

David Keil
Framingham State University

3. Pushdown automata, context-free languages

1. Pushdown automata
2. Context-free languages and grammars
3. Derivations and parsing
4. Properties of PDAs,
CFLs, and CFGs

Reading: Ch. 5-7

Inquiry

- What problems can be solved by computation?
- Can a state-transition system with stack model more computations than a DFA?
- How do compilers work?

Objectives

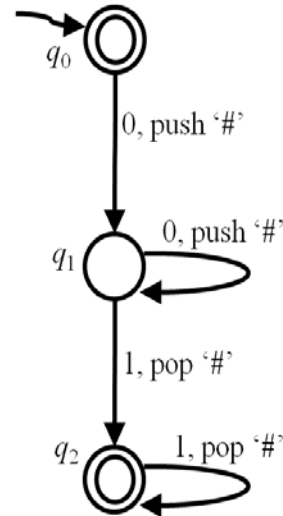
- 3a. Explain the pushdown-automaton model
- 3b. Define a context-free grammar
- 3c. Perform a derivation using a context-free grammar
- 3d. Give a proof of expressiveness for pushdown automata.

1. Pushdown automata

- PDAs (stack machines) are NFAs augmented by a stack, which introduces notion of *memory*
- They accept *context-free languages*, CFL
- They are more expressive than the DFA/NFA model
- *Context-free grammars* generate CFLs
- An application of PDAs is *parsing*, e.g., in program compilation

PDA example A

- This (deterministic) PDA reads zeroes, storing a '#' on stack for each
- Then it pops '#'s and reads ones
- Accepts $0^n 1^n$



Formal definition of PDAs

- PDA $A = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$
- As with DFAs, Q, Σ, q_0, F are state set, alphabet, start state, accept state set, resp.
- Γ (Gamma) is a set of stack symbols
- $\delta: (Q \times \Sigma \times \Gamma \cup \{\lambda\}) \times (Q \times \Gamma \cup \{\lambda\})$ is the transition *relation*
- PDA reads a Σ symbol, popping a Γ symbol from stack, writes a Σ symbol, pushes a Γ symbol
- Acceptance of an input string requires empty stack on termination of input

Transition relation δ (delta)

- $(q, a, x) \rightarrow_A (q', y)$ means that in state q , on input symbol $a \in \Sigma$, and popping $x \in \Gamma$, A can go into state q' and push y onto stack
- Pop- λ transition pops nothing
- Push- λ transition pushes nothing
- PDAs are non-deterministic; may make transitions without consuming input

Formal definition of A

Let $A = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$,

$$Q = \{q_0, q_1, q_2\}$$

$$\Sigma = \{0, 1\}$$

$$\Gamma = \{\#\}$$

$$\delta = \{(q_0, 0, \lambda, q_1, \#),$$

$$(q_1, 0, \#, q_1, \#),$$

$$(q_1, 1, \#, q_2, \lambda),$$

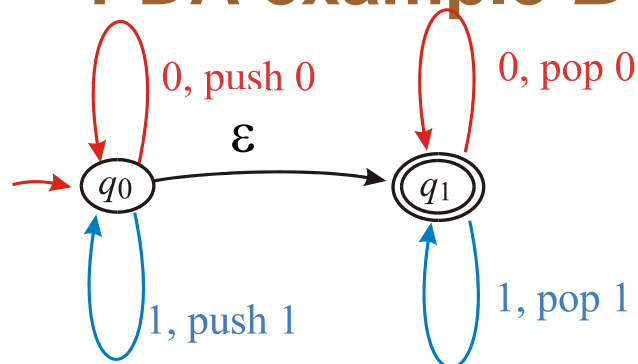
$$(q_2, 1, \#, q_2, \lambda)\}$$

$$F = \{q_0, q_2\}$$

Computations with PDA A

- | | |
|---|---|
| <ul style="list-style-type: none"> • <i>Input:</i> λ
Accepted • <i>Input:</i> 1
No transitions,
rejected • <i>Input:</i> 0011
0 #
0 ##
1 #
1 Accepted | <ul style="list-style-type: none"> • <i>Input:</i> 001
0 #
0 ##
1 #
Nonempty stack,
rejected |
|---|---|

PDA example B



What is δ ?

What is $L(B)$? (Note: not a regular language)

Context-free languages

- *Definition:* A CFL is a language accepted by some PDA
- *Theorem:* All regular languages are CF ($CFL \supset RL$)
- *Proof:* Let PDA A be a DFA M , $L = L(M)$, where δ of A is δ of M , i.e., without any stack operations
- *Theorem:* Some CFLs are not RLs
- *Proof:* $0^n 1^n$ is CF but not regular

2. Context-free grammars

- *Definition:* CFG $G = \langle \Sigma, NT, R, S \rangle$, where
 - Σ is a set of *terminal symbols*
 - NT is a set of *nonterminals* (names)
 - $R \subseteq NT \times (\Sigma \cup NT)^*$ is a set of *production rules*
 - $S \in NT$ is the *start symbol*
- *Production rules* are of the form $X \rightarrow YZ\dots$ where $X \in NT$ and $Y, Z\dots \in (\Sigma \cup NT)$
- The CFGs generate precisely the CFLs

CFG for $L(A)$

$$S \rightarrow \lambda \mid 0S1$$

- This grammar says that a sentence can be defined as a null string or as a 0, followed by a sentence, followed by a 1

Palindrome example (B)

- Palindromes may be defined inductively:
 1. $\lambda, 0, 1 \in PAL$
 2. $(x \in PAL) \Rightarrow (0x0, 1x1 \in PAL)$

- CFG for PAL :

$$S \rightarrow \lambda$$

$$S \rightarrow 0S0 \mid 1S1$$

- For any x , we may prove or disprove by induction that $x \in PAL$

$$L = \{x: n_0(x) = n_1(x)\}$$

- This grammar defines the set of strings that contains an equal number of 0's and 1's
- This is recognized by a stack machine that pushes a marker on '0' and pops on '1'
- A CF grammar defines a set of replacement rules that define "parts of speech", which are the recursive structure of a language

Linear CFGs generate RLs

- *Right-linear* or *left-linear* CFGs generate regular languages; every RL has such a CFG
- *Example:* $0^* | 10^*$ is generated by
 $S \rightarrow X | 1X \quad X \rightarrow \lambda | 0X$
- *Right-linear CFGs* have for each production body at most one nonterminal, and it is the rightmost symbol in the production (e.g., $S \rightarrow aS$)
- A star component of regular grammar follows the pattern for X above

Propositional-logic formulas

- *Example:* Formulas in propositional logic in the language of logic use truth values (*true*, *false*), variable names, and the operators $(,), \neg, \wedge, \vee, \Rightarrow$

- **Grammar:**

$S \rightarrow \text{formula}$

$\text{formula} \rightarrow \text{true} \mid \text{false}$

$\text{ID} \mid (\text{formula}) \mid \neg \text{formula}$

$\text{formula} \wedge \text{formula} \mid \text{formula} \vee \text{formula} \mid$

$\text{formula} \Rightarrow \text{formula}$

Java grammar fragment

$\text{statement} \rightarrow \{ \text{statement-list} \}$

$\text{statement} \rightarrow \text{ID} = \text{expression} ;$

$\text{statement-list} \rightarrow \text{statement statement-list}$

$\text{statement-list} \rightarrow \lambda$

$\text{expression} \rightarrow \text{ID}$

$\text{expression} \rightarrow \text{num-literal}$

$\text{expression} \rightarrow (\text{expression})$

$\text{expression} \rightarrow \text{num-literal} + \text{expression}$

$\text{num-literal} \rightarrow \text{digit} \mid \text{digit num-literal}$

Nondeterminism and PDAs

- Nondeterminism is an essential aspect
- Deterministic PDAs are a weaker model of computation than PDAs
- Some CFLs have no deterministic PDA that accepts them

3. Derivations and parsing

- A *derivation* process demonstrates that a string is in the language of a certain grammar
- *Parse trees* diagram the results of derivations
- *Parsing* is used to build the structure of an utterance, as in compiling a program

Derivation of a string

- Production rules are applied repeatedly, starting with start symbol, by replacing nonterminal on left side of rule with expansions using the right side
- A derivation continues until no nonterminals remain
- Example:

$PAL \Rightarrow 1 PAL 1 \Rightarrow 10 PAL 01 \Rightarrow 1001$

Rule $S \rightarrow 1S1 \quad OS0 \mid \lambda$

Derivation steps

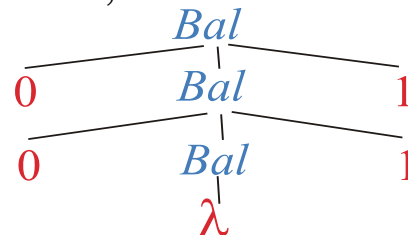
- Suppose β, α are sequences of nonterminal and terminal symbols in a grammar
- $\beta \Rightarrow_G \alpha$ denotes one derivation step, applying one production rule
- $\beta \Rightarrow_G^* \alpha$ denotes that α can be derived from β in multiple steps
- If $x \in L(G)$, and S is G 's starting symbol, then $S \Rightarrow_G^* x$

Derivation example

- $G = \langle \Sigma, NT, R, S \rangle$
 $\Sigma = \{0,1\}$
 $NT = Bal$
 $R = \{Bal \rightarrow \lambda, Bal \rightarrow 0 Bal 1\}$
 $S = Bal$
- *Example:* Derive 0011 from Bal , showing that 0011 is in $L(G)$
 $Bal \Rightarrow 0 Bal 1$ rule 2
 $0 Bal 1 \Rightarrow 00 Bal 11$ rule 2
 $00 Bal 11 \Rightarrow 00 \lambda 11$ rule 1
 $00 \lambda 11 \Rightarrow 0011$ definition of λ
- What is $L(G)$?

Parsing with a CFG

- A *parse tree* is a diagram of a derivation
- Leaves of tree are terminals, internal nodes are nonterminals
- *Parse tree for previous example:*
- A given element of a language generated by a CFG may have multiple parse trees



Ambiguous grammars

- Sometimes more than one rule can be applied.
- *Ambiguous* grammars are those that define languages with some elements that have two or more different parse trees or derivations
- Choosing rightmost possible nonterminal to expand produces *rightmost derivation*
- Use of leftmost or rightmost derivations can resolve ambiguity

Parsing algorithms

- *Table-driven (bottom-up) parsing*: Used in many compilers and parser generators, e.g., YACC, Bison
- *Top down (recursive-descent) parsing*: one recursive method recognizes each element corresponding to one production
 - This method parses a string, consuming its symbols as it proceeds
 - *Option*: method may build a parse tree for the production recognized

Top-down parsing example

- Productions:

$Bal \rightarrow \lambda, Bal \rightarrow 0 Bal 1$

- Parsing algorithm:

Parse-Bal(x)

If $x = \lambda$

return *true*

if $x[1] = '0'$ and $x[length(x)] = '1'$

return *Parse-Bal* ($x[2..length(x) - 1]$)

else

return *false*

Returns *true* iff
string x is in $L(G)$
of example C,
else *false*

Parsing and compilation

- Lexical analyzer separates a sentence or program into *tokens* (lexemes)
- Parser “discovers the structure of a program”
- Compiler checks syntax with lexer and parser
- Parser builds parse tree from sequence of lexemes
- Compiler generates assembler or machine or byte code from a program’s parse tree

CFG problems

Define a CFG for $L =$

1. $0^n 1^{2n}$
2. $\{ x : n_0(x) = n_1(x) \}$
3. $\{ x / n_1(x) > n_0(x) \}$
4. $\{ 0^m 1^n / n > m \}$

4. Properties of PDAs, CFLs, CFGs

- PDAs and CFGs define the same set of languages
- Closure properties of CFL
- Pumping Lemma for CFLs
- Decidable and undecidable properties of PDAs and CFGs

Languages of CF Grammars

- *Language generated by a grammar:*

$$L(G) = \{ w \in \Sigma^* \mid S \Rightarrow_G^* w \}$$
- *Set of languages of all CF grammars:*

$$\mathcal{L}(\text{CFG}) = \{ L \mid \exists G \text{ s.t. } L = L(G), G \text{ is CF} \}$$
- *To prove about CFGs and CFLs:*
 - $\mathcal{L}(\text{CFG}) = \text{CFL}$

$$= \{ L \mid L = L(M), M \in \text{PDA} \}$$
 - i.e., that CFGs define the same set of languages as PDAs

$\mathcal{L}(\text{CFG}) = \text{CFL}$

Theorem: The languages defined by CFGs are precisely the CFLs (i.e, those accepted by some PDA)

Proof

1. $(\forall G \in \text{CFG})(\exists M \in \text{PDA}) L(M) = L(G)$

CFG \rightarrow PDA construction:

Construct a PDA M that works as follows:

- When a nonterminal X is found on stack,
if the CFG has the production $X \rightarrow YZ\dots$,
 M pops X and pushes $YZ\dots$; otherwise rejects
- When a terminal is found on the stack, M gets
next input a and pops stack if a is on stack;
otherwise rejects input

PDA \rightarrow CFG construction

2. $(\forall M \in \mathcal{PDA})(\exists G \in \mathcal{CFG}) L(M) = L(G)$

Reverse the process used in proof of
CFG \rightarrow PDA, expressing

- stack content as variable part of production
- Input as prefix in production

Example:

Construct G from A (PDA Example A) as

$$S \rightarrow 0 S 1, S \rightarrow \lambda$$

Properties of CFLs

- A class of languages is said to be *closed under an operation* if all results of the operation are also in the class
- *Closures:*
 - Kleene star: $(L \in \mathcal{CFL}) \Rightarrow (L^* \in \mathcal{CFL})$
 - Concatenation:
 $(L_1, L_2 \in \mathcal{CFL}) \Rightarrow (L_1 L_2 \in \mathcal{CFL})$
 - Union: $(L_1, L_2 \in \mathcal{CFL}) \Rightarrow (L_1 \cup L_2 \in \mathcal{CFL})$
- CFLs are *not* closed under intersection or complement

Pumping Lemma for CFLs

- Lemma is similar to that for RLs; any $x \in L$ for CFL L may be rewritten $uvwyz$, with $uv^kwy^kz \in L$ for any k
- Example: $a^n b^n c^n \notin CFL$
- This language can be recognized by a simple algorithm
- That this decidable language is not CF implies that a computational hierarchy of at least three levels exists: to recognize RLs, CFLs, and other decidable languages

Algorithmically decidable properties of PDAs, CFGs

- Given PDA A , whether A will accept a string (i.e., $x \in L(A)$) is decidable
- Given CFG G , there exist algorithms to decide the following about its language:
 - $(x \in L(G))$
 - $L(G) = \emptyset$
 - $L(G)$ is finite

Undecidable properties

- Given PDAs A_1 and A_2 , $(L(A_1) = L(A_2))$ is undecidable
- Given CFG G , these are undecidable:
 - the complement of $L(G)$ is CF
 - G is ambiguous
 - $L(G)$ is regular; $L(G) \supseteq L$ given RL L
- Given CFGs G_1 and G_2 , it is undecidable whether $(L(G_1) = L(G_2))$; $(L(G_1) \subseteq L(G_2))$
- $(L(G_1) \cap L(G_2) \text{ is CF})$ is undecidable

Questions

- What most stayed in your mind in discussing this topic?
- For you, what was the *least* clear concept that you encountered in this topic?

References

Daniel I. A. Cohen. *Introduction to Computer Theory*, 2nd Ed. Wiley, 1997.

John Hopcroft, Rajeev Motwani, Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*, 3rd Ed. Addison-Wesley, 2007.

Peter Linz. *An Introduction to Formal Languages and Automata*, 4th Ed. Jones and Bartlett, 2006.