Concepts Introduced in Chapter 4

- Grammars
  - Context-Free Grammars
  - Derivations and Parse Trees
  - Ambiguity, Precedence, and Associativity
- Top Down Parsing
  - Recursive Descent, LL
- Bottom Up Parsing
  - SLR, LR, LALR
- Yacc
- Error Handling

Grammars

$$G = (N, T, P, S)$$

1. N is a finite set of nonterminal symbols
2. T is a finite set of terminal symbols
3. P is a finite subset of

   $(N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$

   An element $(\alpha, \beta) \in P$ is written as

   $\alpha \rightarrow \beta$

   and is called a production.
4. S is a distinguished symbol in N and is called the start symbol.

Example of a Grammar

block $\rightarrow$ <u>begin</u> opt_stmts <u>end</u>

opt_stmts $\rightarrow$ stmt_list | ϵ

stmt_list $\rightarrow$ stmt_list; stmt | stmt

Advantages of Using Grammars

- Provides a precise, syntactic specification of a programming language.
- For some classes of grammars, tools exist that can automatically construct an efficient parser.
- These tools can also detect syntactic ambiguities and other problems automatically.
- A compiler based on a grammatical description of a language is more easily maintained and updated.

Role of a Parser in a Compiler

- Detects and reports any syntax errors.
- Produces a parse tree from which intermediate code can be generated.

Conventions Used for Specifying Grammars in the Text

- terminals
  - lower case letters early in the alphabet (a, b, c)
  - punctuation and operator symbols [(,), ',', +, −]
  - digits
  - boldface words (**if**, **then**)
- nonterminals
  - uppercase letters early in the alphabet (A, B, C)
  - S is the start symbol
  - lower case words

Conventions Used for Specifying Grammars in the Text (cont.)

- grammar symbols (nonterminals or terminals)
  - upper case letters late in the alphabet (X, Y, Z)
- strings of terminals
  - lower case letters late in the alphabet (u, v, ..., z)
- sentential form (string of grammar symbols)
  - lower case Greek letters (α, β, γ)

Chomsky Hierarchy

A grammar is said to be

1. <u>regular</u> if productions in P are all right-linear or are all left-linear

   a. <u>right-linear</u>

   $A \to wB$ or $A \to w$

   b. <u>left-linear</u>

   $A \to Bw$ or $A \to w$

   where $A, B \in N$ and $w \in T^*$

   Recognized by a finite automata (FA).

Chomsky Hierarchy (cont)

2. <u>context-free</u> if each production in P is of the form

   $A \to \alpha$ where $A \in N$ and $\alpha \in (N \cup T)^*$

   Recognized by a pushdown automata (PDA).

3. <u>context-sensitive</u> if each production in P is of the form

   $\alpha \to \beta$ where $|\alpha| \leq |\beta|$

   Recognized by a linear bounded automata (LBA).

4. <u>unrestricted</u> if each production in P is of the form

   $\alpha \to \beta$ where $\alpha \neq \varepsilon$

   Recognized by a Turing machine.

Derivation

Derivation - a sequence of replacements from the start symbol in a grammar by applying productions

Example: $E \to E + E$
$E \to E * E$
$E \to (E)$
$E \to - E$
$E \to id$

Derive - (id) from the grammar

$E \Rightarrow - E \Rightarrow - (E) \Rightarrow - (id)$

thus E derives $- (id)$

or $E \stackrel{+}{\Rightarrow} - (id)$

Derivation (cont.)

leftmost derivation - each step replaces the leftmost nonterminal

Derive id + id * id using leftmost derivation

$E \Rightarrow E + E \Rightarrow id + E \Rightarrow id + E * E \Rightarrow id + id * E \Rightarrow id + id * id$

L(G) - language generated by the grammar G

sentence of G - if $S \stackrel{+}{\Rightarrow} w$, where w is a string of terminals in L(G)

sentential form - if $S \stackrel{*}{\Rightarrow} \alpha$, where α may contain nonterminals

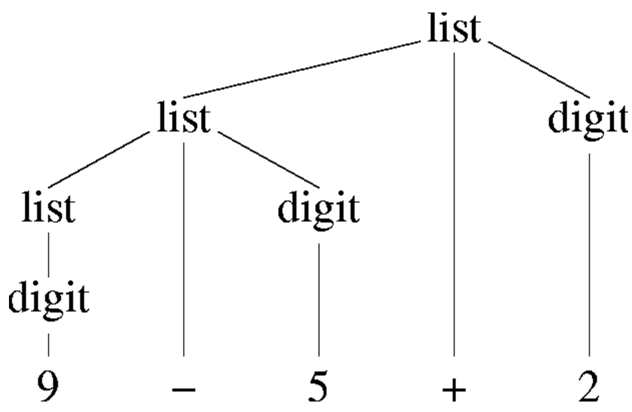
Parse Tree

A parse tree pictorially shows how the start symbol of a grammar derives a specific string in the language.

Given a context-free grammar, a parse tree has the properties:

1. The root is labeled by the start symbol.
2. Each leaf is labeled by a token or ε.
3. Each interior node is labeled by a nonterminal.
4. If A is a nonterminal labeling some interior node and $X_1, X_2, X_3, .., X_n$ are the labels of the children of that node from left to right, then $A \to X_1, X_2, X_3, .. X_n$ is a production of the grammar.

Example of a Parse Tree



list → list + digit | list – digit | digit

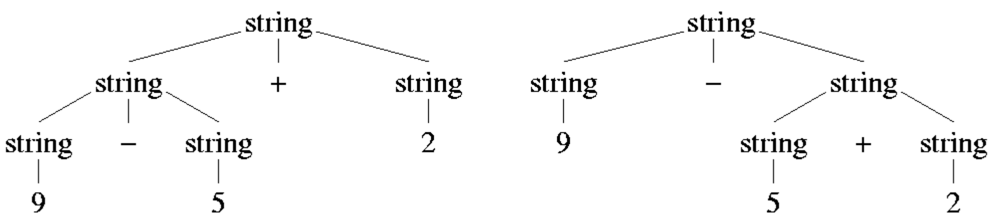
Parse Tree (cont.)

Yield - the leaves of the parse tree read from left to right or the string derived from the nonterminal at the root of the parse tree.

An ambiguous grammar is one that can generate two or more parse trees that yield the same string.

Example of an Ambiguous Grammar

string → string + string
string → string - string
string → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



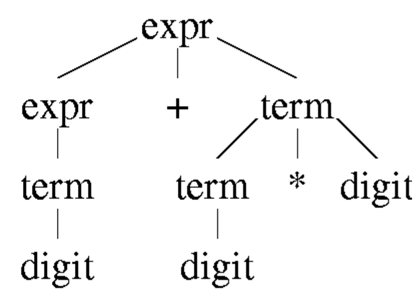
a. string → string + string → string - string + string
→ 9 – string + string → 9 – 5 + string → 9 – 5 + 2

b. string → string - string → 9 – string
→ 9 – string + string → 9 – 5 + string → 9 – 5 + 2

Precedence

By convention
9 + 5 * 2   * has higher precedence than + because it takes its operands before +

expr –> expr + term | term
term –> term * digit | digit



Precedence (cont.)

Different operators have the same precedence when they are defined as alternative productions of the same nonterminal.

expr $\rightarrow$ expr + term | expr – term | term
term $\rightarrow$ term * factor | term / factor | factor
factor $\rightarrow$ digit | (expr)

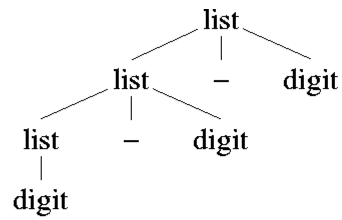
Associativity

By convention

9 – 5 – 2 left (operand with – on both sides, the operation on the left is performed first)

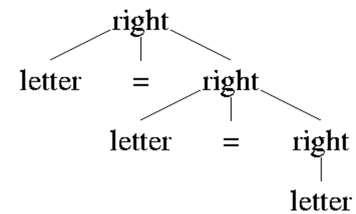
a = b = c right (operand with = on both sides, the operation on the right is performed first)

list –> list – digit
list –> digit



grows to the left

right –> letter = right
right –> letter



grows to the right

Eliminating Ambiguity

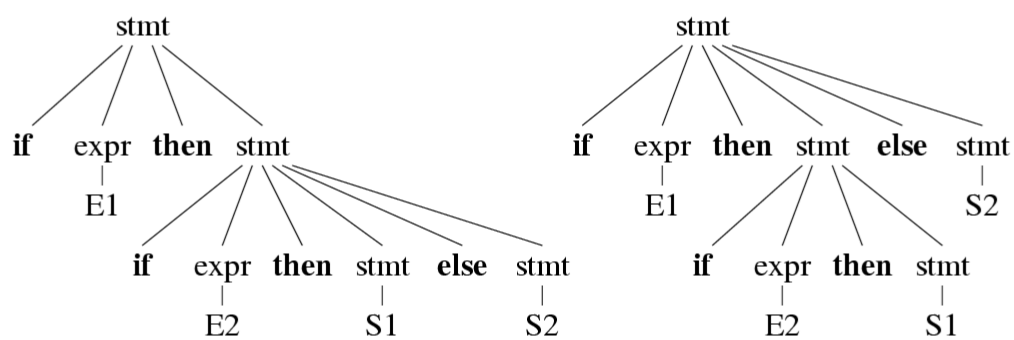
- Sometimes ambiguity can be eliminated by rewriting a grammar.

  stmt $\rightarrow$ **if** expr **then** stmt
  | **if** expr **then** stmt **else** stmt
  | other

- How do we parse:

  if E1 **then** if E2 **then** S1 **else** S2

Two Parse Trees for "if E1 then if E2 then S1 else S2"



Eliminating Ambiguity (cont.)

$$\begin{aligned}
\text{stmt} &\rightarrow \text{matched_stmt} \\
&\mid \text{unmatched_stmt} \\
\text{matched_stmt} &\rightarrow \textbf{if}\ \text{expr}\ \textbf{then}\ \text{matched_stmt}\ \textbf{else}\ \text{matched_stmt} \\
&\mid \text{other} \\
\text{unmatched_stmt} &\rightarrow \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stmt} \\
&\mid \textbf{if}\ \text{expr}\ \textbf{then}\ \text{matched_stmt}\ \textbf{else}\ \text{unmatched_stmt}
\end{aligned}$$

Parsing

- universal
- top-down
  - recursive descent
  - LL
- bottom-up
  - operator precedence
  - LR
    - SLR
    - canonical LR
    - LALR

Top-Down vs Bottom-Up Parsing

- top-down
  - Have to eliminate left recursion in the grammar.
  - Have to left factor the grammar.
  - Resulting grammars are harder to read and understand.
- bottom-up
  - Difficult to implement by hand, so a tool is needed.

Top-Down Parsing

Starts at the root and proceeds towards the leaves.

Recursive-Descent Parsing - a recursive procedure is associated with each nonterminal in the grammar.

Example

$$\begin{aligned}
\text{type} &\rightarrow \text{simple} \mid \uparrow\underline{\text{id}} \mid \underline{\text{array}}\ \underline{[}\ \text{simple}\ \underline{]}\ \underline{\text{of}}\ \text{type} \\
\text{simple} &\rightarrow \underline{\text{integer}} \mid \underline{\text{char}} \mid \underline{\text{num}}\ \underline{\text{dotdot}}\ \underline{\text{num}}
\end{aligned}$$

Example of Recursive Descent Parsing

```
void type() {
    if ( lookahead == INTEGER || lookahead == CHAR ||
        lookahead == NUM)
        simple();
    else if (lookahead == '^') {
        match('^');
        match(ID);
    }
    else if (lookahead == ARRAY) {
        match(ARRAY);
        match('[');
        simple();
        match(']');
        match(OF);
        type();
    }
    else
        error();
}
```

Example of Recursive Descent Parsing (cont.)

```
void simple() {
    if (lookahead == INTEGER)
        match(INTEGER);
    else if (lookahead == CHAR)
        match(CHAR);
    else if (lookahead== NUM) {
        match(NUM);
        match(DOTDOT);
        match(NUM);
    }
    else
        error();
}
```

```
void match(token t)
{
    if (lookahead == t)
        lookahead = nexttoken();
    else
        error();
}
```

Top-Down Parsing (cont.)

- Predictive parsing needs to know what first symbols can be generated by the right side of a production.
- FIRST(α) - the set of tokens that appear as the first symbols of one or more strings generated from α. If α is ε or can generate ε, then ε is also in FIRST(α).
- Given a production

$$A \rightarrow \alpha \mid \beta$$

predictive parsing requires FIRST(α) and FIRST(β) to be disjoint.

Eliminating Left Recursion

- Recursive descent parsing loops forever on left recursion.
- Immediate Left Recursion

Replace $A \rightarrow A\alpha \mid \beta$ with $A \rightarrow \beta A'$

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

Example:

	<u>A</u>	<u>α</u>	<u>β</u>
$E \rightarrow E + T \mid T$	E	+T	T
$T \rightarrow T * F \mid F$	T	*F	F
$F \rightarrow (E) \mid id$			

becomes

$E \rightarrow TE'$

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

$T \rightarrow FT'$

Eliminating Left Recursion (cont.)

- What if a grammar is not immediately left recursive?

  $A \overset{+}{\Rightarrow} A\alpha$

- For instance:

  $A \rightarrow B\alpha 1 \mid \alpha 4$

  $B \rightarrow C\alpha 2$

  $C \rightarrow A\alpha 3$

- For example:

  $A \Rightarrow B\alpha 1 \Rightarrow C\alpha 2\alpha 1 \Rightarrow A\alpha 3\alpha 2\alpha 1$

Eliminating Left Recursion (cont.)

In general, to eliminate left recursion given $A_1, A_2, \ldots, A_n$

for i = 1 to n do
  for j = 1 to i-1 do
    replace each $A_i \rightarrow A_j \gamma$ with $A_i \rightarrow \delta_1 \gamma \mid \ldots \mid \delta_k \gamma$
    where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are the current A_j productions
  end for
  eliminate immediate left recursion in the A_i productions
  eliminate ε transitions in the A_i productions
end for

This fails only if cycles ($A \overset{+}{\Rightarrow} A$) or $A \rightarrow \varepsilon$ for some A.

Example of Eliminating Left Recursion

1. $X \rightarrow YZ \mid a$
2. $Y \rightarrow ZX \mid Xb$
3. $Z \rightarrow XY \mid ZZ \mid a$

A1 = X A2 = Y A3 = Z

i = 1 (eliminate immediate left recursion)
  nothing to do

Example of Eliminating Left Recursion (cont.)

i = 2, j = 1

$Y \rightarrow Xb \Rightarrow Y \rightarrow ZX \mid YZb \mid ab$

now eliminate immediate left recursion

$Y \rightarrow ZXY' \mid abY'$

$Y' \rightarrow ZbY' \mid \varepsilon$

now eliminate ε transitions

$Y \rightarrow ZXY' \mid abY' \mid ZX \mid ab$

$Y' \rightarrow ZbY' \mid Zb$

i = 3, j = 1

$Z \rightarrow XY \Rightarrow Z \rightarrow YZY \mid aY \mid ZZ \mid a$

Example of Eliminating Left Recursion (cont.)

i = 3, j = 2

$Z \rightarrow YZY \;\Rightarrow\; Z \rightarrow ZXY'ZY \mid abY'ZY \mid ZXZY$
$\mid abZY \mid aY \mid ZZ \mid a$

now eliminate immediate left recursion

$Z \rightarrow abY'ZYZ' \mid abZYZ' \mid aYZ' \mid aZ'$

$Z' \rightarrow XY'ZYZ' \mid XZYZ' \mid ZZ' \mid \varepsilon$

eliminate ε transitions

$Z \rightarrow abY'ZYZ' \mid abY'ZY \mid abZYZ' \mid abZY \mid aY$
$\mid aYZ' \mid aZ' \mid a$

$Z' \rightarrow XY'ZYZ' \mid XY'ZY \mid XZYZ' \mid XZY \mid ZZ'$
$\mid Z$

Left-Factoring

$$A \rightarrow \alpha\beta \mid \alpha\gamma \quad \Rightarrow \quad A \rightarrow \alpha A'$$
$$A' \rightarrow \beta \mid \gamma$$

Example:

Left factor

stmt → <u>if</u> cond <u>then</u> stmt <u>else</u> stmt
| <u>if</u> cond <u>then</u> stmt

becomes

stmt → <u>if</u> cond <u>then</u> stmt E
E → <u>else</u> stmt | ε

Grammars must be left factored for predictive parsing so we will know which production to choose.

Nonrecursive Predictive Parsing

- Instead of recursive descent, predictive parsing can be table-driven and use an explicit stack. It uses

  1. a stack of grammar symbols ($ on bottom)
  2. a string of input tokens ($ on end)
  3. a parsing table [NT, T] of productions

Algorithm for Nonrecursive Predictive Parsing

1. If top == input == $ then accept
2. If top == input then
   pop top off the stack
   advance to next input symbol
   goto 1
3. If top is nonterminal
   fetch M[top, input]
   If a production
      replace top with rhs of production
   Else
      parse fails
   goto 1
4. Parse fails

First

FIRST(α) = the set of terminals that begin strings derived from α. If α is ε or generates ε, then ε is also in FIRST(α).

1. If X is a terminal then FIRST(X) = {X}
2. If $X \rightarrow a\alpha$, add a to FIRST(X)
3. If $X \rightarrow \varepsilon$, add ε to FIRST(X)
4. If $X \rightarrow Y_1, Y_2, \ldots, Y_k$ and $Y_1, Y_2, \ldots, Y_{i-1} \stackrel{*}{\Rightarrow} \varepsilon$ where $i \le k$

   Add every non ε in FIRST(Y_i) to FIRST(X)

   If $Y_1, Y_2, \ldots, Y_k \stackrel{*}{\Rightarrow} \varepsilon$, add ε to FIRST(X)

FOLLOW

FOLLOW(A) = the set of terminals that can immediately follow A in a sentential form.

1. If S is the start symbol, add \$ to FOLLOW(S)
2. If $A \rightarrow \alpha B\beta$, add FIRST($\beta$) - {$\varepsilon$} to FOLLOW(B)
3. If $A \rightarrow \alpha B$ or $A \rightarrow \alpha B\beta$ and $\beta \stackrel{*}{\Rightarrow} \varepsilon$,
   add FOLLOW(A) to FOLLOW(B)

Example of Calculating FIRST and FOLLOW

Production	FIRST	FOLLOW
$E \rightarrow TE'$	{ (, id }	{), \$ }
$E' \rightarrow +TE' \mid \varepsilon$	{ +, ε }	{), \$ }
$T \rightarrow FT'$	{ (, id }	{ +,), \$ }
$T' \rightarrow *FT' \mid \varepsilon$	{*, ε }	{ +,), \$ }
$F \rightarrow (E) \mid id$	{ (, id }	{*, +,), \$ }

Another Example of Calculating FIRST and FOLLOW

Production	FIRST	FOLLOW
$X \rightarrow Ya$	{ }	{ }
$Y \rightarrow ZW$	{ }	{ }
$W \rightarrow c \mid \varepsilon$	{ }	{ }
$Z \rightarrow a \mid bZ$	{ }	{ }

Constructing Predictive Parsing Tables

For each $A \rightarrow \alpha$ do

1. Add $A \rightarrow \alpha$ to $M[A, a]$ for each a in FIRST(α)
2. If ε is in FIRST(α)
   a. Add $A \rightarrow \alpha$ to $M[A, b]$ for each b in FOLLOW(A)
   b. If \$ is in FOLLOW(A) add $A \rightarrow \alpha$ to $M[A, \$]$
3. Make each undefined entry of M an error.

LL(1)

First "L"	-	scans input from left to right
Second "L"	-	produces a leftmost derivation
1	-	uses one input symbol of lookahead at each step to make a parsing decision

A grammar whose predictive parsing table has no multiply-defined entries is LL(1).

No ambiguous or left-recursive grammar can be LL(1).

When Is a Grammar LL(1)?

A grammar is LL(1) iff for each set of productions where $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$, the following conditions hold.

1. $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \varnothing$ where $1 \le i \le n$ and $1 \le j \le n$ and $i \ne j$
2. If $\alpha_i \overset{*}{\Rightarrow} \varepsilon$ then
   a. $\alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_n$ does not $\overset{*}{\Rightarrow} \varepsilon$
   b. $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \varnothing$ where $j \ne i$ and $1 \le j \le n$

Checking If a Grammar is LL(1)

Production	FIRST	FOLLOW
$S \rightarrow iEtSS' \mid a$	{ i, a }	{ e, \$ }
$S' \rightarrow eS \mid \varepsilon$	{ e, ε }	{ e, \$ }
$E \rightarrow b$	{ b }	{ t }

Nonterminal	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow eS$ $S' \rightarrow \varepsilon$			$S' \rightarrow \varepsilon$
E		$E \rightarrow b$				

So this grammar is not LL(1).

Shift-Reduce Parsing

- Shift-reduce parsing is bottom-up.
  - Attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root.
- A "handle" is a substring that matches the rhs of a production.
- A "shift" moves the next input symbol on a stack.
- A "reduce" replaces the rhs of a production that is found on the stack with the nonterminal on the left of that production.
- A "viable prefix" is the set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser.
- Shift reduce parsing includes
  - operator-precedence parsing
  - LR parsing

Model of an LR Parser

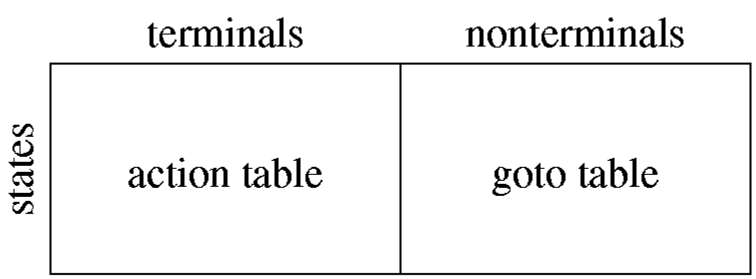
- See Figure 4.35.

- Each s_i is a state.
- Each X_i is a grammar symbol (when implemented these items do not appear in the stack).
- Each a_i is an input symbol.

- All LR parsers can use the same algorithm (code).
- The action and goto tables are different for each LR parser.

Model of an LR Parser (cont.)

- A shift pushes a state on the stack and processes an input symbol.
- A reduce pops states off the stack and pushes one state back on the stack.



LR(k) Parsing

"L" - scans input from left to right
"R" - constructs a rightmost derivation in reverse
"k" - uses k symbols of lookahead at each step to make a parsing decision

Uses a stack of alternating states and grammar symbols. The grammar symbols are optional. Uses a string of input symbols ($ on end).

LR (k) Parsing (cont.)

If config == (s_0 X_1 s_1 X_2 s_2 ... X_m s_m, a_i a_{i+1} ... a_n\$)

1. if action [s_m, a_i] == shift s then

   new config is (s_0 X_1 s_1 X_2 s_2 ... X_m s_m a_is, a_{i+1} ... a_n\$)

2. if action [s_m, a_i] == reduce $A \rightarrow \beta$ and

   goto [s_{m-r}, A] == s (where r is the length of β) then

   new config is (s_0 X_1 s_1 X_2 s_2...X_{m-r} s_{m-r} As, a_i a_{i+1}...a_n\$)

3. if action [s_m, a_i] == ACCEPT then stop

4. if action [s_m, a_i] == ERROR then attempt recovery

Can resolve some shift-reduce conflicts with lookahead.

ex: LR(1)

Can resolve others in favor of a shift.

ex: $S \rightarrow iCtS \mid iCtSeS$

Advantages of LR Parsing

- LR parsers can recognize almost all programming language constructs expressed in context -free grammars.
- Efficient and requires no backtracking.
- Is a superset of the grammars that can be handled with predictive parsers.
- Can detect a syntactic error as soon as possible on a left-to-right scan of the input.

LR Parsing Example

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

See Fig 4.37.

It produces rightmost derivation in reverse:

$E \rightarrow E + T \rightarrow E + F \rightarrow E + id \rightarrow T + id \rightarrow T * F + id$
$\rightarrow T * id + id \rightarrow F * id + id \rightarrow id * id + id$

Calculating the Sets of LR(0) Items

LR(0) item - production with a dot at some position in the rhs indicating how much has been parsed

Example:

$A \rightarrow BC$ has 3 possible LR(0) items

$A \rightarrow \cdot BC$

$A \rightarrow B \cdot C$

$A \rightarrow BC \cdot$

$A \rightarrow \varepsilon$ has 1 possible item

$A \rightarrow \cdot$

3 operations required to construct the sets of LR(0) items: (1) closure, (2) goto, and (3) augment

Example of Computing the Closure of a Set of LR(0) Items

Grammar	Closure (I_0) for $I_0 = \{E' \to \cdot E\}$
$E' \to E$	$E' \to \cdot E$
$E \to E + T \mid T$	$E \to \cdot E + T$
$T \to T * F \mid F$	$E \to \cdot T$
$F \to (E) \mid id$	$T \to \cdot T * F$
	$T \to \cdot F$
	$F \to \cdot(E)$
	$F \to \cdot\ id$

Calculating Goto of a Set of LR(0) Items

Calculate goto (I,X) where I is a set of items and X is a grammar symbol.

Take the closure (the set of items of the form $A \to \alpha X \cdot \beta$) where $A \to \alpha \cdot X \beta$ is in I.

Grammar	Goto $(I_1,+)$ for $I_1 = \{E' \to E\cdot, E \to E\cdot{+}T\}$
$E' \to E$	$E \to E + \cdot T$
$E \to E + T \mid T$	$T \to \cdot T * F$
$T \to T * F \mid F$	$T \to \cdot F$
$F \to (E) \mid id$	$F \to \cdot(E)$
	$F \to \cdot id$
	Goto $(I_2,*)$ for $I_2 = \{E \to T\cdot, T \to T\cdot{*}F\}$
	$T \to T * \cdot F$
	$F \to \cdot(E)$
	$F \to \cdot id$

Augmenting the Grammar

Given grammar G with start symbol S, then an augmented grammar G´ is G with a new start symbol S´ and new production $S' \to S$.

Analogy of Calculating the Set of LR(0) Items with Converting an NFA to a DFA

Constructing the set of items is similar to converting an NFA to a DFA. Each state in the NFA is an individual item. The closure (I) for a set of items is similar to the ε-closure of a set of NFA states. Each set of items is now a DFA state and goto (I,X) gives the transition from I on symbol X.

Constructing SLR Parsing Tables

Let C = {I_0, I_1, ..., I_n} be the parser states.

1. If [$A \rightarrow \alpha \cdot a\beta$] is in I_i and goto (I_i, a) = I_j then set action [i, a] to 'shift j'.
2. If [$A \rightarrow \alpha \cdot$] is in I_i, then set action [i, a] to 'reduce $A \rightarrow \alpha$' for all a in the FOLLOW(A). A may not be S´.
3. If [$S' \rightarrow S\cdot$] is in I_i, then set action [i, $] to 'accept'.
4. If goto (I_i, A)=I_j, then set goto[i, A] to j.
5. Set all other table entries to 'error'.
6. The initial state is the one holding [$S' \rightarrow \cdot S$].

LR(1)

The unambiguous grammar

$S \rightarrow L = R \mid R$

$L \rightarrow *R \mid id$

$R \rightarrow L$

is not SLR.

See Fig 4.39.

action[2, =] can be a "shift 6" or "reduce $R \rightarrow L$"
FOLLOW(R) contains "=" but no form begins with "R="

LR (1) (cont.)

Solution - split states by adding LR(1) lookahead

form of an item

[$A \rightarrow \alpha \cdot \beta$,a]

where $A \rightarrow \alpha\beta$ is a production and
'a' is a terminal or endmarker $

Closure(I) is now slightly different

```
repeat
    for each item [A→α·Bβ, a] in I,
        each production B→ γ in the grammar,
        and each terminal b in FIRST(βa) do
        add [B → ·γ, b] to I (if not there)
until no more items can be added to I
```

Start the construction of the set of LR(1) items by computing the closure of {[$S' \rightarrow \cdot S$, $]}.

LR(1) Example

(0) 1. $S' \rightarrow S$

(1) 2. $S \rightarrow CC$

(2) 3. $C \rightarrow cC$

(3) 4. $C \rightarrow d$

I_0:	[$S' \rightarrow \cdot S$, $]	goto (S)= I_1
	[$S \rightarrow \cdot CC$, $]	goto (C)= I_2
	[$C \rightarrow \cdot cC$, c/d]	goto (c) = I_3
	[$C \rightarrow \cdot d$, c/d]	goto (d) = I_4
I_1:	[$S' \rightarrow S\cdot$, $]	
I_2:	[$S \rightarrow C \cdot C$, $]	goto (C)= I_5
	[$C \rightarrow \cdot cC$, $]	goto (c) = I_6
	[$C \rightarrow \cdot d$, $]	goto (d)= I_7

LR(1) Example (cont.)

I_3: $[C \rightarrow c \cdot C, c/d]$ goto (C) $= I_8$
$[C \rightarrow \cdot cC, c/d]$ goto (c) $= I_3$
$[C \rightarrow \cdot d, c/d]$ goto (d) $= I_4$
I_4: $[C \rightarrow d\cdot, c/d]$
I_5: $[S \rightarrow CC\cdot, \$]$
I_6: $[C \rightarrow c \cdot C, \$]$ goto (C) $= I_9$
$[C \rightarrow \cdot cC, \$]$ goto (c) $= I_6$
$[C \rightarrow \cdot d, \$]$ goto (d) $= I_7$
I_7: $[C \rightarrow d\cdot, \$]$
I_8: $[C \rightarrow cC\cdot, c/d]$
I_9: $[C \rightarrow cC\cdot, \$]$

Constructing the LR(1) Parsing Table

Let $C = \{I_0, I_1, ..., I_n\}$

1. If $[A\rightarrow\alpha\cdot a\beta, b]$ in I_i and goto(I_i, a) = I_j then set action[i, a] to “shift j”.
2. If $[A\rightarrow\alpha\cdot, a]$ is in I_i, then set action[i, a] to 'reduce $A\rightarrow\alpha$'. A may not be S´.
3. If $[S´\rightarrow S\cdot, \$]$ is in I_i, then set action[i, \$] to “accept.”
4. If goto(I_i, A) = I_j, then set goto[i, A] to j.
5. Set all other table entries to error.
6. The initial state is the one holding $[S´\rightarrow\cdot S, \$]$

Constructing LALR Parsing Tables

- Combine LR(1) sets with the same sets of the first parts (ignore lookahead).
- Table is the same size as SLR.
- Will not introduce shift-reduce conflicts since shifts depend only on the core and don't use lookahead.
- May introduce reduce-reduce conflicts but seldom do for grammars describing programming languages.

•Last example collapses to table shown in Fig 4.43.

•Algorithms exist that skip constructing all the LR(1) sets of items.

Compaction of LR Parsing Tables

- A typical programming language may have 50 to 100 terminals and over 100 productions. This can result in several hundred states and a very large action table.
- One technique to save space is to recognize that many rows of the action table are identical. Can create a pointer for each state with the same actions so that it points to the same location.
- Could save further space by creating a list for the actions of each state, where the list consists of terminal-symbol/action pairs. This would eliminate the blank or error entries in the action table. While this technique would save a lot of space, the parser would be much slower.

Using Ambiguous Grammars

1. $E \rightarrow E + E$		$E \rightarrow E + T \mid T$
2. $E \rightarrow E * E$	instead of	$T \rightarrow T * F \mid F$
3. $E \rightarrow (E)$		$F \rightarrow (E) \mid id$
4. $E \rightarrow id$		

See Figure 4.48.

Advantages:

- Grammar is easier to read.
- Parser is more efficient.

Using Ambiguous Grammars (cont.)

Can use precedence and associativity to solve the problem.

See Fig 4.49.

shift / reduce conflict in state action[7,+]=(s4,r1)
s4 = shift 4 or $E \rightarrow E \cdot + E$
r1 = reduce 1 or $E \rightarrow E + E \cdot$

id + id + id
↑ cursor here

action[7,*]=(s5,r1)
action[8,+]=(s4,r2) action[8,*]=(s5,r2)

Another Ambiguous Grammar

0. $S' \rightarrow S$

1. $S \rightarrow iSeS$

2. $S \rightarrow iS$

3. $S \rightarrow a$

See Figure 4.50.

action[4,e]=(s5,r2)

Ambiguities from Special-Case Productions

$E \rightarrow E$ sub E sup E
$E \rightarrow E$ sub E
$E \rightarrow E$ sup E
$E \rightarrow \{ E \}$
$E \rightarrow c$

Ambiguities from Special-Case Productions (cont)

1. $E \rightarrow E$ sub E sup E
2. $E \rightarrow E$ sub E
3. $E \rightarrow E$ sup E
4. $E \rightarrow \{ E \}$
5. $E \rightarrow c$

FIRST(E) = { '{', c}
FOLLOW(E) = {sub,sup,'}',$}

sub, sup have equal precedence
and are right associative

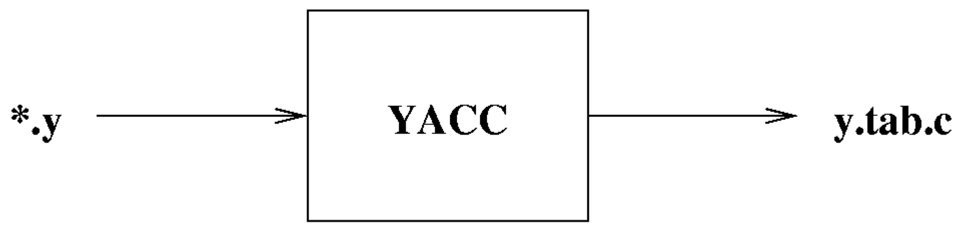
action[7,sub]=(s4,r2)
action[8,sub]=(s4,r3)
action[11,sub]=(s4,r1,r3)
action[11,}]=(r1,r3)

action[7,sup]=(s10,r2)
action[8,sup]=(s5,r3)
action[11,sup]=(s5,r1,r3)
action[11,$]=(r1,r3)

YACC

Yacc source program

declarations
%%
translation rules
%%
supporting C-routines



YACC Declarations

- In declarations:
  - Can put ordinary C declarations in

    %{

    ...

    %}
  - Can declare tokens using
    - %token
    - %left
    - %right
  - Precedence is established by the order the operators are listed (low to high).

YACC Translation Rules

- Form

  A : Body ;

  where A is a nonterminal and Body is a list of nonterminals and terminals.
- Semantic actions can be enclosed before or after each grammar symbol in the body.
- Yacc chooses to shift in a shift/reduce conflict.
- Yacc chooses the first production in a reduce/reduce conflict.

Yacc Translation Rules (cont.)

- When there is more than one rule with the same left hand side, a '|' can be used.

```
A :  B C D ;
A :  E F ;
A :  G ;
```

=>

```
A :  B C D
  |  E F
  |  G
  ;
```

Example of a Yacc Specification

```
%token IF ELSE NAME     /* defines multicharacter tokens */
%right '='              /* low precedence, a=b=c shifts */
%left '+' '-'           /* mid precedence, a-b-c reduces */
%left '*' '/'           /* high precedence, a/b/c reduces */
%%
stmt    : expr ';'
        | IF '(' expr ')' stmt
        | IF '(' expr ')' stmt ELSE stmt
        ;   /* prefers shift to reduce in shift/reduce conflict */
expr    : NAME '=' expr         /* assignment */
        | expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        | '-' expr  %prec  '*'  /* can override precedence */
        | NAME
        ;
%%  /* definitions of yylex, etc. can follow */
```

Yacc Actions

- Actions are C code segments enclosed in { } and may be placed before or after any grammar symbol in the right hand side of a rule.
- To return a value associated with a rule, the action can set $$.
- To access a value associated with a grammar symbol on the right hand side, use $i, where i is the position of that grammar symbol.
- The default action for a rule is

  { $$ = $1; }

Syntax Error Handling

- Errors can occur at many levels
  - lexical - unknown operator
  - syntactic - unbalanced parentheses
  - semantic - variable never declared
  - logical - dereference a null pointer
- Goals of error handling in a parser
  - detect and report the presence of errors
  - recover from each error to be able to detect subsequent errors
  - should not slow down the compilation of correct programs

Syntax Error Handling (cont.)

- Viable–prefix property - detect an error as soon as the parser sees a prefix of the input that is not a prefix of any string in the language.

Error-Recovery Strategies

- Panic-mode - skip until one of a synchronizing set of tokens is found (e.g. ';', "end"). Is very simple to implement but may miss detection of some errors (when more than one error in a single statement).
- Phrase-level - replace prefix of remaining input by a string that allows the parser to continue. Hard for the compiler writer to anticipate all error situations.
- Error productions - augment the grammar of the source language to include productions for common errors. When production is used, an appropriate error diagnostic would be issued. Feasible to only handle a limited number of errors.

Error-Recovery Strategies (cont)

- Global correction - choose minimal sequence of changes to allow a least-cost correction. Often considered too costly to actually be implemented in a parser. Also the closest correct program may not be what the programmer intended.

Error-Recovery in Predictive Parsing

- It is easier to recover from an error in a nonrecursive predictive parser than using recursive descent.
- Panic-mode recovery
  - Assume the nonterminal A is on the stack when we encounter an error. As a starting point can place all symbols in FOLLOW(A) into the synchronizing set for the nonterminal A. May also wish to add symbols that begin higher-level constructs to the synchronizing set of lower-level constructs. If a terminal is on top of the stack, then can pop the terminal and issue a message stating that the terminal was discarded.

Error-Recovery in Predictive Parsing (cont.)

Phrase-level recovery

- Can be implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. The compiler writer would attempt to address each situation appropriately (issue error message and update input symbols and pop from the stack).

Error-Recovery in LR Parsing

- Canonical LR Parser - will never make a single reduction before recognizing an error.
- SLR & LALR Parsers - may make extra reductions but will never shift an erroneous input symbol on the stack.
- Panic-mode recovery - scan down stack until a state with a goto on a particular nonterminal representing a major program construct (e.g. expression, statement, block, etc.) is found. Input symbols are discarded until one is found that is in the FOLLOW of the nonterminal. The parser then pushes on the state in goto. Thus, it attempts to isolate the construct containing the error.

Error-Recovery in LR Parsing (cont)

- Phrase-level recovery - Implement an error recovery routine for each error entry in the table.
- Error productions - Used in YACC. Pops symbols until topmost state has an error production, then shifts error onto stack. Then discards input symbols until it finds one that allows parsing to continue. The semantic routine with an error production can just produce a diagnostic message.