Compilers
Introduction to Parsing

Syntax
CodeGen
Semantics
Types
• Regular languages
  – The weakest formal languages widely used
  – Many applications
Consider the language:

\[ \{(i)^i \mid i \geq 0\} \]
What can regular languages express?
What can regular languages express?

• Languages requiring counting modulo a fixed integer

• Intuition: A finite automaton that runs long enough must repeat states

• Finite automaton can’t remember # of times it has visited a particular state
• **Input:** sequence of tokens from lexer

• **Output:** parse tree of the program
  (But some parsers never produce a parse tree ...)

----

Alex Aiken
Intro to Parsing

- Cool
  
  if \( x=y \) then 1 else 2 fi

- Parser input

  \[
  \text{IF } \text{ID} = \text{ID} \text{ THEN INT ELSE INT FI}
  \]

- Parser output

  \[
  = \text{IF-THEN-ELSE} \]

  \[
  \text{INT} \quad \text{INT}
  \]

  \[
  \text{ID} \quad \text{ID}
  \]
<table>
<thead>
<tr>
<th>Phase</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexer</td>
<td>String of characters</td>
<td>String of tokens</td>
</tr>
<tr>
<td>Parser</td>
<td>String of tokens</td>
<td>Parse tree (may be implicit)</td>
</tr>
</tbody>
</table>

- Some compilers combine Lexer and Parser
Compilers

Syntax
CodeGen
Semantics
Types

Context-Free Grammars
• Not all strings of tokens are programs . . .
• . . . parser must distinguish between valid and invalid strings of tokens

• We need
  - A language for describing valid strings of tokens
  - A method for distinguishing valid from invalid strings of tokens
• Programming language constructs have recursive structure

• An `EXPR` is
  ```
  if `EXPR` then `EXPR` else `EXPR` fi
  while `EXPR` loop `EXPR` pool
  ...
  ```

• Context-free grammars are a natural notation for this recursive structure
• A CFG consists of
  – A set of terminals $T$
  – A set of non-terminals $N$
  – A start symbol $S$ (a non-terminal)
  – A set of productions

$$X \rightarrow Y_1 Y_2 \ldots Y_n$$
where $X \in N$ and $Y_i \in T \cup N \cup \{\varepsilon\}$
• In these lecture notes
  - Non-terminals are written upper-case
  - Terminals are written lower-case
  - The start symbol is the left-hand side of the first production
Consider the language:

\[ \{(i)^i \mid i \geq 0\} \]

\[
\begin{align*}
S & \rightarrow (S) & N = \{S\} \\
S & \rightarrow \varepsilon & T = \{(,\}\}
\end{align*}
\]
Read productions as rules:

\[ X \rightarrow Y_1 \ldots Y_n \]

means \( X \) can be replaced by \( Y_1 \ldots Y_n \)
1. Begin with a string with only the start symbol $S$

2. Replace any non-terminal $X$ in the string by the right-hand side of some production
   \[ X \rightarrow Y_1 \ldots Y_n \]

3. Repeat (2) until there are no non-terminals in the string
More formally, write

\[ X_1 \ldots X_{i-1} X_i X_{i+1} \ldots X_n \rightarrow X_1 \ldots X_{i-1} Y_1 \ldots Y_m X_{i+1} \ldots X_n \]

if there is a production

\[ X_i \rightarrow Y_1 \ldots Y_m \]
Write

\[ X_1 \ldots X_n \rightarrow^* Y_1 \ldots Y_m \]

If

\[ X_1 \ldots X_n \rightarrow \ldots \rightarrow \ldots \rightarrow Y_1 \ldots Y_m \]

In 0 or more steps
Let $G$ be a context-free grammar with start symbol $S$. Then the language of $G$ is:

$$L(G) = \{ a_1 \ldots a_n \mid S \rightarrow^* a_1 \ldots a_n \text{ and every } a_i \text{ is a terminal} \}$$
Let $G$ be a context-free grammar with start symbol $S$. Then Sentential Forms of $G$ are:

$$SF(G) = \{ \alpha \mid S \rightarrow^* \alpha \text{ and } \alpha \in (T \cup N)^* \}$$

Therefore:

$$L(G) \subset SF(G)$$
• Terminals are so-called because there are no rules for replacing them

• Once generated, terminals are permanent

• Terminals ought to be tokens of the language

CFGs
A fragment of COOL

EXPR → if EXPR then EXPR else EXPR fi
| while EXPR loop EXPR pool
| id
Some elements of the language:

id
if id then id else id fi
while id loop id pool
if while id loop id pool then id else id
if if id then id else id fi then id else id fi
Simple arithmetic expressions

$$E \rightarrow E+E \mid E \times E \mid (E) \mid \text{id}$$

Some elements of the language:

- id
- id + id
- (id)
- id \times id
- (id) \times id
- id \times (id)
Which of the strings are in the language of the given CFG?

- □ abcba
- □ acca
- □ aba
- □ abcba
Which of the strings are in the language of the given CFG?

- abcba
- acca
- aba
- abcbcba

CFGs:

- \( S \rightarrow aXa \)
- \( X \rightarrow \varepsilon \)  
  |   \( bY \)
- \( Y \rightarrow \varepsilon \)  
  |   \( cXc \)
The idea of a CFG is a big step. But:

- Membership in a language is “yes” or “no”;
  - We also need a parse tree of the input

- Must handle errors gracefully

- Need an implementation of CFG’s (e.g., Bison)
• Form of the grammar is important
  - Many grammars generate the same language
  - Tools are sensitive to the grammar
Compilers

Syntax
CodeGen
Semantics
Types

Derivations
A derivation is a sequence of productions

\[ S \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \]

A derivation can be drawn as a tree

- Start symbol is the tree’s root
- For a production \( X \rightarrow Y_1\ldots Y_n \) add children \( Y_1\ldots Y_n \) to node \( X \)
• Grammar

\[ E \rightarrow E + E \mid E \ast E \mid (E) \mid id \]

• String

\[ id \ast id + id \]
Derivations

\[
E \\
\rightarrow E + E \\
\rightarrow E \ast E + E \\
\rightarrow id \ast E + E \\
\rightarrow id \ast id + E \\
\rightarrow id \ast id + id
\]
Derivations
Derivations

\[ E \rightarrow E + E \]
Derivations

E

→ E + E

→ E * E + E

E

E + E

E * E

E

E

E

Alex Aiken
Derivations

\[
\begin{align*}
E & \rightarrow E + E \\
E & \rightarrow E \ast E + E \\
E & \rightarrow id \ast E + E
\end{align*}
\]
Derivations

E
→ E + E
→ E * E + E
→ id * E + E
→ id * id + E

E
+ 
E

E
* 
E

id

id

id

Alex Aiken
Derivations

\[ E \quad \rightarrow \quad E + E \]
\[ \quad \rightarrow \quad E * E + E \]
\[ \quad \rightarrow \quad \text{id} * E + E \]
\[ \quad \rightarrow \quad \text{id} * \text{id} + E \]
\[ \quad \rightarrow \quad \text{id} * \text{id} + \text{id} \]
• A parse tree has
  – Terminals at the leaves
  – Non-terminals at the interior nodes

• An in-order traversal of the leaves is the original input

• The parse tree shows the association of operations, the input string does not
• The example is a *left-most* derivation
  - At each step, replace the left-most non-terminal

• There is an equivalent notion of a *right-most* derivation

\[
\begin{align*}
E & \\
\rightarrow & E+E \\
\rightarrow & E+id \\
\rightarrow & E*E + id \\
\rightarrow & E*id + id \\
\rightarrow & id*id + id
\end{align*}
\]
Derivations
$E \rightarrow E + E$
Derivations

\[ E \rightarrow E + E \]
\[ E \rightarrow E + \text{id} \]
Derivations

\[ E \rightarrow E + E \]
\[ E \rightarrow E + \text{id} \]
\[ E \rightarrow E \ast E + \text{id} \]
Derivations

\[
\begin{align*}
E & \rightarrow E + E \\
& \rightarrow E + \text{id} \\
& \rightarrow E * E + \text{id} \\
& \rightarrow E * \text{id} + \text{id}
\end{align*}
\]
Derivations

E
→ E + E
→ E + id
→ E * E + id
→ E * id + id
→ id * id + id

E
  +
  |
E
  *
  |
E
    |
    |
  id
    |
    |
id
• Note that right-most and left-most derivations have the same parse tree

• The difference is the order in which branches are added
Which of the following is a valid derivation of the given grammar?

Derivations:

- **S** → aXa
- **X** → ε | bY
- **Y** → ε | cXc | d

Possible derivations:

- S
  - aXa
  - abYa
  - acXca
  - acca

- S
  - aa

- S
  - aXa
  - abYa
  - abcXca
  - abcYca
  - abcbYca
  - abcbdca

- S
  - aXa
  - abYa
  - abcXcda
  - abccda
Which of the following is a valid derivation of the given grammar?

**Derivations**

- **S → aXa**
- **X → ε | bY**
- **Y → ε | cXc | d**

- S
  - aXa
  - abYa
  - acXca
  - acca

- S
  - aXa
  - abYa
  - abcXca
  - abcYca
  - abcbdca

- S
  - aXa
  - abYa
  - abcXca
  - abcYca
  - abcXcda
  - abccda

- S
  - aa
Which of the following is a valid parse tree for the given grammar?

- $S \rightarrow aXa$
- $X \rightarrow \epsilon \mid bY$
- $Y \rightarrow \epsilon \mid cXc \mid d$

Diagram:

- $S$ (root) with children $a$, $X$, and $a$.
  - $X$ with children $b$, $Y$, $a$, and $d$.
  - $Y$ with children $c$, $X$, $c$, and $d$.

Options:

1. Diagram 1
2. Diagram 2
3. Diagram 3
4. Diagram 4
Which of the following is a valid parse tree for the given grammar?

- $S \rightarrow aXa$
- $X \rightarrow \epsilon \mid bY$
- $Y \rightarrow \epsilon \mid cXc \mid d$

Derivations:

1. $S \rightarrow aXa \rightarrow a\epsilon a \rightarrow aXa$
2. $S \rightarrow aXa \rightarrow a\epsilon a \rightarrow a\epsilon a$
3. $S \rightarrow aXa \rightarrow a\epsilon a \rightarrow a\epsilon a$
4. $S \rightarrow aXa \rightarrow a\epsilon a \rightarrow a\epsilon a$

The valid parse tree is the second one.
• We are not just interested in whether $s \in L(G)$
  – We need a parse tree for $s$

• A derivation defines a parse tree
  – But one parse tree may have many derivations

• Left-most and right-most derivations are important in parser implementation
Compilers

Ambiguity

Syntax
CodeGen
Semantics
Types
• Grammar

\[ E \rightarrow E + E \mid E \ast E \mid (E) \mid \text{id} \]

• String

\[ \text{id} \ast \text{id} + \text{id} \]
This string has two parse trees

```
E
  / \  \
E + E
  /   \
E * E id
   /   \
 id   id

E
  / \  \
E * E
  /   \
 id E + E
   /   \
 id   id
```
A grammar is *ambiguous* if it has more than one parse tree for some string
- Equivalently, there is more than one right-most or left-most derivation for some string

Ambiguity is **BAD**
- Leaves meaning of some programs ill-defined
Which of the following grammars are ambiguous?

- $S \rightarrow SS | a | b$
- $E \rightarrow E + E | id$
- $S \rightarrow Sa | Sb$
- $E \rightarrow E' | E' + E$
  $E' \rightarrow -E' | id | (E)$
Which of the following grammars are ambiguous?

- $S \rightarrow SS \mid a \mid b$
- $E \rightarrow E + E \mid id$
- $S \rightarrow Sa \mid Sb$
- $E \rightarrow E' \mid E' + E$
  
  $E' \rightarrow -E' \mid id \mid (E)$
There are several ways to handle ambiguity.

Most direct method is to rewrite grammar unambiguously:

\[
E \rightarrow E' + E \mid E' \\
E' \rightarrow \text{id} \ast E' \mid \text{id} \mid (E) \ast E' \mid (E)
\]

Enforces precedence of \(*\) over \(+\).
Ambiguity

```
E
  /\  \\
E + E
  |  |
E * E id
  |  |
id id
```

```
E
  /\  \\
E * E
  |  |
id id
```

```
E
  /\  \\
E * E
  |  |
  |  |
  |  |
id E + E
  |  |
id id
```

```
E
  /\  \\
  |  |
id id
```

Alex Aiken
Ambiguity

\[
E \rightarrow \ E + \ E \\
E \bullet \ E \ id
\]

\[
E \rightarrow \ E \ * \ E \\
\mid \id \ E \ + \ E
\]

\[
E \rightarrow \ E \ * \ E \\
\mid \id \ E \ + \ E
\]

\[
\id \ E \ + \ E
\]
• Consider the grammar

\[ E \rightarrow \text{if } E \text{ then } E \]

\[
| \text{if } E \text{ then } E \text{ else } E \\
| \text{OTHER}
\]
• The expression

if \( E_1 \) then if \( E_2 \) then \( E_3 \) else \( E_4 \)

has two parse trees
else matches the closest unmatched then

E → MIF /* all then are matched */
   | UIF /* some then is unmatched */

MIF → if E then MIF else MIF
     | OTHER

UIF → if E then E
     | if E then MIF else UIF
• The expression \( \text{if } E_1 \text{ then if } E_2 \text{ then } E_3 \text{ else } E_4 \)
• The expression if \( E_1 \) then if \( E_2 \) then \( E_3 \) else \( E_4 \)
Choose the unambiguous version of the given ambiguous grammar: \( S \rightarrow SS \mid a \mid b \mid \varepsilon \)

1. \( S \rightarrow Sa \mid Sb \mid \varepsilon \)
2. \( S \rightarrow SS' \)
   \( S' \rightarrow a \mid b \)
3. \( S \rightarrow S \mid S' \)
   \( S' \rightarrow a \mid b \)
4. \( S \rightarrow Sa \mid Sb \)
Choose the unambiguous version of the given ambiguous grammar: \( S \rightarrow SS | a | b | \varepsilon \)

- \( S \rightarrow S \rightarrow S \mid S' \mid \varepsilon \)
  - \( S \rightarrow S \mid S' \mid \varepsilon \)
  - \( S' \rightarrow a | b \)
  - \( S \rightarrow S \mid S' \mid \varepsilon \)
  - \( S \rightarrow S \mid S' \mid \varepsilon \)
  - \( S \rightarrow S \mid S' \mid \varepsilon \)
• Impossible to convert automatically an ambiguous grammar to an unambiguous one

• Used with care, ambiguity can simplify the grammar
  - Sometimes allows more natural definitions
  - We need disambiguation mechanisms
• Instead of rewriting the grammar
  - Use the more natural (ambiguous) grammar
  - Along with disambiguating declarations

• Most tools allow precedence and associativity declarations to disambiguate grammars
• Consider the grammar \( E \rightarrow E + E \mid \text{int} \)
• Ambiguous: two parse trees of \( \text{int} + \text{int} + \text{int} \)

- Left associativity declaration: \%left +
• Consider the grammar \[ E \rightarrow E + E \mid \text{int} \]
• Ambiguous: two parse trees of \( \text{int} + \text{int} + \text{int} \)

\[
\begin{array}{c}
E \\
\quad E + E \\
\quad \quad E + E \\
\quad \quad \quad \text{int} \\
\quad \quad \quad \quad \text{int} \\
\quad \quad \quad \quad \quad \text{int}
\end{array}
\]

\[
\begin{array}{c}
E \\
\quad E + E \\
\quad \quad E + E \\
\quad \quad \quad \text{int} \\
\quad \quad \quad \quad \text{int} \\
\quad \quad \quad \quad \quad \text{int}
\end{array}
\]

• \textbf{Left associativity declaration: }\%\texttt{left} +
• Consider the grammar
  – And the string `int + int * int`

```
E  \rightarrow  E + E | E * E | int
```

• Precedence declarations: 
  `%left +`
  `%left *`

```
E
  +
  E
  *
  E
E
  +
  E
  *
  E
```

Alex Aiken
• **Consider the grammar**
  – And the string `int + int * int`

```
E → E+E | E*E | int
```

• **Precedence declarations:**
  - `%left +`
  - `%left *`