IN2009: Language Processors
Resit Coursework

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: Any parts of a submission which do not compile (see above) will get zero marks.

1. Table of Contents

1. Submission & Grading

This task requires you to implement a parser and compiler for a simple, untyped programming language called LPL. The grammar of LPL and some code are provided to get you started.

Submission deadline
You must submit before **23:59** on **26th August 2020**. In line with school policy, late submissions will not be marked.

Plagiarism
This is an *individual* task, no team work is permitted. If you copy the work of others (either that of fellow students or a third party), with or without their permission, you will score no marks and disciplinary action will follow.

Grading
This task is graded out of **100**. The pass mark is **40**. 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.

2. Getting started

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

1. The LPL language grammar: `LPL-grammar.pdf`
2. Source code:
    - A package (incomplete) of LPL abstract syntax tree classes in `src/lpl/ast`.
    - A partial JavaCC implementation of an LPL parser in file `src/lpl/parser/LPL.jj`.
    - A top-level program `lpl.Parse` for running your parser: this creates an .`html` file which renders a visual description of the AST generated by your parser.
    - A top-level program `lpl.Compile` for running your compiler: this creates an `.ir` file containing the IR code generated by your compiler.
    - Some utilities in `src/lpl/util`.
3. A jar of compiled library code `Backend.jar` . This provides the IR AST classes (package `ir.ast`) an abstract machine (package `tac`) 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.
4. A directory of a some example LPL programs.

3. LPL Language Description

LPL is a small, untyped programming language. An LPL program consists of a non-empty sequence of function declarations (see the grammar for details). When an LPL program is executed, the "top-level" function (the first one in the sequence) is called; if the top-level function has any formal parameters then actual parameter values must be provided on the command-line.

To simplify the task of implementing LPL, the language has some unusual features and limitations:

i) LPL has if-statements but no loop constructs. However, LPL does allow recursive function definitions, so recursion can be used instead of iteration.

ii) All expressions evaluate to integer values. In the condition of an if-statement, any value less than or equal to zero is treated as false, any value greater than 0 is treated as true. The **<** and `==` operators return 0 for false and 1 for true.

iii) There is no support for strings but the `printchar` statement can be used to output characters one at a time (the integer argument to `printchar` is interpreted as a Unicode character code - ASCII codes can be used for ordinary characters).

iv) Function definitions have formal parameters but no other local variables. This does not limit the expressive power of LPL: parameters can be assigned new values in the body of a function and a programmer is free to declare "dummy" parameters to be used as additional locals.

v) A function call returns when a `return` statement is executed, or it returns by default when the end of the function body is reached. If a function returns by default, it returns the value 0.

vi) Because LPL has no type checker your compiler may be presented with programs which have no well-defined meaning. Your compiler is *not* expected to detect or handle such cases. Examples include:

- Programs which call functions which are not declared.
- Programs which call functions with the wrong number of parameters.
- Programs containing uses of undeclared formal parameters.
- Programs which redefine the built-in library functions (see below).
- Programs which try to use the built-in library functions `fst` and `snd` to dereference integers which are not valid memory addresses.

vii) The following are built-in library functions for working with dynamically allocated data (see the example LPL programs in `examples/heap)`:

- `empty()`: returns 0.
- `isempty(`*x*`)`: returns true (1) or false (0) according to whether *x* is 0 or not.
- `fst(`*x*`)`: returns the first element of the pair stored in the heap at address *x*.
- `snd(`*x*`)`: returns the second element of the pair stored in the heap at address *x*.
- `cons(`*x*, *y*`)`: allocates a pair of memory locations in the heap, initialises them to contain the values *x* and *y*, and returns the memory address of the first element.

4. The Task

Study the contents of the folder: `src/lpl/ast`. You will find the following:

1. Abstract Syntax Tree classes `Program`, `FunDecl`, `Exp`, `ExpInteger`, `Stm`, `StmPrintint`. You will need to define additional AST classes corresponding to the various productions in the grammar (see part (a) below).
2. In the abstract class `Exp`, a `compile` method declaration with return type `IRExp` . You will need to implement appropriate versions of this method in each AST class which extends `Exp` (for parts (a) and (b) below it will suffice to provide placeholder implementations which either return null or throw an exception).
3. In the abstract class `Stm`, a `compile` method declaration with return type `List<IRStm>`. You will need to implement appropriate versions of this method in each AST class which extends `Stm` (for parts (a) and (b) below it will suffice to provide placeholder implementations).
4. In class `Program` a prototype `compile` method (you will modify this in part (d)).

In `src/lpl/util/IRFactory.java` you will find a number of static factory methods for building IR ASTs. When you implement the compile methods (parts (c, d, e)) you will write 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). Use a static import to make the factory methods available in each class where you want to use them.

Your compiler should generate code which implements all values as integers (in fact, you have no choice, since this is the only data type available in IR code).

There are five parts to the task: **a**, **b**, **c**, **d**, **e**. Attempt them 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, even if it is less complete.

For Parts **c**, **d**, **e** you will need to read Section **6** below.

a) **[20 marks]** ***The AST classes:*** based on the LPL grammar, define appropriate additional AST classes in the package `lpl.ast`. Classes for expressions must extend `Exp` and classes for statements must extend `Stm` (for now, you can write placeholder implementations of the `compile` functions which either return null or throw an exception).

b) **[30 marks]** ***The Parser:*** complete the parser implementation in `src/lpl/parser/LPL.jj`. You will need to add token definitions for the missing terminal symbols and JavaCC productions for each part of the grammar. Your JavaCC productions should use semantic actions to build ASTs. Remember, each time you edit `LPL.jj` you need to re-run `javacc` and then recompile the generated Java code.

c) [15 marks] *The Basic Compiler:* implement the `compile` methods in your AST classes for the following parts of the language:

- integer constant expressions
- binary operator expressions (those involving, `+`, `*`, <, etc)
- if-statements
- print-statements (`printint` is already done for you)

This will be sufficient to allow you to compile all the example LPL programs in the folder `examples/basic`.

The prototype `compile` implementation in `Program.java` assumes that there is only one function declaration - the top-level function - and that this function has no parameters; `compile` simply compiles the body of the top-level function. **Note**: although the prototype code passes a `FunDecl` parameter to the Stm `compile` methods, none of the cases you implement for this part will actually make any use of the `fd` parameter (it will be needed for variable access in the next part).

d) [20 marks] *The Stack*: add support for function declaration and function call. You will now need to implement all the remaining `compile` methods, including the one in `FunDecl.java`.

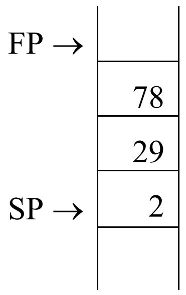
This will be sufficient to allow you additionally to compile all the example LPL programs in the folder `examples/stack`.

Variables must be implemented as MEM expressions using offsets from TEMP FP. The `FunDecl` class provides a method which allows `compile` methods to use their `fd` parameter to calculate the required offset for a formal parameter (recall that the only variables in LPL are formal parameters). Use the stack-frame layout defined in Session 7 (slide 15).

The `compile` method in `Program` should now be changed to generate code which starts with a *call* to the top-level function; the actual parameters will be IR expressions which access the command-line arguments from the call stack (see below). This will be followed by compiled versions of all the `FunDecls` (to compile a `FunDecl`, call its `compile` method). **Note**: the call to the top-level function *must* be followed either by a JUMP to `_END` or a call to the pre-defined `_exit` function, otherwise execution will fall through to the compiled function code. 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 Backend.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.

e) [**15 marks**] ***The Heap*** : add support for dynamically allocated data. Modify the `compile` method in `Program.java` to add IR code implementing the built-in library functions (see **LPL Language Description** above for details). Your IR code for `cons` will need to call the pre-defined `_malloc` method to allocate heap memory.

This will allow you to compile the remaining example LPL programs in folder `examples/heap`.

5. Testing

Parser

To test your parser, use the program `lpl.Parse`. For example:

```
javac lpl/Parse.java
java lpl.Parse ../examples/stack/counter.lpl
```

This will call your parser to parse `counter.lpl` and, if successful, it will render an html description of the abstract syntax tree generated by your parser, leaving it in a file called `counter.html`. Open the html file in a browser and check that the AST corresponds correctly to the LPL source.

Compiler

To test your compiler, compile LPL programs to IR code, then compile the IR code to a `tac` executable, then execute the `tac` code. For example, in the `src` directory (after first ensuring that `Backend.jar` is on your java classpath) you might do:

```
javac lpl/Compile.java
java lpl.Compile ../examples/stack/counter.lpl
java ir.Compile ../examples/stack/counter.ir
java tac.Exec ../examples/stack/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**.

If the IR code generated by your 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 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.

6. Key information about the IR compiler

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

1. LABEL names. Label names must be unique. You can use the `lpl.util.FreshNameGenerator.makeName` methods to generate unique names 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.
2. 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 other names you like for TEMP nodes but for this simple language there will be very few places in your code where you actually need to do so (in part (e) you will probably need a TEMP in the code for `cons`).
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.

   `_exit` : terminates execution. The parameter is an integer which is used as the exit-status code for the terminated process.
   `_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` : [you probably won't need to use this] 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. Function declarations: You will compile a function declaration by compiling its body into a sequence of IR statements, starting with LABEL *foo*, where *foo* is the function name; your code must also ensure that the return value is stored in TEMP RV.

   To compile the IR sequence for a function 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.