IN2009: Language Processors

Coursework Part 2: The Compiler

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 a compiler.

For this part of the coursework we provide functional code for parsing, building a symbol table, type checking and variable allocation. The language, Mapl, is a simplified version of Moopl which supports arrays but not classes or objects.

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 100.

Submission deadline
This part of the coursework should be handed in before **5pm on Sunday 31st March 2019**. 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 Wed 24th April 2019.

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 for Part 1 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.

Grading
Submissions will be graded based on how many tests they pass. Submissions which do not compile cannot be tested and will receive zero marks.

Other stuff you need to know
See **Other stuff you need to know** at the end of this document.

Getting started

Download `mapl-compiler.zip` from Moodle and extract all files. Key contents to be aware of:

- A Mapl parser (in package `parser`).
- A Mapl type checker (in package `staticanalysis`).
- A Mapl pretty-printer (in package `pretty`).
- Top-level test-harness programs for the above.
- A directory of a few example Mapl programs (see **Testing** below).
- A prototype compiler (in package `compiler`).
- A top-level program `Compile` for running your compiler: this creates a `.ir` file containing the IR code generated by your compiler.
- A compiled library `Machine.jar` which provides three programs:
  - `IRPretty`: pretty-prints an `.ir` file.
  - `IRCompile`: takes an `.ir` file as input and creates two new files: binary machine code (with extension `.disc`) and a human-readable assembly version of the same code (with extension `.ass`).
  - `Exec`: for running`.disc` binaries.

The five 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 the old functionality as well as the new (regression testing). We will **not** assess multiple versions so, if a later attempt breaks previously working code, you may gain a better mark by submitting the earlier version for assessment.

Your compiler should generate code which implements all assignable values as integers, as follows:

- **int** values: implement in the obvious way
- **boolean** values: use 0 for false and 1 for true
- **array** values: like Java, Mapl uses reference semantics for arrays; implement as an integer which is a memory address within the heap where the array data resides. Null references are implemented as 0.

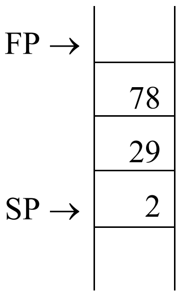
c) [50 marks] ***The Basic Compiler:*** partially complete the implementation of the Compiler visitor in the compiler package but **don't yet implement** support for method declaration, method calls, or arrays. The prototype code assumes that a program consists of a single initial "main" procedure (the actual method name does not matter) which has zero parameters, and it just compiles the body of this procedure. A proper treatment of variables is not possible in this version: compile variables to TEMP expressions for now (this will not give correct results for programs with nested variable declarations).

> **Note**: the correct semantics for the Boolean **`and`** operator is the same as in Java: so-called "short-circuit" semantics, in which the second argument is never evaluated if the first one evaluates to false. However, this complicates the task of code generation (because none of the IR binary operators have short-circuit semantics). For these marks you can ignore this issue.

d) [15 marks] ***Methods***: add support for method declaration and method call. Variables now must be implemented as MEM expressions using offsets from TEMP FP. You should no longer assume that the initial procedure always has zero parameters. Instead, your generated IR code should start with code which loads command-line argument values from the stack and then calls the top-level procedure using these as the actual parameters. You can generate code under the assumption that all command-line arguments have been pushed on the stack, followed by an argument count. For example, suppose you execute a compiled program as follows:

```
java -cp Machine.jar Exec max.disc 78 29
```

Before executing `max.disc`, the Exec program will initialise the machine state so that the stack looks like this:



Note: initially, FP points to an address just before the base of the stack, so your generated code can either use negative offsets from FP or positive offsets from SP to load the command-line arguments. Contrast this with the code that you generate for variables in method bodies, which should always offset from FP.

e) [**15 marks**] ***Arrays***: add support for arrays. Your generated code will need to call the pre-defined **`_malloc`** method to allocate heap memory for array creation. For these marks you are **not** required to generate code for bounds-checking.

f) [**10 marks**] ***Short-circuit* `and`**: generate code for the Boolean **`and`** operator which implements short-circuit semantics.

g) [10 marks] ***Bounds checks***: generate code which detects array bounds errors and outputs an error message before halting execution. To generate string data (for your error message) you will need to include a `strings {…}` block at the start of your generated IR code. To print a message, your generated code will need to call the pre-defined **`_printstr`** method.

Testing

The `examples` folder does **not** contain a comprehensive test-suite. You need to invent and run **your own tests**. The document *Mapl compared with Java* gives a concise summary of how Mapl programs are supposed to behave.

To test that you are generating syntactically valid IR code, run your compiler:

```
java Compile examples/hello.mapl
```

(recompile everything first!) then pretty-print the result:

```
java -cp ../Machine.jar IRPretty examples/hello.ir
```

If you don't get any syntax errors, you can then compile and execute the IR code:

```
java -cp ../Machine.jar IRCompile examples/hello.ir
java -cp ../Machine.jar Exec examples/hello.disc
```

If it doesn't execute as you expect, then you should study your pretty-printed IR code to see why. (You can also look at the assembly code, but this is less likely to be useful for debugging your compiler.) As always, test incrementally throughout development, and craft test inputs which are as simple as possible for the behaviour that you want to test.

Other stuff you need to know

To compile to correct IR code, you need to know a few things about what the IR compiler will do with it:

1. TEMP names. TEMP's get compiled to registers. You are free to invent any names you like for TEMP nodes (taking care to avoid name clashes) but three are treated specially by the IR compiler:

   FP is the frame pointer
   SP is the stack pointer
   RV this where your compiled function bodies must leave their return values

   You also need to be aware that the number of machine registers is limited (32 in total, but only 23 are available for general purpose use). So be economical in your use of TEMP names.

2. LABEL names. There is no limit on the number of label names that you can use. For the most part, you should use `FreshLabelGenerator.makeLabel` to create label names (each use is guaranteed to generate a fresh name). You can also invent your own label names, but don't use any names that start with an underscore: these are reserved for pre-defined routines provided by the IR compiler.

3. Pre-defined routines. The IR compiler provides the following routines which you can call in your generated IR code, as required. Each of them takes a single parameter.

   `_printchar` : the parameter is interpreted as the 16-bit integer code for a Unicode character (higher order bits are ignored).
   `_printint` : the parameter is an integer which will be printed as text (with no newline).
   `_printstr` : the parameter is a memory address for a null-terminated string constant; in practice, you will always specify the parameter as NAME *lab*, where *lab* is a label name defined in the (optional) `strings` section at the start of your generated IR code.
   `_malloc` : the parameter is the number of words of memory to allocate; the start address of the allocated block is returned.

4. Method declarations: You will compile a method declaration by compiling the statements in its body. To turn this into machine code which can be called and returned from, the IR compiler needs to top-and-tail the code with instructions for pushing and popping a stack frame. To give the IR compiler the information it needs, you must add a PROLOGUE node at the start and an EPILOGUE node at the end of the IR code that you generate. Both these nodes require two non-negative integers: the number of parameters and the number of local variable slots, respectively.