CMPT 379
Compilers

Anoop Sarkar

http://www.cs.sfu.ca/~anoop
Parsing

source

program

Lexical
Analyzer

Lexical
Errors

token

next()

Parser

Syntax
Errors

parse
tree

Later
Stages
Context-free Grammars

• Set of rules by which valid sentences can be constructed.

• Example:

  Sentence → Noun Verb Object
  Noun → trees | compilers
  Verb → are | grow
  Object → on Noun | Adjective
  Adjective → slowly | interesting

• What strings can Sentence derive?

• Syntax only – no semantic checking
Derivations of a CFG

• compilers grow on trees
• compilers grow on Noun
• compilers grow Object
• compilers Verb Object
• Noun Verb Object
• Sentence
Derivations and parse trees

Sentence

Noun
  compilers

Verb
  grow

Object
  on
  Noun
    trees
Why use grammars for PL?

- Precise, yet easy-to-understand specification of language
- Construct parser automatically
  - Detect potential problems
- Structure and simplify remaining compiler phases
- Allow for evolution
CFG Notation

• A reference grammar is a concise description of a context-free grammar
• For example, a reference grammar can use regular expressions on the right hand sides of CFG rules
• Can even use ideas like comma-separated lists to simplify the reference language definition
Writing a CFG for a PL

• First write (or read) a reference grammar of what you want to be valid programs
• For now, we only worry about the structure, so the reference grammar might choose to over-generate in certain cases (e.g. `bool x = 20;`)
• Convert the reference grammar to a CFG
• Certain CFGs might be easier to work with than others (this is the essence of the study of CFGs and their parsing algorithms for compilers)
CFG Notation

• Normal CFG notation
  
  \[ E \rightarrow E \ast E \]
  
  \[ E \rightarrow E + E \]

• Backus Naur notation
  
  \[ E ::= E \ast E | E + E \]
  
  (an or-list of right hand sides)
Parse Trees for programs
Arithmetic Expressions

• $E \rightarrow E + E$
• $E \rightarrow E \times E$
• $E \rightarrow (E)$
• $E \rightarrow -E$
• $E \rightarrow id$
Leftmost derivations for id + id * id

\[
E \rightarrow E + E \\
E \rightarrow E * E \\
E \rightarrow ( E ) \\
E \rightarrow - E \\
E \rightarrow id
\]

- \quad E \Rightarrow E + E
  \Rightarrow id + E
  \Rightarrow id + E * E
  \Rightarrow id + id * E
  \Rightarrow id + id * id
Leftmost derivations for

\[ \text{id} + \text{id} \ast \text{id} \]

\[
\begin{align*}
E & \rightarrow E + E \\
E & \rightarrow E \ast E \\
E & \rightarrow (E) \\
E & \rightarrow -E \\
E & \rightarrow \text{id}
\end{align*}
\]

\[
\begin{align*}
\text{• } E & \Rightarrow E \ast E \\
\Rightarrow & \quad E + E \ast E \\
\Rightarrow & \quad \text{id} + E \ast E \\
\Rightarrow & \quad \text{id} + \text{id} \ast E \\
\Rightarrow & \quad \text{id} + \text{id} \ast \text{id}
\end{align*}
\]
Rightmost derivation for \( \text{id} + \text{id} \ast \text{id} \)

- \( E \rightarrow E + E \)
- \( E \rightarrow E \ast E \)
- \( E \rightarrow (E) \)
- \( E \rightarrow -E \)
- \( E \rightarrow \text{id} \)

\[
\begin{align*}
E & \Rightarrow E \ast E \\
& \Rightarrow E \ast \text{id} \\
& \Rightarrow E + E \ast \text{id} \\
& \Rightarrow E + \text{id} \ast \text{id} \\
& \Rightarrow \text{id} + \text{id} \ast \text{id}
\end{align*}
\]
Ambiguity

• Grammar is ambiguous if more than one parse tree is possible for some sentences

• Examples in English:
  – Two sisters reunited after 18 years in checkout counter

• Ambiguity is not acceptable in PL
  – Unfortunately, it’s undecidable to check whether a given CFG is ambiguous
  – Some CFLs are inherently ambiguous (do not have an unambiguous CFG)
Ambiguity

- Alternatives
  - Massage grammar to make it unambiguous
  - Rely on “default” parser behavior
  - Augment parser

Consider the original ambiguous grammar:

\[
\begin{align*}
E &\rightarrow E + E \\
E &\rightarrow E * E \\
E &\rightarrow ( E ) \\
E &\rightarrow - E \\
E &\rightarrow \text{id}
\end{align*}
\]

How can we change the grammar to get only one tree for the input \( \text{id} + \text{id} * \text{id} \)
Ambiguity

• Original ambiguous grammar:
  - $E \rightarrow E + E \quad E \rightarrow E \ast E$
  - $E \rightarrow (E) \quad E \rightarrow - E$
  - $E \rightarrow id$

• Unambiguous grammar:
  - $E \rightarrow E + T \quad T \rightarrow T \ast F$
  - $E \rightarrow T \quad T \rightarrow F$
  - $F \rightarrow (E) \quad F \rightarrow - E$
  - $F \rightarrow id$

• Input: id + id * id

Warning! Is this unambiguous?
Check derivations for – id + id

Compare with F → - F
Dangling else ambiguity

• Original Grammar (ambiguous)
  Stmt → if Expr then Stmt else Stmt
  Stmt → if Expr then Stmt
  Stmt → Other

• Modified Grammar (unambiguous?)
  Stmt → if Expr then Stmt
  Stmt → MatchedStmt
  MatchedStmt → if Expr then MatchedStmt else Stmt
  MatchedStmt → Other
Dangling else ambiguity

• Original Grammar (ambiguous)
  Stmt → \textbf{if} \ Expr \ \textbf{then} \ Stmt \ \textbf{else} \ Stmt
  Stmt → \textbf{if} \ Expr \ \textbf{then} \ Stmt
  Stmt → \textbf{Other}

• Unambiguous grammar
  Stmt → \textbf{MatchedStmt}
  Stmt → \textbf{UnmatchedStmt}
  MatchedStmt → \textbf{if} \ Expr \ \textbf{then} \ MatchedStmt \ \textbf{else} \ MatchedStmt
  MatchedStmt → \textbf{Other}
  UnmatchedStmt → \textbf{if} \ Expr \ \textbf{then} \ Stmt
  UnmatchedStmt → \textbf{if} \ Expr \ \textbf{then} \ MatchedStmt \ \textbf{else} \ UnmatchedStmt
Dangling else ambiguity

• Check unambiguous dangling-else grammar with the following inputs:
  – if Expr then if Expr then Other else Other
  – if Expr then if Expr then Other else Other else Other
  – if Expr then if Expr then Other else if Expr then Other else Other
Other Ambiguous Grammars

• Consider the grammar
  \[ R \rightarrow R \,|\, R \mid R \,|\, R \,|\, R \,|\, R \,|\, R \,|\, R \,|\, R \,|\, ( \mid R \mid ) \mid a \mid b \]

• What does this grammar generate?

• What’s the parse tree for \( a|b*a \)

• Is this grammar ambiguous?
Left Factoring

• Original Grammar (ambiguous)
  Stmt → if Expr then Stmt else Stmt
  Stmt → if Expr then Stmt
  Stmt → Other

• Left-factored Grammar (still ambiguous):
  Stmt → if Expr then Stmt OptElse
  Stmt → Other
  OptElse → else Stmt | ε
Left Factoring

• In general, for rules

\[ A \rightarrow \alpha \beta_1 | \alpha \beta_2 | \ldots | \alpha \beta_n | \gamma \]

• Left factoring is achieved by the following grammar transformation:

\[ A \rightarrow \alpha A' | \gamma \]
\[ A' \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n \]
Grammar Transformations

• G is converted to G’ s.t. L(G’) = L(G)
• Left Factoring
• Removing cycles: A \Rightarrow^+ A
• Removing \epsilon\text{-}rules of the form A \rightarrow \epsilon
• Eliminating left recursion
• Conversion to normal forms:
  – Chomsky Normal Form, A \rightarrow B C and A \rightarrow a
  – Greibach Normal Form, A \rightarrow a \beta
Eliminating Left Recursion

• Simple case, for left-recursive pair of rules:

\[ A \rightarrow A\alpha \mid \beta \]

• Replace with the following rules:

\[ A \rightarrow \beta A' \]
\[ A' \rightarrow \alpha A' \mid \epsilon \]

• Elimination of immediate left recursion
Eliminating Left Recursion

• Example:
  \[ E \rightarrow E + T, \ E \rightarrow T \]

• Without left recursion:
  \[ E \rightarrow T \ E_1, \ E_1 \rightarrow + \ T \ E_1, \ E_1 \rightarrow \epsilon \]

• Simple algorithm doesn’t work for 2-step recursion:
  \[ S \rightarrow A \ a, \ S \rightarrow b \]
  \[ A \rightarrow A \ c, \ A \rightarrow S \ d, \ A \rightarrow \epsilon \]
Eliminating Left Recursion

• Problem CFG:
  \[ S \rightarrow A \, a \, , \, S \rightarrow b \]
  \[ A \rightarrow A \, c \, , \, A \rightarrow S \, d \, , \, A \rightarrow \varepsilon \]

• Expand possibly left-recursive rules:
  \[ S \rightarrow A \, a \, , \, S \rightarrow b \]
  \[ A \rightarrow A \, c \, , \, A \rightarrow A \, a \, d \, , \, A \rightarrow b \, d \, , \, A \rightarrow \varepsilon \]

• Eliminate immediate left-recursion
  \[ S \rightarrow A \, a \, , \, S \rightarrow b \]
  \[ A \rightarrow b \, d \, A_1 \, , \, A \rightarrow A_1 \, , \]
  \[ A_1 \rightarrow c \, A_1 \, , \, A_1 \rightarrow a \, d \, A_1 \, , \, A_1 \rightarrow \varepsilon \]
Eliminating Left Recursion

• We cannot use the algorithm if the non-terminal also derives epsilon. Let’s see why:
  
  \[ A \rightarrow AAa | b | \varepsilon \]

• Using the standard lrec removal algorithm:
  
  \[ A \rightarrow bA_1 | A_1 \]

  \[ A_1 \rightarrow AaA_1 | \varepsilon \]
Eliminating Left Recursion

• First we eliminate the epsilon rule:
  \[ A \rightarrow AAa \mid b \mid \varepsilon \]
• Since A is the start symbol, create a new start symbol to generate the empty string:
  \[
  A_1 \rightarrow A \mid \varepsilon \\
  A \rightarrow A A a \mid A a \mid a \mid b
  \]
• Now we can do the usual \text{lrec} algorithm:
  \[
  A_1 \rightarrow A \mid \varepsilon \\
  A \rightarrow a A_2 \mid b A_2 \\
  A_2 \rightarrow A a A_2 \mid a A_2 \mid \varepsilon
  \]
Context-free languages and Pushdown Automata

• Recall that for each regular language there was an equivalent finite-state automaton

• The FSA was used as a recognizer of the regular language

• For each context-free language there is also an automaton that recognizes it: called a pushdown automaton (pda)
Context-free languages and Pushdown Automata

- Similar to FSAs there are non-deterministic pda and deterministic pda
- Unlike in the case of FSAs we cannot always convert a npda to a dpda
- Our goal in compiler design will be to choose grammars carefully so that we can always provide a dpda for it
- Similar to the FSA case, a DFA construction provides us with the algorithm for lexical analysis,
- In this case the construction of a dpda will provide us with the algorithm for parsing (take in strings and provide the parse tree)
- We will study later how to convert a given CFG into a parser by first converting into a PDA
Pushdown Automata

- PDA has
  - an alphabet (terminals) and
  - stack symbols (like non-terminals),
  - a finite-state automaton, and
  - stack

\[
\begin{align*}
\varepsilon, \varepsilon & \rightarrow \$ \\
0, \varepsilon & \rightarrow A \\
1, A & \rightarrow \varepsilon \\
\varepsilon, \$ & \rightarrow \varepsilon \\
1, A & \rightarrow \varepsilon \\
\end{align*}
\]

- e.g. PDA for language \( L = \{ 0^n1^n : n \geq 0 \} \)
- \( \rightarrow \) implies a push/pop of stack symbol(s)

- push stack symbol A
- pop stack symbol A
- check that stack is empty
Non-CF Languages

\[ L_1 = \{wcw \mid w \in (a|b)^*\} \]

\[ L_2 = \{a^n b^m c^n d^m \mid n \geq 1, m \geq 1\} \]

\[ L_3 = \{a^n b^n c^n \mid n \geq 0\} \]
CF Languages

\[ L_4 = \{wcw^R \mid w \in (a|b)^*\} \]

\[ S \rightarrow aS\ a \mid bS\ b \mid c \]

\[ L_5 = \{a^n b^n c^m d^n \mid n \geq 1, m \geq 1\} \]

\[ S \rightarrow aS\ d \mid aA\ d \]

\[ A \rightarrow bA\ c \mid bc \]
Summary

• CFGs can be used to describe PL
• Derivations correspond to parse trees
• Parse trees represent the structure of programs
• Ambiguous CFGs exist
• Some forms of ambiguity can be fixed by changing the grammar
• Grammars can be simplified by left-factoring
• Left recursion in a CFG can be eliminated
• CF languages can be recognized using Pushdown Automata
Extra Slides
Non-CF Languages

• The pumping lemma for CFLs [Bar-Hillel] is similar to the pumping lemma for RLs
• For a string $wuxvy$ in a CFL for $u,v \neq \varepsilon$ and the string is longer than $p$ and $|xvy| \leq p$ then $wu^nxv^ny$ is also in the CFL for $n \geq 0$
• Not strong enough to work for every non-CF language (cf. Ogden’s Lemma)