';
        end case;
      end exp'
```

Left-Recursion Removed Grammar and its Procedures

- **To compute the value of the expression, exp' needs a parameter from the exp procedure**

exp → term exp'

exp'→ addop term exp'|ε

```
function exp:integer;
    var temp:integer;
    Begin
      Temp:=Term;
      Return Exp'(temp);
    End exp;
```

```
function exp'(valsofar:integer):integer;
   Begin
     If token=+ or token=- then
     Case token of
      +: match(+);
        valsofar:=valsofar+term;
      -: match(-);
        valsofar:=valsofar-term;
     end case;
     return exp'(valsofar);
```

The LL(1) parsing table for the new expression

M[N,T]	(	number	)	+	-	*	$
Exp	exp → term exp’	exp → term exp’					
Exp’			exp’ → ε	exp’ → addop term exp’	exp’ → addop term exp’		exp’ → ε
Addop				addop → +	addop → −		
Term	term → factor term’	term → factor term’					
Term’			term’ → ε	term’ → ε	term’ → ε	term’ → mulop factor term’	term’ → ε
Mulop						mulop →*	
factor	factor → (expr)	factor → number					

Left Factoring

- **Left factoring is required when two or more grammar rule choices share a common prefix string, as in the rule**

$$A \to \alpha\beta | \alpha\gamma$$

- **Example:**

  stmt-sequence→stmt; stmt-sequence | stmt
  stmt→s

- **An LL(1) parser cannot distinguish between the production choices in such a situation**

- **The solution in this simple case is to "factor" the α out on the left and rewrite the rule as two rules:**

$$A \to \alpha A'$$
$$A' \to \beta | \gamma$$

Algorithm for Left Factoring a Grammar

While there are changes to the grammar do

For each non-terminal A do

Let α be a prefix of maximal length that is shared

By two or more production choices for A

If α≠εthen

Let A →α1|α2|…|αn be all the production choices for A

And suppose thatα1,α2,…,αk shareα, so that

A →αβ1|αβ2|…|αβk|αK+1|…|αn, the βj's share

No common prefix, andαK+1,…,αn do not share α

Replace the rule A →α1|α2|…|αn by the rules

A →αA'|αK+1|…|αn

A '→β1|β2|…|βk

Example 4.4

- **Consider the grammar for statement sequences, written in right recursive form:**

  Stmt-sequence→stmt; stmt-sequence | stmt

  Stmt→s

- **Left Factored as follows:**

  Stmt-sequence→stmt stmt-seq'

  Stmt-seq'→; stmt-sequence | ε

Example 4.4

- **Notices:**
  - **if we had written the stmt-sequence rule left recursively,**
  - **Stmt-sequence→stmt-sequence ;stmt | stmt**
- **Then removing the immediate left recursion would result in the same rules:**

  Stmt-sequence→stmt stmt-seq'

  Stmt-seq'→; stmt-sequence | ε

Example 4.5

- **Consider the following grammar for if-statements:**

  If-stmt → if (exp) statement

  | if (exp) statement else statement

- **The left factored form of this grammar is:**

  If-stmt → if (exp) statement else-part

  Else-part → else statement | ε

Example 4.6

- **An arithmetic expression grammar with right associativity operation:**

  $$exp \rightarrow term+exp \mid term$$

- **This grammar needs to be left factored, and we obtain the rules**

  $$exp \rightarrow term\ exp'$$
  $$exp' \rightarrow +\ exp \mid \varepsilon$$

- **Suppose we substitute term exp' for exp, we then obtain:**

  $$exp \rightarrow term\ exp'$$
  $$exp' \rightarrow +\ term\ exp' \mid \varepsilon$$

Example 4.7

- An typical case where a grammar fails to be LL(1)

  Statement → assign-stmt| call-stmt| other

  Assign-stmt→identifier:=exp

  Call-stmt→indentifier(exp-list)

- Where, identifier is shared as first token of both assign-stmt and call-stmt and,
- thus, could be the lookahead token for either.
- But not in the form can be left factored.

Example 4.7

- **First replace assign-stmt and call-stmt by the right-hand sides of their definition productions:**

  Statement → identifier:=exp | indentifier(exp-list)
  | other

- **Then, we left factor to obtain**

  Statement → identifier statement' | other
  Statement' →:=exp |(exp-list)

- **Note:**
  - **this obscures the semantics of call and assignment by separating the identifier from the actual call or assign action.**

4.2.4 Syntax Tree Construction in LL(1) Parsing

Difficulty in Construction

- **It is more difficult for LL(1) to adapt to syntax tree construction than recursive descent parsing**
- **The structure of the syntax tree can be obscured by left factoring and left recursion removal**
- **The parsing stack represents only predicated structure, not structure that have been actually seen**

Solution

- **The solution**
  - ***Delay the construction of syntax tree nodes to the point when structures are removed from the parsing stack.***
- **An extra stack is used to keep track of syntax tree nodes, and**
- **The “action” markers are placed in the parsing stack to indicate when and what actions on the tree stack should occur**

Example

- **A barebones expression grammar with only an addition operation.**

  $E \rightarrow E + n \mid n$

  /* be applied left association*/

- **The corresponding LL(1) grammar with left recursion removal is.**

  $E \rightarrow n E'$

  $E' \rightarrow +nE' \mid \varepsilon$

To compute the arithmetic value of the expression

- Use a separate stack to store the intermediate values of the computation, called the value stack;
  - Schedule two operations on that stack:
    - A push of a number;
    - The addition of two numbers.
    - PUSH can be performed by the match procedure, and
    - ADDITION should be scheduled on the stack, by pushing a special symbol (such as #) on the parsing stack.
  - This symbol must also be added to the grammar rule that match a +, namely, the rule for E':
    - $E' \rightarrow +n\#E' | \varepsilon$
- Notes: The addition is scheduled just after the next number, but before any more E' non-terminals are processed. This guaranteed left associativity.

The actions of the parser to compute the value of the expression 3+4+5

Parsing Stack	Input	Action	Value Stack
$E	3+4+5$	E→n E’	$
$E’n	3+4+5$	Match/push	$
$E’	+4+5$	E’ →+n#E’	3$
$E’#n+	+4+5$	Match	3$
$E’#n	4+5$	Match/push	3$
$E’#	+5$	Addstack	43$
$E’	+5$	E’ →+n#E’	7$
$E’#n+	+5$	Match	7$
$E’#n	5$	Match/push	7$
$E’#	$	Addstack	57$
$E’	$	E’ → ε	12$
$	$	Accept	12$

4.3 First and Follow Sets

The LL(1) parsing algorithm is based on the LL(1) parsing table

The LL(1) parsing table construction involves the First and Follow sets

4.3.1 First Sets

Definition

- **Let X be a grammar symbol(a terminal or non-terminal) or ε. Then First(X) is a set of terminals or ε, which is defined as follows:**
  - **1. If X is a terminal or ε, then First(X) = {X};**
  - **2. If X is a non-terminal, then for each production choice X→X1X2…Xn,**
  - **First(X) contains First(X1)-{ε}.**
  - **If also for some i<n, all the set First(X1)..First(Xi) contain ε,the first(X) contains First(Xi+1)-{ε}.**
  - **IF all the set First(X1)..First(Xn) contain ε, the First(X) contains ε.**

Definition

- **Let α be a string of terminals and non-terminals, X1X2…Xn. First(α) is defined as follows:**

  1.First(α) contains First(X1)-{ε};

  2.For each i=2,…,n, if for all k=1,..,i-1, First(Xk) contains ε, then First(α)

  contains First(Xk)-{ε}.

  3. IF all the set First(X1)..First(Xn) contain ε, the First(α) contains ε.

Algorithm Computing First (A)

- *Algorithm for computing First(A) for all non-terminal A:*

```
For all non-terminal A do First(A):={ };
While there are changes to any First(A) do
  For each production choice A→X1X2…Xn do
    K:=1; Continue:=true;
    While Continue= true and k<=n do
      Add First(Xk)-{ε} to First(A);
      If ε is not in First(Xk) then Continue:= false;
      k:=k+1;
    If Continue = true then addεto First(A);
```

Algorithm Computing First (A)

- *Simplified algorithm in the absence of ε-production.*

  For all non-terminal A do First(A):={ };

  While there are changes to any First(A) do

  For each production choice A→X1X2…Xn do

  Add First(X1) to First(A);

About Nullable Non-Terminal

- **Definition: A non-terminal A is nullable if there exists a derivation A=>ε.**
- **Theorem: A non-terminal A is nullable if and only if First(A) contains ε.**
- **Proof : 1. If A is nullable, then First(A) contains ε.**
  - **As A => *ε, we use induction on the length of a derivation.**
  - **(1) A=> ε, then there must be a production A→ε,by definition,**
  - **First(A) contain First(ε)={ε}.**
  - **(2) Assume the truth of the statement for derivation of length < n,**
  - **and let A=>X1…XK=>*ε be a derivation of length n;**
  - **All the Xi must be non-terminals;**
- **Implying that each Xi =>*ε, and in fewer than n steps.**
- **Thus, by the induction assumption, for each i First(Xi)={ ε}**
- **Finally, by definition, First(A) must containε.**

Example

- Simple integer expression grammar

exp → expr addop term

|term

addop → +|-

term → term mulop factor

| factor

mulop →*

factor →(expr) | number

- Write out each choice separately in order:

(1) exp → exp addop term

(2) exp → term

(3) addop → +

(4) addop → -

(5) term → term mulop factor

(6) term → factor

(7) mulop →*

(8) factor →(exp)

(9) factor →number

First Set for Above Example

- We can use the simplified algorithm as there exists no ε-production
- The First sets are as follows:

  First(exp)={(,number}

  First(term)={(,number}

  First(factor)={(,number}

  First(addop)={+,-}

  First(mulop)={*}

The computation process for above First Set

Grammar Rule	Pass 1	Pass 2	Pass 3
expr → expr addop term			
expr → term			First(exp)={(,number}
addop → +	First(addop)={+}		
addop → -	First(addop)={+,-}		
term → term mulop factor			
term → factor		First(term)={(,number}	
mulop →*	First(mulop)={*}		
factor →(expr)	First(factor)={()		
factor →number	First(factor)={(,number)		

Example

- **Left factored grammar of if-statement**

  Statement → if-stmt | other

  If-stmt → if (exp) statement else-part

  Else-part → else statement | ε

  Exp → 0 | 1

- **We write out the grammar rule choice separately and number them:**

  (1) Statement → if-stmt

  (2) Statement → other

  (3) If-stmt → if (exp) statement else-part

  (4) Else-part → else statement

  (5) Else-part →ε

  (6) Exp → 0

  (7) Exp → 1

The First Set for Above Example

- Note:
  - This grammar does have an ε-production, but the only nullable non-terminal *else-part* will not in the beginning of left side of any rule choice and will not complicate the computation process.

- The First Sets:

First(statement)={if,other}

First(if-stmt)={if}

First(else-part)={else,ε}

First(exp)={0,1}

The computation process for above First Set

Grammar Rule	Pass 1	Pass 2
Statement → if-stmt		First(statement)={if,other}
Statement → other	First(statement)={other}	
If-stmt → if (exp) statement else-part	First(if-stmt)={if}	
Else-part → else statement	First(else-part)={else}	
Else-part → ε	First(else-part)={else, ε }	
Exp → 0	First(exp)={1}	
Exp → 1	First(exp)={0,1}	

Example

- Grammar for statement sequences:
- Stmt-sequence →stmt stmt-seq’
- Stmt-seq’ →; stmt-sequence|ε
- stmt→s
- We list the production choices individually:
- Stmt-sequence →stmt stmt-seq’
- Stmt-seq’ →; stmt-sequence
- Stmt-seq’ →ε
- stmt→s
- The First sets are as follows:
- First(stmt-sequence)={s}
- First(stmt-seq’)={;, ε}
- First(stmt)={s}

4.3.2 Follow Sets

Definition

Given a non-terminal A, the set Follow(A) is defined as follows.

(1) if A is the start symbol, the $ is in the Follow(A).

(2) if there is a production $B \rightarrow \alpha A \gamma$ then First(γ)-{ε} is in Follow(A).

(3) if there is a production $B \rightarrow \alpha A \gamma$ such that ε in First(γ), then Follow(A) contains Follow(B).

Definition

- **Note: The symbol $ is used to mark the end of the input.**
  - **The empty "pseudotoken" ε is never an element of a follow set.**
  - **Follow sets are defined only for non-terminal.**
  - **Follow sets work "on the right" in production while First sets work "on the left" in the production.**
- **Given a grammar rule $A \rightarrow \alpha B$, Follow(B) will contain Follow(A),**
  - **the opposite of the situation for first sets, if $A \rightarrow B\alpha$, First(A) contains First(B), except for ε.**

Algorithm for the computation of follow sets

- Follow(start-symbol):={$};
- For all non-terminals A≠start-symbol do follow(A):={ };
- While there changes to any follow sets do

  For each production A→X1X2…Xn do

  For each Xi that is a non-terminal do

  Add First(Xi+1Xi+2…Xn) – {ε} to Follow(Xi)

  fεis in First(Xi+1Xi+2…Xn) then

  Add Follow(A) to Follow(Xi)

Example

- The simple expression grammar.

    (1) exp → exp addop term

    (2) exp → term

    (3) addop → +

    (4) addop → -

    (5) term → term mulop factor

    (6) term → factor

    (7) mulop →*

    (8) factor →(exp)

    (9) factor →number

Example

- The first sets:

  First(exp)={(,number}

  First(term)={(,number}

  First(factor)={(,number}

  First(addop)={+,-}

  First(mulop)={*}

- The Follow sets:

  Follow(exp)={$,+,-, } }

  Follow(addop)={(,number)

  Follow(term)={ $,+,-, *,}}

  Follow(mulop)={(,number)

  Follow(factor)={ $,+,-, *,}}

The progress of above computation

Grammar rule	Pass 1	Pass 2
exp → exp addop term	Follow(exp)={$,+,- } Follow(addop)={(,number) Follow(term)={ $,+,-}	Follow(term)={ $,+,-, *,）}
Exp → term		
term → term mulop factor	Follow(term)={ $,+,-, *} Follow(mulop)={(,number) Follow(factor)={ $,+,-, *}	Follow(factor)={ $,+,-, *,）}
term →factor		
factor →(exp)	Follow(exp)={$,+,-,）}	

Example

- The simplified grammar of if-statements:

(1) Statement $\rightarrow$ if-stmt

(2) Statement $\rightarrow$ other

(3) If-stmt $\rightarrow$ if (exp) statement else-part

(4) Else-part $\rightarrow$ else statement

(5) Else-part $\rightarrow \varepsilon$

(6) Exp $\rightarrow$ 0

(7) Exp $\rightarrow$ 1

Example

- The First sets:

  First(statement)={if,other}

  First(if-stmt)={if}

  First(else-part)={else,ε}

  First(exp)={0,1}

- Computing the following Follow sets:

  Follow(statement)={$,else}

  Follow(if-statement)={$,else}

  Follow(else-part)={$,else}

  Follow(exp)={)}

Example

- The simplified statement sequence grammar.

  (1) Stmt-sequence →Stmt Stmt-seq'

  (2) Stmt-seq' →; Stmt-sequence

  (3) Stmt-seq' →ε

  (4) Stmt→s

Example

- The First sets are as follows:

  First(Stmt-sequence)={s}

  First(Stmt)={s}

  First(Stmt-seq')={; , ε}

- And, the Follow sets:

  Follow(Stmt-sequence)={$}

  Follow(Stmt)={; , $}

  Follow(Stmt-seq')={$}

4.3.3 Constructing LL(1) Parsing Tables

The table-constructing rules

(1) If $A \rightarrow \alpha$ is a production choice, and there is a derivation $\alpha =>^* a\beta$, where a is a token, then add $A \rightarrow \alpha$ to the table entry M[A,a]

(2) If $A \rightarrow \alpha$ is a production choice, and there are derivations $\alpha =>^* \varepsilon$ and $S\$ =>^* \beta A a \gamma$, where S is the start symbol and a is a token (or $), then add $A \rightarrow \alpha$ to the table entry M[A,a]

- Clearly, the token a in the rule (1) is in First(α), and the token a of the rule (2) is in Follow(A).
- Thus we can obtain the following algorithmic construction of the LL(1) parsing table:

Algorithm and Theorem

- Repeat the following two steps for each non-terminal A and production choice $A \rightarrow \alpha$.
  - For each token a in First(α), add $A \rightarrow \alpha$ to the entry M[A,a].
  - If ε is in First(α), for each element a of Follow(A) (a token or \$), add $A \rightarrow \alpha$ to M[A,a].
- Theorem:A grammar in BNF is LL(1) if the following conditions are satisfied.
  - For every production $A \rightarrow \alpha 1|\alpha 2|\ldots|\alpha n$, First($\alpha i$)$\cap$ First(αj) is empty for all i and j, $1 \leqq i,j \leqq n$, $i \neq j$.
  - For every non-terminal A such that First(A) contains ε, First(A) $\cap$Follow(A) is empty.

Example

- The simple expression grammar.

  exp → term exp'

  exp'→ addop term exp'|ε

  addop → + -

  term → factor term'

  term' → mulop factor term'|ε

  mulop →*

  factor →(expr) | number

The first and follow set

First Sets	Follow Sets
First(exp)={(,number) First(exp')={+,-, ε } First(term)={(,number) First(term')={*, ε } First(factor)={(,number} First(addop)={+,-} First(mulop)={*}	Follow(exp)={$,） } Follow(exp')={$,） } Follow(addop)={(,number) Follow(term)={ $,+,-,） } Follow(term')={$,+,-,） } Follow(mulop)={(,number} Follow(factor)={ $,+,-, *,） }

the LL(1) parsing table

M[N,T]	(	number	)	+	-	*	$
Exp	exp → term exp’	exp → term exp’					
Exp’			exp’→ ε	exp’ → addop term exp’	exp’ → addop term exp’		exp’→ ε
Addop				addop → +	addop → -		
Term	term → factor term’	term → factor term’					
Term’			term’ → ε	term’ → ε	term’ → ε	term’ → mulop factor term’	term’ → ε
Mulop						mulop → *	
factor	factor →(expr)	factor → number					

Example

- The simplified grammar of if-statements

  Statement → if-stmt | other

  If-stmt → if (exp) statement else-part

  Else-part → else statement | ε

  Exp → 0 | 1

The first and follow set

First Sets	Follow Sets
First(statement)={if,other} First(if-stmt)={if} First(else-part)={else, ε } First(exp)={0,1}	Follow(statement)={$,else} Follow(if-statement)={$,else} Follow(else-part)={$,else} Follow(exp)={ ） }

the LL(1) parsing table

M[N,T]	If	Other	Else	0	1	$
Statement	Statement → if-stmt	Statement → other				
If-stmt	If-stmt → if (exp) statement else-part					
Else-part			Else-part → else statement Else-part → ε			Else-part → ε
exp				Exp → 0	Exp → 1	

Example

- Consider the following grammar with left factoring applied.

  (1) Stmt-sequence $\rightarrow$Stmt Stmt-seq'

  (2) Stmt-seq' $\rightarrow$; Stmt-sequence | ε

  (3) Stmt$\rightarrow$s

The first and follow set

First Sets	Follow Sets
First(Stmt-sequence)={s} First(Stmt)={s} First(Stmt-seq')={; , ε }	Follow(Stmt-sequence)={$} Follow(Stmt)={; , $} Follow(Stmt-seq')={$}

the LL(1) parsing table

M[N,T]	S	;	$
Stmt-sequence	Stmt-sequence → Stmt Stmt-seq’		
Stmt	Stmt→s		
Stmt-seq’		Stmt-seq’ → ; Stmt-sequence|	Stmt-seq’ →| ε

4.3.4 Extending the lookahead: LL(k) Parsers

Definition of LL(k)

- The LL(1) parsing method can be extend to k symbols of look-ahead.
- Definitions:
  - Firstk(α)={ wk | α=>* w}, where, wk is the first k tokens of the string w if the length of w > k, otherwise it is the same as w.
  - Followk(A)={ wk | S$=>*$\alpha$A w}, where, wk is the first k tokens of the string w if the length of w > k, otherwise it is the same as w.
- LL(k) parsing table:
  - The construction can be performed as that of LL(1).

Complications in LL(k)

- The complications in LL(k) parsing:
  - The parsing table become larger; since the number of columns increases exponentially with k.
  - The parsing table itself does not express the complete power of LL(k) because the follow strings do not occur in all contexts.
  - Thus parsing using the table as we have constructed it is distinguished from LL(k) parsing by calling it Strong LL(k) parsing, or SLL(k) parsing.

Complications in LL(k)

- The LL(k) and SLL(k) parsers are uncommon.
  – Partially because of the added complex;
  – Primarily because of the fact that a grammar fails to be LL(1) is in practice likely not to be LL(k) for any k.

4.4 A Recursive-Descent Parser For The Tiny Language

The Grammar of the TINY language in BNF

- *program* → *stmt-sequence*
- *stmt-sequence*→ *stmt-sequence*; *statement* | *statement*
- *statement* → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*
- *if-stmt*→ **if** *exp* **then** *stmt-sequence* **end**
- | **if** *exp* **then** *stmt-sequence* **else** *stmt-sequence* **end**
- *repeat-stmt*→ **repeat** *stmt-sequence* **until** *exp*
- *assign-stmt*→ **identifier :=** *exp*
- *read-stmt* → **read identifier**
- *write-stmt* → **write** *exp*

The Grammar of the TINY language in BNF

- *exp* → *simple-exp comparison-op simple-exp* | *simple-exp*
- *comparison-op* → **< | =**
- *simple-exp* → *simple-exp addop term* | *term*
- *addop* → **+|-**
- *term* → *term mulop factor factor* | *factor*
- *mulop* → ***|/**
- *factor* → **(***exp***)** **|number |identifier**

TINY PARSER CODES

- The TINY parser consists of two code files:
  - **parse.h and parse.c**
  - The parse.h: (see appendix B, lines 850-865)
  - **TreeNode * parse(void)**
  - The main routine parse will return a pointer to the syntax tree constructed by the parser.
  - The parse.c(see appendix B, lines 900-1114)
- **11 mutually recursive procedure** that correspond directly to the EBNF grammar.
- **The operators non-terminals** are recognized as part of their associated expressions.

TINY PARSER CODES

- The static variable **token** is used to keep the look-ahead token
- The contents of each recursive procedures should be relatively **self-explanatory** except stmt-sequence
- The **utility procedures** used by the recursive parsing procedures in util.c:(Appendix B, lines 350-526).

- **NewStmtNode**(line 405-421): take the type of statement as parameter;
-
- Allocate a new statement node of this kind;
- Return a pointer to the newly allocated node.

- **NewExpNode**(line 423-440): take the type of exp ad parameter;

TINY PARSER CODES

- Allocate a new exp node of this kind;
- Return a pointer to the new allocated node.
- **Copystring**(line 442-455): take a string as parameter;
- Allocate a sufficient space for a copy, and copy the string;
- Return a pointer to the newly allocated copy.
- A procedure PrintTree in util.c (linge 473-506) writes a linear version of the syntax tree to the listing, so that we may view the result of a parse.

End of Part Two

THANKS