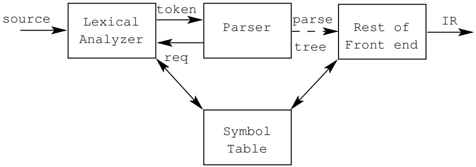
Syntactic Analysis

Introduction

- Second phase of the compiler.
- Main task:
  - Analyze syntactic structure of program and its components
  - to check these for errors.
- Role of parser:



- Approach to constructing parser: similar to lexical analyzer
  - Represent source language by a meta-language, *Context Free Grammar*
  - Use algorithms to construct a recognizer that recognizes strings generated by the grammar.
    This step can be automated for certain classes of grammars. One such tool: YACC.
  - Parse strings of language using the recognizer.

Context Free Grammar (CFG)

- Syntax analysis based on theory of automata and formal languages, specifically the equivalence of two mechanisms of context free grammars and pushdown automata.
- Context free grammars used to describe the *syntactic structures* of programs of a programming language. Describe what elementary constructs there are and how composite constructs can be built from other constructs.

  Stmt $\rightarrow$ `if (Expr) Stmt else Stmt`

  Note recursive nature of definition.
- Formally, a CFG has four components:
  a) a set of tokens V_t, called *terminal symbols*, (token set produced by the scanner) examples: `if`, `then`, `identifier`, etc.
  b) a set of different intermediate symbols, called *non-terminals, syntactic categories, syntactic variables*, V_n
  c) a start symbol, $S \in V_n$, and
  d) a set of productions P of the form
     $A \rightarrow X_1 \cdots X_n$
     where $A \in V_n$, $X_i \in (V_n \cup V_t), 1 \leq i \leq m, m \geq 0$.
- Sentences generated by starting with S and applying productions until left with nothing but terminals.
- Set of strings *derivable* from a CFG G comprises the *context free language*, denoted $L(G)$.

CFG - example.

- Nonterminal start with uppercase letters. rest are non-terminals.
- If-then-else:

```
Stmt     →  IfStmt | other
IfStmt   →  if ( Exp ) Stmt ElseStmt
ElseStmt →  else Stmt | ϵ
Exp      →  0 | 1
```

  Example strings:

```
other
if (0) other
if (1) other else if (0) other else other
```

  Derivation of `if (1) other else if (0) other else other`:

```
Stmt ⇒ IfStmt ⇒ if (Exp) Stmt ElseStmt
⇒ if (1) Stmt ElseStmt
...
```

- Grammar for sequence of statements:

```
StmtSeq →  Stmt; StmtSeq | Stmt
Stmt    →  s
```

  L(G) = { s, s;s, s;s;s, ... }

- What if statment sequence is empty?

```
StmtSeq →  Stmt; StmtSeq | ϵ
Stmt    →  s
```

  L(G) = { ϵ, s;, s;s;, s;s;s;, ... }

  Note: Here ';' is not a statement separator, but a terminator.

  What if we want a statement separator?

```
StmtSeq        →  NonEmpStmtSeq | ϵ
NonEmpStmtSeq  →  Stmt; NonEmpStmtSeq | Stmt
Stmt           →  s
```

Context Free Grammar (CFG) - cont'd.

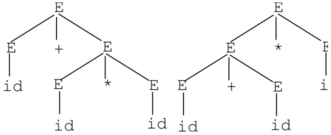
- Notations:
  1. Nonterminals: Uppercase letters such as A, B, C
  2. Terminals: lower case letters such as a,b, c, operators $+$,$-$, etc, punctuation, digits, and boldface strings such as **id**.
  3. Nonterminals or terminals: Upper-case letters late in alphabet, such as X, Y, Z.
  4. Strings of terminals: lower-case letters late in alphabet, such as x, y, z.
  5. Strings of grammar symbols: lower-case greek letters α, β, etc.
  6. Write $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, etc as
     $A \rightarrow \alpha_1 | \alpha_2 | \cdots$
- Example:
  $$E \rightarrow E\ A\ E \mid (\ E\ ) \mid -E \mid \textbf{id}$$
  $$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$$
- Derivation of strings: a production can be thought of as a rewrite rule in which nonterminal on left is replaced by string on right side.
  Notation: Write such a replacement as E $\Rightarrow$ (E).
  Example:
  $$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\textbf{id})$$

CFG - cont'd.

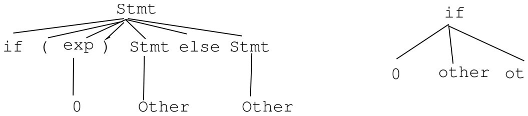
- Notation: Write $\alpha A \beta \;\Rightarrow\; \alpha\gamma\beta$ if $A \;\rightarrow\; \gamma$.
- Notation: Write $\alpha \;\overset{*}{\Rightarrow}\; \beta$ to denote that β can be derived from α in zero or more steps.
  $L(G) = \{\alpha |\; S \;\overset{*}{\Rightarrow}\; \alpha\}$
- Sentential form: α is a sentential form, if $S \;\overset{*}{\Rightarrow}\; \alpha$ and α contains non-terminals.
  Example: $E + E$
- *Leftmost derivation:* Derivation $\alpha \;\Rightarrow\; \beta$ is leftmost if the leftmost terminal in α is replaced.
  Example:
  $E \overset{*}{\Rightarrow} EAE \overset{*}{\Rightarrow} \mathbf{id}AE \overset{*}{\Rightarrow} \mathbf{id} + E \overset{*}{\Rightarrow} \mathbf{id} + \mathbf{id}$
  Production sequence discovered by a large class of parsers (the top-down parsers) is a leftmost derivation; hence, these parsers are said to produce *leftmost parse*.
- Rightmost derivation: Derivation $\alpha \;\Rightarrow\; \beta$ is left most if the rightmost terminal in α is replaced.
  Example:
  $E \overset{*}{\Rightarrow} EAE \overset{*}{\Rightarrow} EA\mathbf{id} \overset{*}{\Rightarrow} E + \mathbf{id} \overset{*}{\Rightarrow} \mathbf{id} + \mathbf{id}$
  Also, called *canonical derivation*. Corresponds well to an important class of parsers (the bottom-up parsers). In particular, as a bottom up parser discovers the productions used to derive a token sequence, it discovers a rightmost derivation, but in *reverse order*: last production applied is discovered first, while the first production is the last to be discovered.

Representations of derivations

- Derivations represented graphically by a derivation of **parse tree:**
  - Root: start symbol, leaves: grammar symbols or ϵ
  - Interior nodes: nonterminals; Offsprings of a nonterminal represent application of a rule.
- Example: Parse tree for leftmost and rightmost derivations of string $id + id * id$:



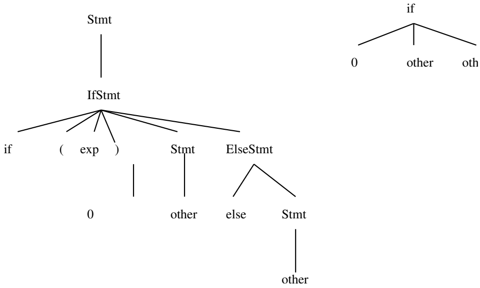
- **Abstract syntax tree**: A more abstract representation of the input string.



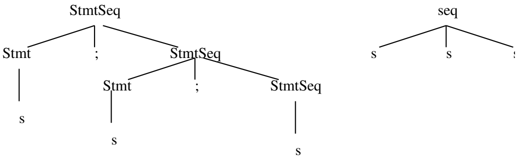
- Parse tree may contain information that may not be needed in later phases of compiler. AST does not include intermediate nodes primary used for derivation purposes.
- In general, during the semantic analysis phase, the parse tree of a string may be converted into an abstract syntax tree.

Parse Tree - Examples

- Parse tree for string: if (o) other else other



- Parse tree for string: s;s;s



Properties of Context Free Grammars

- Context free grammars that are limited to productions of the form A $\rightarrow$ a B and C $\rightarrow$ ϵ form the class of *regular grammars*. Languages defined by regular grammars are a proper subset of the context-free languages.
- Why not use lexical analysis during parsing?
  - Lexical rules are in general simple.
  - RE are more concise and easier to understand.
  - Domain specific language so that efficient lexical analyzer can be constructed.
  - Separate into two manageable parts. Useful for multi-lingual programming.
- *Non-reduced CFGs*: A CFG containing nonterminals that are unreachable or derive no terminal string.
  Example:

  ```
  S → A|B
  A → a
  B → B b
  C → c
  ```

  Nonterminal C cannot be reached from S. B does not derive any strings. Useless terminals can be safely removed from a CFG without affecting the language. Reduced grammar:

  ```
  S → A
  A → a
  ```

  Algorithms exist that check for useless nonterminals.

Properties of Context Free Grammars - Ambiguity

- *Ambiguity*: A context free grammar is *ambiguous* if it allows different derivation trees for a single tree.

Each tree defines a different semantics for −

- No algorithm exists for automatically checking if a grammar is ambiguous (impossibility result). However, for certain grammar classes (including those that generate parsers), one can prove that grammars are unambiguous.
- How to eliminate ambiguity: one way is to rewrite the grammar: Example:

```
S → if E then S | if E then S else S

S → M|U
M → if E then M else M
U → if E then S | if E then M else U
```

Represents semantics:Match each **else** with the closet previous unmatched **then**. The above transformation makes the grammar unnecessarily complex.

- Another approach: Disambiguate by defining additional tokens `end`.

```
S → if E then S end | if E then S else S end
```

- Provide information to the parser so that it can handle it in a certain way.

Properties of Context Free Grammars - cont'd.

- *Left recursion*: G is left recursive if for a nonterminal A, there is a derivation $A \stackrel{+}{\Rightarrow} A\alpha$
  Top-down parsing methods cannot handle left-recursive grammars. So eliminate left recursion.
- *Left factoring*: Factor out the common left prefixes of grammars: Replace grammar $A \rightarrow \alpha\beta_1|\alpha\beta_2$ by the rule:
  $A \rightarrow \alpha A'$
  $A' \rightarrow \beta_1|\beta_2$
- Context free grammars are not powerful enough to represent all constructs of programming languages.
  Cannot distinguish the following:
  - $L_1 = \{wcw|w \in (a|b)^*\}$: Conceptually represents problem of verifying that an identifier is declared before used. Such checkings are done during the semantic analysis phase.
  - $L_2 = \{a^n b^m c^n c^m|n \geq 1 \wedge m \geq 1\}$. Abstracts the problem of checking that number of formal parameters agrees with the number of actual parameters.
  - $L_3 = \{a^n b^n c^n|n \geq 0\}$.

  CFG's can keep count of two items but not three.

Properties of Context Free Grammars - cont'd.

- Context free grammar can capture some of language semantics as well.
- Example grammar:

```
<exp>    ::=<exp> + <term> | <term>
<term>   ::=<term> * <term>
            | '('<exp>')'
            | <number>
<number> ::= 0 | 1 | ··· | 9
```

- Precedence of * over +: by deriving * lower in the parse tree.
- Left recursion

  ```
  <exp> ::= <exp> + <term>
  ```

  left associativity of +
- Right recursion:

  ```
  <exp> ::= <term> + <exp>
  ```

  right associativity of +

Backus-Naur Form(BNF)

- BNF: a kind of CFG.
- First used in Algol60 report. Many extensions since, but all similar and most give power of context-free grammar.
- Has four parts: (i) terminals (atomic symbols), (ii) non-terminals (representing constructs), called *syntactic categories*, iii) *productions* and iv) a starting nonterminal.
- Each nonterminal denotes a set of strings. Set of strings associated with starting nonterminal represents language.
- BNF uses following notations:
  (i) Non-terminals enclosed in $<$ and $>$.
  (ii) Rules written as

$$X ::= Y$$

    (a) X is LHS of rule and can only be a NT.
    (b) Y can be a string, which is a terminal, nonterminal, or concatenation of terminal and nonterminals, or a set of strings separated by alternation symbol $|$.

- Example: Terminals: A, B, $\cdots$Z; 0, 1, $\cdots$ 9
  Nonterminals: <id>, <rest>, <alpha>, <alphanum>, <digit>
  Starting NT: <id>
  Productions/rules:

```
<id>         ::= <alpha> | <alpha><rest>
<rest>       ::= <rest><alphanum> | <alphanum>
<alphanum>   ::= <alpha> | <digit>
<alpha>      ::= A | B | ··· | Z
<digit>      ::= 0 | 1 | ··· | 9
```

Extended BNF (EBNF)

- Extend BNF by adding more meta-notation $\Longrightarrow$ shorter productions
- Nonterminals begin with uppercase letters (discard <>)
- Terminals that are grammar symbols ('[' for instance) are enclosed in ''.
- Repetitions (zero or more) are enclosed in {}
- Options are enclosed in []:
- Use () to group items together:

```
Exp ::= Item {+ Item} | Item {- Item}
    ⟹
Exp ::= Item {(+|-) Item}
```

Conversion from EBNF to BNF and Vice Versa

- BNF to EBNF:
  - i) Look for recursion in grammar:

    ```
    A ::= a A | B  ⟹  { a } B
    ```
  - ii) Look for common string that can be factored out with grouping and options.

    ```
    A ::= a B | a  ⟹  A := a [B]
    ```
- EBNF to BNF:
  - i) Options []:

    ```
    A ::= a [B] C  ⟹
        A’ ::= a N C
        N ::= B | ε
    ```
  - ii) Repetition {}:

    ```
    A ::= a B1 B2 ...  Bn C  ⟹
        A’ ::= a N C
        N ::= B1 B2 ...  Bn N | ε
    ```