Coursework 2

Thursday, October 24, 2019 8:17 PM

Q1:
To implement a lexer for the WHILE language, you first need to design the appropriate regular expressions for the following eight syntactic entities:

1. keywords are
   while, if, then, else, do, for, to, true, false, read, write, skip
2. operators are
   +, -, *, %, /, ==, !=, >, <, :=, &&, ||
3. strings are enclosed by "…"
4. parentheses are (, {,) and }
5. there are semicolons ;
6. whitespaces are either " " (one or more) or \n or \t
7. identifiers are letters followed by underscores __, letters or digits
8. numbers are 0, 1, …and so on; give a regular expression that can recognise 0, but not numbers with leading zeroes, such as 001

You can use the basic regular expressions
0, 1, c, r1 + r2, r1 · r2, $r*$
but also the following extended regular expressions
[c1, c2, . . . , cn] a set of characters
r^+ one or more times r
$r^?$ optional r
$r^{\{n\}}$ n-times r
Later on you will also need the record regular expression:
REC(x : r) record regular expression
Try to design your regular expressions to be as small as possible. For example you should use character sets for identifiers and numbers. Feel free to use the general character constructor CFUN introduced in CW 1.

A1:

```
ZERO_DIGITS = RANGE(('0' to '9').toSet)
ONE_DIGITS = RANGE(('1' to '9').toSet)
SYM = RANGE(('A' to 'Z').toSet ++ ('a' to 'z').toSet ++ ('0' to '9').toSet ++ Set('_'))
NOT_QUOTES = CFUN(λx.x != ")
NOT_NEWLINES = CFUN(λx.x != \n)

Keyword = "while" + "if" + "then" + "else" + "do" + "for" + "to" + "true" + "false" + "read" + "write" +
"skip"
Op = "+" + "-" + "*" + "%" + "/" + "==" + "!=" + ">" + "<" + ":=" + "&&" + "||"
Str = " · NOT_QUOTES* · "
Para = { + } + ( + )
Semi = ;
Whitespace = (" " + "\n" + "\t")+
Id = SYM · (SYM + ZERO_DIGITS)*
Num = "0" + ONE_DIGITS · ZERO_DIGITS*
Comm = "//" · NOT_NEWLINES* · "\n" (comments for the factoring program)
```

Q2:
Implement the Sulzmann & Lu lexer from the lectures. For this you need to implement the functions nullable and der (you can use your code from CW 1), as well as mkeps and inj. These functions need to be appropriately extended for the extended regular expressions from Q1. Write down the clauses for

mkeps([c1, c2, . . . , cn]) $\stackrel{\text{def}}{=}$?
mkeps(r^+) $\stackrel{\text{def}}{=}$?
mkeps($r^?$) $\stackrel{\text{def}}{=}$?
mkeps($r^{\{n\}}$) $\stackrel{\text{def}}{=}$?

inj ([c1, c2, . . . , cn]) c . . . $\stackrel{\text{def}}{=}$?
inj (r^+) c . . . $\stackrel{\text{def}}{=}$?
inj ($r^?$) c . . . $\stackrel{\text{def}}{=}$?
inj ($r^{\{n\}}$) c . . . $\stackrel{\text{def}}{=}$?

where inj takes three arguments: a regular expression, a character and a value.
Test your lexer code with at least the two small examples below:

regex:	string:
$a^{\{3\}}$	aaa
$(a + 1)^{\{3\}}$	aa

Both strings should be successfully lexed by the respective regular expression, that means the lexer returns in both examples a value.
Also add the record regular expression from the lectures to your lexer and implement a function, say env, that returns all assignments from a value (such that you can extract easily the tokens from a value).
Finally give the tokens for your regular expressions from Q1 and the string
"read n;"
and use your env function to give the token sequence.

A2:

mkeps([c1, c2, . . . , cn]) $\stackrel{\text{def}}{=}$ not defined (will never be called)
mkeps(r^+) $\stackrel{\text{def}}{=}$ (Stars([mkeps(r)]))
mkeps($r^{?)}$ $\stackrel{\text{def}}{=}$ if (nullable(r)) Left(mkeps(r)) else Right(Empty)
mkeps($r^{\{n\}}$) $\stackrel{\text{def}}{=}$ Stars(List.fill(n)(mkeps(r)))

inj ([c1, c2, . . . , cn]) c Empty $\stackrel{\text{def}}{=}$ Chr(r)
inj (r^+) c Sequ(v, Stars(vs)) $\stackrel{\text{def}}{=}$ Stars(inj(r, c, v1)::vs)
inj ($r^?$) c Left(v) $\stackrel{\text{def}}{=}$ Left(inj(r, c, v))
inj ($r^?$) c Right(v) $\stackrel{\text{def}}{=}$ Right(inj(r, c, v))
inj ($r^{\{n\}}$) c Sequ(v, Stars(vs)) $\stackrel{\text{def}}{=}$ Stars(inj(r, c, v)::vs)

Value for $a^{\{3\}}$ with aaa: Stars(List(Chr(a), Chr(a), Chr(a)))
Value for $(a + 1)^{\{3\}}$ with aa: Stars(List(Left(Chr(a)), Left(Chr(a)), Right(Empty)))

Tokens for "read n;":
(k,read),
(w," "),
(id,n),
(s,;)

Q3:
Extend your lexer from Q2 to also simplify regular expressions after each derivation step and rectify the computed values after each injection. Use this lexer to tokenize the programs in Figure 1, 2 and 3. Give the tokens of these programs where whitespaces are filtered out. Make sure you can tokenise exactly these programs. (Programs ommited)

Tokens for fib:

```
(k,"write"),
(str,"\"Fib\""),
(s,";"),
(k,"read"),
(id,"n"),
(s,";"),
(id,"minus1"),
(op,":="),
(n,"0"),
(s,";"),
(id,"minus2"),
(op,":="),
(n,"1"),
(s,";"),
(k,"while"),
(id,"n"),
(op,">"),
(n,"0"),
(k,"do"),
(p,"{"),
(id,"temp"),
(op,":="),
(id,"minus2"),
(s,";"),
(id,"minus2"),
(op,":="),
(id,"minus1"),
(op,"+"),
(id,"minus2"),
(s,";"),
(id,"minus1"),
(op,":="),
(id,"temp"),
(s,";"),
(id,"n"),
(op,":="),
(id,"n"),
(op,"-"),
(n,"1"),
(p,"}"),
(s,";"),
(k,"write"),
(str,"\"Result\""),
(s,";"),
(k,"write"),
(id,"minus2")
```

Tokens for triple while:

```
(id,"start"),
(op,":="),
(n,"1000"),
(s,";"),
(id,"x"),
(op,":="),
```

(id,"start"),
(s,";"),
(id,"y"),
(op,":="),
(id,"start"),
(s,";"),
(id,"z"),
(op,":="),
(id,"start"),
(s,";"),
(k,"while"),
(n,"0"),
(op,"<"),
(id,"x"),
(k,"do"),
(p,"{"),
(k,"while"),
(n,"0"),
(op,"<"),
(id,"y"),
(k,"do"),
(p,"{"),
(k,"while"),
(n,"0"),
(op,"<"),
(id,"z"),
(k,"do"),
(p,"{"),
(id,"z"),
(op,":="),
(id,"z"),
(op,"-"),
(n,"1"),
(p,"}"),
(s,";"),
(id,"z"),
(op,":="),
(id,"start"),
(s,";"),
(id,"y"),
(op,":="),
(id,"y"),
(op,"-"),
(n,"1"),
(p,"}"),
(s,";"),
(id,"y"),
(op,":="),
(id,"start"),
(s,";"),
(id,"x"),
(op,":="),
(id,"x"),
(op,"-"),
(n,"1"),
(p,"}")

Tokens for factors:

(k,"write"),
(str,"\"Input n please\""),
(s,";"),
(k,"read"),
(id,"n"),
(s,";"),
(k,"write"),
(str,"\"The factors of n are\""),
(s,";"),
(id,"f"),
(op,":="),
(n,"2"),
(s,";"),
(k,"while"),
(id,"n"),
(op,"!="),
(n,"1"),
(k,"do"),
(p,"{"),
(k,"while"),
(p,"("),
(id,"n"),
(op,"/"),
(id,"f"),
(p,")"),
(op,"*"),
(id,"f"),
(op,"=="),
(id,"n"),
(k,"do"),
(p,"{"),
(k,"write"),
(id,"f"),
(s,";"),
(id,"n"),
(op,":="),
(id,"n"),
(op,"/"),
(id,"f"),
(p,"}"),
(s,";"),
(id,"f"),
(op,":="),
(id,"f"),
(op,"+"),
(n,"1"),
(p,"}")