IN2009: Language Processors

Coursework Part 2: The Compiler

Submission: Submit a **zip** archive (**not** a rar file) of all your source code. If you have developed in an IDE **do not** submit the entire project structure, only the source code and before you submit check that your source code can be compiled and executed using command line tools alone (**javac** and **java**).

Note 1: Submissions which do not compile (see above) 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.

Module marks
This coursework is worth 50% of the coursework marks for the module. This coursework is marked out of **50**.

*Submission deadline [**EXTENDED**]*
This part of the coursework should be handed in before ~~5pm on Sunday 29th March 2020~~ **5pm on Sunday 19th April 2020**. In line with school policy, late submissions will be awarded no marks.

Marks & Feedback
Marks and feedback will be available on or before Mon 20th April 2020.

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 pair for Part 1 you must continue to work in the same pair for Part 2. Both members of a pair must submit the same work in Moodle. Both team members are required to contribute equally to both Part 1 and Part 2. Both team members will receive equal marks.

Grading
Submissions will be graded primarily 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-Cwk2.zip` from Moodle and extract all files. Key contents to be aware of:

1. Source code:
   - A Mapl parser (in package `mapl.parser`).
   - A Mapl type checker (in package `mapl.staticanalysis`).
   - A prototype compiler (`mapl.compiler.Compiler`).
   - A top-level program `mapl.Compile` for running your compiler: this creates an `.ir` file containing the IR code generated by your compiler.
2. A jar of compiled library code `Backend.jar` . This provides the IR AST classes (package `ir.ast`) an abstract machine (package `tac`) an IR parser (package `ir.parser`) an IR compiler (package `ir.compiler`) and top-level programs:
   - `ir.Compile`: takes an `.ir` file as input and creates two new files: binary machine code (with extension `.tac`) and a human-readable assembly version of the same code (with extension `.tacass`).
   - `tac.Exec`: for running `tac` binaries.
3. A directory of a some example Mapl programs (see **Testing** below).

Study the code for the prototype compiler: `mapl/compiler/Compiler.java`. You will find the following:

1. A `compile` method which takes a Mapl program AST as a parameter and returns an IR program AST as its result.
2. A number of convenience static factory methods for building IR ASTs. You will be writing code which uses these factory methods to build an IR program (you don't strictly need to use the factory methods, since you could use IR AST constructors directly, but it will make your life much easier if you do).
3. Two inner classes `StmCompiler` and `ExpCompiler`. The first of these is a Visitor implementation whose visit methods return lists of IR statements; these are used to translate Mapl method declarations and Mapl statements. `ExpCompiler` is a Visitor implementation whose visit methods return IR expressions; these are used to translate Mapl expressions. Your job will be to complete these Visitor implementations as well as the top-level `compile` method. Note that you only need to provide visit methods for the relevant ASTs (`ExpCompiler` should not implement visit methods for any of the Mapl Stm ASTs, for example.)

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.

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.

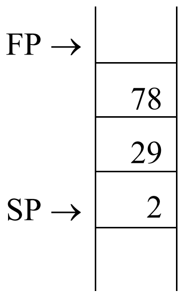
a) [20 marks] ***The Basic Compiler:*** partially complete the implementation of `mapl.compiler.Compiler` but **don't yet implement** support for method declaration, method calls, or arrays. The prototype code assumes that a program consists of a single initial "main" procedure (the actual method name 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.

b) [10 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 calls the top-level procedure using these as the actual parameters. **Note**: the call to the initial procedure **must** be followed by a JUMP to the label _END, otherwise execution will fall through to the following code (the symptom is likely to be an infinite looping behaviour). Label _END is pre-defined and added by the IR compiler (do **not** add your own).

You should 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 CompilerBackend.jar tac.Exec prog.tac 78 29
```



Before executing `prog.tac`, the Exec program will initialise the machine state so that the stack looks like the picture on the left.

Note: initially, FP points to an address just before the base of the stack. Your generated code can use negative offsets from FP to access the command-line arguments.

c) [**10 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.

d) [**5 marks**] ***Short-circuit* `and`**: generate code for the Boolean **`and`** operator which implements short-circuit semantics.

e) [5 marks] ***Array checks***: generate code which detects out-of-bounds errors during execution of array creations, updates and look-ups; the code should output a meaningful error message then halt execution. To generate string data for your error messages you can include a `strings {…}` block at the start of the generated IR program. To print a message, your generated code can call the pre-defined **`_printstr`** method.

Testing

When you have a partially working compiler you can test it by compiling Mapl programs to IR code, then compiling the IR code to a `tac` executable, then executing the `tac` code. For example, in the `src` directory (after first ensuring that `Backend.jar` is on your java classpath) you might do:

```
javac mapl/Compile.java
java mapl.Compile ../examples/methods/counter.mapl
java ir.Compile ../examples/methods/counter.ir
java tac.Exec ../examples/methods/counter.tac 5
```

The expected output in this case would be: `5 4 3 2 1 0`

The provided examples do **not** comprise 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.

If the IR code generated by your Mapl compiler is rejected by the IR compiler, or doesn't execute as you expect, then you should study the `.ir` file to see why. (If it is accepted by the IR compiler you can also look at the assembly code in the `.tacass` file, but this is less likely to be useful for debugging your Mapl 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 are the IR counterpart to machine registers (and will in fact be compiled to registers by the backend compiler). Some names 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

   Apart from these, you are free to invent any names you like for TEMP nodes but care is needed to avoid name clashes so you are advised to use the `FreshNameGenerator.makeName` methods.

   You will probably notice that in some cases it would be possible to be more economical and reuse the same TEMP name in different parts of your code: **DON'T** be tempted to do this. Firstly, it is easy to get this wrong, leading to some very subtle bugs in your compiled code (IR code is deliberately designed to allow an unbounded number of "registers" so that you can avoid these issues). Secondly, even if you manage to get it right, it is actually likely to result in lower-quality executable code because of the register-allocation algorithm used by the backend compiler (register allocation will be covered in one of the later lectures).

2. LABEL names. For the most part, you should use `FreshNameGenerator` to create label names, since label names *must* be unique (but see the remarks below about compiling method declarations). Don't create any labels with names that start with an underscore: these are reserved as label names for use by the backend IR compiler.
3. Pre-defined labels. 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 an integer which will be interpreted as a 16-bit Unicode Plane 0 code point of a character to be printed (the 16 higher-order bits of the integer are ignored). Note that the first 128 code points coincide with ASCII.
   `_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; any valid memory address can be used but 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. Note that `_malloc` will allocate memory which is not currently in use but makes **no** guarantees about the *contents* of the allocated memory (it may contain arbitrary junk).

   The IR compiler also adds a label **`_END`** to the very end of the compiled code.

4. Method declarations: You will compile a Mapl method declaration by compiling its body into a sequence of IR statements, starting with LABEL *foo*, where *foo* is the Mapl method name; for functions, your code should also ensure that the return value is stored in TEMP RV. To compile the IR sequence into machine code which can be called and returned from, the backend IR compiler needs to top-and-tail the code that it generates with instructions for pushing and popping a stack frame; to enable this, your generated code must include a PROLOGUE at the start (immediately after LABEL *foo*) and an EPILOGUE at the end.