Macquarie University, Department of Computing

COMP332 Programming Languages 2019

Assignment 2

Due: 11.55pm Saturday 12th October 2019 (end week 9)
Worth: 15% of unit assessment

Marks breakdown:

- Code: 50% (of which tests are worth 10%)
- Report: 50% (of which test description is worth 10%)

Submit a notice of disruption via Ask@MQ if you are unable to submit on time for medical or other legitimate reasons.

Late penalty without proper justification: 20% of the full marks for the assessment per day or part thereof late.

Overview

This assignment asks you to develop a lexical parser and tree builder for the retro-language weBCPL, a variant of the language BCPL. We will build on these components in later assignments to complete a full implementation of this language in assignment 3.

Building this implementation will give you insight into the way that programming language implementations work in general, as well as specific experience with how programs are written, how they are compiled, and how they are executed.

This kind of task often arises in programming situations other than programming language implementation. For example, many applications have configuration files that are written in simple languages. The application must be able to read these files reliably and understand their structure, just as a compiler must read program files and understand them.

The weBCPL programming language and its syntax are described in detail in the README file which you can find, in markdown and HTML formats, in the root directory of the assignment 2 skeleton.

Important!

You are **strongly** advised not to try to solve the whole assignment in one go. It is best to write code to handle the parsing and tree construction for some simple constructs first and then build up to the full language.

What you have to do

You have to write, document and test a Scala syntax analyser including tree builder for the weBCPL language.

A skeleton **sbt** project for the assignment has been provided on BitBucket as the dominicverity/comp332-webcpl repository. The modules are very similar to those used in the practical exercises. The skeleton contains the modules you will need.

The `SyntaxAnalysis.scala` module in the skeleton already contains code to parse weBCPL declarations and some simple expressions. Your task is to complete this implementation by adding code to parse weBCPL statements and more complex expressions.

Your code must use the Kiama parsing library as discussed in lectures and practicals. You should use the expression language syntax analyser and tree builder from the mixed classes as a guide for your implementation.

The parser skeleton provides a module `LexicalAnalysis.scala` which handles all of the lexical issues involved in parsing weBCPL programs. Parsers provided by that module provide everything necessary to parse integer / string / character constants, BCPL keywords and operators, identifiers, labels and comments. As described in the README it also handles all of the tricky corner cases in the parsing of weBCPL, including the *"semi-colons may be omitted at the end of a line"* rule.

As well as parsing the input it is presented with, your program should construct a suitable source program tree to represent the parsed result. See `BCPLTree.scala` in the skeleton for the full definition and description of the tree structures that you must use. For this part of the assignment, you **must not** modify these tree classes, just create instances in your parser code.

You should not make changes to any of the framework modules other than `SyntaxAnaysis.scala`, which is where you will put your parser code, and `SyntaxAnalysisTests.scala`, which is where you will put the automated test code for your parsers. Look for the `FIXME` comments to find the places where you should add your code.

Source Trees

As an example of the desired tree structure, here is a fragment of BCPL code

```
v!i + v!j - a * b
```

and here is the tree that your syntax analyser should produce on parsing that expression:

```
MinusExp(
    PlusExp(
        BinaryPlingExp(
```

```
            IdnExp(IdnUse("v")),
            IdnExp(IdnUse("i"))),
        BinaryPlingExp(
            IdnExp(IdnUse("v")),
            IdnExp(IdnUse("j")))),
    StarExp(
        IdnExp(IdnUse("a")),
        IdnExp(IdnUse("b"))))
```

Notice that the higher precedence of the dereferencing operator (`!` which is pronounced "pling" in BCPL jargon) over multiplication over addition / subtraction, as well as the left associativity of the latter, has been taken into account in this tree. As a more complex example, consider the following fragment of weBCPL code

```
$(
    LET poss = all & ~(ld | row | rd)
    UNTIL poss = 0 DO
    $(
        LET p = poss & -poss
        poss := poss - p
        try(ld + p << 1, row + p, rd + p >> 1)
    $)
$)
```

from which your parser should produce the following more complicated syntax tree:

```
Block(
    Vector(
        LetDecl(
            Vector(
                LetVarClause(
                    Vector(IdnDef("poss")),
                    Vector(
                        AndExp(
                            IdnExp(IdnUse("all")),
                            NotExp(
                                OrExp(
                                    OrExp(
                                        IdnExp(IdnUse("ld")),
                                        IdnExp(IdnUse("row"))),
                                    IdnExp(IdnUse("rd")))))))))),
    Vector(
        UntilDoStmt(
            EqualExp(IdnExp(IdnUse("poss")), IntExp(0)),
            Block(
                Vector(
```

```
            LetDecl(
                Vector(
                    LetVarClause(
                        Vector(IdnDef("p")),
                        Vector(
                            AndExp(
                                IdnExp(IdnUse("poss")),
                                NegExp(IdnExp(IdnUse("poss")))))))),
        Vector(
            AssignStmt(
                Vector(IdnExp(IdnUse("poss"))),
                Vector(
                    MinusExp(
                        IdnExp(IdnUse("poss")),
                        IdnExp(IdnUse("p"))))),
            CallStmt(
                CallExp(
                    IdnExp(IdnUse("try")),
                    Vector(
                        ShiftLeftExp(
                            PlusExp(
                                IdnExp(IdnUse("ld")),
                                IdnExp(IdnUse("p"))),
                            IntExp(1)),
                        PlusExp(
                            IdnExp(IdnUse("row")),
                            IdnExp(IdnUse("p"))),
                        ShiftRightExp(
                            PlusExp(
                                IdnExp(IdnUse("rd")),
                                IdnExp(IdnUse("p"))),
                            IntExp(1)))))))))
```

You can find a number of other examples of weBCPL code and the corresponding source trees in the `src/test/resources/` directory of the assignment skeleton.

Running the syntax analyser and testing it

The skeleton for this assignment is designed to be run from within sbt. For example, to compile your project and run it on the file `src/test/resources/queens.b` (which is provided for you in the framework code) you use the following command at the sbt console prompt:

```
run src/test/resources/queens.b
```

Assuming your code compiles and runs, the run will print the tree that has been constructed (for correct input), or will print a syntax error message (for

incorrect input). In the same directory as the `queens.b` file you will find a file called `queens.out`, this contains the output produced by our weBCPL syntax analyser when it processed the `queens.b` code.

The project is also set up to do automatic testing. See the file `SyntaxAnalysisTests.scala` which provides the necessary definitions to test the syntax analyser on some sample inputs. Note that the tests we provide are only examples and are not sufficient to test all of your code. You must augment them with other tests.

You can run the tests using the `test` command in sbt. This command will build the project and then run each test in turn, comparing the output produced by your program with the expected output. Any deviations will be reported as test failures.

What you must hand in and how

A zip file containing all of the code for your project and a type-written report.

Submit every source and build file that is needed to build your program from source, including files in the skeleton that you have not changed. Do not add any new files or include multiple versions of your files. Do not include any libraries or generated files (run the sbt `clean` command before you zip your project). We will compile all of the files that you submit using sbt, so you should avoid any other build mechanisms.

Your submission should include all of the tests that you have used to make sure that your program is working correctly. Note that just testing one or two simple cases is not enough for many marks. You should test as comprehensively as you can.

Your report should describe how you have achieved the goals of the assignment. Do not neglect the report since it is worth 50% of the marks for the assignment.

Your report should contain the following sections:

- A title page or heading that gives the assignment details, your name and student number.
- A brief introduction that summarises the aim of the assignment and the structure of the rest of the report.
- A description of the design and implementation work that you have done to achieve the goals of the assignment. Listing some code fragments may be useful to illustrate your description, but don't give a long listing. Leaving out obvious stuff is OK, as long as what you have done is clear. A good rule of thumb is to include enough detail to allow a fellow student to understand it if they are at the stage you were at when you started work on the assignment.
- A description of the testing that you carried out. You should demonstrate that you have used a properly representative set of test cases to be confident that you have covered all the bases. Include details of the tests that

you used and the rationale behind why they were chosen. Do not just print the tests out without explanation.

Submit your code and report electronically in a single zip file called `ass2.zip` using the appropriate submission link on the COMP332 iLearn website by the due date and time. Your report must be in PDF format.

DO NOT SUBMIT YOUR ASSIGNMENT OR DOCUMENTATION IN ANY OTHER FORMAT THAN ZIP and PDF, RESPECTIVELY. Use of any other format slows down the marking and may result in a mark deduction.

Marking

The assignment will be assessed according to the assessment standards for the unit learning outcomes.

Marks will be allocated equally to the code and to the report. Your code will be assessed for correctness and quality with respect to the assignment description. Marking of the report will assess the clarity and accuracy of your description and the adequacy of your testing. 20% of the marks for the assignment will be allocated to testing.

Dominic Verity
Last modified: 17 September 2019