

CMPT 379

Compilers

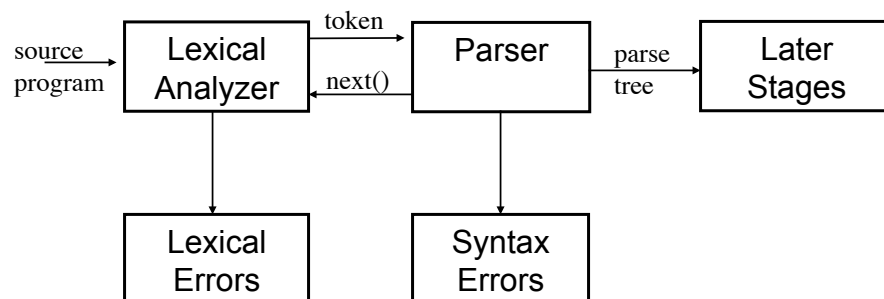
Anoop Sarkar

<http://www.cs.sfu.ca/~anoop>

10/5/11

1

Parsing



10/5/11

2

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

10/5/11

3

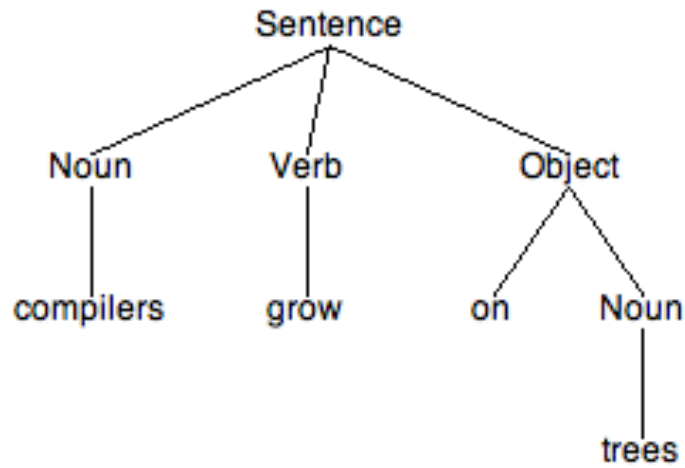
Derivations of a CFG

- *compilers grow on trees*
- *compilers grow on* **Noun**
- *compilers grow* **Object**
- *compilers* **Verb Object**
- **Noun Verb Object**
- **Sentence**

10/5/11

4

Derivations and parse trees



10

5

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

10/5/11

6

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

10/5/11

7

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)

10/5/11

8

CFG Notation

- Normal CFG notation

$E \rightarrow E * E$

$E \rightarrow E + E$

- Backus Naur notation

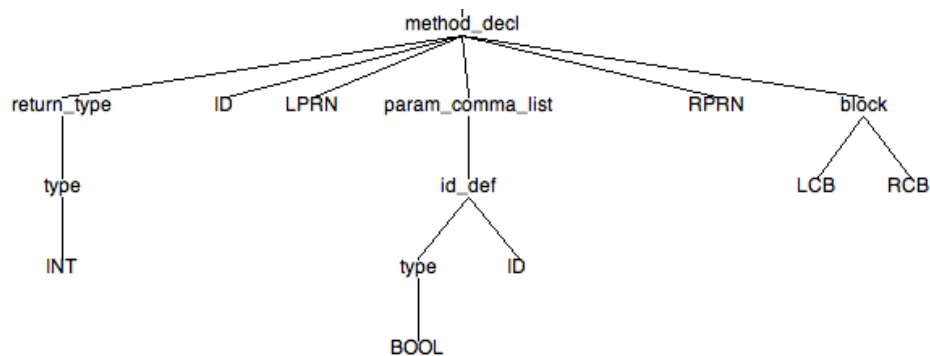
$E ::= E * E \mid E + E$

(an or-list of right hand sides)

10/5/11

9

Parse Trees for programs



10/5/11

10

Arithmetic Expressions

- $E \rightarrow E + E$
- $E \rightarrow E * E$
- $E \rightarrow (E)$
- $E \rightarrow - E$
- $E \rightarrow \text{id}$

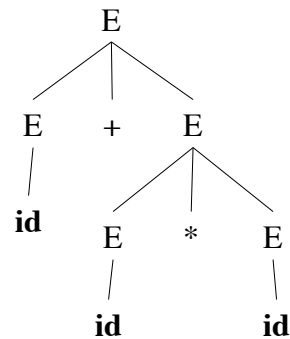
10/5/11

11

Leftmost derivations for **id + id * id**

$E \rightarrow E + E$
 $E \rightarrow E * E$
 $E \rightarrow (E)$
 $E \rightarrow - E$
 $E \rightarrow \text{id}$

• $E \Rightarrow E + E$
 $\Rightarrow \text{id} + E$
 $\Rightarrow \text{id} + E * E$
 $\Rightarrow \text{id} + \text{id} * E$
 $\Rightarrow \text{id} + \text{id} * \text{id}$

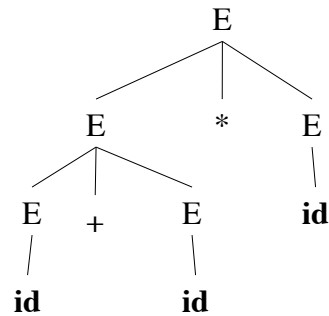


10/5/11

12

Leftmost derivations for **id + id * id**

$E \rightarrow E + E$	• $E \Rightarrow E * E$
$E \rightarrow E * E$	$\Rightarrow E + E * E$
$E \rightarrow (E)$	$\Rightarrow id + E * E$
$E \rightarrow - E$	$\Rightarrow id + id * E$
$E \rightarrow id$	$\Rightarrow id + id * id$

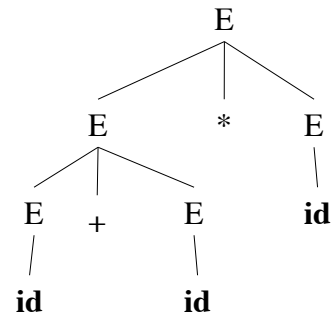


10/5/11

13

Rightmost derivation for **id + id * id**

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



10/5/11

14

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)

10/5/11

15

Ambiguity

- Alternatives
 - Massage grammar to make it unambiguous
 - Rely on “default” parser behavior
 - Augment parser
- Consider the original ambiguous grammar:
$$\begin{array}{ll} E \rightarrow E + E & E \rightarrow E * E \\ E \rightarrow (E) & E \rightarrow - E \\ E \rightarrow \text{id} & \end{array}$$
- How can we change the grammar to get only one tree for the input **id + id * id**

10/5/11

16

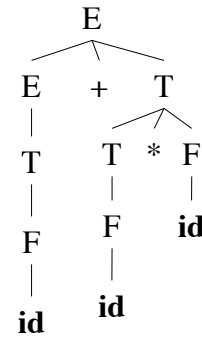
Ambiguity

- Original ambiguous grammar:

- $E \rightarrow E + E$ $E \rightarrow E * E$
- $E \rightarrow (E)$ $E \rightarrow - E$
- $E \rightarrow \text{id}$

- Unambiguous grammar:

- $E \rightarrow E + T$ $T \rightarrow T * F$
- $E \rightarrow T$ $T \rightarrow F$
- $F \rightarrow (E)$ $F \rightarrow - E$
- $F \rightarrow \text{id}$



- Input: $\text{id} + \text{id} * \text{id}$

Warning! Is this unambiguous?

Compare with $F \rightarrow - F$

10/5/11

17

Dangling else ambiguity

- Original Grammar (ambiguous)

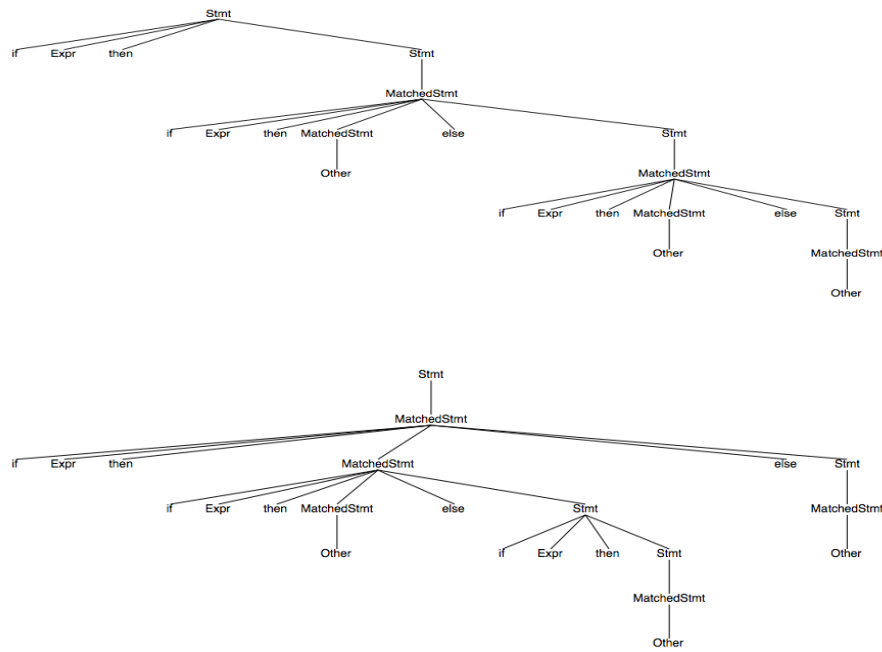
- $\text{Stmt} \rightarrow \text{if Expr then Stmt else Stmt}$
- $\text{Stmt} \rightarrow \text{if Expr then Stmt}$
- $\text{Stmt} \rightarrow \text{Other}$

- Modified Grammar (unambiguous?)

- $\text{Stmt} \rightarrow \text{if Expr then Stmt}$
- $\text{Stmt} \rightarrow \text{MatchedStmt}$
- $\text{MatchedStmt} \rightarrow \text{if Expr then MatchedStmt else Stmt}$
- $\text{MatchedStmt} \rightarrow \text{Other}$

10/5/11

18



Dangling else ambiguity

- Original Grammar (ambiguous)
 - $\text{Stmt} \rightarrow \text{if Expr then Stmt else Stmt}$
 - $\text{Stmt} \rightarrow \text{if Expr then Stmt}$
 - $\text{Stmt} \rightarrow \text{Other}$
- Unambiguous grammar
 - $\text{Stmt} \rightarrow \text{MatchedStmt}$
 - $\text{Stmt} \rightarrow \text{UnmatchedStmt}$
 - $\text{MatchedStmt} \rightarrow \text{if Expr then MatchedStmt else MatchedStmt}$
 - $\text{MatchedStmt} \rightarrow \text{Other}$
 - $\text{UnmatchedStmt} \rightarrow \text{if Expr then Stmt}$
 - $\text{UnmatchedStmt} \rightarrow \text{if Expr then MatchedStmt else UnmatchedStmt}$

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

10/5/11

21

Other Ambiguous Grammars

- Consider the grammar
$$R \rightarrow R \mid R \mid RR \mid R^* \mid ('R') \mid a \mid b$$
- What does this grammar generate?
- What's the parse tree for $a|b^*a$
- Is this grammar ambiguous?

10/5/11

22

Left Factoring

- Original Grammar (ambiguous)

Stmt \rightarrow if Expr then Stmt else Stmt

Stmt \rightarrow if Expr then Stmt

Stmt \rightarrow Other

- Left-factored Grammar (still ambiguous):

Stmt \rightarrow if Expr then Stmt OptElse

Stmt \rightarrow Other

OptElse \rightarrow else Stmt | ϵ

10/5/11

23

Left Factoring

Left Factor:

A \rightarrow XA

| XB

| X

| Y

| Z

- In general, for rules

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$$

- Left factoring is achieved by the following grammar transformation:

$$A \rightarrow \alpha A' \mid \gamma$$

$$A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

10/5/11

24

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

10/5/11

25

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

10/5/11

26

Eliminating Left Recursion

- Example:
 $E \rightarrow E + T, E \rightarrow T$
- Without left recursion:
 $E \rightarrow T E_1, E_1 \rightarrow + T E_1, E_1 \rightarrow \varepsilon$
- Simple algorithm doesn't work for 2-step recursion:
 $S \rightarrow A a, S \rightarrow b$
 $A \rightarrow A c, A \rightarrow S d, A \rightarrow \varepsilon$

10/5/11

27

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
 $S \rightarrow A a, S \rightarrow b$
 $A \rightarrow b d A_1, A \rightarrow A_1, A_1 \rightarrow c A_1, A_1 \rightarrow a d A_1, A_1 \rightarrow \varepsilon$

10/5/11

28

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:

$$A \rightarrow bA_1 \mid A_1$$

$$A_1 \rightarrow AaA_1 \mid \varepsilon$$

10/5/11

29

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:

$$A_1 \rightarrow A \mid \varepsilon \quad A \rightarrow AAa \mid Aa \mid a \mid b$$

- Now we can do the usual lrec algorithm:

$$A_1 \rightarrow A \mid \varepsilon \quad A \rightarrow aA_2 \mid bA_2$$

$$A_2 \rightarrow AaA_2 \mid aA_2 \mid \varepsilon$$

10/5/11

30

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

10/5/11

31

Non-CF Languages

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

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

$$L_3 = \{a^n b^n c^n \mid n \geq 0\}$$

10/5/11

32

CF Languages

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

$$S \rightarrow aSa \mid bSb \mid c$$

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

$$S \rightarrow aSd \mid aAd$$

$$A \rightarrow bAc \mid bc$$

10/5/11

33

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)**

10/5/11

34

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

10/5/11

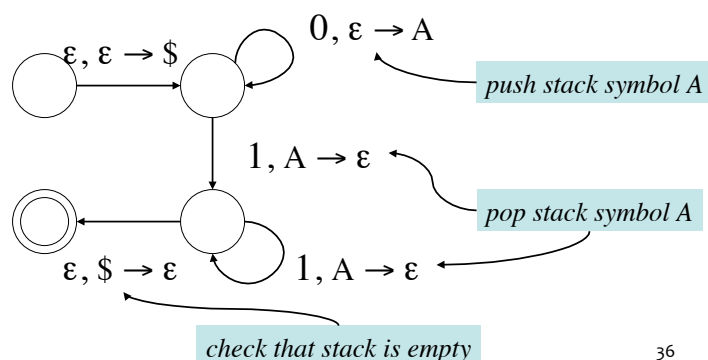
35

Pushdown Automata

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

e.g. PDA for language
 $L = \{ 0^n 1^n : n \geq 0 \}$

→ implies a push/pop of stack symbol(s)



10/5/11

36

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

10/5/11

37