Left Recursion

- Productions of the form

$$
\begin{aligned}
A \rightarrow\ & A\ \alpha \\
& \mid \beta \\
& \mid \gamma
\end{aligned}
$$

  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

$$A_i \rightarrow A_j\, \gamma$$

        with

$$A_i \rightarrow \delta_1\, \gamma \mid \delta_2\, \gamma \mid \ldots \mid \delta_k\, \gamma$$

        where

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

    od

    eliminate the *immediate left recursion* in A_i

od

Immediate Left-Recursion Elimination Method

Rewrite every left-recursive production

$$
\begin{aligned}
A \rightarrow\ & A\ \alpha \\
\mid\ & \beta \\
\mid\ & \gamma \\
\mid\ & A\ \delta
\end{aligned}
$$

into a right-recursive production:

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

Example Left Recursion Elim.

$$\left.\begin{array}{l} A \to B\ C \mid \mathbf{a} \\ B \to C\ A \mid A\ \mathbf{b} \\ C \to A\ B \mid C\ C \mid \mathbf{a} \end{array}\right\} \text{Choose arrangement: } A, B, C$$

$i = 1$: nothing to do

$i = 2, j = 1$: $B \to C\ A \mid \underline{A}\ \mathbf{b}$

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

$\Rightarrow_{\text{(imm)}} \quad B \to C\ A\ B_R \mid \mathbf{a}\ \mathbf{b}\ B_R$

$\qquad B_R \to C\ \mathbf{b}\ B_R \mid \varepsilon$

$i = 3, j = 1$: $C \to \underline{A}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow \quad C \to \underline{B\ C}\ B \mid \underline{\mathbf{a}}\ B \mid C\ C \mid \mathbf{a}$

$i = 3, j = 2$: $C \to \underline{B}\ C\ B \mid \mathbf{a}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow \quad C \to \underline{C\ A\ B_R}\ C\ B \mid \underline{\mathbf{a}\ \mathbf{b}\ B_R}\ C\ B \mid \mathbf{a}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow_{\text{(imm)}} \quad C \to \mathbf{a}\ \mathbf{b}\ B_R\ C\ B\ C_R \mid \mathbf{a}\ B\ C_R \mid \mathbf{a}\ C_R$

$\qquad C_R \to A\ B_R\ C\ B\ C_R \mid C\ C_R \mid \varepsilon$

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

$$A \to \alpha\ \beta_1 \mid \alpha\ \beta_2 \mid \ldots \mid \alpha\ \beta_n \mid \gamma$$

with

$$A \to \alpha\ A_R \mid \gamma$$
$$A_R \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

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

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

FIRST(a) = $\{a\}$ if $a \in T$
FIRST(ε) = $\{\varepsilon\}$
FIRST(A) = $\cup_{A\to\alpha}$ FIRST(α) for $A\to\alpha \in P$
FIRST($X_1X_2\ldots X_k$) =
if for all $j = 1, \ldots, i\text{-}1 : \varepsilon \in$ FIRST(X_j) **then**
add non-ε in FIRST(X_i) to FIRST($X_1X_2\ldots X_k$)
if for all $j = 1, \ldots, k : \varepsilon \in$ 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 ($B \to \alpha\ A\ \beta$) $\in P$ **do**
        add FIRST(β)$\setminus\{\varepsilon\}$ to FOLLOW(A)
    for all ($B \to \alpha\ A\ \beta$) $\in P$ and $\varepsilon \in$ FIRST(β) **do**
        add FOLLOW(B) to FOLLOW(A)
    for all ($B \to \alpha\ A$) $\in P$ **do**
        add FOLLOW(B) to FOLLOW(A)
    if A is the start symbol S **then**
        add **$** to FOLLOW($A$)

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. $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \varnothing$ 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. $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \varnothing$ for all $i \neq j$

Non-LL(1) Examples

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

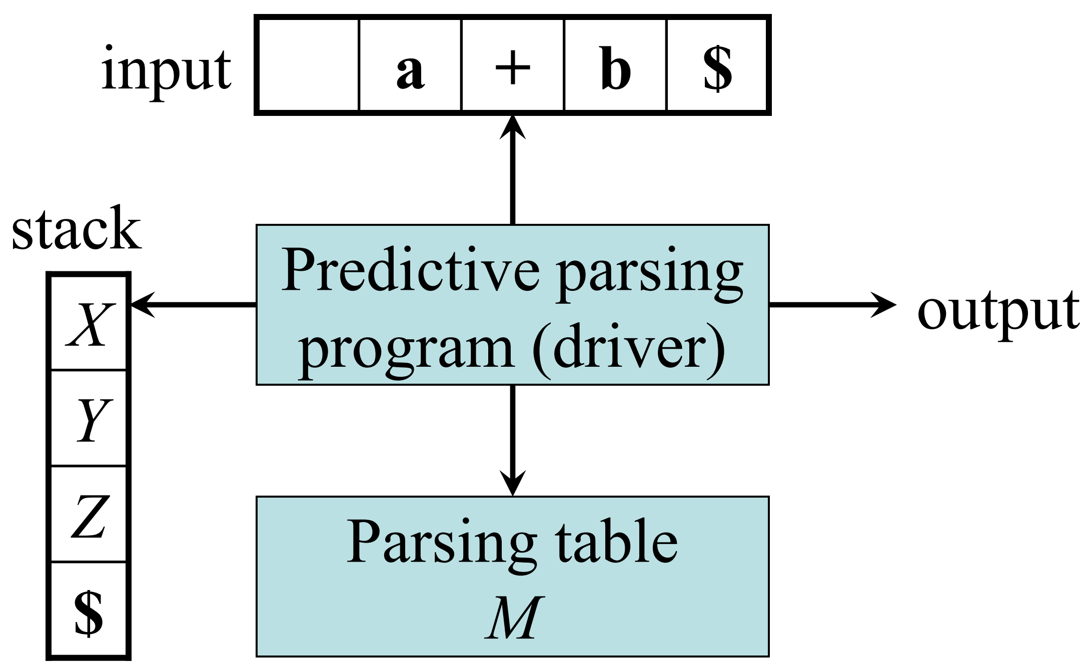
$$
\begin{aligned}
expr &\rightarrow term\ rest \\
rest &\rightarrow +\ term\ rest \\
&\mid -\ term\ rest \\
&\mid \varepsilon \\
term &\rightarrow \mathbf{id}
\end{aligned}
$$

procedure *rest*();
begin
  if *lookahead* in FIRST(+ *term rest*) **then**
    match('+'); *term*(); *rest*()
  else if *lookahead* in FIRST(- *term rest*) **then**
    match('-'); *term*(); *rest*()
  else if *lookahead* in FOLLOW(*rest*) **then**
    return
  else error()
end;

FIRST(+ *term rest*) = { + }
FIRST(- *term rest*) = { - }
FOLLOW(*rest*) = { **$** }

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

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

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

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

LL(1) Grammars are Unambiguous

Ambiguous grammar

$S \to \mathbf{i}\ E\ \mathbf{t}\ S\ S_R \mid \mathbf{a}$

$S_R \to \mathbf{e}\ S \mid \varepsilon$

$E \to \mathbf{b}$

$A \to \alpha$	FIRST(α)	FOLLOW(A)
$S \to \mathbf{i}\ E\ \mathbf{t}\ S\ S_R$	**i**	**e** \$
$S \to \mathbf{a}$	**a**	**e** \$
$S_R \to \mathbf{e}\ S$	**e**	**e** \$
$S_R \to \varepsilon$	ε	**e** \$
$E \to \mathbf{b}$	**b**	**t**

Error: duplicate table entry

	a	**b**	**e**	**i**	**t**	**\$**
S	$S \to \mathbf{a}$			$S \to \mathbf{i}\ E\ \mathbf{t}\ S\ S_R$		
S_R			$S_R \to \varepsilon$ $S_R \to \mathbf{e}\ S$			$S_R \to \varepsilon$
E		$E \to \mathbf{b}$				

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 $M[X,a] = X \to Y_1 Y_2 \ldots Y_k$ **then**
        push(Y_k, $Y_{k\text{-}1}$, …, Y_2, Y_1) // *such that* Y_1 *is on top*
        ... invoke actions and/or produce IR output …
    else   error()
    endif
until X = **$**

Example Table-Driven Parsing

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

Panic Mode Recovery

Add synchronizing actions to undefined entries based on FOLLOW

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

Pro:	Can be automated
Cons:	Error messages are needed

	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		$T_R \to \varepsilon$	$T_R \to *\,F\,T_R$		$T_R \to \varepsilon$	$T_R \to \varepsilon$
F	$F \to$ **id**	***synch***	***synch***	$F \to (\,E\,)$	***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	$F \to$ **id**	***synch***	***synch***	$F \to (\ E\ )$	***synch***	***synch***

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