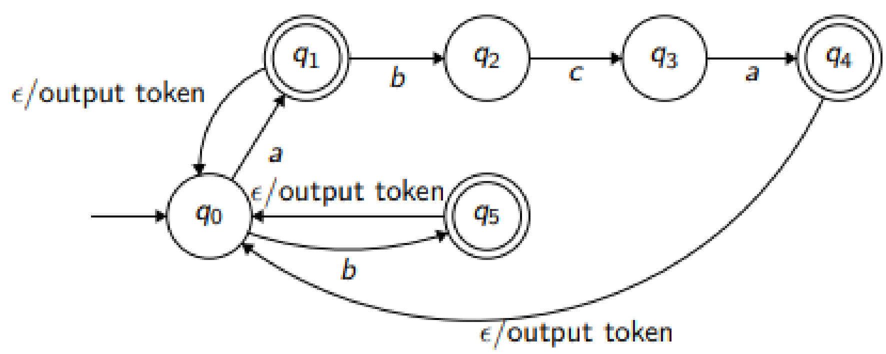
CS 241, Lecture 9 - Maximal Munch and Context Free Grammars

Tues, Feb 05, 2019

1 Maximal Munch and Simplified Maximal Munch

- The general idea is to consume the largest possible token that makes sense, then produce the token and then proceed.
- Maximal munch differs from simplified maximal munch:
  - Maximal munch consumes chars until there is no valid transition. If there are still characters, backtrack to the last valid accepting state and resume.
  - Simplified maximal munch consumes chars until there is no longer a valid transition. If the state is accepting, produce the token and proceed. Otherwise go to error state.
- Observe the following DFA for w = "ababca":

  $\Sigma = \{a, b, c\}$, $L = \{a, b, abca\}$, consider $w = ababca$.
  Note that $w \in LL^*$. What follows is an ϵ-NFA for LL^* based on our algorithm:



- With SMM, when we hit the second "a", we stop and output an error, which is not the correct answer.
- With MM, the second "a" is reached, and the last accepting state was "a". We backtrack back to "a" and resume munch (go back to q_0, resume by consuming "b". We would have to keep track of the last accepting state.
- Note SMM is usually good enough and thus we typically use this without issue - that is, we make our language work with SMM rather than the other way around.

- Examples of the MM and SMM algorithms:

Algorithm for Maximal Munch

Algorithm 1 Maximal Munch

```
s = q0, ta = ε, tcur = ε
str = input, pos = 0, posAccepting = −1
while pos! = len(str) do
    c = str[pos], s = δ(s, c), tcur = tcur + c
    if s == ERROR then
        if ta = ε then
            Fatal Error
        end if
        Output ta
        s = q0, ta = ε, tcur = ε, pos = posAccepting
    else if s ∈ A then
        posAccepting = pos
        ta = tcur
    end if
    pos = pos + 1
end while
if s ∈ A then
    Output t and Accept
else
    Reject/Crash/Fatal Error
end if
```

Algorithm for Simplified Maximal Munch

Algorithm 2 Simplified Maximal Munch

```
s = q0
ta = ε
while not EOF do
    c = read_char()
    if δ(s, c) == ERROR then
        if s ∈ A then
            Output ta
            s = q0, ta = ε.
        else
            Reject/Crash/Fatal Error
        end if
    else
        s = δ(s, c)
        ta = ta + c
    end if
end while
if s ∈ A then
    Output ta and Accept
else
    Reject/Crash/Fatal Error
end if
```

- This concludes the process of scanning.

2 Syntactic Analysis - Context-Free Grammars

- Things we have to consider: syntax (is the order of tokens correct, are parentheses balanced?) and semantics (does what is written make sense?).
- A **grammar** is the language of languages - they help us describe what we are and are not allowed to say.
- **Context-free grammars** is a 4-tuple (N, Σ, P, S) where:
  - N is a finite non-empty set of non-terminal symbols (symbols you cannot stop on)
  - Σ is an alphabet, or in other words a set of non-empty terminal symbols, and $N\hat{Z} = \emptyset$.
  - P is a finite set of productions, each of the form $A \to \beta$ where $A \in N$ and $\beta \in (N \cup \Sigma)*$.
  - $S \in N$ is a starting symbol.
- We set $V = N \cup \Sigma$ to denote the vocabulary - the set of *all* symbols in our language.
- For example, in rustcc, we defined various CFGs, such as Fn containing BlockItems which contained Statements or Declarations which contained...
- **Conventions**:
  - Lower case letters from the start of the alphabet are elements of Σ
  - Lower case letters from the end of the alphabet are elements of Σ^* (words)
  - Upper case letters from the start of the alphabet are elements of N
  - S is always our start symbol
  - Greek letters like α, β, γ are elements of $V^* = (N \cup \Sigma)^*$.
- For example, consider $\Sigma = \{(,)\}$, and let $L = \{w : w$ is a balance string of parentheses$\}$. Thus, $S \to \epsilon, S \to (S), S \to SS \Rightarrow S \to \epsilon|(S)|SS$
- A **derivation** over a CFG(N, Σ, P, S) is such that:
  - α derives β and we write $\alpha \Rightarrow \beta$ iff β can be obtained from α using a rule from P.
  - $\alpha A \beta \Rightarrow \alpha\gamma\beta$ iff there is a rule $A \to \gamma$ in P.
  - $\alpha \Rightarrow^* \beta$ iff a derivation exists, that is, there exists $\delta_i \in V^*$ for $0 \le i \le k$ such that $\alpha = \delta_0 \Rightarrow \delta_1 \Rightarrow \cdots \Rightarrow \delta_k = \beta$. Note k can be 0.
- Another example: find a derivation of (()()). Recall our above CFG:

$$\begin{aligned} S &\Rightarrow (S) \Rightarrow (SS) \Rightarrow ((S)S) \\ &\Rightarrow ((S)(S)) \Rightarrow ((\epsilon)(S)) \\ &\Rightarrow (()()) \\ S &\Rightarrow^* (()()) \quad \text{, short form for the above implications} \end{aligned}$$

- Why is it "context-free"? It's because our grammar does not care for the context - that is, it does not care for *where* your symbols are.
- This is the opposite of **context-bounded grammars**, which is where the context of the other symbols around other symbols *will* affect our productions (note we don't *really* need to know about this)
- We define the **language** of a CFG(N, Σ, P, S) to be $L(G) = \{w \in \Sigma^* : S \Rightarrow^* w\}$.
- A language is **context-free** iff there exists a CFG G such that $L = L(G)$.
- Informally, we can show regular languages are context-free:
  1. $\emptyset$: $(\{S\}, \{a\}, \emptyset, S)$
  2. $\{\epsilon\}$: $(\{S\}, \{a\}, S \to \epsilon, S)$.
  3. $\{a\}$: $(\{S\}, \{a\}, S \to a, S)$.
  4. Union: $\{a\} \cup \{b\}$: $(\{S\}, \{a, b\}, S \to a|b, S)$.
  5. Concatenation: $\{ab\}$: $(\{S\}, \{a, b\}, S \to ab, S)$.
  6. Kleene Star: $\{a\}^*$: $(\{S\}, \{a\}, S \to Sa|\epsilon, S)$.