Outline

- Limitations of Regular Languages
- Parser Overview
- Context-free Grammars (CFGs)
- Derivations
- Ambiguity

Regular Languages

- Last time, we saw that regular languages are very useful for partitioning input into tokens
- But regular languages are not expressive enough to turn a stream of tokens into structure
- For this, we need a more expressive formal language

Beyond Regular Languages

- Many languages are not regular
- Classic Example: Strings of balanced parenthesis:
  \[
  \{(i)^j \mid i \geq 0\}
  \]
- Question: Why is there no automata that can recognize this language?

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 cannot remember the number of times it has visited a particular state

Side Note: Comments in L

- Recall: Comments in L start with (*, end with *) and can be nested
- Also Recall: Comments are removed during lexing
- Question: Are comments in L a regular language?
The Functionality of the Parser

- **Input**: sequence of tokens from the lexer
- **Output**: parse tree of the program

Example

Consider the following L expression:
if x<>y then 1 else 2

Parse Input:
TOKEN_IF TOKEN_ID("x") TOKEN_NEQ TOKEN_ID("y") TOKEN_THEN TOKEN_INT(1) TOKEN_ELSE TOKEN_INT(2)

Parser Output:

```
NEQ      INT :1      INT :2
ID:x     ID:y
```

 Parsing vs. Lexing

<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</td>
</tr>
</tbody>
</table>

The Role of the Parser

- 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 recognizing if a string of tokens is in this language or not

Context-free Grammars (CFGs)

- Programming language constructs have **recursive** structure
- **Example**: An L expression is expression + expression, if expression then expression else expression, ...
- Context free grammars are a natural notation for this recursive structure

CFGs in more detail

- A CFG consists of:
  - A set of terminals T
  - A set of non-terminals N
  - A start symbol S (non-terminal)
  - A set of productions

\[ X \rightarrow Y_1 Y_2 \ldots Y_n \]

where \( X \in N \) and \( Y_i \in (T \cup N \cup \{\varepsilon}\)
Notational Conventions in this Class

- Non-terminals are always written upper-case
- Terminals are written lower-case
- The start symbol is the left-hand side of the first production

CFG Examples

- A fragment of L
  EXPR → if EXPR then EXPR else EXPR
  | EXPR + EXPR
  | id

CFG Examples continued

- Simple arithmetic expressions:
  EXPR → E * E
  | E + E
  | (E)
  | id

The Language of a CFG

- Recall production rules: X → Y₁ . . . Yₙ
  - Means that X can be replaced by Y₁ . . . Yₙ
  - More specifically:
    1. Begin with string consisting of the start symbol S
    2. Replace any non-terminal X in string with the right-hand side of some production
       X → Y₁ . . . Yₙ
    3. Repeat (2) until there are no non-terminals in the string

The Language of a CFG continued

- More formally, write
  \[ X₁ \ldots Xₖ \ldots Xₙ → X₁ \ldots Xₖ⁻¹ Y₁ \ldots Yₙ Xₖ₊₁ \ldots Xₙ \]
  if there is a production
  \[ Xᵢ → Y₁ \ldots Yₙ \]
- Abbreviation: Write \[ X₁ \ldots Xₙ →^* Y₁ \ldots Yₙ \] if
  \[ X₁ \ldots Xₙ → \ldots → Y₁ \ldots Yₙ \] in 0 or more steps

The Language of a CFG continued

- Now, let G be a context-free grammar with start symbol S.
  Then the language of G is:
  \[ \{ a₁ \ldots aₙ | S →^{*} a₁ \ldots aₙ \text{ and every } aᵢ \text{ is a terminal} \} \]
Terminals

- Terminals are called "terminals" because there are no rules for replacing them.
- Once generated, terminals are permanent.
- Question: What should terminals be when parsing a programming language?
- Answer: Tokens.

Examples

- $L(G)$ is the language of CFG $G$.
- Strings of balanced parentheses:
  $$\{(i^j|i \geq 0}\}
  $\begin{align*}
  & S \rightarrow (S) \\
  & S \rightarrow \varepsilon
  
  \text{or equivalently}
  & S \rightarrow (S) | \varepsilon
  \end{align*}$

- Recall the earlier fragment of $L$:
  $$\begin{align*}
  \text{EXPR} & \rightarrow \text{if} \ \text{EXPR} \ \text{then} \ \text{EXPR} \ \text{else} \ \text{EXPR} \\
  & \mid \text{EXPR} + \text{EXPR} \\
  & \mid \text{id}
  \end{align*}$$
- Some strings in this language:
  $$\begin{align*}
  & \text{id} \\
  & \text{IF} \ \text{id} \ \text{THEN} \ \text{id} \ \text{ELSE} \ \text{id} \\
  & \text{id} + \text{id} \\
  & \text{IF} \ \text{id} \ \text{THEN} \ \text{id} \ \text{ELSE} \ \text{IF} \ \text{THEN} \ \text{id} \ \text{ELSE} \ \text{id}
  \end{align*}$$

- Recall simple arithmetic expressions:
  $$\begin{align*}
  \text{EXPR} & \rightarrow \text{E} \ \ast \ \text{E} \\
  & \mid \text{E} + \text{E} \\
  & \mid (\text{E}) \\
  & \mid \text{id}
  \end{align*}$$
- Some strings in this language:
  $$\begin{align*}
  & \text{id} \\
  & (\text{id}) \\
  & (\text{id}) \ast \text{id} \\
  & \text{id} + \text{id} \\
  & \text{id} \ast \text{id} \\
  & \text{id} \ast (\text{id})
  \end{align*}$$

Where are we?

- The idea of a CFG is a big step towards parsing tokens.
- But we don’t just want to know if a string of tokens is in a language, we also need parse tree of input tokens.
- Must also handle errors gracefully.
- Need an implementation of CFGs (e.g., bison).

From Derivations to Parse Trees

- A derivation is a sequence of productions:
  $$S \rightarrow \ldots \rightarrow \ldots$$
- 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 $X$. 


4
Derivation Example

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

Derivation in Detail

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

Notes on Derivations

- A parse tree has terminals at the leaves and non-terminals at the interior nodes.
- An in-order traversal of the leaves is the original input.
- The parse tree shows the associativity of operations, the input token string does not.
- Example: The parse tree from the last slide encodes that times has higher precedence than plus.

Left-most and Right-most Derivations

- The example we looked at is a left-most derivation.
- This means: At each step, we replace the left-most non-terminal.
- There is also an equivalent notion of right-most derivation.

Right-most Derivation in Detail

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

Derivations and Parse Trees

- Observe that left-most and right-most derivations have the same parse tree.
- The only difference is the order in which branches are added.
- But when parsing tokens, we only care about the final parse tree, which may have many different derivations.
- Left-most and right-most derivations are important in parser implementations.
Recall our example grammar:
\[
\text{EXPR} \rightarrow E \ast E \\
\quad | E + E \\
\quad | (E) \\
\quad | \text{id}
\]
Now, consider the string \text{id\ast id+id}.

This string has two parse trees!

A grammar is ambiguous if it has more than one parse tree for some string.
Equivalently: There is more than one left-most or right-most derivation for some string.
Ambiguity is bad!
Leaves meaning of programs ill-defined.

However, converting grammars to unambiguous form can be very difficult.
It also often results in horrible, unintuitive grammars with many non-terminals.
It is also fundamentally impossible to transform an ambiguous grammar into an unambiguous grammar.
For this reason, tools such as \text{bison} include disambiguation mechanisms.

First method: Rewrite grammar unambiguously.
Question: How can we write simple arithmetic expressions unambiguously?
Solution: Enforce precedence of times over plus by generating all pluses first:
\[
S \rightarrow E + S | E \\
E \rightarrow \text{id} \ast E | \text{id} | (S) + E | (S)
\]

Instead of rewriting the grammar:
Use the more natural ambiguous grammar.
Along with disambiguating declarations.
The parser tool \text{bison} allows you to declare precedence and associativity for this.

Precedence and Associativity

Dealing with Ambiguity

Precedence and Associativity

Ambiguity

Ambiguity

Ambiguity

Ambiguity
Associativity Declarations

- Consider the grammar $E \to E + E \mid id$
- Ambiguous: Two parse trees of input $id + id + id$

- Declare left associativity of plus as: %left +

Precedence Declarations

- Consider the grammar $E \to E + E \mid id$ and input $id + id * id$

- Precedence Declaration: %left + %left *

Conclusion

- We have seen how to specify programming language syntax with CFGs
- We built parse trees that express the high-level syntactic structure
- Parse trees of programs are known as abstract syntax trees
- We discussed ambiguity of CFGs