COMP332 Programming Languages 2019 Assignment 2

Assignment 2 - Syntax Analysis for the weBCPL language.

MACQUARIE
University

Prepared By: Ashrey Ranjit

Student number: 44241119

Unit: COMP332 Programming Languages

COMP332 Assignment 2

COMP332 Programming Languages 2019 Assignment 2

Report

NAME: Ashrey Ranjit

Student number: 44241119

Description of the aim of the Assignment

This is the assignment 2 of comp332. We must develop a lexical parser and tree builder for the retro-language weBCPL. We have to design the syntax analyser which is a compiler designed to translate the BCPL to WebAssembly. Likewise, we are also required to write test to test our parsers for parsed expressions. In the report I will be explaining the parser built and explaining the test cases made to test the Scala syntax analyser including tree builder for the weBCPL language.

The motivation behind this assignment is to understand what kinds od tasks often arises in programming situations other than implementations of a programming language. As simple languages are used to write a lot of configuration files for many applications, the structure and files reliably must be able to be understood by the applications, similar to as how a reading program files and to understand them is to be done by a compiler.

In the report I am going to divide the report into two sections, in the beginning section of the report I will be explaining in detail of the descriptions of the syntax analyser, and then on the later of the report I will be explain each of the tests to check if the parser are functional or not.

Description of the Design and Implementation

```
// FIXME Replace this stubbed parser.
lazy val unlabelledStmt: PackratParser[Statement] =
  repeatableStmt | iteratedStmt | testStmt

// FIXME Add your parsers for weBCPL statements here.
```

```
lazy val iteratedStmt: PackratParser[Statement] = {
  UNTIL ~ expression ~ DO ~ statement ^^ { case a ~ b ~ c ~ d => UntilDoStmt(b, d) } |
    WHILE ~ expression ~ DO ~ statement ^^ { case a ~ b ~ c ~ d => WhileDoStmt(b, d) } |
    FOR ~ idndef ~ equal ~ expression ~ TO ~ expression ~
    opt(BY ~> expression) ~ DO ~ statement ^^ { case a ~ b ~ c ~ d ~ e ~ f ~ g ~ h ~ i => ForStmt(b, d, f, g, i) }
}

lazy val testStmt: PackratParser[Statement] = {
  TEST ~ expression ~ THEN ~ statement ~ ELSE ~ statement ^^ { case a ~ b ~ c ~ d ~ e ~ f => TestThenElseStmt(b, d, f) } |
    IF ~ expression ~ DO ~ statement ^^ { case a ~ b ~ c ~ d => IfDoStmt(b, d) } |
    UNLESS ~ expression ~ DO ~ statement ^^ { case a ~ b ~ c ~d => UnlessDoStmt(b, d) }
}

lazy val repeatableStmt: PackratParser[Statement] =
  repeatableStmt ~ REPEAT ^^ { case a ~ b => RepeatStmt(a) } |
    repeatableStmt ~ REPEATWHILE ~ expression ^^ { case a ~ b ~ c => RepeatWhileStmt(a, c) } |
    repeatableStmt ~ REPEATUNTIL ~ expression ^^ { case a ~ b ~ c => RepeatUntilStmt(a, c) } |
    simpleStmt


lazy val simpleStmt: PackratParser[Statement] =
  rep1sep(expression, comma) ~ assign ~ rep1sep(expression, comma) ^^ { case a ~ b ~ c => AssignStmt(a, c) } |
    callExp ^^ { case a => CallStmt(a) } | BREAK ^^^ BreakStmt() | LOOP ^^^ LoopStmt() | ENDCASE ^^^ EndCaseStmt() |
    RETURN ^^^ ReturnStmt() | FINISH ^^^ FinishStmt() |GOTO ~ labuse ^^ { case a ~ b => GotoStmt(b) } |
    RESULTIS ~ expression ^^ { case a ~ b => ResultIsStmt(b) |
    SWITCHON ~ expression ~ INTO ~ blockStmt ^^ { case a ~ b ~ c ~ d => SwitchOnStmt(b, d) } |
    blockStmt

lazy val blockStmt: PackratParser[Block] =
  leftBrace ~> repsep(declaration, semiColon) ~ rep1sep(statement, semiColon) <~ rightBrace ^^ Block
```

The block statement parser is a block that can contain declarations and statements.

The simple statement parser parse call expressions, labels, switch on cases and block statements. The repeatable statement parser parses repeating statements. The test statement parser parse conditional statements. The iterated statement parser parse iterations of statements through conditional expressions.

The unlabelled statement parser parses a combination of the above statement parsers.

```
// FIXME Add your expression parsers for levels 1-6 of the precedence hierarchy here.
lazy val condExp: PackratParser[Expression] =
  level_2 ~ rightArrow ~ level_2 ~ comma ~ condExp ^^ { case a ~ b ~ c ~ d ~ e => IfExp(a, c, e) } |
    level_2

lazy val level_2: PackratParser[Expression] =
  level_2 ~ EQV ~ level_3 ^^ { case a ~ b ~ c => EqvExp(a, c) } |
    level_2 ~ XOR ~ level_3 ^^ { case a ~ b ~ c => XorExp(a, c) } |
    level_3

lazy val level_3: PackratParser[Expression] =
  level_3 ~ pipe ~ level_4 ^^ { case a ~ b ~ c => OrExp(a, c) } |
    level_4

lazy val level_4: PackratParser[Expression] =
  level_4 ~ apersand ~ level_5 ^^ { case a ~ b ~ c => AndExp(a, c) |
    level_5

lazy val level_5: PackratParser[Expression] =
  NOT ~ level_5 ^^ { case a ~ b => NotExp(a, b) } |
    level_6

lazy val level_6: PackratParser[Expression] =
  level_6 ~ shiftLeft ~ relExp ^^ { case a ~ b ~ c => ShiftLeftExp(a, c) } |
    level_6 ~ shiftRight ~ relExp ^^ { case a ~ b ~ c => ShiftRightExp(a, c) } |
    relExp
```

The following expression parsers follow the precedence and associativity rules of the grammar of the language. Starting from the elementary expression parser, which contains the primitive data types of the language, there are 13 levels of precedence. Most associativity is on the left, except for the if-then-else expression, which is on the right. Each level of expression parser defaults to the previous expression parser.

```
lazy val addExp: PackratParser[Expression] =
  addExp ~ minus ~ level_9 ^^ { case a ~ b ~ c => MinusExp(a, c) } |
    addExp ~ plus ~ level_9 ^^ { case a ~ b ~ c => PlusExp(a, c) } |
    level_9

// FIXME Add your expression parsers for levels 8-12 of the precedence hierarchy here.
lazy val level_9: PackratParser[Expression] =
  unaryMinus ~ level_9 ^^ { case a ~ b => NegExp(b) } |
    unaryPlus ~ level_9 ^^ { case a ~ b => b } |
    ABS ~ level_9 ^^ { case a ~ b => AbsExp(b) } |
    level_10

lazy val level_10: PackratParser[Expression] =
  level_10 ~ star ~ level_11 ^^ { case a ~ b ~ c => StarExp(a,c) } |
    level_10 ~ slash ~ level_11 ^^ { case a ~ b ~ c => SlashExp(a, c) } |
    level_10 ~ MOD ~ level_11 ^^ { case a ~ b ~ c => ModExp(a, c) } |
    level_11

lazy val level_11: PackratParser[Expression] =
  unaryPling ~ level_11 ^^ { case a ~ b => UnaryPlingExp(b) } |
    unaryPercent ~ level_11 ^^ { case a ~ b => UnaryBytePlingExp(b) } |
    at ~ level_11 ^^ { case a ~ b => AddrOfExp(b) } |
    level_12

lazy val level_12: PackratParser[Expression] =
    level_12 ~ pling ~ primaryExp ^^ { case a ~ b ~ c => BinaryPlingExp(a, c) } |
    level_12 ~ percent ~ primaryExp ^^ { case a ~ b ~ c => BinaryBytePlingExp(a, c) } |
    primaryExp
```

Description of the Testing

The main motivations behind writing the tests was to check if the parsers I wrote was functional or not. I have written tests to check each methods.

Test 1.

This is the test to test the addition expression. The addition parser test checks to see if the syntax analyser can parse a tree for addition expressions and checks to see if the it performs addition for two identifiers.

I did not make a test for the subtractions as it is like the addition expression.

```
// FIXME Add your automated tests here.
test("parsing addition") {
  phrase(expression)("x + y") should parseTo[Expression](
    PlusExp(IdnExp(IdnUse("x")), IdnExp(IdnUse("y"))))
}
```

Test 2.

This is the test to test the if-then-else expression. The if -else parser test checks to see if the syntax analyser can parse a tree for if-then-else expression, checks the validity of the expressions. As a Boolean as if true first if false the next one.

```
test("parsing if-then-else") {
  phrase(expression)("x > y -> z = x , z = y") should parseTo[Expression](
    IfExp(
      GreaterExp(IdnExp(IdnUse("x")), IdnExp(IdnUse("y"))),
      EqualExp(IdnExp(IdnUse("z")), IdnExp(IdnUse("x"))),
      EqualExp(IdnExp(IdnUse("z")), IdnExp(IdnUse("y")))))
}
```

Test 3.

This is the test to test the equivalency expression (i.e. EQV). The equivalence parser test checks to see if the syntax analyser can parse a tree for equivalent expressions, as it it provides an eqvalency expression of two expressions when using EQV.

```
test("parsing EQV") {
  phrase(expression)("a EQV b") should parseTo[Expression](
    EqvExp(IdnExp(IdnUse("a")), IdnExp(IdnUse("b"))))
}
```

Test 4.

This is the test to test the O expression. The OR parser test checks to see if the syntax analyser can parse a tree for two identifiers if the OR expressions by using “|” operator.

```
test("parsing Or") {
  phrase(expression)("a | b") should parseTo[Expression](
    OrExp(IdnExp(IdnUse("a")), IdnExp(IdnUse("b"))))
}
```

Test 5.

This is the test to test the AN expression. The AND parser test checks to see if the syntax analyser can parse a tree for AND expressions of two identifiers A and B when using the “&” operator.

```
test("parsing And") {
  phrase(expression)("a & b") should parseTo[Expression](
    AndExp(IdnExp(IdnUse("a")), IdnExp(IdnUse("b"))))
}
```

Test 6.

This is the test to test the NOT expression. The NOT parser test checks to see if the syntax analyser can parse a tree for NOT expressions of two identifiers A and B when using the “NOT” expression.

```
test("parsing Not") {
  phrase(expression)("NOT a") should parseTo[Expression](
    NotExp(IdnExp(IdnUse("a"))))
}
```

Test 7.

This test will check to see if the parser is parsing the provided identifier to shift right when using the “>>” operator between the two identifiers.

```
test("parsing ShiftRight") {
  phrase(expression)("a >> b") should parseTo[Expression](
    ShiftRightExp(IdnExp(IdnUse("a")), IdnExp(IdnUse("b"))))
}
```

Test 8.

This is the test to test the Negative expression. The Negative parser test checks to see if the syntax analyser can parse a tree for Negative expressions of two identifiers A and B when using the “-” operator.

```
test("parsing Negative") {
  phrase(expression)("- a") should parseTo[Expression](
    NegExp(IdnExp(IdnUse("a"))))
}
```

Test 9.

This is the test to test the AB expression. The ABS parser test checks to see if the syntax analyser can parse a tree for AB expressions of two identifiers A and B when using the “ABS” keyword.

```
test("parsing ABS") {
  phrase(expression)("ABS a") should parseTo[Expression](
    AbsExp(IdnExp(IdnUse("a"))))
}
```

Test 10.

The multiplication test checks to see if the multiplication expression is provided when "*" operator is used between two operators.

```
test("parsing multiplication") {
  phrase(expression)("a * b") should parseTo[Expression](
    StarExp(IdnExp(IdnUse("a")), IdnExp(IdnUse("b"))))
}
```

Test 11.

This test is used to see if the "/" expression when use will parse the expression between the identifiers as a division or not.

```
test("parsing division") {
  phrase(expression)("a / b") should parseTo[Expression](
    SlashExp(IdnExp(IdnUse("a")), IdnExp(IdnUse("b"))))
}
```

Test 12.

The MOD test is used to check if using the "MOD" keyword between the two identifiers will perform the MOD operation on it or not.

```
test("parsing MOD") {
  phrase(expression)("a MOD b") should parseTo[Expression]
    ModExp(IdnExp(IdnUse("a")), IdnExp(IdnUse("b"))))
}
```

Test 13.

The test check to see if the parser is able to provide an unary pling expression when using the "!" operator on the two expression.

```
test("parsing unary pling") {
  phrase(expression)("! a") should parseTo[Expression](
    UnaryPlingExp(IdnExp(IdnUse("a"))))
}
```

Test 14.

The test is used to check if the parser when used the “%” will perform the pling expression when used with an expression

```
test("parsing unary byte pling") {
  phrase(expression)("% a") should parseTo[Expression](
    UnaryBytePlingExp(IdnExp(IdnUse("a"))))
}
```

Test 15.

The test is used to check if the parser when used the “@” will perform the pling expression when used with an expression

```
test("parsing at") {
  phrase(expression)("@ a") should parseTo[Expression](
    AddrOfExp(IdnExp(IdnUse("a"))))
}
```

Test 16.

The test is used to check if the parser when used the “%” will perform the pling expression when used with two expression.

```
test("parsing binary pling") {
  phrase(expression)("a % b") should parseTo[Expression](
    BinaryBytePlingExp(IdnExp(IdnUse("a")), IdnExp(IdnUse(
}
```

Test 17.

The test is used to check if the parser when used the “!” will perform the pling expression when used with two expression.

```
test("parsing binary pling") {
  phrase(expression)("a ! b") should parseTo[Expression](
    BinaryPlingExp(IdnExp(IdnUse("a")), IdnExp(IdnUse("b")
}
```

18.

```
  test("parsing block") {
    phrase(statement)("{ LET a = 5 b := a }") should parseTo
      Block(
        Vector(
          LetDecl(
            Vector(
              LetVarClause(
                Vector(IdnDef("a")),
                Vector(IntExp(5)))))),
        Vector(
          AssignStmt(
            Vector(IdnExp(IdnUse("b"))),
            Vector(IdnExp(IdnUse("a")))))))
  }
```

19.

```
  test("parsing assign") {
    phrase(statement)("a := 2") should parseTo[Statement](
      AssignStmt(
        Vector(IdnExp(IdnUse("a"))),
        Vector(IntExp(2))))
  }
```

20.

```
test("parsing call") {
  phrase(statement)("f(u, v)") should parseTo[Statement](
    CallStmt(
      CallExp(
        IdnExp(IdnUse("f")),
        Vector(IdnExp(IdnUse("u")), IdnExp(IdnUse("v")))))
}
```

21. 22.

```
test("parsing break") {
  phrase(statement)("BREAK") should parseTo[Statement](
    BreakStmt())
}

test("parsing switchon") {
  phrase(statement)("SWITCHON x INTO { a := 1 }") should p
    SwitchOnStmt(
      IdnExp(IdnUse("x")),
      Block(
        Vector(),
        Vector(
          AssignStmt(
            Vector(IdnExp(IdnUse("a"))),
            Vector(IntExp(1)))))))
}
```

23.

```
  test("parsing repeat while") {
    phrase(statement)("a := 1 REPEATWHILE b") should parseTo
      RepeatWhileStmt(
        AssignStmt(
              Vector(IdnExp(IdnUse("a"))),
              Vector(IntExp(1))),
        IdnExp(IdnUse("b"))))
  }
```

24.

```
test("parsing repeat until") {
  phrase(statement)("a := 1 REPEATUNTIL b") should parseT
    RepeatUntilStmt(
      AssignStmt(
            Vector(IdnExp(IdnUse("a"))),
            Vector(IntExp(2))),
      IdnExp(IdnUse("b"))))
}
```

25.

```
test("parsing test") {
  phrase(statement)("TEST a THEN a := 1 ELSE b := 2") sh
    TestThenElseStmt(
      IdnExp(IdnUse("a")),
      AssignStmt(
            Vector(IdnExp(IdnUse("a"))),
            Vector(IntExp(1))),
      AssignStmt(
            Vector(IdnExp(IdnUse("b"))),
            Vector(IntExp(2)))))
}
```

26.

```
test("parsing ifdo") {
  phrase(statement)("IF a DO a := 2") should parseTo[Stat
    IfDoStmt(
      IdnExp(IdnUse("a")),
      AssignStmt(
            Vector(IdnExp(IdnUse("a"))),
            Vector(IntExp(2)))))
}
```

27.

```
test("parsing unlessdo") {
  phrase(statement)("UNLESS a DO b := 6") should parseTo[S
    UnlessDoStmt(
      IdnExp(IdnUse("a")),
      AssignStmt(
            Vector(IdnExp(IdnUse("b"))),
            Vector(IntExp(6)))))
}
```

28.

```
test("parsing untildo") {
  phrase(statement)("UNTIL a DO b := 2") should parseTo[St
    UntilDoStmt(
      IdnExp(IdnUse("a")),
      AssignStmt(
            Vector(IdnExp(IdnUse("b"))),
            Vector(IntExp(2)))))
}
```

29.

```
test("parsing whiledo") {
  phrase(statement)("WHILE a DO a := 2") should parseTo[St
    WhileDoStmt(
      IdnExp(IdnUse("a")),
      AssignStmt(
            Vector(IdnExp(IdnUse("a"))),
            Vector(IntExp(2)))))
}
```

30.

```
test("parsing for") {
  phrase(statement)("FOR g = a TO b DO c := 3") should parseTo[Statement](
    ForStmt(
      IdnDef("g"),
      IdnExp(IdnUse("a")),
      IdnExp(IdnUse("b")),
      None,
      AssignStmt(
          Vector(IdnExp(IdnUse("c"))),
          Vector(IntExp(3)))))
}
```