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.

- \text{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:

\[ a, b, c, \ldots, +, -, \ldots, 0, 1, \ldots, \text{if, do} \]

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:

\[ A \rightarrow \alpha_1, A \rightarrow \alpha_2, \ldots, A \rightarrow \alpha_k, \text{ or} \]
\[ A \rightarrow \alpha_1 | \alpha_2 | \cdots | \alpha_k. \]
Derivations

• Productions as rewriting 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 \( A \rightarrow \alpha \):
     \[ S \Rightarrow \text{id} := E \Rightarrow \text{id} := E + E \Rightarrow \]
     \[ \text{id} := \text{id} + E \Rightarrow \text{id} := \text{id} + \text{num} \]
     \( S \Rightarrow \text{id} := \text{id} + \text{num} \).

Example Grammar:

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

Slide 14–4

Terminology

• A grammar is a 4-tuple
  \( (N, \Sigma, P, S) \)
  (non-terminals, terminals, productions, start-symbol)

• 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 \( \Rightarrow^* \) as “directly derives using rule \( r \)”.

• Read \( \Rightarrow^* \) as “derives in zero or more steps”.

Slide 14–5

Derivations...

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

More notation:

\( \Rightarrow \): Derives in one step.

\( \Rightarrow^* \): Derives in 0 or more steps.

\( \Rightarrow^+ \): Derives in 1 or more steps.

\( \Rightarrow^L \): Leftmost derivation.

\( \Rightarrow^R \): 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.

Example Grammar:

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

Slide 14–6

Derivations...

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

\[
S \Rightarrow \text{id} := E \\
\Rightarrow \text{id} := E + E \\
\Rightarrow \text{id} := \text{id} + E \\
\Rightarrow \text{id} := \text{id} + \text{num} \\
S \Rightarrow \text{id} := \text{id} + \text{num}
\]

Example Grammar:

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

Slide 14–7
Parse Trees...

- If one step of our derivation is
  \[ A \rightarrow XYZ \]
  (i.e., we used the rule \( A \rightarrow XYZ \)) then we'll get a parse (sub-)tree

\[ \cdots \]

\[ A \cdots \Rightarrow \cdots XYZ \]

Slide 14–8

Parse Trees...

Program \( \Rightarrow \) BEGIN Stat END
\[ \Rightarrow \) BEGIN ident := Expr END
\[ \Rightarrow \) BEGIN "a" := Expr END
\[ \Rightarrow \) BEGIN "a" := Expr + Expr END
\[ \Rightarrow \) BEGIN "a" := 5 \( + \) Expr END
\[ \Rightarrow \) BEGIN "a" := 5 \( + \) Expr END
\[ \Rightarrow \) BEGIN "a" := 5 \( + \) Expr END
\[ \Rightarrow \) BEGIN "a" := 5 \( + \) 4 \( * \) Expr END
\[ \Rightarrow \) BEGIN "a" := 5 \( + \) 4 \( * \) 3 END

\[ \text{Program} \]

\[ \text{BEGIN Stat END} \]

\[ \text{ident := Expr} \]

\[ \"a" \]

\[ \text{Expr + Expr} \]

\[ \text{5} \text{ Expr * Expr} \]

\[ \text{4} \text{ 3} \]

Slide 14–9

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.

\[ S \rightarrow a A d \]

\[ A \rightarrow a \]

\[ S \rightarrow c A d \]

\[ A \rightarrow c A d \]

\[ S \rightarrow a d \]

\[ a d \]

\[ a d \]

\[ a d \]

\[ a d \]

Slide 14–10

Top-Down Backtracking Parser...

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

\[ S \rightarrow A d \]

\[ A \rightarrow a \]

\[ d \]

\[ A \rightarrow c A d \]

\[ S \rightarrow a d \]

\[ a d \]

\[ a d \]

\[ a d \]

Slide 14–11
**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:
  
  $$2 \cdot 3 \cdot 4 = 2^{(3 \cdot 4)}.$$  

**Expression Grammars**

- 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.

<table>
<thead>
<tr>
<th>Resulting Expression Grammar:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>`expr ::= expr + term</td>
<td>term`</td>
<td></td>
</tr>
<tr>
<td>`term ::= term * factor</td>
<td>factor`</td>
<td></td>
</tr>
<tr>
<td>`factor ::= ( expr )</td>
<td>number`</td>
<td></td>
</tr>
</tbody>
</table>

**Ambiguous Grammars**

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

\[
E ::= E + E \mid E \cdot E \mid \text{number}
\]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[5 + 4 \cdot 3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operator Precedence**

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

- Usually * has higher precedence than +:

  $$4 + 5 \cdot 3$$

  means
  
  $$4 + (5 \cdot 3),$$

  not
  
  $$(4 + 5) \cdot 3.$$  

- We say that * binds harder than +.
1. Create one non-terminal for each precedence level, for example $p_1, p_2, \ldots, p_n$, 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:
   $$ p_i := p_i | op(p_{i+1}) $$
   $$ p_i := p_{i+1} | op(p_i) $$

3. Construct a production for nonterminal $p_{i+1}$ which represents primary expressions such as identifiers, numbers, parentheses, etc.

   $p_{i+1} := (p_i) | \text{num} | id$,

Expression Grammars...

$$ E ::= E + T | T $$
$$ T ::= T * F | F $$
$$ F ::= \text{number} $$

Recursivie Descent Parsing

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

Example Grammar:

$$ S \rightarrow \text{id} := E | \text{if} E \text{ then } S $$
$$ E \rightarrow E + E | \text{id} | \text{num} $$

Recursive Descent Parsing...

Small Problem 1: We may loop forever:

PROCEDURE E ()
IF \ldots
  ELSE E(); match(+); E();
\ldots

Small Problem 2: What about productions that start out similarly:

$$ S \rightarrow \text{if} E \text{ then } S | $$
$$ \text{if} E \text{ then } S \text{ else } S $$

PROCEDURE S ()
IF curr_tok = if THEN
  match(if); E(); match(then); S();
ELSIF curr_tok = if THEN
  match(if); E(); match(then);
  S(); match(else); S();
ELSIF \ldots ENDIF
Recursive Descent Parsing...

Small Problem 3: What if there are several possible “next” tokens:

- \( prog \to decl | stat \)
- \( stat \to if...id(id) | while... \)
- \( decl \to int id | real id \)

```plaintext
PROCEDURE prog ();
IF curr_tok ∈ \{if, id, while\} THEN
stat();
ELSIF curr_tok ∈ \{int, real\} THEN
decl();
ELSE syntax error ENDIF;
END;

PROCEDURE stat (); ... END;
PROCEDURE decl () ; ... END;
```

---

Left Recursion Removal

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

```
Algorithm:

\[
A \to A\alpha | \beta \\
\]

\[
A \to \beta R \\
R \to \alpha R | \epsilon
\]
```

Example:

```
expr \to expr +term | term \\
\]

expr \to term R \\
R \to \ast term R | \epsilon
```

---

Left Recursion Removal...

- After left recursion removal, our expression grammar

\[
E \to E + T | T \\
T \to T + F | F \\
F \to (E) | id
\]

turns into

\[
E \to T E' \\
E' \to \ast T E' | \epsilon \\
T \to FT' \\
T' \to \ast F T' | \epsilon \\
F \to (E) | id
\]

---

Left Factoring

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

```
Algorithm:

\[
A \to \alpha \beta_1 | \alpha \beta_2 \\
\]

\[
A \to \alpha A' \\
A' \to \beta_1 | \beta_2
\]
```

Example:

```
S \to if E then S else S | if E then S \\
\]

S \to if E then SS' \\
S' \to else S | \epsilon
```

---
Readings and References

- 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