Introduction to Parsing

CPSC 310 - Programming Languages

Outline

- Regular languages revisited
- Parser overview
- Context-free grammars (CFGs)
- Derivations
- Ambiguity
- Syntax errors

Languages and Automata

- Formal languages are important in computer science, especially in programming languages.
- Regular languages are the weakest formal languages that are widely used
- We also need to study context-free languages

Limitations of Regular Languages

- Intuition: A finite automaton that runs long enough must repeat states
- A finite automaton cannot remember the number of times it has visited a particular state
- A finite automaton has finite memory, so:
  - it can only store which state it is currently in, and
  - cannot count, except up to a finite limit.
- Example, the language of balanced parentheses is not regular: $\{(^i)^i \mid i \geq 0\}$

The Role of the Parser

- The parsing phase of a compiler can be thought of as a function:
  - Input: sequence of tokens from the lexer
  - Output: parse tree of the program
- Not all sequences of tokens are programs, so a parser must distinguish between valid and invalid sequences of tokens
- So, we need
  - a language for describing valid sequences of tokens, and
  - a method for distinguishing valid from invalid sequences of tokens.

Context-Free Grammars

- Many programming language constructs have a recursive structure
- Example, a statement is of the form:
  - if condition then statement else statement, or
  - while condition do statement, or
  - . . .
- Context-free grammars (CFGs) are a natural notation for this recursive structure

Context-Free Grammars Definition

- A context-free grammar is a 4-tuple (T, N, P, S) where:
  - T – alphabet (finite set of symbols or terminals)
  - N – a finite, nonempty set of nonterminal symbols
  - P – a set of productions of the form; assuming that $X \in N$, productions are of the form
    - $X \to \epsilon$, or
    - $X \to Y_1 Y_2 \ldots Y_n$ where $Y_i \in N \cup T$ N)*$
  - $S \in N$ – the start symbol

Notational Conventions

- In these lecture notes
  - Non-terminals are written in uppercase
  - Terminals are written in lowercase
  - The start symbol is the left-hand side of the first production

CFG Example

- A fragment of a simple language

$$STMT \rightarrow if\ COND\ then\ STMT\ else\ STMT$$
$$STMT \rightarrow while\ COND\ do\ STMT$$
$$STMT \rightarrow\ id\ =\ int$$

- Notational abbreviation

$$\begin{aligned} STMT \rightarrow\ & if\ COND\ then\ STMT\ else\ STMT \\ \mid\ & while\ COND\ do\ STMT \\ \mid\ & id\ =\ int \end{aligned}$$

CFG Example

- Classic CFG example: simple arithmetic expressions

$$
\begin{aligned}
E \rightarrow\ & E * E \\
\mid\ & E + E \\
\mid\ & (E) \\
\mid\ & id
\end{aligned}
$$

The Language of a CFG

- Productions can be read as replacement rules
- $X \to Y_1 \dots Y_n$ means that X can be replaced by $Y_1 \dots Y_n$
- $X \to \epsilon$ means that X can be erased (replaced with the empty string)

The Language of a CFG: Key Idea

1. Begin with a string consisting of the start symbol S
2. Replace any non-terminal X in the string by a right-hand side of some production $X \to Y_1 \ldots Y_n$
3. Repeat step 2 until there are no non-terminals in the string

The Language of a CFG

- Let G be a context-free grammar with start symbol S. Then the language of G ($L(G)$) is:

$$\{a_1 \dots a_n \mid S \overset{*}{\rightarrow} a_1 \dots a_n \wedge \mathit{every}\ a_i \in T\}$$

  where

$$X_1 \dots X_n \overset{*}{\rightarrow} Y_1 \dots Y_m$$

  denotes

$$X_1 \dots X_n \rightarrow \dots \rightarrow Y_1 \dots Y_m$$

Terminals

- A terminal has no rules for replacing it, hence the name terminal
- Once a terminal is generated, it is permanent
- Terminals ought to be the tokens of the language

Parentheses Example

- Strings of balanced parentheses $\{(^i)^i \mid i \geq 0\}$
- Grammar

$$
\begin{aligned}
S &\rightarrow (S) \\
&\mid \epsilon
\end{aligned}
$$

Example

- A fragment of a simple language

$$
\begin{aligned}
STMT &\rightarrow \mathit{if}\ COND\ \mathit{then}\ STMT\ \mathit{else}\ STMT \\
&\mid \mathit{while}\ COND\ \mathit{do}\ STMT \\
&\mid id\ =\ int \\
COND &\rightarrow (id == id) \\
&\mid (id != id)
\end{aligned}
$$

Example Continued

- Some elements of the language
    - `id = int`
    - `if (id == id) then id = int else id = int`
    - `while (id != id) do id = int`
    - `while (id == id) do while (id != id) do id = int`

Arithmetic Example

- Simple arithmetic expressions:

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

- Some elements of the language
  - id
  - (id)
  - (id) * id
  - id $+$ id

Designing Grammars

- Use recursive productions to generate an arbitrary number of symbols

$$A \to xA \,|\epsilon \quad // \text{ Language: } x*$$
$$A \to yA \,|y \quad // \text{ Language: } y+$$

- Use separate non-terminals to generate disjoint parts of a language, and then combine in a production

$$S \to AB // \text{ Language: } a * b*$$
$$A \to aA \,|\epsilon$$
$$B \to bB \,|\epsilon$$

Designing Grammars

- To generate languages with matching, balanced, or related numbers of symbols, write productions which generate strings from the middle

$$S \rightarrow aAb \quad // \text{ Language:}\{a^n b^n \mid n \leq 0\}$$
$$S \rightarrow aAbb \quad // \text{ Language:}\{a^n b^{2n} \mid n \leq 0\}$$

- For a language that is the union of other languages, use separate non-terminals for each part of the union and then combine

$$S \rightarrow T \mid V \quad // \text{ Language: union of T and V}$$
$$T \rightarrow aTb \mid U \quad // \text{ Language:}\{a^n b^m \mid m \geq n \geq 0\}$$
$$U \rightarrow Ub \mid b$$
$$V \rightarrow aVc \mid W \quad // \text{ Language:}\{a^n c^m \mid m \geq n \geq 0\}$$
$$W \rightarrow Wc \mid c$$

Notes

- The idea of a CFG is a big step
- But,
  - Membership in a language is boolean; we also need the parse tree of the input
  - Must handle errors gracefully
  - Need an implementation of CFGs
- Form of the grammar is important
  - Many grammars generate the same language
  - Parsing tools are sensitive to the grammar

Derivations and Parse Trees

- A derivation is a sequence of productions

$$S \to \ldots \to \ldots \to \ldots$$

- A derivation can be depicted as a tree
  - The start symbol is the tree's root
  - For a production $X \to Y_1 \ldots Y_n$ add children $Y_1 \ldots Y_n$ to node X

Derivation Example

- Simple arithmetic expressions:

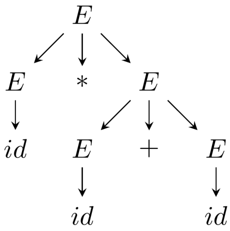
$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

- String

$$id * id + id$$

Derivation Example

$$
\begin{aligned}
& E \\
\rightarrow & E + E \\
\rightarrow & E * E + E \\
\rightarrow & id * E + E \\
\rightarrow & id * id + E \\
\rightarrow & id * id + id
\end{aligned}
$$



Notes on Derivations

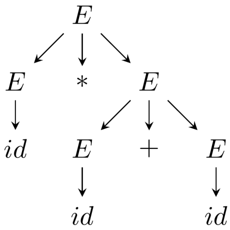
- A parse tree has:
  - terminals at the leaves, and
  - non-terminals at the interior nodes
- An in-order traversal of the leaves is the original input
- The parse tree shows the association of the operations, the input string does not

Left-most and Right-most Derivations

- The previous example was a left-most derivation
  - At each step, replace the left-most non-terminal
- There is an equivalent notion of a right-most derivation
  - At each step, replace the right-most non-terminal

Right-most Derivation Example

$$
\begin{aligned}
& E \\
\rightarrow & E + E \\
\rightarrow & E + id \\
\rightarrow & E * E + id \\
\rightarrow & E * id + id \\
\rightarrow & id * id + id
\end{aligned}
$$



Derivations and Parse Trees

- Note that right-most and left-most derivations have the same parse tree
- The difference is the order in which branches are added

Summary of Derivations

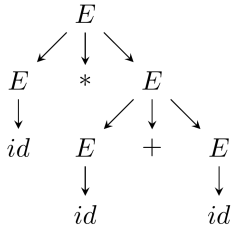
- We are not only interested in whether $S \in L(G)$, we also need a parse tree for S
- A derivation defines a parse tree, but one parse tree may have many derivations
- Left-most and right-most derivations are important in the parser implementation

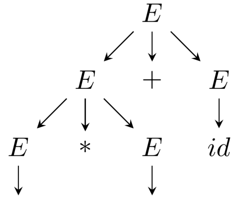
Ambiguity

- Grammar

$$E \to E + E \mid E * E \mid (E) \mid id$$

- The string $id * id + id$ has two parse trees:





Ambiguity

- A grammar is ambiguous if it has more than one parse tree for some string
- Ambiguity leaves the meaning of some programs ill-defined
- Ambiguity is common in programming languages

Dealing with Ambiguity

- There are several ways to handle ambiguity
- The most direct method is to rewrite the grammar unambiguously
- Example: enforcing precedence in the previous grammar

$$
\begin{aligned}
E &\rightarrow T + E \\
&\mid T \\
T &\rightarrow id * T \\
&\mid id \\
&\mid (E)
\end{aligned}
$$

Ambiguity: The Dangling Else

- Consider the following grammar

$$\begin{aligned} S \rightarrow\ & \mathit{if}\ C\ \mathit{then}\ S \\ |\ & \mathit{if}\ C\ \mathit{then}\ S\ \mathit{else}\ S \\ |\ & \mathit{OTHER} \end{aligned}$$

- This grammar is ambiguous: the expression "$\mathit{if}\ C_1\ \mathit{then}\ \mathit{if}\ C_2\ \mathit{then}\ S_3\ \mathit{else}\ S_4$" has two parse trees

The Dangling Else: a Fix

- We want "else" to match the closest unmatched "then"
- We can describe this in the grammar

$$
\begin{aligned}
S &\rightarrow MIF \\
&\mid UIF \\
MIF &\rightarrow \textit{if } C \textit{ then } MIF \textit{ else } MIF \\
&\mid OTHER \\
UIF &\rightarrow \textit{if } C \textit{ then } S \\
&\mid \textit{if } C \textit{ then } MIF \textit{ else } UIF
\end{aligned}
$$

Ambiguity

- No general techniques for handling ambiguity
- Impossible to automatically convert an ambiguous grammar to an unambiguous one
- Used with care, ambiguity can simplify the grammar
    - Sometimes allows more natural definitions
    - but, we need disambiguation mechanisms

Precedence and Associativity Declarations

- 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

Error Handling

- The purpose of the compiler is to
  - detect invalid programs
  - translate valid programs
- Many kinds of possible errors

Error Kind	Detected by
Lexical	Lexer
Syntax	Parser
Semantic	Type Checker
Correctness	Tester/User

Syntax Error Handling

- Error handler should
  - report errors accurately and clearly
  - recover from an error quickly
  - not slow down the compilation of valid programs
- Good error handling is typically difficult to achieve

Approaches to Syntax Error Recovery

- From simple to complex
  - panic mode
  - error productions
  - automatic local or global correction
- Not all are supported by all parser generator tools

Syntax Error Recovery: Panic Mode

- Simplest, most popular method
- When an error is detected:
  - discard tokens until one with a clear role is found
  - continue from there
- Such tokens are called synchronizing tokens and are typically the statement or expression terminators

Syntax Error Recovery: Error Productions

- Idea: specify in the grammar know common mistakes
- Essentially promotes common errors to alternative syntax
- Example
  - Common mistake: write "5 x" instead of "5 * x"
  - Fix: add the production "$E \rightarrow \ldots \mid EE$"
- Disadvantage: this complicates the grammar

Syntax Error Recovery: Past and Present

- Past
  - Slow recompilation cycle (even once a day)
  - Find as many errors in one cycle as possible
  - Researchers could not let go of the topic
- Present
  - Quick recompilation cycle
  - Users tend to correct one error per cycle
  - Complex error recovery is needed less
  - Panic-mode seems good enough in practice