University of Waterloo
CS 241 Winter 2018
Review

Formal definition of DFA:

A DFA is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$ where:

- finite alphabet Σ
- finite set of states Q
- start state $q_0 \in Q$
- set of final/accepting states $A \subseteq Q$
- transition function $\delta : Q \times \Sigma \rightarrow Q$

Formal definition of NFA:

A NFA is a 5-tuple (Σ, Q, q_0, A, T) where:

- finite alphabet Σ
- finite set of states Q
- start state $q_0 \in Q$
- set of final/accepting states $A \subseteq Q$
- transition relation $T : Q \times \Sigma \rightarrow 2^Q$

For example, if $Q = \{A, B\}$, then $2^Q = \{\{\epsilon\}, \{A\}, \{B\}, \{A, B\}\}$. $|2^Q| = 2^Q$.
NFA to DFA: subset construction

ϵ-NFA to NFA:

1. take ϵ shortcuts; replace ϵx with x $\forall x \in \Sigma$
2. pull back final states
3. remove ϵ transitionis
4. remove dead states (states that have no transitions into it)

Scanning: the Simplified Maximal Munch Algorithm

1. Start at the start state of the scanning DFA.
2. Read characters until an error is reached or input is exhausted, keep tracking of the current state and the previous state. In addition, keep track of the characters read.
3. If the input is exhausted and the current state is an accepting state, emit the characters read as a token. Otherwise, signal an error in tokenizing.
4. If an error is reached and the previous state is an accepting state, emit the characters read (excluding the one that caused the error) as a token. Then resume from step 1 using the remaining input (starting at the character that caused the error) with both the current and previous states set to the start state of the scanning DFA.

Regular expressions:

RE is

- $\emptyset$, or
- ϵ, or
- a, where $a \in \Sigma$, or
- E_1E_2 where E_1, E_2 are REs (series), or
- $E_1|E_2$ where E_1, E_2 are REs (parallel), or
- E^* where E is RE (feedback)

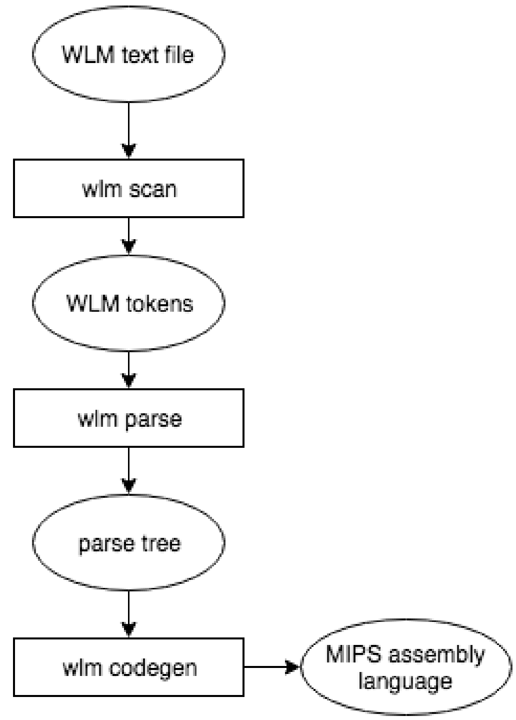
Definition of Regular Language:

A regular language is a language which is

- sepcified by a regular expression
- recognized by an ϵ-NFA
- recognized by an NFA
- recognized by a DFA

Compilation:

- Lexical analysis: scanning/tokenizing
- Syntactic analysis: parsing
- Context-sensitive (semantic) analysis
- Synthesis (code generation)



Context-free grammar(CFG):

G: CFG
$L(G)$: set of words specified by G(i.e. the language specified by G)
N: finite set of non-terminals (non-ending)
T: finite set of terminals (ending)
P: finite set of production rules (rewriting rules)
S: $S \in N$, start symbol

We say that $\alpha A \beta$ directly derives $\alpha\gamma\beta$ if there exists a production rule $A \to \gamma$. Also called a *derivation step.*
We say that $\alpha A \beta$ derives $\alpha\gamma\beta$ if there exist two or more derivation steps such that $\alpha A \beta \to^* \alpha\gamma\beta$.

Leftmost derivation: when there are 2 or more non-terminals in a derivation, we pick the leftmost non-terminal.
Rightmost derivation: when there are 2 or more non-terminals in a derivation, we pick the rightmost non-terminal.

Ambiguity in CFGs:

A string x is ambiguous if $x \in L(G)$ and there are more than one parse trees for x.
A CFG G is ambiguous if some word $w \in L(G)$ is ambiguous.
A grammar is ambiguous if there is a word x such that x has

1. ≥ 2 different parse trees, or
2. ≥ 2 different leftmost derivations, or
3. ≥ 2 different rightmost derivations

left recursion: leftmost symbol of the RHS is the LHS ($E \to E + B$)
right recursion: rightmost symbol of the RHS is the LHS ($E \to B + E$)
Related to associativity.

Top-down parsing with a stack:

Invariant: derivation = input already read + stack (stack is read from the top-down)
Use a rule: pop the stack which has the LHS non-terminal, push RHS in reverse onto the stack.
Accept the input when simultaneously stack and input are empty.
The orcale: Predict$(A, x) = A \to \alpha$
A is on top of the stack, and x is the first symbol of input to be read.

LL(1) Grammar:
$\forall$ non-terminal $A \in N, x \in T,\ |\text{Predict}(A, x)| \leq 1$.
L: Left-to-right input
L: leftmost derivation
1: one token of "lookahead" (in terms of input)

Constructing a Predictor Table: form search trees (search until the first terminal symbol)
Algorithm:
$\alpha, \beta \in (N \cup T)^*,\ x, y \in T, A \in N$

```
Empty(α) = true if α →* ϵ
// can α disappear?
First(α) = {x | α →* xβ}
// starting from α, what can I generate as a first terminal symbol?
Follow(A) = {y | S' →* αAyβ}
// starting from the start symbol, does the terminal y ever appear following the non-terminal A?
Predict(A,x) = {A → α | x ∈ First(α)} ∪ {A → β | x ∈ Follow(A) and Empty(β)}
```

Algorithm for parsing:

```
Input: w
Push S'
for each x ∈ w
   while(top of stack is some A ∈ N){
      pop A
      if Predict(A,x) = {A → α}
         push α
      else
          reject
   }
   pop c
   if c ≠ x reject
end for
accept w
```

Bottom-up parsing with a stack:
stack + input to be read = current derivation (stack is read from bottom to top)
shift: push
reduce($A \to ab$): pop RHS, push LHS
oracle: in the form of a DFA

LR(0) ϵ-NFA formal definition:
Given a CFG $G = (N, T, P, S)$, construct an ϵ-NFA$(Q, N \cup T, q_0, F, D)$ as follows:

- $Q = \{A \to \alpha \bullet \beta | A \to \alpha\beta \in P\}$
- $q_0 = \{S' \to \vdash \bullet S \dashv\}$

- $D[A \to \alpha \bullet X\beta, X] = \{A \to \alpha X \bullet \beta\}$
- $D[A \to \alpha \bullet X\beta, \epsilon] = \{B \to \bullet\gamma | B \to \gamma \in P\}$
- $F = \{S' \to \vdash S \bullet \dashv\}$

2 actions:
If you have stack K and input a, and Ka is recognized, you can **shift**.
If you have stack K and input a, and the top of K is a state containing $A \to \alpha\bullet$ and a can follow A, **reduce** $A \to \alpha\bullet$.
If more than one of these are defined, you have a **conflict**.

Building an LR(0) automaton:
An item is a production with a dot ($\bullet$) somewhere on the RHS (which indicates a partially completed rule).
How to construct:

- make the start state the first rule, with the dot ($\bullet$) in front of the leftmost symbol of the RHS
- for each state, label an arc with the symbol that follows $\bullet$ and advance the $\bullet$ one position to the right in the next state
- if the $\bullet$ precedes a non-terminal, add all productions with that non-terminal on the LHS to the current state, with the $\bullet$ in the leftmost position

Using the automaton:

- If there is a transition out of our current state on the current input, then *shift* (push) that input onto the stack
- We know we can *reduce* if the current state has only one item and the $\bullet$ is the rightmost symbol
- To *reduce*, pop the RHS off the stack, reread the stack (from the bottom-up), follow the transition for the LHS and push the LHS onto the stack

Conflict: shift-reduce, reduce-reduce
If such conflicts are present, the grammar is not LR(0)
Adding a lookahead token to the automaton fixes the conflict.
For each $A \to \alpha$, attach Follow(A).
Increasing efficiency: store (state, input) on stack, and start the transducer at the top of the stack.

Outputting a derivation (parse tree):

– Create a "tree stack"

– Each time we reduce, pop the RHS nodes from tree stack

– Push the LHS node and make its children the nodes we just popped

Traversing the tree twice:
- Symbol table: variable values
- Detect semantic errors
Context-sensitive analysis:
Input: variable names, procedure names, parse tree (syntactically valid)
Output: ERROR if there are any context-sensitive errors; (decorated) parse tree otherwise

Selected problems from tutorials:

Write context-free grammars for the following languages:

(a) The language $L = \{\text{my, name, is, inigo, montoya}\}$.
$S \to$ my
$S \to$ name
$S \to$ is
$S \to$ inigo
$S \to$ montoya

(b) The language of all words over $\Sigma = \{a, b, c\}$ not beginning with b.
$S \to aA$
$S \to cA$
$A \to aA$
$A \to bA$
$A \to cA$
$A \to \epsilon$
$S \to \epsilon$

(c) The language defined by the regular expression $(0|1)^*((00)^*1)^*010$.
$S \to AB010$
$A \to 0A$
$A \to 1A$
$A \to \epsilon$
$B \to C1B$
$B \to \epsilon$
$C \to 00C$
$C \to \epsilon$

(d) The language $L = \{a^n b^n c^m d^m : \ n, m \in \mathbb{N}\}$, where a^n means "n copies of a in a row".

$S \to AB$
$A \to aAb$
$A \to \epsilon$
$B \to cBd$
$B \to \epsilon$

(e) The language of palindromes over $\Sigma = \{a, b\}$.

$S \to a$
$S \to b$
$S \to \epsilon$
$S \to aSa$
$S \to bSb$
$S \to \epsilon$

Not on final but interesting stuff:

LR(1) but not LALR(1) and not SLR(1):
$S \to AaBa$
$S \to BbAb$
$A \to c$
$B \to c$
After building the LR(0) state set for this grammar, get:
$A \to c$
$B \to c$
The SLR(1) construction uses the follow set for A, $\{a, b\}$, to determine whether to reduce $A \to c$ and the follow set for B, $\{a, b\}$, to determine whether to reduce $B \to c$. Since these sets intersect there's a reduce/reduce conflict.
LR(1) is clever enough to build two different states, one containing
$A \to c\ \{a\}$
$B \to c\ \{b\}$
and the other containing
$A \to c\ \{b\}$
$B \to c\ \{a\}$

LR(1) and LALR(1) but not SLR(1):
$S \to AaBa$
$S \to BbAb$
$S \to c$
$A \to c$
$B \to c$
For LALR(1), get states:
$A \to c$
$B \to c$
and
$A \to c$
$B \to c$
$S \to c$
If you look at the grammar carefully, you'll see that the only reductions that make sense are:
$A \to c\ \{b\}$
$B \to c\ \{a\}$
and
$A \to c\ \{a\}$
$B \to c\ \{b\}$
$S \to c\ \{\mathrm{EOF}\}$
But SLR(1) is not powerful enough to make this inference as it uses the same lookahead set for each reduce action, irrespective of the state that it occurs in.