CS 241 – Week 4 Tutorial

Extending Your Assembler, Regular Languages

Winter 2020

1 Assembler: Handling Multiple Instructions

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.

Depending on your approach to handling .word directives, it might be difficult to add support for all these instructions. 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.

Here is one approach to checking the format of instructions. You are not required to use this approach, but you might find it helpful. The approach will be described using C++ syntax, but a similar idea could be done in Racket.

Define a TokenSet to be a set of token kinds, then define a Pattern to be a list of TokenSets:

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

Now you can define various Patterns representing the valid format of different instructions.

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

// Examples of Patterns
Pattern AddSubSltSltuPattern {REG, COMMA, REG, COMMA, REG};
Pattern BeqBnePattern {REG, COMMA, REG, COMMA, NUM_OR_ID};
Pattern LwSwPattern {REG, COMMA, NUM, LPAREN, REG, RPAREN};

Then you can define a pattern-matching function. It takes a vector of Tokens representing the current line, a starting position on the line, and a Pattern. Then it checks if the remaining Tokens on the line match the Pattern.

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

This is gives a flexible and concise way to check the syntax of different types of instructions.
Some notes on this approach:

- Why use a `std::list` instead of an `std::vector` for the Pattern? Deleting the first element of a vector is inefficient, while a list has an efficient `pop_front` method. Using a list makes it fast to delete elements from the front of the Pattern as you match them, which will make the `match` function easier to write.

- Your match function doesn’t have to just return a `bool` saying whether the match was successful. If you are storing instructions in an Instruction class, you could build an Instruction as you match the operands. Return a pointer to the Instruction instance if the match is successful, and a null pointer otherwise.

2 Regular Languages

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: $\{\text{ID, LABEL, WORD, COMMA, LPAREN, RPAREN, INT, HEXINT, REG}\}$. We usually don’t define alphabets with “multi-symbol symbols” like this to avoid confusion, although they do come up in practice (this alphabet 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 word or string over an alphabet $\Sigma$ is a finite sequence of symbols from $\Sigma$. The sequence with zero symbols in it (the empty word) is denoted $\varepsilon$.

- $bac$, $aba$, $c$ given that $\Sigma = \{a, b, c\}$.
- $\varepsilon$, $b$, $bb$, $bbb$ where $\Sigma = \{b\}$.
- ID REG COMMA REG COMMA REG (one word of length six) where $\Sigma = \{\text{ID, LABEL, WORD, COMMA, LPAREN, RPAREN, INT, HEXINT, REG}\}$.
  We add spaces just for readability; the spaces are not part of the alphabet or the word.
- DEADBEEF, FACE, C001BABE where $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$.

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)
- $L = RR' = \{xy : x \in R, y \in R'\}$ 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 is equivalent to replace the first three cases by the statement “$L$ is finite”.

2
3 Regular Expressions

Regular expressions are a means of expressing regular languages more concisely than set notation. The following table shows the correspondence between set notation and regular expression notation. In the table, \( \mathcal{L}(R) \) means “the language corresponding to the regular expression \( R \).”

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Corresponding Language</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td></td>
</tr>
<tr>
<td>( \varepsilon )</td>
<td>( {\varepsilon} )</td>
<td></td>
</tr>
<tr>
<td>( a )</td>
<td>( {a} )</td>
<td></td>
</tr>
<tr>
<td>( (R</td>
<td>R') )</td>
<td>( \mathcal{L}(R) \cup \mathcal{L}(R') )</td>
</tr>
<tr>
<td>( RR' )</td>
<td>( \mathcal{L}(R)\mathcal{L}(R') )</td>
<td>( R ) and ( R' ) are regular expressions</td>
</tr>
<tr>
<td>( R^* )</td>
<td>( \mathcal{L}(R)^* )</td>
<td>( R ) is a regular expression</td>
</tr>
</tbody>
</table>

Parentheses can be omitted in regular expressions. The conventional order of operations when parentheses are omitted is: Kleene star first, then concatenation, then union. So for example, the regular expression \( aa|\emptyset b^* \) is equivalent to \( ((aa)|(\emptyset(b^*)))). \) The language corresponding to this expression is \( \{aa\} \cup \{b^n : n \geq 1\}. \)

3.1 Regular Expression Problems

Describe the following languages using set notation.

1. The language of binary strings whose second symbol is a “0” and whose 5th is a “1”.
2. The language of binary strings that contain the substring “1101 01”.

Provide a regular expression for each of the following languages.

1. The two languages from the problems above.
2. \( \Sigma = \{a, b\}, \ L = \{aa, ab, ba, bb\}. \)
3. 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).
4. \( \Sigma = \{0, 1, 2\}, \ L = \{x \in \Sigma^* : x \text{ contains an even number of 0’s and at least one 1.}\} \)

4 Deterministic Finite Automata (DFAs)

A deterministic finite automaton (DFA) is a 5-tuple \( \langle \Sigma, Q, q_0, A, \delta \rangle \) 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, like $\rightarrow\bigcirc$.
• The accepting states in $A$ are marked by drawing an extra interior circle, like $\bigcirc$.
• 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 $a$, like $\bigcirc^a\bigcirc$.

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:

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

### 4.1 DFA Problems

Draw DFA diagrams for the following languages:

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

### Appendix: DFAs and Programs

While DFAs might seem like an entirely new concept, you can think of them as just a slightly different form of programming. DFAs are equivalent to programs where you only have a fixed constant amount of memory.

Consider the following problem:

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.

It is very easy to write a program that takes a string over $\{a, b\}$ and determines whether it has this form. Here is pseudocode for such a program.

```plaintext
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 actually translate this program directly into a DFA! The correspondence is as follows:

- The states correspond to the possible different values of the variables defined at the top of the program. You can think of DFA states as representing “states of memory” in a program!

In this case, there are four possible states of memory, since `parity_a` and `parity_b` can each only take on two possible values. So the state set of the DFA is \{(0,0), (0,1), (1,0), (1,1)\}. Each pair represents the values of `parity_a` and `parity_b` at different points in the program.

This trick only works because the program uses a constant amount of memory for each input. If the amount of memory used was unbounded, we would end up with infinitely many states and we would not be able to make a DFA. So for example if you tried to use this trick with a program that has something like a “counter” or “sum” variable, you would have to prove that the number cannot get arbitrarily large.

- The initial state corresponds to the initial values of the variables, so it is (0, 0).

- The set of accepting states corresponds to the condition under which we return true. In this case there is only one state of memory in which we return true: when `parity_a` is 0 and `parity_b` is 1. So the set of accepting states is \{(0,1)\}.

- The transition function corresponds to the body of the loop. For example, if the state of memory is `parity_a == 1` and `parity_b == 0`, and the next symbol is a `b`, after the loop executes we will have `parity_a == 1` and `parity_b == 1`. This means there is a transition from `(1,0)` to `(1,1)` on `b`. The other transitions can be determined in a similar way.

If you draw this DFA you will get the following diagram.

You are not required to understand this correspondence between DFAs and programs for the midterm and final exam. However, you might find it useful to think about DFAs in this way.