MACM 300
Formal Languages and Automata

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

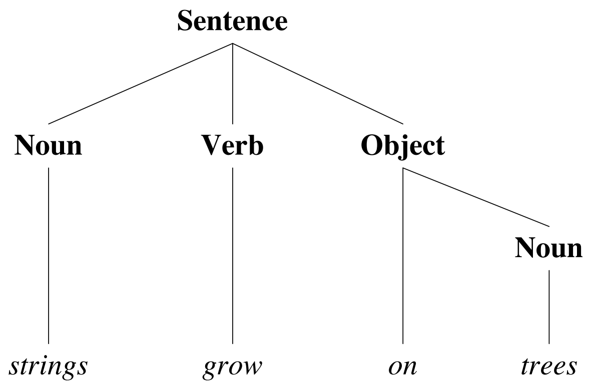
Context-free Grammars

- Set of rules by which valid sentences can be constructed.
- Example:
  - Sentence → Noun Verb Object
  - Noun → *trees* | *strings*
  - Verb → *are* | *grow*
  - Object → *on* Noun | Adjective
  - Adjective → *slowly* | *interesting*
- What strings can Sentence *derive*?
- Syntax only – no semantic checking

Derivations of a CFG

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

Derivations and parse trees



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)

CFG: Formal Definition

- A CFG is a 4-tuple (V, Σ, R, S), where:
  - V is a finite non-empty set of symbols (called *variables*, or *non-terminals*)
  - Σ is a finite set of symbols (called set of terminal symbols, or the alphabet)

    We generally assume that $V \cap \Sigma = \varnothing$
  - R is a finite non-empty set of rules, where each rule is of the form: $A \rightarrow w$, $w \in (V \cup \Sigma)^*$
  - $S \in V$ is the start variable (or start non-terminal)

CFGs and Languages

- For the regular expression $(01)^*$ we know that it corresponds to a language $L = \{\varepsilon, 01, 0101, 010101, \ldots\}$
- We know this because we know the definition of the Kleene closure operator *
- Similarly we define a relation between CFGs and a language as follows:
  1. Write down the start variable
  2. Find a variable written down so far and a rule that has that variable on the left-hand side of a rule in the CFG. Replace the variable with the right-hand side of the rule
  3. Repeat step 2 until no variable remains

CFGs and Languages

- We can formally specify the 3 rules for deriving a language from a CFG as follows:
  - If u, v, w are strings from $(V \cup \Sigma)^*$
  - Then if we have a string uAv written down
  - And if we have a rule in the CFG, $A \rightarrow w$,
  - Then we can replace A with w, written as:

    $uAv \Rightarrow uwv$ (called a **derivation** step)

CFGs and Languages

- Consider CFG: A → 0A1 | ε, here are some strings derived from this grammar:
  - A ⇒ ε
  - A ⇒ 0A1 ⇒ 0ε1 = 01 (derived in 0 or more steps: A ⇒* 01)
  - A ⇒ 0A1 ⇒ 00A11 ⇒ 00ε11 = 0011
  - A ⇒ 0A1 ⇒ 00A11 ⇒ 000A111 ⇒ 000ε111 = 000111
  - …and so on…
- The set of all strings that can be derived is the *language* of the CFG
  - In this case the language is { $0^n 1^n$ | n ≥ 0 }

Arithmetic Expressions

- E → E + E
- E → E * E
- E → (E)
- E → - E
- E → **id**

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

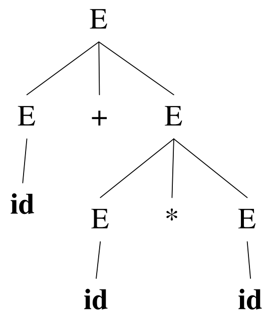
- E ⇒ E + E

⇒ **id** + E

⇒ **id** + E * E

⇒ **id** + **id** * E

⇒ **id** + **id** * **id**



$E \Rightarrow^*_{lm}$ id + id * E

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

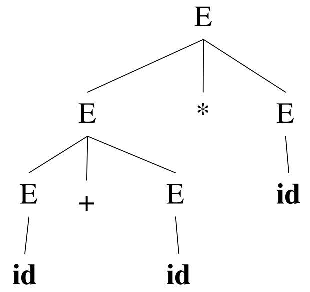
- E ⇒ E * E

⇒ E + E * E

⇒ **id** + E * E

⇒ **id** + **id** * E

⇒ **id** + **id** * **id**

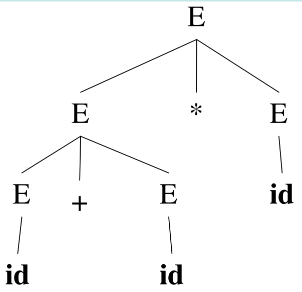


Rightmost derivation for id + id

Note that the parse tree is the same as one of the leftmost derivations.

E → E + E
E → E * E
E → (E)
E → - E
E → **id**

E ⇒ E * E
⇒ E * **id**
⇒ E + E * **id**
⇒ E + **id** * **id**
⇒ **id** + **id** * **id**



$E \Rightarrow^*_{rm} E + E * id$

A grammar G is **ambiguous** iff there are **two parse trees** or two distinct leftmost or two distinct rightmost derivations for **any** string in L(G)

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

---

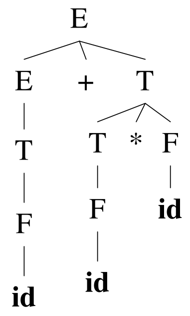
Ambiguity

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

  E → E + E   E → E * E
  E → (E)   E → - E
  E → **id**
- How can we change the grammar to get only one tree for the input **id + id * id**

Ambiguity

- Original ambiguous grammar:
  - E → E + E   E → E * E
  - E → (E)   E → - E
  - E → id
- Unambiguous grammar:
  - E → E + T   T → T * F
  - E → T   T → F
  - F → (E)   F → - E
  - F → id
- Input: id + id * id



Other Ambiguous Grammars

- Consider the grammar

  R → R ‘∪’ R | R R | R ‘*’ | ‘(‘ R ‘)’ | a | b
- What does this grammar generate?
- What’s the parse tree for $a \cup b*a$
- Is this grammar ambiguous?

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 → ε
- Eliminating left recursion
- Conversion to normal forms:
  - Chomsky Normal Form, A → B C and A → a
  - Greibach Normal Form, A → a β

Chomsky Normal Form

- First step, add S_0 and then remove epsilon rules

  S → ASA | aB

  A → B | S

  B → b | ε
- After adding S_0 and then ε-removal (remove *):

  S_0 → S

  S → ASA | aB | **a | SA | AS | S**

  A → B | S | ε*

  B → b | ε*

Chomsky Normal Form

- Second step, remove unit rules or chain rules

  S_0 → S

  S → ASA | aB | **a | SA | AS | S**

  A → B | S

  B → b
- After removal of unit rules S_0 → S and S → S:

  S_0 → ASA | aB | a | SA | AS

  S → ASA | aB | a | SA | AS

  A → B | S

  B → b

Chomsky Normal Form

- After removal of unit rules $S_0 \rightarrow S$ and $S \rightarrow S$:

  $S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS$
  $S \rightarrow ASA \mid aB \mid a \mid SA \mid AS$
  $A \rightarrow B \mid S$
  $B \rightarrow b$

- After removal of unit rules $A \rightarrow B$ and $A \rightarrow S$:

  $S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS$
  $S \rightarrow ASA \mid aB \mid a \mid SA \mid AS$
  $A \rightarrow \mathbf{b \mid ASA \mid aB \mid a \mid SA \mid AS}$
  $B \rightarrow b$

Chomsky Normal Form

Consider some simpler examples for next two steps

- Removing terminals from the rhs of rules

  $A \rightarrow B\ a\ C\ d$

- After removal of terminals from the rhs:

  $A \rightarrow B\ N_1\ C\ N_2$
  $N_1 \rightarrow a$
  $N_2 \rightarrow d$

Chomsky Normal Form

- Converting the rhs of each rule to have two non-terminals

  $A \rightarrow B\ N_1\ C\ N_2$
  $N_1 \rightarrow a$
  $N_2 \rightarrow d$

- After converting to binary form:

  $A \rightarrow B\ N_3$ $N_1 \rightarrow a$
  $N_3 \rightarrow N_1\ N_4$ $N_2 \rightarrow d$
  $N_4 \rightarrow C\ N_2$

Chomsky Normal Form

Back to original example

- After removal of unit rules $A \rightarrow B$ and $A \rightarrow S$:

  $S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS$
  $S \rightarrow ASA \mid aB \mid a \mid SA \mid AS$
  $A \rightarrow \mathbf{b \mid ASA \mid aB \mid a \mid SA \mid AS}$
  $B \rightarrow b$

- After adding variables:

  $S_0 \rightarrow A\mathbf{A_1} \mid \mathbf{U}B \mid a \mid SA \mid AS$
  $S \rightarrow A\mathbf{A_1} \mid \mathbf{U}B \mid a \mid SA \mid AS$
  $A \rightarrow b \mid A\mathbf{A_1} \mid \mathbf{U}B \mid a \mid SA \mid AS$
  $\mathbf{A_1 \rightarrow SA}$ $\mathbf{U \rightarrow a}$ $B \rightarrow b$

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 $n \geq 0$
- Not strong enough to work for every non-CF language (cf. Ogden's Lemma)

Non-CF Languages

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

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

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

CF Languages

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

$$S \to aSa \mid bSb \mid c$$

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

$$S \to aSd \mid aAd$$

$$A \to bAc \mid bc$$

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

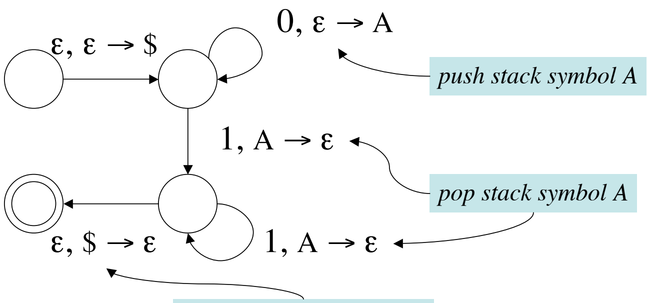
- Similar to FSAs there are non-deterministic pda and deterministic pda
- Unlike in the case of FSAs we cannot always convert a npda to a dpda
- Similar to the FSA case, a DFA construction provides us with the algorithm for lexical analysis,
- In this case the construction of a dpda will provide us with the algorithm for parsing (take in strings and provide the parse tree)

Pushdown Automata

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

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

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



push stack symbol A

pop stack symbol A

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
- CF languages can be recognized using Pushdown Automata