Homework: Compiler

The goal of this assignment is build a simple compiler for a language that has first-class functions (similar to languages you've been building so far). Producing fast code or small code is explicitly *not* a goal of this assignment. Instead, this assignment focuses on semantics-preserving program transformations that eliminate the need for language features. After all transformations are done, you'll be left with a program that can be trivially translated to assembly language. Your compiler will target the ARM Cortex A9 processor, which powers the Raspberry Pi 2, iPhone 4S, and several other tablets and smartphones. You won't need to own any such device: we'll show you how to simulate this processor using QEMU.

1 Setup

- Install QEMU. It is available using `apt-get` on Linux and `brew install` on Mac OS X.
- Download and unzip the file `cortex-a9-vm.tar.gz` that is linked on the course page. This archive contains a ARM-based Linux virtual machine, with tools like *gcc*, *as*, and *gdb* installed. You can install anything else you like on this VM.

2 Requirements

Write a program that takes two arguments:

`./compile.d.byte` *input output.s*

The argument *input* should name a file that contains a program written using the grammar in fig. 18.1. Your program should translate the input to ARM assembly and write it to a new file called *output.s*.

The assembly file produced above should link with the file init.s (that we've provided):

```
as -o init.o init.s
as -o output.o output.s
gcc -o output init.o output.o
```

The result, output, should be an executable that runs on the provided virtual machine:

```
./output
```

Similar to earlier assignment, the output program can print anything you find useful to the screen. When the program terminates normally, it should exit with code 0. However, if it exits abnormally (e.g., because the abort expression is evaluated), it should exit with a non-zero exit code.

3 Support Code

The `Compiler_util` module defines an abstract syntax for the language and a parser for the concrete syntax shown in the figure. It also defines a grammar for the A normalized subset of the language and several other helper functions. As usual, you do not need to use this code, but you will probably fine it useful.

The module `Arm` has functions to generate a very small subset of ARM assembly that should be adequate to complete this assignment. It has a few conveniences (see the support code documentation) but is otherwise very bare-bones.

Expressions

e	::=	x	`Id x`	where x has letters, digits, and underscores
	|	n	`Const (Int n)`	where n is a decimal integer
	|	`true`	`Const (Bool true)`	Boolean true
	|	`false`	`Const (Bool false)`	Boolean false
	|	e_1 `+` e_2	`Op2 (Add,` e_1`,` e_2`)`	Integer addition
	|	e_1 `-` e_2	`Op2 (Sub,` e_1`,` e_2`)`	Integer subtraction
	|	e_1 `*` e_2	`Op2 (Mul,` e_1`,` e_2`)`	Integer multiplication
	|	e_1 `/` e_2	`Op2 (Div,` e_1`,` e_2`)`	Integer division
	|	e_1 `%` e_2	`Op2 (Mod,` e_1`,` e_2`)`	Modulus
	|	e_1 `<` e_2	`Op2 (LT,` e_1`,` e_2`)`	Integer less-than
	|	e_1 `>` e_2	`Op2 (GT,` e_1`,` e_2`)`	Intege greater-than
	|	e_1 `==` e_2	`Op2 (Eq,` e_1`,` e_2`)`	Equality of booleans and integers
	|	`(`e`)`	`e`	Parentheses
	|	e_f `(`e_1`,`$\cdots$`,`e_n`)`	`App (`e_f`, [`e_1`;`$\cdots$`;`e_n`])`	Function application
	|	`if` e_1 `then` e_2 `else` e_3	`If (`e_1`,` e_2`,` e_3`)`	Conditional (e_1 must be a boolean)
	|	`let` x `=` e_1 `in` e_2	`Let (`x`,` e_1`,` e_2`)`	Let binding
	|	`fun` f `(`x_1`,`$\cdots$`,`x_n `) ->` e	`Fun (`f`, [`x_1`;`$\cdots$`,`x_n`],` e`)`	Recursive function definition
	|	`array(`e_1`,`e_2`)`	`MkArray (`e_1`,`e_2`)`	New array of length e_1 with all elements initialized to e_2
	|	e_1`[`e_2`]`	`GetArray (`e_1`,` e_2`)`	Array indexing
	|	e_1`[`e_2`]` `=` e_3	`SetArray (`e_1`,` e_2`,` e_3`)`	Array update
	|	`abort`	`Abort`	Abort immediately with non-zero exit code

Figure 18.1: The concrete syntax and abstract syntax of the language.

4 Assumptions

1. You may assume that the size of the heap and stack are large enough to run any program. There is no need to implement garbage collection. (But go ahead and do it, it is fun!)

2. You may assume that the program is type correct. (If you want, you can easily adapt your type inference implementation for this language.)

3. After closure conversion and A-normalization, you may assume that all sub-expressions of the program have at most five free variables. Since you have 12 registers available, you can implement register allocation naively and don't need to worry about register-spilling.

4. You may implement any calling convention you like.

5. You may **not** assume that array accesses will be in bounds. If a program tries to access get or set an index that is out of bounds, it must abort cleanly with a non-zero exit code.

5 Directions

1. Get familiar with ARM assembly. We recommend the guides linked on the course page.

2. Read the file `init.s` and understand what it does for you. You're free to modify this file in any way (or to not use it at all).

3. Do not write to build the compiler all at once, but break the problem into pieces. Here is one suggested breakdown:

   (a) Implement arithmetic and let-binding, ignoring functions, arrays, and conditionals. You will still have to implement register allocation and a core piece of A-normalization and get them right.

   (b) Implement conditionals and boolean expressions.

   (c) Implement arrays.

   (d) Implement closures.

6 Template and Hand In

A template file for the assignment is provided on the course web page. Solve the assignment in this file and submit only this file using Moodle.

In addition:

1. If you haven't modified `init.s` and don't need to change the linking process described above, just handin `compile.ml`.
2. If you've modified `init.s` but continue to use the linking process described above, handin both `compile.ml` and `init.s`.
3. If you've changed the linking process, include a `README` and any other files needed to run your code.