1 Introduction

The final project has two parts. The first is *closure conversion*, which lowers the higher-order Simple AST IR to a first-order control-flow-graph (CFG) representation. The second part is to generate LLVM assembly code from the CFG IR. This document largely focuses on the first part, which is the more substantial; the details of LLVM code generation are covered in a separate document.

2 Sample code

We will seed your repositories with sample code in a directory called `proj4`. The sample code includes a solution to Project 3, as well a library to support the generation of LLVM assembly. This code is incomplete, but does compile. You should read this code carefully (especially the signatures, which are well documented), so as to avoid reimplementing mechanisms that are already provided. There are five directories that are of particular importance to the project.

- The `cfg` directory contains the definitions for the first-order CFG representation of `LangF` programs. You will not have to make any modifications to the code in this directory, but you will need to understand the data structures and operations that are provided.

- The `llvm` directory contains the modules that implement support for generating LLVM assembly code. You will not have to make any modifications to the code in this directory, but you will need to understand the data structures and operations that are provided.

- The `runtime` directory contains a simple runtime system to support execution of `LangF` programs. The runtime system, which is written in C, includes implementation of the various runtime system functions used to implement the `LangF` Basis, functions for allocating heap objects, and a simple semi-space garbage collector.

- The `closure` directory contains the modules that implement closure conversion and the translation to the first-order CFG IR. You will need to be complete this code; look for expressions of the form `raise Fail "YOUR CODE HERE"`.

- The `codegen` directory contains the modules that implement the code generator, which translates the CFG IR to LLVM assembly code. You will need to be complete this code; look for expressions of the form `raise Fail "YOUR CODE HERE"`. 
Figure 1: The CFG Intermediate Representation

You will notice that there have been a few changes to the SimpleAST representation used in Project 3. First, we have restricted the let bindings to bind just a single variable and, likewise, return expressions to return just a single value. The other change is that we have added a wrapper around expressions to associate a program point with each expression (much the way that the AST IR associates type information with each expression).

```
datatype exp = Exp of ProgPt.t * exp_rep
and exp_rep
  = LetExp of var * rhs * exp
    ...
```

Program points provide a mechanism to associate analysis information with specific expressions.

## 3 The CFG IR

The main IR for this project is called the CFG IR, which is a transitional IR to help bridge the gap between the higher-order SimpleAST IR and LLVM assembly code. Figure 1 gives an abstract syntax for the CFG IR, which we will use in examples. In this syntax, we use \( f \) to represent code labels and \( x \) to represent variables. The CFG IR represents the program as a list of top-level functions, where each function consists of one or more fragments. The first fragment in a function is the entry fragment and the first function in the program is the entry function (discussed below.

\(^1\)The more general forms would be necessary to support certain optimizations, such as arity raising, but since we do not have any of those, we have reverted to the simpler form.
in Section 4.5). Functions and fragment names are represented as *labels* in the CFG IR, which are distinct from variables. The scope of a function label is the whole program, while the scope of a non-entry fragment’s label is the function that it is defined in. A fragment consists of a sequence of computational steps (i.e., *let* bindings) terminated by a control transfer (tail *apply*, *if*, *ret*, or *goto*). Each fragment defines the limits of the scope of the variables that are bound in it (i.e., a variable’s scope extends from its binding to the end the fragment). Fragments are essentially basic blocks with parameters.

As an example of the translation to CFG, consider the recursive factorial function in *LangF*.

```plaintext
fun fact (n : Int) -> Int = if (n <= 1) then 1 else n * fact(n - 1);
```

The SimpleAST version of this function looks something like

```plaintext
fun fact (n) =
  let t1 = IntLte(n, 1) in
  if t1
    then ret 1
  else let t2 = IntSub(n, 1) in
    let t3 = fact (t2) in
    let t4 = IntMul(n, t3) in
    ret t4
```

The CFG representation of this function breaks the control regions into separate fragments and makes explicit which variables are live across control transfers (i.e., jumps).

```plaintext
fun fact_lab (fact_clos, n) {
  let t1 = IntLte(n, 1) in
  if t1
     then goto frag1() else goto frag2(fact_clos, n)
}
and frag1 () {
  ret 1
}
and frag2 (fact_clos2, n1) {
  let t2 = IntSub(n1, 1) in
  let t3 = apply fact_lab (fact_clos2, t2) in
  let t4 = IntMul(n1, t3) in
  ret t4
}
```

For this example, the body of the function has been broken up into three fragments. The extra *fact_clos* parameter to the function is its closure (in this example, the closure is trivial), which is part of the CFG calling convention. Furthermore, notice that we have multiple CFG variables representing the SimpleAST variable *n* (we have used distinct names to emphasize this point).

The translation from SimpleAST to CFG involves two global transformations, plus a number of local transformations. The global transformations are

- *Closure conversion*, which is the process if converting higher-order functions to first-order functions by introducing closures.

- *Fragmentation*,[^2] which is the process of making explicit the local control flow inside functions

[^2]: The term “fragmentation” is not a standard term; we just coined it for this project; “closure conversion,” on the other hand, is a standard term.
(i.e., converting the SimpleAST expression that defines a function body into a collection of fragments).

In addition to these transformations, we also lower data constructors to their machine representation, replace `case` expressions with conditional tests, and add error checking for certain primitive operators.

### 4 Closure Conversion

The SimpleAST representation is a higher-order representation. By that, we mean that functions are first-class values that may be used as arguments, results, and embedded in data structures. Because a function value captures the values of its free variables at the time that it is defined, we need a representation for functions that supports this semantics. For this project, we use one of the simplest such representations — the flat closure — which is a heap-allocated tuple that includes the code address of the function’s machine code (in the first slot) and the values of any free variables in the function.\(^3\)

When translating to the CFG IR, every function is lifted to top level and an additional parameter is added for its closure value (as we saw in the fact example above). In the body of the function, references to free variables are replaced by selections from the closure tuple. At the previous sites of function definitions, we allocate the closure representation for the function, and at function application sites, we extract the code address of the function from its closure and then apply it to the function’s closure and other arguments. In the special case where a function calls itself, we can directly use the function’s label.

#### 4.1 An Example

Before getting into the details of how the closure conversion is implemented, let us consider another example of what the translation looks like. We start with the following LangF code fragment:

```lang
fun f (a : Int) (b : Int) -> Int = a + b;
fun g (h : Int -> Int) -> Int = h 42;
···
g (f 17)
···
```

The evaluation of the expression \( f \ 17 \) creates a function value that will add 17 to its argument. This function value is then passed to \( g \), which applies it to 42, resulting in 59.

After the transformation described above (Section 4.2), we have the following SimpleAST fragment:

```lang
fun f (a) =
  fun f2 (b) = let t1 = IntAdd(a, b) in ret t1
  in ret f2
in
fun g (h) = h 42 in
···
let t2 = f 17 in
let t3 = g t2 in
```

\(^3\)The sample code includes a variable analysis pass that both computes the free variables of functions needed to do closure conversion and the live variables of expressions needed to implement fragmentation as described in Section 5.
When we convert to the CFG IR, the various function definitions ($f$, $f_2$, and $g$) are all going to be lifted to top level and so they will have to access their free variables from their closures (in this example, only $f_2$ has a free variable, which is $a$). The resulting CFG functions are

```plaintext
fun f_lab (f_clos, a) {
  let f2 = alloc { f2_lab, a } // allocate closure for f2
  ret f2
}

fun f2_lab (f_clos2, b) {
  let a = #1 (f_clos2)
  let t1 = IntAdd (a, b)
  ret t1
}

fun g_lab (g_clos, h) {
  let h_cp = #0(h)
  apply h_cp(h, 42)
}
```

```plaintext
let f = alloc { f_lab } // allocate closure for f
let g = alloc { g_lab } // allocate closure for g
...
let f_cp = #0(f)
let t2 = apply f_cp (f, 17)
let g_cp = #0(g)
let t3 = apply g_cp (g, t2)
```

### 4.2 Simplifying Functions with Multiple Parameter Lists

The first step toward converting SimpleAST to CFG is to simplify function definitions. Recall that SimpleAST has function definitions with multiple parameters,\(^4\) but not applications to multiple lists of arguments. The sample code includes a rewrite pass (`NormalizeFuns.transform`) for the SimpleAST that replaces multiple-parameter function definitions of the form

```plaintext
fun f param_1 ... param_n = exp in exp'
```

where $n > 1$, with nested functions of the form:

```plaintext
fun f param_1 =
  fun f_2 param_2 =
    ...
  fun f_n param_n = exp in ret (f_n)
    ...
  in ret (f_2)
  in exp'
```

With this transformation, closure conversion only need handle the case of a function definition that has a single list of parameter variables.

\(^4\)This syntactic form was included to make the uncurrying optimization easier to implement
4.3 Converting function definitions

For each function \( f \) in the program, we assign an entry-fragment label \( f_{lab} \). During the translation to CFG, if we encounter a definition \( \text{fun } f \text{ param } = \text{exp} \), we replace it with the closure allocation

\[
\text{let } f = \text{alloc} \{ f_{lab}, x_1, \ldots, x_n \}
\]

where the \( x_i \) are free variables of \( \text{exp} \) (excluding \( f \) itself).

Furthermore, we generate a CFG function definition for the function by translating its body. In this translation, we need to replace references to the function’s free variables by references to elements of its closure. The easiest way to do so, is to introduce a sequence of bindings to local variables upon entry to the function. A more efficient scheme is to delay loading the variable from the closure until you know that it will be needed.

4.4 Converting function applications

In the general case, a non-tail application of the form \( \text{let } x = f (val_1, \ldots, val_n) \text{ in } \text{exp} \) is translated to code that first fetches the code address of the function from its closure and then calls the function.

\[
\begin{align*}
\text{let } &cp = \#0 (f) \\
\text{let } &x = \text{apply } cp (f, val_1, \ldots, val_n) \\
&\ldots
\end{align*}
\]

In the special case where \( f \) is the current function, we can use the function’s label directly, instead of having to load it from the closure.\(^5\)

\[
\begin{align*}
\text{let } &x = \text{apply } f_{lab} (f, val_1, \ldots, val_n) \\
&\ldots
\end{align*}
\]

Tail calls are handled with the same protocol as regular calls (note, however, that the CFG data structures distinguish between tail and non-tail calls). In the case of tail calls to the current function, however, we can further optimize the code by using a jump to the beginning of the function, instead of an application. In such a case, however, we actually want to define a loop-header fragment that follows the entry fragment. For example, consider the following SimpleAST code for the iterative factorial function:

\[
\begin{align*}
\text{fun } &ifact (i, acc) = \\
&\text{let } tl = \text{IntLte}(i, n) \text{ in} \\
&\text{if } tl \\
&\quad \text{then let } t2 = \text{IntAdd}(i, 1) \text{ in} \\
&\quad \text{let } t3 = \text{IntMul}(i, acc) \text{ in} \\
&\quad \text{ifact } (t2, t3) \\
&\quad \text{else } \text{ret } acc
\end{align*}
\]

(note that \( n \) is a free variable of this function). A possible translation to CFG is as follows, where we have defined an entry fragment that loads the free variables from the closure and a separate loop-header fragment.

\(^{5}\)In fact, we can apply this optimization whenever we call a function \( f \) in the scope of its \texttt{fun} binding.
fun ifact_lab (ifact_clos, i, acc) {
    let n = #1(ifact_clos)
    goto frag_hdr (n, i, acc)
}
and ifact_hdr (n1, i1, acc1) {
    let t1 = IntLte(i1, n1)
    if t1 then goto frag1 (n1, i1, acc1) else goto frag2 (acc1)
}
and frag1 (n2, i2, acc2) {
    let t2 = IntAdd(i2, 1)
    let t3 = IntMul(i2, acc2)
    goto ifact_hdr (n2, t2, t3)
}
and frag2 (acc3) { ret acc3 }

By separating the loop header from the entry fragment, we avoid redundant loads from the closure. The easiest way to implement this approach is to always split the entry fragment into an entry fragment that just loads the free variables and a header fragment that can be the target for any self-tail-recursive calls. The LLVM optimizer will get rid of any unnecessary jumps.

4.5 Converting programs

A program in the SimpleAST IR is just an expression. For the CFG IR, this representation must be encapsulated in a function, which will be the first function in the program (i.e., the entry function). The entry function does not need a closure and takes no arguments. We use the name "_langf_entry" for its label.

5 Fragmentation

The other major transformation of the SimpleAST is the use of fragments to explicitly represent both split and join points in the control flow.

5.1 Liveness Information

We say that a variable is live at a program point if there is a control-flow path from that point to a use occurrence of the variable. In general, liveness information is computed as a backward-flow analysis, but for a tree-based IR like SimpleAST, we can mostly define it inductively as the overloaded function

\[ \text{Live} \left[ \cdot \right] : \text{SIMPLEEXP} \rightarrow \mathbb{2}^{\text{SIMPLEVAR}} \rightarrow \mathbb{2}^{\text{SIMPLEVAR}} \]

that takes a SimpleAST expression and a set of variables that are live going out of the expression and returns the variables that are live going into the expression. We extend this definition to also include case rules.

\[ \text{Live} \left[ \cdot \right] : \text{SIMPLERULE} \rightarrow \mathbb{2}^{\text{SIMPLEVAR}} \rightarrow \mathbb{2}^{\text{SIMPLEVAR}} \]

The definition of these functions is given in Figure 2, where \( FV(f) \) denotes the free variables of \( f \) and \( \text{VarsOf} \) denotes the variables occurring in a value, sequence of values, or non-expression.
\[
\begin{align*}
\text{Live}\left[ \text{fun } f \text{ param }= \text{exp}_1 \text{ in } \text{exp}_2 \right] L &= \text{FV}(f) \cup \text{Live}\left[ \text{exp}_2 \right] \setminus \{f\} \\
\text{Live}\left[ \text{let } x = \text{exp}_1 \text{ in } \text{exp}_2 \right] L &= \text{Live}\left[ \text{exp}_1 \right] (\text{Live}\left[ \text{exp}_2 \right] L \setminus \{x\}) \\
\text{Live}\left[ \text{let } x = \text{rhs} \text{ in } \text{exp} \right] L &= \text{VarsOf}(\text{rhs}) \cup (\text{Live}\left[ \text{exp} \right] L \setminus \{x\}) \\
\text{Live}\left[ f (\text{val}) \right] L &= L \cup \{f\} \cup \text{VarsOf}(\text{val}) \\
\text{Live}\left[ \text{if } \text{val} \text{ then } \text{exp}_1 \text{ else } \text{exp}_2 \right] L &= \text{VarsOf}(\text{val}) \cup \text{Live}\left[ \text{exp}_1 \right] L \cup \text{Live}\left[ \text{exp}_2 \right] L \\
\text{Live}\left[ \text{case } \text{val} \text{ of } \text{rule} \right] L &= L \cup \text{VarsOf}(\text{val}) \cup \left( \bigcup_{\text{rule} \in \text{rule}} \text{Live}\left[ \text{rule} \right] L \right) \\
\text{Live}\left[ \text{ret}(\text{val}) \right] L &= L \cup \text{VarsOf}(\text{val}) \\
\text{Live}\left[ \{ \text{pat } \Rightarrow \text{exp} \} \right] L &= \text{Live}\left[ \text{exp} \right] L \setminus \text{VarsOf}(\text{pat})
\end{align*}
\]

Figure 2: Computing liveness for SimpleAST expressions

right-hand-side. For a function body or program expression \( \text{exp} \), we compute \( \text{Live}\left[ \text{exp} \right] \emptyset \). This liveness analysis is implemented as part of the variable analysis pass provided in the sample code. The analysis records the live-in set for every program point. Note that you will have to implement a simpler liveness analysis for the CFG fragments as part of your code generator.

5.2 Fragmenting Conditionals

Liveness information is necessary for the translation of splits and joins in the SimpleAST to fragments in the CFG IR. In the CFG, conditionals and case expressions no longer have sub-expressions are their arms; instead they have jumps to fragments. The variables that are live on entry to the fragment will be its parameters. There is an additional complication when the conditional (or case) expression is the right-hand-side of a \textit{let} binding. In this case, control is going to flow from the return position of the arms to the join point defined by the \textit{let} binding.\footnote{The discussion in Lecture 5.3 on conditionals in ANF is related to this aspect of the translation.}

The sample code defines the \textit{continuation} datatype to symbolically represent the “continuation” of an expression.

\[
\begin{align*}
\text{datatype continuation} \\
= \text{TAIL} \\
\mid \text{JOIN} \text{ of } \text{Env}.t * \text{CFG}.value \rightarrow \text{CFG}.jump
\end{align*}
\]

We use the constructor \text{TAIL} to represent the tail position of the current function and \text{JOIN} to represent a non-tail position, which will be the right-hand-side of a \textit{let} binding. The parameters of the join point will be the left-hand-side variable of the \textit{let} binding plus the live variables of the \textit{let}’s body, but we will only know how to translate the free variables to CFG values at the point where the control-transfer is generated. For this reason, the argument to the \text{JOIN} constructor is a function that takes the translation environment at the point of the control transfer, plus the result of the right-hand-side, and returns the jump.
5.3 An Example of Fragmentation

For example, consider the following SimpleAST function, where we have annotated each line with the set of variables that are live on entry of the expression:

```latex
fun f (b) =
  let x = 17 in // {b,g}
  let y = 42 in // {b,x,g}
  let p = if b // {b,y,x,g}
      then {x, y} // {y,x,g}
      else {y, x} // {y,x,g}
  let t1 = g (p) // {x,g,p}
  let t2 = IntMul(x, t1) // {x,t1}
  ret t2  // {t2}
```

The liveness information is used to determine the parameters of the fragments when translating to the CFG representation.

```latex
fun f_lab (clos, b) {
  let g = #1(clos)
  let x = 17
  let y = 42
  if b then goto frag1 (y, x, g) else goto frag2 (y, x, g)
}
```

```latex
and frag1 (y1, x1, g1) { // then branch
  let p1 = {x1, y1}
  goto frag3 (p1, x1, g1)
}
```

```latex
and frag2 (y2, x2, g2) { // else branch
  let p2 = {y2, x2}
  goto frag3 (p2, x2, g2)
}
```

```latex
and frag3 (p3, x3, g3) { // join for 'let p ...'
  let g_cp = #0(g3)
  let t1 = apply g_cp (g3, p3)
  let t2 = IntMul(x3, t1)
  ret t2
}
```

In this example, `frag3` is the join fragment for the `let` binding of the variable `p`; the variables `x` and `g` are free in the body of the `let` binding, so they are included in `frag3`'s parameter list.

The one complication that we have ignored in this discussion is an addition needed to translate self-recursive-tail calls (such as in the `ifact` example in Section 4.4). In this example, the free variable `n` is actually live at the tail-call site, since we need it to flow back to the header fragment. The implementation of liveness analysis provided in the sample code handles this detail.

6 Lowering Data Constructors and Pattern Matching

Recall from Project 3 that we defined concrete representations for the data types defined in the program. As part of the translation to the CFG IR, we replace data constructors with their machine representation.
6.1 Data Constructor Representations

Recall from Section 4.1 of the Project 3 handout that we determined a representation for nullary data constructors \( C \) and data-constructor functions \( F \) that could be one of four possible representations (these choices are represented by the \textit{SimpleDataCon.con_rep} type).

<table>
<thead>
<tr>
<th>SimpleAST</th>
<th>Constructor Representation</th>
<th>CFG Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C )</td>
<td>\text{Enum}(n)</td>
<td>( n )</td>
</tr>
<tr>
<td>( F \ v )</td>
<td>\text{Immediate}</td>
<td>( v )</td>
</tr>
<tr>
<td>( F \ v )</td>
<td>\text{Box}</td>
<td>\text{alloc}{v}</td>
</tr>
<tr>
<td>( F \ v )</td>
<td>\text{TaggedBox}(n)</td>
<td>\text{alloc}{n, v}</td>
</tr>
</tbody>
</table>

The nullary constructors of a datatype will always have the \text{Enum} representation while the data-constructor functions of a given datatype will always have the \textit{same} representation.

6.2 Pattern Matching

Pattern matching in the simple AST is only applied to data type values. The first step is to examine the kind of the case expression’s argument, which can be either unboxed, boxed, or mixed.

6.2.1 Cases on Unboxed Arguments

If the argument has an unboxed representation, then it is sufficient to generate a sequence of conditional tests against each of the possible values. If the case has \( k \) cases, then we will end up with \( k - 1 \) conditional tests.

For example, if we have a case rule \( \{ C \Rightarrow \text{exp} \} \), where the representation of \( C \) is \text{Enum}(n), then we get CFG code of the form

\[
\{ \\
\quad \ldots \\
\quad \text{let } t = \text{IntEq}(\text{arg}, n) \\
\quad \quad \text{if } t \text{ then goto frag1}(\overline{x}) \text{ else goto frag2}(\overline{y}) \\
\} \\
\quad \text{and frag1 } (\overline{x'}) \{ \\
\quad \quad \ldots \\
\} \\
\quad \text{and frag2 } (\overline{y'}) \{ \\
\quad \quad \ldots \\
\}
\]

where \text{frag1} is the translation of \( \text{exp} \) and \text{frag2} handles the rest of the rules in the case. Here the \( \overline{x} \) (and \( \overline{x'} \)) are the translation of \text{Live[exp]}, and the variables \( \overline{y} \) (and \( \overline{y'} \)) are the translation of the \text{union} of the live variables for the remaining cases.

For the last case in the rule, we do not have to do the conditional test, since we know that case expressions are exhaustive.
6.2.2 Cases on Boxed Arguments

Cases on boxed arguments depend on the representation of the constructors. For the Immediate representation, there will only be one rule in the case (since all rules must be useful) and it can be implemented by binding the pattern variable to the argument value. Likewise, there will be only one rule if the representation is Box and it can be implemented by binding the pattern variable to the selection of the contents of the box. When the constructor-function representation is a TaggedBox, then we have code that is similar to the unboxed case described above. We extract the tag value from the representation and do a series of conditional tests as is done for the unboxed scenario. The one addition is that we need to bind the pattern argument variable to the selection of the argument value’s data component.

6.2.3 Cases on Mixed Arguments

The most complicated scenario is when the type of the match-case argument has mixed representation. We handle this situation by partitioning the rules into those that cover unboxed constructors and those that cover boxed constructors. The primitive operator Boxed, which returns true for heap-allocated values (i.e., tuples and strings) and false for immediate values, is used to test the argument to see which case should handle it. We can use the same implementation techniques described above for each of the two sets of rules.

One subtlety in the mixed representation scenario is handling a variable rule (if present). This rule might apply to both boxed and unboxed constructors or just one of the sets of rules. In the sample code, we have annotated the SimpleAST expressions with the list of SimpleAST data constructors that are possible (see the file simple-ast/case-info.sml). Using this information, one can determine if the variable rule needs to be present in just one or both of the rule sets. In the situation where the variable rule covers both boxed and unboxed values, you should be careful to avoid duplicating the expression part of the rule (i.e., you should share the fragment(s) generated for the expression).

Let us consider an example of how this translation should work. We start with the following LangF datatype:

```
data T = A | B | C | F of Int | G of String

fun f (x : T) -> Int = case x of
  { A => 1 }
  { B => 2 }
  { F y => y }
  { _ => 42 }
end
```

7Actually, if the argument representation is boxed, then the Box representation is not possible, but we describe it here because it is the right place to do so.

8This primitive operator was added to the Prim.t datatype for this project.
The representations of the constructors are

\[
\begin{align*}
[A] &= \text{Enum}(0) \\
[B] &= \text{Enum}(1) \\
[C] &= \text{Enum}(2) \\
[F_n] &= \text{TaggedBox}(0) \\
[G_s] &= \text{TaggedBox}(1)
\end{align*}
\]

and the translation of \( f \) to CFG will be something like the following:

```plaintext
fun f_lab (clos, x) {
  let t1 = Boxed(x)
  if t1 then goto fragb0(x) else goto fragu0(x)
}
and fragb0 (x3) { // boxed scenario
  let t4 = #0(x3)
  let t5 = IntEq(t4, 0)
  if t5 then goto fragF(x3) else goto fragWild()
}
and fragu0 (x1) { // unboxed scenario
  let t2 = IntEq(x1, 0)
  if t2 then goto fragA() else goto fragu1()
}
and fragu1 (x2) { // unboxed scenario
  let t3 = IntEq(x2, 1)
  if t3 then goto fragB() else goto fragWild()
}
and fragA () { ret 1 }
and fragB () { ret 2 }
and fragF (x4) {
  let y = #1(x4)
  ret y
}
and fragWild () { ret 42 }
```

7 Translating Primitive Operators

[NOTE: this section has changed substantially from the original version!]

Some primitive operators (\( \text{IntDiv}, \text{IntRem}, \text{StrSub}, \) and \( \text{StrChr} \)) require error checking to ensure that their arguments are valid. To handle this requirement, we have added code to the simplification pass in the sample code that adds these checks. Thus, any application of these operators is guaranteed to only by on valid arguments.

Therefore, almost all of the primitive operators can be directly translated to CFG. The two exceptions are

- The \( \text{StrChr} \) primitive operator, which creates a one-character string from an ASCII code point, is translated to a runtime system function call.
- The \( \text{RefNew} \) primitive operator, which allocates a new reference cell, is translated to a one-word heap-object allocation containing its argument.
8 Extra Credit

Getting the baseline closure converter and code generator working is a non-trivial task, but if you have the time, there are number of improvements that you can include for extra credit. If you choose to implement any of these, please include a file named EXTRA in the root of your project describing what you did.

8.1 Uncurrying (20 points)

If you did not implement the uncurrying extra-credit optimization for Project 3, you may do so in this project.

8.2 Optimizing known function calls (5 points)

As described above, if you call a function in the scope of its fun binding, you can directly refer to its code label in the call, instead of loading it from the closure.

8.3 Lazy free-variable loading (10 points)

Instead of loading all of the free variables in the function’s entry fragment, delay loading them until they must been used. The goal is to minimize both the dynamic and static numbers of loads, with a bias toward reducing dynamic loads. The liveness information computed by the variable analysis can help with this optimization.

9 Hints

The project is structured in such a way that you can focus on one part at a time. It is recommended that you start with the translation from SimpleAST to CFG.

As with Projects 2 and 3, you will have to manage an environment that maps SimpleAST variables to CFG variables. You will also need some additional context information, such as tracking the current function and whether the expression that is currently being translated is in a join position.

Be careful about the management of live variables when translating to CFG. The hard work has been done by the variable analysis pass, but in certain situations you will need to merge live-variable sets, etc. Also, live variables have different CFG names in different fragments, so be careful to keep the renaming environments straight.

There are significant opportunities for sharing code between the different scenarios when translating case expressions to CFG.

10 Submission

We will collect the projects at 23:59 (Chicago time) on December 9, 2020 from the SVN repositories, so make sure that you have committed your final version before then. If you choose to
implement any of the extra credit tasks, remember to include the EXTRA file in the root of your project.

**Important note:** You are expected to submit code that **compiles** and that is well documented. Remember that points for project code are assigned 30% for coding style (documentation, choice of variable names, and program structure), and 70% for correctness. Code that does not compile will **not** receive any points for correctness.

### 11 Document history

**December 9, 2020** Fixed the example in Section 6.2.3; boxed test was backwards.

**December 4, 2020** Expanded the discussion of fragmentation, including a new subsection on conditionals. Also fixed the fragmentation example to pass the result to the join point as the first argument, with the free variables following, instead of the other way around.

**December 3, 2020** Fixed a small typo in Section 6.2.3.

**November 23, 2020** Simplified assignment by moving the introduction of error-checking for primitive operators to the simplification phase provided in the sample code.

**November 21, 2020** Original version.