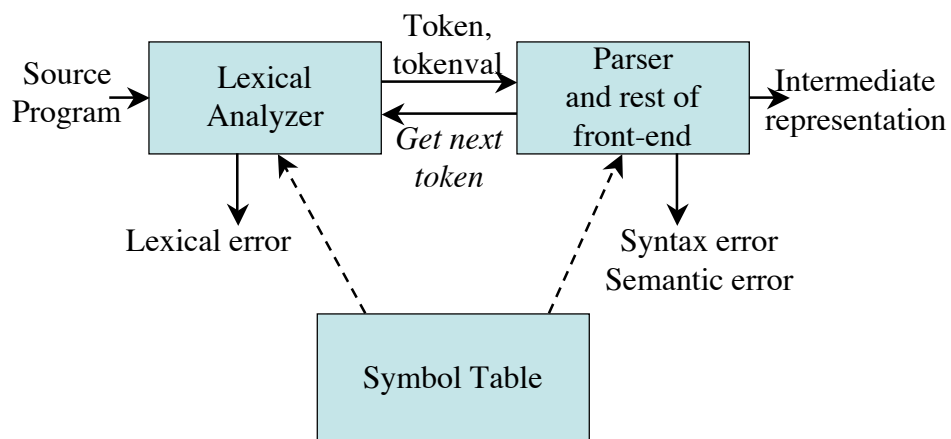


Syntax Analysis Part I

Chapter 4

COP5621 Compiler Construction
Copyright Robert van Engelen, Florida State University, 2005

Position of a Parser in the Compiler Model



The Parser

- The task of the parser is to check syntax
- The syntax-directed translation stage in the compiler's front-end checks static semantics and produces an intermediate representation (IR) of the source program
 - Abstract syntax trees (ASTs)
 - Control-flow graphs (CFGs) with triples, three-address code, or register transfer lists
 - 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 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

Prefix { ...
 ↓ Error is detected here
 for (;
 ...
 ...

Prefix { ...
 ↓ Error is detected here
 DO 10 I = 1;0
 ...
 ...

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 $\alpha \rightarrow \beta$
 where $\alpha \in (NUT)^* N (NUT)^*$
 and $\beta \in (NUT)^*$
 - S is a designated *start symbol* $S \in N$

Notational Conventions Used

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

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

Derivation (Example)

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow (E)$$

$$E \rightarrow - E$$

$$E \rightarrow \mathbf{id}$$

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

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

$$E \Rightarrow^* E$$

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

Chomsky Hierarchy: Language Classification

- A grammar G is said to be
 - *Regular* if it is *right linear* where each production is of the form

$$A \rightarrow wB \quad \text{or} \quad A \rightarrow w$$
 or *left linear* where each production is of the form

$$A \rightarrow Bw \quad \text{or} \quad A \rightarrow w$$
 - *Context free* if each production is of the form

$$A \rightarrow \alpha$$
 where $A \in N$ and $\alpha \in (NUT)^*$
 - *Context sensitive* if each production is of the form

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$
 where $A \in N$, $\alpha, \gamma, \beta \in (NUT)^*$, $|\gamma| > 0$
 - *Unrestricted*

Chomsky Hierarchy

$$\mathcal{L}(\text{regular}) \subseteq \mathcal{L}(\text{context free}) \subseteq \mathcal{L}(\text{context sensitive}) \subseteq \mathcal{L}(\text{unrestricted})$$

Where $\mathcal{L}(T) = \{ L(G) \mid G \text{ is of type } T \}$

That is, the set of all languages generated by grammars G of type T

Examples:

Every *finite language* is regular

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

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

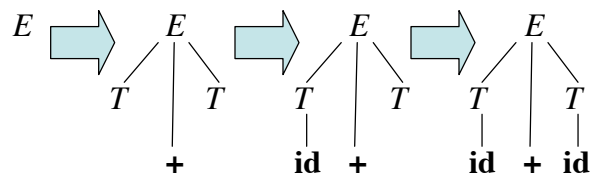
Parsing

- *Universal* (any C-F grammar)
 - Cocke-Younger-Kasimi
 - Earley
- *Top-down* (C-F grammar with restrictions)
 - Recursive descent (predictive parsing)
 - LL (Left-to-right, Leftmost derivation) methods
- *Bottom-up* (C-F grammar with restrictions)
 - Operator precedence parsing
 - LR (Left-to-right, Rightmost derivation) methods
 - SLR, canonical LR, LALR

Top-Down Parsing

- LL methods (Left-to-right, Leftmost derivation) and recursive-descent parsing

Grammar:	Leftmost derivation:
$E \rightarrow T + T$	$E \Rightarrow_{lm} T + T$
$T \rightarrow (E)$	$\Rightarrow_{lm} \mathbf{id} + T$
$T \rightarrow - E$	$\Rightarrow_{lm} \mathbf{id} + \mathbf{id}$
$T \rightarrow \mathbf{id}$	



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 may loop forever

General Left Recursion Elimination

Arrange the nonterminals in some order A_1, A_2, \dots, A_n

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

for $j = 1, \dots, i-1$ **do**

 replace each

$$A_i \rightarrow A_j \gamma$$

 with

$$A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$$

 where

$$A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$$

enddo

 eliminate the immediate left recursion in A_i

enddo

Immediate Left-Recursion Elimination

Rewrite every left-recursive production

$$\begin{array}{l}
 A \rightarrow A \alpha \\
 \quad | \beta \\
 \quad | \gamma \\
 \quad | A \delta
 \end{array}$$

into a right-recursive production:

$$\begin{array}{l}
 A \rightarrow \beta A_R \\
 \quad | \gamma A_R \\
 A_R \rightarrow \alpha A_R \\
 \quad | \delta A_R \\
 \quad | \varepsilon
 \end{array}$$

Example Left Rec. Elimination

$$\left. \begin{array}{l}
 A \rightarrow BC | \mathbf{a} \\
 B \rightarrow CA | \mathbf{A} \mathbf{b} \\
 C \rightarrow AB | CC | \mathbf{a}
 \end{array} \right\} \text{Choose arrangement: } A, B, C$$

$i = 1$: nothing to do

$$\begin{array}{l}
 i = 2, j = 1: B \rightarrow CA | \underline{\mathbf{A}} \mathbf{b} \\
 \Rightarrow B \rightarrow CA | \underline{BC} \mathbf{b} | \mathbf{a} \mathbf{b} \\
 \Rightarrow_{(\text{imm})} B \rightarrow CA B_R | \mathbf{a} \mathbf{b} B_R \\
 B_R \rightarrow C \mathbf{b} B_R | \varepsilon
 \end{array}$$

$$\begin{array}{l}
 i = 3, j = 1: C \rightarrow \underline{\mathbf{A}} B | CC | \mathbf{a} \\
 \Rightarrow C \rightarrow \underline{BC} B | \mathbf{a} B | CC | \mathbf{a}
 \end{array}$$

$$\begin{array}{l}
 i = 3, j = 2: C \rightarrow \underline{B} C B | \mathbf{a} B | CC | \mathbf{a} \\
 \Rightarrow C \rightarrow \underline{C A B_R} C B | \mathbf{a} \mathbf{b} B_R C B | \mathbf{a} B | CC | \mathbf{a} \\
 \Rightarrow_{(\text{imm})} C \rightarrow \mathbf{a} \mathbf{b} B_R C B C_R | \mathbf{a} B C_R | \mathbf{a} C_R \\
 C_R \rightarrow A B_R C B C_R | C C_R | \varepsilon
 \end{array}$$

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

$$A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \dots \mid \alpha \beta_n \mid \gamma$$
 with

$$A \rightarrow \alpha A_R \mid \gamma$$

$$A_R \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

Predictive Parsing

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

FIRST (Revisited)

- $\text{FIRST}(\alpha)$ = the set of terminals that begin all strings derived from α

$\text{FIRST}(a) = \{a\}$ if $a \in T$
 $\text{FIRST}(\epsilon) = \{\epsilon\}$
 $\text{FIRST}(A) = \cup_{A \rightarrow \alpha} \text{FIRST}(\alpha)$ for $A \rightarrow \alpha \in P$
 $\text{FIRST}(X_1 X_2 \dots X_k) =$
 if for all $j = 1, \dots, i-1 : \epsilon \in \text{FIRST}(X_j)$ **then**
 add non- ϵ in $\text{FIRST}(X_i)$ to $\text{FIRST}(X_1 X_2 \dots X_k)$
 if for all $j = 1, \dots, k : \epsilon \in \text{FIRST}(X_j)$ **then**
 add ϵ to $\text{FIRST}(X_1 X_2 \dots X_k)$

FOLLOW

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

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

LL(1) Grammar

- A grammar G is LL(1) if for each collections of productions

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$

for nonterminal A the following holds:

- $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$ for all $i \neq j$
- if $\alpha_i \Rightarrow^* \epsilon$ then
 - $\alpha_j \not\Rightarrow^* \epsilon$ for all $i \neq j$
 - $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \emptyset$ for all $i \neq j$

Non-LL(1) Examples

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

Recursive Descent Parsing

- Grammar must be LL(1)
- Every nonterminal has one (recursive) procedure responsible for parsing the nonterminal's syntactic category of input tokens
- When a nonterminal has multiple productions, each production is implemented in a branch of a selection statement based on input look-ahead information

Using FIRST and FOLLOW to Write a Recursive Descent Parser

$expr \rightarrow term\ rest$ $rest \rightarrow +\ term\ rest$ $- term\ rest$ ϵ $term \rightarrow id$		<pre> procedure rest(); begin if lookahead in <u>FIRST(+ term rest)</u> then match('+'); term(); rest() else if lookahead in <u>FIRST(- term rest)</u> then match('-'); term(); rest() else if lookahead in <u>FOLLOW(rest)</u> then return else error() end; </pre>
--	--	---

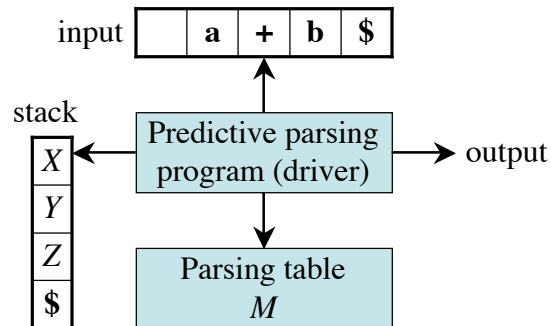
$FIRST(+ term\ rest) = \{ + \}$

$FIRST(- term\ rest) = \{ - \}$

$FOLLOW(rest) = \{ \$ \}$

Non-Recursive Predictive 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 a Predictive Parsing Table

```

for each production  $A \rightarrow \alpha$  do
  for each  $a \in \text{FIRST}(\alpha)$  do
    add  $A \rightarrow \alpha$  to  $M[A,a]$ 
  enddo
  if  $\epsilon \in \text{FIRST}(\alpha)$  then
    for each  $b \in \text{FOLLOW}(A)$  do
      add  $A \rightarrow \alpha$  to  $M[A,b]$ 
    enddo
  endif
enddo
Mark each undefined entry in  $M$  error

```

Example Table

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



$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$E \rightarrow T E_R$	(id	\$)
$E_R \rightarrow + T E_R$	+	\$)
$E_R \rightarrow \epsilon$	ϵ	
$T \rightarrow F T_R$	(id	+ \$)
$T_R \rightarrow * F T_R$	*	+ \$)
$T_R \rightarrow \epsilon$	ϵ	
$F \rightarrow (E)$	(* + \$)
$F \rightarrow \text{id}$	id	

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

LL(1) Grammars are Unambiguous

Ambiguous grammar
 $S \rightarrow \text{i E t S S}_R \mid \text{a}$
 $S_R \rightarrow \text{e S} \mid \epsilon$
 $E \rightarrow \text{b}$



$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$S \rightarrow \text{i E t S S}_R$	i	e \$
$S \rightarrow \text{a}$	a	
$S_R \rightarrow \text{e S}$	e	e \$
$S_R \rightarrow \epsilon$	ϵ	
$E \rightarrow \text{b}$	b	t

Error: duplicate table entry

	a	b	e	i	t	\$
S	$S \rightarrow \text{a}$			$S \rightarrow \text{i E t S S}_R$		
S_R			$S_R \rightarrow \epsilon$ $S_R \rightarrow \text{e S}$			$S_R \rightarrow \epsilon$
E		$E \rightarrow \text{b}$				

Predictive Parsing Program (Driver)

```

push($)
push(S)
a := lookahead
repeat
  X := pop()
  if X is a terminal or X = $ then
    match(X) // move to next token, a := lookahead
  else if  $M[X,a] = X \rightarrow Y_1 Y_2 \dots Y_k$  then
    push( $Y_k, Y_{k-1}, \dots, Y_2, Y_1$ ) // such that  $Y_1$  is on top
    produce output and/or invoke actions
  else error()
endif
until X = $

```

Example Table-Driven Parsing

Stack	Input	Production applied
$\$E$	$\text{id+id*id}\$$	
$\$E_R T$	$\text{id+id*id}\$$	$E \rightarrow T E_R$
$\$E_R T_R F$	$\text{id+id*id}\$$	$T \rightarrow F T_R$
$\$E_R T_R \text{id}$	$\text{id+id*id}\$$	$F \rightarrow \text{id}$
$\$E_R T_R$	$\text{+id*id}\$$	
$\$E_R$	$\text{+id*id}\$$	$T_R \rightarrow \epsilon$
$\$E_R T_+$	$\text{+id*id}\$$	$E_R \rightarrow + T E_R$
$\$E_R T$	$\text{id*id}\$$	
$\$E_R T_R F$	$\text{id*id}\$$	$T \rightarrow F T_R$
$\$E_R T_R \text{id}$	$\text{id*id}\$$	$F \rightarrow \text{id}$
$\$E_R T_R$	$\text{*id}\$$	
$\$E_R T_R F^*$	$\text{*id}\$$	$T_R \rightarrow * F T_R$
$\$E_R T_R F$	$\text{id}\$$	
$\$E_R T_R \text{id}$	$\text{id}\$$	$F \rightarrow \text{id}$
$\$E_R T_R$	$\$$	
$\$E_R$	$\$$	$T_R \rightarrow \epsilon$
$\$$	$\$$	$E_R \rightarrow \epsilon$

Panic Mode Recovery

Add synchronizing actions to
undefined entries based on FOLLOW

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

	id	+	*	()	\$
E	$E \rightarrow T E_R$			$E \rightarrow T E_R$	<i>synch</i>	<i>synch</i>
E_R		$E_R \rightarrow + T E_R$			$E_R \rightarrow \epsilon$	$E_R \rightarrow \epsilon$
T	$T \rightarrow F T_R$	<i>synch</i>		$T \rightarrow F T_R$	<i>synch</i>	<i>synch</i>
T_R		$T_R \rightarrow \epsilon$	$T_R \rightarrow * F T_R$		$T_R \rightarrow \epsilon$	$T_R \rightarrow \epsilon$
F	$F \rightarrow \text{id}$	<i>synch</i>	<i>synch</i>	$F \rightarrow (E)$	<i>synch</i>	<i>synch</i>

synch: pop A and skip input till *synch* token
or skip until $\text{FIRST}(A)$ found

Phrase-Level Recovery

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

	id	+	*	()	\$
E	$E \rightarrow T E_R$			$E \rightarrow T E_R$	<i>synch</i>	<i>synch</i>
E_R		$E_R \rightarrow + T E_R$			$E_R \rightarrow \epsilon$	$E_R \rightarrow \epsilon$
T	$T \rightarrow F T_R$	<i>synch</i>		$T \rightarrow F T_R$	<i>synch</i>	<i>synch</i>
T_R	<i>insert *</i>	$T_R \rightarrow \epsilon$	$T_R \rightarrow * F T_R$		$T_R \rightarrow \epsilon$	$T_R \rightarrow \epsilon$
F	$F \rightarrow \text{id}$	<i>synch</i>	<i>synch</i>	$F \rightarrow (E)$	<i>synch</i>	<i>synch</i>

insert *: insert missing * and redo the production

Error Productions

$$\begin{aligned}
 E &\rightarrow T E_R \\
 E_R &\rightarrow + T E_R \mid \varepsilon \\
 T &\rightarrow F T_R \\
 T_R &\rightarrow * F T_R \mid \varepsilon \\
 F &\rightarrow (E) \mid \mathbf{id}
 \end{aligned}$$

Add error production:

$$T_R \rightarrow F T_R$$

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

	id	+	*	()	\$
<i>E</i>	$E \rightarrow T E_R$			$E \rightarrow T E_R$	<i>synch</i>	<i>synch</i>
<i>E_R</i>		$E_R \rightarrow + T E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
<i>T</i>	$T \rightarrow F T_R$	<i>synch</i>		$T \rightarrow F T_R$	<i>synch</i>	<i>synch</i>
<i>T_R</i>	$T_R \rightarrow F T_R$	$T_R \rightarrow \varepsilon$	$T_R \rightarrow * F T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
<i>F</i>	$F \rightarrow \mathbf{id}$	<i>synch</i>	<i>synch</i>	$F \rightarrow (E)$	<i>synch</i>	<i>synch</i>