PUNE VIDYARTHI GRIHA's

COLLEGE OF ENGINEERING, NASHIK.

• "*SYNTAX ANALYSIS CFG*"

PREPARED BY :

PROF. ANAND N. GHARU

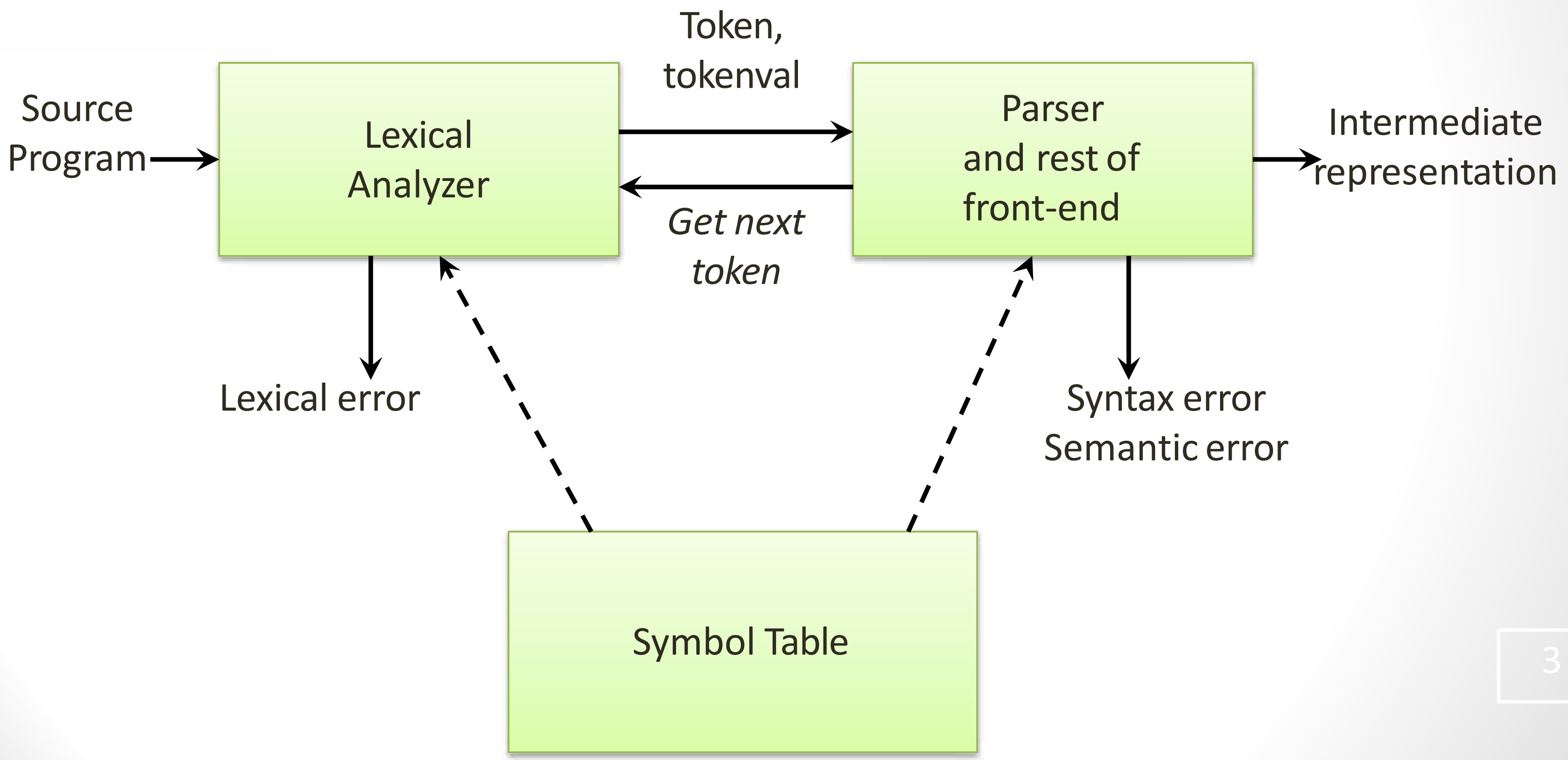
ASSISTANT PROFESSOR

COMPUTER DEPARTMENT

SUBJECT – COMPILER (BE COMPUTER SPPU-2019)

Syntax Analysis

Position of a Parser in the Compiler Model



The Role Of Parser

- A parser implements a C-F grammar
- The role of the parser is two fold:

1. To check syntax (= string recognizer)
   - And to report syntax errors accurately
2. To invoke semantic actions
   - For static semantics checking, e.g. type checking of expressions, functions, etc.
   - For syntax-directed translation of the source code to an intermediate representation

Syntax-Directed Translation

- One of the major roles of the parser is to produce an intermediate representation (IR) of the source program using syntax-directed translation methods
- Possible IR output:
  - Abstract syntax trees (ASTs)
  - Control-flow graphs (CFGs) with triples, three-address code, or register transfer list notation
  - WHIRL (SGI Pro64 compiler) has 5 IR levels!

Error Handling

- A good compiler should assist in identifying and locating errors
  - *Lexical errors*: important, compiler can easily recover and continue
  - *Syntax errors*: most important for compiler, can almost always recover
  - *Static semantic errors*: important, can sometimes recover
  - *Dynamic semantic errors*: hard or impossible to detect at compile time, runtime checks are required
  - *Logical errors*: hard or impossible to detect

Viable-Prefix Property

- The *viable-prefix property* of LL/LR parsers allows early detection of syntax errors
  - Goal: detection of an error *as soon as possible* without further consuming unnecessary input
  - How: detect an error as soon as the prefix of the input does not match a prefix of any string in the language

```
…
for (;)
…
```

Prefix

Error is detected here

```
…
DO 10 I = 1;0
…
```

Prefix

Error is detected here

Error Recovery Strategies

- *Panic mode*
  - Discard input until a token in a set of designated synchronizing tokens is found
- *Phrase-level recovery*
  - Perform local correction on the input to repair the error
- *Error productions*
  - Augment grammar with productions for erroneous constructs
- *Global correction*
  - Choose a minimal sequence of changes to obtain a global least-cost correction

Grammars (Recap)

- Context-free grammar is a 4-tuple $G = (N, T, P, S)$ where
  - T is a finite set of tokens (*terminal* symbols)
  - N is a finite set of *nonterminals*
  - P is a finite set of *productions* of the form

    $\alpha \to \beta$

    where $\alpha \in (N \cup T)^* \, N \, (N \cup T)^*$ and $\beta \in (N \cup T)^*$
  - $S \in N$ is a designated *start symbol*

Notational Conventions Used

- Terminals
  $a,b,c,\ldots \in T$
  specific terminals: **0**, **1**, **id**, **+**
- Nonterminals
  $A,B,C,\ldots \in N$
  specific nonterminals: *expr*, *term*, *stmt*
- Grammar symbols
  $X,Y,Z \in (N \cup T)$
- Strings of terminals
  $u,v,w,x,y,z \in T^*$
- Strings of grammar symbols
  $\alpha,\beta,\gamma \in (N \cup T)^*$

Derivations (Recap)

- The *one-step derivation* is defined by

  $$\alpha\, A\, \beta \Rightarrow \alpha\, \gamma\, \beta$$

  where $A \rightarrow \gamma$ is a production in the grammar
- In addition, we define
  - $\Rightarrow$ is *leftmost* $\Rightarrow_{lm}$ if α does not contain a nonterminal
  - $\Rightarrow$ is *rightmost* $\Rightarrow_{rm}$ if β does not contain a nonterminal
  - Transitive closure $\Rightarrow^*$ (zero or more steps)
  - Positive closure $\Rightarrow^+$ (one or more steps)
- The *language generated by G* is defined by

  $$L(G) = \{w \in T^* \mid S \Rightarrow^+ w\}$$

Derivation (Example)

Grammar $G = (\{E\}, \{\mathbf{+},\mathbf{*},\mathbf{(},\mathbf{)},\mathbf{-},\mathbf{id}\}, P, E)$ with
productions $P =$ $E \to E \mathbf{+} E$
$E \to E \mathbf{*} E$
$E \to \mathbf{(}\ E\ \mathbf{)}$
$E \to \mathbf{-}\ E$
$E \to \mathbf{id}$

Example derivations:

$E \Rightarrow \mathbf{-}\ E \Rightarrow \mathbf{-\ id}$

$E \Rightarrow_{rm} E \mathbf{+} E \Rightarrow_{rm} E \mathbf{+ id} \Rightarrow_{rm} \mathbf{id + id}$

$E \Rightarrow^{*} E$

$E \Rightarrow^{*} \mathbf{id + id}$

$E \Rightarrow^{+} \mathbf{id * id + id}$

Chomsky Hierarchy: Language Classification

- A grammar G is said to be
  - *Regular* if it is *right linear* where each production is of the form
    $A \to w\, B$ or $A \to w$
    or *left linear* where each production is of the form
    $A \to B\, w$ or $A \to w$
  - *Context free* if each production is of the form
    $A \to \alpha$
    where $A \in N$ and $\alpha \in (N \cup T)^*$
  - *Context sensitive* if each production is of the form
    $\alpha\, A\, \beta \to \alpha\, \gamma\, \beta$
    where $A \in N$, $\alpha, \gamma, \beta \in (N \cup T)^*$, $|\gamma| > 0$
  - *Unrestricted*

Chomsky Hierarchy

$\mathrm{L}(\textit{regular}) \subset \mathrm{L}(\textit{context free}) \subset \mathrm{L}(\textit{context sensitive}) \subset \mathrm{L}(\textit{unrestricted})$

Where $\mathrm{L}(T) = \{ L(G) \mid G \text{ is of type } T \}$
That is: the set of all languages
generated by grammars G of type T

Examples:

Every *finite language* is regular! (construct a FSA for strings in $L(G)$)

$L_1 = \{ \mathbf{a}^n\mathbf{b}^n \mid n \geq 1 \}$ is context free

$L_2 = \{ \mathbf{a}^n\mathbf{b}^n\mathbf{c}^n \mid n \geq 1 \}$ is context sensitive

Parsing

Parser	Top-Down	BackTrack	**Recursive Descent**
		Non-BackTrack	**(Predictive/(Non -Recursive Descent/LL(1))**
	Bottom-Up	Operator Precedence	
		Shift Reduce	**SLR/LR(0)** **Canonical LR or LR(1)** **LALR**

Down…Recursive Descent…BackTrack P…

- Recursive descent parsing is a top-down method of syntax analysis in which a set **recursive procedures** to process the input **is executed**.
- **A procedure is associated with each nonterminal of a grammar.**
- Top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string.
- Equivalently, it attempts to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.
- **Recursive descent parsing involves backtracking**.

Top-Down Parsing…Non-Recursive

- LL methods (Left-to-right, Leftmost derivation)

Grammar:

$E \rightarrow T + T$

$T \rightarrow (E)$

$T \rightarrow - E$

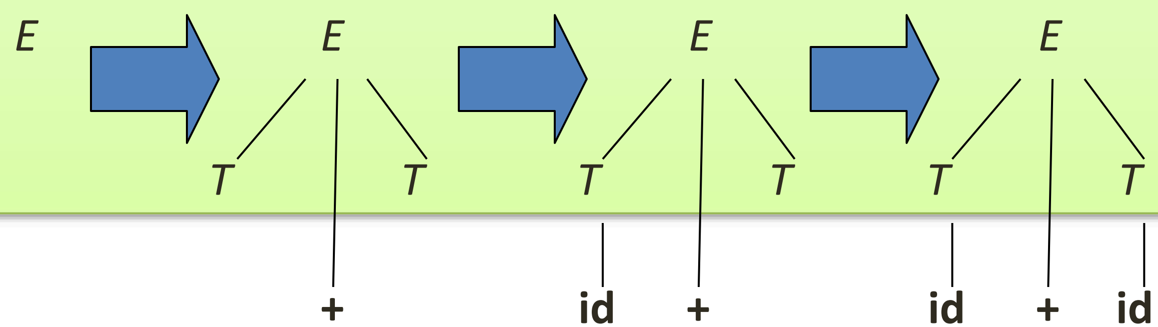
$T \rightarrow \mathbf{id}$

Leftmost derivation:

$E \Rightarrow_{lm} T + T$

$\Rightarrow_{lm} \mathbf{id} + T$

$\Rightarrow_{lm} \mathbf{id} + \mathbf{id}$



Predictive Parsing…LL(1) Parser

- Eliminate left recursion from grammar
- Left factor the grammar
- Compute FIRST and FOLLOW
- Two variants:
  - Recursive (recursive calls)
  - Non-recursive (table-driven)

Left Recursion (Recap)

- Productions of the form

  $$A \rightarrow A\,\alpha$$
  $$/\ \beta$$
  $$|\ \gamma$$

  are left recursive

- When one of the productions in a grammar is left recursive then a predictive parser loops forever on certain inputs

General Left Recursion Elimination Method

Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$

for $i = 1, \ldots, n$ **do**

  for $j = 1, \ldots, i\text{-}1$ **do**

    replace each

      $A_i \to A_j \gamma$

    with

      $A_i \to \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$

    where

      $A_j \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$

  enddo

  eliminate the *immediate left recursion* in A_i

enddo

Immediate Left-Recursion Elimination Method

Rewrite every left-recursive production

$$
\begin{aligned}
A \rightarrow\ & A\,\alpha \\
/\ & \beta \\
|\ & \gamma \\
|\ & A\,\delta
\end{aligned}
$$

into a right-recursive production:

$$
\begin{aligned}
A \rightarrow\ & \beta\,A_R \\
/\ & \gamma\,A_R \\
A_R \rightarrow\ & \alpha A_R \\
/\ & \delta\,A_R \\
/\ & \varepsilon
\end{aligned}
$$

Example Left Recursion Elim.

$A \to B\ C \mid \mathbf{a}$
$B \to C\ A \mid A\ \mathbf{b}$
$C \to A\ B \mid C\ C \mid \mathbf{a}$

Choose arrangement: *A*, *B*, *C*

$i = 1$:	nothing to do	
$i = 2, j = 1$:	$B \to C\ A \mid \underline{A}\ \mathbf{b}$	
	$\Rightarrow$	$B \to C\ A \mid \underline{B\ C}\ \mathbf{b} \mid \underline{\mathbf{a}}\ \mathbf{b}$
	$\Rightarrow_{\text{(imm)}}$	$B \to C\ A\ B_R \mid \mathbf{a}\ \mathbf{b}\ B_R$
		$B_R \to C\ \mathbf{b}\ B_R \mid \varepsilon$
$i = 3, j = 1$:	$C \to \underline{A}\ B \mid C\ C \mid \mathbf{a}$	
	$\Rightarrow$	$C \to \underline{B\ C}\ B \mid \underline{\mathbf{a}}\ B \mid C\ C \mid \mathbf{a}$
$i = 3, j = 2$:	$C \to \underline{B}\ C\ B \mid \mathbf{a}\ B \mid C\ C \mid \mathbf{a}$	
	$\Rightarrow$	$C \to \underline{C\ A\ B_R}\ C\ B \mid \underline{\mathbf{a}\ \mathbf{b}\ B_R}\ C\ B \mid \mathbf{a}\ B \mid C\ C \mid \mathbf{a}$
	$\Rightarrow_{\text{(imm)}}$	$C \to \mathbf{a}\ \mathbf{b}\ B_R\ C\ B\ C_R \mid \mathbf{a}\ B\ C_R \mid \mathbf{a}\ C_R$
		$C_R \to A\ B_R\ C\ B\ C_R \mid C\ C_R \mid \varepsilon$

Left Factoring

- When a nonterminal has two or more productions whose right-hand sides start with the same grammar symbols, the grammar is not LL(1) and cannot be used for predictive parsing
- Replace productions

$$A \rightarrow \alpha\, \beta_1 \,/\, \alpha\, \beta_2 \,/\, \dots \mid \alpha\, \beta_n \mid \gamma$$

with

$$A \rightarrow \alpha\, A_R \mid \gamma$$
$$A_R \rightarrow \beta_1 \,/\, \beta_2 \,/\, \dots \,/\, \beta_n$$

FIRST (Revisited)

- FIRST(α) = { *the set of terminals that begin all strings derived from* α }

  FIRST(a) = {a} if $a \in T$
  FIRST(ε) = {ε}
  FIRST(A) = $\cup_{A\to\alpha}$ FIRST(α) for $A\to\alpha \in P$
  FIRST($X_1X_2\ldots X_k$) =
  if for all j = 1, …, i-1 : $\varepsilon \in$ FIRST(X_j) **then**
  add non-ε in FIRST(X_i) to FIRST($X_1X_2\ldots X_k$)
  if for all j = 1, …, k : $\varepsilon \in$ FIRST(X_j) **then**
  add ε to FIRST($X_1X_2\ldots X_k$)

FOLLOW

- FOLLOW(A) = { *the set of terminals that can immediately follow nonterminal A* }

  FOLLOW(A) =
  for all ($B \rightarrow \alpha\ A\ \beta$) $\in P$ **do**
  add FIRST(β)\{ε} to FOLLOW(A)
  for all ($B \rightarrow \alpha\ A\ \beta$) $\in P$ and $\varepsilon \in$ FIRST(β) **do**
  add FOLLOW(B) to FOLLOW(A)
  for all ($B \rightarrow \alpha\ A$) $\in P$ **do**
  add FOLLOW(B) to FOLLOW(A)
  if A is the start symbol S **then**
  add **$** to FOLLOW($A$)

$$\text{First}(\alpha = A\beta) = \begin{cases} \text{First}(A), \text{ if } \lambda \notin \text{First}(A) \\ \text{First}(A) - \{\lambda\} \cup \text{First}(\beta), \text{ if } \lambda \in \text{First}(A) \end{cases}$$

Red : A   Blue : β

$$G_0 \begin{cases} S \rightarrow aSe \\ S \rightarrow B \\ B \rightarrow bBe \\ B \rightarrow C \\ C \rightarrow cCe \\ C \rightarrow d \end{cases}$$

$$\text{First}(\alpha = A\beta) = \begin{cases} \text{First}(A), \text{ if } \lambda \notin \text{First}(A) \\ \text{First}(A) - \{\lambda\} \cup \text{First}(\beta), \text{ if } \lambda \in \text{First}(A) \end{cases}$$

Red : A Blue : β

$$G_0 \begin{cases} S \to aSe \\ S \to B \\ B \to bBe \\ B \to C \\ C \to cCe \\ C \to d \end{cases}$$

Step 1:

- First ($S \to aSe$) = First(a) = {a}
- First ($S \to B\lambda$) = First(B)
- First ($B \to bBe$) = First(b)={b}
- First ($B \to C\lambda$) = First(C)
- First ($C \to cCe$) = First(c) ={c}
- First ($C \to d\lambda$) = First(d)={d}

$$\text{First}(\alpha = A\beta) = \begin{cases} \text{First}(A), \text{ if } \lambda \notin \text{First}(A) \\ \text{First}(A) - \{\lambda\} \cup \text{First}(\beta), \text{ if } \lambda \in \text{First}(A) \end{cases}$$

$$G_0 \begin{cases} S \to aSe \\ S \to B \\ B \to bBe \\ B \to C \\ C \to cCe \\ C \to d \end{cases}$$

Red : A Blue : β

Step 1:

- First ($S \to aSe$) = {a}
- First ($S \to B\lambda$) = First(B)
- First ($B \to bBe$) = {b}
- First ($B \to C\lambda$) = First(C)
- First ($C \to cCe$) = {c}
- First ($C \to d\lambda$) = {d}

Step	First Set						
	S	B	C	a	b	c	d
Step 1	{a}UFirst(B)	{b}UFirst(C)	{c, d}				

$$\text{First}(\alpha = A\beta) = \begin{cases} \text{First}(A), \text{ if } \lambda \notin \text{First}(A) \\ \text{First}(A) - \{\lambda\} \cup \text{First}(\beta), \text{ if } \lambda \in \text{First}(A) \end{cases}$$

Red : A Blue : β

$$G_0 \begin{cases} S \to aSe \\ S \to B \\ B \to bBe \\ B \to C \\ C \to cCe \\ C \to d \end{cases}$$

Step 2:

- First ($S \to aSe$) = {a}
- First ($S \to B\lambda$) = First(B) = {b}UFirst(C)
- First ($B \to bBe$) = {b}
- First ($B \to C\lambda$) = First(C)
- First ($C \to cCe$) = {c}
- First ($C \to d\lambda$) = {d}

Step	First Set						
	S	B	C	a	b	c	d
Step 1	{a}UFirst(B)	{b}UFirst(C)	{c, d}				

$$\text{First}(\alpha = A\beta) = \begin{cases} \text{First}(A), \text{ if } \lambda \notin \text{First}(A) \\ \text{First}(A) - \{\lambda\} \cup \text{First}(\beta), \text{ if } \lambda \in \text{First}(A) \end{cases}$$

$$G_0 \begin{cases} S \to aSe \\ S \to B \\ B \to bBe \\ B \to C \\ C \to cCe \\ C \to d \end{cases}$$

Red : A Blue : β

Step 2:

- First ($S \to aSe$) = {a}
- First ($S \to B\lambda$) = {b}UFirst(C)
- First ($B \to bBe$) = {b}
- First ($B \to C\lambda$) = First(C)
- First ($C \to cCe$) = {c}
- First ($C \to d\lambda$) = {d}

Step	First Set						
	S	B	C	a	b	c	d
Step 1	{a}UFirst(B)	{b}UFirst(C)	{c, d}				
Step 2	{a}U {b}UFirst(C)						

$$\text{First}(\alpha = A\beta) = \begin{cases} \text{First}(A), \text{ if } \lambda \notin \text{First}(A) \\ \text{First}(A) - \{\lambda\} \cup \text{First}(\beta), \text{ if } \lambda \in \text{First}(A) \end{cases}$$

Red : A Blue : β

$$G_0 \begin{cases} S \to aSe \\ S \to B \\ B \to bBe \\ B \to C \\ C \to cCe \\ C \to d \end{cases}$$

Step 2:

- First ($S \to aSe$) = {a}
- First ($S \to B\lambda$) = {b}UFirst(c)
- First ($B \to bBe$) = {b}
- First ($B \to C\lambda$) = First(c) = {c, d}
- First ($C \to cCe$) = {c}
- First ($C \to d\lambda$) = {d}

Step	First Set						
	S	B	C	a	b	c	d
Step 1	{a}UFirst(B)	{b}UFirst(c)	{c, d}				
Step 2	{a}U {b}UFirst(c)						

$$\text{First}(\alpha = A\beta) = \begin{cases} \text{First}(A), \text{ if } \lambda \notin \text{First}(A) \\ \text{First}(A) - \{\lambda\} \cup \text{First}(\beta), \text{ if } \lambda \in \text{First}(A) \end{cases}$$

$$G_0 \begin{cases} S \rightarrow aSe \\ S \rightarrow B \\ B \rightarrow bBe \\ B \rightarrow C \\ C \rightarrow cCe \\ C \rightarrow d \end{cases}$$

Red : A Blue : β

Step 2:

- First ($S \rightarrow aSe$) = {a}
- First ($S \rightarrow B\lambda$) = {b}UFirst(c)
- First ($B \rightarrow bBe$) = {b}
- First ($B \rightarrow C\lambda$) = {c, d}
- First ($C \rightarrow cCe$) = {c}
- First ($C \rightarrow d\lambda$) = {d}

Step	First Set						
	S	B	C	a	b	c	d
Step 1	{a}UFirst(B)	{b}UFirst(c)	{c, d}				
Step 2	{a}U {b}UFirst(c)						

$$\text{First}(\alpha = A\beta) = \begin{cases} \text{First}(A), \text{ if } \lambda \notin \text{First}(A) \\ \text{First}(A) - \{\lambda\} \cup \text{First}(\beta), \text{ if } \lambda \in \text{First}(A) \end{cases}$$

$$G_0 \begin{cases} S \to aSe \\ S \to B \\ B \to bBe \\ B \to C \\ C \to cCe \\ C \to d \end{cases}$$

Red : A Blue : β

Step 3:

- First ($S \to aSe$) = {a}
- First ($S \to B\lambda$) = {b}UFirst(C) = {b}U {c, d}
- First ($B \to bBe$) = {b}
- First ($B \to C\lambda$) = {c, d}
- First ($C \to cCe$) = {c}
- First ($C \to d\lambda$) = {d}

Step	First Set						
	S	B	C	a	b	c	d
Step 1	{a}UFirst(B)	{b}UFirst(C)	{c, d}				
Step 2	{a}U {b}UFirst(C)						

$$\text{First}(\alpha = A\beta) = \begin{cases} \text{First}(A), \text{ if } \lambda \notin \text{First}(A) \\ \text{First}(A) - \{\lambda\} \cup \text{First}(\beta), \text{ if } \lambda \in \text{First}(A) \end{cases}$$

Red : A Blue : β

$$G_0 \begin{cases} S \rightarrow aSe \\ S \rightarrow B \\ B \rightarrow bBe \\ B \rightarrow C \\ C \rightarrow cCe \\ C \rightarrow d \end{cases}$$

Step 3:

- First $(S \rightarrow aSe)$ = {a}
- First $(S \rightarrow B\lambda)$ = {b, c, d}
- First $(B \rightarrow bBe)$ = {b}
- First $(B \rightarrow C\lambda)$ = {c, d}
- First $(C \rightarrow cCe)$ = {c}
- First $(C \rightarrow d\lambda)$ = {d}

Step	First Set						
	S	B	C	a	b	c	d
Step 1	{a}UFirst(B)	{b}UFirst(C)	{c, d}				
Step 2	{a}U {b}UFirst(C)	{b}U{c, d} = {b,c,d}	{c, d}				
Step 3	{a}U {b}U{c, d} = {a,b,c,d}	{b}U{c, d} = {b,c,d}	{c, d}				

$$\text{First}(\alpha = A\beta) = \begin{cases} \text{First}(A), \text{ if } \lambda \notin \text{First}(A) \\ \text{First}(A) - \{\lambda\} \cup \text{First}(\beta), \text{ if } \lambda \in \text{First}(A) \end{cases}$$

$$G_0 \begin{cases} S \to aSe \\ S \to B \\ B \to bBe \\ B \to C \\ C \to cCe \\ C \to d \end{cases}$$

Red : A Blue : β

Step 3:

- First ($S \to aSe$) = {a}
- First ($S \to B\lambda$) = {b, c, d}
- First ($B \to bBe$) = {b}
- First ($B \to C\lambda$) = {c, d}
- First ($C \to cCe$) = {c}
- First ($C \to d\lambda$) = {d}

If no more change…
The first set of a terminal symbol is itself

Step	First Set						
	S	B	C	a	b	c	d
Step 1	{a}UFirst(B)	{b}UFirst(C)	{c, d}				
Step 2	{a}U {b}UFirst(C)	{b}U{c, d} = {b,c,d}	{c, d}				
Step 3	{a}U {b}U{c, d} = {a,b,c,d}	{b}U{c, d} = {b,c,d}	{c, d}	{a}	{b}	{c}	{d}

Another Example….

$$\text{First}(\alpha = A\beta) = \begin{cases} \text{First}(A), \text{ if } \lambda \notin \text{First}(A) \\ \text{First}(A) - \{\lambda\} \cup \text{First}(\beta), \text{ if } \lambda \in \text{First}(A) \end{cases}$$

Red : A Blue : β

$$G_0 \left\{ \begin{array}{l} S \rightarrow ABc \\ A \rightarrow a \\ A \rightarrow \lambda \\ B \rightarrow b \\ B \rightarrow \lambda \end{array} \right.$$

$$\text{First}(\alpha = A\beta) = \begin{cases} \text{First}(A), \text{ if } \lambda \notin \text{First}(A) \\ \text{First}(A) - \{\lambda\} \cup \text{First}(\beta), \text{ if } \lambda \in \text{First}(A) \end{cases}$$

$$G_0 \begin{cases} S \rightarrow ABc \\ A \rightarrow a \\ A \rightarrow \lambda \\ B \rightarrow b \\ B \rightarrow \lambda \end{cases}$$

Red : A Blue : β

Step 1:

- First ($S \rightarrow ABc$) = First(ABc)
- First ($A \rightarrow a\lambda$) = First(a)
- First ($A \rightarrow \lambda\lambda$) = First($\lambda$)UFirst($\lambda$)
- First ($B \rightarrow b\lambda$) = First(b)
- First ($B \rightarrow \lambda\lambda$) = First($\lambda$)UFirst($\lambda$)

$$\text{First}(\alpha = A\beta) = \begin{cases} \text{First}(A), & \text{if } \lambda \notin \text{First}(A) \\ \text{First}(A) - \{\lambda\} \cup \text{First}(\beta), & \text{if } \lambda \in \text{First}(A) \end{cases}$$

$$G_0 \begin{cases} S \rightarrow ABc \\ A \rightarrow a \\ A \rightarrow \lambda \\ B \rightarrow b \\ B \rightarrow \lambda \end{cases}$$

Red : A   Blue : β

Step 1:

- $\text{First}(S \rightarrow ABc) = \text{First}(ABc)$
- $\text{First}(A \rightarrow a\lambda) = \{a\}$
- $\text{First}(A \rightarrow \lambda\lambda) = \{\lambda\}$
- $\text{First}(B \rightarrow b\lambda) = \{b\}$
- $\text{First}(B \rightarrow \lambda\lambda) = \{\lambda\}$

Step	First Set					
	S	A	B	a	b	c
Step 1	First(ABc)	{a, λ}	{b, λ}			

$$\text{First}(\alpha = A\beta) = \begin{cases} \text{First}(A), \text{ if } \lambda \notin \text{First}(A) \\ \text{First}(A) - \{\lambda\} \cup \text{First}(\beta), \text{ if } \lambda \in \text{First}(A) \end{cases}$$

Red : A   Blue : β

$$G_0 \begin{cases} S \rightarrow ABc \\ A \rightarrow a \\ A \rightarrow \lambda \\ B \rightarrow b \\ B \rightarrow \lambda \end{cases}$$

Step 2:

- First ($S \rightarrow ABc$) = First(ABc) = $\{a, \lambda\}$
  = $\{a, \lambda\}$ - $\{\lambda\}$ ∪ First(Bc)
  = $\{a\}$ ∪ First(Bc)
- First ($A \rightarrow a\lambda$) = $\{a\}$
- First ($A \rightarrow \lambda\lambda$) = $\{\lambda\}$
- First ($B \rightarrow b\lambda$) = $\{b\}$
- First ($B \rightarrow \lambda\lambda$) = $\{\lambda\}$

Step	First Set					
	S	A	B	a	b	c
Step 1	First(ABc)	{a, λ}	{b, λ}			
Step 2	{a} ∪ First(Bc)	{a, λ}	{b, λ}			

$$\text{First}(\alpha = A\beta) = \begin{cases} \text{First}(A), \text{ if } \lambda \notin \text{First}(A) \\ \text{First}(A) - \{\lambda\} \cup \text{First}(\beta), \text{ if } \lambda \in \text{First}(A) \end{cases}$$

Red : A Blue : β

$$G_0 \begin{cases} S \to ABc \\ A \to a \\ A \to \lambda \\ B \to b \\ B \to \lambda \end{cases}$$

Step 3:

- First ($S \to ABc$) $= \{a\} \cup \text{First}(Bc)$
  $= \{a\} \cup \{b, \lambda\}$
  $= \{a\} \cup \{b, \lambda\} - \{\lambda\} \cup \text{First}(c)$
  $= \{a\} \cup \{b,c\}$
- First ($A \to a\lambda$) $= \{a\}$
- First ($A \to \lambda\lambda$) $= \{\lambda\}$
- First ($B \to b\lambda$) $= \{b\}$
- First ($B \to \lambda\lambda$) $= \{\lambda\}$

Step	First Set					
	S	A	B	a	b	c
Step 1	**First(ABc)**	{a, λ}	{b, λ}			
Step 2	{a} U **First(Bc)**	{a, λ}	{b, λ}			
Step 3	{a} U {b, c}= {a,b,c}	{a, λ}	{b, λ}			

$$\text{First}(\alpha = A\beta) = \begin{cases} \text{First}(A), \text{ if } \lambda \notin \text{First}(A) \\ \text{First}(A) - \{\lambda\} \cup \text{First}(\beta), \text{ if } \lambda \in \text{First}(A) \end{cases}$$

Red : A Blue : β

$$G_0 \begin{cases} S \to ABc \\ A \to a \\ A \to \lambda \\ B \to b \\ B \to \lambda \end{cases}$$

Step 3:

- First $(S \to ABc)$ = $\{a,b,c\}$
- First $(A \to a\lambda)$ = $\{a\}$
- First $(A \to \lambda\lambda)$ = $\{\lambda\}$
- First $(B \to b\lambda)$ = $\{b\}$
- First $(B \to \lambda\lambda)$ = $\{\lambda\}$

If no more change…
The first set of a terminal symbol is itself

Step	First Set					
	S	A	B	a	b	c
Step 1	**First(ABc)**	$\{a, \lambda\}$	$\{b, \lambda\}$			
Step 2	{a} U **First(Bc)**	$\{a, \lambda\}$	$\{b, \lambda\}$			
Step 3	{a} U {b, c}= {a,b,c}	$\{a, \lambda\}$	$\{b, \lambda\}$	{a}	{b}	{c}

LL(1) Grammar

- A grammar G is LL(1) if it is not left recursive and for each collection of productions
  $$A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$
  for nonterminal A the following holds:

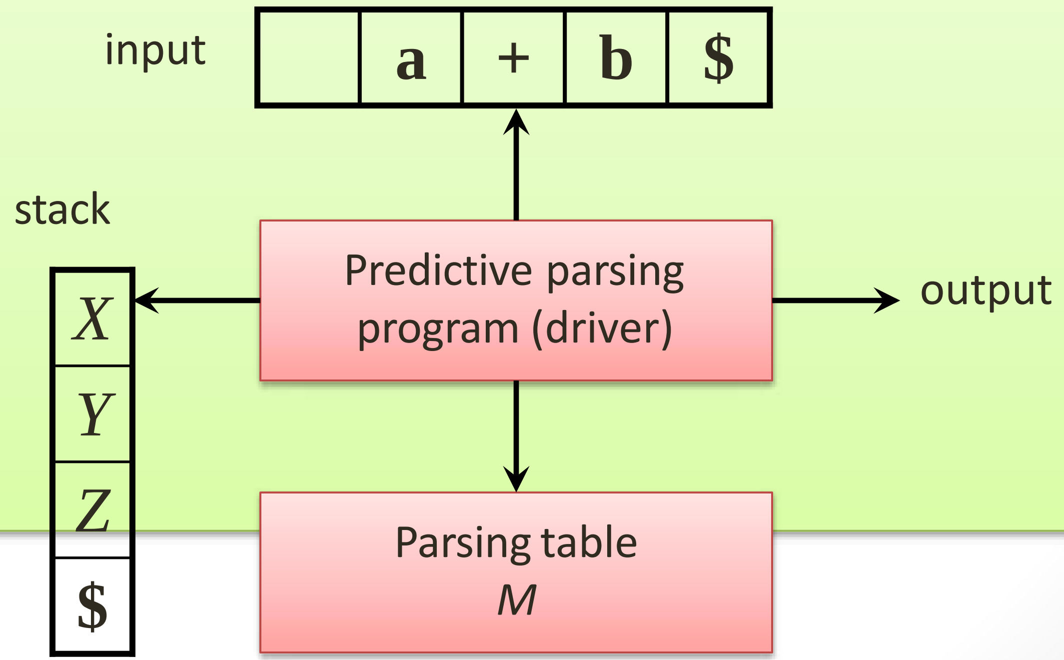
  1. $\mathrm{FIRST}(\alpha_i) \cap \mathrm{FIRST}(\alpha_j) = \emptyset$ for all $i \neq j$
  2. if $\alpha_i \Rightarrow^* \varepsilon$ then
     2.a. $\alpha_j \not\Rightarrow^* \varepsilon$ for all $i \neq j$
     2.b. $\mathrm{FIRST}(\alpha_j) \cap \mathrm{FOLLOW}(A) = \emptyset$
     for all $i \neq j$

Non-LL(1) Examples

Grammar	Not LL(1) because:
$S \rightarrow S\ a \mid a$	Left recursive
$S \rightarrow a\ S \mid a$	$\text{FIRST}(a\ S) \cap \text{FIRST}(a) \neq \emptyset$
$S \rightarrow a\ R \mid \varepsilon$ $R \rightarrow S \mid \varepsilon$	For R: $S \Rightarrow^* \varepsilon$ and $\varepsilon \Rightarrow^* \varepsilon$
$S \rightarrow a\ R\ a$ $R \rightarrow S \mid \varepsilon$	For R: $\text{FIRST}(S) \cap \text{FOLLOW}(R) \neq \emptyset$

Non-Recursive Predictive Parsing: Table-Driven Parsing

- Given an LL(1) grammar $G = (N, T, P, S)$ construct a table $M[A,a]$ for $A \in N$, $a \in T$ and use a *driver program* with a *stack*



Constructing an LL(1) Predictive Parsing Table

for each production $A \to \alpha$ **do**

  for each $a \in$ FIRST(α) **do**

    add $A \to \alpha$ to $M[A,a]$

  enddo

  if $\varepsilon \in$ FIRST(α) **then**

    for each $b \in$ FOLLOW(A) **do**

      add $A \to \alpha$ to $M[A,b]$

    enddo

  endif

enddo

Mark each undefined entry in M error

Example Table

$E \to T\,E_R$
$E_R \to \mathbf{+}\,T\,E_R \mid \varepsilon$
$T \to F\,T_R$
$T_R \to \mathbf{*}\,F\,T_R \mid \varepsilon$
$F \to \mathbf{(}\,E\,\mathbf{)} \mid \mathbf{id}$

$A \to \alpha$	FIRST(α)	FOLLOW(A)
$E \to T\,E_R$	(id	$)
$E_R \to + T\,E_R$	+	$)
$E_R \to \varepsilon$	ε	$)
$T \to F\,T_R$	(id	+ $)
$T_R \to *\,F\,T_R$	*	+ $)
$T_R \to \varepsilon$	ε	+ $)
$F \to (\,E\,)$	(	* + $)
$F \to id$	id	* + $)

	id	+	*	(	)	$
E	$E \to T\,E_R$			$E \to T\,E_R$		
E_R		$E_R \to + T\,E_R$			$E_R \to \varepsilon$	$E_R \to \varepsilon$
T	$T \to F\,T_R$			$T \to F\,T_R$		
T_R		$T_R \to \varepsilon$	$T_R \to *\,F\,T_R$		$T_R \to \varepsilon$	$T_R \to \varepsilon$
F	$F \to id$			$F \to (\,E\,)$		

LL(1) Grammars are Unambiguous

Ambiguous grammar
$S \rightarrow \mathbf{i}\ E\ \mathbf{t}\ S\ S_R \mid \mathbf{a}$
$S_R \rightarrow \mathbf{e}\ S \mid \varepsilon$
$E \rightarrow \mathbf{b}$

$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$S \rightarrow$ i E t S S_R	i	e \$
$S \rightarrow$ a	a	e \$
$S_R \rightarrow$ e S	e	e \$
$S_R \rightarrow \varepsilon$	ε	e \$
$E \rightarrow$ b	b	t

Error: duplicate table entry

	a	b	e	i	t	\$
S	$S \rightarrow$ a			$S \rightarrow$ i E t S S_R		
S_R			$S_R \rightarrow \varepsilon$ $S_R \rightarrow$ e S			$S_R \rightarrow \varepsilon$
E		$E \rightarrow$ b				

Predictive Parsing Program (Driver)

```
push($)
push(S)
a := lookahead
repeat
        X := pop()
        if X is a terminal or X = $ then
                match(X) // moves to next token and a := lookahead
        else if M[X,a] = X → Y1Y2…Yk then
                push(Yk, Yk-1, …, Y2, Y1) // such that Y1 is on top
                … invoke actions and/or produce IR output …
        else    error()
        endif
until X = $
```

Example Table-Driven Parsing

Stack	Input	Production applied
$\$\underline{E}$	$\underline{\mathbf{id}}\mathbf{+id*id\$}$	$E \to T\,E_R$
$\$E_R\underline{T}$	$\underline{\mathbf{id}}\mathbf{+id*id\$}$	$T \to F\,T_R$
$\$E_RT_R\underline{F}$	$\underline{\mathbf{id}}\mathbf{+id*id\$}$	$F \to \mathbf{id}$
$\$E_RT_R\underline{\mathbf{id}}$	$\underline{\mathbf{id}}\mathbf{+id*id\$}$	
$\$E_R\underline{T_R}$	$\underline{\mathbf{+}}\mathbf{id*id\$}$	$T_R \to \varepsilon$
$\$\underline{E_R}$	$\underline{\mathbf{+}}\mathbf{id*id\$}$	$E_R \to \mathbf{+}\,TE_R$
$\$E_RT\underline{\mathbf{+}}$	$\underline{\mathbf{+}}\mathbf{id*id\$}$	
$\$E_R\underline{T}$	$\underline{\mathbf{id}}\mathbf{*id\$}$	$T \to F\,T_R$
$\$E_RT_R\underline{F}$	$\underline{\mathbf{id}}\mathbf{*id\$}$	$F \to \mathbf{id}$
$\$E_RT_R\underline{\mathbf{id}}$	$\underline{\mathbf{id}}\mathbf{*id\$}$	
$\$E_R\underline{T_R}$	$\underline{\mathbf{*}}\mathbf{id\$}$	$T_R \to \mathbf{*}\,F\,T_R$
$\$E_RT_RF\underline{\mathbf{*}}$	$\underline{\mathbf{*}}\mathbf{id\$}$	
$\$E_RT_R\underline{F}$	$\underline{\mathbf{id}}\mathbf{\$}$	$F \to \mathbf{id}$
$\$E_RT_R\underline{\mathbf{id}}$	$\underline{\mathbf{id}}\mathbf{\$}$	
$\$E_R\underline{T_R}$	$\underline{\mathbf{\$}}$	$T_R \to \varepsilon$
$\$\underline{E_R}$	$\underline{\mathbf{\$}}$	$E_R \to \varepsilon$
$\underline{\$}$	$\underline{\mathbf{\$}}$	

Panic Mode Recovery

Add synchronizing actions to undefined entries based on FOLLOW

Pro: Can be automated
Cons: Error messages are needed

FOLLOW(E) = { **)** **$** }
FOLLOW(E_R) = { **)** **$** }
FOLLOW(T) = { **+** **)** **$** }
FOLLOW(T_R) = { **+** **)** **$** }
FOLLOW(F) = { **+** ***** **)** **$** }

	id	+	*	(	)	$
E	$E \rightarrow T\ E_R$			$E \rightarrow T\ E_R$	synch	synch
E_R		$E_R \rightarrow +\ T\ E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F\ T_R$	synch		$T \rightarrow F\ T_R$	synch	synch
T_R		$T_R \rightarrow \varepsilon$	$T_R \rightarrow *\ F\ T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow \text{id}$	synch	synch	$F \rightarrow (\ E\ )$	synch	synch

synch: the driver pops current nonterminal A and skips input till synch token or skips input until one of FIRST(A) is found

Phrase-Level Recovery

Change input stream by inserting missing tokens
For example: **id id** is changed into **id * id**

Pro: Can be automated
Cons: Recovery not always intuitive

Can then continue here

	id	+	*	(	)	$
E	$E \to T\, E_R$			$E \to T\, E_R$	synch	synch
E_R		$E_R \to +\, T\, E_R$			$E_R \to \varepsilon$	$E_R \to \varepsilon$
T	$T \to F\, T_R$	synch		$T \to F\, T_R$	synch	synch
T_R	insert *	$T_R \to \varepsilon$	$T_R \to *\, F\, T_R$		$T_R \to \varepsilon$	$T_R \to \varepsilon$
F	$F \to \mathrm{id}$	synch	synch	$F \to (E)$	synch	synch

***insert* ***: driver inserts missing * and retries the production

Error Productions

$E \rightarrow T\, E_R$
$E_R \rightarrow \mathbf{+}\, T\, E_R \mid \varepsilon$
$T \rightarrow F\, T_R$
$T_R \rightarrow \mathbf{*}\, F\, T_R \mid \varepsilon$
$F \rightarrow \mathbf{(}\, E\, \mathbf{)} \mid \mathbf{id}$

Add "*error production*":

$$T_R \rightarrow F\, T_R$$

to ignore missing *****, e.g.: **id id**

Pro: Powerful recovery method
Cons: Cannot be automated

	id	+	*	(	)	$
E	$E \rightarrow T\, E_R$			$E \rightarrow T\, E_R$	synch	synch
E_R		$E_R \rightarrow +\, T\, E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F\, T_R$	synch		$T \rightarrow F\, T_R$	synch	synch
T_R	$T_R \rightarrow F\, T_R$	$T_R \rightarrow \varepsilon$	$T_R \rightarrow *\, F\, T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow \text{id}$	synch	synch	$F \rightarrow (\, E\,)$	synch	synch

Bottom-Up Parsing

- LR methods (Left-to-right, Rightmost derivation)
  - SLR, Canonical LR, LALR
- Other special cases:
  - Shift-reduce parsing
  - Operator-precedence parsing

Operator-Precedence Parsing

- Special case of shift-reduce parsing
- We will not further discuss (you can skip textbook section 4.6)

Shift-Reduce Parsing

Grammar:

$S \rightarrow$ **a** A B **e**

$A \rightarrow A$ **b c** | **b**

$B \rightarrow$ **d**

Reducing a sentence:

a b b c d e

a A **b c d e**

a A **d e**

a A B **e**

S

These match production's right-hand sides

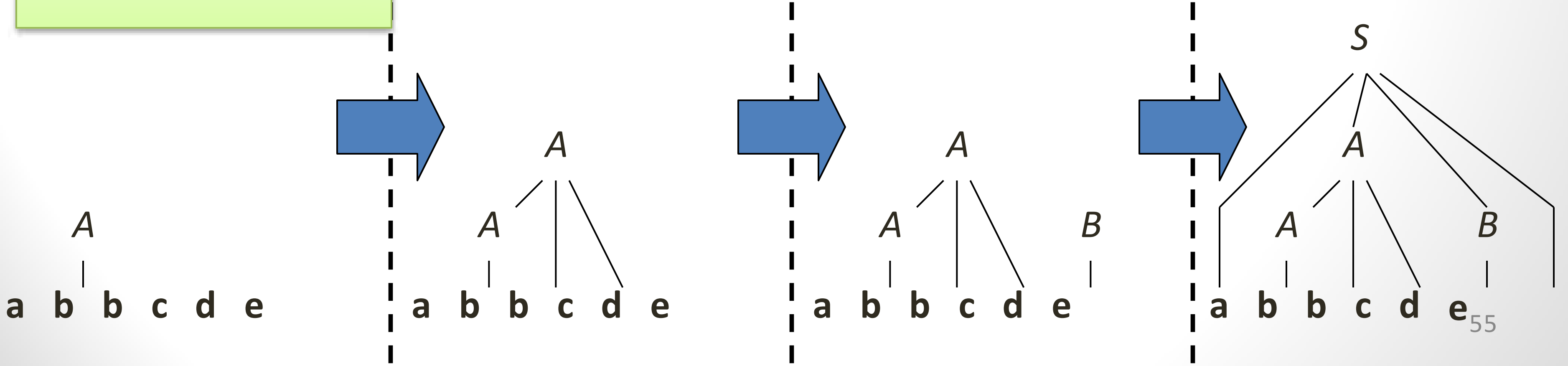
Shift-reduce corresponds to a rightmost derivation:

$S \Rightarrow_{rm}$ **a** A B **e**

$\Rightarrow_{rm}$ **a** A **d e**

$\Rightarrow_{rm}$ **a** A **b c d e**

$\Rightarrow_{rm}$ **a b b c d e**



Handles

A *handle* is a substring of grammar symbols in a *right-sentential form* that matches a right-hand side of a production

Grammar:

$S \rightarrow \mathbf{a}\ A\ B\ \mathbf{e}$

$A \rightarrow A\ \mathbf{b}\ \mathbf{c} \mid \mathbf{b}$

$B \rightarrow \mathbf{d}$

$\mathbf{a}\ \underline{\mathbf{b}}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ \underline{A\ \mathbf{b}\ \mathbf{c}}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ A\ \underline{\mathbf{d}}\ \mathbf{e}$

$\underline{\mathbf{a}\ A\ B\ \mathbf{e}}$

S

Handle

$\mathbf{a}\ \underline{\mathbf{b}}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ A\ \underline{\mathbf{b}}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ A\ A\ \mathbf{e}$

… ?

NOT a handle, because further reductions will fail (result is not a sentential form)

Implementation of Shift-Reduce Parsing

Grammar:
$E \to E + E$
$E \to E * E$
$E \to (E)$
$E \to \mathbf{id}$

Stack	Input	Action
$	**id+id*id$**	shift
$**id**	**+id*id$**	reduce $E \to$ **id**
$*E*	**+id*id$**	shift
$*E*+	**id*id$**	shift
$*E*+**id**	***id$**	reduce $E \to$ **id**
$*E*+*E*	***id$**	shift (or reduce?)
$*E*+*E**	**id$**	shift
$*E*+*E**id**	**$**	reduce $E \to$ **id**
$*E*+*E***E*	**$**	reduce $E \to E * E$
$*E*+*E*	**$**	reduce $E \to E + E$
$*E*	**$**	accept

Find handles to reduce

How to resolve conflicts?

Conflicts

- *Shift-reduce* and *reduce-reduce* conflicts are caused by
  - The limitations of the LR parsing method (even when the grammar is unambiguous)
  - Ambiguity of the grammar

Shift-Reduce Parsing: Shift-Reduce Conflicts

Stack	Input	Action
$…	**…$**	…
$…if *E* **then** *S*	**else…$**	shift or reduce?

Ambiguous grammar:
S → **if** *E* **then** *S*
| **if** *E* **then** *S* **else** *S*
| **other**

Resolve in favor of shift, so **else** matches closest **if**

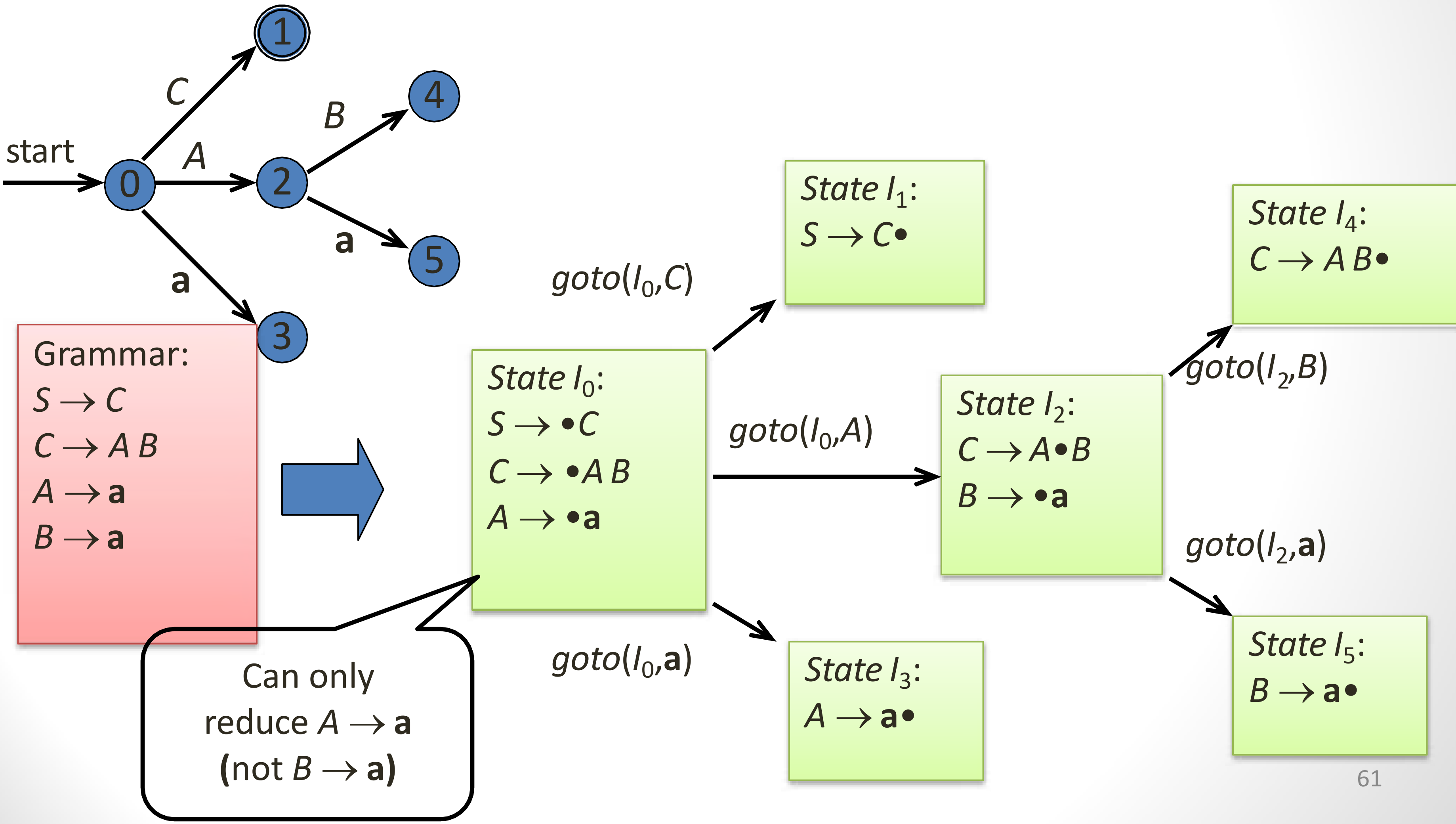
Shift-Reduce Parsing: Reduce-Reduce Conflicts

Stack	Input	Action
$	**aa$**	shift
$<u>a</u>	**a$**	reduce $A \to$ **a** <u>or</u> $B \to$ **a** ?

Grammar:
$C \to A\ B$
$A \to$ **a**
$B \to$ **a**

Resolve in favor
of reduce $A \to$ **a**,
otherwise we're stuck!

LR(k) Parsers: Use a DFA for Shift/Reduce Decisions



DFA for Shift/Reduce Decisions

Grammar:
$S \rightarrow C$
$C \rightarrow A\ B$
$A \rightarrow \mathbf{a}$
$B \rightarrow \mathbf{a}$

The states of the DFA are used to determine if a handle is on top of the stack

Stack	Input	Action
$ 0	**aa$**	start in state 0
$ 0	**aa$**	shift (and goto state 3)
$ 0 **a** 3	**a$**	reduce $A \rightarrow \mathbf{a}$ (goto 2)
$ 0 *A* 2	**a$**	shift (goto 5)
$ 0 *A* 2 **a** 5	**$**	reduce $B \rightarrow \mathbf{a}$ (goto 4)
$ 0 *A* 2 *B* 4	**$**	reduce $C \rightarrow AB$ (goto 1)
$ 0 *C* 1	**$**	accept ($S \rightarrow C$)

State I_0:
$S \rightarrow \bullet C$
$C \rightarrow \bullet A\ B$
$A \rightarrow \bullet \mathbf{a}$

goto(I_0,**a**)

State I_3:
$A \rightarrow \mathbf{a} \bullet$

DFA for Shift/Reduce Decisions

The states of the DFA are used to determine if a handle is on top of the stack

Grammar:
$S \to C$
$C \to A\ B$
$A \to \mathbf{a}$
$B \to \mathbf{a}$

Stack	Input	Action
$ 0	**aa$**	start in state 0
$ 0	**aa$**	shift (and goto state 3)
$ <u>0</u> <u>**a**</u> 3	**a$**	reduce $A \to \mathbf{a}$ (goto 2)
$ 0 *A* 2	**a$**	shift (goto 5)
$ 0 *A* 2 **a** 5	**$**	reduce $B \to \mathbf{a}$ (goto 4)
$ 0 *A* 2 *B* 4	**$**	reduce $C \to AB$ (goto 1)
$ 0 *C* 1	**$**	accept ($S \to C$)

State I_0:
$S \to \bullet C$
$C \to \bullet A\ B$
$A \to \bullet \mathbf{a}$

goto(I_0,A)

State I_2:
$C \to A \bullet B$
$B \to \bullet \mathbf{a}$

DFA for Shift/Reduce Decisions

Grammar:
$S \rightarrow C$
$C \rightarrow A\ B$
$A \rightarrow \mathbf{a}$
$B \rightarrow \mathbf{a}$

The states of the DFA are used to determine if a handle is on top of the stack

Stack	Input	Action
\$ 0	**aa\$**	start in state 0
\$ 0	**aa\$**	shift (and goto state 3)
\$ 0 **a** 3	**a\$**	reduce $A \rightarrow$ **a** (goto 2)
\$ 0 A <u>2</u>	<u>**a**</u>**\$**	shift (goto 5)
\$ 0 A 2 **a** 5	**\$**	reduce $B \rightarrow$ **a** (goto 4)
\$ 0 A 2 B 4	**\$**	reduce $C \rightarrow AB$ (goto 1)
\$ 0 C 1	**\$**	accept ($S \rightarrow C$)

State I_2:
$C \rightarrow A\bullet B$
$B \rightarrow \bullet\mathbf{a}$

goto(I_2,**a**)

State I_5:
$B \rightarrow \mathbf{a}\bullet$

DFA for Shift/Reduce Decisions

Grammar:
$S \rightarrow C$
$C \rightarrow A\ B$
$A \rightarrow \mathbf{a}$
$B \rightarrow \mathbf{a}$

The states of the DFA are used to determine if a handle is on top of the stack

Stack	Input	Action
\$ 0	**aa\$**	start in state 0
\$ 0	**aa\$**	shift (and goto state 3)
\$ 0 **a** 3	**a\$**	reduce $A \rightarrow \mathbf{a}$ (goto 2)
\$ 0 *A* 2	**a\$**	shift (goto 5)
\$ 0 *A* <u>2</u> <u>**a**</u> 5	**\$**	reduce $B \rightarrow \mathbf{a}$ (goto 4)
\$ 0 *A* 2 *B* 4	**\$**	reduce $C \rightarrow AB$ (goto 1)
\$ 0 *C* 1	**\$**	accept ($S \rightarrow C$)

State I_2:
$C \rightarrow A \bullet B$
$B \rightarrow \bullet \mathbf{a}$

$goto(I_2,B)$

State I_4:
$C \rightarrow A\ B\bullet$

DFA for Shift/Reduce Decisions

Grammar:
$S \rightarrow C$
$C \rightarrow A\ B$
$A \rightarrow \mathbf{a}$
$B \rightarrow \mathbf{a}$

The states of the DFA are used to determine if a handle is on top of the stack

Stack	Input	Action
$ 0	**aa$**	start in state 0
$ 0	**aa$**	shift (and goto state 3)
$ 0 **a** 3	**a$**	reduce $A \rightarrow \mathbf{a}$ (goto 2)
$ 0 *A* 2	**a$**	shift (goto 5)
$ 0 *A* 2 **a** 5	**$**	reduce $B \rightarrow \mathbf{a}$ (goto 4)
$ $\underline{0}$ $\underline{A}$ 2 $\underline{B}$ 4	**$**	reduce $C \rightarrow AB$ (goto 1)
$ 0 *C* 1	**$**	accept ($S \rightarrow C$)

State I_0:
$S \rightarrow \bullet C$
$C \rightarrow \bullet A\ B$
$A \rightarrow \bullet \mathbf{a}$

goto(I_0,C)

State I_1:
$S \rightarrow C\bullet$

DFA for Shift/Reduce Decisions

Grammar:
$S \to C$
$C \to A\ B$
$A \to \mathbf{a}$
$B \to \mathbf{a}$

The states of the DFA are used to determine if a handle is on top of the stack

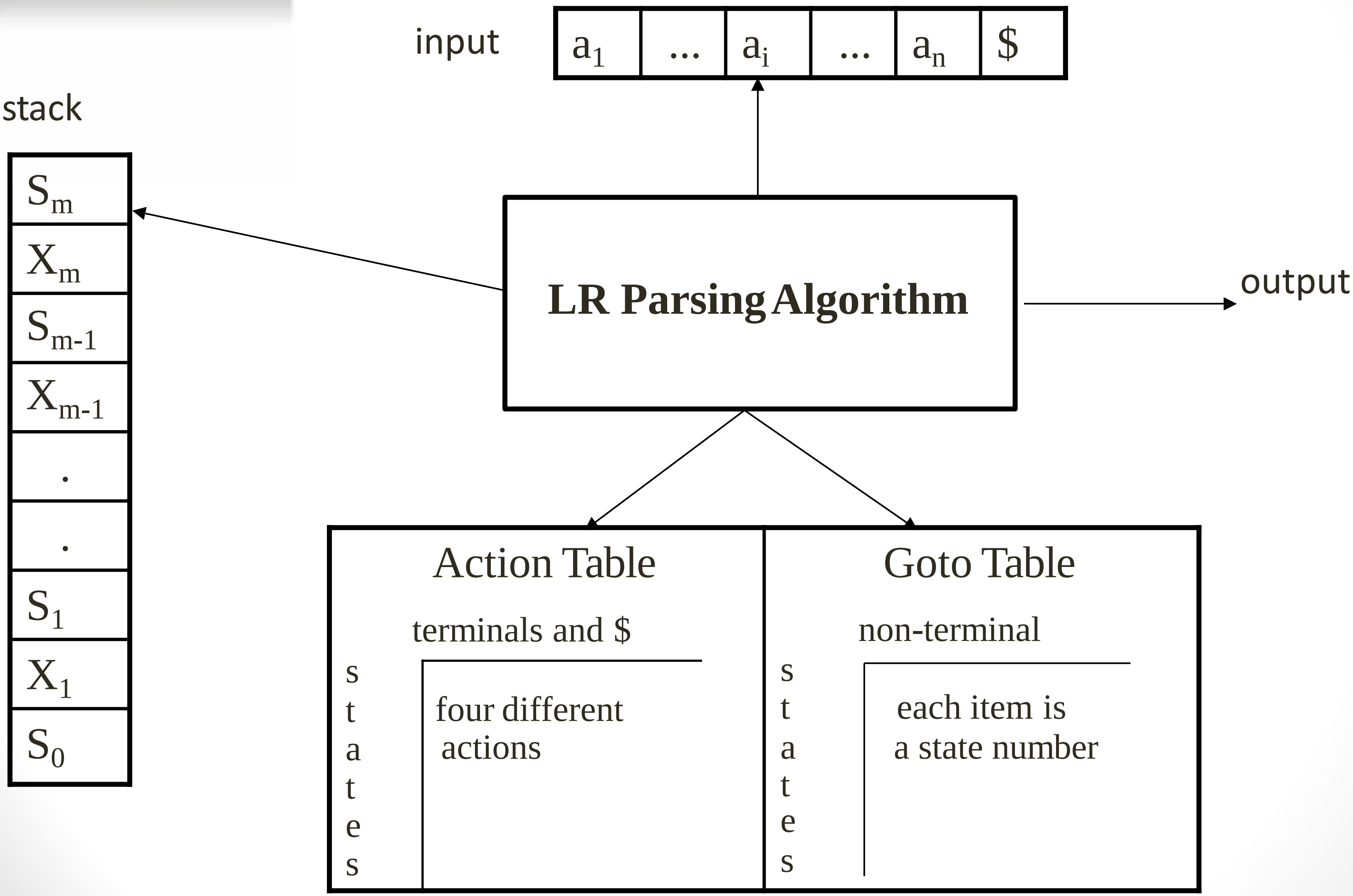
Stack	Input	Action
\$ 0	**aa\$**	start in state 0
\$ 0	**aa\$**	shift (and goto state 3)
\$ 0 **a** 3	**a\$**	reduce $A \to \mathbf{a}$ (goto 2)
\$ 0 A 2	**a\$**	shift (goto 5)
\$ 0 A 2 **a** 5	**\$**	reduce $B \to \mathbf{a}$ (goto 4)
\$ 0 A 2 B 4	**\$**	reduce $C \to AB$ (goto 1)
\$ 0 C <u>1</u>	<u>**\$**</u>	accept ($S \to C$)

State I_0:
$S \to \bullet C$
$C \to \bullet A\ B$
$A \to \bullet \mathbf{a}$

$\xrightarrow{goto(I_0,C)}$

State I_1:
$S \to C \bullet$

Model of an LR Parser



A Configuration of LR Parsing Algorithm

- A configuration of a LR parsing is:

$$(S_o X_1 S_1 \ldots X_m S_m, a_i a_{i+1} \ldots a_n \$)$$

Stack — Rest of Input

- S_m and a_i decides the parser action by consulting the parsing action table. (*Initial Stack* contains just S_o)

- A configuration of a LR parsing represents the right sentential form:

$X_1 \ldots X_m a_i a_{i+1} \ldots a_n \$$

Actions of A LR-Parser

1. **shift s** -- shifts the next input symbol and the state **s** onto the stack

   (S_o X_1 S_1 ... X_m S_m, a_i a_{i+1} ... a_n \$) ➔ (S_o X_1 S_1 ... X_m S_m a_i s, a_{i+1} ... a_n \$)

2. **reduce A→β** (or **`rn`** where n is a production number)
   - pop 2|β| (=r) items from the stack;
   - then push **A** and **s** where **s=goto[s_{m-r},A]**

   (S_o X_1 S_1 ... X_m S_m, a_i a_{i+1} ... a_n \$) ➔ (S_o X_1 S_1 ... X_{m-r} S_{m-r} A s, a_i ... a_n \$)

   - Output is the reducing production reduce A→β

2. **Accept** – Parsing successfully completed

3. **Error** -- Parser detected an error (an empty entry in the action table)

Reduce Action

- pop $2|\beta|$ (=r) items from the stack; let us assume that $\beta = Y_1Y_2 \ldots Y_r$
- then push **A** and **s** where **$s = goto[s_{m-r}, A]$**

$$(S_o X_1 S_1 \ldots X_{m-r} S_{m-r} Y_1 S_{m-r+1} \ldots Y_r S_m, a_i a_{i+1} \ldots a_n \$)$$

$$\rightarrow (S_o X_1 S_1 \ldots X_{m-r} S_{m-r} A\ s, a_i \ldots a_n \$)$$

- In fact, $Y_1Y_2 \ldots Y_r$ is a handle.

$$X_1 \ldots X_{m-r} A\ a_i \ldots a_n \$ \Rightarrow X_1 \ldots X_m Y_1 \ldots Y_r a_i a_{i+1} \ldots a_n \$$$

(SLR) Parsing Tables for Expression Grammar

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

Action Table | Goto Table

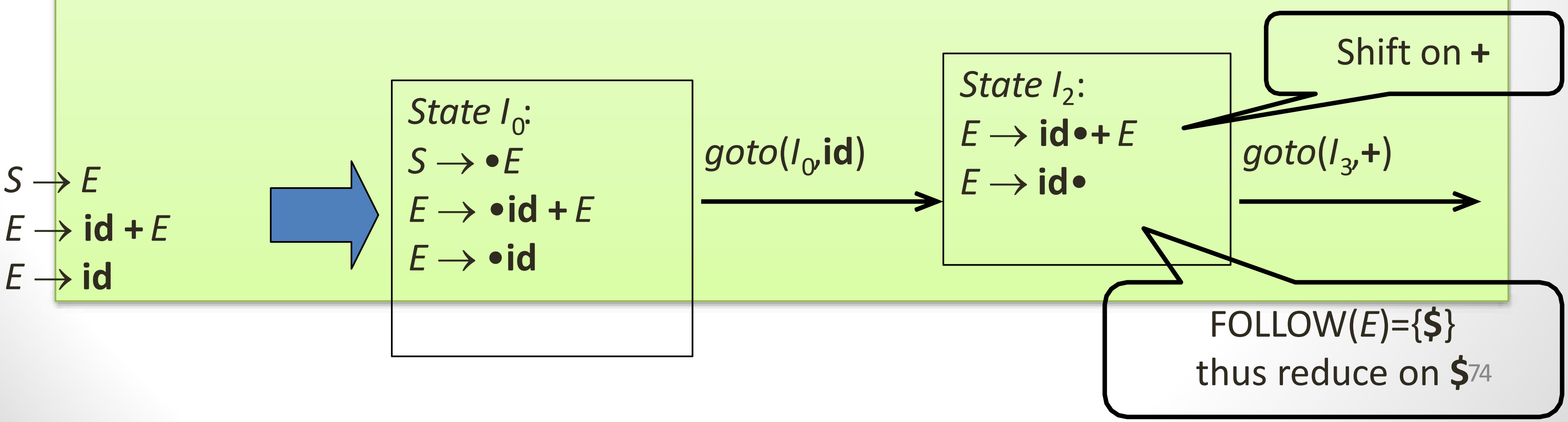
state	id	+	*	(	)	$		E	T	F
0	s5			s4				1	2	3
1		s6				acc				
2		r2	s7		r2	r2				
3		r4	r4		r4	r4				
4	s5			s4				8	2	3
5		r6	r6		r6	r6				
6	s5			s4					9	3
7	s5			s4						10
8		s6			s11					
9		r1	s7		r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

Actions of A (S)LR-Parser -- Example

stack	input	action	output
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*FT→T*F	
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3$		reduce by T→F	T→F
0E1+6T9$		reduce by E→E+T	E→E+T
0E1	$	accept	

SLR Grammars

- SLR (Simple LR): a simple extension of LR(0) shift-reduce parsing
- SLR eliminates some conflicts by populating the parsing table with reductions $A \rightarrow \alpha$ on symbols in FOLLOW(A)



SLR Parsing Table

- Reductions do not fill entire rows
- Otherwise the same as LR(0)

1. $S \to E$
2. $E \to$ **id** + E
3. $E \to$ **id**

	id	+	**$**	E
0	s2			1
1			acc	
2		s3	r3	
3	s2			4
4			r2	

Shift on **+**

FOLLOW(E)={**$**}
thus reduce on **$**

SLR Parsing

- An LR(0) state is a set of LR(0) items
- An LR(0) item is a production with a • (dot) in the right-hand side
- Build the LR(0) DFA by
  - *Closure operation* to construct LR(0) items
  - *Goto operation* to determine transitions
- Construct the SLR parsing table from the DFA
- LR parser program uses the SLR parsing table to determine shift/reduce operations

LR(0) Items of a Grammar

- An *LR*(0) *item* of a grammar G is a production of G with a • at some position of the right-hand side
- Thus, a production

  $$A \to X\,Y\,Z$$

  has four items:

  $$[A \to \bullet\, X\,Y\,Z]$$
  $$[A \to X \bullet Y\,Z]$$
  $$[A \to X\,Y \bullet Z]$$
  $$[A \to X\,Y\,Z\,\bullet]$$

- Note that production $A \to \varepsilon$ has one item $[A \to \bullet]$

Constructing the set of LR(0) Items of a Grammar

1. The grammar is augmented with a new start symbol S' and production $S' \to S$
2. Initially, set $C = closure(\{[S' \to \bullet S]\})$ (this is the start state of the DFA)
3. For each set of items $I \in C$ and each grammar symbol $X \in (N \cup T)$ such that $goto(I,X) \notin C$ and $goto(I,X) \neq \varnothing$, add the set of items $goto(I,X)$ to C
4. Repeat 3 until no more sets can be added to C

The Closure Operation for LR(0) Items

1. Initially, every LR(0) item in I is added to closure(I)
2. If $[A \rightarrow \alpha \bullet B\beta] \in \mathit{closure}(I)$ then for each production $B \rightarrow \gamma$ in the grammar, add the item $[B \rightarrow \bullet\gamma]$ to I if not already in I
3. Repeat 2 until no new items can be added

The Closure Operation (Example)

closure$(\{[E' \rightarrow \bullet E]\})$ =

$\{ [E' \rightarrow \bullet\ E] \}$

$\{ [E' \rightarrow \bullet\ E]$
$[E \rightarrow \bullet\ E + T]$
$[E \rightarrow \bullet\ T] \}$

Add $[E \rightarrow \bullet \gamma]$

$\{ [E' \rightarrow \bullet\ E]$
$[E \rightarrow \bullet\ E + T]$
$[E \rightarrow \bullet\ T]$
$[T \rightarrow \bullet\ T * F]$
$[T \rightarrow \bullet\ F] \}$

Add $[T \rightarrow \bullet \gamma]$

$\{ [E' \rightarrow \bullet\ E]$
$[E \rightarrow \bullet\ E + T]$
$[E \rightarrow \bullet\ T]$
$[T \rightarrow \bullet\ T * F]$
$[T \rightarrow \bullet\ F]$
$[F \rightarrow \bullet\ \mathbf{(}\ E\ \mathbf{)}]$
$[F \rightarrow \bullet\ \mathbf{id}] \}$

Add $[F \rightarrow \bullet \gamma]$

Grammar:
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow \mathbf{(}\ E\ \mathbf{)}$
$F \rightarrow \mathbf{id}$

The Goto Operation for LR(0) Items

1. For each item $[A \to \alpha \bullet X\beta] \in I$, add the set of items *closure*($\{[A \to \alpha X \bullet \beta]\}$) to *goto*($I$,$X$) if not already there
2. Repeat step 1 until no more items can be added to *goto*(I,X)
3. Intuitively, *goto*(I,X) is the set of items that are valid for the viable prefix γX when I is the set of items that are valid for γ

The Goto Operation (Example 1)

Suppose I =

$\{ [E' \to \bullet\ E]$
$[E \to \bullet\ E\ \mathbf{+}\ T]$
$[E \to \bullet\ T]$
$[T \to \bullet\ T\ \mathbf{*}\ F]$
$[T \to \bullet\ F]$
$[F \to \bullet\ \mathbf{(}\ E\ \mathbf{)}]$
$[F \to \bullet\ \mathbf{id}] \}$

Then $goto(I,E)$
$= closure(\{[E' \to E\ \bullet,\ E \to E\ \bullet\ \mathbf{+}\ T]\})$
=

$\{ [E' \to E\ \bullet]$
$[E \to E\ \bullet\ \mathbf{+}\ T] \}$

Grammar:
$E \to E\ \mathbf{+}\ T \mid T$
$T \to T\ \mathbf{*}\ F \mid F$
$F \to \mathbf{(}\ E\ \mathbf{)}$
$F \to \mathbf{id}$

The Goto Operation (Example 2)

Suppose I = { [$E' \rightarrow E\,\bullet$], [$E \rightarrow E \bullet \mathbf{+}\, T$] }

Then $goto(I,\mathbf{+})$ = $closure$({[$E \rightarrow E \,\mathbf{+} \bullet T$]}) =

{ [$E \rightarrow E \,\mathbf{+} \bullet T$]
[$T \rightarrow \bullet\, T \,\mathbf{*}\, F$]
[$T \rightarrow \bullet\, F$]
[$F \rightarrow \bullet\, \mathbf{(}\, E\, \mathbf{)}$]
[$F \rightarrow \bullet\, \mathbf{id}$] }

Grammar:
$E \rightarrow E \,\mathbf{+}\, T \mid T$
$T \rightarrow T \,\mathbf{*}\, F \mid F$
$F \rightarrow \mathbf{(}\, E\, \mathbf{)}$
$F \rightarrow \mathbf{id}$

Constructing SLR Parsing Tables

1. Augment the grammar with $S' \to S$
2. Construct the set $C=\{I_0,I_1,\ldots,I_n\}$ of $LR(0)$ *items*
3. If $[A\to\alpha\bullet a\beta] \in I_i$ and $goto(I_i,a)=I_j$ then set $action[i,a]=\text{shift } j$
4. If $[A\to\alpha\bullet] \in I_i$ then set $action[i,a]=\text{reduce } A\to\alpha$ for all $a \in \text{FOLLOW}(A)$ (apply only if $A\neq S'$)
5. If $[S'\to S\bullet]$ is in I_i then set $action[i,\$]=\text{accept}$
6. If $goto(I_i,A)=I_j$ then set $goto[i,A]=j$
7. Repeat 3-6 until no more entries added
8. The initial state i is the I_i holding item $[S'\to\bullet S]$

The Canonical LR(0) Collection -- Example

I_0: E' → .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_1: E' → E.
E → E.+T

I_2: E → T.
T → T.*F

I_3: T → F.

I_4: F → (.E)
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_6: E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id

I_7: T → T*.F
F → .(E)
F → .id

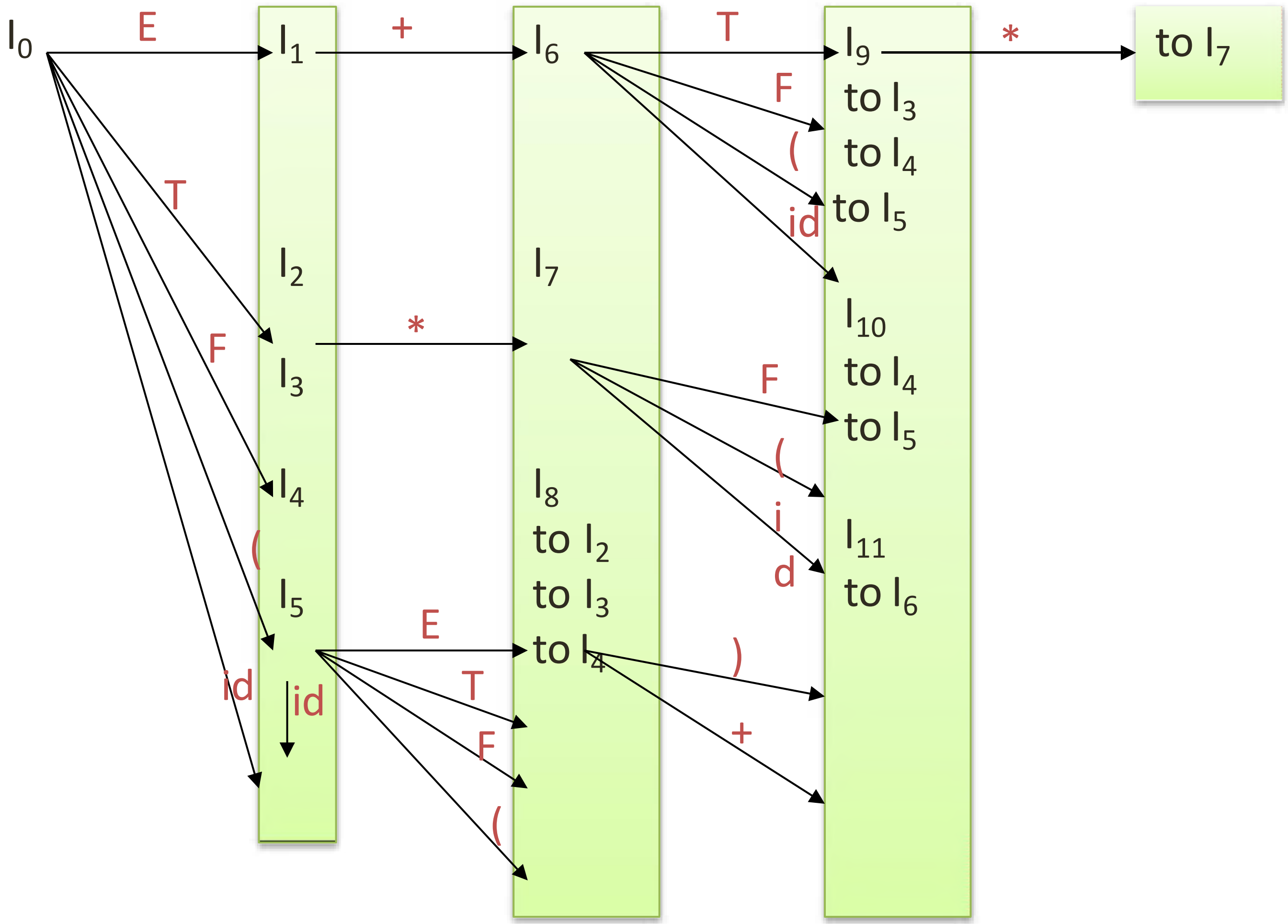
I_8: F → (E.)
E → E.+T

I_9: E → E+T.
T → T.*F

I_{10}: T → T*F.

I_{11}: F → (E).

Transition Diagram (DFA) of Goto Function



Example SLR Grammar and LR(0) Items

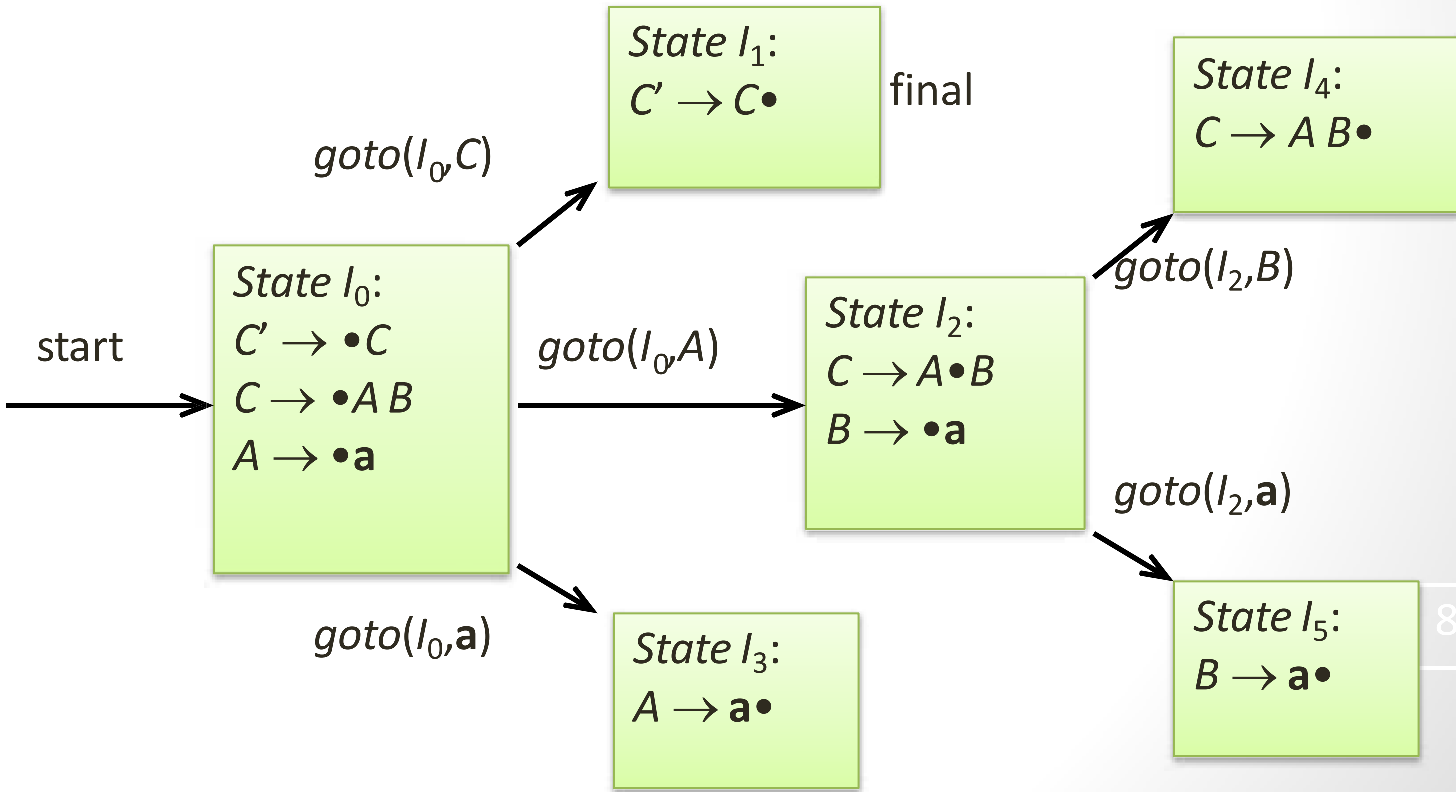
Augmented grammar:

1. $C' \to C$
2. $C \to A\ B$
3. $A \to \mathbf{a}$
4. $B \to \mathbf{a}$

$I_0 = closure(\{[C' \to \bullet C]\})$

$I_1 = \text{goto}(I_0, C) = closure(\{[C' \to C\bullet]\})$

…



Example SLR Parsing Table

State I_0:
$C' \to \bullet C$
$C \to \bullet A\, B$
$A \to \bullet\mathbf{a}$

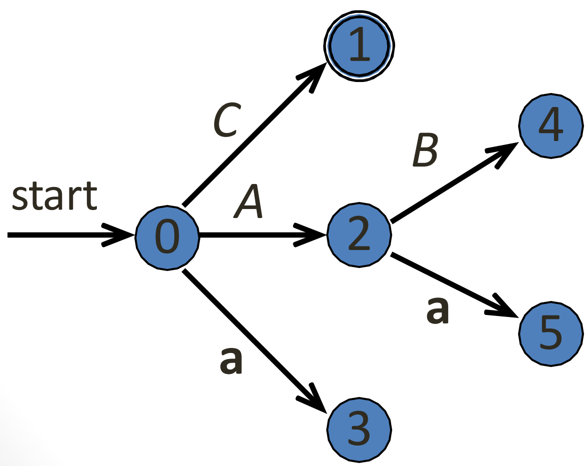
State I_1:
$C' \to C\bullet$

State I_2:
$C \to A\bullet B$
$B \to \bullet\mathbf{a}$

State I_3:
$A \to \mathbf{a}\bullet$

State I_4:
$C \to A\, B\bullet$

State I_5:
$B \to \mathbf{a}\bullet$



	a	**$**	*C*	*A*	*B*
0	s3		1	2	
1		acc			
2	s5				4
3	r3				
4		r2			
5		r4			

Grammar:
1. $C' \to C$
2. $C \to A\, B$
3. $A \to \mathbf{a}$
4. $B \to \mathbf{a}$

SLR and Ambiguity

- Every SLR grammar is unambiguous, but **not** every unambiguous grammar is SLR
- Consider for example the unambiguous grammar

$$S \rightarrow L = R \mid R$$
$$L \rightarrow * R \mid \mathbf{id}$$
$$R \rightarrow L$$

I_0:
$S' \rightarrow \bullet S$
$S \rightarrow \bullet L=R$
$S \rightarrow \bullet R$
$L \rightarrow \bullet * R$
$L \rightarrow \bullet \mathbf{id}$
$R \rightarrow \bullet L$

I_1:
$S' \rightarrow S \bullet$

I_2:
$S \rightarrow L \bullet = R$
$R \rightarrow L \bullet$

I_3:
$S \rightarrow R \bullet$

I_4:
$L \rightarrow * \bullet R$
$R \rightarrow \bullet L$
$L \rightarrow \bullet * R$
$L \rightarrow \bullet \mathbf{id}$

I_5:
$L \rightarrow \mathbf{id} \bullet$

I_6:
$S \rightarrow L = \bullet R$
$R \rightarrow \bullet L$
$L \rightarrow \bullet * R$
$L \rightarrow \bullet \mathbf{id}$

I_7:
$L \rightarrow * R \bullet$

I_8:
$R \rightarrow L \bullet$

I_9:
$S \rightarrow L=R \bullet$

action[2,=]=s6
action[2,=]=r5

no

Has no SLR parsing table

LR(1) Grammars

- SLR too simple
- LR(1) parsing uses lookahead to avoid unnecessary conflicts in parsing table
- LR(1) item = LR(0) item + lookahead

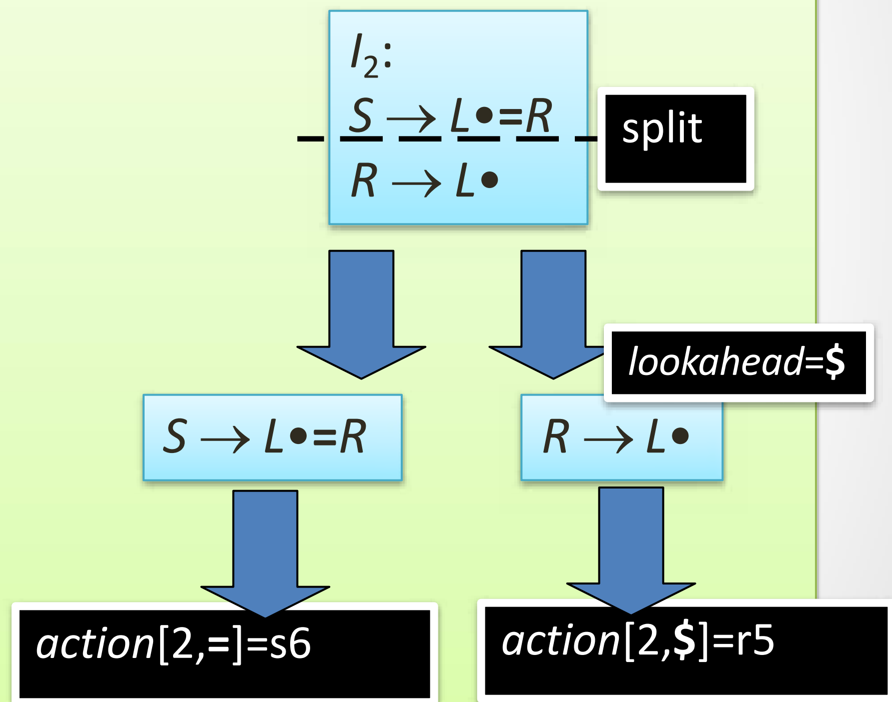
LR(0) item:
$[A \to \alpha \bullet \beta]$

LR(1) item:
$[A \to \alpha \bullet \beta, a]$

SLR Versus LR(1)

- Split the SLR states by adding LR(1) lookahead
- Unambiguous grammar

  1. $S \rightarrow L = R$
  2. $\quad\quad | \; R$
  3. $L \rightarrow * \; R$
  4. $\quad\quad | \; \textbf{id}$
  5. $R \rightarrow L$



Should not reduce on =, because no right-sentential form begins with *R*=

LR(1) Items

- An *LR*(1) *item*

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

  contains a *lookahead* terminal *a*, meaning α already on top of the stack, expect to see βa
- For items of the form

  $[A \rightarrow \alpha \bullet, a]$

  the lookahead *a* is used to reduce $A \rightarrow \alpha$ only if the next input is *a*
- For items of the form

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

  with $\beta \neq \varepsilon$ the lookahead has no effect

The Closure Operation for LR(1) Items

1. Start with *closure*(I) = I
2. If $[A \to \alpha \bullet B\beta, a] \in closure(I)$ then for each production $B \to \gamma$ in the grammar and each terminal $b \in \text{FIRST}(\beta a)$, add the item $[B \to \bullet\gamma, b]$ to I if not already in I
3. Repeat 2 until no new items can be added

The Goto Operation for LR(1) Items

1. For each item $[A \rightarrow \alpha \bullet X \beta, a] \in I$, add the set of items *closure*($\{[A \rightarrow \alpha X \bullet \beta, a]\}$) to *goto*($I$,$X$) if not already there
2. Repeat step 1 until no more items can be added to *goto*(I,X)

Constructing the set of LR(1) Items of a Grammar

1. Augment the grammar with a new start symbol S' and production $S' \to S$
2. Initially, set $C = closure(\{[S' \to \bullet S, \mathbf{\$}]\})$ (this is the start state of the DFA)
3. For each set of items $I \in C$ and each grammar symbol $X \in (N \cup T)$ such that $goto(I,X) \notin C$ and $goto(I,X) \neq \emptyset$, add the set of items $goto(I,X)$ to C
4. Repeat 3 until no more sets can be added to C

Example Grammar and LR(1) Items

- Unambiguous LR(1) grammar:

$$
\begin{aligned}
S &\rightarrow L \mathbf{=} R \\
&\mid R \\
L &\rightarrow \mathbf{*}\, R \\
&\mid \mathbf{id} \\
R &\rightarrow L
\end{aligned}
$$

- Augment with $S' \rightarrow S$
- LR(1) items (next slide)

I_0: $[S' \to \bullet S,\ \$]$ goto(I_0,S)=I_1
$[S \to \bullet L{=}R,\ \$]$ goto(I_0,L)=I_2
$[S \to \bullet R,\ \$]$ goto(I_0,R)=I_3
$[L \to \bullet {*}R,\ {=}/\$]$ goto(I_0,*)=I_4
$[L \to \bullet \mathbf{id},\ {=}/\$]$ goto(I_0,**id**)=I_5
$[R \to \bullet L,\ \$]$ goto(I_0,L)=I_2

I_1: $[S' \to S\bullet,\ \$]$

I_2: $[S \to L\bullet{=}R,\ \$]$ goto(I_0,=)=I_6
$[R \to L\bullet,\ \$]$

I_3: $[S \to R\bullet,\ \$]$

I_4: $[L \to {*}\bullet R,\ {=}/\$]$ goto(I_4,R)=I_7
$[R \to \bullet L,\ {=}/\$]$ goto(I_4,L)=I_8
$[L \to \bullet {*}R,\ {=}/\$]$ goto(I_4,*)=I_4
$[L \to \bullet \mathbf{id},\ {=}/\$]$ goto(I_4,**id**)=I_5

I_5: $[L \to \mathbf{id}\bullet,\ {=}/\$]$

I_6: $[S \to L{=}\bullet R,\ \$]$ goto(I_6,R)=I_9
$[R \to \bullet L,\ \$]$ goto(I_6,L)=I_{10}
$[L \to \bullet {*}R,\ \$]$ goto(I_6,*)=I_{11}
$[L \to \bullet \mathbf{id},\ \$]$ goto(I_6,**id**)=I_{12}

I_7: $[L \to {*}R\bullet,\ {=}/\$]$

I_8: $[R \to L\bullet,\ {=}/\$]$

I_9: $[S \to L{=}R\bullet,\ \$]$

I_{10}: $[R \to L\bullet,\ \$]$

I_{11}: $[L \to {*}\bullet R,\ \$]$ goto(I_{11},R)=I_{13}
$[R \to \bullet L,\ \$]$ goto(I_{11},L)=I_{10}
$[L \to \bullet {*}R,\ \$]$ goto(I_{11},*)=I_{11}
$[L \to \bullet \mathbf{id},\ \$]$ goto(I_{11},**id**)=I_{12}

I_{12}: $[L \to \mathbf{id}\bullet,\ \$]$

I_{13}: $[L \to {*}R\bullet,\ \$]$

Constructing Canonical LR(1) Parsing Tables

1. Augment the grammar with $S' \to S$
2. Construct the set $C=\{I_0,I_1,\dots,I_n\}$ of LR(1) items
3. If $[A \to \alpha \bullet a\beta,\ b] \in I_i$ and $goto(I_i,a)=I_j$ then set $action[i,a]$=shift j
4. If $[A \to \alpha \bullet,\ a] \in I_i$ then set $action[i,a]$=reduce $A \to \alpha$ (apply only if $A \neq S'$)
5. If $[S' \to S\bullet,\ \$]$ is in I_i then set $action[i,\$]$=accept
6. If $goto(I_i,A)=I_j$ then set $goto[i,A]=j$
7. Repeat 3-6 until no more entries added
8. The initial state i is the I_i holding item $[S' \to \bullet S, \$]$

Example LR(1) Parsing Table

Grammar:
1. $S' \rightarrow S$
2. $S \rightarrow L = R$
3. $S \rightarrow R$
4. $L \rightarrow * R$
5. $L \rightarrow$ **id**
6. $R \rightarrow L$

	id	*	=	$	*S*	*L*	*R*
0	s5	s4			1	2	3
1				acc			
2			s6	r6			
3				r3			
4	s5	s4				8	7
5			r5	r5			
6	s12	s11				10	4
7			r4	r4			
8			r6	r6			
9				r2			
10				r6			
11	s12	s11				10	13
12				r5			
13				r4			

LALR(1) Grammars

- LR(1) parsing tables have many states
- LALR(1) parsing (Look-Ahead LR) combines LR(1) states to reduce table size
- Less powerful than LR(1)
  - Will not introduce shift-reduce conflicts, because shifts do not use lookaheads
  - May introduce reduce-reduce conflicts, but seldom do so for grammars of programming languages

Constructing LALR(1) Parsing Tables

1. Construct sets of LR(1) items
2. Combine LR(1) sets with sets of items that share the same first part

I_4:

$[L \rightarrow * \bullet R,$	$=]$
$[R \rightarrow \bullet L,$	$=]$
$[L \rightarrow \bullet * R,$	$=]$
$[L \rightarrow \bullet \mathbf{id},$	$=]$

I_{11}:

$[L \rightarrow * \bullet R,$	$\$]$
$[R \rightarrow \bullet L,$	$\$]$
$[L \rightarrow \bullet * R,$	$\$]$
$[L \rightarrow \bullet \mathbf{id},$	$\$]$

$[L \rightarrow * \bullet R,$	$=/\$]$
$[R \rightarrow \bullet L,$	$=/\$]$
$[L \rightarrow \bullet * R,$	$=/\$]$
$[L \rightarrow \bullet \mathbf{id},$	$=/\$]$

Shorthand
for two items
in the same set

Example LALR(1) Grammar

- Unambiguous LR(1) grammar:

$$
\begin{aligned}
S &\to L\ \mathbf{=}\ R \\
&\mid R \\
L &\to \mathbf{*}\ R \\
&\mid \mathbf{id} \\
R &\to L
\end{aligned}
$$

- Augment with $S' \to S$
- LALR(1) items (next slide)

I_0: $[S' \rightarrow \bullet S, \$]$ $\quad$ $goto(I_0,S)=I_1$

$[S \rightarrow \bullet L=R, \$]$ $\quad$ $goto(I_0,L)=I_2$

$[S \rightarrow \bullet R, \$]$ $\quad$ $goto(I_0,R)=I_3$

$[L \rightarrow \bullet {*}R, =/\$]$ $\quad$ $goto(I_0,*)=I_4$

$[L \rightarrow \bullet \mathbf{id}, =/\$]$ $\quad$ $goto(I_0,\mathbf{id})=I_5$

$[R \rightarrow \bullet L, \$]$ $\quad$ $goto(I_0,L)=I_2$

$goto(I_0,=)= I_6$

I_1: $[S' \rightarrow S\bullet, \$]$

I_2: $[S \rightarrow L\bullet=R, \$]$

$[R \rightarrow L\bullet, \$]$

I_3: $[S \rightarrow R\bullet, \$]$

I_4: $[L \rightarrow {*}\bullet R, =/\$]$ $\quad$ $goto(I_4,R)=I_7$

$[R \rightarrow \bullet L, =/\$]$ $\quad$ $goto(I_4,L)=I_9$

$[L \rightarrow \bullet {*}R, =/\$]$ $\quad$ $goto(I_4,*)=I_4$

$[L \rightarrow \bullet \mathbf{id}, =/\$]$ $\quad$ $goto(I_4,\mathbf{id})=I_5$

I_5: $[L \rightarrow \mathbf{id}\bullet, =/\$]$

I_6: $[S \rightarrow L=\bullet R, \quad \$]$ $\quad$ $goto(I_6,R)=I_8$

$[R \rightarrow \bullet L, \$]$ $\quad$ $goto(I_6,L)=I_9$

$[L \rightarrow \bullet {*}R, \quad \$]$ $\quad$ $goto(I_6,*)=I_4$

$[L \rightarrow \bullet \mathbf{id}, \quad \$]$ $\quad$ $goto(I_6,\mathbf{id})=I_5$

I_7: $[L \rightarrow {*}R\bullet, =/\$]$

I_8: $[S \rightarrow L=R\bullet, \$]$

I_9: $[R \rightarrow L\bullet, =/\$]$

$[R \rightarrow L\bullet, =]$

$[R \rightarrow L\bullet, \$]$

Example LALR(1) Parsing Table

Grammar:
1. $S' \to S$
2. $S \to L = R$
3. $S \to R$
4. $L \to$ ***** R
5. $L \to$ **id**
6. $R \to L$

	id	*****	=	**$**	*S*	*L*	*R*
0	s5	s4			1	2	3
1				acc			
2			s6	r6			
3				r3			
4	s5	s4				9	7
5			r5	r5			
6	s5	s4				9	8
7			r4	r4			
8				r2			
9			r6	r6			

LL, SLR, LR, LALR Summary

- LL parse tables computed using FIRST/FOLLOW
  - Nonterminals $\times$ terminals $\rightarrow$ productions
  - Computed using FIRST/FOLLOW
- LR parsing tables computed using closure/goto
  - LR states $\times$ terminals $\rightarrow$ shift/reduce actions
  - LR states $\times$ nonterminals $\rightarrow$ goto state transitions
- A grammar is
  - LL(1) if its LL(1) parse table has no conflicts
  - SLR if its SLR parse table has no conflicts
  - LALR(1) if its LALR(1) parse table has no conflicts
  - LR(1) if its LR(1) parse table has no conflicts

Dealing with Ambiguous Grammars

1. $S' \to E$
2. $E \to E$ **+** E
3. $E \to$ **id**

	id	+	$	*E*
0	s2			1
1		s3	acc	
2		r3	r3	
3	s2			4
4		s3/r2	r2	

stack	input
$ 0	**id+id+id$**
…	…
$ 0 *E* 1 **+** 3 *E* 4	**+id$**

Shift/reduce conflict:
action[4,**+**] = shift 4
action[4,**+**] = reduce $E \to E$ **+** E

When shifting on **+**:
yields right associativity
id+(id+id)

When reducing on **+**:
yields left associativity
(id+id)+id

Using Associativity and Precedence to Resolve Conflicts

- Left-associative operators: reduce
- Right-associative operators: shift
- Operator of higher precedence on stack: reduce
- Operator of lower precedence on stack: shift

$S' \rightarrow E$
$E \rightarrow E \textbf{ + } E$
$E \rightarrow E \textbf{ * } E$
$E \rightarrow \textbf{id}$

stack	input	
$ 0	**id*id+id$**	
…	…	
$ 0 E 1 ***** 3 E 5	**+id$**	reduce $E \rightarrow E$ ***** E

Error Detection in LR Parsing

- Canonical LR parser uses full LR(1) parse tables and will never make a single reduction before recognizing the error when a syntax error occurs on the input
- SLR and LALR may still reduce when a syntax error occurs on the input, but will never shift the erroneous input symbol

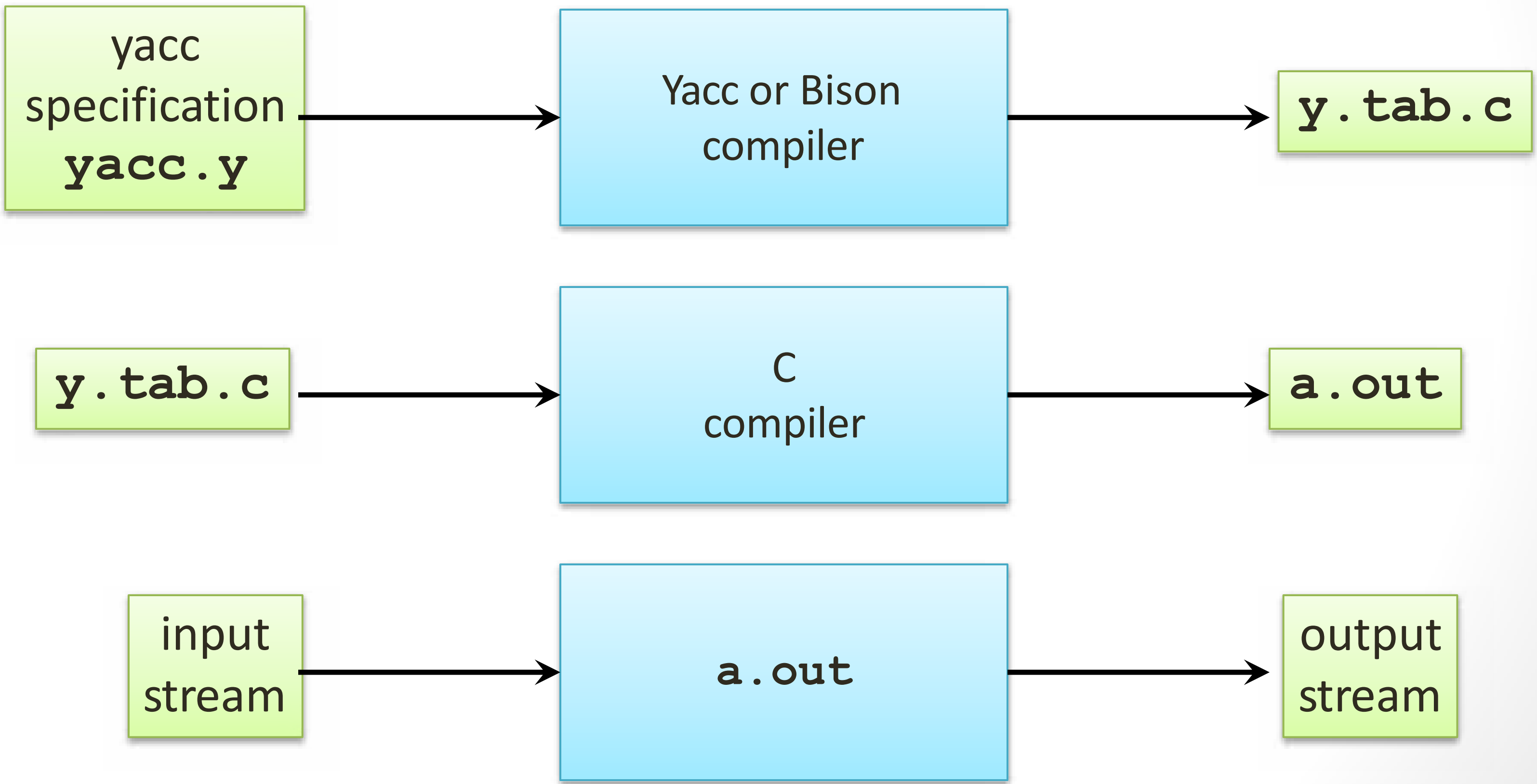
Error Recovery in LR Parsing

- Panic mode
  - Pop until state with a goto on a nonterminal *A* is found, (where *A* represents a major programming construct), push *A*
  - Discard input symbols until one is found in the FOLLOW set of *A*
- Phrase-level recovery
  - Implement error routines for every error entry in table
- Error productions
  - Pop until state has error production, then shift on stack
  - Discard input until symbol is encountered that allows parsing to continue

ANTLR, Yacc, and Bison

- *ANTLR* tool
  - Generates LL(*k*) parsers
- *Yacc* (Yet Another Compiler Compiler)
  - Generates LALR(1) parsers
- *Bison*
  - Improved version of Yacc

Creating an LALR(1) Parser with Yacc/Bison



Yacc Specification

- A *yacc specification* consists of three parts:

  yacc declarations, and C declarations within %{ %}
  %%
  translation rules
  %%
  user-defined auxiliary procedures

- **The *translation rules* are productions with actions:**

  production$_1$ { ***semantic action***$_1$ }
  production$_2$ { ***semantic action***$_2$ }
  …
  production$_n$ { ***semantic action***$_n$ }

Writing a Grammar in Yacc

- Productions in Yacc are of the form

```
Nonterminal : tokens/nonterminals { action }
            | tokens/nonterminals { action }
            …
            ;
```

- Tokens that are single characters can be used directly within productions, e.g. `'+'`
- Named tokens must be declared first in the declaration part using

  `%token` *TokenName*

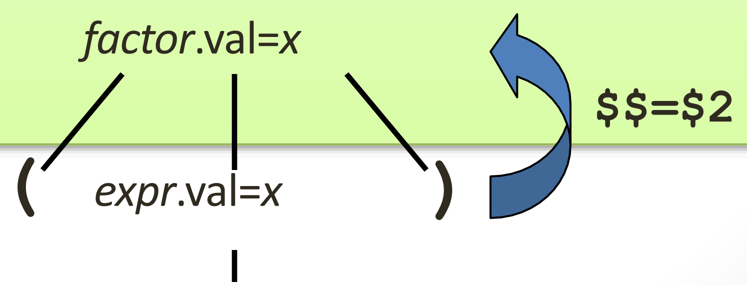
Synthesized Attributes

- Semantic actions may refer to values of the *synthesized attributes* of terminals and nonterminals in a production:

  $X : Y_1\ Y_2\ Y_3 \ldots Y_n \qquad \{\ action\ \}$

  - `$$` refers to the value of the attribute of X
  - `$`*i* refers to the value of the attribute of Y_i
- For example

  ```
  factor : '(' expr ')' { $$=$2; }
  ```



Example 1

```
%{ #include <ctype.h> %}
%token DIGIT
%%
line     : expr '\n'               { printf("%d\n", $1); }
         ;
expr     : expr '+' term           { $$ = $1 + $3; }
         | term                    { $$ = $1; }
         ;
term     : term '*' factor         { $$ = $1 * $3; }
         | factor                  { $$ = $1; }
         ;
factor   : '(' expr ')'            { $$ = $2; }
         | DIGIT                   { $$ = $1; }
         ;
%%
int yylex()
{ int c = getchar();
  if (isdigit(c))
  { yylval = c-'0';
    return DIGIT;
  }
  return c;
}
```

Also results in definition of
`#define DIGIT xxx`

Attribute of **`term`** (parent)

Attribute of **`factor`** (child)

Attribute of token (stored in **`yylval`**)

Example of a very crude lexical analyzer invoked by the parser

Dealing With Ambiguous Grammars

Example 2

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
lines   : lines expr '\n'         { printf("%g\n", $2); }
        | lines '\n'
        | /* empty */
        ;
expr    : expr '+' expr           { $$ = $1 + $3; }
        | expr '-' expr           { $$ = $1 - $3; }
        | expr '*' expr           { $$ = $1 * $3; }
        | expr '/' expr           { $$ = $1 / $3; }
        | '(' expr ')'            { $$ = $2; }
        | '-' expr %prec UMINUS { $$ = -$2; }
        | NUMBER
        ;
%%
```

Double type for attributes and **yylval**

Example 2 (cont'd)

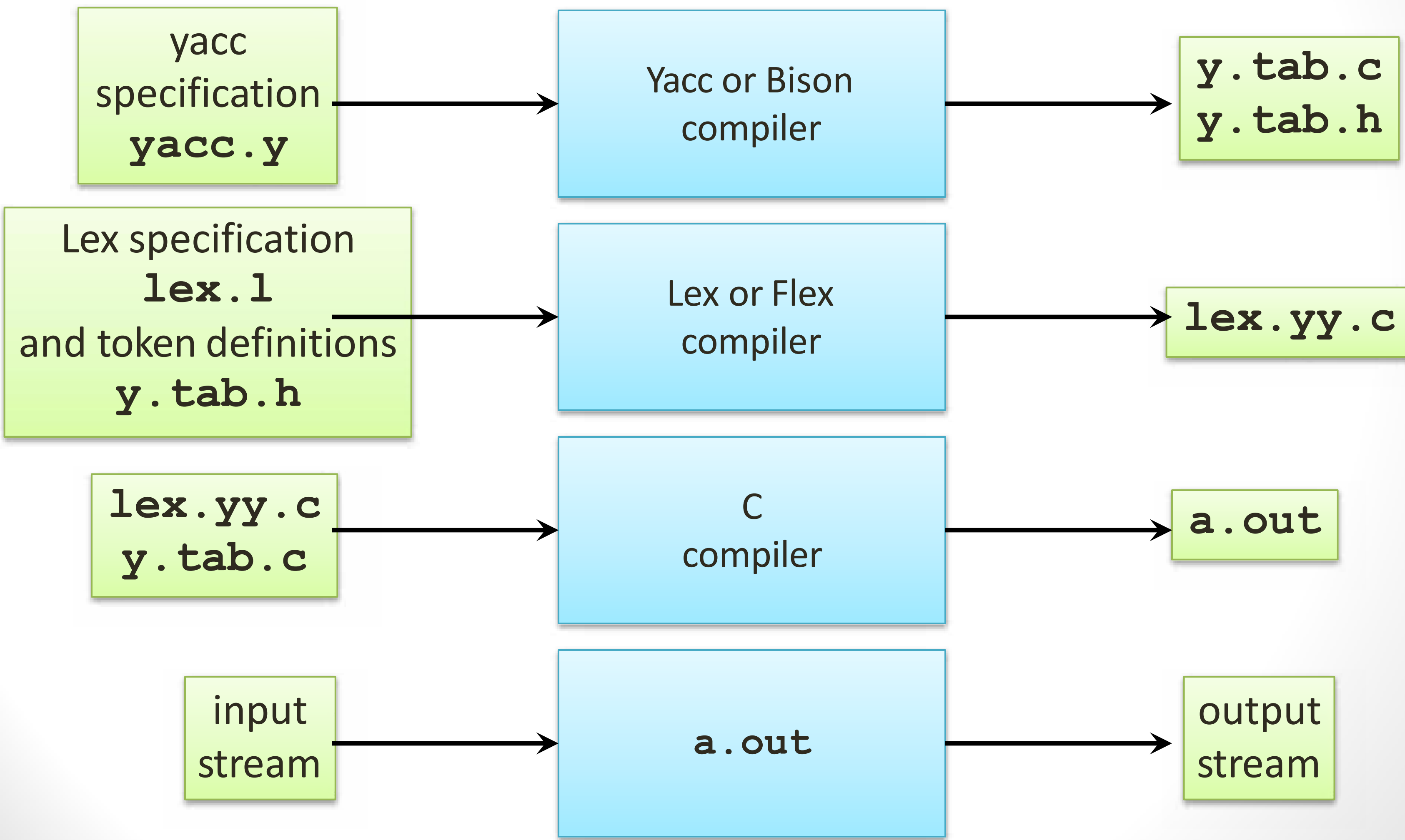
```
%%
int yylex()
{ int c;
  while ((c = getchar()) == ' ')
    ;
  if ((c == '.') || isdigit(c))
  { ungetc(c, stdin);
    scanf("%lf", &yylval);
    return NUMBER;
  }
  return c;
}
int main()
{ if (yyparse() != 0)
    fprintf(stderr, "Abnormal exit\n");
  return 0;
}
int yyerror(char *s)
{ fprintf(stderr, "Error: %s\n", s);
}
```

Crude lexical analyzer for fp doubles and arithmetic operators

Run the parser

Invoked by parser to report parse errors

Combining Lex/Flex with Yacc/Bison



Lex Specification for Example 2

```
%option noyywrap
%{
#include "y.tab.h"

extern double yylval;
%}
number  [0-9]+\.?|[0-9]*\.[0-9]+
%%
[ ]             { /* skip blanks */ }
{number}        { sscanf(yytext, "%lf", &yylval);
                  return NUMBER;
                }
\n|.            { return yytext[0]; }
```

Generated by Yacc, contains **#define NUMBER xxx**

Defined in **y.tab.c**

```
yacc-d  example2.y
lex example2.l
gcc y.tab.c lex.yy.c
./a.out
```

```
bison-d  -y example2.y
flex example2.l
gcc y.tab.c lex.yy.c
./a.out
```

Error Recovery in Yacc

```
%{
…
%}
…
%%
lines   : lines expr '\n'       { printf("%g\n", $2; }
        | lines '\n'
        | /* empty */
        | error '\n'            { yyerror("reenter last line: ");
                                  yyerrok;
                                }
        ;
…
```

Error production: set error mode and skip input until newline

Reset parser to normal mode

Semantic Analysis

The Compiler So Far

- Lexical analysis
  - Detects inputs with illegal tokens
- Parsing
  - Detects inputs with ill-formed parse trees
- Semantic analysis
  - Last "front end" phase
  - Catches all remaining errors

What's Wrong?

- Example 1

```
int y = x + 3;
```

- Example 2

```
String y = "abc" ;
 y ++ ;
```

Why a Separate Semantic Analysis?

- Parsing cannot catch some errors
- Some language constructs are not context-free
  - Example: All used variables must have been declared (i.e. scoping)
  - ex: { int x { .. { .. x ..} ..} ..}
  - Example: A method must be invoked with arguments of proper type (i.e. typing)
  - ex: int f(int, int) {…} called by f('a', 2.3, 1)

More problems require semantic analysis

1. Is x a scalar, an array, or a function?
2. Is x declared before it is used?
3. Is x defined before it is used?
4. Are any names declared but not used?
5. Which declaration of x does this reference?
6. Is an expression *type-consistent?*
7. Does the dimension of a reference match the declaration?
8. Where can x be stored? *(heap, stack, . . .)*
9. Does *p reference the result of a malloc()?
10. Is an array reference *in bounds?*
11. Does function foo produce a constant value?

Why is semantic analysis hard?

- need non-local information
- answers depend on values, not on syntax
- answers may involve computation

How can we answer these questions?

1. use context-sensitive grammars (CSG)
   – general problem is P-space complete
2. use attribute grammars(AG)
   – augment context-free grammar with rules
   – calculate attributes for grammar symbols
3. use *ad hoc* techniques
   – augment grammar with arbitrary code
   – execute code at corresponding reduction
   – store information in attributes, symbol tables

Types

- What is a type?
  - The notion varies from language to language
- Consensus
  - A set of values
  - A set of operations on those values
- Classes are one instantiation of the modern notion of type

Why Do We Need Type Systems?

Consider the assembly language fragment

addi r1, r2, r3

What are the types of r1, r2, r3?

Types and Operations

- Certain operations are legal for values of each type
  - It doesn't make sense to add a function pointer and an integer in C
  - It does make sense to add two integers
  - But both have the same assembly language implementation!

Type Systems

- A language's type system specifies which operations are valid for which types
- The goal of type checking is to ensure that operations are used with the correct types
  - Enforces intended interpretation of values, because nothing else will!
- Type systems provide a concise formalization of the semantic checking rules

What Can Types do For Us?

- Can detect certain kinds of errors :
  - “abc” ++ ; x = ar[“abc”] ; int x = “abc” ;
- Memory errors:
  - Reading from an invalid pointer, etc.
  - int x[50] ; x[50] = 3;
- expressiveness (overloading, polymorphism)
  - help determine which methods/constructors would be invoked.
  - Ex: add(Complex, Complex), add(int,int), add(String,String),..
  - add(23,14) => add(int, int) invoked
- provide information for code generation
  - ex: memory size

Type Checking Overview

Three kinds of languages:

Statically typed: All or almost all checking of types is done as part of compilation (C, Java, Cool)

Dynamically typed: Almost all checking of types is done as part of program execution (Scheme)

Untyped: No type checking (machine code)

Pros and cons

Static typing:

- catches many programming errors at compile time
- Avoids overhead of runtime type checks

Dynamic typing:

- Static type systems are restrictive
- Rapid prototyping easier in a dynamic type system

Type checking

- **Type checker**
  - enforces rules of type system
  - may be strong/weak, static/dynamic
- **Static type checking**
  - performed at compile time
  - early detection, no run-time overhead
  - not always possible (e.g., A[i])
- **Dynamic type checking**
  - performed at run time
  - more flexible, rapid prototyping
  - overhead to check run-time type tags

Translation scheme for declarations

- P → D ; E
- D → D ; D
- D → **id** : T { addtype(**id**.entry, T.type) }
- T → **char** { T.type := char }
- T → **integer** { T.type := integer }
- T → ↑T_1 { T.type := pointer(T_1.type) }
- T → **array** [**num**] **of** T_1
- { T.type := array(1 .. **num**.val, T_1.type) }

Try to derive the annotated parse tree for the declaration X: array[100] of ↑ char

Type checking for expressions

Once the identifiers and their types have been inserted into the symbol table, we can check the type of the elements of an expression:

- E → **literal** { E.type := char }
- E → **num** { E.type := integer }
- E → **id** { E.type := lookup(**id**.entry) }
- E → E_1 **mod** E_2 { **if** E_1.type =integer **and** E_2.type = integer
- **then** E.type := integer
- **else** E.type := type_error }
- E → E_1 [E_2] { **if** E_2.type = integer **and** E_1.type = array(s,t)
- **then** E.type := t **else** E.type := type_error }
- E → E_1↑ { **if** E_1.type = pointer(t)
- **then** E.type := t **else** E.type := type-error }

How about boolean types?

- Try adding
- T -> boolean
- Relational operators < <= = >= > <>
- Logical connectives **and or not**

- to the grammar, then add appropriate type checking semantic actions.

Type checking for statements

- Usually we assign the type VOID to statements.
- If a type error is found during type checking, though, we should set the type to type_error

- Let's change our grammar allow statements:
- $P \rightarrow D \; ; \; S$

- i.e., a program is a sequence of declarations followed by a sequence of statements.

Type checking for statements

Now we need to add productions and semantic actions:

- $S \to$ **id** := E { **if id**.type = E.type **then** S.type := void
- **else** S.type := type_error }
- $S \to$ **if** E **then** S_1 { **if** E.type = boolean
- **then** S.type := S_1.type
- **else** S.type := type_error }
- $S \to$ **while** E **do** S_1 { **if** E.type = boolean
- **then** S.type := S_1.type
- **else** S.type := type_error }
- $S \to S_1 ; S_2$ { **if** S_1.type = void **and** S_2.type = void
- **then** S.type := void
- **else** S.type := type_error.

Type checking for function calls

- Suppose we add a production E → E (E)
- Then we need productions for function declarations:

T → T1 → T2 { T.type := T1.type → T2.type }

and function calls:

E → E1 (E2) { **if** E2.type = s **and** E1.type = s → t
then E.type := t
else E.type := type_error }

Type checking for function calls

- Multiple-argument functions, however, can be modeled as functions that take a single PRODUCT argument.

$$\text{root} : (\text{real} \to \text{real}) \times \text{real} \to \text{real}$$

- this would model a function that takes a real function over the reals, and a real, and returns a real.
- In C: float root(float (*f)(float), float x);

Type conversion

- Suppose we encounter an expression x+i where x has type float and i has type int.
- CPU instructions for addition could take EITHER float OR int as operands, but not a mix.
- This means the compiler must sometimes convert the operands of arithmetic expressions to ensure that operands are consistent with operators.

•With postfix as an intermediate language for expressions,

we could express the conversion as follows:

x i inttoreal float+

where real+ is the floating point addition operation.

Type coercion

- If type conversion is **done by the compiler** without the programmer requesting it, it is called **IMPLICIT** conversion or type **COERCION.**
- **EXPLICIT** conversions are those that the **programmer specifices**,(**CASTING**) e.g.

  x = (int)y * 2;

- Implicit conversion of CONSTANT expressions should be done at compile time.

Type checking example with coercion

Production	Semantic Rule
E -> num	E.type := integer
E -> num . num	E.type := real
E -> id	E.type := lookup(id.entry)
E -> E_1 op E_2	E.type := if E_1.type == integer and E_2.type == integer then integer else if E_1.type == integer and E_2.type == real then real else if E_1.type == real and E_2.type == integer then real else if E_1.type == real and E_2.type == real then real else type_error

- ***THANK YOU !!!!!!***

My Blog : anandgharu.wordpress.com