More Elimination of Left-Recursion

- In general

$$S \to S\,\alpha_1 \mid \ldots \mid S\,\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

- All strings derived from S start with one of $\beta_1,\ldots,\beta_m$ and continue with several instances of $\alpha_1,\ldots,\alpha_n$
- Rewrite as

$$S \to \beta_1\,S' \mid \ldots \mid \beta_m\,S'$$
$$S' \to \alpha_1\,S' \mid \ldots \mid \alpha_n\,S' \mid \varepsilon$$

Contd., Removal of left recursion due to many productions …

- Left recursion may also be introduced by two or more grammar rules.

- For example: S → Aa | b

  A → Ac | Sd | Є

$S \to Aa \mid b$

$\cancel{A \to Ac \mid Sd \mid \varepsilon}$

$A \to \underbrace{Ac}_{\alpha_1} \mid A\underbrace{ad}_{\alpha_2} \mid \underbrace{bd}_{\beta} \mid \varepsilon$

$A \to bdA' \mid A'$

- After the first step (substitute S by its RHS in the rules) the grammar becomes

$A' \to cA' \mid adA'$

S → Aa | b

A → Ac | Aad | bd | Є

Removal of left recursion due to many productions …

- Split productions into α and β
- General rule: If we have A → Aα1 | Aα2 | … | β1 | β2 | …
  Then rewrite as:
    - A → β1 A' | β2 A' | …
    - A' → α1 A' | α2 A' | … | ε

S → Aa | b

A → Ac | Aad | bd | Є

- **Left-recursive parts (α):**
    - Ac → α1 = c
    - Aad → α2 = ad
- **Non-left-recursive parts (β):**
    - **(β1)** bd
    - **(β2)** ε

Removal of left recursion due to many productions …

- Rewrite A without left recursion

  $A \rightarrow bd\ A' \mid A'$

  $A' \rightarrow c\ A' \mid ad\ A' \mid \varepsilon$

- After the second step (removal of left recursion) the grammar becomes

  $S \rightarrow Aa \mid b$

  $A \rightarrow bdA' \mid A'$

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

Left Factoring

- In top-down parsing when it is not clear which production to choose for expansion of a symbol

Defer (accept) the decision till we have seen enough input

- In general, if $A \rightarrow \alpha\beta1 \mid \alpha\beta2$
- Defer (Accept) decision by expanding $A \rightarrow \alpha A'$
- we can then expand $A' \rightarrow \beta1$ or $\beta2$

Therefore $A \rightarrow \alpha\beta1 \mid \alpha\beta2$ →→ transforms to →→ $A \rightarrow \alpha A'$
$A' \rightarrow \beta1 \mid \beta2$

Here:

- α = the common prefix
- A' = a new non-terminal that handles the "rest of the options."

Dangling else problem again

- Dangling else problem can be handled by left factoring

stmt → if expr then stmt else stmt
 | if expr then stmt

can be transformed to

stmt → if expr then stmt S'
S' → else stmt | ϵ

Left Factoring:

1. The common prefix α is if (Expr) Stmt.
2. The varying suffixes β_1 is else Stmt and β_2 is ε (the empty string, meaning "nothing").

Rule : A → αA'
A' → β1 | β2

Summary of Recursive Descent

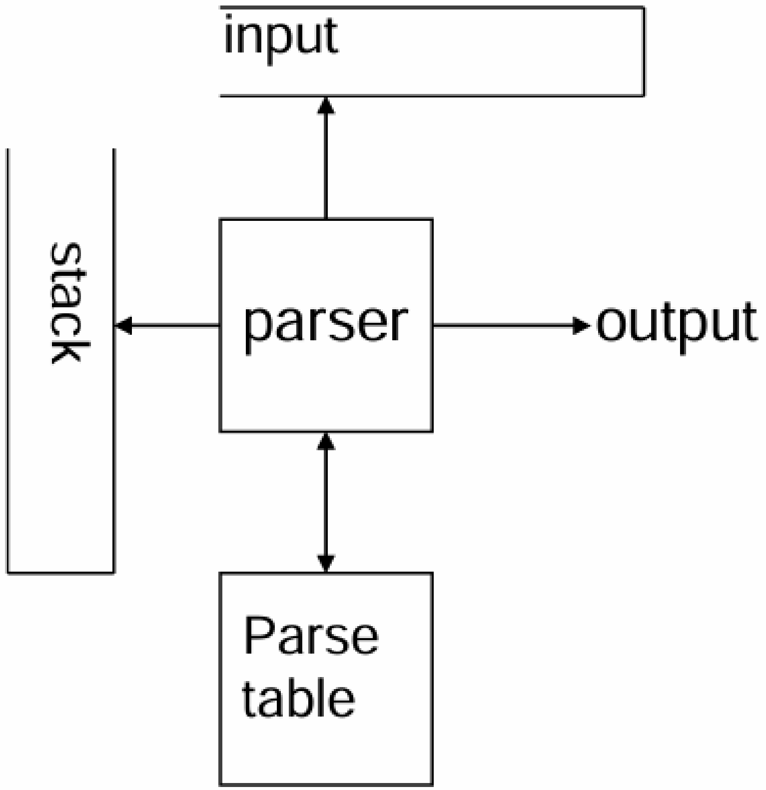
- Simple and general parsing strategy Left-recursion must be eliminated first … but that can be done automatically
- Unpopular because of backtracking
- Thought to be too inefficient
- In practice, backtracking is eliminated by restricting the grammar

Predictive Parsers

- Like recursive-descent but parser can “predict” which production to use
  - By looking at the next few tokens
  - No backtracking
- Predictive parsers accept **LL(k)** grammars
  - **L** means “left-to-right” scan of input
  - **L** means “leftmost derivation”
  - **k** means “predict based on k tokens of lookahead”
- In practice, LL(1) is used

Predictive parsing Contd.,

- Predictive parser can be implemented by maintaining an external stack



Parse table is a two dimensional array M[X,a] where "X" is a non terminal and "a" is a terminal of the grammar

Parsing algorithm

- The parser considers 'X' the symbol on top of stack, and 'a' the current input symbol
- ~~These two symbols determine the action to be~~ taken by the parser
- Assume that '$' is a special token that is at the bottom of the stack and terminates the input string.

```
if X = a = $ then halt

if X = a ≠ $ then pop(x) and ip++

if X is a non terminal
        then if M[X,a] = {X → UVW}
                then begin pop(X); push(W,V,U)
                        end
                else error
```

LL(1) Languages

- In recursive-descent, for each non-terminal and input token there may be a choice of production
- LL(1) means that for each non-terminal and token there is only one production
- Can be specified via 2D tables
  - One dimension for current non-terminal to expand
  - One dimension for next token
  - A table entry contains one production

Contd., with Left Factoring

Here:

- α = the common prefix
- A' = a new non-terminal that handles the "rest of the options."

- Formula of left factoring

$A \rightarrow \alpha\beta1|\alpha\beta2$ can be written as $A \rightarrow \alpha A'$, $A' \rightarrow \beta1 \mid \beta2$

Apply to Expression Grammar: $E \rightarrow T+E \mid T$

Here, both productions start with common prefix T.

So, $\alpha = T$, $\beta1 = +E$, $\beta2 = \epsilon$.

After factoring

$E \rightarrow TX,$
$X \rightarrow +E|\epsilon$

Here X, Y are added as new Non-terminals

Apply to Term Grammar $T \rightarrow (E) \mid int \mid int*T$

Here, **int** is the common prefix.

So $\alpha = int$, and $\beta1 = *T$, $\beta2 = \epsilon$

$T \rightarrow (E) \mid intY$

$Y \rightarrow *T|\epsilon$

final grammar became:

$E \rightarrow TX,\quad X \rightarrow +E \mid \epsilon$
$T \rightarrow (E) \mid intY,\quad Y \rightarrow *T \mid \epsilon$

Left-Factoring Example

- Recall the grammar

$$E \rightarrow T + E \mid T$$
$$T \rightarrow (E) \mid \text{int} \mid \text{int} * T$$

- Factor out common prefixes of productions

$$E \rightarrow T X$$
$$X \rightarrow + E \mid \varepsilon$$
$$T \rightarrow (E) \mid \text{int } Y$$
$$Y \rightarrow * T \mid \varepsilon$$

LL(1) Parsing Table Example

- Left-factored grammar

$E \rightarrow T\,X$ $\qquad$ $X \rightarrow +\,E \mid \varepsilon$

$T \rightarrow (\,E\,) \mid \text{int}\,Y$ $\qquad$ $Y \rightarrow *\,T \mid \varepsilon$

- The LL(1) parsing table:

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

Contd.,

$$E \to TX$$

$$T \to (E) \mid int\, Y$$

$$X \to +E \mid \varepsilon$$

$$Y \to *T \mid \varepsilon$$

FIRST sets:

- FIRST(E) = FIRST(T) = { (, int }
- FIRST(T) = { (, int }
- FIRST(X) = { +, ε}
- FIRST(Y) = { ∗, ε}

FOLLOW sets:

- FOLLOW(E) = {), $ }
- FOLLOW(T) = FIRST (X) ∪ FOLLOW (E) = { +,), $ }
- FOLLOW (X) = FOLLOW (E) = {), $ }
- FOLLOW (Y) = FOLLOW (T) = { +,), $}

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε