# Parser

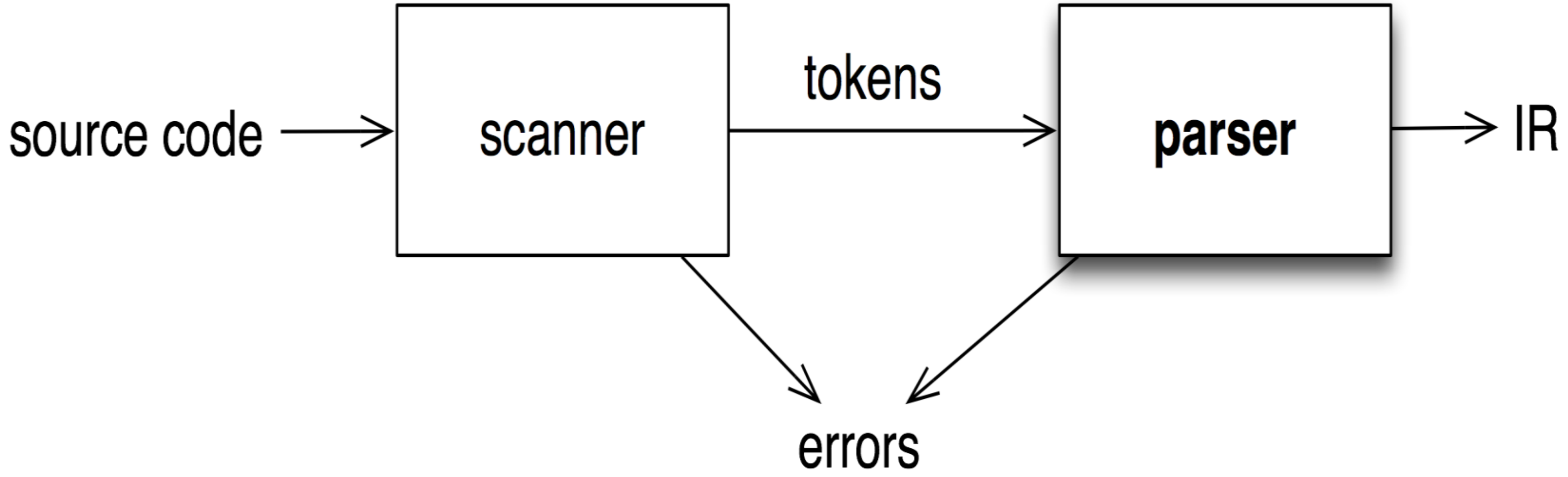
# Second Phase of the compiler

- Parser – Typically integrated with the lexical phase of the compiler
- Top Down Parser
- Bottom Up Parser

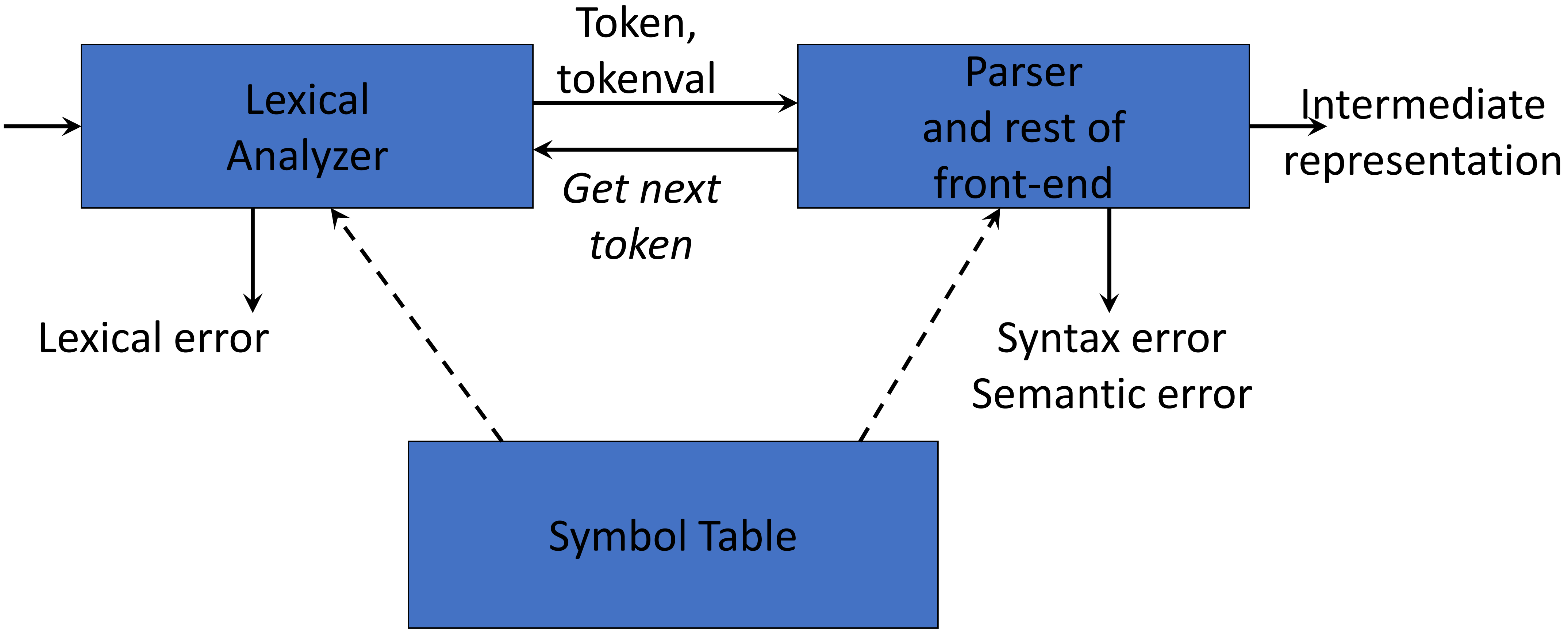
# Functions of the Parser

- Validate the syntax of the programming language
- Points out errors in the statements

# Role of the Parser



# Role of the Parser



# General Types of Parsers

- Universal Parsers
  - Cocke- Younger-Kasami
  - Earley’s Algorithm
- Top-Down Parsers
- Bottom Up Parsers

# Universal Parsers

- Can parse any Grammar
- Use in NLP
- But too inefficient in Compilers

# Top Down Parsers

- Build the parse trees from the top to the bottom
- Recursive Descent parsers – requires backtracking
- LL Parsers – No Backtracking

# Example

Consider the Grammar

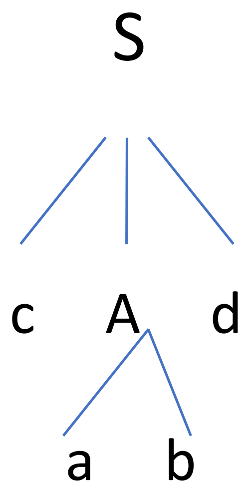
$$S \rightarrow c\ A\ d$$

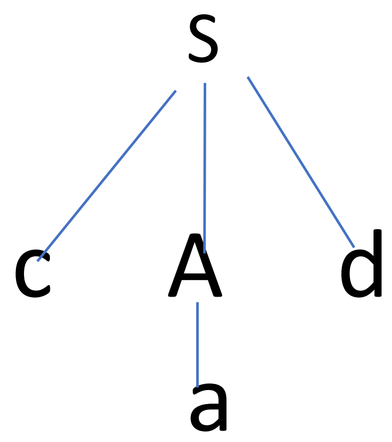
$$A \rightarrow ab \mid a$$

Let the input be "*cad*"

# Parsing (Recursive Descent)

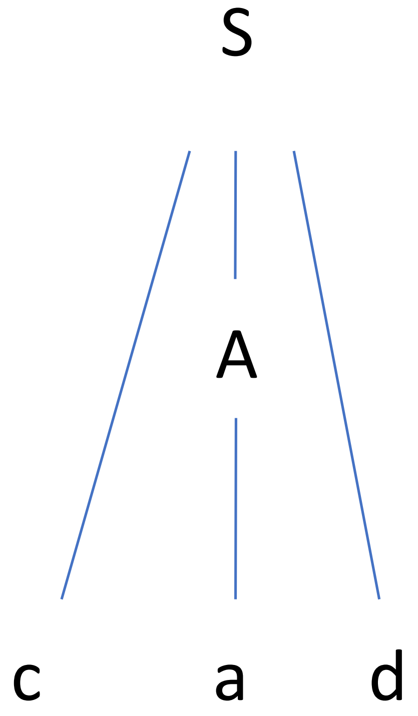
Expand A using the first alternative A → ab



# Bottom up Parsers

- Start from the bottom and work up to the root for parsing a string
- LR parsers are bottom up parsers



# Parsing

- Both Top Down and Bottom up parsers parse the string based on a viable-prefix property
- This property states that before the string is fully processed, if there is an error, the parser will identify it and recovers

# Context Free Grammars - CFG

- Programming language constructs are defined using context free grammar
- For example

  $E \rightarrow E + E \mid E * E \mid (E) \mid id$

  Expression grammar involving the operators, +, *, ( )

# Context Free Grammars

- Defined formally as (V, T, P, S)

  V – Variables / Non-terminals

  T – Terminals that constitute the string

  P – Set of Productions that has a LHS and RHS

  S – Special Symbol, subset of V

# Context Free Grammar

- Example

stmt → if E then stmt else stmt

stmt → if E then stmt

stmt → a

E → b

Here, stmt, E are Non-terminals,

if, then, else, a, b are all terminals

# Grammar - notations

- Terminals - $a,b,c,\ldots \in T$
  - specific terminals: **0**, **1**, **id**, **+**
- Non-terminals - $A,B,C,\ldots \in$
  - specific non-terminals: *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

  $\alpha,\beta,\gamma \in (N \cup T)^*$

# Derivation

- 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 $\alpha$ does not contain a nonterminal
  - $\Rightarrow$ is *rightmost* $\Rightarrow_{rm}$ if $\beta$ does not contain a nonterminal
  - Transitive closure $\Rightarrow^*$ (zero or more steps)
  - Positive closure $\Rightarrow^+$ (one or more steps)

# Derivation

- The *language generated by G* is defined by

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

# Derivation

$E \rightarrow E \mathbf{+} E$

$E \rightarrow E \mathbf{*} E$

$E \rightarrow \mathbf{(}\ E\ \mathbf{)}$

$E \rightarrow \mathbf{-}\ E$

$E \rightarrow \mathbf{id}$

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

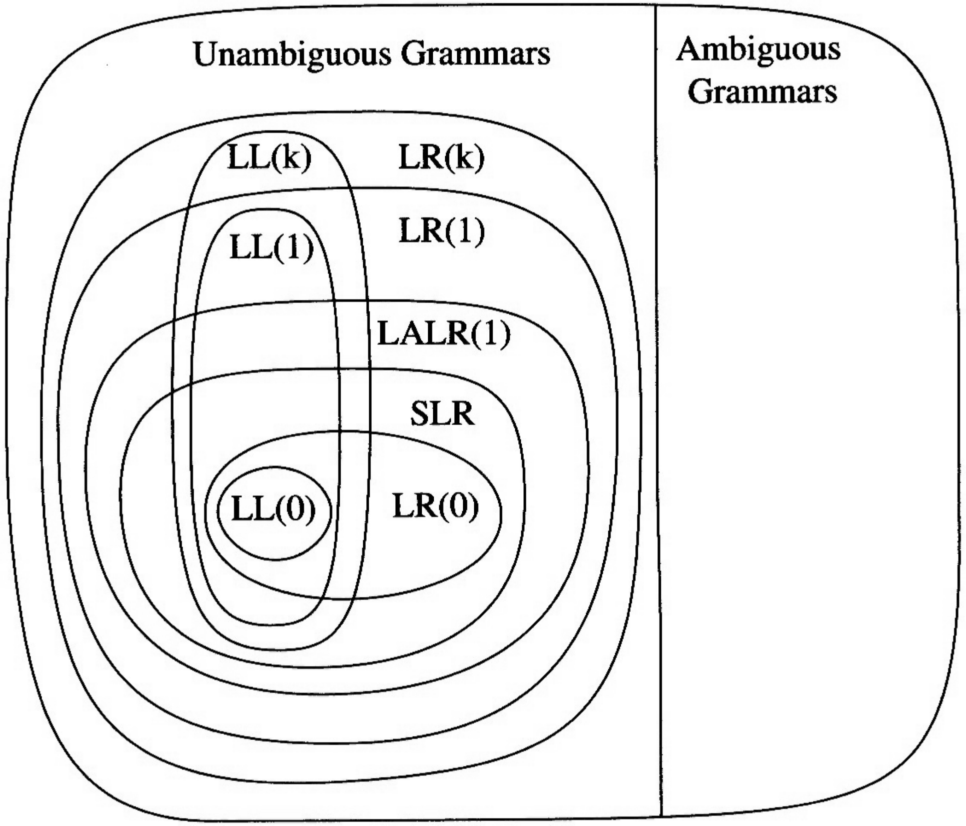
$$E \Rightarrow_{rm} E \mathbf{+} E \Rightarrow_{rm} E \mathbf{+\ id} \Rightarrow_{rm} \mathbf{id + id}$$

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

# Parsers

- Context Free grammars are already defined for all programming constructs
- All strings that are part of the programming language will be based on this construct
- Hence, parsers are designed keeping in mind the CFG

# Hierarchy of Grammar Classes



# Hierarchy

- **LL(*k*):**
  - **L**eft-to-right, **L**eftmost derivation, *k* tokens lookahead
- **LR(*k*):**
  - **L**eft-to-right, **R**ightmost derivation, *k* tokens lookahead
- **SLR:**
  - **S**imple **LR** (uses "follow sets")
- **LALR:**
  - **L**ook**A**head **LR** (uses "lookahead sets")

# Top Down Parsers

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

Grammar:

$E \rightarrow T \mathbf{+} T$

$T \rightarrow \mathbf{(}\ E\ \mathbf{)}$

$T \rightarrow \mathbf{-}\ E$

$T \rightarrow \mathbf{id}$

Leftmost derivation:

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

# Top Down Parsers – LL (1) Parsers

- LL parsers cannot handle
  - Left Recursive Grammar
  - Left Factoring

# Left Recursive Grammar

- *Formally, a grammar* is *left recursive* if $\exists$ $A \in NT$ such that $\exists$ a derivation $A \Rightarrow^{+} A\alpha$, for some string $\alpha \in (NT \cup T)^{+}$
- $A \rightarrow A\ \alpha$ / $\beta$ | $\gamma$

# Left Factor

- When a non-terminal has two or more productions whose right-hand sides start with the same grammar symbols the grammar is said to have left-factor property
- *Example* $A \rightarrow \alpha\ \beta_1$ / $\alpha\ \beta_2$ / … | $\alpha\ \beta_n$ | $\gamma$

# Pre-requisites for Top-Down Parser

- Eliminate Left Recursion
- Left Factor the grammar

$A \gamma \beta$

$A \to A\alpha \mid B\beta$

$B \to A\gamma \mid \varepsilon$

# Eliminating Left Recursion

Arrange the non-terminals 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

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

**end**

eliminate the immediate left recursion in $A_i$

**end**

# Eliminate Left Recursion

- Rewrite every left-recursive production

$$A \rightarrow A\,\alpha \;/\; \beta \mid \gamma \mid A\,\delta$$

- into a right-recursive production:

$$A \rightarrow \beta\, A_R \;/\; \gamma\, A_R$$
$$A_R \rightarrow \alpha\, A_R \;/\; \delta\, A_R \;/\; \varepsilon$$

# Example

- $A \rightarrow B\ C \mid \mathbf{a}$
  $B \rightarrow C\ A \mid A\ \mathbf{b}$
  $C \rightarrow A\ B \mid C\ C \mid \mathbf{a}$

- $i$ = 1: nothing to do

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

  $\Rightarrow$ $B \rightarrow C\,A \mid \underline{B\,C}\ \mathbf{b} \mid \underline{\mathbf{a}}\ \mathbf{b}$

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

  $B_R \rightarrow C\ \mathbf{b}\ B_R \mid \varepsilon$

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

  $\Rightarrow$ $C \rightarrow \underline{B\,C}\ B \mid \underline{\mathbf{a}}\ B \mid C\,C \mid \mathbf{a}$

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

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

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

  $C_R \rightarrow A\ B_R\ C\ B\ C_R \mid C\ C_R \mid \varepsilon$

# Example - Expression Grammar

E→ E+T | T

T → T * F | F

F → (E) | id

# Modified Grammar

- E → TE’
- E’ → +TE’ | ε
- T → FT’
- T’ → *FT’ | ε
- F→ (E) | id

# Left Factoring

- Replace productions

  $A \rightarrow \alpha\ \beta_1 \ /\ \alpha\ \beta_2 \ /\ \ldots\ |\ \alpha\ \beta_n\ |\ \gamma$

  with

  $A \rightarrow \alpha\ A_R\ |\ \gamma$

  $A_R \rightarrow \beta_1 \ /\ \beta_2 \ /\ \ldots\ /\ \beta_n$

# Left Factoring

**METHOD**: For each nonterminal $A$, find the longest prefix $\alpha$ common to two or more of its alternatives. If $\alpha \neq \epsilon$ — i.e., there is a nontrivial common prefix — replace all of the $A$-productions $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, where $\gamma$ represents all alternatives that do not begin with $\alpha$, by

$$
\begin{aligned}
A &\to \alpha A' \mid \gamma \\
A' &\to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n
\end{aligned}
$$

Here $A'$ is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix. $\square$

# Left Factoring - Example

S → <u>iCtS</u> | <u>iCtS</u>eS | a

C → b

# Left Factoring

- S → iCTSS' | a
- S' → eS | ε
- C→ b

# LL (1) Parser – Predictive parser

- L – input is scanned from left to right
- L – left derivation
- (1) – looking at 1 input symbol

# Predictive Parser LL (1)

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

# Recursive descent with Recursive calls

- Recursive-descent parsing is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input
- One procedure is associated with each nonterminal of a grammar
- A simple form of recursive-descent parsing, called predictive parsing, in which the lookahead symbol unambiguously determines the flow of control through the procedure body for each nonterminal
- The sequence of procedure calls during the analysis of an input string implicitly defines a parse tree for the input, and can be used to build an explicit parse tree, if desired.

```
void A() {
    Choose an A-production, A → X_1 X_2 ··· X_k;
    for ( i = 1 to k ) {
        if ( X_i is a nonterminal )
            call procedure X_i();
        else if ( X_i equals the current input symbol a )
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}
```

# Recursive descent with Recursive calls

$$
\begin{array}{lcl}
\mathit{stmt} & \rightarrow & \textbf{expr} \; ; \\
& | & \textbf{if} \; ( \; \textbf{expr} \; ) \; \mathit{stmt} \\
& | & \textbf{for} \; ( \; \mathit{optexpr} \; ; \; \mathit{optexpr} \; ; \; \mathit{optexpr} \; ) \; \mathit{stmt} \\
& | & \textbf{other} \\
\\
\mathit{optexpr} & \rightarrow & \epsilon \\
& | & \textbf{expr}
\end{array}
$$

# Recursive descent with Recursive calls

```
void stmt() {
        switch ( lookahead ) {
        case expr:
                match(expr); match(';'); break;
        case if:
                match(if); match('('); match(expr); match(')'); stmt();
                break;
        case for:
                match(for); match('(');
                optexpr(); match(';'); optexpr(); match(';'); optexpr();
                match(')'); stmt(); break;
        case other;
                match(other); break;
        default:
                report("syntax error");
        }
}
```

# Recursive descent with Recursive calls

void $optexpr()$ {
    if ( $lookahead$ == **expr** ) $match$(**expr**);
}

void $match$(terminal $t$) {
    if ( $lookahead$ == $t$ ) $lookahead$ = nextTerminal;
    else $report$(`"syntax error"`);
}