

6CCS3CFL

Coursework

Rui Han Ji Chen
K20027110

Problem 1

```

1 // arithmetic expressions
2 lazy val AExp: Parser[List[Token], AExp] =
3   (Te ~ p"+" ~ AExp).map[AExp]{ case x ~ _ ~ z => Aop("+", x, z) } ||
4   (Te ~ p"-" ~ AExp).map[AExp]{ case x ~ _ ~ z => Aop("-", x, z) } || Te
5 lazy val Te: Parser[List[Token], AExp] =
6   (Fa ~ p"*" ~ Te).map[AExp]{ case x ~ _ ~ z => Aop("*", x, z) } ||
7   (Fa ~ p"/" ~ Te).map[AExp]{ case x ~ _ ~ z => Aop("/", x, z) } ||
8   (Fa ~ p"%" ~ Te).map[AExp]{ case x ~ _ ~ z => Aop("%", x, z) } || Fa
9 lazy val Fa: Parser[List[Token], AExp] =
10  (p "(" ~ AExp ~ p ")").map{ case _ ~ y ~ _ => y } ||
11  IdParser.map(Var) ||
12  NumParser.map(Num)
13
14
15 // boolean expressions with some simple nesting
16 lazy val BExp: Parser[List[Token], BExp] =
17  (AExp ~ p"==" ~ AExp).map[BExp]{ case x ~ _ ~ z => Bop("==", x, z) } ||
18  (AExp ~ p"!=" ~ AExp).map[BExp]{ case x ~ _ ~ z => Bop("!=", x, z) } ||
19  (AExp ~ p">=" ~ AExp).map[BExp]{ case x ~ _ ~ z => Bop(">=", x, z) } ||
20  (AExp ~ p"<=" ~ AExp).map[BExp]{ case x ~ _ ~ z => Bop("<=", x, z) } ||
21  (AExp ~ p"<" ~ AExp).map[BExp]{ case x ~ _ ~ z => Bop("<", x, z) } ||
22  (AExp ~ p">" ~ AExp).map[BExp]{ case x ~ _ ~ z => Bop(">", x, z) } ||
23  (p "(" ~ BExp ~ p ")" ~ p"&&" ~ BExp).map[BExp]{ case _ ~ y ~ _ ~ _ ~ v =>
    And(y, v) } ||
24  (p "(" ~ BExp ~ p ")" ~ p"||" ~ BExp).map[BExp]{ case _ ~ y ~ _ ~ _ ~ v => Or
    (y, v) } ||
25  (p"true".map[BExp]{ _ => True }) ||
26  (p>false".map[BExp]{ _ => False }) ||
27  (p "(" ~ BExp ~ p ")").map[BExp]{ case _ ~ x ~ _ => x }

```

Listing 1: Problem 1

Problem 2

```
1
2 case class TokenParser(s: String) extends Parser[List[Token], String] {
3   def parse(sb: List[Token]) = sb match {
4     case T_KWD(x)::rest => if (x == s) Set((s, rest)) else Set()
5     case T_OP(x)::rest  => if (x == s) Set((s, rest)) else Set()
6     case T_SYM(x)::rest => if (x == s) Set((s, rest)) else Set()
7     case T_SEMI::rest  => Set((";", rest))
8     case T_LPAREN::rest => Set("(" , rest))
9     case T_RPAREN::rest => Set(")", rest))
10    case _ => Set()
11  }
12 }
13
14 case object StrParser extends Parser[List[Token], String] {
15   def parse(tk: List[Token]) = tk match {
16     case T_STR(x)::rest => Set((x, rest))
17     case _ => Set()
18   }
19 }
20
21 case object IdParser extends Parser[List[Token], String] {
22   def parse(tk: List[Token]) = tk match {
23     case T_ID(x)::rest => Set((x, rest))
24     case _ => Set()
25   }
26 }
27
28
29 case object NumParser extends Parser[List[Token], Int] {
30   def parse(tk: List[Token]) = tk match {
31     case T_NUM(x)::rest => Set((x, rest))
32     case _ => Set()
33   }
34 }
```

Listing 2: Problem 2

Problem 3

```

1 def eval_aexp(a: AExp, env: Env) : Int = a match {
2   case Num(i) => i
3   case Var(s) => env(s)
4   case Aop("+", a1, a2) => eval_aexp(a1, env) + eval_aexp(a2, env)
5   case Aop("-", a1, a2) => eval_aexp(a1, env) - eval_aexp(a2, env)
6   case Aop("*", a1, a2) => eval_aexp(a1, env) * eval_aexp(a2, env)
7   case Aop("/", a1, a2) => eval_aexp(a1, env) / eval_aexp(a2, env)
8   case Aop("%", a1, a2) => eval_aexp(a1, env) % eval_aexp(a2, env)
9 }
10
11 def eval_bexp(b: BExp, env: Env) : Boolean = b match {
12   case True => true
13   case False => false
14   case Bop("==", a1, a2) => eval_aexp(a1, env) == eval_aexp(a2, env)
15   case Bop("!=", a1, a2) => (eval_aexp(a1, env) != eval_aexp(a2, env))
16   case Bop("<=", a1, a2) => eval_aexp(a1, env) <= eval_aexp(a2, env)
17   case Bop(">=", a1, a2) => eval_aexp(a1, env) >= eval_aexp(a2, env)
18   case Bop(">", a1, a2) => eval_aexp(a1, env) > eval_aexp(a2, env)
19   case Bop("<", a1, a2) => eval_aexp(a1, env) < eval_aexp(a2, env)
20   case And(b1, b2) => eval_bexp(b1, env) && eval_bexp(b2, env)
21   case Or(b1, b2) => eval_bexp(b1, env) || eval_bexp(b2, env)
22 }
23
24 def eval_stmt(s: Stmt, env: Env) : Env = s match {
25   case Skip => env
26   case Assign(x, a) => env + (x -> eval_aexp(a, env))
27   case If(b, b1, b2) => if (eval_bexp(b, env)) eval_b1(b1, env) else
    eval_b1(b2, env)
28   case While(b, b1) =>
    if (eval_bexp(b, env)) eval_stmt(While(b, b1), eval_b1(b1, env))
    else env
29   case WriteStr(x) => { print(x) ; env }
30   case WriteVar(x) => { print(env(x)) ; env }
31   case Read(x) => env + (x -> readInt())
32 }
33
34 }

```

Listing 3: Problem 1