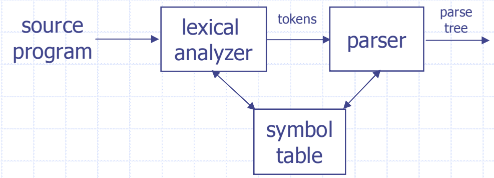
編譯器設計

Chapter 4:
Syntax Analysis

Syntax Analysis

- The role of the syntax analyzer
  - To accept a string of tokens from the scanner
  - To verify the string can be generated by the grammar



- Three general types of parsers for grammars
  - Universal parsing methods
    - e.g. Cocke-Younger-Kasami, Earley's
  - Top-down
  - Bottom-up

Syntax Error Handling

- The goals of the error handler in a parser
  - It should report the presence of errors clearly and accurately
  - It should recover from each error quickly enough to be able to detect subsequent errors
  - If should not significantly slow down the processing of correct programs
- Many errors could be classified [Ripley et al 1978]:
  - 60% punctuation errors
  - 20% operator and operand errors
  - 15% keyword errors
  - 5% others

Error Recovery Strategies

- Panic mode
  - The simplest method and can be used by most parsers
  - On discovering an error, the parser discards input symbols one at a time until one of synchronizing tokens is found
    - The synchronizing tokens are usually delimiters, e.g. ; or end
- Phrase level
  - On discovering an error, a parser may perform local correction on the remaining input
    - e.g. replacing a comma by a semicolon, inserting a missing ;
- Error production
  - Augment the grammar with productions that generate the erroneous constructs
- Global correction
  - Make as few changes as possible in processing the input

Context-Free Grammars

- The syntax of programming language constructs can be described by *context-free grammars*
  - Or BNF (Backus-Naur Form)
- Grammars offer significant advantages
  - A grammar gives a precise syntactic specification of a programming language
  - From certain classes of grammars we can automatically construct an efficient parser
    - The parser construction process can reveal syntactic ambiguities and difficult-to-parse constructs
  - A properly designed grammar facilitates the translation into target code
  - New language constructs can be added easily

Context-Free Grammars

- A context-free grammar G can be denoted by $G = (V_N, V_T, P, S)$
  - V_N: nonterminals
    - Syntactic variables that denotes sets of strings
  - V_T: terminals
    - The basic symbols (i.e. tokens) from which strings are formed
  - P : productions
    - The manner in which the terminals and nonterminals can be combined for form strings, e.g.

      expr → *expr op expr*

      expr → id

      op → + | − | ∗ | /
  - S : start symbol

Why Not Using Regular Languages

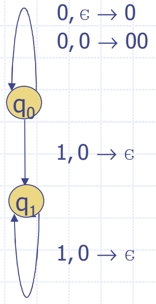
- Regular languages are not power enough to represent certain constructs in program languages, e.g. $\{^n\}^n$
  - i.e. $\{^n\}^n$ is not a regular language
- Recall the ways to prove a language is regular
  - Find a regular grammar (or regular expressions) that generates the language
  - Find an NFA that recognizes the language
- However, they can not prove $\{^n\}^n$ is not regular
  - The pumping lemma of regular languages will be used
  - Prove by contradiction

Pumping Lemma

- If A is a regular language and string $s \in A$, where $\exists n \in N$ such that $|s| \geq n$, then $s = xyz$ where
  - $xy^iz \in A$, $i \geq 0$,
  - $|y| > 0$, and
  - $|xy| \leq n$
- Example: prove $L = \{0^n1^n \mid n \geq 0\}$ is not regular
  - Assume L is regular. Let $s = wyz$, $s \in L$
  - If y consists of 0's only, then xy^iz has more 0's than 1's
  - If y consists of 1's only, then xy^iz has more 1's than 0's
  - If y consists of 0's and 1's, then $xy^iz \neq 0^n1^n$

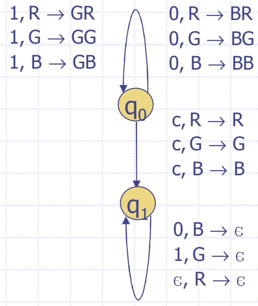
Pushdown Automata

- $M = (K, \Sigma, \Gamma, \delta, q_0, Z_0, F)$
  - K: Finite set of states
  - Σ: Input alphabet
  - Γ: Pushdown alphabet
  - δ: Mapping $K \times (\Sigma \cup \epsilon) \times \Gamma \to K \times \Gamma^*$
  - q_0 : Initial state ($q_0 \in K$)
  - Z_0 : Initial pushdown symbol ($Z_0 \in \Gamma$)
  - F: Set of final states
- Example: $L = \{0^n1^n \mid n \geq 0\}$
  - $M = (\{q_0, q_1\}, \{0, 1\}, \{0, 1\}, \delta, q_0, \epsilon, \phi)$
    - $\delta(q_0, 0, \epsilon) = (q_0, 0)$
    - $\delta(q_0, 0, 0) = (q_0, 00)$
    - $\delta(q_0, 1, 0) = (q_1, \epsilon)$
    - $\delta(q_1, 1, 0) = (q_1, \epsilon)$



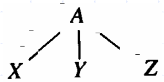
Pushdown Automata

- Example: $L = \{wcw^R \mid w \in \{0, 1\}^*\}$
  - $M = (\{q_0, q_1\}, \{0, 1, c\}, \{R, G, B\}, \delta, q_0, R, \phi)$
    - $\delta(q_0, 0, R) = (q_0, BR)$
    - $\delta(q_0, 0, G) = (q_0, BG)$
    - $\delta(q_0, 0, B) = (q_0, BB)$
    - $\delta(q_0, 1, R) = (q_0, GR)$
    - $\delta(q_0, 1, G) = (q_0, GG)$
    - $\delta(q_0, 1, B) = (q_0, GB)$
    - $\delta(q_0, c, R) = (q_1, R)$
    - $\delta(q_0, c, G) = (q_1, G)$
    - $\delta(q_0, c, B) = (q_1, B)$
    - $\delta(q_1, 0, B) = (q_1, \epsilon)$
    - $\delta(q_1, 1, G) = (q_1, \epsilon)$
    - $\delta(q_1, \epsilon, R) = (q_1, \epsilon)$



Parse Trees

- A parse tree pictorially shows how the start symbol of a grammar derives a string in the language
  - e.g. $A \to XYZ$

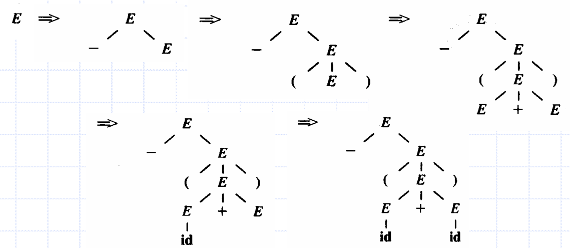


- Formally, given a context-free grammar, a parse tree is a tree with the following properties
  - The root is labeled by the start symbol
  - Each leaf is labeled by a token or by ϵ
  - Each interior node is labeled by a nonterminal
  - If A is the nonterminal labeling some interior node and $X_1, X_2, \ldots, X_n$ are the labels of the children of that node from left to right, then $A \to X_1 X_2 \ldots X_n$ is a production

Parse Trees and Derivations

- There are several ways to view the process by which a grammar defines a language
  - Building parse trees
  - Derivations
- Derivations give a precise description of the top-down construction of a parse tree
  - Example: $E \to E + E \mid E * E \mid (E) \mid -E \mid \text{id}$
  - $w = -(\text{id} + \text{id})$
  - Derivation:

$$
\begin{aligned}
E &\Rightarrow -E \\
&\Rightarrow -(E) \\
&\Rightarrow -(E + E) \\
&\Rightarrow -(\text{id} + E) \\
&\Rightarrow -(\text{id} + \text{id})
\end{aligned}
$$

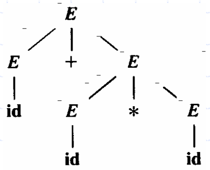


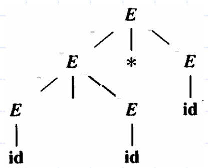
Parse Trees and Derivations

- Every parse tree has associated with it a unique leftmost and a unique rightmost derivation
  - However, not every sentence has exactly one leftmost or rightmost derivation
  - e.g. id + id * id

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow \text{id} + E \\ &\Rightarrow \text{id} + E * E \\ &\Rightarrow \text{id} + \text{id} * E \\ &\Rightarrow \text{id} + \text{id} * \text{id} \end{aligned}$$

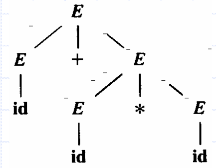
$$\begin{aligned} E &\Rightarrow E * E \\ &\Rightarrow E + E * E \\ &\Rightarrow \text{id} + E * E \\ &\Rightarrow \text{id} + \text{id} * E \\ &\Rightarrow \text{id} + \text{id} * \text{id} \end{aligned}$$





Ambiguity

- A grammar that produces more than one parse tree for some sentence is said to be *ambiguous*
  - i.e. more than one leftmost or rightmost derivation for the same sentence
- Two options to deal with ambiguity
  - *Disambiguating rules* are used to throw away undesirable parse trees
  - To eliminate ambiguity
    - Precedence and associativity
    - Rewriting the grammar



Eliminating Ambiguity

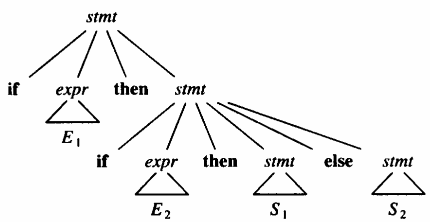
- Example (Dangling Else):

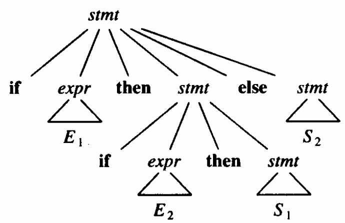
  stmt → if *expr* then *stmt*

  stmt → if *expr* then *stmt* else *stmt*

  stmt → other

  - if E_1 then if E_2 then S_1 else S_2





Eliminating Ambiguity

- Rewrite the grammar:

  stmt → matched_stmt

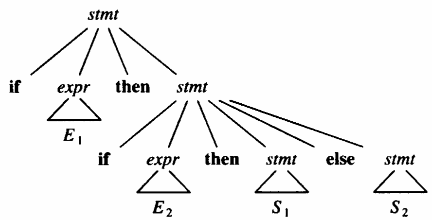
  | unmatched_stmt

  matched_stmt → if *expr* then *matched_stmt* else *matched_stmt*

  | other

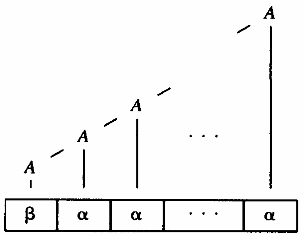
  unmatched_stmt → if *expr* then *stmt*

  | if *expr* then *matched_stmt* else *unmatched_stmt*



Left Recursion

- A grammar is *left recursive* if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$
  - e.g. $A \rightarrow A\alpha \mid \beta$
- It is possible that a parser to loop forever if the grammar is left recursive



Eliminating Left Recursion

- A left-recursive pair of production $A \rightarrow A\alpha \mid \beta$ can be changed to non-left-recursive productions

  $A \rightarrow \beta A'$

  $A' \rightarrow \alpha A' \mid \epsilon$

  without changing the set of strings derivable from A
- Example:

$E \rightarrow E + T \mid T$	$\Rightarrow$	$E \rightarrow TE'$
$T \rightarrow T * F \mid F$		$E' \rightarrow +TE' \mid \epsilon$
$F \rightarrow (E) \mid \mathrm{id}$		$T \rightarrow FT'$
		$T' \rightarrow *FT' \mid \epsilon$
		$F \rightarrow (E) \mid \mathrm{id}$

Eliminating Left Recursion

- Algorithm
  - Input. Grammar G with no cycles on ϵ-productions
  - Output. An equivalent grammar with no left recursion
  - Method.

    Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$
    for i = 1 to n do
      for j = 1 to i-1 do
        replace each production of the form $A_i \rightarrow A_j\gamma$ by
          the production $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,
          where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current A_j-productions
        eliminate the immediate left recursion among the A_i-productions
      end
    end

Eliminating Left Recursion

- Example

  $S \rightarrow Aa \mid b$

  $A \rightarrow Ac \mid Sd \mid \epsilon$

  - Arrange the nonterminals in order S, A
  - i = 1
    - No immediate left recursion among the S-productions
  - i = 2
    - Substitute the S-productions in $A \rightarrow Sd$ to obtain

      $A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$
    - Eliminate the left recursion and yield

      $S \rightarrow Aa \mid b$

      $A \rightarrow bdA' \mid A'$

      $A \rightarrow cA' \mid adA' \mid \epsilon$

Left Factoring

- Left factoring is a grammar transformation
  - Usually applied when it is not clear which of two alternative productions to use to expand a nonterminal *A*
  - Rewrite the *A*-productions to defer the decision
  - e.g. *stmt* $\rightarrow$ if *expr* then *stmt* else *stmt*

    stmt $\rightarrow$ if *expr* then *stmt*
- Algorithm
  - For each nonterminal *A* find the longest prefix α common to two or more of its alternatives
  - Replace $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$ by

    $A \rightarrow \alpha A' \mid \gamma$

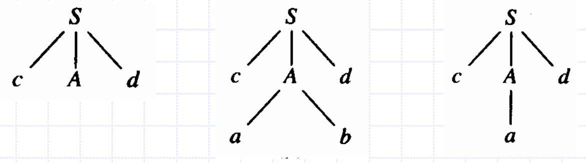
    $A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

Top-Down Parsing

- Top-down parsing an be viewed as an attempt to find a leftmost derivation for an input string
  - Equivalently, it can be viewed as an attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder
- Example

  $S \rightarrow cAd$

  $A \rightarrow ab \mid a$



Top-Down Parsing

- Recursive-Descent Parsing
  - Start with the root, labeled with the starting symbol, and repeatedly perform the following steps
    - At node *n* labeled with nonterminal *A*, select one of the productions for *A* and construct children at *n* for the symbols on the right side of the production
    - When a terminal is added that does not match the input string, then backtrack
    - find the next node
- Predictive Parsing
  - Recursive-descent parsing without backtracking

Designing a Predictive Parser

- A *predictive parser* is a program consisting a procedure for every nonterminal. Each procedure does two things
  - For all *A*-productions it decides which production to use by looking at the lookahead symbol, say a, i.e.
    - The production $A \rightarrow \alpha$ is used if

      $a \in \text{FIRST}(\alpha)$

      where FIRST(α) is the set of terminals that begins the strings derived from α, or
    - The production $A \rightarrow \epsilon$ is used if

      $a \notin \text{FIRST}(\alpha)$
  - The procedure uses a production by mimicking the right side:
    - A nonterminal results a call to the procedure for the nonterminal
    - A token matching the lookahead symbol results in the next input toke being read

Designing a Predictive Parser

- Example

  type → *simple* | ^id | array [*simple*] of type

  simple → integer | char | num dotdot num

```
procedure match (t : token);
begin
  if lookahead = t then
    lookahead : = nextoken
  else error
end;
```

```
procedure type ;
begin
  if lookahead ∈ {integer, char, num} then
    simple
  else if lookahead = '^' then begin
    match('^'); match (id)
  end
  else if lookahead = array then begin
    match (array); match('['); simple;
    match(']'); match (of); type
  end
  else error
end;
```

```
procedure simple ;
begin
  if lookahead = integer then
    match (integer)
  else if lookahead = char then
    match (char)
  else if lookahead = num then begin
    match(num); match(dotdot); match(num)
  end
  else error
end;
```


Transition Diagrams for Predictive Parsers

- Features
  - There is one diagram (procedure) for each nonterminal
  - The labels of edges are tokens (terminals) and nonterminals
  - A transition on a token is taken if that token is the next input symbol
  - A transition on a nonterminal *A* is a call of the procedure of *A*
- To construct the transition diagram from a grammar
  - Eliminate left recursion from the grammar
  - Left factor the grammar
  - For each nonterminal *A*, do the following
    - Create an initial state and a final (return) state
    - For each production $A \rightarrow X_1X_2 \ldots X_n$, create a path from the initial to the final state, with edges labeled $X_1, X_2, \ldots, X_n$

Transition Diagrams for Predictive Parsers

- Example

  $E \rightarrow TE'$

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

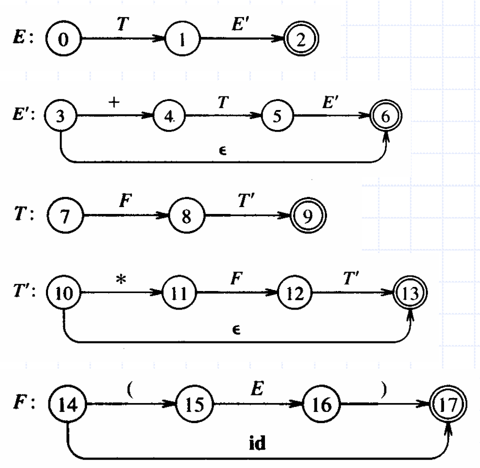
  $T \rightarrow FT'$

  $T' \rightarrow *FT' \mid \epsilon$

  $F \rightarrow (E) \mid \text{id}$

  - w = id + id * id

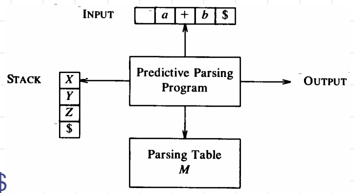
```
E( ) {
  T( ); E'( );
}

E'( ) {
  if (lookahead == '+') {
    T( ); E'( );
  }
}
```



Nonrecursive Predictive Parser

- The key problem during predictive parsing is that of determining the production to be applied for a nonterminal
- A table-drive predictive parser has an input buffer, a stack, a parsing table, and an output stream
- The action of the parser is determined by *X*, the symbol on top of the stack, and a, the current input symbol
  - If *X* = a = '$', the parser halts and announces successful completion of parsing
  - If *X* = a ≠ '$', the parser pops *X* off the stack and advances the input pointer to the next input symbol
  - If *X* is a nonterminal, the program consults entry M[X, a] of the parsing table M.
    - If, for example, M[X, a] = {$X \rightarrow UVW$}, the parser replaces *X* on top of the stack by *WVU* (with *U* on top).
    - If M[X, a] = error, the parser calls an error recovery routine.

Nonrecursive Predictive Parser



- Algorithm
  - Input. A string $w\$$ and a parsing table M
  - Output. A leftmost derivation of w
  - Method. Initially, stack = $S

```
set ip to point to the first symbol of w$
repeat
  let X be the top stack symbol and a the symbol pointed by ip
  if X is a termonal or $ then
    if X = a then
      pop X off the stack and advance ip
  else        /* X is a nonterminal */
    if M[X, a] = X → Y1 Y2 … Yk then
      pop X from the stack
      push Yk, Yk-1, …, Y1 onto the stack, with Y1 on the top
      Output the production X → Y1 Y2 … Yk
    else error( )
until X = $       /* stack is empty */
```


Nonrecursive Predictive Parser

- Example
  - $E \rightarrow TE'$
  - $E' \rightarrow +TE' \mid \epsilon$
  - $T \rightarrow FT'$
  - $T' \rightarrow *FT' \mid \epsilon$
  - $F \rightarrow (E) \mid \text{id}$
- Parsing Table
  - Fig. 4.15

NONTERMINAL	INPUT SYMBOL id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

Stack	Input	Output	Leftmost Derivation
$\$E$	id + id * id$		$E \Rightarrow$
$\$E'T$	id + id * id$	$E \rightarrow TE'$	$TE' \Rightarrow$
$\$E'T'F$	id + id * id$	$T \rightarrow FT'$	$FT'E' \Rightarrow$
$\$E'T'\text{id}$	id + id * id$	$F \rightarrow \text{id}$	$\text{id}\,T'E' \Rightarrow$
$\$E'T'$	+ id * id$		$\text{id}\,T'E' \Rightarrow$
$\$E'$	+ id * id$	$T' \rightarrow \epsilon$	$\text{id}\,E' \Rightarrow$
$\$E'T+$	+ id * id$	$E' \rightarrow +TE'$	$\text{id} + T\,E' \Rightarrow$
$\$E'T$	id * id$		
$\$E'T'F$	id * id$	$T \rightarrow FT'$	$\text{id} + FT'E' \Rightarrow$
$\$E'T'\text{id}$	id * id$	$F \rightarrow \text{id}$	$\text{id} + \text{id}\,T'E' \Rightarrow$
$\$E'T'$	* id$		
$\$E'T'F*$	* id$	$T' \rightarrow *FT'$	$\text{id} + \text{id} * FT'E' \Rightarrow$
$\$E'T'F$	id$		
$\$E'T'\text{id}$	id$	$F \rightarrow \text{id}$	$\text{id} + \text{id} * \text{id}\ T'E' \Rightarrow$
$\$E'T'$	$		
$\$E'$	$	$T' \rightarrow \epsilon$	$\text{id} + \text{id} * \text{id}\ E' \Rightarrow$
$\$$	$	$E' \rightarrow \epsilon$	$\text{id} + \text{id} * \text{id}$

FIRST and FOLLOW

- The construction of a predictive parser is aided by two functions associated with a grammar G
- FIRST(α)
  - The set of terminals that begins the strings derived from α
  - To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added
    - If X is a terminal, then FIRST(X) = $\{X\}$
    - If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X)
    - If $X \rightarrow Y_1\ Y_2\ \ldots\ Y_k$ is a production, then
      - Place a in FIRST(X) if
        $\exists i\ (1 \le i \le k) \ni a \in \text{FIRST}(Y_i)$ and $\epsilon \in \text{FIRST}(Y_j)\ \forall j\ (1 \le i \le j)$
      - Place ε in FIRST(X) if
        $\forall i\ (1 \le i \le k)\ \epsilon \in \text{FIRST}(Y_i)$

FIRST and FOLLOW

- FOLLOW(A)
  - The set of terminals a that can appear immediately to the right of A in some sentential form, i.e.
    a $\in$ FOLLOW(A) if there exists a derivation $S \Rightarrow \alpha A a \beta$
  - To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added
    - Place $ in FOLLOW(S)
    - If there is production $A \rightarrow \alpha B \beta$, then add FIRST($\beta$) – $\{\epsilon\}$ to FOLLOW(B)
    - If there is production $A \rightarrow \alpha B$, then add FOLLOW(A) to FOLLOW(B)
    - If there is production $A \rightarrow \alpha B \beta$ and $\epsilon \in$ FIRST(β), then add FOLLOW(A) to FOLLOW(B)

FIRST and FOLLOW

- Example
  - $E \to TE'$
  - $E' \to +TE' \mid \epsilon$
  - $T \to FT'$
  - $T' \to *FT' \mid \epsilon$
  - $F \to (E) \mid \text{id}$
- FIRST
  - FIRST(E) = FIRST(T) = FIRST(F) = {(, id}
  - FIRST(E') = {+, ε}
  - FIRST(T') = {*, ε}
- FOLLOW
  - FOLLOW(E) = FOLLOW(E') = {), $}
  - FOLLOW(T) = FOLLOW(T') = {+,), $}
  - FOLLOW(F) = = {+, *,), $}

Construction of Predictive Parsing Table

- Algorithm
  - Input. Grammar *G*
  - Output. Parsing table M
  - Method.
    1. For each production $A \to \alpha$, do steps 2 to 4
    2. For each terminal a ∈ FIRST(α), then add $A \to \alpha$ to M[*A*, a]
    3. If ε ∈ FIRST(α), then add $A \to \alpha$ to M[*A*, b] for each terminal b ∈ FOLLOW(*A*)
    4. If ε ∈ FIRST(α) and $ ∈ FOLLOW(*A*), then add $A \to \alpha$ to M[*A*, $]
    5. Make each undefined entry of M be *error*

Construction of Predictive Parsing Table

- Example
  - $E \to TE'$
  - $E' \to +TE' \mid \epsilon$
  - $T \to FT'$
  - $T' \to *FT' \mid \epsilon$
  - $F \to (E) \mid \text{id}$
- FIRST(*E*) = {(, id}
- FIRST(*T*) = {(, id}
- FIRST(*F*) = {(, id}
- FIRST(*E'*) = {+, ε}
- FIRST(*T'*) = {*, ε}
- FOLLOW(*E*) ={), $}
- FOLLOW(*E'*) = {), $}
- FOLLOW(*T*) = {+,), $}
- FOLLOW(*T'*) = {+,), $}
- FOLLOW(*F*) = = {+, *,), $}

Nonter-minal	Input Symbol					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \text{id}$			$F \to (E)$		

Construction of Predictive Parsing Table

- Example
  - $S \to iEtSS' \mid a$
  - $S' \to eS \mid \epsilon$
  - $E \to b$
- FIRST(S) = {i, a}
- FIRST(S') = {e, ε}
- FIRST(E) = {b}
- FOLLOW(S) ={e, $}
- FOLLOW(S') = {e, $}
- FOLLOW(E) = {t}

Nonter-minal	Input Symbol					
	a	b	e	i	t	$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to eS$ $S' \to \epsilon$			$S' \to \epsilon$
E		$E \to b$				

LL(1) Grammars

- A grammar whose predictive parsing table has no multiply-defined entries is said to be *LL*(1)
  - 1st *L*: scanning the input from left to right
  - 2nd *L*: producing a leftmost derivation
  - 1: using one input symbol for lookahead at each step
- Properties
  - Not ambiguous
  - Not left recursive
  - Where $A \rightarrow \alpha \mid \beta$ are two distinct productions in *G*, the following conditions hold:
    - For no terminal a do both α and β derive strings beginning with a
    - At most one of α and β can derive the empty string
    - If β derives the empty string then α does not derive any string beginning with a terminal in FOLLOW(*A*)

Error Handling in Predicting Parsing

- An error is detected when
  - The terminal on top of the stack does not match the next input symbol, or
  - Nonterminal *A* is on top of the stack, a is the next input symbol, and M[*A*, a] is empty
- Panic-mode error recovery
  - Based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears
  - Some heuristics of choosing the synchronizing tokens
    - Place all symbols in FOLLOW(*A*) into the synchronizing set of *A*
    - Add symbols in FIRST(*A*) to the synchronizing set of *A*
    - If a nonterminal can generate ϵ, then the production deriving ϵ can be used as a default
    - If a terminal on top of the stack cannot be matched, then pop the terminal

Error Handling in Predicting Parsing

- Example
  - $E \rightarrow TE'$
  - $E' \rightarrow +TE' \mid \epsilon$
  - $T \rightarrow FT'$
  - $T' \rightarrow *FT' \mid \epsilon$
  - $F \rightarrow (E) \mid \text{id}$
- Parsing table
  - Fig. 4.18
- Actions
  - M[A, a] = ϕ
    input symbol is skipped
  - M[A, a] = sync
    nonterminal is popped
  - Token on stack isn't matched
    pop the token

Stack	Input	Output
$E	+ id * + id$	error, skip +
$E	id * + id$	id ∈ FIRST(E)
$E'T	id * + id$	
$E'T'F	id * + id$	
$E'T'id	id * + id$	
$E'T'	* + id$	
$E'T'F*	* + id$	
$E'T'F	+ id$	error, M[F,+] = sync
$E'T'	+ id$	F has been popped
$E'	+ id$	
$E'T+	+ id$	
$E'T	id$	
$E'T'F	id$	
$E'T'id	id$	
$E'T'	$	
$E'	$	
$	$	

Bottom-Up Parsing

- Attempt to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards to the start symbol
  - Example. Consider the grammar
    - $S \rightarrow aABe$
    - $A \rightarrow Abc \mid b$
    - $B \rightarrow d$

The sentence abbcde can be reduced to S:

Reduction	Rightmost Derivation
abbcde	S ⇒
aAbcde	aABe ⇒
aAde	aAde ⇒
aABe	aAbcde ⇒
S	abbcde

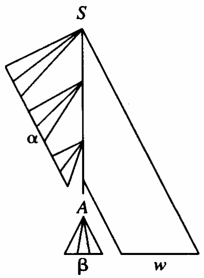
Bottom-Up Parsing

- A general style of bottom-up syntax analysis, known as *shift-reduce parsing,* will be introduced
  - The process of constructing a parse tree can be thought of as "reducing" a string w to the start symbol
  - It is equivalent to performing a rightmost derivation in reverse
  - At each reduction step, a particular substring matching the right side of a production is replaced by the symbol on the left
    - The matched substring can be called a *handle*
    - The reduction of a handle represents one step along the rightmost derivation in reverse
  - Formally, a handle of γ is
    - A production $A \rightarrow \beta$, and
    - A position of γ where the substring β may be found
    - e.g. $A \rightarrow b$ at position 2 is a handle of abbcde

      $A \rightarrow Abc$ at position 2 is a handle of aAbcde

Handle Pruning

- Suppose $A \rightarrow \beta$ is a handle
  - It represents a leftmost complete subtree consisting of a node and all its children
  - Reducing β to A in $\alpha\beta w$ can be thought of "pruning the handle"
    - i.e. removing the children of A
- A rightmost derivation in reverse can be obtained by handle pruning



Stack Implementation of Shift-Reduce Parsing

- Two problems for parsing by handle pruning
  - To locate the substring to be reduced
  - To determine what production to choose in case there is more than one production with that substring
- A convenient way to implement a shift-reduce parser is to use a stack to hold grammar symbols
  - Actions
    - Shift
      - the next input symbol is shifted onto the top of the stack
    - Reduce
      - The handle is at the top of the stack
      - Remove the handle and place the left side nonterminal on the stack
    - Accept
      - Announce successful completion of parsing
    - error

Stack Implementation of Shift-Reduce Parsing

- Example. (1) $E \rightarrow E + E$

  (2) $E \rightarrow E * E$

  (3) $E \rightarrow (E)$

  (4) $E \rightarrow \text{id}$

Stack	Input	Action
$	id + id * id$	shift
$id	+ id * id$	reduce by (4)
$*E*	+ id * id$	shift
$*E* +	id * id$	shift
$*E* + id	* id$	reduce by (4)
$*E* + *E*	* id$	reduce by (1)
$*E*	* id$	shift
$*E* *	id$	shift
$*E* * id	$	reduce by (4)
$*E* * *E*	$	reduce by (2)
$*E*	$	accept

Stack	Input	Action
$	id + id * id$	shift
$id	+ id * id$	reduce by (4)
$*E*	+ id * id$	shift
$*E* +	id * id$	shift
$*E* + id	* id$	reduce by (4)
$*E* + *E*	* id$	shift
$*E* + *E* *	id$	shift
$*E* + *E* * id	$	reduce by (4)
$*E* + *E* * *E*	$	reduce by (2)
$*E* + *E*	$	reduce by (1)
$*E*	$	accept

Conflicts During Shift-Reduce Parsing

- For some grammars, a shift-reduce parser can't decide
  - whether to shift or to reduce (a *shift/reduce conflict*), or
  - which of several reductions to make (a *reduce/reduce conflict*)
  - Example 1

Stack	Input	Action	Stack	Input	Action
$\$E + E$	* id\$	shift	$\$E + E$	* id\$	reduce by $E \to E + E$
$\$E + E$ *	id\$		$\$E$	* id\$	

  - Example 2

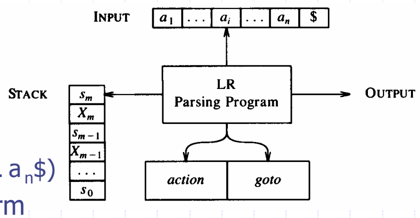
stmt → if *expr* then *stmt*
　　| if *expr* then *stmt* else *stmt*
　　| other

Stack	Input
… if *expr* then *stmt*	else … \$

LR Parsers

- LR(k) parsing
  - an efficient, bottom-up parsing technique
  - L: scanning the input from left to right
  - R: producing a rightmost derivation in reverse
  - k: the number of input symbols of lookahead
- Advantages
  - Can recognize virtually all programming-language constructs for which context-free grammars can be written
  - The most general nonbacktracking shift-reduce parsing
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers
  - Can detect a syntactic error as soon as possible
- Drawback
  - Too much work to construct an LR parser by hand

LR Parsing Algorithm



- Configuration
  - The stack contents
  - The unexpended input
  ($s_0\ X_1\ s_1\ X_2\ s_2 \ldots X_m\ s_m$, $a_i\ a_{i+1} \ldots a_n\$$)
  - It represents the sentential form
  $X_1 X_2 \ldots X_m a_i\ a_{i+1} \ldots a_n$
- Next move is determined by the input symbol a_i and stack top s_m
  - action[s_m, a_i] = shift s
  ($s_0\ X_1\ s_1\ X_2\ s_2 \ldots X_m\ s_m\ a_i\ s$, $a_{i+1} \ldots a_n\$$)
  - action[s_m, a_i] = reduce $A \to \beta$
  ($s_0\ X_1\ s_1\ X_2\ s_2 \ldots X_{m-r}\ s_{m-r}\ A\ s$, $a_i \ldots a_n\$$)
  where s = goto[s_{m-r}, A] and $|\beta| = r$
  - action[s_m, a_i] = accept
  - action[s_m, a_i] = error

LR Parsing Algorithm

- Input. String w and an LR parsing table
- Output. A bottom-up parsing for w
- Method. Initially, stack = s_0 and input = $w\$$

```
set ip to point to the first symbol of w$
repeat forever
  let s be the state on stack top and a the symbol pointed by ip
  if action[s, a] = shift s′ then
    push a and s′ onto the stack and advance the ip
  else if action[s, a] = reduce A → β then
    push 2*|β| symbols off the stack and now s′ is the top state
    push A and then goto[s′, A] on top of the stack
    output A → β
  else if action[s, a] = accept then
    return
  else error()
end
```

LR Parsing Algorithm

- Example.
  - (1) $E \to E + T$
  - (2) $E \to T$
  - (3) $T \to T * F$
  - (4) $T \to F$
  - (5) $F \to (E)$
  - (6) $F \to \text{id}$
- Parsing table
  - Fig. 4.31

STATE	action: id	+	*	(	)	$	goto: E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Stack	Input	Action
$0	id * id + id$	shift 5
$0 id 5	* id + id$	reduce by $F \to$ id
$0 F3	* id + id$	reduce by $T \to F$
$0 T2	* id + id$	shift
$0 T2 * 7	id + id$	shift
$0 T2 * 7 id 5	+ id$	reduce by $F \to$ id
$0 T2 * 7 F 10	+ id$	reduce by $T \to T * F$
$0 T2	+ id$	reduce by $E \to T$
$0 E1	+ id$	shift
$0 E1 + 6	id$	shift
$0 E1 + 6 id 5	$	reduce by $F \to$ id
$0 E1 + 6 F3	$	reduce by $T \to F$
$0 E1 + 6 T9	$	reduce by $E \to E + T$
$0 E1	$	accept

Constructing SLR Parsing Tables

- Three techniques for constructing an LR parsing table
  - Simple LR (SLR)
  - Canonical LR (LR)
  - Lookahead LR (LALR)
- LR(0) item
  - A production of G with a dot at some position of the right side,
  - e.g. production $A \to XYZ$ yields
    - $A \to \cdot XYZ$
    - $A \to X \cdot YZ$
    - $A \to XY \cdot Z$
    - $A \to XYZ \cdot$

    Production $A \to \epsilon$ yields
    - $A \to \cdot$
- Augmented grammar G'
  - G with a new start symbol S' and production $S' \to S$
  - Acceptance occurs when the parser is to reduce by $S' \to S$

Closure Operation

- If I is a set of items, then $closure(I)$ is the set of items constructed from I:
  - Initially, every item in I is added to $closure(I)$
  - If $A \to \alpha \cdot B \beta$ is in $closure(I)$ and $B \to \gamma$ is a production, then add the item $B \to \cdot \gamma$ to $closure(I)$
  - Repeat until no more new items can be added

```
function closure(I)
  J = I
  repeat
    for each item A → α·Bβ ∈ J
    and each B → γ
    such that B → ·γ ∉ J do
        add B → ·γ to J
  until no more items can be added to J
  return J
end
```

- Example

	$closure(\{E' \to E\}) = \{$
$E' \to E$	$E' \to \cdot E$
$E \to E + T$	$E \to \cdot E + T$
$E \to T$	$E \to \cdot T$
$T \to T * F$	$T \to \cdot T * F$
$T \to F$	$T \to \cdot F$
$F \to (E)$	$F \to \cdot (E)$
$F \to \text{id}$	$F \to \cdot \text{id}\ \}$

Goto Operation

- If I is a set of items and X is a grammar symbol, then $goto(I, X)$ is defined to be the closure of the set of all items $A \to \alpha X \cdot \beta$ such that $A \to \alpha \cdot X \beta$ is in I

  Example

$E' \to E$	$goto(\{E' \to E \cdot, E \to E \cdot + T\}, +) =$
$E \to E + T$	$closure(\{E \to E + \cdot T\}) = \{$
$E \to T$	$E \to E + \cdot T$
$T \to T * F$	$T \to \cdot T * F$
$T \to F$	$T \to \cdot F$
$F \to (E)$	$F \to \cdot (E)$
$F \to \text{id}$	$F \to \cdot \text{id}\ \}$

- Viable prefixes
  - The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser

Set-of-Items Construction

procedure *items* (G')
  $C = \{closure\{S' \to \cdot S\}\}$
  repeat
    for each set of items I in C and each grammar symbol X
    such that $goto(I, X)$ is not empty and not in C do
      add $goto(I, X)$ to C
  until no more set of items can be added to C
end

Example G =

$E' \to E$	I_0: $E' \to \cdot E$
$E \to E + T$	$E \to \cdot E + T$
$E \to T$	$E \to \cdot T$
$T \to T * F$	$T \to \cdot T * F$
$T \to F$	$T \to \cdot F$
$F \to (E)$	$F \to \cdot(E)$
$F \to$ id	$F \to \cdot$ id

Set-of-Items Construction

I_0: $E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot$ id

$goto(I_0, E)$ =
I_1: $E' \to E \cdot$
$E \to E \cdot + T$

$goto(I_0, T)$ =
I_2: $E \to T \cdot$
$T \to T \cdot * F$

$goto(I_0, F)$ =
I_3: $T \to F \cdot$

$goto(I_0, \text{"("})$ =
I_4: $F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot$ id

$goto(I_0, \text{id})$ =
I_5: $F \to \text{id} \cdot$

$goto(I_1, \text{"+"})$ =
I_6: $E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot$ id

$goto(I_2, \text{"*"})$ =
I_7: $T \to T * \cdot F$
$F \to \cdot(E)$
$F \to \cdot$ id

$goto(I_4, E)$ =
I_8: $F \to (E \cdot)$
$E \to E \cdot + T$

$goto(I_4, T) = I_2$

$goto(I_4, F) = I_3$

$goto(I_4, \text{"("}) = I_4$

$goto(I_4, \text{id}) = I_5$

$goto(I_6, T)$ =
I_9: $E \to E + T \cdot$
$T \to T \cdot * F$

$goto(I_6, F) = I_3$

$goto(I_6, \text{"("}) = I_4$

$goto(I_6, \text{id}) = I_5$

$goto(I_7, F)$ =
I_{10}: $T \to T * F \cdot$

$goto(I_7, \text{"("}) = I_4$

$goto(I_7, \text{id}) = I_5$

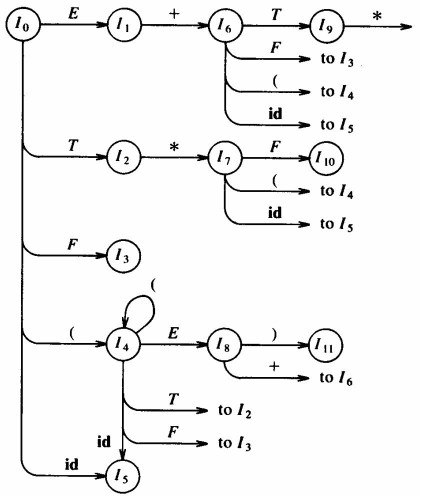
$goto(I_8, \text{")"})$ =
I_{11}: $F \to (E) \cdot$

$goto(I_8, \text{"+"}) = I_6$

$goto(I_9, \text{"*"}) = I_7$

Transition Diagram of DFA for Viable Prefixes

- The *goto* function for the sets of items can be represented as the transition diagram of a DFA *D*
  - Each set of items I_i is a state of D



Rightmost Derivation
$E \Rightarrow$
$E + T \Rightarrow$
$E + F \Rightarrow$
E + id $\Rightarrow$
T + id $\Rightarrow$
$T * F$ + id $\Rightarrow$
T * *id* + id $\Rightarrow$
F * *id* + id $\Rightarrow$
id + id * id

Transition Diagram of DFA for Viable Prefixes

- *D* recognizes exactly the viable prefixes of *G* if
  - Each state is a final state, and
  - I_0 is the initial state
- For every grammar *G*, the *goto* function of the collection of sets of items defines a DFA that recognizes the viable prefix of *G*
  - If each item is treated as a state, an NFA *N* can be formed:
    - There is a transition from $A \to \alpha \cdot X\beta$ to $A \to \alpha X \cdot \beta$ label X, and
    - There is a transition from $A \to \alpha \cdot B\beta$ to $B \to \cdot\gamma$ labeled ϵ
  - Then *closure* (I) for the set of items I is exactly the ϵ-*closure* of a set of NFA states
  - Thus goto(I, X) gives the transition from I on symbol X in the DFA constructed from *N* by the subset construction
  - The procedure *items* (G') is just the subset construction itself applied to the NFA *N*

Constructing an SLR Parsing Table

- **Algorithm**
  - Input. An augmented grammar G'
  - Output. The SLR parsing table for G'
  - Method.
    1. Construct $C = \{I_0, I_1, I_2, \ldots, I_n\}$, the sets of LR(0) items
    2. State i is constructed from I_i. Actions for state i
       - If $A \to \alpha \cdot a\beta \in I_i$ and goto(I_i, a) = I_j, then
         action[i, a] = shift j
       - If $A \to \alpha \cdot \in I_i$, then
         action[i, a] = reduce $A \to \alpha$ $\forall$a $\in$ FOLLOW(A)
       - If $S' \to S \cdot \in I_i$, then
         action[i, \$] = accept
    3. If goto(I_i, A) = I_j, then
       goto[i, A] = j
    4. The set of items containing $S' \to \cdot S$ is the initial state

Constructing an SLR Parsing Table

- Example
  (0) $E' \to E$
  (1) $E \to E + T$
  (2) $E \to T$
  (3) $T \to T * F$
  (4) $T \to F$
  (5) $F \to (E)$
  (6) $F \to$ id

- FOLLOW(E) ={+,), \$}
- FOLLOW(T) = {+, *,), \$}
- FOLLOW(F) = {+, *,), \$}

	action						*goto*		
	id	+	*	(	)	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Constructing an SLR Parsing Table

- Some grammars are not SLR, e.g.
  (0) $S' \to S$
  (1) $S \to L = R$
  (2) $S \to R$
  (3) $L \to *R$
  (4) $L \to$ id
  (5) $R \to L$

I_0: $S' \to \cdot S$
$S \to \cdot L = R$
$S \to \cdot R$
$L \to \cdot *R$
$L \to \cdot$ id
$R \to \cdot L$

goto(I_0, S) =
I_1: $S' \to S \cdot$

goto(I_0, L) =
I_2: $S \to L \cdot = R$
$R \to L \cdot$

goto(I_0, R) =
I_3: $S \to R \cdot$

goto(I_0, "*") =
I_4: $L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot$ id

goto(I_0, id) =
I_5: $L \to$ id $\cdot$

goto(I_2, "=") =
I_6: $S \to L = \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot$ id

goto(I_4, R) =
I_7: $L \to *R \cdot$

goto(I_4, L) =
I_8: $R \to L \cdot$

goto(I_6, R) =
I_9: $S \to L = R \cdot$

Constructing an SLR Parsing Table

- Some grammars are not SLR, e.g.
  (0) $S' \to S$
  (1) $S \to L = R$
  (2) $S \to R$
  (3) $L \to * R$
  (4) $L \to$ id
  (5) $R \to L$

- FOLLOW(S) ={\$}
- FOLLOW(L) = {=, \$}
- FOLLOW(R) = {=, \$}

I_2: $S \to L \cdot = R$
$R \to L \cdot$

	action				*goto*		
	id	=	*	\$	S	L	R
0							
1							
2		s6 r5					
3							
4							
5							
6							
7							
8							

Constructing Canonical LR Parsing Tables

- Why does SLR sometimes fail
  - $R \to L \cdot \in I_2 \Rightarrow$ action[2, '='] = reduce $R \to L$
    - ∵ = ∈ FOLLOW(R)
  - However, there is no sentential form begins R = …
  - Extra information will be incorporated by redefining items
- LR(1) items
  - The general form of an items becomes $[A \to \alpha \cdot \beta, a]$
    - The lookahead a has no effect when $\beta \neq \epsilon$
    - $[A \to \alpha \cdot, a]$ calls for a reduction by $A \to \alpha$ if the next symbol is a
  - Formally, LR(1) is valid for a viable prefix γ if there is a derivation $S \overset{*}{\Rightarrow} \delta A w \Rightarrow \delta\alpha\beta w$, where
    - $\gamma = \delta\alpha$, and
    - Either a is the first symbol of w, or w is ϵ and a is $

Sets of LR(1) Items

function *closure* (I)
  $J = I$
  repeat
    for each item $[A \to \alpha \cdot B \beta, a] \in J$
    each $B \to \gamma$ in G′ and
    each terminal b $\in$ FIRST(βa)
    such that $[B \to \cdot\, \gamma, b] \notin J$ do
      add $[B \to \cdot\, \gamma, b]$ to J
  until no more items can be added to J
  return J
end

procedure *items* (G')
  $C = \{closure(\{S' \to \cdot\, S\})\}$
  repeat
    for each set of item $I \in C$ and each grammar symbol X
    such that *goto* $(I, X) \notin C$ do
      add *goto* (I, X) to C
  until no more sets of items can be added to C
end

function *goto* (I, X)
  J = set of items $[A \to \alpha X \cdot \beta, a]$
    such that $[A \to \alpha \cdot X \beta, a] \in i$
  return *closure* (J)
end

Sets of LR(1) Items

Example G' =
$S' \to S$
$S \to CC$
$C \to cC$
$C \to d$

I_0: $S' \to \cdot S, \$$
$S \to \cdot\, CC, \$$
$C \to \cdot\, cC$, c/d
$C \to \cdot$ d, c/d

goto (I_0, S) =
I_1: $S' \to S \cdot, \$$

goto (I_0, C) =
I_2: $S \to C \cdot C, \$$
$C \to \cdot\, cC, \$$
$C \to \cdot$ d, $

goto $(I_0$, c) =
I_3: $C \to c \cdot C$, c/d
$C \to \cdot\, cC$, c/d
$C \to \cdot$ d, c/d

goto $(I_0$, d) =
I_4: $C \to$ d ·, c/d

goto (I_2, C) =
I_5: $S \to CC \cdot, \$$

goto $(I_2$, c) =
I_6: $C \to c \cdot C, \$$
$C \to \cdot\, cC, \$$
$C \to \cdot$ d, $

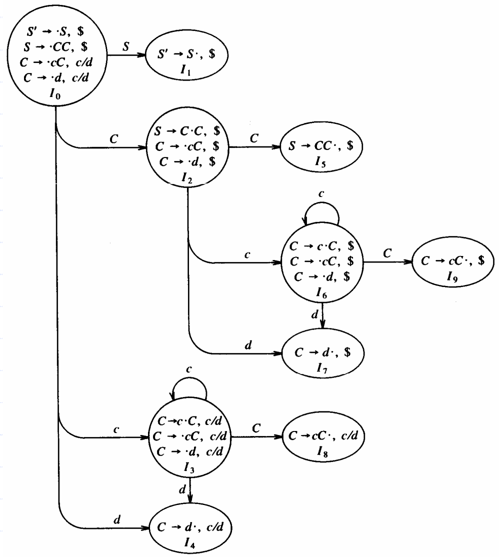
goto $(I_2$, d) =
I_7: $C \to$ d ·, $

goto (I_3, C) =
I_8: $C \to cC \cdot$, c/d
goto $(I_3$, c) = I_3
goto $(I_3$, d) = I_4

goto (I_6, C) =
I_9: $C \to cC \cdot$, $
goto $(I_6$, c) = I_6
goto $(I_6$, d) = I_7

Sets of LR(1) Items

The *goto* graph



Construction of the LR Parsing Table

- Algorithm
  - Input. An augmented grammar G'
  - Output. The canonical LR parsing table for G'
  - Method.
    1. Construct $C = \{I_0, I_1, I_2, \ldots, I_n\}$, the sets of LR(1) items
    2. State i is constructed from I_i. Actions for state i
       - If $[A \rightarrow \alpha \cdot a\beta, b] \in I_i$ and goto(I_i, a) = I_j, then action[i, a] = shift j
       - If $[A \rightarrow \alpha \cdot, a] \in I_i$ and , $A \neq S'$ then action[i, a] = reduce $A \rightarrow \alpha$
       - If $[S' \rightarrow S \cdot, \$] \in I_i$, then action[$i$, \$] = accept
    3. If goto(I_i , A) = I_j, then goto[i, A] = j
    4. The set of items containing $[S' \rightarrow \cdot S, \$]$ is the initial state

Constructing an SLR Parsing Table

Example G' =

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC$

$C \rightarrow d$

	action			goto	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Constructing LALR Parsing Tables

- Lookahead LR (LALR) technique
  - Often used in practice because the tables are much smaller than the LR tables
  - Yet most common syntactic constructs of programming languages can be expressed by an LALR grammar
  - Consider the following sets of LR(1) items

I_3: $C \rightarrow c \cdot C$, c/d
$C \rightarrow \cdot cC$, c/d
$C \rightarrow \cdot$ d, c/d

I_6: $C \rightarrow c \cdot C$, \$
$C \rightarrow \cdot cC$, \$
$C \rightarrow \cdot$ d, \$

I_4: $C \rightarrow$ d $\cdot$, c/d

I_7: $C \rightarrow$ d $\cdot$, \$

    - The only difference is the lookahead symbols
    - Note the G' generates the language c*dc*d
  - An LALR parsing table can be obtained by merging the sets of items of a LR table with the same *core*

Construction of the LALR Parsing Table

- Algorithm
  - Input. An augmented grammar G'
  - Output. The LALR parsing table for G'
  - Method.
    1. Construct $C = \{I_0, I_1, I_2, \ldots, I_n\}$, the sets of LR(1) items
    2. For each core in C, find all sets with that core and replace these sets by their union. Let result be $C' = \{J_0, J_1, J_2, \ldots, J_m\}$
    3. State i is constructed from J_i. Actions for state i
       - If $[A \rightarrow \alpha \cdot a\beta, b] \in J$ and goto(J_i, a) = J_j, then action[i, a] = shift j
       - If $[A \rightarrow \alpha \cdot, a] \in J_i$ and , $A \neq S'$ then action[i, a] = reduce $A \rightarrow \alpha$
       - If $[S' \rightarrow S \cdot, \$] \in J_i$, then action[$i$, \$] = accept
    4. If $J_i = I_1 \cup I_2 \cup \ldots \cup I_k$, then $goto(I_1,X)=goto(I_2,X)=\ldots=goto(I_k,X)=J_j$
       $\therefore$ goto[i, X] = j

Construction of the LALR Parsing Table

I_3: $C \to c \cdot C$, c/d
    $C \to \cdot cC$, c/d
    $C \to \cdot$ d, c/d

I_6: $C \to c \cdot C$, \$
    $C \to \cdot cC$, \$
    $C \to \cdot$ d, \$

I_4: $C \to$ d $\cdot$, c/d

I_7: $C \to$ d $\cdot$, \$

I_8: $C \to cC \cdot$, c/d

I_9: $C \to cC \cdot$, \$

	action			*goto*	
	c	d	$	*S*	*C*
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Error Detection by LALR

- LALR parser may proceed to do some reductions after the LR parser has declared an error
  - LALR parser will never shift another symbol after the LR parser declares an error
  - Example w = ccd\$

Stack	Input	Action
\$0	ccd\$	shift 36
\$0 c 36	cd\$	shift 36
\$0 c 36 c 36	d\$	shift 36
\$0 c 36 c 36 d 47	\$	reduce by $C \to$ d
\$0 c 36 c 36 C 89	\$	reduce by $C \to cC$
\$0 c 36 C 89	\$	reduce by $C \to cC$
\$0 C 2	\$	error

Stack	Input	Action
\$0	ccd\$	shift 3
\$0 c 3	cd\$	shift 3
\$0 c 3 c 3	d\$	shift 3
\$0 c 3 c 3 d 4	\$	error

Conflicts Caused by Merging States

- Merging states with common cores might cause conflicts
  - No shift-reduce conflicts will be introduced
    - Because shift actions depend only on the core, not the lookahead
  - Reduce-reduce conflicts might be produced by merging

Example G' =
  $S' \to S$
  $S \to$ aAd | bBd | aBe | bAe
  $A \to$ c
  $B \to$ c
 goto (I_0, a) =
 I_2: $S \to$ a $\cdot$ Ad | a $\cdot$ Be, \$
    $A \to \cdot$ c, d
    $B \to \cdot$ c, e
 goto (I_2, c) =
 I_4: $A \to$ c $\cdot$, d
    $B \to$ c $\cdot$, e

I_0: $S' \to \cdot S$, \$
   $S \to \cdot$ aAd | $\cdot$ bBd | $\cdot$ aBe | $\cdot$ bAe, \$

goto (I_0, S) = I_1: $S' \to S \cdot$, \$

goto (I_0, b) =
I_3: $S \to$ b $\cdot$ Bd | b $\cdot$ Ae, \$
   $A \to \cdot$ c, e
   $B \to \cdot$ c, d
goto (I_3, c) =
I_5: $A \to$ c $\cdot$, e
   $B \to$ c $\cdot$, d

merge I_4 and I_5 =
I_{45}: $A \to$ c $\cdot$, d/e
   $B \to$ c $\cdot$, d/e

Using Ambiguous Grammars

- Every ambiguous grammar fails to be LR
  - Certain types of ambiguous grammars are useful in the specification and implementation of languages
    - An ambiguous grammar provides a shorter, more natural specification than any equivalent unambiguous grammar
      e.g. $E \to E + E \mid E * E \mid (E) \mid$ id
    - Ambiguous grammars can be used to isolate commonly occurring syntactic constructs for special case optimization
  - Disambiguating rules must be specified to allow only one parse tree for each sentence
    - e.g. using precedence and associativity to resolve conflicts
    - In this way, the overall language specification still remains unambiguous

Using Precedence and Associativity

Example G'

(0) $E' \to E$
(1) $E \to E + E$
(2) $E \to E * E$
(3) $E \to (E)$
(4) $E \to \text{id}$

I_0: $E' \to \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \text{id}$

$goto(I_0, E) =$
I_1: $E' \to E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

$goto(I_0, \text{`('}) =$
I_2: $E \to (\cdot E)$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \text{id}$

$goto(I_0, \text{id}) =$
I_3: $E \to \cdot \text{id}$

$goto(I_1, \text{`+'}) =$
I_4: $E \to E + \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \text{id}$

$goto(I_1, \text{`*'}) =$
I_5: $E \to E * \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \text{id}$

$goto(I_0, E) =$
I_6: $E \to (E \cdot)$
$E \to E \cdot + E$
$E \to E \cdot * E$

$goto(I_4, E) =$
I_7: $E \to E + E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

$goto(I_5, E) =$
I_8: $E \to E * E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

$goto(I_6, \text{`)'}) =$
I_6: $E \to (E) \cdot$

	action			*goto*
	+	*	…	
…				
7	r2	s5		
8	r3	r3		
9				

The "Dangling-else" Ambiguity

Example G'

(0) $S' \to S$
(1) $S \to \text{i}S\text{e}S$
(2) $S \to \text{i}S$
(3) $S \to \text{a}$

I_0: $S' \to \cdot S$
$S \to \cdot \text{i}S\text{e}S$
$S \to \cdot \text{i}S$
$S \to \cdot \text{a}$

$goto(I_0, S) =$
I_1: $S' \to S \cdot$

$goto(I_0, \text{i}) =$
I_2: $S \to \text{i} \cdot S\text{e}S$
$S \to \text{i} \cdot S$
$S \to \cdot \text{i}S\text{e}S$
$S \to \cdot \text{i}S$
$S \to \cdot \text{a}$

$goto(I_0, \text{a}) =$
I_3: $S \to \text{a} \cdot$

$goto(I_2, S) =$
I_4: $S \to \text{i}S \cdot \text{e}S$
$S \to \text{i}S \cdot$

$goto(I_4, \text{e}) =$
I_5: $S \to \text{i}S\text{e} \cdot S$
$S \to \cdot \text{i}S\text{e}S$
$S \to \cdot \text{i}S$
$S \to \cdot \text{a}$

$goto(I_5, S) =$
I_6: $S \to \text{i}S\text{e}S \cdot$

	action			*goto*
	i	e	…	
…				
4		s5		
8				
9				

Ambiguities from Special-Case Productions

Example G'

(0) $E' \to E$
(1) $E \to E \text{ sub } E \text{ sup } E$
(2) $E \to E \text{ sub } E$
(3) $E \to E \text{ sup } E$
(4) $E \to \{E\}$
(5) $E \to \text{c}$

I_0: $E' \to \cdot E$
$E \to \cdot E \text{ sub } E \text{ sup } E$
$E \to \cdot E \text{ sub } E$
$E \to \cdot E \text{ sup } E$
$E \to \cdot \{E\}$
$E \to \cdot \text{c}$

I_7: $E' \to E \cdot$
$E \to E \cdot \text{sub } E \text{ sup } E$
$E \to E \text{ sub } E \cdot \text{sup } E$
$E \to E \cdot \text{sub } E$
$E \to E \text{ sub } E \cdot$
$E \to E \cdot \text{sup } E$

I_8: $E' \to E \cdot$
$E \to E \cdot \text{sub } E \text{ sup } E$
$E \to E \cdot \text{sub } E$
$E \to E \cdot \text{sub } E$
$E \to E \text{ sup } E \cdot$

I_{11}: $E' \to E \cdot$
$E \to E \cdot \text{sub } E \text{ sup } E$
$E \to E \text{ sub } E \text{ sup } E \cdot$
$E \to E \cdot \text{sub } E$
$E \to E \cdot \text{sup } E$
$E \to E \text{ sup } E \cdot$

	action				*goto*
	sub	sup	}	$	
…					
7	s4 r2	s10 r2	r2	r2	
8	s4 r3	s5 r3	r3	r3	
9					
11	s4 r1 r3	s5 r1 r3	r1 r3	r1 r3	

Operator-Precedence Parsing

- Works on a class of grammars: *operator grammars*
  - No production right side is ε or has two adjacent nonterminals
  - e.g. $E \to EAE \mid (E) \mid -E \mid \text{id}$
    $A \to + \mid - \mid * \mid /$
    is not an operator grammar
  - $E \to E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid -E \mid \text{id}$
    is an operator grammar
- Advantages
  - Easy to implement
- Disadvantages
  - Hard to handle tokens like the minus sign (with 2 precedences)
  - Can't be sure the parser accepts exactly the desired language
  - Only a small class of grammars can be parsed

Operator-Precedence Parsing

- Precedence relations

Relation	Meaning
a <· b	a "yields precedence to" b
a ≐ b	a "has the same precedence as" b
a ·> b	a "takes precedence over" b

  - Example $E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid -E \mid$ id

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Operator-Precedence Parsing

- The handle can be found by the steps
  - Scan the string from left until the first ·> is encountered
  - Then scan back backwards over any ≐ until a <· is encountered
  - The handle contains everything within the <· and ·> above, including any intervening or surrounding nonterminals
  - e.g. w = id + id * id

    $ <· id ·> + <· id ·> * <· id ·> $

    $ <· *E* + <· id ·> * <· id ·> $

    $ <· *E* + <· *E* * <· id ·> $

    $ <· *E* + <· *E* * *E* ·> $

    $ <· *E* + *E* ·> $

    $*E*$