Extending Your Assembler, Regular Languages

University of Waterloo

February 5, 2020
• In Assignment 3, your assembler only needs to handle `.word` directives. In Assignment 4, you must extend it to support the 17 other instructions in our dialect of MIPS.
• For Assignment 3, simply using nested if statements to check the format of a `.word` directive will work fine, but checking instructions with multiple operands this way is tedious and prone to error.
• One way to check instructions with multiple operands have the correct format is to use a pattern-matching approach. For example:

```java
Pattern BeqBnePattern {REG, COMMA, REG, COMMA, NUM_OR_ID};
...
if(token.lexeme == "beq" || token.lexeme == "bne") {
    if(match(tokenLine, startPosition, BeqBnePattern)) {
        // process instruction
    } else {
        // throw an error
    }
}
```

• How is “Pattern” defined?
• Idea: define sets of token kinds, and then let a pattern be a list of these sets.

• In C++ we can use typedef to create aliases for the token set and pattern types:

  ```cpp
typedef std::set<Token::Kind> TokenSet;
typedef std::list<TokenSet> Pattern;
```

  This lets us type `Pattern` instead of `std::list<std::set<Token::Kind>>` when declaring a pattern.

• You can use C++ initializer list syntax to concisely define sets and patterns:

  ```cpp
  TokenSet REG {Token::REG};
  TokenSet COMMA {Token::COMMA};
  TokenSet NUM_OR_ID {Token::INT, Token::HEXINT, Token::ID};

  Pattern AddSubSltSltuPattern {REG, COMMA, REG, COMMA, REG};
  Pattern BeqBnePattern {REG, COMMA, REG, COMMA, NUM_OR_ID};
  ```
Once the patterns are set up, write a pattern-matching function:

```cpp
bool match(std::vector<Token>& line, int start, Pattern pattern)
```

Loop over the tokens on the line, starting from the start position, and check if each token’s kind is in the set specified by the pattern.

We use `std::list` for the Pattern because it has a fast `pop_front` method. As you check each token, pop the front element from the Pattern – when the Pattern is empty you have matched everything.

Make sure to handle error cases:

1. If `start` is past the end of the line & there are no tokens to loop over.
2. If there are fewer tokens past `start` than pattern elements to match.
3. If there are more tokens past `start` than pattern elements to match.

If you are storing instruction data in an Instruction class, instead of returning a bool you could build up an Instruction instance as you match and return a pointer to it (null pointer indicates match failure).
An *alphabet* (denoted $\Sigma$) is a finite non-empty set of symbols. Here are some examples:

- Often in examples, we use subsets of the usual English alphabet: $\{a, b, c\}$.
- An alphabet can have just one symbol: $\{b\}$.
- The symbols can be anything, even things that look like they’re made of multiple symbols: $\{\text{ID}, \text{LABEL}, \text{WORD}, \text{COMMA}, \text{LPAREN}, \text{RPAREN}, \text{INT}, \text{HEXINT}, \text{REG}\}$. We usually don’t define alphabets with “multi-symbol symbols” like this to avoid confusion, although they do come up in practice (the above is the set of token kinds returned by the MIPS scanner).
- Alphabet of hexadecimal digits: $\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$
A \textit{word} or \textit{string} over an alphabet $\Sigma$ is a finite sequence of symbols from $\Sigma$. The sequence with zero symbols in it (the \textit{empty word}) is denoted $\varepsilon$.

- \textit{bac}, \textit{aba}, \textit{c} where $\Sigma = \{a, b, c\}$.
- \textit{b}, \textit{bb}, \textit{bbb} where $\Sigma = \{b\}$.
- ID REG COMMA REG COMMA REG (one word of length six) where $\Sigma = \{$MIPS token kinds$\}$. In this example we added spaces just for readability; the spaces are not part of the alphabet or the word.
- \textit{DEADBEEF}, \textit{FACE}, \textit{C001BABE} where $\Sigma = \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$.
- The empty word $\varepsilon$ is a word over every alphabet.
A *language* over $\Sigma$ is a set of words over $\Sigma$.

- Languages can be finite or infinite.
- Take note of the difference between the empty language $\emptyset$, which contains no words, and the language $\{\varepsilon\}$, which contains one word that has length zero.

A regular language $L$ over an alphabet $\Sigma$ is a set of words satisfying the following recursive definition:

- $L = \emptyset$.
- $L = \{\varepsilon\}$.
- $L = \{a\}$ where $a$ is a word consisting of a single symbol from $\Sigma$.
- $L = R \cup R'$, where $R$ and $R'$ are regular languages. (Union)
- $L = RR'$ where $R$ and $R'$ are regular languages. (Concatenation)
- $L = R^* = \bigcup_{i=0}^{\infty} R^i$, where $R$ is a regular language, $R^0 = \{\varepsilon\}$, and $R^i = RR^{i-1}$ for $i > 0$. (Kleene Star)

It’s equivalent to replace the first three cases by the statement “$L$ is finite”. 
A deterministic finite automaton (DFA) is a 5-tuple \((\Sigma, Q, q_0, A, \delta)\) where:

- \(\Sigma\) is the input alphabet.
- \(Q\) is a finite set of states.
- \(q_0 \in Q\) is the initial or starting state.
- \(A \subseteq Q\) is the set of accepting states.
- \(\delta : Q \times \Sigma \rightarrow Q\) is the transition function.
When drawing DFA diagrams, here is the correspondence between elements of the diagram and elements of the formal definition:

- The states in $Q$ are drawn as circles.
- The initial state $q_0$ has an unlabeled arrow pointing into it.
- The accepting states in $A$ are marked by drawing an extra inner circle.
- The transition function is represented as follows: if $\delta(q, a) = q'$ for states $q$ and $q'$ and a symbol $a$, draw an arrow between $q$ and $q'$ labelled with $a$.

Sometimes drawing every transition in a DFA diagram becomes messy. Thus we have a convention for omitting transitions. If a transition is not drawn on the diagram, we assume that it goes to an “error” state, which is also not drawn. The error state is non-accepting, and it is impossible to leave the error state:

$$\delta(\text{error}, a) = \text{error}, \text{ for all } a \in \Sigma.$$
DFA Problems

Draw DFA diagrams for the following languages:

1. The language of strings over $\Sigma = \{a, b\}$ that contain an even number of $a$'s and an odd number of $b$'s.

2. The language of strings over $\Sigma = \{a, b, c\}$ that contain exactly one $a$ and an even number of $c$'s, with no restriction on the number of $b$'s.

3. The language of strings over $\Sigma = \{0, 1\}$ that end in 1011. How would the solution change if 1011 could appear anywhere in the string?

4. The language of strings over $\Sigma = \{0, 1, 2, 3\}$ whose digit sum is 3. Leading zeroes are permitted.

5. The language of strings over $\Sigma = \{a, b, c\}$ that end in $cab$ and contain an even number of $a$'s (no restriction on the number of $b$'s or $c$'s).
Draw a DFA for the language of strings over $\Sigma = \{a, b\}$ that contain an even number of $a$’s and an odd number of $b$’s.

- What if we first try writing a program that recognizes these strings?
- This is straightforward to do: (in psuedocode)

```pseudo
even_a_odd_b(input) {
    parity_a = 0
    parity_b = 0
    while(input is non-empty) {
        pop symbol from input
        if(symbol == a) {
            parity_a = (parity_a + 1) mod 2
        } else if (symbol == b) {
            parity_b = (parity_b + 1) mod 2
        }
    }
    return (parity_a == 0 and parity_b == 1)
}
```
even_a_odd_b(input) {
    parity_a = 0
    parity_b = 0
    while(input is non-empty) {
        pop symbol from input
        if(symbol == a) {
            parity_a = (parity_a + 1) mod 2
        } else if(symbol == b) {
            parity_b = (parity_b + 1) mod 2
        }
    }
    return (parity_a == 0 and parity_b == 1)
}

• We can turn this program into a DFA by letting the states represent states of memory during program execution.
• Ignoring the memory needed to store the input and current symbol, we only have two variables: parity_a and parity_b. Each variable has two possible values: 0 or 1.
• This gives four possible states of memory: {
  (0, 0), (0, 1), (1, 0), (1, 1)\}.
even_a_odd_b(input) {
    parity_a = 0
    parity_b = 0
    while(input is non-empty) {
        pop symbol from input
        if(symbol == a) {
            parity_a = (parity_a + 1) mod 2
        } else if(symbol == b) {
            parity_b = (parity_b + 1) mod 2
        }
    }
    return (parity_a == 0 and parity_b == 1)
}

- Our states are: \{(0,0), (0,1), (1,0), (1,1)\}.
- Initial state corresponds to initial variable values: (0,0).
- Accepting state set is set of variable values for which we return true: \{(0,1)\}.
- Transition function is described by the loop body!
DFAs and Programs

Draw a DFA for the language of strings over \( \Sigma = \{a, b, c\} \) that contain exactly one \( a \) and an even number of \( c \)'s, with no restriction on the number of \( b \)'s.

```cpp
one_a_even_c(input) {
    count_a = 0
    parity_c = 0
    while(input is non-empty) {
        pop symbol from input
        if(symbol == a) {
            count_a += 1
            if(count_a > 1) {
                return false;
            }
        } else if(symbol == c) {
            parity_c = (parity_c + 1) mod 2
        }
    } else if(symbol == c) {
        parity_c = (parity_c + 1) mod 2
    }
    return (count_a == 1 and parity_c = 0)
}
```

- Possible values of \( \text{count}_a \) are \( \{0, 1\} \) and possible values of \( \text{parity}_c \) are \( \{0, 1\} \).
- The early return \( \text{false} \) corresponds to an error state!
- We have five states but the error state can be omitted from the diagram.
Draw a DFA for the language of strings over $\Sigma = \{0, 1\}$ that end in 1011.

ends_in_1011(input) {
    ending = ""
    while(input is non-empty) {
        pop symbol from input
        ending += symbol
        if(length(ending) > 4) {
            ending = last 4 characters of ending
        }
    }
    return (ending == "1011")
}

- This program works, but... trying to convert it directly to a DFA would give us a huge number of states because of all the possible string values.
- Can we rewrite this to use fewer states of memory? We can, but the algorithm becomes harder to understand.
Draw a DFA for the language of strings over $\Sigma = \{0, 1\}$ that end in 1011.

```python
def ends_in_1011(input):
    ending = ""
    while(input is non-empty):
        pop symbol from input
        if(ending + symbol is a prefix of "1011") {
            ending += symbol
        } else {
            ending = (longest suffix of ending + symbol
                      that is a prefix of "1011")
        }
    return (ending == "1011")
```

- Now ending only has five possible values since it must always be a prefix of 1011.
- However, we have to be careful when figuring out the transitions.
How would the solution change if 1011 could appear anywhere in the string?

```java
contains_1011(input) {
    seen = ""
    while(input is non-empty) {
        pop symbol from input
        if(seen != "1011") {
            if(seen + symbol is a prefix of "1011") {
                seen += symbol
            } else {
                seen = (longest suffix of seen + symbol
                        that is a prefix of "1011")
            }
        } else {
            return (seen = "1011")
        }
    }
    return (seen = "1011")
}
```

• Same idea but once we’ve seen the string once, we stay in the accepting state forever.
Draw a DFA for the language of strings over $\Sigma = \{0, 1, 2, 3\}$ whose digit sum is 3. Leading zeroes are permitted.

```plaintext
digit_sum_3(input) {
    sum = 0
    while(input is non-empty) {
        pop symbol from input
        sum += symbol
        if(sum > 3) {
            return false;
        }
    }
    return (sum == 3)
}
```
Draw a DFA for the language of strings over $\Sigma = \{a, b, c\}$ that end in $cab$ and contain an even number of $a$’s (no restriction on the number of $b$’s or $c$’s).

```c
ends_cab_even_a(input) {
    ending = ""
    parity_a = 0
    while(input is non-empty) {
        pop symbol from input
        if(symbol == a) {
            parity_a = (parity_a + 1) mod 2
        }
        if(ending + symbol is a prefix of "cab") {
            ending += symbol
        } else {
            ending = (longest suffix of ending + symbol that is a prefix of "cab")
        }
    }
    return (ending == "cab" and parity_a == 0)
}
```

- Following this program will produce a rather messy 8-state DFA. There is a clever way to reduce the number of required states to 5.
You don’t need to know about this relationship between DFAs and programs for the midterm or final exam, since this was not covered in lectures. However, you might find this correspondence useful for coming up with DFAs.
Regular Expressions

- Regular expressions are a means of expressing regular languages more concisely than set notation.
- In the table, $L(R)$ means “the language corresponding to the regular expression $R$”.

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Corresponding Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>${\varepsilon}$</td>
</tr>
<tr>
<td>$a$</td>
<td>${a}$</td>
</tr>
<tr>
<td>$(R</td>
<td>R')$</td>
</tr>
<tr>
<td>$(RR')$</td>
<td>$L(R)L(R')$</td>
</tr>
<tr>
<td>$(R^*)$</td>
<td>$L(R)^*$</td>
</tr>
</tbody>
</table>

- Parentheses can be omitted in regular expressions.
- Order of operations: Kleene star first, concatenation, then union.
- For example, the regular expression $aa|bb^*$ is equivalent to $((aa)|(b(b^*)))$. The corresponding language is $\{aa\} \cup \{b^n : n \geq 1\}$. 
Regular Expression Problems

Describe the following languages using set notation, then give a regular expression for them.

1. The language of binary strings whose second symbol is a “0” and whose fifth is a “1”.

2. The language of binary strings that contain the substring “110101”.

Give a regular expression for each of the following languages.

3. $\Sigma = \{a, b\}$, $L = \{aa, ab, ba, bb\}$.

4. Strings over the alphabet $\Sigma = \{a, b, +, -, \cdot, /\}$ representing valid arithmetic expressions with no parentheses. All operators should be binary (thus $a + b$ is not valid) and multiplication must be written explicitly (thus $a \cdot b$ is valid but $ab$ is not).

5. $\Sigma = \{0, 1, 2\}$,
$L = \{x \in \Sigma^* : x \text{ contains an even number of 0’s and at least one 1.}\}$