CSC326 Programming Languages
(Week 2, Friday)

Yilan Gu
guyilan@ecf.toronto.edu
http://www.cs.toronto.edu/~yilan/326f09/

Last Time
• Syntax and Semantics
  – Language specification
  – Regular expression
  – BNF

Today
• Syntax and Semantics
  – CFG (continue)

Regular Expressions

Examples:
- $(0 + 1)^*$
- $1^* \ (\cdot + :)^*$
- $(a + b)^*aa(a+b)^*$

Notation:
- **Kleene Closure**: * superscript denotes 0 or more repetitions
- **Positive Closure**: + superscript denotes 1 or more repetitions
- **Alternation**: binary `+` denotes choice. It is also denoted by 1, i.e., $(011)^*$.
- **(`` and ``)**: are used for grouping
- **ε (epsilon)** denotes the empty or "null" string.
- **∅** denotes the (empty) language with no strings.

Review:

Regular expression
Context Free Grammar (Definition)
Regular Expressions: An Example

- What is the regular express for
  a set of all strings over 0 and 1 starting with 00 and ending with a single 0
  a. 00(0+1)*0
  b. 001(0+1)*0
  c. 00(0+1)*10
  d. 00(0+1)*0+00
- How about:
  a set of all strings over 0 and 1 starting with 00 and ending with 0

Definition of CFG

- Grammar:
  A Context Free Grammar is a formalism that describes which sequence of terminals are meaningful in a PL. Mathematically, it is defined as a quadruple \((N, T, P, S)\) where:
  - \(N\) is the set of symbols called Nonterminals
  - \(T\) is the set of symbols called Terminals
  - \(P\) is the set of productions
    - LHS: 1 nonterminal
    - RHS: a sequence of terminals and nonterminals
  - \(S\) is a nonterminal called the starting symbol
- Example:
  \(G = (N,T,P,S)\) where \(N = \{<S>\}\), \(T = \{a,b\}\),
  \(P = \{<S> ::= a<S>, <S> ::= b<S>a\}\)

- Production:
  A production is a rule of the form \(X ::= Y\) where \(X\) is a string of symbols (terminals or nonterminals) containing at least one nonterminal, and \(Y\) is a string of symbols (terminals or nonterminals).

Limitations of Regular Expressions

- Regular expressions are not powerful enough to describe some languages.
  Examples:
  - The language consisting of all strings of one or more a's followed by the same number of b's.
  - The language consisting of strings containing a's, left brackets, and right brackets, such that the brackets match.

  Question: How can we be sure there is no regular expression for these languages?

  Question: Exactly what things can and cannot be expressed with a regular expression?
A Regular Grammar is a CFG $G=(N, T, P, S)$, where:

- $P$ is the set of productions, taking one of the following three forms
  - $A ::= Ca$
  - $A ::= a$
  - $A ::= \varepsilon$

or

- $P$ is the set of productions, taking one of the following three forms
  - $A ::= aC$
  - $A ::= a$
  - $A ::= \varepsilon$

The power of regular grammar is the same as regular expression, i.e.

- For any language that can be generated using a regular grammar, there is a regular expression for the same language.
- For any language that can be represented using a regular expression, there is a regular grammar that can generate the same language.

**Grammar: derivation types & parsers**

- **Leftmost Derivation:**
  - In a leftmost derivation, the replaced nonterminal is always the leftmost nonterminal.

- **Rightmost Derivation:**
  - In a rightmost derivation, the replaced nonterminal is always the rightmost nonterminal.

- Typically, language parsers either do leftmost or rightmost derivation on the grammar
  - LR parser: left-to-right, right most derivation
  - LL parser: left-to-right, left most derivation

**Grammar: definitions...**

- **Sentence**
  - A finite sequence of terminals, constructed according to the rules of the grammar for that PL.

- **Sentential form**
  - A finite sequence of terminals and non-terminals, constructed according to the rules of the grammar for that PL.

- **Derivation**

- **Parse**

**Regular grammar: An Example**

- What is the regular express for
  
  a set of all strings over 0 and 1 starting with 00 and ending with a single 0

  - $00(0+1)^*10$
  - $<S>::=0<A>$
  - $<A>::=0<B>$
  - $<B>::=1<C>$
  - $<C>::=0$
  - $<B>::=1<B>$
  - $<B>::=0<B>$
**Grammar: definitions...**

- **Sentence**
  - A finite sequence of **terminals**, constructed according to the rules of the grammar for that PL.

- **Sentential form**
  - A finite sequence of **terminals and non-terminals**, constructed according to the rules of the grammar for that PL.

- **Derivation**

- **Parse**

**Grammar: derivation**

- **Given the grammar**

  
  \[
  \begin{align*}
  \text{<letter>} & ::= \text{abccldefgihlijklmnopqrstuvwxyz} & \text{[1]} \\
  \text{<digit>} & ::= \text{0123456789} & \text{[2]} \\
  \text{<identifier>} & ::= \text{<letter>} | \text{<identifier>} \text{<letter>} | \text{<identifier>} \text{<digit>} & \text{[3]} \\
  \text{<assign-stmt>} & ::= \text{<identifier>} = 0 & \text{[4]}
  \end{align*}
  \]

- **Can we generate** \text{x2=0} **from these rules? (top-down)**

  \[
  \begin{align*}
  \text{<assign-stmt>} \Rightarrow & \text{<identifier>} = 0 & \text{(using 4)} \\
  \Rightarrow & \text{<identifier>} \text{<digit>} = 0 & \text{(using 3c)} \\
  \Rightarrow & \text{<letter>} \text{<digit>} = 0 & \text{(using 3a)} \\
  \Rightarrow & \text{<identifier>} = 0 & \text{(using 1)} \\
  \Rightarrow & \text{x} \text{<digit>} = 0 & \text{(using 2)} \\
  \Rightarrow & \text{x} & \text{(using 2)}
  \end{align*}
  \]

- **Yes! This is a** derivation of a sentence in the language described by the grammar above. Each sequence in this derivation is a **sentential form**. At each step, the rule indicated is used to substitute the rhs of the rule for the leftmost **non-terminal** in the sentential form.

**Grammar: parsing**

- **Given the grammar**

  \[
  \begin{align*}
  \text{<letter>} & ::= \text{abccldefgihlijklmnopqrstuvwxyz} & \text{[1]} \\
  \text{<digit>} & ::= \text{0123456789} & \text{[2]} \\
  \text{<identifier>} & ::= \text{<letter>} | \text{<identifier>} \text{<letter>} | \text{<identifier>} \text{<digit>} & \text{[3]} \\
  \text{<assign-stmt>} & ::= \text{<identifier>} = 0 & \text{[4]}
  \end{align*}
  \]

- **Can we recognize** \text{x2=0} **from these rules? (bottom-up)**

  \[
  \begin{align*}
  \text{x2=0} \Rightarrow & \text{<letter>} 2 = 0 & \text{(using 1)} \\
  \Rightarrow & \text{<identifier>} 2 = 0 & \text{(using 3a)} \\
  \Rightarrow & \text{<identifier>} \text{<digit>} = 0 & \text{(using 2)} \\
  \Rightarrow & \text{<identifier>} = 0 & \text{(using 3c)} \\
  \Rightarrow & \text{<assign-stmt>}
  \end{align*}
  \]

- **This is a** parse of the sentence \text{x2 = 0}

**Grammar: building parse trees**

- **A parse tree** describes the hierarchical syntactic structure of the sentence based on a given language.

- **In a parse tree**
  - Each internal node is a **non-terminal**, its children are the rhs of a rule for that non-terminal.
  - All leafs are **terminals**.
Grammar: grammars are not unique

- This grammar generates the same language (i.e., set of trees whose frontiers/leafs are the same), but has different parse trees than the previous grammar.

Many grammars can correspond to 1 PL, but only 1 PL should correspond to any useful grammar!

Grammar: ambiguity

- Ambiguity
  - If there are 2 different left-most derivations (or alternatively, right-most derivationa 2 parse trees) for 1 sentence then the grammar is ambiguous
  - This is undecidable

- There is no algorithm which can examine two arbitrary context-free grammars and tell if they generate the same language
  - This is undecidable

Grammar: inherently ambiguous

- Sometimes we can remove an ambiguity from a grammar by restructuring the productions but it is not always possible

- An inherently ambiguous language does not possess an unambiguous grammar

Grammar: sources of ambiguity

- Operator Problem: associativity and precedence
  - E.g.
  - Precedence of multiplication/subtraction/addition/…
  - Solution:
    - Change the grammar to reflect operator precedence
    - $X*Y-Z$ means $((X*Y) - Z)$

- Obscure recursion
  - E.g.
    - Exp → Exp Exp
    - A → A B
  - Solution:
    - ??

- Substructure Problem: extent of a substructure
  - E.g.
    - Dangling else
  - Solution:
    - Coming slides!
Grammar: is this ambiguous?

\[ \begin{align*}
\text{< assign> } & ::= \text{< identifier> = < expression>} & [1] \\
\text{< identifier> } & ::= \text{ABC} & [2] \\
\text{< expression> } & ::= \text{< expression> + < expression>} & [3] \\
& | \text{< expression> - < expression>} & [4] \\
& | (\text{< expression>}) & [5] \\
& | \text{< identifier>} & [6]
\end{align*} \]

Yes, because the sentence \( A = B - C - A \) has two different parse trees.
The grammar does not force "normal" left-to-right evaluation of addition and subtraction.

---

Grammar: If-then-else

- Grammar:
  \[ \begin{align*}
  \text{< if stmt> } & ::= \text{if < logic expression> then < stmt>} & \\
  & | \text{if < logic expression> then < stmt> else < stmt>} & \\
  \text{< stmt> } & ::= \text{if stmt> | ...}
  \end{align*} \]

- Consider: if (logic-expression) then
  
  if (logic-expression) then
    statement 1
  else
    statement 2

---

Grammar: If-then-else

- Grammar:
  \[ \begin{align*}
  \text{< if stmt> } & ::= \text{if < logic expression> then < stmt>} & \\
  & | \text{if < logic expression> then < stmt> else < stmt>} & \\
  \text{< stmt> } & ::= \text{< stmt> | ...}
  \end{align*} \]

- Example: if (x=0) then
  
  if (y = 0) then
    \( z := 1 \)
  else
    \( w := 2 \)

---

**Tree for** \( A = B + C * A \)

**Tree for** \( A = B + C * A \)

**Tree for** \( A = B - C - A \)

**Tree for** \( A = B - C - A \)
Grammar: substructure problem, how to solve ambiguity?

- Use block structure to enclose if statement (e.g. Algol60)
  - E.g. if x = 0 then
    begin
    if y = 0 then
      z := 1
    end
  else
    w := 2

- Use statement begin/end markers (e.g. Algol68)
  - E.g. if x = 0 then
    if y = 0 then
      z := 1
    else
      w := 2

- Change the if statement grammar to disallow parse tree 2; that is, always associate an else with the closest if (e.g. Pascal)

Dealing with Ambiguity

- Can’t always remove an ambiguity from a grammar by restructuring productions.

- When specifying a programming language, we want the grammar to be completely unambiguous.

- An inherently ambiguous language does not possess an unambiguous grammar.

- There is no algorithm that can examine an arbitrary context-free grammar and tell if it is ambiguous, i.e., detecting ambiguity in context-free grammars is an undecidable problem.

Grammar: If-then-else, Pascal solution

Dealing with Ambiguity

- Can’t always remove an ambiguity from a grammar by restructuring productions.

- When specifying a programming language, we want the grammar to be completely unambiguous.

- An inherently ambiguous language does not possess an unambiguous grammar.

- There is no algorithm that can examine an arbitrary context-free grammar and tell if it is ambiguous, i.e., detecting ambiguity in context-free grammars is an undecidable problem.

Limitation of CFGs

- CFGs are not powerful enough to describe some languages.

- Example:
The language consisting of strings with one or more a’s followed by the same number of b’s then the same number of c’s.

- Questions:
  - Exactly what things can and cannot be expressed with a CFG
  - Can we write an algorithm which examines an arbitrary CFG and tells if it is ambiguous or not?
  - Is there an algorithm that can examine two arbitrary CFGs and determine if they generate the same language?
Using CFGs for PL Syntax

- CFGs can be used to describe most PLs' syntax
- Some aspects of programming language syntax can’t be specified with CFGs:
  - Cannot declare the same identifier twice in the same block.
  - Must declare an identifier before using it.
  - A[i,j] is valid only if A is two-dimensional.
  - The number of actual parameters must equal the number of formal parameters.
- Other things are awkward to say with CFGs:
  - Identifier names must be no more than 50 characters long.

Implementations in Translation

The Translation Process:

1. **Lexical Analysis**: Converts source code into sequence of tokens. We use regular grammars and finite state automata (recognizers)
2. **Syntactic Analysis**: Structures tokens into initial parse tree. We use CFGs and parsing algorithms.
3. **Semantic Analysis**: Annotates parse tree with semantic actions.
4. **Code Generation**: Produces final machine code.

Readings:

*Today: Mitchell, Chapter 4.1*

*Next lecture: Mitchell, Chapter 3*