Context Free Grammars

CFGs are used to describe the syntax of programming languages. A production

\[
S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2
\]

in a CFG says

“If \(S_1\) and \(S_2\) are statements and \(E\) an expression then ‘if \(E\) then \(S_1\) else \(S_2\)’ is a statement”.

Notice that this production is recursive; it allows if-statements to occur within if-statements.

if, then, and else are terminal symbols or tokens.

\(S, S_1, S_2,\) and \(E\) are non-terminals. They are like “variables”, that represent the kinds of strings that the grammar defines as statements or expressions, respectively.

CFG Notation

- **terminals:** \(a, b, c, \ldots, +, -, 0, 1, \ldots, \text{if, do}\).
- **nonterminals:** \(A, B, C, \ldots, S, \ldots, \text{expr, stmt}.\)
- **grammar symbols:** \(X, Y, Z, \ldots\) (either terminals or nonterminals).
- **strings of terminals:** \(u, v, w \ldots\).
- **strings of grammar symbols:** \(\alpha, \beta, \gamma, \ldots\) (strings of terminals or nonterminals).
- **productions:** \(A \rightarrow \alpha_1, A \rightarrow \alpha_2, \ldots, A \rightarrow \alpha_k, \) or \(A \rightarrow \alpha_1 | \alpha_2 \ldots | \alpha_k\).

Derivations — Productions as Rewrite Rules

1. Start with the **start symbol**, \(S\).
2. Pick any production \(S \rightarrow \alpha\), eg. \(S \rightarrow \text{id := } E\).
3. We say that \(S\) derives \(\text{id := } E\), or \(S \Rightarrow \text{id := } E\). ‘\(\text{id := } E\)’ is a sentential form derived from \(S\).
4. Repeat: pick a nonterminal \(A\) from the sentential form, replace with the RHS of a production \(A \rightarrow \alpha\):
   \[
   \begin{align*}
   S \Rightarrow \text{id := } E \Rightarrow \text{id := } E + E \\
   \Rightarrow \text{id := } \text{id + } E \Rightarrow \text{id := } \text{id + num}.
   \end{align*}
   \]
   \(S \Rightarrow \text{id := } \text{id + num}\).

\[
S \rightarrow \text{id := } E \mid \text{if } E \text{ then } S
\]

\[
E \rightarrow E + E \mid \text{id | num}
\]
Terminology

- A grammar is a 4-tuple
  
  (non-terminals, terminals, productions, start-symbol)

  or

  \((N, \Sigma, P, S)\)

- A production is of the form \(\alpha \rightarrow \beta\) where \(\alpha, \beta\) are taken from \(N \cup \Sigma\).

- Read \(\alpha \rightarrow \beta\) as "rewrite \(\alpha\) with \(\beta\)."
- Read \(\Rightarrow\) as "directly derives".
- Read \(r \Rightarrow\) as "directly derives using rule \(r\)."
- Read \(\ast \Rightarrow\) as "derives in zero or more steps".

Derivations...

\[ \alpha A \beta \Rightarrow \alpha \gamma \beta \]  
if \(A \rightarrow \gamma\) is a production, and \(\alpha, \beta\) are strings of grammar symbols.

\(\Rightarrow\): Derives in one step.
\(\Rightarrow\): Derives in 0 or more steps.
\(\Rightarrow\): Derives in 1 or more steps.
\(\Rightarrow\): Leftmost derivation.
\(\Rightarrow\): Rightmost derivation.

\(L(G)\): The language generated by grammar \(G\). This is the set of strings \(w\), such that there is a derivation \(S \Rightarrow w\), where \(S\) is \(G\)'s start-symbol.

Parse Trees...

The string of terminal symbols \([\text{id}:=\text{id}+\text{num}]\) is generated by a leftmost derivation:

\[
\begin{align*}
S & \Rightarrow_{lm} \text{id} := E \Rightarrow_{lm} \text{id} := E + E \\
& \Rightarrow_{lm} \text{id} := \text{id} + E \Rightarrow_{lm} \text{id} := \text{id} + \text{num} \\
S & \Rightarrow \text{id} := \text{id} + \text{num}
\end{align*}
\]

Example Grammar: 

\[
\begin{align*}
S & \rightarrow \text{id} := E \mid \text{if } E \text{ then } S \\
E & \rightarrow E + E \mid \text{id} \mid \text{num}
\end{align*}
\]

If one step of our derivation is

\[
\ldots A \ldots \Rightarrow \ldots X Y Z \ldots
\]
(i.e., we used the rule \(A \rightarrow XYZ\)) then we'll get a parse (sub-)tree

```
    .
   /
  A
  /|
X Y Z
 .|.
 .|.
```
Top-Down Parsing

Program ⇒ BEGIN Stat END
⇒ BEGIN ident := Expr END
⇒ BEGIN "a" := Expr END
⇒ BEGIN "a" := Expr + Expr END
⇒ BEGIN "a" := 5 + Expr END
⇒ BEGIN "a" := 5 + Expr * Expr END
⇒ BEGIN "a" := 5 + 4 * Expr END
⇒ BEGIN "a" := 5 + 4 * 3 END

Top-Down Backtracking Parser

- Top-down parsing involves building a parse tree for the input string by starting at the root and adding nodes in preorder.

  \[ S \rightarrow cAd \quad A \rightarrow ab \mid a \]

- If a backtracking top-down parser chooses the wrong production rule to expand a node, it backs up over the input and undoes some of the parse tree construction:

  ![Parse Tree Examples]

  - For the production \( S \rightarrow cAd \), the parser constructs a parse tree:

    \[
    S \rightarrow cAd \\
    \downarrow \\
    c \quad a \quad d \\
    \]

  - For the production \( A \rightarrow ab \mid a \), the parser constructs a parse tree:

    \[
    A \rightarrow ab \\
    \downarrow \\
    a \quad b \\
    \]

  - If the parser backtracks over the input, it undoes some of the parse tree construction:

    \[
    S \rightarrow cAd \\
    \downarrow \\
    c \quad a \quad d \\
    \]
Ambiguous Grammars

Grammar Rewriting

A grammar is ambiguous if some string of tokens can produce two (or more) different parse trees.

\[ E ::= E + E | E * E | \text{number} \]

\[
\begin{align*}
E & \Rightarrow E + E \\
& \Rightarrow 5 + E \\
& \Rightarrow 5 + E * E \\
& \Rightarrow 5 + 4 * E \\
& \Rightarrow 5 + 4 * 3 \\
E & \Rightarrow E * E \\
& \Rightarrow E * 3 \\
& \Rightarrow E + E * 3 \\
& \Rightarrow E + 4 * 3 \\
& \Rightarrow 5 + 4 * 3
\end{align*}
\]

Operator Precedence

The precedence of an operator is a measure of its binding power, i.e. how strongly it attracts its operands.

Usually \( \ast \) has higher precedence than \(+\): \[ 4 + 5 \ast 3 \]

means \[ 4 + (5 \ast 3) \]

not \[ (4 + 5) \ast 3 \]

We say that \( \ast \) binds harder than \(+\).

Operator Associativity

The associativity of an operator describes how operators of equal precedence are grouped.

\( + \) and \( - \) are usually left associative:

\[ 4 - 2 + 3 \]

means \[ (4 - 2) + 3 = 5 \]

not \[ 4 - (2 + 3) = -1 \]

We say that \( + \) associates to the left.

\( ^\) associates to the right:

\[ 2^{3^4} = 2^{(3^4)} \]
We must write unambiguous expression grammars that reflect the associativity and precedence of all operators.

The next slide gives the algorithm for writing such grammars.

**Resulting Expression Grammar:**

```plaintext
expr ::= expr + term | term
term ::= term * factor | factor
factor ::= ( expr ) | number
```

1. Create one non-terminal for each precedence level, for example $p_1, p_2, \cdots, p_n$, where $p_n$ has the highest precedence level.
2. For operator $op$ at precedence level $i$ construct the following production if the operator is
   - left associative:
     
     $p_i ::= p_i op p_{i+1} | p_{i+1}$
   - right associative:
     
     $p_i ::= p_{i+1} op p_i | p_{i+1}$
3. Construct a production for nonterminal $p_{n+1}$ which represents primary expressions such as identifiers, numbers, parenthesized expressions, etc:

   $p_{n+1} ::= ( p_1 ) | num | id$

**Top-Down Parsing**

```
E ::= E + T | T
T ::= T * F | F
F ::= number
```

![Diagram of top-down parsing]

```
E ⇒ E + T  E ⇒ E + T
  ⇒ T + T  ⇒ E + T * F
  ⇒ F + T  ⇒ E + T * 3
  ⇒ 5 + T  ⇒ E + F * 3
  ⇒ 5 + T * F  ⇒ E + 4 * 3
  ⇒ 5 + F * F  ⇒ T + 4 * 3
  ⇒ 5 + 4 * F  ⇒ F + 4 * 3
  ⇒ 5 + 4 * 3  ⇒ 5 + 4 * 3
```
Recursive Descent Parsing

PROCEDURE S ();
    IF curr_tok = \texttt{if} THEN
        match(if); E();
        match(then); S();
    ELSIF curr_tok = \texttt{id} THEN
        match(id); match(:=); E();
    ELSE syntax error ENDIF;
PROCEDURE E ();
    IF curr_tok = \texttt{id} THEN match(id);
    ELSE IF curr_tok = \texttt{num} THEN match(num);
    ELSE E(); match(\texttt{+}); E();
    ENDIF;
S \rightarrow \texttt{id} := E
| \texttt{if} \ E \ \texttt{then} \ S
E \rightarrow E + E

Recursive Descent—\textit{Small Problem 1}

We may loop forever:

PROCEDURE E ();
    IF \ldots
    ELSE E(); match(\texttt{+}); E();
    \ldots

Recursive Descent—\textit{Small Problem 2}

What about productions that start out similarly:

\[ S \rightarrow \texttt{if} \ E \ \texttt{then} \ S \mid \texttt{if} \ E \ \texttt{then} \ S \ \texttt{else} \ S \]

PROCEDURE S ();
    IF curr_tok = \texttt{if} THEN
        match(if); E(); match(then); S();
    ELSIF curr_tok = \texttt{id} THEN
        match(id); E(); match(then);
    ELSE syntax error ENDIF;
PROCEDURE E ();
    IF curr_tok = \texttt{id} THEN match(id);
    ELSE IF curr_tok = \texttt{num} THEN match(num);
    ELSE E(); match(\texttt{+}); E();
    ENDIF;

Recursive Descent—\textit{Small Problem 3}

What if there are several possible "next" tokens:

\[ \texttt{prog} \rightarrow \texttt{decl} \mid \texttt{stat} \]
\[ \texttt{stat} \rightarrow \texttt{if} \ldots \mid \texttt{id()} \mid \texttt{while} \ldots \]
\[ \texttt{decl} \rightarrow \texttt{int} \ \texttt{id} \mid \texttt{real} \ \texttt{id} \]

PROCEDURE prog ();
    IF curr_tok \in \{\texttt{if, id, while}\} THEN stat();
    ELSIF curr_tok \in \{\texttt{int, real}\} THEN decl()
    ELSE syntax error ENDIF;
END;
PROCEDURE stat (); \ldots \ END;
PROCEDURE decl () \ldots \ END;
**Left Recursion Removal**

Left recursion must be removed from the grammar, by turning it into right recursion:

\[
A \rightarrow A \alpha | \beta \\
\Rightarrow \quad A \rightarrow \beta R
\]

\[
R \rightarrow \alpha R | \epsilon
\]

Example:

\[
expr \rightarrow expr + term | term
\]

\[
\\
expr \rightarrow term R
R \rightarrow + term R | \epsilon
\]

**Left Factoring**

A top-down parser that reads input from left-to-right, can't choose between productions \( E \rightarrow abF \) and \( E \rightarrow abcF \). These must be left factored:

\[
A \rightarrow \alpha \beta_1 | \alpha \beta_2 \\
\Rightarrow \quad A \rightarrow \alpha A'
\]

\[
A' \rightarrow \beta_1 | \beta_2
\]

Example:

\[
S \rightarrow \text{if } E \text{ then } S \text{ else } S | \Rightarrow \quad S \rightarrow \text{if } E \text{ then } S S'
\]

\[
S' \rightarrow \text{else } S | \epsilon
\]

**Top-Down Expression Parser**

After left recursion removal, our expression grammar turns into:

\[
E \rightarrow E + T | T
\]

\[
T \rightarrow T * F | F
\]

\[
F \rightarrow (E) | \text{id}
\]
Top-Down Expression Parser...

Read Louden, pp. 143–196.

Or, the Dragon Book:
Top-Down Parsing 181–190
Error Recovery 192–195
Recursive Descent Parsing 40–55, 75–76