CS321 Week 5: Top-Down Parsing

Jingke Li, Portland State University

Winter 2014

Today's Topics

- Introduction to top-down parsing
- First and follow sets
- Construction of LL(1) parsing table

Syntax Analysis (Parsing)

token stream → $\boxed{\text{Parser}}$ → *syntax tree*

Main Tasks:

- Recognizing the hierarchical syntactic structure of the input program, and representing it in a syntax tree.
- Detecting syntax errors

Optional Task:

- Managing symbol information

Parsing Techniques

- Top-Down Parsing (*a.k.a.* Predictive Parsing, LL Parsing)
  - Start at the start symbol of the grammar, repeatedly "predict" the next production to apply (with the help of peeking at the incoming token(s)), until the whole input token sequence is derived.
  - Build a syntax tree from *top down*.
  - *Implementation:* recursive descent or table-driven.
- Bottom-Up Parsing (*a.k.a.* LR Parsing)
  - Start at the beginning of the input token sequence, repeatedly look for a subsequence that matches a production's right-hand-side, and "reduce" it to the left-hand-side nonterminal, until the whole input token sequence is reduced to the start symbol of the grammar.
  - Build a syntax tree from *bottom up*.
  - *Implementation:* table-driven.

Recursive Descent Predictive Parsing

- Represent grammar in BNF form, with no extended operators.

  Example:

  0. *Program0* → *Program* \$
  1. *Program* → begin *StmtList* end
  2. *StmtList* → *Stmt* ; *StmtList*
  3. *StmtList* → ϵ
  4. *Stmt* → simpleS
  5. *Stmt* → begin *StmtList* end

  A Note on the Augmented Production:

  When building a parser for a grammar, we want to make sure that the parser sees all the tokens in the input before making an "accept" or "reject" decision.

  A common approach is to augment the grammar with a bogus production to allow an end-marker ("\$") to be added at the end of the start symbol. In a parser implementation, the end-marker is typically mapped to `<EOF>`.

Recursive Descent Parsing ('2)

- Associate each nonterminal with a parsing procedure; and each of its productions a "clause" within the procedure.

```
void Program0() { // Program0 -> Program $
  Program(), accept if next token is "$"
}
void Program() {  // Program ->
  <clause 1>      //   begin StmtList end
}
void StmtList() { // StmtList ->
  <clause 1>      //   Stmt ; StmtList
  <clause 2>      //   ε
}
void Stmt() {     // Stmt ->
  <clause 1>      //   simpleS
  <clause 2>      //   begin StmtList end
}
```

Recursive Descent Parsing ('3)

- The body of each clause consists of a sequence of match and call statements; corresponding to the rhs symbols of the production.

```
void Program() {  // Program ->
  <clause 1>      //   begin StmtList end
    match("begin"); call StmtList(); match("end");
}

void StmtList() { // StmtList ->
  <clause 1>      //   Stmt ; StmtList
    call Stmt(); match(";"); call StmtList();
  <clause 2>      //   ϵ
    /* empty */
}

void Stmt() {     // Stmt ->
  <clause 1>      //   simpleS
    match("simpleS");
  <clause 2>      //   begin StmtList end
    match("begin"); call StmtList(); match("end");
}
```


Recursive Descent Parsing ('4)

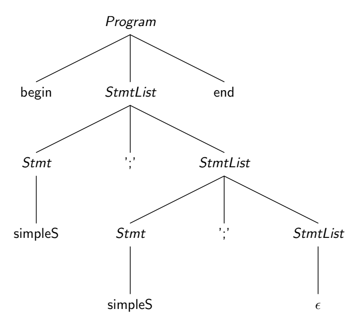
- Start the parsing process with the start symbol's procedure. In each step, either a terminal is matched or a nonterminal's procedure is called. Lookahead(s) help to determine the correct clause to follow.

Input: begin simpleS ; simpleS ; end

Next Token	*Parsing Action*
—	call Program()
begin	match("begin")
simpleS	call StmtList(), pick <clause 1>
simpleS	call Stmt(), pick <clause 1>
simpleS	match("simpleS"), return
;	match(";")
simpleS	call StmtList(), pick <clause 1>
simpleS	call Stmt(), pick <clause 1>
simpleS	match("simpleS"), return
;	match(";")
end	call StmtList, pick <clause 2>
end	return
end	return
end	return
end	match("end"), return
$	accept

Recursive Descent Parsing ('5)

- Along the way, a parse tree can be constructed from top down.



Key Issue for Recursive Descent Parsing

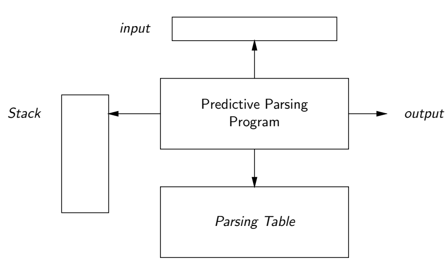
- Using the next incoming token (*lookahead symbol*) to predict a production to apply at every step.

Equivalently,

- Finding the lookahead symbols for each production, so that the correct clause in the corresponding parsing routine can be picked.

Table-Driven Predictive Parsing

Use a *parsing table* and a *stack* to replace recursive calls.



Parsing Table

Given a terminal and a nonterminal, the parsing table will predict the production to use.

Example:

0. *Program0* $\rightarrow$ *Program* $
1. *Program* $\rightarrow$ begin *StmtList* end
2. *StmtList* $\rightarrow$ *Stmt* ; *StmtList*
3. *StmtList* $\rightarrow$ ϵ
4. *Stmt* $\rightarrow$ simpleS
5. *Stmt* $\rightarrow$ begin *StmtList* end

	begin	simpleS	;	end	$
Program	1				
StmtList	2	2		3	
Stmt	5	4			

Parsing Actions for the Example

0. *Program0* $\rightarrow$ *Program* \$
1. *Program* $\rightarrow$ begin *StmtList* end
2. *StmtList* $\rightarrow$ *Stmt* ; *StmtList*
3. *StmtList* $\rightarrow$ ϵ
4. *Stmt* $\rightarrow$ simpleS
5. *Stmt* $\rightarrow$ begin *StmtList* end

	begin	simpleS	;	end	\$
Program	1				
StmtList	2	2		3	
Stmt	5	4			

Input: begin simpleS ; simpleS ; end

			Stmt	simpleS
	begin		;	;
	StmtList	*StmtList*	*StmtList*	*StmtList*
Program	end	end	end	end
\$	\$	\$	\$	\$

	. . .		
StmtList			
end		end	
\$		\$	\$

Key Issue for Table-Drive Predictive Parsing

- Converting production lookahead information into a parsing table.

Recursive Descent Parsing with Backtracking

Similar to regular recursive descent approach, but uses less or no lookahead symbol. Instead, it allows *backtracking* when gets to a dead end.

Example:

$S \rightarrow$ c A d
$A \rightarrow$ a b | a
Input: c a d

Input	*Parsing Action*
—	select $S \rightarrow$ c
c a d	match c
a d	select $A \rightarrow$ a b
a d	match a;
d	fail to match b; *backtrack!*
a d	select $A \rightarrow$ a
a d	match a
d	match d
—	accept

The Main Question

How to find lookahead symbols for a production $P \rightarrow \alpha\beta_1 \cdots \beta_k$?

Finding Lookahead — Simple Case

- The first symbol on the rhs is a *distinctive* terminal, *i.e.* no other production of the same nonterminal starts with the same symbol.

This symbol then is the lookahead for this production.

Example:

1. *Program* $\rightarrow$ begin *StmtList* end
4. *Stmt* $\rightarrow$ simpleS
5. *Stmt* $\rightarrow$ begin *StmtList* end

```
void Stmt() {
  // <clause 1> Stmt -> simpleS
  if (nextToken is "simpleS")
    match("simple_statement");
  // <clause 2> Stmt -> begin StmtList end
  if (nextToken is "begin")
    { match("begin"); call StmtList(); match("end"); }
}
```


Finding Lookahead — Difficult Case 1

- The first symbol on the rhs is a nonterminal.

In this case, there is no direct lookahead available. However, the nonterminal can derive the needed lookahead — the *first* symbols that can be derived from the nonterminal are the lookahead.

Example:

2. *StmtList* $\rightarrow$ *Stmt* ; *StmtList*
4. *Stmt* $\rightarrow$ simpleS
5. *Stmt* $\rightarrow$ begin *StmtList* end

```
void StmtList() {
  // <clause 1> StmtList -> Stmt ; StmtList
  if (nextToken is "simpleS" or "begin")
    { call Stmt(); call StmtList(); }
}
```

Finding Lookahead — Difficult Case 2

- The rhs is an ϵ.

In this case, there is no symbol on the rhs at all. However, an ϵ-production is selected for an immediate removal of the lhs nonterminal from the current derivation sequence. Therefore, the symbols that can *follow* the lhs nonterminal in any derivation become the lookahead for the production.

Example:

3. *StmtList* $\rightarrow$ ϵ

Program $\rightarrow$ begin *StmtList* end $\rightarrow \cdots$

(For this grammar, end happens to be the only symbol that can appear right after *StmtList* in any derivations.)

```
void StmtList() {
  // <clause 2> StmtList -> ϵ
  if (nextToken is "end")
    return;
}
```


First and Follow Sets

- *First* Set:

  Given a production $A \rightarrow \alpha$, this set consists of the *first symbol* of every sentence that can be generated from α.

  $$\boxed{First(\alpha) = \{a \mid a \in V_t \text{ and } \alpha \stackrel{*}{\Rightarrow} a\beta\}}$$

- *Follow* Set:

  Given a nonterminal A, this is the set of possible terminal symbols that can *follow* A in some legal derivations.

  $$\boxed{Follow(A) = \{a \mid a \in V_t \text{ and } S \stackrel{+}{\Rightarrow} \alpha A a\beta\}}$$

Production Prediction

- *Nullable* Predicate:

  Given a nonterminal A, we want to know if ϵ can be derived from A.

  $$\boxed{Nullable(A) = \text{true if } A \text{ can derive } \epsilon}$$

- *Lookahead* Set (a.k.a. *Predict* Set):

  Given a production $A \rightarrow \alpha$, this is the set of lookahead terminal symbols that predict the production.

  $$\boxed{\begin{array}{l} Lookahead(A \rightarrow \alpha) \\ = \begin{cases} First(\alpha) & \text{if } \neg Nullable(\alpha) \\ First(\alpha) \cup Follow(A) & \text{otherwise} \end{cases} \end{array}}$$

Computing First Sets

$$\boxed{First(\alpha) = \{a \mid a \in V_t \text{ and } \alpha \stackrel{*}{\Rightarrow} a\beta\}}$$

- $First(b\beta) = \{b\}$ for any terminal b and any string β
- $First(B\beta) = \begin{cases} First(B) & \text{if not } Nullable(B) \\ First(B) \cup First(\beta) & \text{otherwise} \end{cases}$

  Assume $B \rightarrow \beta_1 \mid \beta_1 \mid \cdots \mid \beta_k$ are the productions of B, then

- $First(B) = First(\beta_1) \cup \text{FIRST}(\beta_2) \cup \cdots \cup First(\beta_k)$

Computing Follow Sets

$$\boxed{Follow(A) = \{a \mid a \in V_t \text{ and } S \stackrel{+}{\Rightarrow} \alpha A a\beta\}}$$

We don't need to enumerate all derivations to find out all symbols that can follow a nonterminal. Instead, we can find the same information from productions.

- If $\exists\, A \rightarrow \alpha B\beta$, then everything in $First(\beta)$ is placed in $Follow(B)$.
- If $\exists\, A \rightarrow \alpha B$, or $A \rightarrow \alpha B\beta$ and $Nullable(\beta)$, then everything in $Follow(A)$ is placed in $Follow(B)$.

Example

0. *Program0* $\rightarrow$ *Program* \$
1. *Program* $\rightarrow$ begin *StmtList* end
2. *StmtList* $\rightarrow$ *Stmt* ; *StmtList*
3. *StmtList* $\rightarrow$ ϵ
4. *Stmt* $\rightarrow$ simpleS
5. *Stmt* $\rightarrow$ begin *StmtList* end

First(begin *StmtList* end) = {begin}
First(*Stmt*; *StmtList*) = *First*(*Stmt*) = {simpleS, begin}
First(ϵ) = {}
First(simpleS) = {simpleS}
First(begin *StmtList* end) = {begin}

Follow(*Program*) = {}
Follow(*StmtList*) = {end}
Follow(*Stmt*) = {;}

Nullable(begin *StmtList* end) = no
Nullable(*Stmt*; *StmtList*) = no
Nullable(ϵ) = yes
Nullable(simpleS) = no

Nullable(begin *StmtList* end) = no

Example (cont.)

$$\textit{Lookahead}(A \to \alpha) = \begin{cases} \textit{First}(\alpha) & \text{if } \neg \textit{Nullable}(\alpha) \\ \textit{First}(\alpha) \cup \textit{Follow}(A) & \text{otherwise} \end{cases}$$

1. *Lookahead*(*Program* → begin *StmtList* end) = {begin}
2. *Lookahead*(*StmtList* → *Stmt* ; *StmtList*) = {simpleS, begin}
3. *Lookahead*(*StmtList* → ϵ) = {end}
4. *Lookahead*(*Stmt* → simpleS) = {simpleS}
5. *Lookahead*(*Stmt* → begin *StmtList* end) = {begin}

Constructing a Parsing Table

$$M : V_n \times V_t \to \textit{Productions} \cup \{\text{error}\}$$

$$M[A][t] = \begin{cases} A \to X_1 \cdots X_m & \text{if } t \in \textit{Lookahead}(A \to X_1 \cdots X_m) \\ \text{error} & \text{otherwise} \end{cases}$$

For our example:

	begin	simpleS	;	end	$
Program	1				
StmtList	2	2		3	
Stmt	5	4			

Problem with Left Recursions

Production	*First*	*Follow*	*Lookahead*
1. $E \to E + T$	id	+	id
2. $E \to T$	id	+	id
3. $T \to T * P$	id	+, *	id
4. $T \to P$	id	+, *	id
5. $P \to$ id	id	+, *	id

Multiple productions are predicted by the same lookahead symbol.

Parsing Table:

	id	+	*	$
E	1, 2			
T	3, 4			
P	5			

This parsing table contains *conflicting* entries. A parser cannot be constructed based on this table.

Solution: Eliminating Left Recursions

Recall the transformation rules:

Replace $A \to A\alpha \mid \beta$ with $A \to \beta A'$, $A' \to \alpha A' \mid \epsilon$

Example:

0. $E0 \to E\,\$$
1. $E \to E + T$
2. $E \to T$
3. $T \to T * P$
4. $T \to P$
5. $P \to$ id

$\Rightarrow$

0. $E0 \to E\,\$$
1. $E \to TE'$
2. $E' \to +TE'$
3. $E' \to \epsilon$
4. $T \to PT'$
5. $T' \to *PT'$
6. $T' \to \epsilon$
7. $P \to$ id

After Left-Recursion Eliminating

Production	*Nullable*	*First*	*Follow*	*Lookahead*
1. $E \to TE'$	no	id	$	id
2. $E' \to +TE'$	no	+	$	+
3. $E' \to \epsilon$	yes		$	$
4. $T \to PT'$	no	id	+	id
5. $T' \to *PT'$	no	*	$	*
6. $T' \to \epsilon$	yes		$	$
7. $P \to$ id	no	id	+, *, $	id

Parsing Table:

	id	+	*	$
E	1			
E'		2		3
T	4			
T'			5	6
P	7			

Problem with Common Prefix

1. $S \to$ if E then S end if ;
2. $S \to$ if E then S else S end if ;

Production	*First*	*Follow*	*Lookahead*
1. $S \to$ if E then S end if ;	if	end, else	if
2. $S \to$ if E then S else S end if ;	if	end, else	if

Multiple productions are predicted by the same lookahead symbol!

Parsing Table:

	if	end	else	⋯
S	1, 2			

There is a *conflict!*

Solution: Factoring Out the Common Prefix

1. $S \rightarrow$ if E then S T
2. $T \rightarrow$ end if ;
3. $T \rightarrow$ else S end if ;

Production	*First*	*Follow*	*Lookahead*
1. $S \rightarrow$ if E then S T	if	end, else	if
2. $T \rightarrow$ end if ;	end	end, else	end
3. $T \rightarrow$ else S end if ;	else	end, else	else

Parsing Table:

	if	end	else	...
S	1			
T		2	3	

The LL Parser Family

For all the previous examples, we assumed one lookahead symbol. They therefore all correspond to *LL(1)* — *LL(1) parsing tables*, *LL(1) parsers*, and *LL(1) grammars*.

Meaning of the Ls:

1st "L" — scanning the input from left to right.
2nd "L" — producing a leftmost derivation.

By varying the number of lookahead symbols, we can define a whole family of LL parsers:

- *LL(0) Parser* — a predictive parser with no lookahead
- *LL(1) Parser* — an predictive parser with one lookahead
- *LL(2) Parser* — an predictive parser with one lookahead
- ...
- *LL(k) Parser* — an predictive parser with k lookaheads

LL(1) Grammars and Parsers

A grammar is LL(1) iff all entries in the LL(1) parsing table contain unique prediction or an error flag.

LL(1) grammars are of special importance:

- Many programming languages have an LL(1) (or near-LL(1)) grammar.
- LL(1) parsers can be implemented efficiently.

Converting a Grammar into LL Form

This is a critical step for developing a top-down parser. It consists of the following tasks:

- eliminating grammar ambiguity
- eliminating left recursions
- factoring out common prefixes — to minimize the size of lookahead

Once we have an LL grammar, we can follow the steps discussed earlier to construct a top-down parser:

- removing extended BNF symbols
- computing First, Follow, and Lookahead sets
- writing recursive parsing routines or constructing a parsing table

Example: A Simple Language

Program → begin {*Decl*} *StmtList* end
Decl → var *IdList* ';'
StmtList → *Stmt* {*Stmt*}
Stmt → id := *Expr* ';'
| read '(' *IdList* ')' ';'
| write '(' *ExprList* ')' ';'
IdList → id {',' id}
ExprList → *Expr* {',' *Expr*}
Expr → *Expr* {*Op Expr*}
| '(' *Expr* ')' | id | num
Op → '+' | '−' | '*' | '/'

Example: Eliminating Grammar Ambiguity

Expr → *Expr* {*Op Expr*}
| '(' *Expr* ')' | id | num
Op → '+' | '−' | '*' | '/'

⇒

Expr → *Expr AddOp Term* | *Term*
Term → *Term MulOp Primary* | *Primary*
Primary → '(' *Expr* ')' | id | num
AddOp → '+' | '−'
MulOp → '*' | '/'

Example: Eliminating Left Recursions

Expr	$\rightarrow$	*Expr AddOp Term* | *Term*
Term	$\rightarrow$	*Term MulOp Primary* | *Primary*

$\Rightarrow$

Expr	$\rightarrow$	*Term* {*AddOp Term*}
Term	$\rightarrow$	*Primary* {*MulOp Primary*}

Example: Resulting in an LL(1) Grammar

Program	$\rightarrow$	begin {*Decl*} *StmtList* end
Decl	$\rightarrow$	var *IdList* ;
StmtList	$\rightarrow$	*Stmt* {*Stmt*}
Stmt	$\rightarrow$	id := *Expr* ;
Stmt	$\rightarrow$	read (*IdList*) ;
Stmt	$\rightarrow$	write (*ExprList*) ;
IdList	$\rightarrow$	id {',' id}
ExprList	$\rightarrow$	*Expr* {',' *Expr*}
Expr	$\rightarrow$	*Term* {*AddOp Term*}
Term	$\rightarrow$	*Primary* {*MulOp Primary*}
Primary	$\rightarrow$	(*Expr*) | id | num
AddOp	$\rightarrow$	+ | −
MulOp	$\rightarrow$	* | /

Example: Same Grammar in BNF

1	*Program*	$\rightarrow$	begin *OptDeclList StmtList* end
2,3	*OptDeclList*	$\rightarrow$	*Decl OptDeclList* | ϵ
4	*Decl*	$\rightarrow$	var *IdList* ;
5	*StmtList*	$\rightarrow$	*Stmt OptStmtList*
6,7	*OptStmtList*	$\rightarrow$	*Stmt OptStmtList* | ϵ
8	*Stmt*	$\rightarrow$	id := *Expr* ;
9	*Stmt*	$\rightarrow$	read (*IdList*) ;
10	*Stmt*	$\rightarrow$	write (*ExprList*) ;
11	*IdList*	$\rightarrow$	id *OptIdList*
12,13	*OptIdList*	$\rightarrow$	, id *OptIdList* | ϵ
14	*ExprList*	$\rightarrow$	*Expr OptExprList*
15,16	*OptExprList*	$\rightarrow$	, *Expr OptExprList* | ϵ
17	*Expr*	$\rightarrow$	*Term OptExpr*
18,19	*OptExpr*	$\rightarrow$	*AddOp Term OptExpr* | ϵ
20	*Term*	$\rightarrow$	*Primary OptTerm*
21,22	*OptTerm*	$\rightarrow$	*MulOp Primary OptTerm* | ϵ
23	*Primary*	$\rightarrow$	(*Expr*)
24	*Primary*	$\rightarrow$	id
25	*Primary*	$\rightarrow$	num
26,27	*AddOp*	$\rightarrow$	+ | −
28,29	*MulOp*	$\rightarrow$	* | /