COMP3000
Programming Languages

Assignment 2
Lexical Analysis

Chaz Lambrechtsen

45426317

Table of Contents

1. Summary

This report will walk through the design and implementation in order to develop a lexical analyser, parser and tree builder for a simple functional programming language called FunLang. This report will also include a description of the testing that has been carried out with details and explanations of different tests that have been utilised.

2. Design and Implementation

While designing and implementing this Lexical Analyser I have utilised useful resources provided to us such as the Assignment 2 Spec [2] and Syntax analysis from Week 7 [3]. External resources such as Kiama documentation [1] have also been used in this implementation.

2.1 Basic Expressions

First implementing the syntax of program and expressions

Provided in Assignment Spec [2]

```
program : exp.

exp : app
    | block
    | cond
    | exp "==" exp
    | exp "<" exp
```

```
    | exp "+" exp
    | exp "-" exp
    | exp "*" exp
    | exp "/" exp
    | "false"
    | "true"
    | idnuse
    | integer
    | "(" exp ")".
```

Implemented in Scala as the following:

```
lazy val factor : PackratParser[Exp] =
    block |
    app |
    cond |
    exp ~ ("==" ~> exp) ^^ {case exp1 ~ exp2 => EqualExp(exp1,exp2)} |
    exp ~ ("<" ~> exp) ^^ {case exp1 ~ exp2 => LessExp(exp1,exp2)} |
    exp ~ ("+" ~> exp) ^^ {case exp1 ~ exp2 => PlusExp(exp1,exp2)} |
    exp ~ ("-" ~> exp) ^^ {case exp1 ~ exp2 => MinusExp(exp1,exp2)} |
    exp ~ ("*" ~> exp) ^^ {case exp1 ~ exp2 => StarExp(exp1,exp2)} |
    exp ~ ("/" ~> exp) ^^ {case exp1 ~ exp2 => SlashExp(exp1,exp2)} |
    "false" ^^ (_ => BoolExp (false)) |
    "true" ^^ (_ => BoolExp (true)) |
    idnuse |
    integer ^^ (s => IntExp (s.toInt)) |
    "(" ~> exp <~ ")" |
    failure ("exp expected")
```

Each symbol is assigned to the corresponding expression block in the tree, each symbol must contain 2 expressions for each side of the symbol, e.g. (1 + 2) or ((2*3) + (4-5))

2.2 Blocks and Functions

Block will simply create a BlockExp with (Vector(definitions), exp) only when the parsed input contains a string such as “{ insert-definition insert-exp }”

```
lazy val block : PackratParser[Exp] =
    //block : "{" definitions exp "}".
    "{" ~> exp <~ "}" ^^ {
        case exp => BlockExp(Vector(), exp)
    }
```

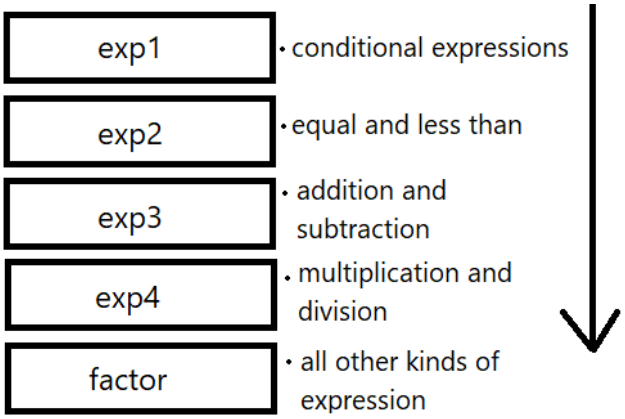
Fundefn will create the following sequence: Fun(IdnDef(id), ar, exp) only when the parsed input contains a string such as “def func1 (i : Int) = a + 1”

```
lazy val fundefn : PackratParser[Fun] =
    //"def" idndef "(" arg ")" "=" exp.
    ("def" ~> identifier) ~ ("(" ~> arg <~ ")") ~ ("=" ~> exp) ^^ {
        case id ~ ar ~ exp => Fun(IdnDef(id), ar, exp)
    }
```

2.3 Precedence and Associativity

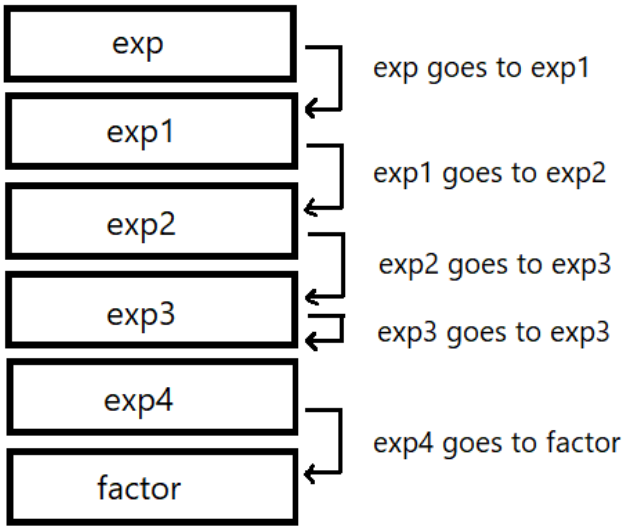
Precedence levels have been implemented by creating functional flow of precedence levels.

The following diagram shows each function in precedence order.



Associativity has been implemented by calling the respective Associative path at the end of each level of precedence.

The following diagram shows the order of associativity in my program.



2.4 Types (Tipe)

Parsing the strings Int and Bool directly will result in the corresponding type e.g. IntType() and BoolType() respectively.

Parsing a string such as “(tipe => tipe)” containing a type followed by => followed by a type will result in a call to FunType with the given parameters.

A tipe can also be parenthesised for example (Int) or (Bool)

```
lazy val tipe : PackratParser[Type] =
    //"some type" ^^^ Type()
    "Int" ^^^ IntType() |
    "Bool" ^^^ BoolType() |
    // tipe will recurse back to itself
    tipe ~ ("=>" ~> tipe) ^^ { case tipe1 ~ tipe2 => FunType(tipe1,tipe2) } |
    "(" ~> tipe <~ ")"
```

3. Testing

These tests confirm that my program is working correctly because they include edge cases, syntax error as well as testing for correct syntax and lexical expressions which produces the correct FunLang Tree corresponding to the assignment specification.

3.1 Block Tests

Test 1 - "{ val a = 1 a + 1 + 2 }"

This test ensures the block expression is working correctly and takes a definition and expression, as defined in the specifications

Test 2 - "{ 1 + 2 }"

This test ensures that the block cannot just pass an expression and fails with the correct error, because it should take definition first then expression therefore this is not valid

Test 3 - "{ }"

Test 3 will intentionally fail and give an error because the block is empty, because this should not be a valid in FunLang

Test 4 - "{ "

Test 4 will intentionally fail and give an error because the block is missing the right closing bracket.

Test 5 - " }"

Test 5 will intentionally fail and give an error because the block is missing the left opening bracket.

Test 6 - "{ val a = 3 a + 1 } * { val a = 3 a + 1 }"

This test will ensure that block can be nested and return the correct tree output which is StarExp(BlockExp(…), BlockExp(…))

Function Tests

Test 1 - "{ val x = 100 def func (a : Int) = a + 1 func (x) } "

Parsing a block starting with a value then a function definition will create the corresponding valid tree

Test 2 - “{ def inc (a : Int) = a * 1 a + 1 }”

Parsing a block starting with a function definition should also be a valid input and create the correct corresponding tree.

3.2 Associativity Tests

Test 1 - "1 + 2 + 3 + 4"

Parsing consecutive plus expressions should result in a left associative output, or from left side to right side. E.g. ((1 + 2) + 3) + 4

Test 2 - "1 * 2 * 3 * 4"

Parsing consecutive star expressions should result in a left associative output, or from left side to right side. E.g. ((1 * 2) * 3) * 4

Test 3 - "if (1 == 2) then 2 + 3 else 1 + 2 + 3"

This will test == expression as it is not associative and consecutive plus expressions within an if statement which should result in the same order as the input because plus is left associative.

Test 4 - "if (1 == 2 + 3 - 4 + 5) then 1 - 2 + 3 else 1 + 2 "

This test includes == expression with consecutive plus expressions directly after. This should result in the same order because == is not associative.

3.3 Precedence Tests

Test 1 – "if (1 == 1) then 1 + 1 else 1 + 1 + 1"

Since in my program, cond is the first precedence order, then this will return a valid if expression tree. The values do not matter in this test otherwise this will test associativity as well, hence only using values 1.

Test 2 - "1 * 2 + 3"

Precedence order defined in the assignment spec [2] and in my program should always give a tree with plus or minus first. This test should result in PlusExp before StarExp e.g. PlusExp (StarExp(…),…)

Test 3 - "1 - 2 / 3"

Precedence order defined in the assignment spec [2] and in my program should always give a tree with plus or minus first. This test should result in MinusExp(…,SlashExp(…)

3.4 Types (tipe) tests

Test 1 - "{def func (a : Bool) = False 1 + 1}"

A tipe should only be able to be parsed within a function definition so this should produce the correct tree.

Test 2 - "a : Bool = False 1 + 1"

This test will ensure that a Type cannot be create outside a function definition

3.5 Type Associativity Tests

Test 1 - "{def func (a : Bool => Int) = False 1 + 1}"

As shown in the specifications a tipe should be able to recurse on itself this should also be right associative in order to produce the correct tree.

Test 2 - "{def func (a : Bool => Int => Bool => Int) = False 1 + 1}"

Consecutive tipe using => should be right associative and produce the correct tree.

4. References

[1] "inkytonik/kiama", GitHub, 2020. [Online]. Available: https://github.com/inkytonik/kiama/blob/master/wiki/ParserCombs.md. [Accessed: 29- Sep- 2020].

[2] "iLearn Assignment 2 Spec", Ilearn.mq.edu.au, 2020. [Online]. Available: https://ilearn.mq.edu.au/pluginfile.php/6676539/mod_resource/content/6/COMP3000ass2.html. [Accessed: 29- Sep- 2020].

[3] "Week 7 SyntaxAnalysis.scala", Ilearn.mq.edu.au, 2020. [Online]. Available: https://ilearn.mq.edu.au/pluginfile.php/6430526/mod_resource/content/1/SyntaxAnalysis.scala. [Accessed: 29- Sep- 2020].