Chapter 4
Top-Down Parsing

2022 Spring&Summer

Outline

- Top-down parsing
- Recursive-descent parsing
- LL(k) grammars
- Transforming a grammar into LL form

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 algorithm 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

   $X \rightarrow X_1 X_2 \ldots X_n$, First($X$) contains First($X_1$)-$\{\varepsilon\}$.

4.3.1 First Sets

Let $\alpha = X_1X_2 \ldots X_n$ be a string of terminals and non-terminals,. First(α) is defined as follows:

1. First(α) contains First(X_1)-{ε};
2. For each i=2,…,n, if for all $k=1,..,i-1$, First(X_k) contains ε, then First(α) constains First(X_k)-{ε}.
3. If all the set First(X_1)..First(X_n) contain ε, the First(α) contains ε.

4.3.1 First Sets

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 $A \rightarrow X_1 X_2 \ldots X_n$ *do*

k:=1; Continue:=true;

while Continue= true and k<=n do

add First(X_k*)-{ε} to First(A);*

if ε is not in First(X_k*) then Continue:= false;*

k:=k+1;

if Continue = true then addεto First(A);

4.3.1 First Sets

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 $A \rightarrow X_1 X_2 \ldots X_n$ *do*
    add First(X_1*) to First(A);*

4.3.1 First Sets

- Definition:

  A non-terminal A is *nullable* if there exists a derivation $A \Rightarrow^* \varepsilon$.

- Theorem:

  A non-terminal A is *nullable* if and only if First(A) contains ε.

4.3.1 First Sets

- Grammar for statement sequences:

  stmt-sequence $\rightarrow$ *stmt stmt-seq'*

  stmt-seq' $\rightarrow$ *; stmt-sequence* | ε

  stmt $\rightarrow$ *s*

- List the production choices individually:

  stmt-sequence $\rightarrow$ *stmt stmt-seq'*

  stmt-seq' $\rightarrow$ *; stmt-sequence*

  stmt-seq' $\rightarrow$ ε

  stmt $\rightarrow$ *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($\gamma$)-{$\varepsilon$} 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).

4.3.2 Follow Sets

- Note:
  1. The symbol *$* is used to mark the end of the input.
  2. The empty "pseudotoken" ε is never an element of a follow set.
  3. Follow sets are defined only for non-terminal.

4.3.2 Follow Sets

- Follow sets work "on the right" in production
- 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 possibly for ε.

4.3.2 Follow Sets

Algorithm for the computation of follow sets:

follow(start-symbol):={$\$$};

for all non-terminals $A\neq$start-symbol do follow(A):={ };

while there changes to any follow sets do

for each production $A \rightarrow X_1 X_2 \ldots X_n$ do

for each X_i that is a non-terminal do

add First($X_{i+1} X_{i+2} \ldots X_n$) – {ε} to Follow($X_i$)

if ε is in First($X_{i+1} X_{i+2} \ldots X_n$) then

add Follow(A) to Follow(X_i)

4.3.2 Follow Sets

$S \rightarrow ES'$
$S' \rightarrow \varepsilon \mid +S$
$E \rightarrow$ **num** $\mid (S)$

- Nullable
  - Only S' is nullable
- FIRST
  - FIRST(ES') = {**num**,**(**}
  - FIRST(+S) = {**+**}
  - FIRST(**num**) = {**num**}
  - FIRST(**(S)**) = {**(**}, FIRST(S') = {**+, ε**}
  - FIRST(S) = {**num**,**(**},
- FOLLOW
  - FOLLOW(S) = { **), $** }
  - FOLLOW(S') = { **), $** }
  - FOLLOW(E) = { **+,) , $** }

4.3.2 Follow Sets

- Example:

stmt-sequence $\rightarrow$ *stmt stmt-seq'*
stmt-seq' $\rightarrow$ *; stmt-sequence*
stmt-seq' $\rightarrow$ ε
stmt $\rightarrow$ *s*

The First sets :

First(stmt-sequence)={s}

First(stmt)={s}

First(stmt-seq')={;, ε}

the Follow sets:

Follow(stmt-sequence)={$}

Follow(stmt)={;, $}

Follow(stmt-seq')={$}

4.3.2 Follow Sets

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

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)={$,+,-,*,)}

4.3.2 Follow Sets

Example: The simplified grammar of if-statements:

(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 sets:

First(statement)={if,other},

First(if-stmt)={if}

First(else-part)={else,ε},

First(exp)={0,1}

the Follow sets:

Follow(statement)={$,else},

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

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

4.3.3 Constructing LL(1) Parsing Tables

The table-constructing rules:

- 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];
- 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];

4.3.3 Constructing LL(1) Parsing Tables

The following algorithmic construction of the LL(1) parsing table:

Repeat the following two steps for each non-terminal A and production choice $A \rightarrow \alpha$.

1. For each token a in First(α*), add* $A \rightarrow \alpha$ *to the entry M[A,a].*

2. If ε *is in First(*α*), for each element a of Follow(A) (a token or \$), add* $A \rightarrow \alpha$ *to M[A,a].*

4.3.3 Constructing LL(1) Parsing Tables

- Theorem:

  A grammar in BNF is LL(1) if the following conditions are satisfied.

  1. For every production $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$, First($\alpha_i$)$\cap$ First(α_j) is empty for all i and j, $1 \leqq i,j \leqq n$, $i \neq j$.
  2. For every non-terminal A such that First(A) contains ε, First(A) $\cap$Follow(A) is empty.

4.3.3 Constructing LL(1) Parsing Tables

Algorithm for automatically generating a predictive parse table from a grammar

FOLLOW(S) = { **), $** }
FOLLOW(S') = { **), $** }
FOLLOW(E) = { **+,) , $** }

FIRST(E) = {**num,(**}
FIRST(S') = {**+, ε**}
FIRST(S) = {**num,(**},

$S \rightarrow ES'$

$S' \rightarrow \varepsilon \mid +S$

$E \rightarrow num \mid (S)$

	num	**+**	**(**	**)**	**$**
S	$S \rightarrow ES'$		$S \rightarrow ES'$		
S'		$S' \rightarrow +S$		$S' \rightarrow \varepsilon$	$S' \rightarrow \varepsilon$
E	$E \rightarrow num$		$E \rightarrow (S)$		

4.3.3 Constructing LL(1) Parsing Tables

- Example: The simple expression grammar.

exp → *term exp'*
exp' → *addop term exp'* | ε
addop → + | -
term → *factor term'*
term' → *mulop factor term'* | ε
mulop → *
factor → (*exp*) | *number*

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

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

4.3.3 Constructing LL(1) Parsing Tables

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* → (*exp*)	*factor* → **number**					

4.3.3 Constructing LL(1) Parsing Tables

- 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*

First sets:
First (statement)={if,other},
First (if-stmt)={if}
First (else-part)={else,ε},
First (exp)={0,1}

Follow sets:
Follow(statement)={$,else},
Follow(if-stmt)={$,else}
Follow(else-part)={$,else},
Follow(exp)={)}

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	

4.3.3 Constructing LL(1) Parsing Tables

- Example: Consider the following grammar with left factoring applied.

(1) stmt-sequence →stmt stmt-seq'

(2) stmt-seq'→; stmt-sequence|ε *(3) stmt→s*

First sets :
First(stmt-sequence)={s}
First(stmt)={s}
First(stmt-seq')={;, ε}

Follow sets:
Follow(stmt-sequence)={$}
Follow(stmt)={;, $}
Follow(stmt-seq')={$}

M[N,T]	*s*	*;*	*$*
stmt-sequence	*stmt-sequence→ stmt stmt-seq'*		
stmt	*stmt→s*		
stmt-seq'		*stmt-seq'→ stmt-sequence*	*stmt-seq'→ε*

4.5 Error Recovery in Top-Down Parsers

- Different levels of response to errors.

  Give a meaningful error message;

  Determine as closely as possible the location where that error has occurred.

  Some form of error correction; (*error repair*)

  The parser attempts to infer a correct program from the incorrect one given.

- Most of the techniques for error recovery are ad hoc, and general principles are hard to come by.

4.5 Error Recovery in Top-Down Parsers

- Some important considerations that apply are the following:
  - (1) To determine that an error has occurred as soon as possible.
  - (2) After an error has occurred
    - pick a likely place to resume the parse
    - try to parse as much of the code as possible
  - (3) To avoid the error cascade problem.
  - (4) To avoid infinite loops on error.

4.5 Error Recovery in Top-Down Parsers

Some goals conflict with each other, so that a compiler writer is forced to make trade-offs during the construction of an error handler.

- *Panic mode:*

  A standard form of *error recovery* in recursive-decent parsers

  The error handler will consume a possibly large number of tokens in an attempt to find a place to resume parsing;

4.5 Error Recovery in Top-Down Parsers

- The basic mechanism of panic mode:
  - A set of *synchronizing tokens* are provided to each recursive procedure;
  - If an error is encountered, the parser scans ahead, throwing away tokens until one of the synchronized tokens is seen in the input, and then parsing is resumed.

4.5 Error Recovery in Top-Down Parsers

- The important decisions to be made in this error recovery method:

  (1) *What tokens to add to the synchronizing set at each point in the parse?*

  Follow sets are important candidates

  (2) First sets:

  - Prevent the error handler from skipping important tokens that begin major new constructs
  - Detect errors early in the parse.

4.5 Error Recovery in Top-Down Parsers

- *check-input* procedure: performs the early look-ahead checking

```
procedure checkinput(Firstset, Followset)
begin
    if not (token in firstset) then
        error;
        scanto(firstset∪followset)
    end if;
end
```

- *scanto* procedure:

```
procedure scanto(synchset)
begin
    while not (token in synchset∪{$}) do
        gettoken;
end scanto
```

4.5 Error Recovery in Top-Down Parsers

```
procedure exp(synchset)
 begin
      checkinput({(,number},synchset}
      if not (token in synchset) then
            term(synchset);
            while token=+ or token=- do
                  match(token);
                  term(synchset);
            end while
          checkinput(synchset,{(,number});
      end if;
 end exp;
```

4.5 Error Recovery in Top-Down Parsers

```
procedure factor(synchset);
     begin
          checkinput({(,number},synchset};
          if not (token in synchset) then
                case token of
                      ( : match( ( );
                            exp( { ) });
                            match());
                      number:
                            match(number);
                      else error;
                end case;
                checkinput(synchset,{ (,number});
          end if;
     end factor;
```

4.5 Error Recovery in Top-Down Parsers

- Note: *checkinput* is called twice in each procedure

  (1) once to verify that a token in the *First set* in the next token in the input;

  (2) a second time to verify that a token in the *Follow set* is the next token on exit.

- This form of panic mode will generate reasonable errors.

For example, *(2+-3)*4-+5* will generate exactly two error messages,

(1) one at the first minus sign, and

(2) one at the second plus sign.

4.5 Error Recovery in Top-Down Parsers

- Note:
  - In general, *synchset* is to be passed down in the recursive calls, with new synchronizing tokens added as appropriate.
  - As an exception, in the case of *factor, exp* is called with right parenthesis only as its follow set (*synchset* is discarded)
  - The kind of ad hoc analysis accompanies *panic mode* error recovery.

4.5.2 Error Recovery in LL(1) Parsers

- Panic mode error recovery can be implemented in LL(1) parsers
  - A new stack is required to keep the *synchset* parameters;
  - A call to *checkinput* must be scheduled by the algorithm before each generate action of the algorithm;
- The primary error situation occurs
  - The current input token is not in First(A)(or Follow(A), if ε is in First(A)), A is the non-terminal at the top of the stack.

4.5.2 Error Recovery in LL(1) Parsers

- An alternative to the use of an extra stack is :

Build the sets of *synchronizing tokens* directly into the LL(1) parsing table, together with the corresponding actions that *checkinput* would take.

4.5.2 Error Recovery in LL(1) Parsers

- Given a non-terminal A at the top of the stack and an input token that is not in First(A) (or Follow(A), if ε is in First(A)), there are three possible alternative:
  1. Pop A from the stack.
  2. Successively pop tokens from the input until a token is seen for which we can restart the parse.
  3. Push a new non-terminal onto the stack.

4.5.2 Error Recovery in LL(1) Parsers

- Alternative 1 (Pop)

  if the current input token is $ or is in Follow(A);
- Alternative 2 (scan)

  if the current input token is not $ and is not in First(A)∪Follow(A)
- Option 3 (Push a new non-terminal)

  occasionally useful in special situation.

Homework of Chapter 4

4.8

Consider the grammar

lexp→atom|list

atom→number|***identifier***

list→(lexp-seq)

lexp-seq→lexp-seq lexp|lexp

(1) Remove the left recursion.

(2) Construct First and Follow sets for the nonterminals of the resulting gram mar.

(3) Show that the resulting grammar is LL(1).

(4) Construct the LL(1) parsing table for the resulting grammar.

(5) Show the actions of the corresponding LL(1) parser,given the input string (a (b (2)) (c)).

Homework of Chapter 4

4.12 Questions:

(1) Can an LL(1) grammar be ambiguous? Why or why not?

(2) Can an ambiguous grammar be LL(1)? Why or why not?

(3) Must an unambiguous grammar be LL(1)? Why or why not?