CMPT 379
Compilers

Anoop Sarkar
http://www.cs.sfu.ca/~anoop

Parsing

Sources program → Lexical Analyzer

Lexical Analyzer → Parser

Parser → Later Stages

Lexical Errors → Syntax Errors

Lexical Errors

Syntax Errors

parse tree

next()
Context-free Grammars

• Set of rules by which valid sentences can be constructed.
• Example:
  Sentence $\rightarrow$ Noun Verb Object
  Noun $\rightarrow$ trees | compilers
  Verb $\rightarrow$ are | grow
  Object $\rightarrow$ on Noun | Adjective
  Adjective $\rightarrow$ 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

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
  \[ E \rightarrow E \times E \]
  \[ E \rightarrow E + E \]

• Backus Naur notation
  \[ E ::= E \times E | E + E \]
  (an or-list of right hand sides)

Parse Trees for programs
Arithmetic Expressions

- $E \rightarrow E + E$
- $E \rightarrow E \ast E$
- $E \rightarrow (E)$
- $E \rightarrow -E$
- $E \rightarrow id$

Leftmost derivations for $id + id \ast id$

$$E \Rightarrow E + E$$
$$E \Rightarrow E \ast E$$
$$E \Rightarrow (E)$$
$$E \Rightarrow -E$$
$$E \Rightarrow id$$
Leftmost derivations for
\( id + id * id \)

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

\[
\begin{align*}
\bullet E &\Rightarrow E * E \\
E &\Rightarrow E + E * E \\
( E ) &\Rightarrow id + E * E \\
E &\Rightarrow id + id * E \\
E &\Rightarrow id + id * id
\end{align*}
\]

Rightmost derivation for
\( id + id * id \)

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

\[
\begin{align*}
E &\Rightarrow E * E \\
E &\Rightarrow E * id \\
E &\Rightarrow E + E * id \\
E &\Rightarrow E + id * id \\
E &\Rightarrow id + id * id
\end{align*}
\]
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:
  \[
  E \rightarrow E + E \quad E \rightarrow E * E \\
  E \rightarrow ( E ) \quad E \rightarrow - E \\
  E \rightarrow \text{id}
  \]
• How can we change the grammar to get only one tree for the input \text{id + id} * \text{id}
Ambiguity

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

- Unambiguous grammar:
  - $E \rightarrow E + T$
  - $E \rightarrow ( E )$
  - $E \rightarrow id$

- Input: $id + id * id$

Warning! Is this unambiguous?

Compare with $F \rightarrow - F$

Dangling else ambiguity

- Original Grammar (ambiguous)
  Stmt $\rightarrow$ if Expr then Stmt else Stmt
  Stmt $\rightarrow$ if Expr then Stmt
  Stmt $\rightarrow$ Other

- Modified Grammar (unambiguous?)
  Stmt $\rightarrow$ if Expr then Stmt
  Stmt $\rightarrow$ MatchedStmt
  MatchedStmt $\rightarrow$ if Expr then MatchedStmt else Stmt
  MatchedStmt $\rightarrow$ Other
Dangling else ambiguity

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

- Unambiguous grammar
  Stmt → MatchedStmt
  Stmt → UnmatchedStmt
  MatchedStmt → if Expr then MatchedStmt else MatchedStmt
  MatchedStmt → Other
  UnmatchedStmt → if Expr then Stmt
  UnmatchedStmt → if Expr then MatchedStmt 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
  \[ R \rightarrow R \, | \, R \, | \, R \, R \, | \, R \, \ast \, | \, ( \, R \, ) \, | \, a \, | \, b \]
• 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

• 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 ε-rules of the form A \rightarrow ε
• 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:
  \[ E \rightarrow T \ E_,\ E_ \rightarrow + T \ E_,\ E_ \rightarrow \varepsilon \]

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

• 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:
  \[ A \rightarrow AAa \mid b \mid \varepsilon \]

• Using the standard lrec removal algorithm:
  \[ A \rightarrow bA \mid A \]
  \[ A \rightarrow Aa \mid \varepsilon \]

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:
  \[ A_1 \rightarrow A \mid \varepsilon \]
  \[ A \rightarrow AAa \mid Aa \mid a \mid b \]

• Now we can do the usual lrec algorithm:
  \[ A_1 \rightarrow A \mid \varepsilon \]
  \[ A \rightarrow aA_2 \mid bA_2 \]
  \[ A_2 \rightarrow AaA_2 \mid aA_2 \mid \varepsilon \]
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 $wu^nxvy^ny$ is also in the CFL for $n \geq 0$
• Not strong enough to work for every non-CF language (cf. Ogden’s Lemma)

Non-CF Languages

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

$L_2 = \{a^nb^mc^nd^m \mid n \geq 1, m \geq 1\}$

$L_3 = \{a^nb^nc^n \mid n \geq 0\}$
CF Languages

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

\[ S \to aSa \mid bSb \mid c \]

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

\[ S \to aSd \mid aAd \]

\[ A \to bAc \mid bc \]

---

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
e.g. PDA for language
L = \{ 0^n1^n : n \geq 0 \}

implies a push/pop of stack symbol(s)

\[ \varepsilon, \varepsilon \rightarrow \$ \]

\[ 0, \varepsilon \rightarrow A \]

push stack symbol A

\[ 1, A \rightarrow \varepsilon \]

pop stack symbol A

\[ \varepsilon, \$ \rightarrow \varepsilon \]

check that stack is empty
Summary

• CFGs can be used describe PL
• Derivations correspond to parse trees
• Parse trees represent 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