IN2009: Language Processors

Coursework Part 2: The Interpreter

Submission: Submit a **zip** archive (**not** a rar file) of all your source code (the `src` folder of your project). We **do not want** the other parts of your NetBeans project, only the source code.

Note 1: Submissions which do not compile will get zero marks.
Note 2: You **must not** change the names or types of any of the existing packages, classes or public methods.

Introduction

This is the 2nd and final part of the coursework. In Part 1 you created a parser for the Moopl grammar which, given a syntactically correct Moopl program as input, builds an AST representation of the program. In Part 2 you will develop an interpreter which executes Moopl programs by visiting their AST representations.

For this part of the coursework we provide functional code (with limitations, see below) for parsing, building a symbol table, type checking and variable allocation.

Marks
This part of the coursework is worth 15 of the 30 coursework marks for the Language Processors module. This part of the coursework is marked out of 15.

Submission deadline
This part of the coursework should be handed in before **5pm on Sunday 22nd April 2018**. In line with school policy, late submissions will be awarded no marks.

Return & Feedback
Marks and feedback will be available as soon as possible, certainly on or before Tue 14th May 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 further disciplinary action will be taken against you.

Teams
If you chose to work in a team you must continue to work in the same team for Part 2. 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.

Getting started

Download either `moopl-interp.zip` or `moopl-interp.tgz` from Moodle and extract all files. Key contents to be aware of:

- A fully implemented Moopl parser (also implements a parser for the interpreter command language; see below).
- A partially implemented Moopl type checker.
- Test harnesses for the type checker and interpreter.
- A directory of a few example Moopl programs (see **Testing** below).
- Folder `interp` containing prototype interpreter code.

The type-checker is only partially implemented but a more complete implementation will be provided following Session 6. That version is still not fully complete because it doesn't support inheritance. Part d) below asks you to remove this restriction.

The VarAllocator visitor in the interp package uses a simple implementation which only works for methods in which all parameter and local variable names are different. Part e) below asks you to remove this restriction.

The three parts below should be attempted in sequence. When you have completed one part you should make a back-up copy of the work and keep it safe, in case you break it in your attempt at the next part. Be sure to test old functionality as well as new (regression testing). We will **not** assess multiple versions so, if your attempt at part d) or e) breaks previously working code, you may gain a better mark by submitting the earlier version for assessment.

c) [10 marks] ***The Basic Interpreter:*** complete the implementation of the Interpreter visitor in the interp package.

d) [3 marks] ***Inheritance***: extend the type-checker, variable allocator and interpreter to support inheritance.

e) [2 marks] ***Variable Allocation***: extend the variable allocator to fully support block-structure and lexical scoping, removing the requirement that all parameter and local variable names are different. Aim to minimise the number of local variable slots allocated in a stack frame. **Note**: variable and parameter names declared at *the same scope level* are still required to be different from each other (a method cannot have two different parameters called x, for example) and this is enforced by the existing type-checking code. But variables declared in *different blocks* (even when nested) can have the same name.

Exceptions

Your interpreter will only ever be run on Moopl code which is type-correct (and free from uninitialised local variables). But it is still possible that the Moopl code contains logical errors which may cause runtime errors (such as null-reference or array-bound errors). Your interpreter should throw a MooplRunTimeException with an appropriate error message in these cases. The *only* kind of exception your interpreter should ever throw is a MooplRunTimeException.

Testing

> The `examples` folder does **not** contain a comprehensive test-suite. You need to invent and run **your own tests**. The document *Moopl compared with Java* gives a concise summary of how Moopl programs are supposed to behave. You can (and should) also compare the behaviour of your interpreter with that of the online tool: https://smcse.city.ac.uk/student/sj353/langproc/Moopl.html (Note: the online tool checks for uninitialised local variables. Your implementation is **not** expected to do this.)

To test your work, run the top-level Interpret harness, providing the name of a Moopl source file as a command-line argument. When run on a type-correct Moopl source file, Interpret will pretty-print the Moopl program then display a command prompt (>) at which you can enter one of the following commands:

`:quit`
: This will quit the interpreter.

`:call main()`
: This will call the top-level proc `main`, interpreted in the context defined by the Moopl program. (Any top-level proc can be called this way).

`:eval` *Exp* `;`
: This will evaluate expression *Exp,* interpreted in the context defined by the Moopl program, and print the result. Note the required terminating semi-colon.

Testing your Expression visitors

To unit-test your Exp visit methods, run the top-level Interpret harness on a complete Moopl program (though it can be trivial) and use the `:eval` command. For example, to test your visit methods for the Boolean-literals (ExpTrue and ExpFalse), you would enter the commands

```
> :eval true ;
> :eval false ;
```

which should output 1 and 0, respectively. For the most basic cases, the Moopl program is essentially irrelevant (a single top-level proc with empty body may be sufficient). For other cases you will need to write programs containing class definitions (in order, for example, to test object creation and method call).

Testing your Statement visitors

To unit-test your Stm visit methods, write very simple Moopl programs, each with a top-level `proc main()` containing just a few lines of code. Run the top-level Interpret harness on these simple programs and enter the command

```
> :call main()
```

You will find a few examples to get you started in the folder `examples/unit-tests`. As for the Exp tests, simple cases can be tested using Moopl programs with just a main proc but for the more complex tests you will need to write Moopl programs containing class definitions.

Grading criteria
Solutions will be graded according to their functional correctness, and the elegance of their implementation. 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.