# 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

$$expression \rightarrow expression + term$$

$$expression \rightarrow expression - term$$

$$expression \rightarrow term$$

$$term \rightarrow term * factor$$

$$term \rightarrow term / factor$$

$$term \rightarrow factor$$

$$factor \rightarrow ( expression )$$

$$factor \rightarrow \mathbf{id}$$

# 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.

*followed by Fig. 4.1*

# Conventions 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 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 it is

   where each production in P has the form

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

   $A \rightarrow wB$ or $A \rightarrow w$

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

   $A \rightarrow Bw$ or $A \rightarrow w$

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

# Chomsky Hierarchy (cont)

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

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

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

   $\alpha \rightarrow \beta$ where $|\alpha| \leq |\beta|$

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

   $\alpha \rightarrow \beta$ where $\alpha \neq \varepsilon$

# Derivation

- Derivation
  - a sequence of replacements from the start symbol in a grammar by applying productions
  - $E \rightarrow E + E \mid E * E \mid ( E ) \mid - E \mid \mathbf{id}$
- Derive
  - - ( **id** + **id** ) from the grammar
  - $E \Rightarrow - E \Rightarrow - ( E ) \Rightarrow - ( E + E ) \Rightarrow - ( \mathbf{id} + E ) \Rightarrow - ( \mathbf{id} + \mathbf{id} )$
  - thus E derives - ( **id** + **id** )

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

# Derivation (cont.)

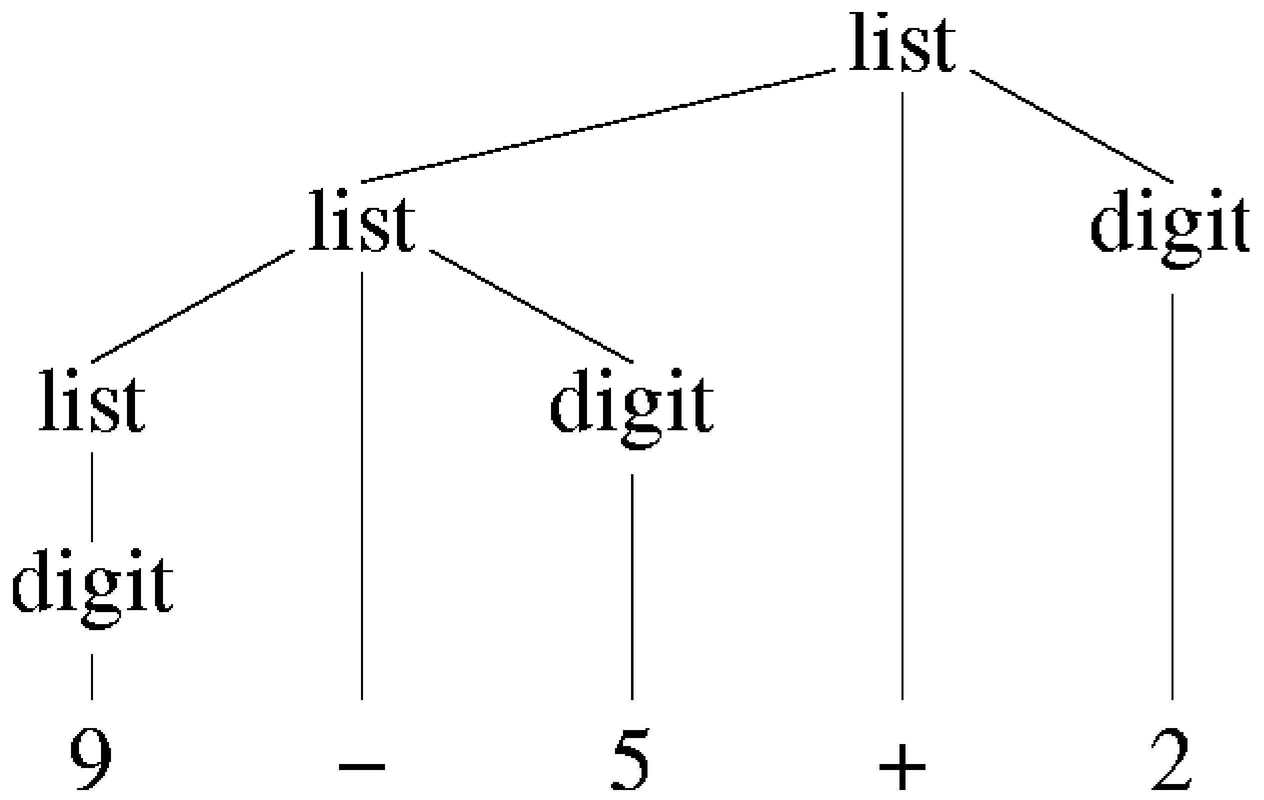
- Leftmost derivation
  - each step replaces the leftmost nonterminal
  - derive id + id * id using leftmost derivation
    - $E \Rightarrow E + E \Rightarrow id + E \Rightarrow id + E * E \Rightarrow id + id * E \Rightarrow id + id * id$
- L(G) - language generated by the grammar G
- Sentence of G
  - if $S \overset{+}{\Rightarrow} w$, where w is a string of terminals inL(G)
- Sentential form
  - if $S \overset{*}{\Rightarrow} \alpha$, where $\alpha$ may contain nonterminals

# Parse Tree

- 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:
  - The root is labeled by the start symbol.
  - Each leaf is labeled by a token or ε.
  - Each interior node is labeled by a nonterminal.
  - 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

$A \rightarrow X_1, X_2, X_3, .. X_n$ is a production of the grammar.

# Example of a Parse Tree




list → list + digit | list – digit | digit

*followed by Fig. 4.4*

# Parse Tree (cont.)

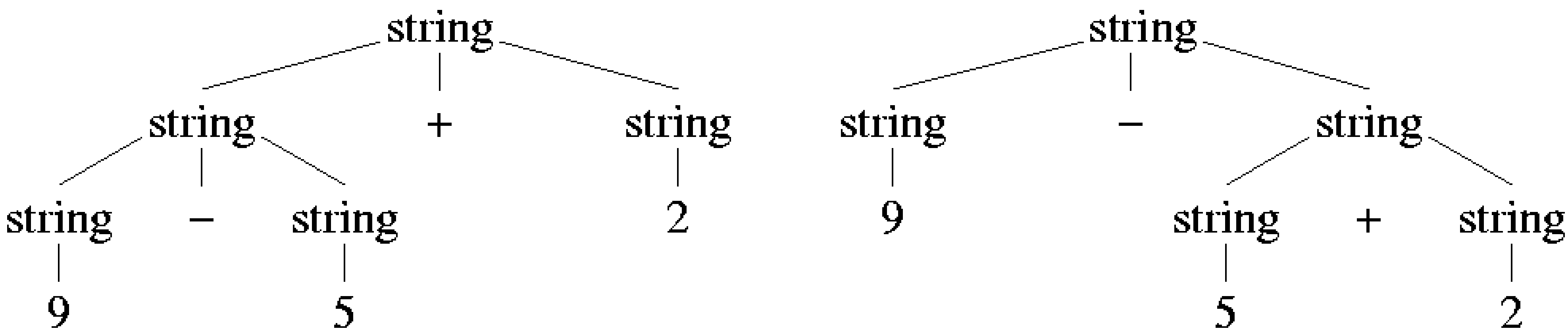
- Yield
  - the leaves of the parse tree read from left to right, or
  - the string derived from the nonterminal at the root of the parse tree
- An ambiguous grammar is one that can generate two or more parse trees that yield the same string.

# 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

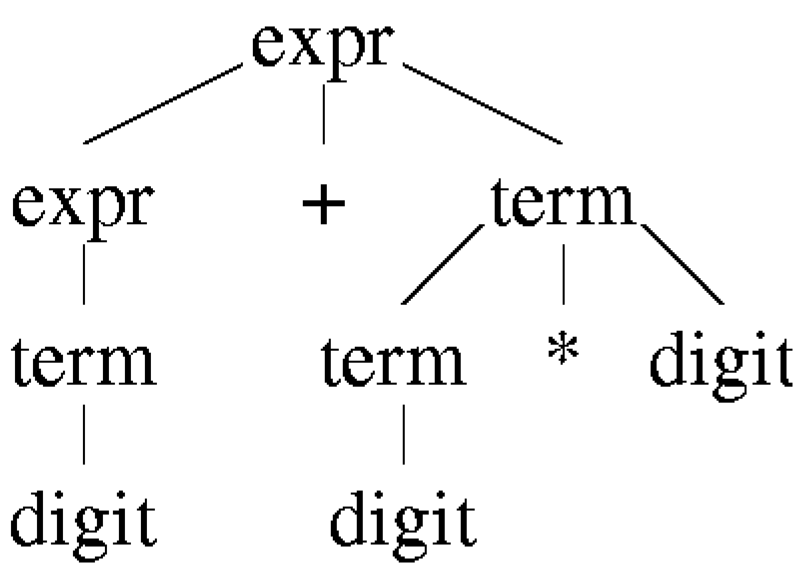
b. string ⟶ string - string ⟶ 9 – string
⟶ 9 – string + string ⟶ 9 – 5 + string ⟶ 9 – 5 + 2

# 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.)

- If different operators have the same precedence then they are defined as alternative productions of the same nonterminal.

  expr → expr + term | expr – term | term
  term → term * factor | term / factor | factor
  factor → digit | (expr)

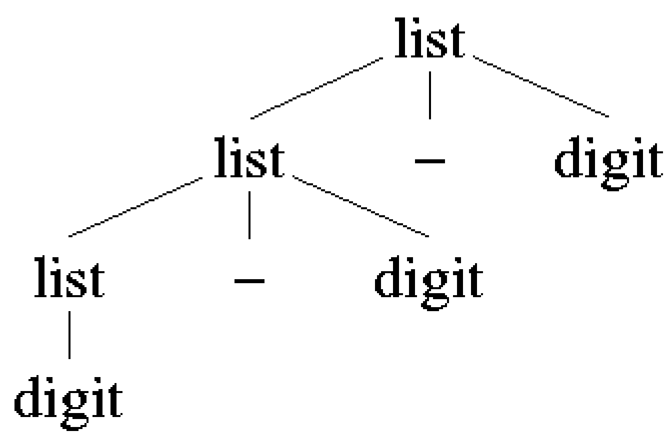
# Associativity

By convention

9 – 5 – 2 left (operand with – on both sides is taken by the operator to its left)

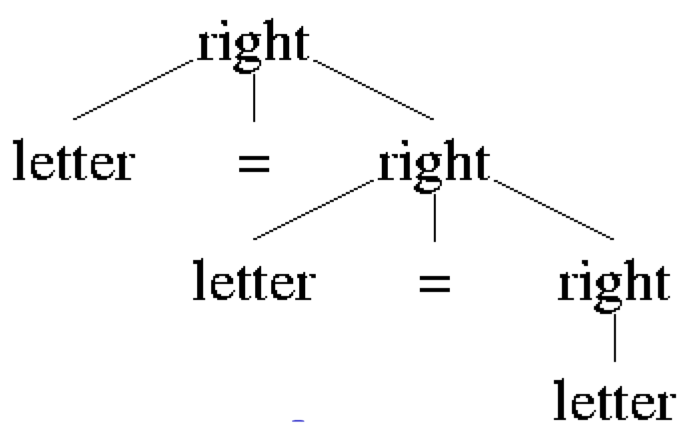
a = b = c right

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 → **if** expr **then** stmt

  | **if** expr **then** stmt **else** stmt

  | other
- How do we parse:

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

*followed by Fig. 4.9*

# Eliminating Ambiguity (cont.)

- stmt $\longrightarrow$ matched_stmt
  | unmatched_stmt
- matched_stmt $\longrightarrow$ **if** expr **then** matched_stmt **else** matched_stmt
  | other
- unmatched_stmt $\longrightarrow$ **if** expr **then** stmt
  | **if** expr **then** matched_stmt **else** unmatched_stmt

# Parsing

- Universal
- Top-down
  - recursive descent
  - LL
- Bottom-up
  - 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

- type → simple | ↑id | array [ simple ] of type
- simple → integer | char | num dotdot num

*followed by Fig. 4.12*

# 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 , 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 $A \longrightarrow A\alpha \mid \beta$ with $A \longrightarrow \beta A'$

  $A' \longrightarrow \alpha A' \mid \varepsilon$

Example:

| | A | α | β |
|---|---|---|---|
| $E \longrightarrow E + T \mid T$ | E | +T | T |
| $T \longrightarrow T * F \mid F$ | T | *F | F |
| $F \longrightarrow (E) \mid id$ | | | |

becomes

$E \longrightarrow TE'$

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

$T \longrightarrow FT'$

# Eliminating Left Recursion (cont.)

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

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

  }

  eliminate immediate left recursion in $A_i$ productions

  eliminate ε transitions in the $A_i$ productions

}

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 | a
2. $Y \rightarrow$ ZX | Xb
3. $Z \rightarrow$ XY | ZZ | 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

$Y \rightarrow ZXY' \mid ab\,Y'$

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

now eliminate □ 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.)

$i = 3, j = 2$

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

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

$$A \rightarrow \alpha\beta \mid \alpha\gamma \quad \Rightarrow \quad \begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow \beta \mid \gamma \end{array}$$

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

Useful for predictive parsing since we will know which production to choose.

# Nonrecursive Predictive Parsing

- Instead of recursive descent, it is table-driven and uses 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

*followed by Fig. 4.19*

# 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

*followed by Fig. 4.17, 4.21*

# 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 $Y_1, Y_2, \ldots, Y_{i-1} \overset{*}{\Rightarrow} \varepsilon$ where $i \leq k$

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

   If $Y_1, Y_2, \ldots, Y_k \overset{*}{\Rightarrow} \varepsilon$, 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 $\beta \overset{*}{\Rightarrow} \varepsilon$,
   add FOLLOW(A) to FOLLOW(B)

# Example of Calculating FIRST and FOLLOW

| Production | FIRST | FOLLOW |
| --- | --- | --- |
| E → TE´ | { (, id } | { ), $ } |
| E´ → +TE´ \| ε | { +, ε } | { ), $ } |
| T → FT´ | { (, id } | { +, ), $ } |
| T´ → *FT´ \| ε | {*, ε } | { +, ), $ } |
| F → (E) \| id | { (, id } | {*, +, ), $ } |

# Another Example of Calculating FIRST and FOLLOW

| Production | | FIRST | FOLLOW |
|---|---|---|---|
| X | $\rightarrow$ Ya | { } | { } |
| Y | $\rightarrow$ ZW | { } | { } |
| W | $\rightarrow$ c \| ε | { } | { } |
| Z | $\rightarrow$ a \| 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($\alpha$)
2. If $\varepsilon$ is in FIRST($\alpha$)
   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. FIRST($\alpha_i$) intersect FIRST($\alpha_j$) = $\emptyset$
   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. $\alpha_1, .., \alpha_{i-1}, \alpha_{i+1}, .., \alpha_n$ does not $\overset{*}{\Rightarrow} \varepsilon$
   b. FIRST($\alpha_j$) intersect FOLLOW(A) = $\emptyset$
      where $j \ne i$ and $1 \le j \le n$

# Checking If a Grammar is LL(1)

| Production | FIRST | FOLLOW |
|---|---|---|
| S → iEtSS′ \| a | { i, a } | { e, $ } |
| S′→ eS \| ε | { e, ε } | { e, $ } |
| E → b | { b } | { t } |

| Nonterminal | a | b | e | i | t | $ |
|---|---|---|---|---|---|---|
| S | S→a | | | S→iEtSS′ | | |
| S′ | | | S′→eS<br>S′→ε | | | S′→ε |
| E | | E→b | | | | |

So this grammar is not LL(1).

# Bottom-Up Parsing

- Bottom-up parsing
  - attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root
  - is the process of ***reducing*** the string *w* to the start symbol of the grammar
  - at each step, we need to decide
    - when to reduce
    - what production to apply
  - actually, constructs a right-most derivation in reverse

*followed by Fig. 4.25*

# Shift-Reduce Parsing

- Shift-reduce parsing is bottom-up.
- 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

*followed by Fig. 4.35*

# Model of an LR Parser

- 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.

# 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). Parsing table has an action part and a goto part.

# 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 ($s_0$ $X_1$ $s_1$ $X_2$ $s_2$ ... $X_m$ $s_m$ $a_i$s, $a_{i+1}$ ... $a_n$\$)
2. if action [$s_m$, $a_i$] == reduce A→β 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 →iCtS | 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. $F \rightarrow id$

*followed by Fig. 4.37*

# LR Parsing Example

- It produces rightmost derivation in reverse:

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

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

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

$$\rightarrow id * id + id$$

*followed by Fig. 4.38*

# Calculating the Sets of LR(0) Items

LR(0) item - production with a dot at some position in the right side

Example:

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

$A \to \cdot BC$

$A \to B \cdot C$

$A \to BC \cdot$

$A \to \varepsilon$ has 1 possible item

$A \to \cdot$

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

*followed by Fig. 4.32*

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

<u>Grammar</u>

E´ →E
E →E + T | T
T →T * F | F
F →( E ) | id

<u>Closure</u> ($I_0$) for $I_0$ = {E´→·E}

E´ →·E
E →·E + T
E →·T
T →·T * F
T →·F
F →·( E )
F →· 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 $A \rightarrow \alpha X \cdot \beta$)

where $A \rightarrow \alpha \cdot X \beta$ is in I.

<u>Grammar</u>

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow ( E ) \mid id$

<u>Goto</u> $(I_1,+)$ for $I_1 = \{E' \rightarrow E\cdot, E \rightarrow E\cdot+T\}$

$E \rightarrow E + \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot( E )$

$F \rightarrow \cdot id$

<u>Goto</u> $(I_2,*)$ for $I_2 = \{E \rightarrow T\cdot, T \rightarrow T\cdot*F\}$

$T \rightarrow T * \cdot F$

$F \rightarrow \cdot( E )$

$F \rightarrow \cdot id$

# 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´→S.

*followed by Fig. 4.33, 4.31*

# 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 the same as 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

*followed by Fig. 4.31, A*

# Sets of LR(0) Items Example

$S \rightarrow L = R \mid R$

$L \rightarrow {*R} \mid id$

$R \rightarrow L$

*followed by Fig. 4.39*

# 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 $[S' \rightarrow \cdot S]$.

*followed by Fig. 4.37*

# Using Ambiguous Grammars

| | | |
|---|---|---|
| 1. $E \rightarrow E + E$<br>2. $E \rightarrow E * E$<br>3. $E \rightarrow ( E )$<br>4. $E \rightarrow id$ | instead of | $E \rightarrow E + T \mid T$<br>$T \rightarrow T * F \mid F$<br>$F \rightarrow ( E ) \mid id$ |

See Figure 4.48.

Advantages:

Grammar is easier to read.

Parser is more efficient.

*followed by Fig. 4.48*

# 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)

*followed by Fig. 4.49*

# 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)

*followed by Fig. 4.50, 4.51*

# Ambiguities from Special-Case Productions

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

# Ambiguities from Special-Case Productions (cont)

1. E → E sub E sup E
2. E → E sub E
3. E → E sup E
4. E → { E }
5. E → c

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

sub, sup have equal precedence and are right associative

*followed by Fig. B*

# Ambiguities from Special-Case Productions (cont)

1. E → E sub E sup E
2. E → E sub E
3. E → E sup E
4. E → { E }
5. E → 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]=(s5,r1,r3)
action[11,}]=(r1,r3)

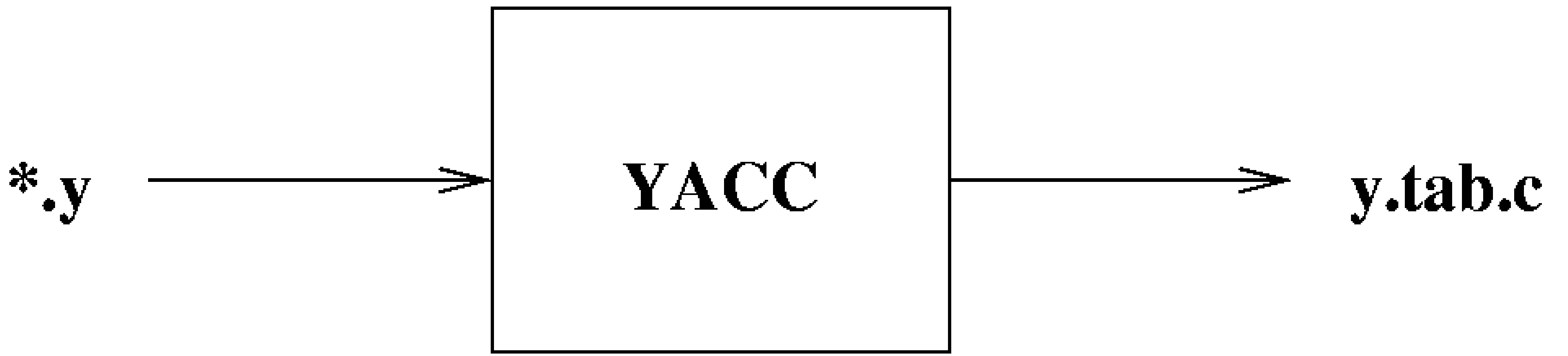
action[7,sup]=(s10,r2)
action[8,sup]=(s5,r3)
action[11,sup]=(s5,r1,r3)
action[11,$]=(r1,r3)

*followed by Fig. C*

# YACC

Yacc source program

declaration
%%
translation rules
%%
supporting C-routines




*followed by Fig. 4.57*

# 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; }

*followed by Fig. 4.58, 4.59*

# 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 processing of correct programs

# Syntax Error Handling (cont.)

- Viable−prefix property - detect an error as soon as see 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 error (when more than one error in a single statement)
- Phase- 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-Recovery Strategies (cont...)

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