Warm Up Problem

- Write an ϵ-NFA over $\Sigma = \{a, b, c\}$ that accepts $L = \{ab\} \cup \{ab^n c : n \in \mathbb{N}\}^*$.

CS 241 Lecture 9

Maximal Munch and Context Free Grammars

With thanks to Brad Lushman, Troy Vasiga and Kevin Lanctot

Scanning

Is C a regular language? The following are regular:

- C keywords
- C identifiers
- C literals
- C operators
- C comments

Sequences of these are hence also regular. Finite automata can do our tokenization (ie. our scanning).

What about punctuation? Even simpler, set $\Sigma = \{ \text{ (,) } \}$ and $L = \{$ set of balanced parentheses$\}$. Is L regular?

Scanning

Is C a regular language? The following are regular:

- C keywords
- C identifiers
- C literals
- C operators
- C comments

Sequences of these are hence also regular. Finite automata can do our tokenization (ie. our scanning).

What about punctuation? Even simpler, set $\Sigma = \{ \text{ (,) } \}$ and $L = \{$ set of balanced parentheses$\}$. Is L regular? More on this later.

Scanning Continued

How does our scanner work?

- Recall our goal is given some text, break the text up into tokens (eg. (ID, div), (REG, $1), (COMMA, ‘,’) (REG, $2))
- Problem: Some tokens can be recognized in multiple different ways!
- Eg. $0x12cc$. Could be a single HEXINT or could be an INT (0) followed by an ID (x) followed by another INT (1) followed by another INT (2) and followed by an ID (cc). (There are lots of other interpretations).

Which interpretation is correct?

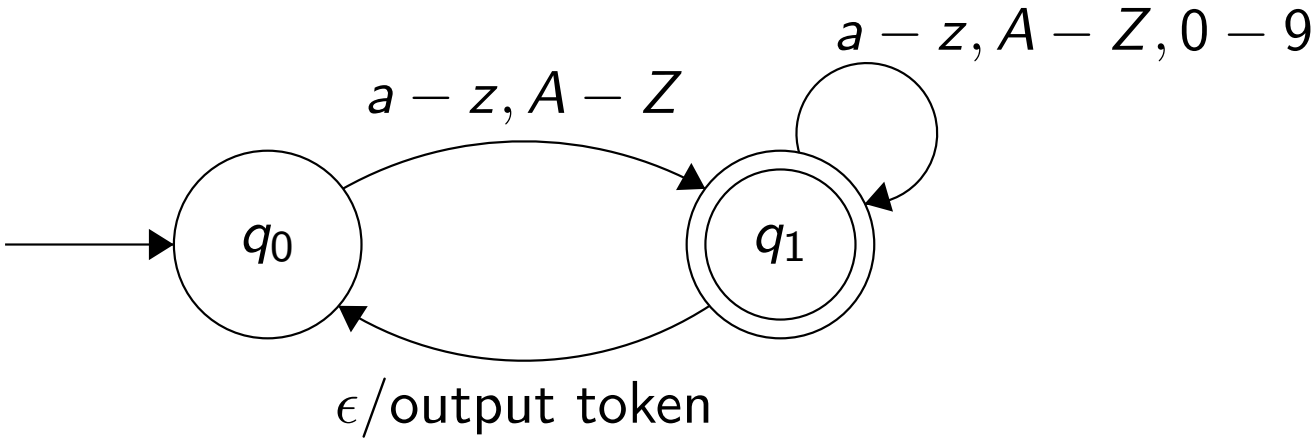
Formalization of our Problem

Given a regular language L (say L is all valid MIPS or C tokens), determine if a given word w is in LL^* (or in other words, is $w \in L^* \backslash \{\epsilon\}$?)

(We don't want to consider the empty program as being valid hence why we drop ϵ).

Concrete Example for C

Consider the language L of just ID tokens in C:



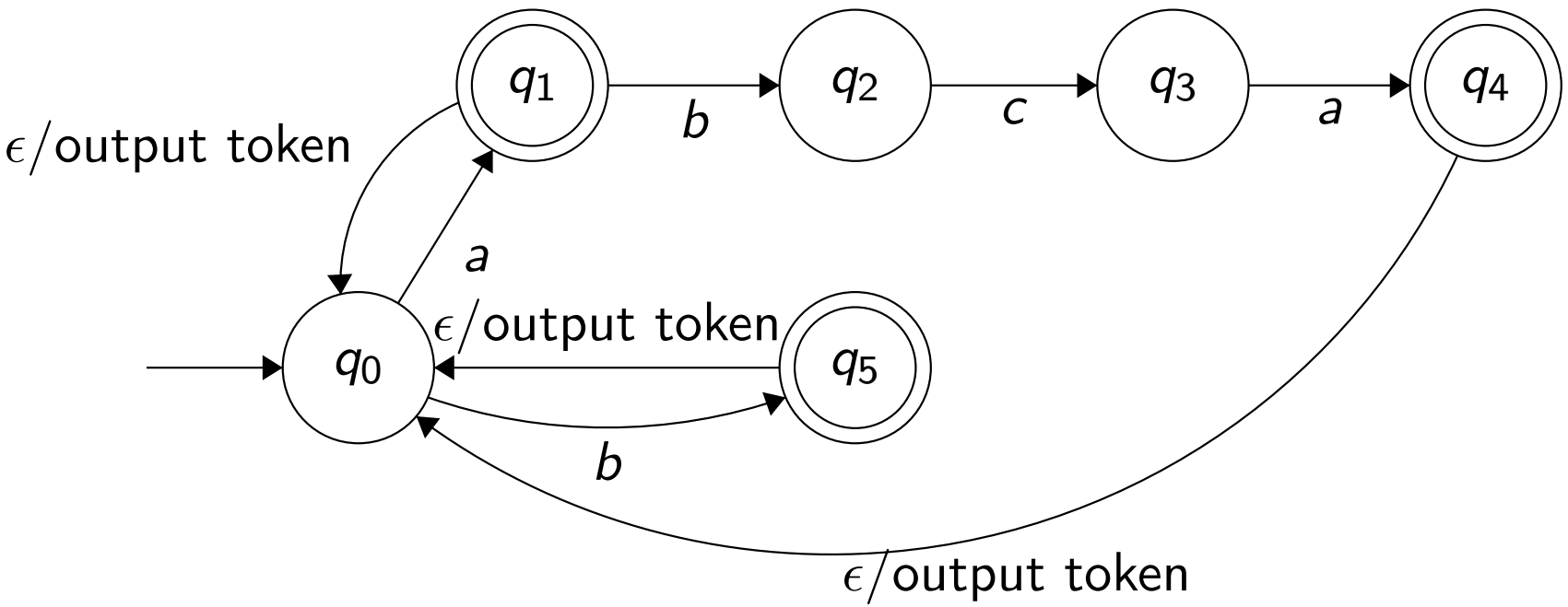
With the input *abcde*, this could be considered as anywhere from 1 to 5 different tokens! What do we do?

What do to?

- We will discuss two algorithms called *maximal munch* and *simplified maximal munch*.
- General idea: Consume the largest possible token that makes sense. Produce the token and then proceed.
- Difference:
  - Maximal Munch: Consume characters until you no longer have a valid transition. If you have characters left to consume, backtrack to the last valid accepting state and resume.
  - Simplified Maximal Munch: Consume characters until you no longer have a valid transition. If you are currently in an accepting state, produce the token and proceed. Otherwise go to an error state.

DFA for next two slides

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



Examples

Maximal Munch: $\Sigma = \{a, b, c\}$, $L = \{a, b, abca\}$, $w = ababca$. Note that $w \in LL^*$.

- Algorithm consumes a **and flags this state as it is accepting**, then b then tries to consume a but ends up in an error state.
- Algorithm then backtracks to first a since that was the last accepting state. Token a is output.
- Algorithm then resumes consuming b **and flags this state as it is accepting**, then tries to consume a but ends up in an error state.
- Algorithm then backtracks to first b since that was the last accepting state. Token b is output.
- Algorithm then consumes the second a, the second b, the first c, the third a and runs out of input. This last state is accepting so output our last token $abca$ and accept.

Examples

Simplified Maximal Munch: $\Sigma = \{a, b, c\}$, $L = \{a, b, abca\}$, $w = ababca$. Note that $w \in LL^*$.

- Algorithm consumes a, then b then tries to consume a but ends up in an error state. **Note there is no keeping track of the first accepting state!**
- Algorithm then checks to see if ab is accepting. It is not (as $ab \notin L$).
- Algorithm rejects $ababca$.
- Note: This gave the wrong answer! However this algorithm is usually good enough and is used in practice.

Practical Implications

Consider the following $C++$ line:

```
vector<pair<string,int>> v;
```

Notice that at the end, there is the token >>! This on its own is a valid token! With Simplified Maximal Munch, we would actually reject this declaration. This would have been the case in versions of C++ compliers before C++11. How did they avoid this problem?

Practical Implications

Consider the following $C++$ line:

```
vector<pair<string,int>> v;
```

Notice that at the end, there is the token >>! This on its own is a valid token! With Simplified Maximal Munch, we would actually reject this declaration. This would have been the case in versions of C++ compliers before C++11. How did they avoid this problem? Requiring an extra space after the first > character!

```
vector<pair<string,int> > v;
```

The above code would be valid in C++03 or C++11 (for example).

Algorithm for Maximal Munch (Single Backtrack)

Algorithm 1 Maximal Munch Single Backtrack

1: $s = q_0$, $t_a = \epsilon$, $t_{cur} = \epsilon$, $t_{lastacc} = \epsilon$, $s_a = NULL$.
2: **if** $s \in A$ **then**
3: $\quad s_a = s$.
4: **end if**
5: **while** not EOF **do**
6: $\quad$ **if** $t_{lastacc} != \epsilon$ **then**
7: $\quad\quad c = t_{lastacc}[0]$, $t_{lastacc} = t_{lastacc}[1..n]$, $s = \delta(s, c)$
8: $\quad$ **else**
9: $\quad\quad$ c = read_char(), $s = \delta(s, c)$
10: $\quad$ **end if**
11: $\quad t_{cur} = t_{cur} + c$
12: $\quad$ **if** $s ==$ ERROR **then**
13: $\quad\quad$ **if** $s_a ==$ NULL **then**
14: $\quad\quad\quad$ Fatal Error
15: $\quad\quad$ **end if**
16: $\quad\quad$ Output t_a, epsilon transition back to q_0
17: $\quad\quad s = \delta(q_0, c)$, $t_a = \epsilon$, $t_{lastacc} = t_{cur}$, $t_{cur} = \epsilon$
18: $\quad$ **else if** $s \in A$ **then**
19: $\quad\quad s_a = s$, $t_a = t_a + t_{cur}$, $t_{lastacc} = \epsilon$, $t_{cur} = \epsilon$
20: $\quad$ **end if**
21: **end while**
22: **if** $s \in A$ **then**
23: $\quad$ Output t_a and Accept
24: **else**
25: $\quad$ Reject/Crash/Fatal Error
26: **end if**

Algorithm for Simplified Maximal Munch

Algorithm 2 Simplified Maximal Munch

```
1:  s = q_0
2:  t_a = ε
3:  while not EOF do
4:      c = read_char()
5:      if δ(s, c) == ERROR then
6:          if s ∈ A then
7:              Output t_a
8:              s = q_0, t_a = ε.
9:          else
10:             Reject/Crash/Fatal Error
11:         end if
12:     else
13:         s = δ(s, c)
14:         t_a = t_a + c
15:     end if
16: end while
17: if s ∈ A then
18:     Output t_a and Accept
19: else
20:     Reject/Crash/Fatal Error
21: end if
```

Check In

Where are we now?

1. Identify tokens (Scanning) [Complete!]
2. Check order of tokens (Syntactic Analysis) [Now]
3. Type Checking (Semantic Analysis) [Later]
4. Code Generation [Also later]

- Syntax: Is the order of the tokens correct? Do parentheses balance?
- Semantics: Does what is written make sense (right type of variables in functions etc.)

Our Motivating Example

Consider $\Sigma = \{(,\ )\}$ and
$L = \{w : w \text{ is a balanced string of parentheses}\}$.
Is this language regular? Can we build a DFA for L?

Our Motivating Example

Consider $\Sigma = \{(,\ )\}$ and
$L = \{w : w \text{ is a balanced string of parentheses}\}$.
Is this language regular? Can we build a DFA for L?

You can probably convince yourself fairly quickly that this should be impossible - once you have arbitrarily large levels of open parentheses, it is tough to be able to know how many (symbols you have processed after seeing so many characters *without some kind of memory tool*.

Definitions

- A *grammar* is the language of languages. (Matt Might)
- In some sense, grammars help us to describe what we are allowed and not allowed to say.
- Context-free grammars are a set of rewrite rules that we can use to describe a language.

Grammar Example

The following is a CFG for C++ compound statements:

```
<compound stmt> --> { <stmt list> }
<stmt list> --> <stmt> <stmt list> | epsilon
<stmt> --> <compound stmt>
<stmt> --> if ( <expr> ) <stmt>
<stmt> --> if ( <expr> ) <stmt> else <stmt>
<stmt> --> while ( <expr> ) <stmt>
<stmt> --> do <stmt> while ( <expr> ) ;
<stmt> --> for ( <stmt> <expr> ; <expr> ) <stmt>
<stmt> --> break ; | continue ;
<stmt> --> return <expr> ; | goto <id> ;
```

Source: https://www.cs.rochester.edu/~nelson/courses/csc_173/grammars/cfg.html

Formal Definition

Definition

A **Context Free Grammar (CFG)** is a 4 tuple (N, Σ, P, S) where

- N is a finite non-empty set of *non-terminal symbols*.
- Σ is an alphabet; a set of non-empty *terminal symbols*.
- 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

Note: We set $V = N \cup \Sigma$ to denote the *vocabulary*, that is, the set of all symbols in our language.

Conventions

- Lower case letters from the start of the alphabet, say $a, b, c, \ldots$ are elements of Σ
- Lower case letters from the end of the alphabet, say w, x, y, z are elements of Σ^* (words)
- Upper case letters from the start of the alphabet, say $A, B, C, \ldots$ are elements of N (*non-terminals*)
- S is always our start symbol.
- Greek letters, α, β, γ are elements of V^* (recall this is $(N \cup \Sigma)^*$)

Example

Let's revisit $\Sigma = \{(,\ )\}$ and
$L = \{w : w \text{ is a balanced string of parentheses}\}$.

$$
\begin{aligned}
S &\rightarrow \epsilon \\
S &\rightarrow (S) \\
S &\rightarrow SS
\end{aligned}
$$

We can also write this using a short hand:

$$S \rightarrow \epsilon \mid (S) \mid SS$$

Example

Find a derivation of (()()). Recall our CFG: $S \to \epsilon \mid (S) \mid SS$

Definition

Over a CFG (N, Σ, P, S), we say that...

- A *derives* γ and we write $A \Rightarrow \gamma$ if and only if there is a rule $A \to \gamma$ in P.
- $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if and only if there is a rule $A \to \gamma$ in P.
- $\alpha \Rightarrow^* \gamma$ if and only if a *derivation* exists, that is, there exists $\delta_i \in V^*$ for $0 \leq i \leq k$ such that $\alpha = \delta_0 \Rightarrow \delta_1 \Rightarrow ... \Rightarrow \delta_k = \gamma$. Note that k can be 0.

Example

Find a derivation of (()()). Recall our CFG: $S \to \epsilon \mid (S) \mid SS$

Definition

Over a CFG (N, Σ, P, S), we say that...

- A *derives* γ and we write $A \Rightarrow \gamma$ if and only if there is a rule $A \to \gamma$ in P.
- $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if and only if there is a rule $A \to \gamma$ in P.
- $\alpha \Rightarrow^* \gamma$ if and only if a *derivation* exists, that is, there exists $\delta_i \in V^*$ for $0 \leq i \leq k$ such that $\alpha = \delta_0 \Rightarrow \delta_1 \Rightarrow ... \Rightarrow \delta_k = \gamma$. Note that k can be 0.

Solution:

$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow ((S)S) \Rightarrow (()S) \Rightarrow (()(S)) \Rightarrow (()())$.

Hence $S \Rightarrow^* (()())$.

Aside: Why Context-Free

Why are these called *context-free* grammars? Are there *context-bounded* grammars?

Aside: Why Context-Free

Why are these called *context-free* grammars? Are there *context-bounded* grammars?

Yes! These are actually called **context sensitive** grammars. These involve multiple symbols of LHS production rules. For example:

For example, let $\Sigma = \{a, b, c\}$ and $L = \{a^n b^n c^n : n \in \mathbb{N}\}$.

$$
\begin{aligned}
S &\rightarrow \epsilon \mid abc \mid aAbc \\
Ab &\rightarrow bA \\
Ac &\rightarrow Bbcc \\
bB &\rightarrow Bb \\
aB &\rightarrow aa \mid aaA
\end{aligned}
$$

(Idea: Move A to the right then move B to the left and you add an a, b and c in the correct places each time.)

Final Definitions

Definition

Define the *language of a CFG* (N, Σ, P, S) to be $L(G) = \{w \in \Sigma^* : S \Rightarrow^* w\}$.

Definition

A language is *context-free* if and only if there exists a CFG G such that $L = L(G)$.

Example: Every regular language is context-free! (Why?)

Regular Languages are Context-Free (Informally)

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: $\{a\}^*$: $(\{S\}, \{a\}, S \to Sa|\epsilon, S)$.

Practice

Let $\Sigma = \{a, b\}$. Find a CFG for each of the following:

- $a(a|b)^*b$ (can also write as $a(a+b)^*b$)
- $\{a^n b^n : n \in \mathbb{N}\}$
- Palindromes over $\{a, b, c\}$

Further, for the second language, find a derivation in your grammar of *aaabbb*.