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.

Context Free Grammars...

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

- \text{if}, \text{then}, and \text{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 \text{statements} or \text{expressions}, respectively.
Terminology

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

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

• Read \(\alpha \to \beta\) as “rewrite \(\alpha\) with \(\beta\).”

• Read \(\Rightarrow\) as “directly derives”.

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

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

CGF 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:
\(A \to \alpha_1, A \to \alpha_2, \ldots, A \to \alpha_k, \text{ or } A \to \alpha_1 | \alpha_2 | \ldots | \alpha_k\).

Derivations...

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

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

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

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

\(\Rightarrow^{lm}\): Leftmost derivation.

\(\Rightarrow^{rm}\): Rightmost derivation.

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

Derivations — Productions as Rewrite Rules

1. Start with the start symbol, \(S\).

2. Pick any production \(S \to \alpha\), eg. \(S \to \text{id} := E\).

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 \(A \to \alpha\):
   \(S \Rightarrow \text{id} := E \Rightarrow \text{id} := E + E \Rightarrow \text{id} := \text{id} + E \Rightarrow \text{id} := \text{id} + \text{num}\).

\(S\): Derivations

\begin{align*}
S & \to \text{id} := E | \text{if} \ E \ \text{then} \ S
E & \to \ E + E | \text{id} | \text{num}
\end{align*}
Derivations...

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

$$
S \rightarrow \text{id}:=E \quad \text{lm} \quad \text{id}:=\text{id}+E
$$

$$
S \rightarrow \text{id}:=\text{id}+\text{num}
$$

Example Grammar:

$$
S \rightarrow \text{id}:=E \mid \text{if} \; E \; \text{then} \; S
$$

$$
E \rightarrow E+E \mid \text{id} \mid \text{num}
$$

Parse Trees...

- If one step of our derivation is
  
  $$
  \cdots \; A \; \cdots \Rightarrow \cdots \; X \; Y \; Z \; \cdots
  $$

  (i.e., we used the rule $A \rightarrow XYZ$) then we’ll get a parse (sub-)tree

  $A$

  $X \quad Y \quad Z$

  $\cdots \quad \cdots \quad \cdots$
**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*.
- $\ast$ associates to the right:
  
  $$2 \ast 3 \ast 4 = 2^{(3 \ast 4)}.$$

---

**Top-Down Backtracking Parser…**

- 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:

---

**Ambiguous Grammars**

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

$$E ::= E \ast E \mid E + E \mid \text{number}$$

$$E \Rightarrow E \ast E$$

$$\Rightarrow 5 \ast E \ast E$$

$$\Rightarrow 5 \ast E \ast E$$

$$\Rightarrow 5 \ast (E \ast E)$$

We say that $\ast$ *associates to the right*.  

---

**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*.
- $\ast$ associates to the right:
  
  $$2 \ast 3 \ast 4 = 2^{(3 \ast 4)}.$$

---

**Top-Down Backtracking Parser…**

- 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:

---

**Ambiguous Grammars**

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

$$E ::= E \ast E \mid E + E \mid \text{number}$$

$$E \Rightarrow E \ast E$$

$$\Rightarrow 5 \ast E \ast E$$

$$\Rightarrow 5 \ast E \ast E$$

$$\Rightarrow 5 \ast (E \ast E)$$

We say that $\ast$ *associates to the right*.  

---

**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*.
- $\ast$ associates to the right:
  
  $$2 \ast 3 \ast 4 = 2^{(3 \ast 4)}.$$

---

**Top-Down Backtracking Parser…**

- 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:

---

**Ambiguous Grammars**

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

$$E ::= E \ast E \mid E + E \mid \text{number}$$

$$E \Rightarrow E \ast E$$

$$\Rightarrow 5 \ast E \ast E$$

$$\Rightarrow 5 \ast E \ast E$$

$$\Rightarrow 5 \ast (E \ast E)$$

We say that $\ast$ *associates to the right*.  

---

**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*.
- $\ast$ associates to the right:
  
  $$2 \ast 3 \ast 4 = 2^{(3 \ast 4)}.$$

---

**Top-Down Backtracking Parser…**

- 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:

---

**Ambiguous Grammars**

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

$$E ::= E \ast E \mid E + E \mid \text{number}$$

$$E \Rightarrow E \ast E$$

$$\Rightarrow 5 \ast E \ast E$$

$$\Rightarrow 5 \ast E \ast E$$

$$\Rightarrow 5 \ast (E \ast E)$$

We say that $\ast$ *associates to the right*.  

---

**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*.
- $\ast$ associates to the right:
  
  $$2 \ast 3 \ast 4 = 2^{(3 \ast 4)}.$$

---

**Top-Down Backtracking Parser…**

- 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:

---

**Ambiguous Grammars**

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

$$E ::= E \ast E \mid E + E \mid \text{number}$$

$$E \Rightarrow E \ast E$$

$$\Rightarrow 5 \ast E \ast E$$

$$\Rightarrow 5 \ast E \ast E$$

$$\Rightarrow 5 \ast (E \ast E)$$

We say that $\ast$ *associates to the right*.  

---
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.

Resulting Expression Grammar:

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

Recursive Descent Parsing

```plaintext
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;
PROCEDURE E ();
  IF curr_tok = id THEN match(id);
  IF curr_tok = num THEN match(num);
  ELSE E(); match(+); E();
  ENDIF;
```

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:
     \( p_i ::= p_i \ op \ p_{i+1} | p_{i+1} \)
   - 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:
   \( p_{n+1} ::= ( p_1 ) | \text{num} | \text{id} \)
Recursive Descent—Small Problem 3

What if there are several possible “next” tokens:

\[
\begin{align*}
\text{prog} & \rightarrow \text{decl} \mid \text{stat} \\
\text{stat} & \rightarrow \text{id} \mid \text{id()} \mid \text{while} \\
\text{decl} & \rightarrow \text{id} \mid \text{real id}
\end{align*}
\]

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;

Recursive Descent—Small Problem 1

We may loop forever:

PROCEDURE E ()
  IF ...
  ELSE E(); match(+); E();
  ...

Recursive Descent—Small Problem 2

What about productions that start out similarly:

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

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 ... ENDIF

Left Recursion Removal

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

\[
\begin{align*}
A & \rightarrow A\alpha \mid \beta \\
\Rightarrow A & \rightarrow \beta R \\
R & \rightarrow \alpha R \mid \varepsilon
\end{align*}
\]

Example:

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr } \pm \text{ term } \mid \text{term} \\
\downarrow \\
\text{expr} & \rightarrow \text{term } R \\
R & \rightarrow \pm \text{ term } R \mid \varepsilon
\end{align*}
\]
Left Recursion Removal...

- After left recursion removal, our expression grammar

\[
E \rightarrow E + T | T \\
T \rightarrow T * F | F \\
F \rightarrow (E) | id
\]

turns into

\[
E \rightarrow T E' \\
E' \rightarrow + T E' | \epsilon \\
T \rightarrow F T' \\
T' \rightarrow * F T' | \epsilon \\
F \rightarrow (E) | id
\]

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.

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

Example:

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