CMPT 379 Compilers

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

Parsing



Context-free Grammars

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



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
 - $E \rightarrow E * E$

 $E \rightarrow E + E$

• Backus Naur notation

 $\mathbf{E} ::= \mathbf{E} * \mathbf{E} | \mathbf{E} + \mathbf{E}$

(an or-list of right hand sides)

Parse Trees for programs



Arithmetic Expressions

- $E \rightarrow E + E$
- $E \rightarrow E * E$
- $E \rightarrow (E)$
- E → E
- E → id

Leftmost derivations for id + id * id

- $E \Rightarrow E + E$ $\Rightarrow id + E$ $\Rightarrow id + E * E$
- \Rightarrow id + id * E
- \Rightarrow id + id * id



Leftmost derivations for id + id * id

• $E \Rightarrow E * E$ $\Rightarrow E + E * E$ $\Rightarrow id + E * E$ $\Rightarrow id + id * E$ $\Rightarrow id + id * id$



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 grammar is ambiguous

Ambiguity

- Alternatives
 - Massage grammar to make it unambiguous
 - Rely on "default" parser behavior
 - Augment parser
- Consider the original ambiguous grammar:

```
E \rightarrow E + E \qquad E \rightarrow E * EE \rightarrow (E) \qquad E \rightarrow -EE \rightarrow id
```

• How can we change the grammar to get only one tree for the input **id** + **id** * **id**

Ambiguity



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 → if Expr then Stmt else Stmt
 Stmt → if Expr then Stmt
 - Stmt \rightarrow Other
- Unambiguous grammar
 - Stmt \rightarrow MatchedStmt
 - Stmt → UnmatchedStmt
 - MatchedStmt -> if Expr then MatchedStmt else MatchedStmt
 - MatchedStmt \rightarrow 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 $R \rightarrow R' |' R | R R | R' *' |' (' R')' | a | b$
- 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 \rightarrow Other

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

Left Factoring

• In general, for rules

$$A \to \alpha \beta_1 \mid \alpha \beta_2 \mid \ldots \mid \alpha \beta_n \mid \gamma$$

• Left factoring is achieved by the following grammar transformation:

$$A \to \alpha A' \mid \gamma$$

 $A' \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

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 $A \rightarrow \varepsilon$
- Eliminating left recursion
- Conversion to normal forms:
 - Chomsky Normal Form, $A \rightarrow B C$ and $A \rightarrow a$
 - Greibach Normal Form, $A \rightarrow a \beta$

- Simple case, for left-recursive pair of rules: $A \rightarrow A\alpha \mid \beta$
- Replace with the following rules:

 $\begin{array}{c} A \to \beta A' \\ A' \to \alpha A' \mid \epsilon \end{array}$

• Elimination of immediate 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:

$$S \rightarrow A a, S \rightarrow b$$

 $A \rightarrow A c, A \rightarrow S d, A \rightarrow \varepsilon$

• 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 → A a , S → b
 A → A c , A → A a d , A → b d , A → ε
- Eliminate immediate left-recursion $S \rightarrow A a, S \rightarrow b$ $A \rightarrow b d A_1, A \rightarrow A_1, A_1 \rightarrow c A_1, A_1 \rightarrow a d A_1, A_1 \rightarrow \epsilon$

- We cannot use the algorithm if the nonterminal also derives epsilon. Let's see why:
 A → AAa|b|ε
- Using the standard lrec removal algorithm: $A \rightarrow bA_1 \mid A_1$ $A_1 \rightarrow AaA_1 \mid \varepsilon$

- First we eliminate the epsilon rule: $A \rightarrow AAa \mid b \mid \epsilon$
- Since A is the start symbol, create a new start symbol to generate the empty string: $A_1 \rightarrow A \mid \epsilon$ $A \rightarrow AAa \mid Aa \mid a \mid b$
- Now we can do the usual lrec algorithm: $A_1 \rightarrow A \mid \epsilon \qquad A \rightarrow aA_2 \mid bA_2$ $A_2 \rightarrow AaA_2 \mid aA_2 \mid \epsilon$

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 long enough then $wu^n xv^n y$ is also in the CFL for $n \ge 0$
- Not strong enough to work for every non-CF language (cf. Ogden's Lemma)

Non-CF Languages $L_1 = \{wcw \mid w \in (a|b)*\}$ $L_2 = \{a^n b^m c^n d^m \mid n \ge 1, m \ge 1\}$ $L_3 = \{a^n b^n c^n \mid n \ge 0\}$

CF Languages

 $L_4 = \{wcw^R \mid w \in (a|b)*\}$ $S \rightarrow aSa \mid bSb \mid c$ $L_5 = \{a^n b^m c^m d^n \mid n \ge 1, m \ge 1\}$ $S \rightarrow aSd \mid aAd$ $A \rightarrow bAc \mid bc$

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



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