Homework: Compiling with Continuations

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. Unfortunately, real assembly languages are complex and require you to address a low-level issues that take a long time to understand. Therefore, we will target a small assembly language that we've designed for this assignment that we call ILVM. ILVM provides a few convenient features that will save a lot of time. You can read about the assembly language and download its implementation here:

https://github.com/plasma-umass/ilvm

1 Requirements

Write a program that takes two arguments:

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

The argument *input* should name a file that contains a program written using the grammar in fig. 19.1. Your program should translate the input to ILVM code write it to a new file called *output.il*.

You should be able to run the output file using the ILVM evaluator:

```
ilvm output.il
```

Your compiled program can print anything to the screen. However, it if source program exits normally, it should execute the `exit` instruction in ILVM. Conversely, if the source program encounters an error, the compiled program should execute the `abort` instruction.

2 Support Code

The `Compiler_util` module defines an abstract syntax for the language and a parser for the concrete syntax shown in the figure.

3 Assumptions

1. The ILVM evaluator has command-line arguments that you can use to customize the size of the heap and the number of available registers.

2. You do not have to implement garbage collection. (But go ahead and do it, it is fun!)

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

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 get or set an index that is out of bounds, it must abort cleanly with a non-zero exit code.

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 19.1: The concrete syntax and abstract syntax of the language.

4 Directions

1. Get familiar with ILVM. The repository README has several small examples.
2. Do not try 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 CPS and get them right. You'll be able to ignore memory loads and stores while you implement this bit.
   (b) Implement conditionals and boolean expressions.
   (c) Implement arrays. To do this, you'll have to learn how to load and store values from memory.
   (d) Implement first-order functions. i.e., assume that functions have no free variables.
   (e) Implement higher-order functions. To do so, you'll need to implement closure-conversion.