6CCS3CFL

Coursework

Rui Han Ji Chen
K20027110

Problem 1

```
// arithmetic expressions
lazy val AExp: Parser[List[Token], AExp] =
  (Te ~ p"+" ~ AExp).map[AExp]{ case x ~ _ ~ z => Aop("+", x, z) } ||
  (Te ~ p"-" ~ AExp).map[AExp]{ case x ~ _ ~ z => Aop("-", x, z) } || Te
lazy val Te: Parser[List[Token], AExp] =
  (Fa ~ p"*" ~ Te).map[AExp]{ case x ~ _ ~ z => Aop("*", x, z) } ||
  (Fa ~ p"/" ~ Te).map[AExp]{ case x ~ _ ~ z => Aop("/", x, z) } ||
  (Fa ~ p"%" ~ Te).map[AExp]{ case x ~ _ ~ z => Aop("%", x, z) } || Fa
lazy val Fa: Parser[List[Token], AExp] =
   (p"(" ~ AExp ~ p")").map{ case _ ~ y ~ _ => y } ||
   IdParser.map(Var) ||
   NumParser.map(Num)


// boolean expressions with some simple nesting
lazy val BExp: Parser[List[Token], BExp] =
   (AExp ~ p"==" ~ AExp).map[BExp]{ case x ~ _ ~ z => Bop("==", x, z) } ||
   (AExp ~ p"!=" ~ AExp).map[BExp]{ case x ~ _ ~ z => Bop("!=", x, z) } ||
   (AExp ~ p">=" ~ AExp).map[BExp]{ case x ~ _ ~ z => Bop(">=", x, z) } ||
   (AExp ~ p"<=" ~ AExp).map[BExp]{ case x ~ _ ~ z => Bop("<=", x, z) } ||
   (AExp ~ p"<" ~ AExp).map[BExp]{ case x ~ _ ~ z => Bop("<", x, z) } ||
   (AExp ~ p">" ~ AExp).map[BExp]{ case x ~ _ ~ z => Bop(">", x, z) } ||
   (p"(" ~ BExp ~ p")" ~ p"&&" ~ BExp).map[BExp]{ case _ ~ y ~ _ ~ _ ~ v => And(y, v) } ||
   (p"(" ~ BExp ~ p")" ~ p"||" ~ BExp).map[BExp]{ case _ ~ y ~ _ ~ _ ~ v => Or(y, v) } ||
   (p"true".map[BExp]{ _ => True }) ||
   (p"false".map[BExp]{ _ => False }) ||
   (p"(" ~ BExp ~ p")").map[BExp]{ case _ ~ x ~ _ => x }
```

Listing 1: Problem 1

Problem 2

```scala

case class TokenParser(s: String) extends Parser[List[Token], String] {
  def parse(sb: List[Token]) = sb match {
    case T_KWD(x)::rest => if (x == s) Set((s, rest)) else Set()
    case T_OP(x)::rest => if (x == s) Set((s, rest)) else Set()
    case T_SYM(x)::rest => if (x == s) Set((s, rest)) else Set()
    case T_SEMI::rest => Set((";", rest))
    case T_LPAREN::rest => Set(("(", rest))
    case T_RPAREN::rest => Set((")", rest))
    case _ => Set()
  }
}

case object StrParser extends Parser[List[Token], String] {
  def parse(tk: List[Token]) = tk match {
    case T_STR(x)::rest => Set((x, rest))
    case _ => Set()
  }
}

case object IdParser extends Parser[List[Token], String] {
  def parse(tk: List[Token]) = tk match {
    case T_ID(x)::rest => Set((x, rest))
    case _ => Set()
  }
}


case object NumParser extends Parser[List[Token], Int] {
  def parse(tk: List[Token]) = tk match {
    case T_NUM(x)::rest =>  Set((x, rest))
    case _ => Set()
  }
}
```

Listing 2: Problem 2

Problem 3

```
def eval_aexp(a: AExp, env: Env) : Int = a match {
  case Num(i) => i
  case Var(s) => env(s)
  case Aop("+", a1, a2) => eval_aexp(a1, env) + eval_aexp(a2, env)
  case Aop("-", a1, a2) => eval_aexp(a1, env) - eval_aexp(a2, env)
  case Aop("*", a1, a2) => eval_aexp(a1, env) * eval_aexp(a2, env)
  case Aop("/", a1, a2) => eval_aexp(a1, env) / eval_aexp(a2, env)
  case Aop("%", a1, a2) => eval_aexp(a1, env) % eval_aexp(a2, env)
}

def eval_bexp(b: BExp, env: Env) : Boolean = b match {
  case True => true
  case False => false
  case Bop("==", a1, a2) => eval_aexp(a1, env) == eval_aexp(a2, env)
  case Bop("!=", a1, a2) => (eval_aexp(a1, env) != eval_aexp(a2, env))
  case Bop("<=", a1, a2) => eval_aexp(a1, env) <= eval_aexp(a2, env)
  case Bop(">=", a1, a2) => eval_aexp(a1, env) >= eval_aexp(a2, env)
  case Bop(">", a1, a2) => eval_aexp(a1, env) > eval_aexp(a2, env)
  case Bop("<", a1, a2) => eval_aexp(a1, env) < eval_aexp(a2, env)
  case And(b1, b2) => eval_bexp(b1, env) && eval_bexp(b2, env)
  case Or(b1, b2) => eval_bexp(b1, env) || eval_bexp(b2, env)
}

def eval_stmt(s: Stmt, env: Env) : Env = s match {
  case Skip => env
  case Assign(x, a) => env + (x -> eval_aexp(a, env))
  case If(b, bl1, bl2) => if (eval_bexp(b, env)) eval_bl(bl1, env) else
   eval_bl(bl2, env)
  case While(b, bl) =>
    if (eval_bexp(b, env)) eval_stmt(While(b, bl), eval_bl(bl, env))
    else env
  case WriteStr(x) => { print(x) ; env }
  case WriteVar(x) => { print(env(x)) ; env }
  case Read(x) => env + (x -> readInt())
}
```

Listing 3: Problem 1