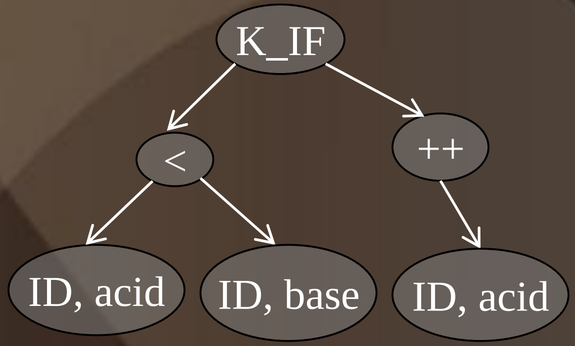
Lecture #7

Syntax Analysis - I

Syntax Analysis

1. **if (acid** < base) ++acid;
    - **Parser Input:** K_IF, (, <ID, acid>, <, <ID, base>,), ++, <ID, acid>, DELIM
    - **Parser Output:**



- After lexical analysis, we have a series of tokens.
- In **syntax analysis (or parsing), we** want to interpret what those tokens mean.
- Goal: Obtain the *structure* described by the series of tokens
  Report *errors* if the tokens do not properly encode a structure.

Formal Grammar

A grammar, G, is a 4-tuple G={S,N,T,P}, where:

S is a starting symbol; N is a set of non-terminal symbols;
T is a set of terminal symbols; P is a set of production rules.

Example:

LAUGH $\rightarrow$	LAUGH hah	rule 1
	| hah	rule 2

We can use this grammar to create sentences: E.g.:

Rule	**Sentential Form**
-	LAUGH
1	LAUGH hah
2	hah hah

Such a sequence of rewrites is called a *derivation*

Derivations

$$\alpha A \beta \rightarrow \alpha\gamma\beta \quad \text{if} \quad A \rightarrow \gamma$$

$$\begin{cases} \alpha \rightarrow \alpha \\ \alpha \xrightarrow{*} \beta \quad \text{and} \quad \beta \rightarrow \gamma \quad \text{then} \quad \alpha \xrightarrow{*} \gamma \end{cases}$$

$S \xrightarrow{*} \alpha$ $\begin{cases} \alpha \text{ is a sentential form} \\ \alpha \text{ is a sentence if it contains only terminal symbols} \end{cases}$

The process of discovering a derivation for some sentence is called parsing!

Regular Grammars and DFAs

- We need a *language* for describing valid strings of tokens and a m*ethod* for distinguishing valid from invalid strings of tokens.

- Regular Grammar - Productions have the form: $A \rightarrow xB$ or $A \rightarrow \varepsilon$

- Regular Grammar to DFA:
  - Terminal symbols – *Input alphabet* for the DFA
  - Non-terminal symbols – Represent *States*
  - If a production has the form: $A \rightarrow \varepsilon$, then 'A' is an accepting state
  - A production of the form $A \rightarrow xB$ denotes a transition from state 'A' to state 'B' on input symbol 'x'.

Syntax Analysis using DFAs?

- Can we do syntax analysis with DFAs?

  We could design a grammar for expressions of the form '*num* + *num*' using a DFA

  $E \rightarrow$ num A
  $A \rightarrow + B$
  $B \rightarrow$ num C
  $C \rightarrow \varepsilon$

- Programming languages have recursive structures
  - For eg: *EXPR* $\rightarrow$ if *EXPR* then *EXPR* else *EXPR*
    | *OTHER*

- DFAs cannot count – cannot handle such programming structures
  - Example: Regular grammars cannot define: *Expressions with properly balanced parentheses* $\{(^i\,)^i \mid i \geq 0\}$ [S$\rightarrow$ (S) | ε]
  - Similarly: *Functions with properly nested block structure*

Push Down Automata

- The situation can be handled if the DFA is augmented with *memory*
  - Memory implemented as a *stack*
  - Such an automata is called *Push Down Automata (PDA)*

- State table for a PDA that recognizes nested parentheses:

		Input Symbol		
		(	)	EOF
State	0	Push 1	Error	Accept
	1	Push 1	Pop	Error

- Determine the stack values and actions at each step as the following string is parsed: (( ) (( )))

Context Free Grammars

- PDAs have the power to accept Context Free Grammars (CFGs)

- Formally a CFG G = (T, N, S, P), where:
  - T is the set of *terminal* symbols in the grammar (i.e., the set of tokens returned by the lexical analyzer)
  - N, the *non-terminals*, are variables that denote sets of (sub)strings occurring in the language. These impose a structure on the grammar.
  - S is the *goal symbol*, a distinguished non-terminal in N denoting the entire set of strings in L(G).
  - P is a finite set of *productions* specifying how terminals and non-terminals can be combined to form strings in the language. Each production must have a single non-terminal on its left hand side.

Context Free Grammars

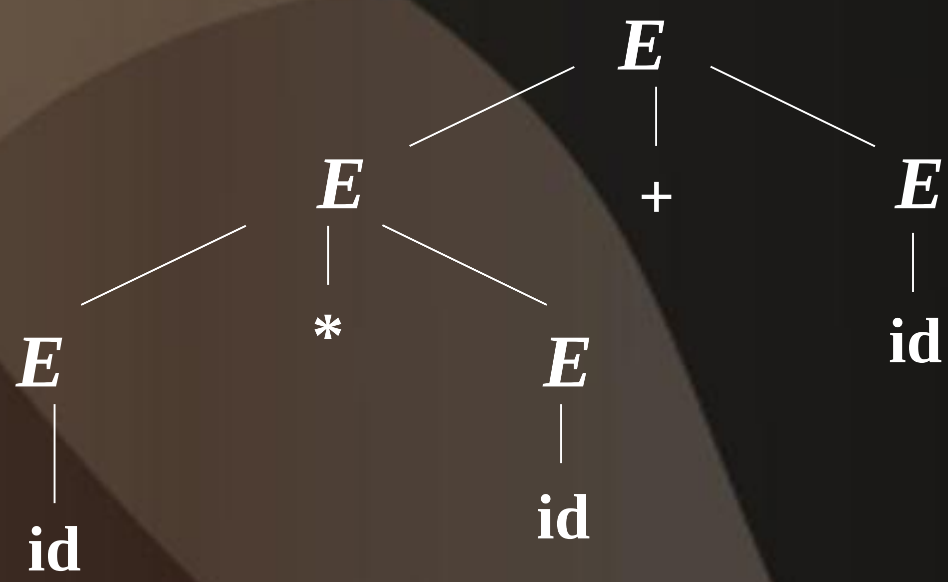
- A possible CFG for arithmetic operations:

$$\mathbf{E \rightarrow E + E \mid E * E \mid (E) \mid id}$$

- Input string: id * id + id

$$
\begin{aligned}
\mathbf{E} &\rightarrow \mathbf{E + E}\\
&\rightarrow \mathbf{E * E + E}\\
&\rightarrow \mathbf{id * E + E}\\
&\rightarrow \mathbf{id * id + E}\\
&\rightarrow \mathbf{id * id + id}
\end{aligned}
$$

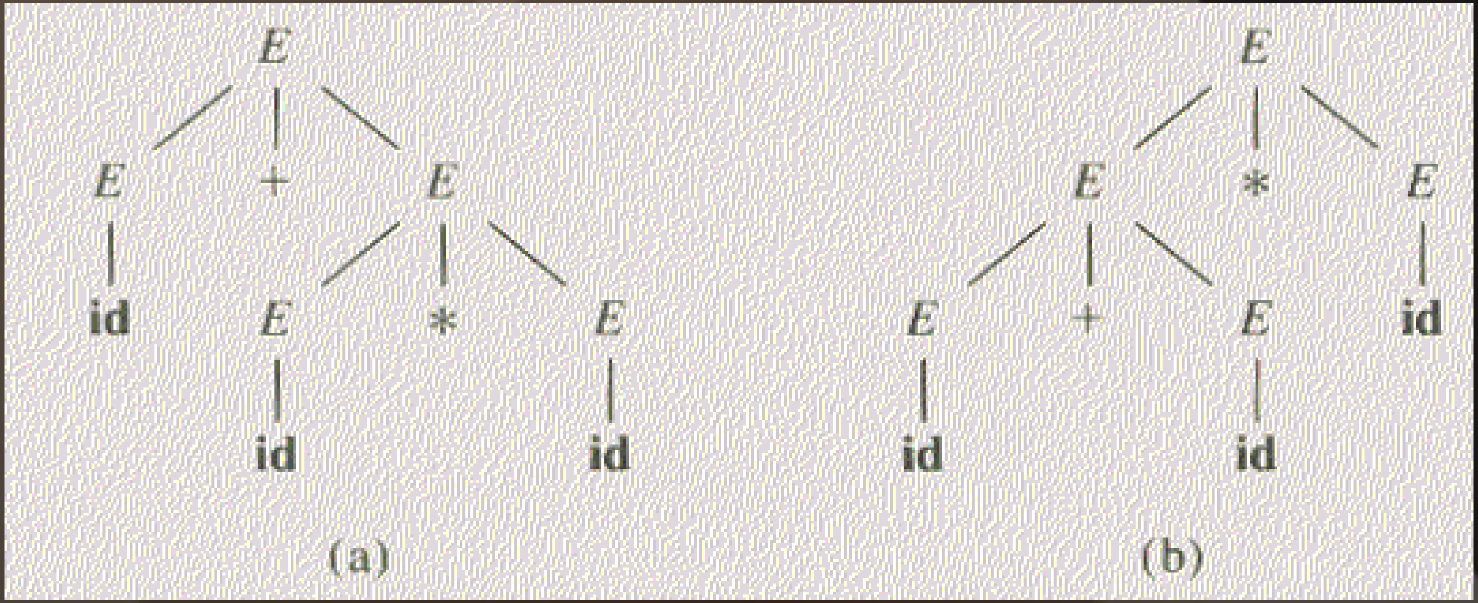
Left-Most Derivation



- Inorder traversal of the leaves give the original input string
- All derivations of a string should yield the same parse tree

Ambiguity

- Derivation defines a parse tree
- A grammar is *ambiguous* if more than one right-most or left-most derivations may be obtained for some string / sentence
- Equivalently, a grammar is ambiguous if more than one parse tree may be obtained for some string



Eliminating Ambiguity

- Ambiguity is bad
  - Leaves meaning of some programs ill-defined
  - Leaves up to the compiler which parse tree to accept
- Several ways to handle ambiguity
- Layering (most direct)

$$\mathbf{E \rightarrow E + E \mid E * E \mid (E) \mid id} \quad \Rightarrow \quad \begin{array}{l} \mathbf{E \rightarrow E' + E \mid E'} \\ \mathbf{E' \rightarrow id * E' \mid id \mid (E) * E' \mid (E)} \end{array}$$

  - Enforces precedence of ‘*’ over ‘+’
  - **E** controls **‘+’** and **E’** controls **‘*’**
  - All the ‘+’s will be handled before any of the ‘*’s
  - ‘*’s will always be nested more deeply inside the parse tree than the ‘+’s

Eliminating Ambiguity

- Another Expression

  E → if E then E | if E then E else E | OTHER

- The expression: **if E_1 then if E_2 then E_3 else E_4** has two separate parse trees

- The property that we want is: *else matches the closest unmatched then*

- Can be resolved as:

  E → MIF| UIF
  MIF → if E then MIF else MIF | OTHER
  UIF → if E then E | if E then MIF else UIF

Eliminating Left-Recursion

- Direct Left-Recursion

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

$$\Downarrow$$

$$A \to \beta A'$$
$$A' \to \alpha A' \mid \varepsilon$$

$$A \to A\alpha 1 \mid \ldots \mid A\alpha m \mid \beta 1 \mid \ldots \mid \beta n$$

$$\Downarrow$$

$$A \to \beta 1 A' \mid \ldots \mid \beta n A'$$
$$A' \to \alpha 1 A' \mid \ldots \mid \alpha n A' \mid \varepsilon$$

Eliminating Left-Recursion

- Indirect Left-Recursion

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid \varepsilon$$

- Algorithm

Arrange the non-terminals in some order $A_1,\ldots,A_n$.
for (i in 1..n) {
    for (j in 1..i-1) {
        replace each production of the form $A_i \rightarrow A_j\gamma$ by the
        productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where
        $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
    }
    eliminate the immediate left recursion among A_i productions
}

Left Factoring

$$A \rightarrow \alpha\beta1 \mid \ldots \mid \alpha\beta n \mid \gamma$$

$$\downarrow$$

$$A \rightarrow \alpha A' \mid \gamma$$
$$A' \rightarrow \beta1 \mid \ldots \mid \beta n$$

Next Lecture

Top-Down Parsing