Recap: Regular Expressions

Regular expression over alphabet $\Sigma$:

1. $\epsilon$ is a RE denoting the set $\{\epsilon\}$
2. if $a \in \Sigma$, then $a$ is a RE denoting $\{a\}$
3. if $r$ and $s$ are REs, denoting $L(r)$ and $L(s)$, then:
   - $r \mid s$ is a RE denoting $L(r) \cup L(s)$
   - $r \cdot s$ is a RE denoting $L(r).L(s)$
   - $r^*$ is a RE denoting $L(r)^*$
Exercise

Which regular expression is equivalent to \((0|1) \ast 1(0|1)\ast\)

- \((01|11) \ast (0|1)\ast\)
- \((0|1) \ast (10|11|1)(0|1)\ast\)
- \((0|1) \ast (0|1)(0|1)\ast\)
Exercise

Which regular expression is equivalent to \((0|1)^* 1(0|1)^*\)

- \((01|11)^* (0|1)^*\)
  - no (it allows 0)
- \((0|1)^* (10|11|1)(0|1)^*\)
- \((0|1)^* (0|1)(0|1)^*\)
Exercise

Which regular expression is equivalent to \((0\mid1)\ast 1(0\mid1)\ast\)

- \((0\mid1)\ast (0\mid1)\ast\)  no  (it allows 0)
- \((0\mid1)\ast (10\mid11\mid1)(0\mid1)\ast\)  yes
- \((0\mid1)\ast (0\mid1)(0\mid1)\ast\)
Which regular expression is equivalent to \((0|1)^* 1(0|1)^*\)

- \((01|11)^* (0|1)^*\)  
  - no (it allows 0)

- \((0|1)^* (10|11|1)(0|1)^*\)  
  - yes

- \((0|1)^* (0|1)(0|1)^*\)  
  - no (it allows 0)
Lexical Analysis

Input:

```plaintext
if ( x == 0 ) x = x + 1 ;
```

Output:

```
IF, LPAREN, ID(x),_EQUALS, INTLIT(0), RPAREN, ID(x), EQSIGN, ID(x), PLUS, INTLIT(1), SEMICOLON
```

Two approaches to construct lexical analyzers:

1. Manual construction: use first character to decide on token class (This lecture)
2. Automatic construction: conversion of regular expressions to automata
   - Tools like JFlex are lexer generators for Java
In practice, a lexer reads characters and generate tokens on demand.

It works with streams instead of sequences, with procedures like:

- `current`: returns current element in stream.
- `next`: advance the current element.

Lexer operates on a character input stream and returns a token output stream.
class CharStream {
String fileName;
FileReader reader = new FileReader(fileName);
BufferedReader file = new BufferedReader(reader);
char current = ' ';
Boolean eof = false;
void next() throws Exception {
if (eof)
throw EndOfInput("reading");
int c = file.read();
eof = (c == -1);
current = (char) c;
}

// representation of a token
public class Token {
public static final int EOF = 0;
public static final int ID = 1; // x
public static final int INT = 2;
public static final int LPAREN = 3;
public static final int RPAREN = 4;
public static final int SCOLON = 5;
public static final int WHILE = 6;
public static final int AssignEQ = 7;
public static final int CompareEQ = 8;
public static final int MUL = 9;
public static final int DIV = 10;
public static final int PLUS = 11;
public static final int LEQ = 12;
public static final int IF = 13;
// ...
}

class Lexer {
CharStream ch;
Token current;
void next() {
/* lexer code goes here */
}
}
Recognizing Identifiers and Keywords

char c = ch.current;
if (Character.isLetter(c)) {
    StringBuffer b = new StringBuffer();
    while (Character.isLetter(c) || Character.isDigit(c)) {
        b.append(c);
        ch.next(); c = ch.current;
    }
}
if(!keywords.containsKey(b.toString)){
    token.kind = ID;
    token.id = b;
}
else token.kind = KW;

- regular expression for identifiers: letter (letter|digit)*

- Keywords look like identifiers but are reserved as keywords in language definition

- keywords: A constant Map from strings to keyword tokens

- if identifier is not in map, then it is ordinary identifier
char c = ch.current;
if (Character.isDigit(c)) {
    int k = 0;
    while (Character.isDigit(c)) {
        k = 10 * k +
        Character.getNumericValue(c);
        ch.next(); c = ch.current;
    }
    token.kind = INT;
    token.value = k;
}
Deciding which Token is Coming

• How do we know the class of the token we are supposed to analyze (string, integer, identifier, ...)?

• Manual construction: use lookahead (next symbol in stream) to decide on token class

• compute \( \text{FIRST}(e) \) - symbols with which \( e \) can start

• check in which \( \text{FIRST}(e) \) current token is

• If \( L \subseteq \Sigma^* \) is a language, then \( \text{FIRST}(L) \) is set of all alphabet symbols that start some word in \( L \)

\[
\text{FIRST}(L) = \{ a \in \Sigma \mid \exists v \in \Sigma^* . (a.v) \in L \}
\]
FIRST of Some Languages

- \( \text{FIRST}(\{ab, bb, a\}) = \{a, b\} \)
- \( \text{FIRST}(\{a, ab\}) = \{a\} \)
- \( \text{FIRST}(\{bbbbbbbbbb\}) = \{b\} \)
- \( \text{FIRST}(\{a\}) = \{a\} \)
- \( \text{FIRST}(\{\} \} = \) }
- \( \text{FIRST}(\{\epsilon\}) = \) }
- \( \text{FIRST}(\{\epsilon, ba\}) = \{b\} \)
Given regular expression $e$, how to compute $\text{FIRST}(e)$?

- Use automata (will discuss later)
- Rules that directly compute them
  (also work for grammars, we will see them for parsing)
Given regular expression $e$, how to compute FIRST($e$)?

- Use automata (will discuss later)
- Rules that directly compute them
  (also work for grammars, we will see them for parsing)

Examples of FIRST($e$) computation:
- FIRST($ab*$) = $\{a\}$
- FIRST($ab*|c$) = $\{a, c\}$
- FIRST($a*b*c$) = $\{a, b, c\}$
- FIRST(($cb|a*c*)d*e$) =
Given regular expression $e$, how to compute FIRST($e$)?

- Use automata (will discuss later)
- Rules that directly compute them
  (also work for grammars, we will see them for parsing)

Examples of FIRST($e$) computation:

- FIRST($ab*$) = \{a\}
- FIRST($ab * |c$) = \{a, c\}
- FIRST($a * b * c$) = \{a, b, c\}
- FIRST($((cb |a * c*)d * e)$) = \{a, c, d, e\}
FIRST of Regular Expression

FIRST: RegExp → Σ ,  FIRST(e) ⊆ Σ

Define recursively:

- FIRST(∅) = ∅
- FIRST(ε) = ∅
- FIRST(a) = {a}
- FIRST(e₁ ∣ e₂) = FIRST(e₁) ∪ FIRST(e₂)
- FIRST(e*) = FIRST(e)
- FIRST(e₁ . e₂) = FIRST(e₁) ∪ FIRST(e₂) , if nullable(e₁)
  FIRST(e₁) , otherwise

We need the notion of nullable(e):
whether ε belongs to the regular language
Can regular expr contain empty word? nullable(\(L\)) means \(\epsilon \in L\)
nullable: RegExp → {true, false}
Define recursively:

- nullable(\(\emptyset\)) = false
- nullable(\(\epsilon\)) = true
- nullable(\(a\)) = false
- nullable(\(e_1 \mid e_2\)) = nullable(\(e_1\)) \lor nullable(\(e_2\))
- nullable(\(e^*\)) = true
- nullable(\(e_1.e_2\)) = nullable(\(e_1\)) \land nullable(\(e_2\))
### From RE to Programs

- **Converting Well-Behaved Regular Expression into Programs**

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>if (current=a) next else error</td>
</tr>
<tr>
<td>$r_1.r_2$</td>
<td>(code for $r_1$) ; (code for $r_2$)</td>
</tr>
</tbody>
</table>
| $(r_1 \mid r_2)$  | if (current in FIRST($r_1$))  
                      | code for $r_1$  
                      | else  
                      | code for $r_2$ |
| when $\text{FIRST}(r_1) \cap \text{FIRST}(r_2) = \emptyset$ | |
| $r^*$             | while(current in FIRST($r$))  
                      | code for $r$ |
switch (ch.current) {
    case '(' : { current = OPAREN; ch.next(); return; } 
    case ')' : { current = CPAREN; ch.next(); return; } 
    case '+' : { current = PLUS; ch.next(); return; } 
    case '/' : { current = DIV; ch.next(); return; } 
    case '*' : { current = MUL; ch.next(); return; } 
    case '=' : { // more tricky because there can be =, ==
        ch.next();
        if (ch.current == '=')
            { ch.next(); current = CompareEQ; return; }
        else { current = AssignEQ; return; }
    }
    case '<' : { // more tricky because there can be <, <=
        ch.next();
        if (ch.current == '=')
            { ch.next(); current = LEQ; return; }
        else { current = LESS; return; }
    }
}
Subtleties in General Case

- Sometimes $\text{FIRST}(e_1)$ and $\text{FIRST}(e_2)$ overlap for two different token classes
  - e.g. AssignEQ "=" and CompareEQ "=="
- Must remember where we were and go back, or work on recognizing multiple tokens at the same time
- Example: comment begins with division sign, so we should not decide on division token when checking for comment
if (ch.current == '/*') {
  ch.next();
  if (ch.current == '*/') {
    while (!isEOL && !isEOF) {
      ch.next();
    }
  } else {
    token.kind = DIV;
  }
} else {

Question: how can we handle nested comments?

/* foo /* bar */ baz */
if (ch.current == '//' ) {
    ch.next();
    if (ch.current == '//' ) {
        while (!isEOL && !isEOF) {
            ch.next();
        }
    } else {
        token.kind = DIV;
    }
} else {
    token.kind = DIV;
}

**Question:** how can we handle nested comments?

/* foo */ bar */ baz */

**Answer:** use a counter for nesting depth
• Whitespace can be defined as a token using space character, tabs, and various end-of-line characters
• In most languages (Java, ML, C) white spaces and comments can occur between any two other tokens
  • They have no meaning, so parser does not want to see them
• Convention: lexical analyzer removes those “tokens” from its output
• Lexical analyzer always finds the next non-whitespace non-comment token
• What kind of applications care about the comments and white spaces in source code?