CS 241 Lecture 12

Graham Cooper

June 15th, 2015

recall: (see diagram)
Input abab could be 1,2,3 or 4 tokens.
Sol'n - take the ϵ-move only when no other choice. ie. match the longest token possible
Could fail, eg L = {aa,aa}, w = aaaa

Concrete Realization: Maximal Munch Algorithm

Run DFA (no ϵ-moves) until no non-error move is possible.
If in an accepting state, token found
else, back up to the most recent accepting state.
– – input to that point is the new token, resume scanning from here
end if
Output token; ϵ move back to q_0

We would use a variable to keep track of the most recent accepting state.

Simplified Maximal Munch Algorithm

As above, but:
If not inan accepting state when no transitions are possible, error, (ie. ignore backtracks)

Example: Tokens: must start and end with a letter, can contain -
operator: –
Take: "ab–," as input

once we scan to the second "-", no further move is possible but ab– is not a valid token so we are not in an accepting state. Simplified MM: ERROR
MM: Back up to the previous accepting state (a,b) and then scan from there, we then have tokens (ab) and (–)

MM Works, but Simplified MM does not.

In practice, simplified MM is usually good enough.

Example: C++:
vector<vector<int>> v;
C++ scans this as one token >> rather than as two >'s

C++ Must seperate tokens by space.
Vector <vector<int> > v;

What (if any) specific features of c (or scheme) programs cannot be verified with a DFA?

Consider $\Sigma = \{(,)\}$
$L = \{w \in \Sigma|$ w is a string with alanced parens$\}$.
eg:
$\epsilon \in L$
$() \in L$
$()() \in L$
$(()) \in L$
... $)(\notin L$

Can we build a DFA for L?

See picture

Each new state recognizes one more level of nesting. But no finite number of states recognizes all levels of nesting. And DFA's have a finite number of states.

Context Free Languages

Context free languages are languages that can be described by a context-free grammar.

Intuition

Balanced parens.

- $S \to \epsilon$ "A word in the language is either empty"
- $S \to (S)$ or a word in the language surrounded by ( )
- $S \to SS$ or the concatentation of two words in the language

Shorthand:
$S \to \epsilon|(S)|SS$

Show: This system generates (())()
$S \implies SS \implies (S)S \implies ((S))S \implies (())S \implies (())(S) \implies (())()$

Notation:
" $\implies$ " $\equiv$"derives"
"$\alpha \implies \beta$" means β can be obtained from α by one application of a grammar rule

Formal Definition

A context free grammar consists of:

- An alphabet Σ of terminal symbols
- A finite non-empty set N of Non-terminal symbols N$\cap\Sigma = \emptyset$ (we use V ("vocabulary") to denote $N \cup \Sigma$)
- A finite set p of productions which have the form: $A \to B$ where $A \in N, BinV^*$
- An alement S $\in$ N which is our start symbol

Conventions

a,b,c ... - elements of Σ (characters)
w,x,y,... - elements of Σ^* (strings)
A,B,C ... - elements of N (non-terminals)
S - Usually the start symbol (not always)
α, β, γ... - elements of V^* (ie $\Sigma \cup N)^*$)

We write: $\alpha A \beta \implies \alpha\gamma\beta$ if there is a production A $\rightarrow \gamma$ in P. (RHS derivable from LHS in one step)

$\alpha \implies *\beta$ - means $\alpha \implies ... \implies \beta$ (0 or more steps)

Definitions

L(G) = $\{w \in \Sigma^* | S \implies *w\}$
L(G) is the language specified by G, and Σ^* are the strings of terminals derivable from S.

A language L is <u>Context-free</u> if L = L(G) for some context-free grammar G.

Examples:

Palindromes over {a,b,c}

$S \rightarrow aSa|bSb|cSc|M$
$M \rightarrow \epsilon|a|b|c$

Show: $S \implies *abcba$
$S \implies aSa \implies abSba \implies abMba \implies abcba$ (called a derivation)

Expressions

$\Sigma = \{a, b, c, +, -, *, /\}$
L = {Arithmetic expressions using symbols from Σ }
$S \rightarrow SOpS|a|b|c$

$Op \rightarrow +|-|*|/$

$\Sigma = \{a, b, c, +, -, *, /, (,)\}$ $S \rightarrow SOpS|a|b|c|(S)$
$Op \rightarrow +|-|*|/$

Show: $S \implies *a + b$
$S \implies SOpS \implies aOpS \implies a + S \implies a + b$
We have a choice to expand which part,
Leftmost derivation - we should always expand the leftmost symbol first.
Rightmost derivation - we always expand the rightmost symbol first