CITY
UNIVERSITY OF LONDON
EST 1894

IN2009: Language Processors

Coursework: Part 1, Parsing

Introduction

The coursework (both Part 1 and Part 2) is concerned with building an interpreter for the Moopl language. On completing Part 1 of the coursework, you should be able to:

- Use JavaCC to implement a lexer and parser for a context-free grammar, factoring the grammar where necessary to resolve any look-ahead issues which make the grammar unsuitable for recursive descent parsing.
- Add semantic actions to your JavaCC specification so that the parser builds an abstract syntax tree for every successfully parsed input.

Key Information

Module marks
This coursework is worth 15 of the 30 coursework marks for the module. This coursework is marked out of 15.

Submission deadline
This coursework should be submitted before **5pm on Sun 11th March 2018**. In line with school policy, late submissions will awarded no marks.

Submissions
You should submit your solutions in Moodle as *two separate* JavaCC (`.jj`) files: one named `Moop-grammar.jj` (this version should contain NO semantic actions and all productions should have return type `void`) and one named `Moopl.jj` (this version should contain semantic actions to build and return abstract syntax trees).

Marks & Feedback
Marks and feedback will be available as soon as possible, certainly on or before Wed 4th April 2018.

Plagiarism
If you copy the work of others (either that of fellow students or of a third party), with or without their permission, you will score no marks and disciplinary action for Academic Misconduct will follow.

Teams
If you chose to work in a team then you must work in that team. And you must actually work *as* a team. All members of a team must submit the same work in Moodle. All team members are required to contribute equally to both Part 1 and Part 2. All team members will receive equal marks.

The Task: Build an interpreter for Moopl

The overall task is to build an interpreter for the Moopl language. Download one of the archive files `moopl.tgz` or `moopl.zip` and extract the files (provided as a NetBeans project but you can just use the **`src`** folder if you don't use NetBeans):

- A skeleton JavaCC file `Moopl-grammar.jj`
- A Java file `TestTokens.java` for testing your token definitions.
- A Java file `Parse.java` to act as a harness for testing your parser.
- A directory `syntaxtree` containing the abstract syntax classes for Moopl.
- A directory `examples` of example Moopl programs.
- A directory `visitor` which contains the `Visitor` interface (you may inherit from `VisitorAdapter` rather than implement `Visitor` directly).
- A pretty-printer `PrettyPrint.java`, that uses `PrettyPrinter.java` in the `pretty` package. Uncomment the call to `PrettyPrinter` in `PrettyPrint.java` (but this will not compile until you do part **b**) below).
- A (currently empty) directory `interp` that you will need for Part 2.
- A `staticanalysis` directory. Currently contains just an example "maximum loop-depth" analysis but for Part 2 will contain the typechecker.
- Other Java files generated by JavaCC.

The only file you should modify is `Moopl-grammar.jj` (see **a** and **b** below). Later, in Part 2, you will be adding additional classes to the `interp` package.

a) [8 marks] ***Syntax and Grammar.*** You should:

i) Complete the `TOKEN` definitions in `Moopl-grammar.jj`. You can test your token definitions by writing a list of tokens in a text file (for example `tokens.txt`) and then running

```
java TestTokens tokens.txt
```

ii) Complete the grammar productions in `Moopl-grammar.jj`.
- The grammar avoids left-recursion and ambiguity, but, as discussed in the lectures, there are still lookahead issues that you will need to resolve.
- When implementing this grammar you should *not* include any semantic actions. The return type for each method should be `void`. Part **b** asks you to change this. You can test your parser by running

```
java Parse filename
```

(advice on unit testing will be given in the lectures).

You are required to submit this version as a separate file (`Moopl-grammar.jj`) so make a copy and put it somewhere safe before moving on to part **b**.

b) [7 marks] ***Abstract Syntax Tree.*** You should:

i) Make a copy of your completed JavaCC file. Call this copy `Moopl.jj`. Add semantic actions to `Moopl.jj` so that each production returns an appropriate type of abstract syntax tree, using the AST classes provided in package `syntaxtree`. In order to do this you will need to specify an appropriate return type for each production and construct and return an AST object. In most cases you will need to add local variable declarations at the start of your productions and you will need to assign appropriate values to those local variables within your semantic actions.

ii) The pretty-printer that we have provided will take an abstract syntax tree and turn it back into well formatted source code. You should use this to check that the AST built by your parser is a faithful representation of its input. You can do this by running:

`java PrettyPrint` *filename.*

You are required to submit this version as a separate file (`Moopl.jj`).

Looking ahead to Part 2 *(not yet released)*

Typechecking… in the lectures and labs we will look at how a symbol table is built and how this is used to implement a typechecker as a visitor that checks the AST of a Moopl program for type errors.

Interpreter… you will be asked to code an interpreter as a visitor that interprets the AST of a Moopl program.

Grading criteria

Solutions will be graded according to their functional correctness and clarity. Below are criteria that guide the award of marks.

70 – 100 [1st class] Work that meets all the requirements in full, constructed and presented to a professional standard. Showing evidence of independent reading, thinking and analysis.
60 – 69 [2:1] Work that makes a good attempt to address the requirements, realising all to some extent and most well. Well-structured and well presented.
50 – 59 [2:2] Work that attempts to address requirements realising all to some extent and some well but perhaps also including irrelevant or underdeveloped material. Structure and presentation may not always be clear.
40 – 49 [3rd class] Work that attempts to address the requirements but only realises them to some extent and may not include important elements or be completely accurate. Structure and presentation may lack clarity.
0 – 39 [fail] Unsatisfactory work that does not adequately address the requirements. Structure and presentation may be confused or incoherent.