CMSC 22610	Implementation of	Project 4
Winter 2015	Computer Languages I	March 1

Flang bytecode generation

Due: March 16 at 10pm

1 Introduction

The final part of the project is to implement a code generator for **Flang**. The target of the code generator will be a virtual machine (VM) that is described in a separate document. This project involves two stages:

1. Translate the typed AST produced by the type checker to a simple untyped first-order IR. This translation replaces higher-level constructs, such as data constructors, nested functions, and pattern matching with lower-level code.
2. Generate byte code instructions for a stack-based virtual machine from the Simple IR produced by Step 1.

Unlike the previous projects, we will expect you to design and implement your own data structures for the intermediate representation.

2 A simplified IR

Figure 1 gives a grammar for an untyped first-order functional language that can be used as an intermediate representation (IR) for **Flang** programs prior to code generation. A program in this representation consists of a sequence of function definitions, where the last definition is a function that takes no arguments and contains the body of the program. A function definition is written as

$$\mathbf{fun}\ f\ (\overline{y};\ \langle\overline{x}\rangle) = e$$

where the $\overline{x}$ are the *free variables* of the function and the $\overline{y}$ are the function parameters.

The expression forms for the Simple IR are mostly familiar, but there are a couple of new forms. In addition to function application, there is a *primitive* application form that is used for implementing the primitive operations (excluding `::`) and built-in functions. These operations are mapped to special implementations during code generation, so it is useful to identify them at this stage of the translation. Another new form is the tuple selection operation, which is used to extract elements from a tuple (tuple indexing is 0-based). The last new form is the special variable **self**, which is used in the implementation of self-recursive functions (see Section 4).

$$
\begin{array}{lcll}
p & ::= & d^{+} & \\
d & ::= & \textbf{fun}\, f\, (\overline{y};\, \langle \overline{x} \rangle) = e & \text{closed function definition} \\
e & ::= & \textbf{let}\, x = e_1\, \textbf{in}\, e_2 & \text{local binding} \\
 & | & \textbf{if}\, e_1\, \textbf{then}\, e_2\, \textbf{else}\, e_3 & \text{conditional} \\
 & | & e_1\, (\overline{e};\, e_2) & \text{function application} \\
 & | & p\, (\overline{e}) & \text{primitive-operator application} \\
 & | & \langle \overline{e} \rangle & \text{tuple construction} \\
 & | & e.i & \text{select } i\text{th component of tuple} \\
 & | & f & \text{function} \\
 & | & x & \text{variable} \\
 & | & \textbf{self} & \text{the current function's closure} \\
 & | & n & \text{number} \\
 & | & s & \text{string literal}
\end{array}
$$

Figure 1: The grammar of the Simple IR

As an example, consider the following **Flang** program:

```
let n = 1;
fun inc (i : Integer) -> Integer = i + n;
inc 2
```

This example can be represented as the following Simple IR code:

$$
\begin{array}{l}
\textbf{fun}\, \mathrm{inc}_{\mathit{def}}\, (\mathrm{i};\, \langle \mathrm{n} \rangle) = +\, (\mathrm{i}, \mathrm{n}); \\
\textbf{fun}\, \mathrm{main}\, (-;\, \langle\rangle) = \\
\quad \textbf{let}\, \mathrm{n} = 1\, \textbf{in} \\
\quad \textbf{let}\, \mathrm{inc} = \langle \mathrm{inc}_{\mathit{def}}, \mathrm{n} \rangle\, \textbf{in} \\
\quad\quad \mathrm{inc}.0\, (2;\, \mathrm{inc});
\end{array}
$$

Here we are representing the function `inc` as a pair of its definition (*i.e.*, code) and the variable `n`, which is free in its body. Functions are written with two parameters: their environment (enclosed in $\langle\ \rangle$) and their regular parameter. We write "$-$" where there are no regular parameters (*e.g.*, the main function). We describe the details of this translation in more detail in the next four sections.

3 Data constructor representations

All **Flang** values are represented by a single machine word. In many cases, this word is a pointer to heap-allocated storage, but it might also be an immediate value. We refer to values that are represented as pointers as *boxed* values, while values that are represented as immediate integers are *unboxed*. As we explain below, the values of a datatype may be both boxed and unboxed, in which case we describe the type as having a *mixed* representation. Since type variables can be instantiated to any type, they have a mixed representation. The `Integer` type maps directly onto unboxed integers and the `String` type is represented as a pointer. Function types are also represented as pointers and are described below in Section 4.

The representation of data constructors is the interesting case. Let T be a data type with n nullary constructors $C_1, \ldots, C_n$ and m data-constructor functions F_1 `of` $\tau_1, \ldots, F_m$ `of` τ_m.

The following table gives the representation of the various constructors based on the number of constructors and the representations of the τ's.

n	m	C_i	$F_j(v)$	T's representation
> 0	0	i	n.a.	unboxed
0	1	n.a.	v	τ_1's representation
> 0	1	i	v if τ_j is boxed	mixed
> 0	1	i	$\langle v \rangle$ if τ_j is unboxed or mixed	mixed
0	> 1	n.a.	$\langle j, v \rangle$	boxed
> 0	> 1	i	$\langle j, v \rangle$	mixed

In this chart, i means the immediate (unboxed) value i and $\langle \cdots \rangle$ means a heap-allocated tuple. Applying this algorithm to the the builtin datatypes we get:

$$\begin{aligned} \texttt{Unit} &\rightarrow 0 \\ \texttt{False} &\rightarrow 0 \\ \texttt{True} &\rightarrow 1 \\ \texttt{Nil} &\rightarrow 0 \\ a\texttt{::}b &\rightarrow \langle a, b \rangle \end{aligned}$$

3.1 Data constructors as values

Data-constructor functions can appear in contexts other than application. In this case, the compiler needs to η-convert the constructor. For example, we might map a constructor over a list as in the following example:

```
data Point with con Pt of Integer * Integer;
  ...
    map [Integer * Integer, Point] Point
      ((1,2)::(3,4)::Nil[Integer*Integer])
  ...
```

The compiler needs to replace the occurrence of `Point` here with the expression

```
{ fun mkPoint (arg : (Integer * Integer)) -> Point = Point arg; mkPoint }
```

As an optimization, if there are multiple occurrences of `Point` as a value, they can share a single `mkPoint` function defined at the top level.

3.2 Polymorphic data constructors

In addition to handling data-constructor functions used as values, we must also consider how to handle polymorphic data types. For example, consider the following **Flang** code fragment:

```
data Option[a] with
  con None
  con Some of a;
let x : Option[Integer] = None[Integer];
let y : [a] Option[a]   = None;
let z : Option[Boolean] = y[Boolean];
```

Based on the algorithm above, we would represent `None` as the immediate value 0, but the question

remains about how to handle the type argument? Since types do not have a run-time significance, we would like to treat application of a polymorphic data constructor to type arguments as a no-op.[1] This choice means that the right-hand-side of `y`'s binding has to be η-converted as we do for data-constructor functions that are used as values.

```
let y : [a] Option[a] = {fun mkNone [a] -> Option[a] = None[a]; mkNone};
```

Strictly speaking, the eta-converted `none` function is not legal **Flang** code, since it violates the requirement that functions have at least one value parameter, but we relax this restriction inside the compiler.

4 Representing functions

In a lexically-scoped higher-order language like **Flang**, we need a representation of function values that will support passing them as arguments, returning them as results, and embedding them in data structures. We use heap-allocated flat closures for this purpose. A function

```
fun f (y : τ) -> τ' = exp;
```

with free variables $x_1, \ldots, x_n$ is represented in the Simple IR as a tuple of $n + 1$ elements: $\langle f_{def}, x_1, \ldots, x_n \rangle$, where f_{def} is the name of the first-order function that implements f. We call this tuple the *closure* of f. The definition of f_{def} will have the form

$$\mathbf{fun}\, f_{def}\,(y;\ \langle x_1,\ \ldots,\ x_n\rangle) = \widehat{exp}$$

where $\widehat{exp}$ is the translation of exp to the Simple IR.[2] The example on Page 2 illustrates this translation.

Function applications are translated into applying the 0th component of the function's closure to the pair of the function's closure and argument. More formally, the application "$e_1\ e_2$" is translated to the Simple IR expression

$$\mathbf{let}\ f = \widehat{e_1}\ \mathbf{in}\ f.0\,(\widehat{e_2};\ f)$$

In the special case where we are applying making a self-recursive call, we can optimize the call by directly referring to the name of the definition, instead of extracting it from the closure, and using special variable **self** to refer to the closure. For example, consider the following program that computes the factorial of 5:

```
...
fun fact (n : Integer) -> Integer =
      if (n <= 1) then 1 else n * fact(n-1);
fact 5
```

This program can be translated to the following Simple IR representation:

$$\begin{array}{l}\mathbf{fun}\,\mathrm{fact}_{def}\ (\mathrm{n};\ \langle\rangle) = \mathbf{if}\ \leq\ (\mathrm{n}, 1)\ \mathbf{then}\ 1\ \mathbf{else} * (\mathrm{n}, \mathrm{fact}_{def}\ (-\ (n, 1);\ \mathbf{self}))\\ \mathbf{fun}\,\mathrm{main}\ (-;\ \langle\rangle) =\\ \quad \mathbf{let}\,\mathrm{fact} = \langle \mathrm{fact}_{def}\rangle\ \mathbf{in}\\ \quad\quad \mathrm{fact}.0\ (5;\ \mathrm{fact})\end{array}$$

Because `fact` as no free variables, its closure consists solely of the code address (fact_{def}).

[1] Note that this choice is *sound* for data constructors, since they are pure values, but it is not be sound for functions in general.

[2] We present the translation from AST to Simple IR in a semi-formal style. If we were being more rigorous, we would have to account for the mapping of bound AST variables to Simple IR variables in the translation of subexpressions.

4.1 Local variables

Part of the translation from AST to Simple IR involves classifying variable occurrences into one of three kinds and determining their location: global variables whose location is specified as an index into the function's closure, the function parameter whose location is fixed, and local variables whose location is in the function's stack frame (see Section 8). A naïve approach to assigning local variable locations is to give each variable its own slot, but this is wasteful of stack space. A better approach is to assign a variable x's slot based on the number of local variables that are in scope at the point of x's definition.

4.2 Curried definitions

Function definitions in **Flang** can be curried, which adds some complexity to the translation to the Simple IR. Conceptually, we can think of a curried definition[3]

```
fun f param1 param2 ··· paramn = exp;
```

as being syntactic sugar for the nested definition

```
fun f1 param1 = {
    fun f2 param2 = {
        ··· {
            fun fn paramn = exp;
            fn
        };
        ···
    };
    f2
};
```

When combined with the translation to the Simple IR, we will see parameters become part of the closures of the inner functions. For example, consider the following definition of a curried addition function:

```
···
fun add (a : Integer) (b : Integer) -> Integer = a + b;
    ···
```

When translated to Simple IR, this function produces two definitions; one for the `a` parameter and one for the `b` parameter.

$$\begin{aligned}
&\textbf{fun}\ \mathrm{add}_{\mathit{def2}}\ (b;\ \langle a\rangle) = +\ (a)b\\
&\textbf{fun}\ \mathrm{add}_{\mathit{def1}}\ (a;\ \langle\rangle) = \langle \mathrm{add}_{\mathit{def2}}, a\rangle\\
&\cdots\\
&\textbf{let}\ \mathrm{add} = \langle \mathrm{add}_{\mathit{def1}}\rangle\ \textbf{in}\\
&\qquad\cdots
\end{aligned}$$

Notice that the parameter a is in the inner function's closure.

4.3 Type arguments

Since there are no types in the Simple IR, type parameters in functions and type arguments can be erased. Note, however, that we must preserve the underlying function abstraction and application to

[3] We omit the types here to reduce clutter.

preserve the program semantics. For example, consider the following, somewhat contrived, **Flang** expression:

```
{ fun f (a : Unit) [b] -> Unit = print "hi\n";
  let g : [b] Unit = f Unit;
  g [Integer];
  g [Boolean]
}
```

The effect of executing this expression should be that the string `"hi\n"` is printed twice. Thus, the translation of this expression to Simple IR should produce something like

$$
\begin{array}{l}
\textbf{fun}\, \mathrm{f}_{def2}\,(-;\,\langle\rangle) = \mathrm{print}\,(\text{"hi\textbackslash n"});\\
\textbf{fun}\, \mathrm{f}_{def1}\,(a;\,\langle\rangle) = \langle \mathrm{f}_{def2}\rangle;\\
\textbf{fun}\, \mathrm{main}\,(-;\,\langle\rangle) =\\
\quad \textbf{let}\, \mathrm{f} = \langle \mathrm{f}_{def1}\rangle\, \textbf{in}\\
\quad \textbf{let}\, \mathrm{g} = \mathrm{f}.0\,(0;\,\mathrm{f})\, \textbf{in}\\
\quad \textbf{let}\, \mathrm{x} = \mathrm{g}.0\,(-;\,\mathrm{g})\, \textbf{in}\\
\qquad \mathrm{g}.0\,(-;\,\mathrm{g})
\end{array}
$$

4.4 Primitive functions

The AST representation does not distinguish between application of a primitive operator (*e.g.*, `+`) or Basis function (*e.g.*, `print`), and application of a user-defined function. But for code generation purposes, we need to handle these applications as special cases. To support this need, the Simple IR distinguishes between application of user-defined functions and primitive operators.

There is a further level of distinction that is made when the Simple IR is translated to VM bytecode. The following table summarizes the primitive operations and their mapping to VM instructions or run-time system functions:

`==`	⇒	**`equ`**	`argc`	⇒	`VM_argc`
`<=`	⇒	**`lesseq`**	`arg`	⇒	`VM_arg`
`<`	⇒	**`less`**	`fail`	⇒	`VM_fail`
`@`	⇒	`VM_concat`	`ignore`	⇒	*see below*
`+`	⇒	**`add`**	`neg`	⇒	**`neg`**
`-`	⇒	**`sub`**	`not`	⇒	**`not`**
`*`	⇒	**`mul`**	`print`	⇒	`VM_print`
`/`	⇒	**`div`**	`size`	⇒	**`size`**
`%`	⇒	**`mod`**	`sub`	⇒	**`subscript`**

As noted in the table, the `ignore` function should be treated specially. In general, an application "`ignore[`*ty*`](`*e*`)`" should be treated as the expression "`{` *e*`; Unit }`."

Basis functions may also appear as values. As with data-constructor-function values, we use η-conversion to wrap primitive operations as a regular function values. Lastly, the instructions that implement primitive operations (including calling run-time functions) expect multiple arguments, whereas **Flang** functions take one value argument (possibly a tuple) at a time. Thus your translation to Simple IR needs to analyze the argument of primitive applications and expand them into tuples as necessary.

5 Translating pattern matching

The Simple IR does not have data constructors or pattern matching, so one of the tasks in translating a program will be translating the AST match-case and pattern constructs into Simple IR code.

A **Flang** match case "`case` *exp* `of` *rules* `end`" is translated to Simple IR code that evaluates the argument of the case and binds it to a variable.

$$\textbf{let } x = \widehat{exp} \textbf{ in } \widehat{rules}$$

where $\widehat{exp}$ and $\widehat{rules}$ are the translations to Simple IR of the subcomponents. The simplest implementation of the rules is to test x against each pattern until a match is found. For a rule of the form `{`*pat* `=>` *exp*`}`, we generate code that is based on the syntax of the pattern pat. There are basically four kinds of patterns found in the AST.[4]

1. A variable pattern: `{`y `=>` *exp*`}`. Variable patterns are exhaustive and must occur as the last rule of the match case. We handle this case by translating exp in a context that maps the AST variable y to the Simple IR variable x.

2. A tuple pattern: `{(`$y_1, \ldots, y_n$`) =>` *exp*`}`. Tuple patterns are exhaustive and must occur as the last rule of the match case. We handle this case by creating Simple IR variables ($\widehat{y_i}$) corresponding to the AST variables (y_i) in the pattern and generating the code

   $$\textbf{let } \widehat{y_1} = x.0 \textbf{ in } \cdots \textbf{let } \widehat{y_n} = x.(n-1) \textbf{ in } \widehat{exp}$$

   Note that tuples are indexed from 0.

3. A nullary-data-constructor pattern: `{`C `=>` *exp*`}`. If this pattern is the last pattern in the match case, then it is exhaustive (recall that the type checker checks the exhaustiveness of match cases). In that situation, the resulting code is the translation of the right-hand-side expression. If the rule is not the last rule, then we need to test x against the representation of the constructor C.

4. A data-constructor pattern: `{`$C(y)$ `=>` *exp*`}`. If this pattern is the last pattern in the match case, then it is exhaustive (recall that the type checker checks the exhaustiveness of match cases). In that case, the Simple IR code extract the constructor's argument from the value and bind it to y in the translation of exp. If the rule is not the last rule, then we need to test to see if the argument matches the constructor and, if so, extract the constructor's argument from the value and bind it to y in the translation of exp.

When the type of the match-case argument has mixed representation (Section 3), we need to test if the representation is boxed or unboxed before checking constructor tag values. We assume the existence of a primitive operator **isBoxed** that returns true for heap-allocated values (*i.e.*, tuples and strings) and false for immediate values.

To illustrate how the pattern matching translation works, we present a few simple examples. The first is a match case that tests for the empty list:

```
case f Unit of { _::_ => False } { Nil[a] => True } end
```

[4] We regard the list cons (`::`) patterns as a special case of the data-constructor pattern.

Recall that the `List` type has a mixed representation. We can test for list cons by testing if the argument is boxed. The Simple IR implementation of this example is as follows:

$$\textbf{let}\ x = f.0\,(0;\ f)\ \textbf{in if isBoxed}\,(x)\ \textbf{then}\,0\,\textbf{else}\,1$$

Note that the nullary-constructors `Unit` and `False` have been mapped to 0 and `True` has been mapped to 1 in the resulting code.

For the second example, we consider a data type definition with three nullary constructors:

```
data Color with con Red con Green con Blue;
```

and a match case that maps the constructors to strings

```
case f Unit of { Red => "r" } { Green => "g" } { Blue => "b" } end
```

In this example, the `Color` type has an unboxed representation, so we just test against the constructor tag values.

$$\begin{array}{l}\textbf{let}\ x = f.0\,(0;\ f)\ \textbf{in}\\ \quad\textbf{if}\ ==\ (x,0)\ \textbf{then}\ \text{"}r\text{"}\ \textbf{else if}\ ==\ (x,1)\ \textbf{then}\ \text{"}g\text{"}\ \textbf{else}\ \text{"}b\text{"}\end{array}$$

Note that we do not have to test against the tag for `Blue` (*i.e.*, 2), since that is the last rule in the match case and we know that the argument must be `Blue`.

The final example involves a more complicated data type for representing arithmetic expressions:

```
data Exp with
  con Add of Exp * Exp
  con Mul of Exp * Exp
  con Neg of Exp
  con Num of Integer
```

The representation of the constructors is as follows: `Add(`v`)` is represented by $\langle 0, v\rangle$, `Mul(`v`)` is represented by $\langle 1, v\rangle$, `Neg(`v`)` is represented by $\langle 2, v\rangle$, and `Num(`n`)` is represented by $\langle 3, n\rangle$. With the representation determined, consider the following function for evaluating expressions:

```
fun eval (e : Exp) -> Integer =
    case e of
    { Add p => case p of { (e1, e2) => eval e1 + eval e2 } }
    { Mul p => case p of { (e1, e2) => eval e1 * eval e2 } }
    { Neg e => neg (eval e) }
    { Num n => n }
    end
```

Translating this function yields the following Simple IR code:

$$
\begin{array}{l}
\textbf{fun}\,\mathrm{eval}_{\mathit{def}}\,(\mathrm{e};\,\langle\rangle) = \\
\quad \textbf{if}\ \ == \ (\mathrm{e}.0, 0)\ \textbf{then} \\
\quad\quad \textbf{let}\,\mathrm{p} = \mathrm{e}.1\,\textbf{in} \\
\quad\quad \textbf{let}\,\mathrm{e1} = \mathrm{p}.0\,\textbf{in} \\
\quad\quad \textbf{let}\,\mathrm{e2} = \mathrm{p}.1\,\textbf{in} \\
\quad\quad\quad +\,(\mathrm{eval}_{\mathit{def}}\,(\mathrm{e1};\,\textbf{self}), \mathrm{eval}_{\mathit{def}}\,(\mathrm{e2};\,\textbf{self})) \\
\quad \textbf{else if}\ \ == \ (\mathrm{e}.0, 1)\ \textbf{then} \\
\quad\quad \textbf{let}\,\mathrm{p} = \mathrm{e}.1\,\textbf{in} \\
\quad\quad \textbf{let}\,\mathrm{e1} = \mathrm{p}.0\,\textbf{in} \\
\quad\quad \textbf{let}\,\mathrm{e2} = \mathrm{p}.1\,\textbf{in} \\
\quad\quad\quad *\,(\mathrm{eval}_{\mathit{def}}\,(\mathrm{e1};\,\textbf{self}), \mathrm{eval}_{\mathit{def}}\,(\mathrm{e2};\,\textbf{self})) \\
\quad \textbf{else if}\ \ == \ (\mathrm{e}.0, 2)\ \textbf{then} \\
\quad\quad \textbf{let}\,\mathrm{e} = \mathrm{e}.1\,\textbf{in} \\
\quad\quad\quad \textbf{neg}\,(\mathrm{eval}_{\mathit{def}}\,(\mathrm{e};\,\textbf{self})) \\
\quad\quad \textbf{else let}\,\mathrm{n} = \mathrm{e}.1\,\textbf{in}\,\mathrm{n}
\end{array}
$$

Notice that since `eval` is self-recursive, we use the **self** expression form for the closure on recursive calls.

6 Hints for Part 1

The first part of this project involves orchestrating a number of different analyses and transformations. A key to implementing this code successfully is keeping your code modular; do not try to do every thing in one monolithic pass. We recommend structuring your code as follows:

- An AST to AST pass that simplifies the AST by converting curried definitions to nested definitions (see Section 4.2), η expanding where necessary (see Sections 3.1, 3.2, and 4.4), and expanding applications of the built-in function `ignore` (see Section 4.4).
- An analysis pass over the simplified AST that computes the free variables of every function definition. You can represent the result of this information using a hash table that maps functions to sets of their free variables.
- An analysis pass over the simplified AST that computes the representations of data constructors. You can represent the result of this information using a hash table that maps data constructors to a description of their representation.
- The translation pass that converts the simplified AST to your Simple IR representation. This pass must handle the translation of functions and data constructors, identify primitive operations, and expand pattern matching. Because of simplification, however, it has fewer special cases to handle (*e.g.*, primitive operators will always be in application contexts).

In addition to these components, you will need to design a representation of the Simple IR. Your representation does not need to slavishly follow the abstract syntax given in Figure 1, but rather should be designed to make code generation straightforward. For example, the important information about variable occurrences in the Simple IR is where they are bound (global, parameter,

or local) and their offset. You may also want to include the variable name for debugging purposes, but one approach to a Simple IR expression representation might have three different constructors for variables:

```
datatype exp
  = GlobalVarExp of int  (* global-variable reference *)
  | ParameterExp         (* function parameter *)
  | LocalVarExp of int   (* local-variable reference *)
  | ...
```

Another design decision will be how to represent primitive operations in the Simple IR.

7 The virtual machine

The second phase of this project is to translate the Simple IR to virtual-machine bytecode. The virtual machine is a stand-alone program that takes an executable file and runs it. A VM executable consists of a code sequence, a literal table that contains string literals, and a C function table that contains runtime system functions used to implement services such as I/O. The details of this machine are described in a separate document, but we give a brief overview here.

The VM is a 32-bit machine that supports three types of values: 31-bit tagged integers, 32-bit pointers to heap-allocated records of values, and 32-bit pointers to strings. A integer value n is represented by $2n + 1$ in the VM (this tagging is required for the garbage collector). The VM takes care of tagging/untagging, so the only impact of this representation on your code generator is that integer literals must be in the range -2^{30} to $2^{30} - 1$.

The VM has four special registers: the stack pointer (SP), which points to the current top of the stack; the frame pointer (FP), which points to the base of the current stack frame and is used to access function parameters and local variables; the environment pointer (EP), which points to the current function's closure and is used to access global variables; and the program counter (PC), which points to the next instruction to execute.

8 Calling convention

The VM has a number of special operations and registers designed to support higher-order functions. As discussed above, a **Flang** function application "$e_1\ e_2$" is translated to the Simple IR expression

$$\textbf{let } f = \widehat{e_1} \textbf{ in } f.0\,(\widehat{e_2};\ f)$$

which fixes the order of evaluation (*i.e.*, e_1 is evaluated before e_2). We implement the Simple IR function call "$e\,(e_1,\ \ldots,\ e_n;\ e')$" using a four-part protocol (or *calling convention*):

1. The first stage of the protocol is the call, which is executed by the caller. The caller evaluates the arguments $(e_1,\ \ldots,\ e_n)$ in order pushing the results on the stack (for **Flang**, functions only have zero or one arguments). Then the caller evaluates the function closure (e') and pushes it on the stack (the closure will always be a variable or **self**). Next the caller evaluates the function address (e) and pushes it on the stack. Finally, the caller executes a `call` instruction, which pops the function's address and transfers control to the function.

2. The second stage of the protocol is the function prologue, which is executed by the callee. The first instruction in the prologue is an `entry(`n`)` instruction, where n is the number of

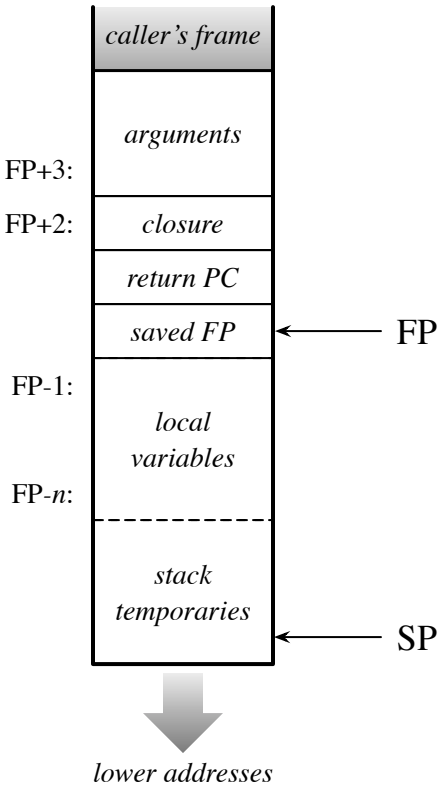

Figure 2: Stack-frame layout

local-variable slots required for the function. Executing this instruction pushes the caller's frame-pointer, sets the new frame pointer to point to the top of the stack, and then allocates space for local variables. The callee then pushes its closure on the stack using a `loadlocal(2)` instruction and sets the EP using `popep`. Figure 2 illustrates the layout of a function's stack frame after the prologue has been executed.

3. The third stage of the protocol is the function epilogue, which is executed by the callee after the function body has completed execution and the return result is on top of the stack. The callee uses a `storelocal(2)` instruction to store the result in the closure slot of the stack and then executes a `ret` instruction that deallocates the local variable space, restores the caller's frame pointer, and transfers control to the return address with the result on the top of the stack.

4. The final stage of the protocol is the function return, which is executed by the caller. Upon return, the top of the stack is the function's result and below that are the function arguments. The caller needs discard the arguments to the call, which are below the result, and restore its EP. The first step is done by calling `swap` to swap the argument with the function result, followed by a `pop` to discard the argument.[5] The caller then restores its EP by executing a `loadlocal(2)` followed by a `popep` instruction.

[5]Remember that type functions have *no* arguments, so these instructions are only required for value functions.

$$
\begin{array}{rcl}
\mathcal{E}[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!] & = & \mathcal{E}[\![e_1]\!];\ \texttt{storelocal}(x);\ \mathcal{E}[\![e_2]\!] \\
\mathcal{E}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!] & = & \mathcal{E}[\![e_1]\!];\ \texttt{jmpif}(l);\ \mathcal{E}[\![e_3]\!];\ \texttt{jmp}(l');\ l{:}\,\mathcal{E}[\![e_2]\!];\ l'{:} \\
 & & \quad \text{where } l \text{ and } l' \text{ are fresh labels} \\
\mathcal{E}[\![f_{def}\ (e_1, \ldots, e_n; \textbf{self})]\!] & = & \mathcal{E}[\![e_1]\!];\ \cdots\ ;\ \mathcal{E}[\![e_n]\!];\ \texttt{label}(l_f);\ \texttt{call} \\
\mathcal{E}[\![e\ (e_1, \ldots, e_n; e')]\!] & = & \mathcal{E}[\![e_1]\!];\ \cdots\ ;\ \mathcal{E}[\![e_n]\!];\ \mathcal{E}[\![e']\!];\ \mathcal{E}[\![e]\!];\ \texttt{call}; \\
 & & \texttt{loadlocal}(2);\ \texttt{popep} \\
\mathcal{E}[\![p\ (e_1, \ldots, e_n)]\!] & = & \begin{cases} \mathcal{E}[\![e_1]\!];\ \cdots\ ;\ \mathcal{E}[\![e_n]\!];\ \mathit{INSTR}_p & \text{if } p \text{ is an operator} \\ \mathcal{E}[\![e_1]\!];\ \cdots\ ;\ \mathcal{E}[\![e_n]\!];\ \texttt{ccall}(p) & \text{if } p \text{ is a built-in function} \end{cases} \\
\mathcal{E}[\![\langle e_1, \ldots, e_n \rangle]\!] & = & \mathcal{E}[\![e_1]\!];\ \cdots\ ;\ \mathcal{E}[\![e_n]\!];\ \texttt{alloc}(n) \\
\mathcal{E}[\![e.i]\!] & = & \mathcal{E}[\![e]\!];\ \texttt{select}(i) \\
\mathcal{E}[\![f_{def}]\!] & = & \texttt{label}(l_f) \\
 & & \quad \text{where } l_f \text{ is the label for } f_{def} \\
\mathcal{E}[\![x]\!] & = & \begin{cases} \texttt{loadlocal}(x) & \text{if } x \text{ is a parameter or local} \\ \texttt{loadglobal}(x) & \text{if } x \text{ is a global} \end{cases} \\
\mathcal{E}[\![\textbf{self}]\!] & = & \text{cannot happen, since } \textbf{self} \text{ only occurs in applications} \\
\mathcal{E}[\![n]\!] & = & \texttt{int}(n) \\
\mathcal{E}[\![s]\!] & = & \texttt{literal}(i) \\
 & & \quad \text{where } i \text{ is the index of the string literal in the literal table}
\end{array}
$$

Figure 3: VM code generation for Simple IR expressions

There are several optimizations of this protocol for when the call is a tail call, or self-recursive call, or when the function does not have global variables.

9 Bytecode generation

The Simple IR is designed to translate easily to the VM bytecode. We define the basic translation of Simple IR expressions as a function

$$\mathcal{E}[\![\cdot]\!] : \textsc{Exp} \to \textsc{Code}$$

where CODE is a finite sequence of VM instructions and labels. We use ";" to represent code concatenation, l: to represent a label in the sequence (and omit the following ";"), and INSTR_p to represent the VM instruction corresponding to a primitive operator p (*e.g.*, `add` for `+` and `equ` for `==`). The definition of $\mathcal{E}$ is given in Figure 3 The translation is mostly self-explanatory, but a couple of cases are worth further remark. The caller-side parts of the calling convention are captured by the translation and there is a special case for self-recursive calls too. The translation of variables depends on the kind of variable (see Section 4.1).

For a function definition $\textbf{fun}\, f_{def}\, (\overline{y};\, \langle \overline{x} \rangle) = e$ with globals $\overline{x}$ and parameters $\overline{y}$, we wrap the translation $\mathcal{E}[\![e]\!]$ with the function prologue and epilogue, resulting in the code:

```
l_f:
    entry(n)
    loadlocal(2)
    popep
    E[[e]]
    storelocal(2)
    ret
```

where n is the number of local-variable slots required for execution.

9.1 Bootstrapping

The VM starts executing the first instruction in the generated code. Sine the program should start by executing the `main` function, we need to emit a short sequence of code at the beginning of the code stream (see Section 10.1) to call the `main` function and then halt the VM upon return. The bootstrap code is

```
label(main)      // push the label main
alloc(1)         // allocate main's closure
label(main)      // push the label main
call             // call main
halt             // halt on return
```

9.2 Improvements

For extra credit, there are a couple of improvements you can make to the code generation process.

1. Function calls that are in tail position can be implemented using the **`tailcall`** instruction (instead of a **`call`** followed by a **`ret`**). Furthermore, self-recursive calls can be implemented by storing the argument in the function's parameter slot and jumping to the code following the function's prelude.

2. As discussed in lecture, you can eliminate jumps to jumps as part of the code generation process.

9.3 An example

To illustrate the code generation process, we revisit the example from Section 2, which translated to the following Simple IR:

$$
\begin{array}{l}
\textbf{fun}\, \mathrm{inc}_{def}\, (\mathrm{i};\, \langle \mathrm{n} \rangle) = +\, (\mathrm{i}, \mathrm{n}); \\
\textbf{fun}\, \mathrm{main}\, (-;\, \langle\rangle) = \\
\quad \textbf{let}\, \mathrm{n} = 1\, \textbf{in} \\
\quad \textbf{let}\, \mathrm{inc} = \langle \mathrm{inc}_{def}, \mathrm{n} \rangle\, \textbf{in} \\
\qquad \mathrm{inc}.0\, (2;\, \mathrm{inc});
\end{array}
$$

The VM code for this example is given in Figure 4 (including the bootstrap code).

```
        label(main)     // push the label main
        alloc(1)        // allocate main's closure
        label(main)     // push the label main
        call            // call main
        halt            // halt on return

inc:    entry(0)        // inc has no locals
        loadlocal(2)    // push the closure
        popep           // set the EP
        loadlocal(3)    // push the argument i
        loadglobal(1)   // push the global n
        add
        storelocal(2)   // save result in closure slot
        ret

main:   entry(2)        // main has two locals: n and inc
        loadlocal(2)    // push the closure
        popep           // set the EP
        int(1)
        storelocal(-1)  // store value of local n
        label(inc)      // push the label inc
        loadlocal(-1)   // push the local n
        alloc(2)        // allocate inc's closure
        storelocal(-2)  // store value of local inc
        int(2)
        loadlocal(-2)   // push inc's closure
        loadlocal(-2)   // push inc's closure
        select(0)       // pop closure and push code address
        call            // apply inc(2)
        swap            // swap result and argument
        pop             // discard argument
        storelocal(2)   // save result in closure slot
        ret
```

Figure 4: VM code for the example from Section 2

10 The code generation API

The code generation API is organized into three user-facing modules. The `Emit` module implements code streams, which are an abstraction of the generated output file, the `Labels` module implements labels for naming code locations, and the `Instructions` module implements an abstract type of VM instructions. Each of these modules is described below.

10.1 Code streams

A code stream provides a container to collect the instructions emitted by your code generator. You create a code stream from a file name and once code generation is complete, you invoke the `finish` operation, which does an assembly pass and then writes the binary object file to disk. The `Emit` module also provides hooks for registering string literals and the runtime functions used in Section 4.4.

10.2 Labels

The `Labels` module defines an abstract type of label that is used to represent code locations. The `Emit` structure provides the `defineLabel` function for associating a label with the current position in the code stream, and the control-flow instructions take labels as arguments. There is also an instruction for pushing the value of a label on the stack, which is required to create function closures.

10.3 Instructions

The `Instructions` module provides an abstract type that represents VM instructions. For those instructions that take arguments, it provides constructor functions and for those without arguments, it provides abstract values.

11 Submission

We will create a `prog4` director in your `phoenixforge` repositories and seed it with with a sample implementation of the type checker and the code generation API. Your project is due on Monday, March 16 at 10pm, so make sure that you have committed your final version before then.

12 Document history

March 16 Corrected description of function-call epilogue to include the discarding of function arguments. Also corrected Figure 4.

March 3 Added a bit more discussion about handling primitive operators.

March 2 Changed the data-construction-representation table to add a row for when there are no nullary constructors and multiple constructor functions.

March 1 Original version.