COMP332 ASSIGNMENT 2

SYNTAX ANALYSIS FOR THE weBCPL LANGUAGE

HANNES VENTER

44908903

Table of Contents

1. Introduction

This report will describe the process undertaken to create a Scala program which develops a lexical parser and tree builder for the retro-language weBCPL [1].

This compiler takes the language BCPL and translates it into WebAssembly which allows a user to run a few simple BCPL programs in their browser.

Basic Combined Programming Language (BCPL) is described as a “simple systems programming language with a small fast compiler which is easily ported to new machines” and was created “to provide a simple language for writing an operating environment and compiler for CPL” [2].

WebAssembly was created to enable high performance applications on the Web. It is a “safe, portable, low-level code format designed for efficient execution and compact representation” [2].

The parser uses the complete context-free grammar for the weBCPL language which was supplied with the assignment. It builds upon this grammar and removes any ambiguities which may exist.

Stage 2 of this report will describe the steps taken to implement the Statement parsers, and the methods used to remove the ambiguity in the Expression grammar and implement the Expression parsers.

Stage 3 of this report will describe the various types of tests used to ensure that the program parses the language correctly.

2. Design and Implementation

2.1 Statement Parsers

The context-free grammar for the weBCPL language was primarily used to implement the Statement parsers. The module `LexicalAnalysis.scala` was used to retrieve the lexical components for parsing integers / strings / character constants, BCPL keywords and operators, identifiers, labels and comments. In conjunction, the module `BCPLTree.scala` provided the full definition and description of the tree structures that were to be used.

The `statement` non-terminal called for the `unlabelledStmt` non-terminal which in turn called `repeatableStmt | iteratedStmt | testStmt`. Each grammar production was then implemented in the parser according to the keywords in `LexicalAnalysis.scala`.

These parsers were all relatively similar in their implementation, with each one requiring slightly different keywords or operators from `LexicalAnalysis.scala`. The module `BCPLTree.scala` was also closely examined to ensure that each of the operators ^^ and ^^^ were correctly used and returned the specific values required by each clause.

Some notable grammar productions were:

1. `iteratedStmt`, which required an optional (`?`) to be given to `ForStmt` as a parameter.

```
"FOR" idndef "=" expression "TO" expression "BY" expression)? "DO" statement
```

It was found that the operator "opt" delivered the required Optional and as such, the `?` was replaced by `opt`.

```
(FOR ~> idndef) ~ (equal ~> expression) ~ (TO ~> expression) ~ opt(BY ~> expression
) ~ (DO ~> statement) ^^ ForStmt
```

2. `simpleStmt` which required a `Block` to be given to `SwitchOnStmt` as a parameter.

   After various compiler errors, Dom's forum post suggested to analyse the return types of the parsers. This resulted in a change of blockStmt to return a Block, instead of a Statement (`blockStmt: PackratParser[Block]`), and fixed the compilation error.

2.2 Expression Parsers

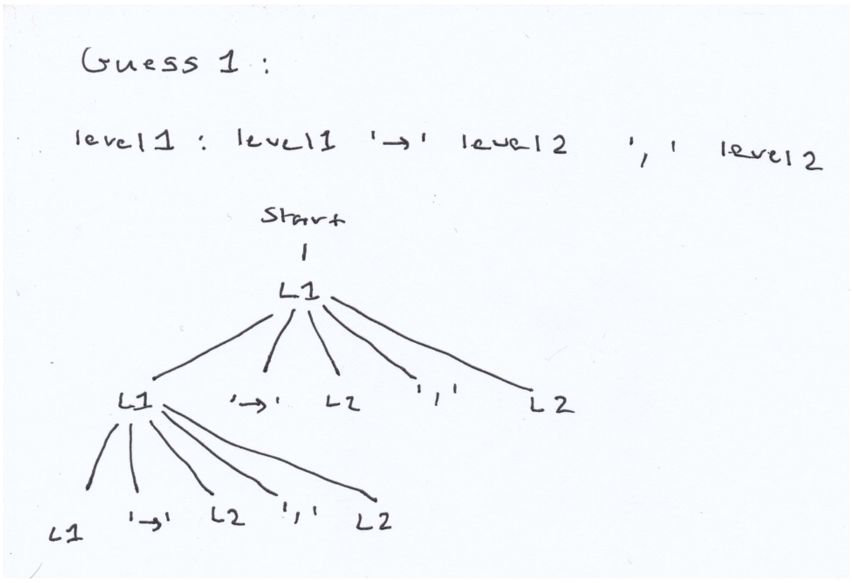
It was noted early on that “the expression non-terminal is ambiguous since it makes no allowance for precedence and associativity of the operators”. As such, the grammar production was rewritten “to implement the precedence and associativity rules described the [provided] table” [2].

2.2.1 Level 1

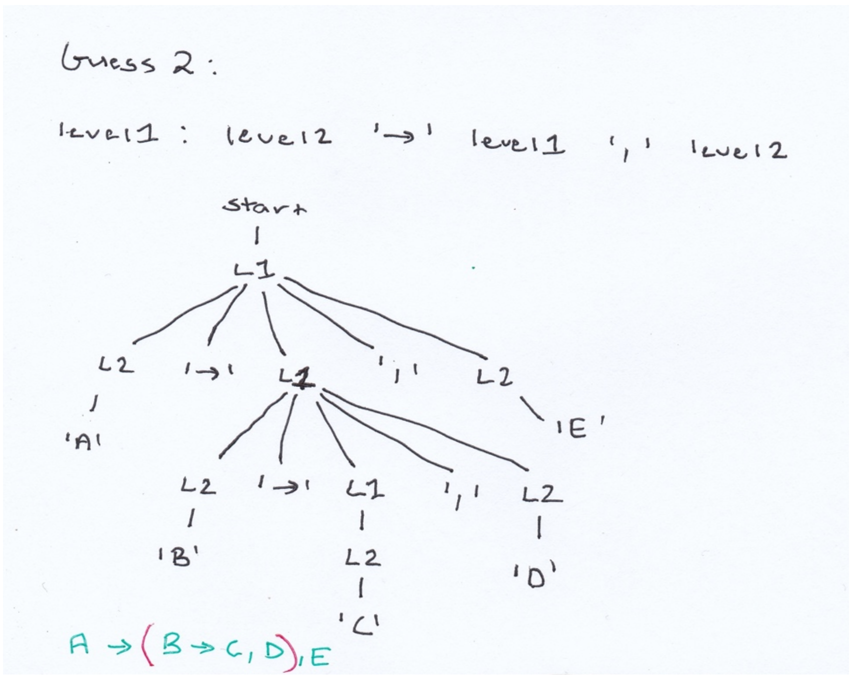
The highest Precedence expression was the ternary/if expression and as such, its parser was implemented first. It proved to be a relatively difficult parser implement, however Dom’s forum post provided a clear method with which to implement its associativity rule.

The trees below were drawn to figure out how to find the correct associativity of the grammar production.

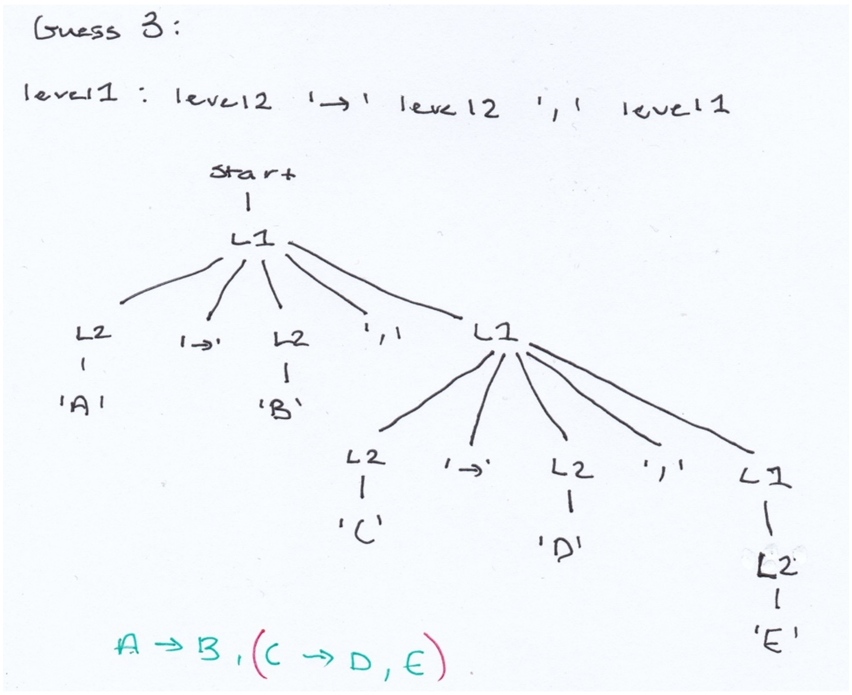
```
start: level1
level1: //guess
     | level2
level2: ‘A’ | … | ‘Z’
```



As seen above, Guess 1 proved to implement left associativity and as such was not suitable for this grammar.

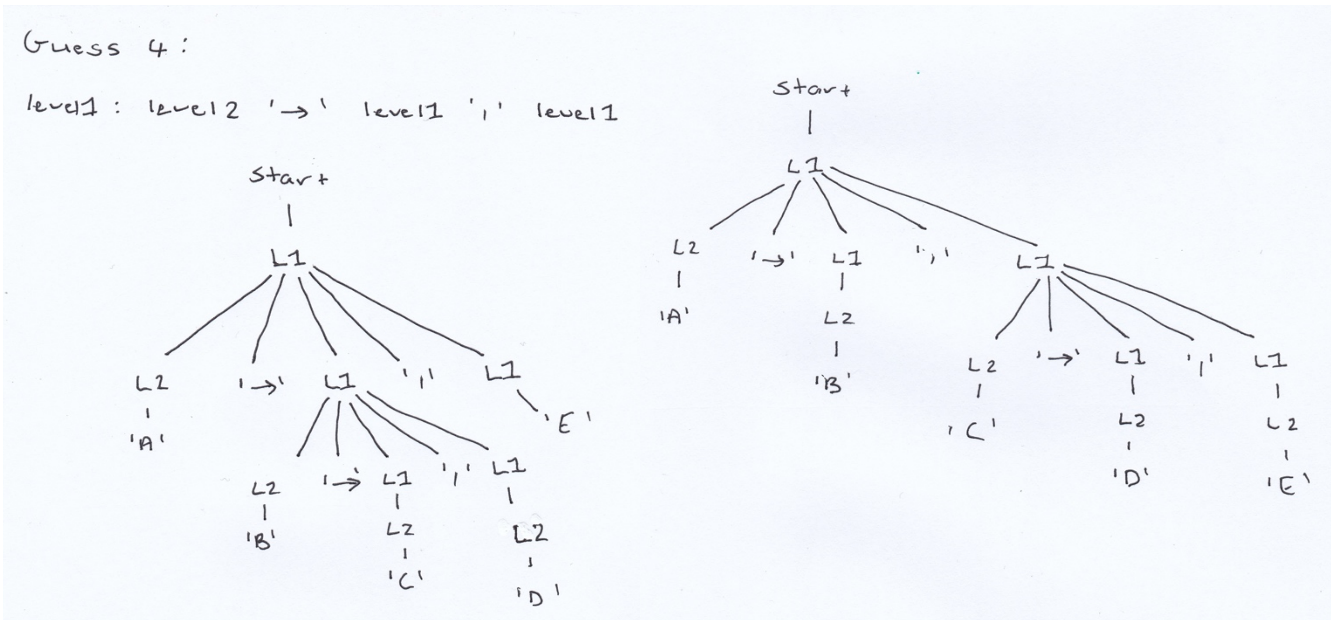


As seen above, Guess 2 proved to implement the grammar for the string `A->B->C,D,E`.



As seen above, Guess 3 proved to implement the grammar for the string `A->B,C->D,E`.

After studying the trees, it was found that using `level1: level2 '->' level1 ',' level1` in the grammar would result in the correct associativity for both strings.



As seen above, this grammar proved to correctly implement the right associativity of the ternary if expression. It was thus placed in the code.

2.2.2 Levels 2 – 8

Levels 2 – 8 (not 7) were relatively similar. All the expressions except the unary not expression were left associative which meant that their grammar followed the pattern; `level-n: level-n 'op' level-(n+1).`[3]

2.2.3 Level 9

Level 9 proved slightly more challenging since unary + called for an identifier use instead of another expression. The keyword `idnuse` was thus put in place, and the rest of the function was implemented without any associativity.

2.2.3 Levels 10-12

Levels 10 – 12 were relatively similar as well. Extra care had to be taken when implementing the `unaryPling` and `unaryPercent`.

3. Testing

3.1 Statements

Testing of the Statements were primary done using tests which Dom supplied and using tests which combined Expressions and Statements.

3.2 Expressions

3.2.1 Ternary If Expression

Test were created to ensure that both cases of the ternary if expressions were correctly implemented in the parsers.

The first tested whether:

```
  expression("A -> B, C -> D, E") should parseTo[Expression](...
```

And the resulting tree should take the form: `A -> B, (C -> D, E)`

The second tested whether:

```
  expression("A -> B -> C, D, E") should parseTo[Expression](...
```

And the resulting tree should take the form: `A -> (B -> C, D), E)`

3.2.2 XOR and EQV

The following tests were conducted to ensure that the XOR and EQV expressions were correct. They are simple tests which pass a small line in and check the corresponding trees.

3.2.3 OR and AND

The next tests were conducted to ensure that the OR and AND expressions were correct. Both also implemented an additional test which tested whether they were both correctly left associative.

3.2.4 BitShifting

A similar approach was taken when testing both the left and right bit shifting. They were both first tested with a simple 2 expression statement, and then further tested for the left associativity with a greater number of expressions.

3.2.5 Address

Testing of the address expression was done using the string `v!i + v!j - a * b`, which Dom supplied in the Assignment specification. This was tested against the tree which he also supplied. [1]

3.3 Further Testing

A significant number of further tests were used to ensure the parser works correctly. These tests were taken and modified from the sample `SyntaxAnalysisTests.scala` which Dom posted on the forums. [3]

The majority of the tests only required minor alterations and then worked successfully in this program. Other tests which were obviously irrelevant were not included.

A notable change includes:

- Unary – actually being of lower precedence the `*`, which contradicts what the tests say. This was updated to correctly test for this program in which Unary – has a lower precedence than `*`.

```
test("unary - has higher precedence than * (to left)") {
   expression("-a * b") should parseTo[Expression](
     StarExp(NegExp(IdnExp(IdnUse("a"))), IdnExp(IdnUse("b"))))
 }
```

4. Bibliography

[1] D. Verity, “assignment2.md,” 17 September 2019. [Online]. Available: https://bitbucket.org/dominicverity/comp332-webcpl/src/master/assignment2.md. [Accessed 6 October 2019].

[2] D. Verity, “README.md,” 17 September 2019. [Online]. Available: https://bitbucket.org/dominicverity/comp332-webcpl/src/master/README.md. [Accessed 6 October 2019].

[3] D. Verity, “*** Assignment 2 Q&A ***,” 1 October 2019. [Online]. Available: https://ilearn.mq.edu.au/mod/forum/discuss.php?d=1290667. [Accessed 2 October 2019].