More Elimination of Left-Recursion

- In general

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

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

- Rewrite as

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

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

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

  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 → bd A' | A'

    A' → c A' | ad A' | ε

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

    S → Aa | b

    A → bdA' | A'

    A' → cA' | adA' | Є

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 → αβ1 | αβ2

- Defer (Accept) decision by expanding A → αA'

- we can then expand A' → β1 or β2

Therefore A → αβ1 | αβ2   →→ transforms to →→   A → αA'
A' → β1 | β2

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 | ∈

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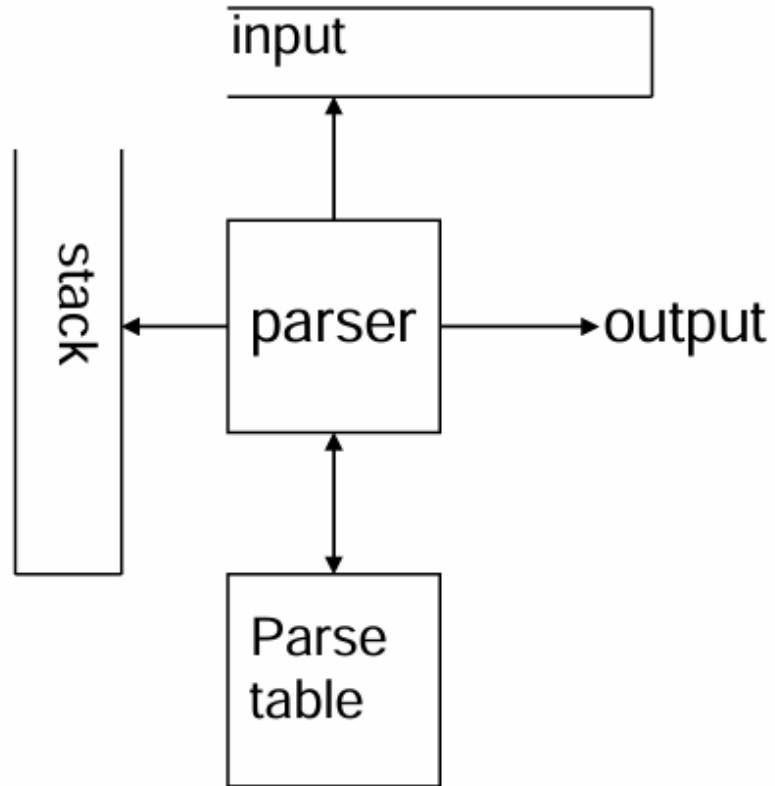
Parse table is a two dimensional array M[X,a] where "X" is a non terminal and "a" is a terminal of the grammar

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

$$A→αβ1|αβ2 \quad \text{can be written as} \quad A→αA', \quad A'→β1 \mid β2$$

Apply to Expression Grammar: E→T+E | T

Here, both productions start with common prefix T.
So, α = T, β1 = +E, β2 = ϵ.

After factoring

$$E→TX,$$
$$X→+E|ϵ$$

Here X, Y are added as new Non-terminals

Apply to Term Grammar T→(E) | int | int∗T

Here, **int** is the common prefix.
So α = int, and β1=∗T, β2 = ϵ

$$T → (E) \mid intY$$
$$Y → ∗T|ϵ$$

final grammar became:

$$E→ TX, \qquad X→ +E \mid ϵ$$
$$T→(E) \mid intY, \quad Y→∗T \mid ϵ$$

Left-Factoring Example

- Recall the grammar

  $E \rightarrow T + E \mid T$

  $T \rightarrow (E) \mid int \mid int * T$

- Factor out common prefixes of productions

  $E \rightarrow T X$

  $X \rightarrow + E \mid \varepsilon$

  $T \rightarrow (E) \mid int Y$

  $Y \rightarrow * T \mid \varepsilon$

LL(1) Parsing Table Example

- **Left-factored grammar**

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

- **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.,

$$E \rightarrow TX$$

$$T \rightarrow (E) \mid int\ Y$$

$$X \rightarrow +E \mid \varepsilon$$

$$Y \rightarrow *T \mid \varepsilon$$

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