1 Introduction

The third project has two parts. The first is to implement a simplifier that translates the AST produced by the type checker into a normalized intermediate representation (IR) that is more amenable to analysis and optimization. The second part is to implement some basic optimizations on this IR. You may implement additional optimizations for extra credit.

2 Sample code

We will seed your repositories with sample code in a directory called proj3. The sample code includes a solution to Project 2. This code is incomplete, but does compile. You should read this code carefully (especially the signatures, which are well documented), so that you understand the data structures and available interfaces. There are four new directories for the project:

- The `simple-ast` directory contains the definitions for a simplified Abstract Syntax Tree (SimpleAST) 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 `prim` directory contains definitions of the primitive operators, primitive types, and runtime-system functions. These definitions are used in the SimpleAST IR and in the IR for Project 4. 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 `simplify` directory contains the modules that implement the simplifier. Look for statements of the form “`raise Fail "YOUR CODE HERE"`.”

- The `optimize` directory contains the modules that implement the optimizer. Look for statements of the form “`raise Fail "YOUR CODE HERE"`.”

3 AST Normalization

Before translating to SimpleAST, we perform a normalization pass over the AST IR to eliminate a number of corner cases that would complicate the translation to SimpleAST. Specifically, we
eliminate incomplete applications of constructors and the use of Basis functions as arguments. For
example, the following \texttt{LangF} code

\begin{verbatim}
  data T[t] = A | B of t * t;
  let x : [t] T(t) = A;
  let y : Int * Int -> T[Int] = B[Int];
  fun f (g : String * Int -> Int) -> String = chr(g (s, 0));
  val z : String = f sub;
\end{verbatim}

is normalized to

\begin{verbatim}
  data T[t] = A | B of t * t;
  fun mkB [t] (x : t * t) -> T[t] = B [t] (x);
  fun sub' (arg : String * Int) -> Int = case arg of
       { (a, b) => sub(a, b) }
   end;
  let x : [t] T(t) = mkA;
  let y : Int * Int -> T[Int] = mkB[Int];
  fun f (g : Int -> String) -> String = g 42;
  val z : String = f sub';
\end{verbatim}

Note that the \texttt{mkA} function would not be legal in a \texttt{LangF} source program, since functions must
have value parameters, but we can represent it in the AST.

The AST that results from this normalization pass is guaranteed to supply all arguments to any
occurrence of data constructors or basis functions in an expression.

\section{SimpleAST}

The \textit{SimpleAST} representation is an example of a \textit{normalized} representation (specifically, it is a
variant of Monadic-Normal Form). In addition to simplifying and lowering\textsuperscript{1} the AST, the Sim-
pleAST IR has the property that all intermediate values are given a name and all arguments are
either variables or constants. For example, the \texttt{LangF} function

\begin{verbatim}
  fun f (a : Int) (b : Int) (c : Int) -> Int = a * b + c
\end{verbatim}

will be simplified to

\begin{verbatim}
  fun f (a) (b) (c) =
       let t1 = IntMul(a, b) in
       let t2 = IntAdd(t1, c) in
       ret t2
\end{verbatim}

where \texttt{IntMul} and \texttt{IntAdd} are primitive operators.

Another major difference between the SimpleAST IR and AST is that all type abstractions and
applications are removed. In the case of data constructors, these are erased, while in the case of
functions and type applications, they are converted to zero-argument functions and applications.
For example, the \texttt{LangF} function

\begin{verbatim}
  fun id [t] (x : t) -> t = x;
  let y : Int = id [Int] 42
\end{verbatim}

\textsuperscript{1}By “lowering,” we mean translating to lower-level mechanisms.
will be simplified to

```plaintext
fun id () (x) = ret x;
let y = let t = id () in t 42
```

The syntax of SimpleAST is given in Figure 1. For this syntax, we introduce the convention of using $C$ to denote nullary data constructors and $F$ to denote data-constructor functions. We also use a blue color scheme to distinguish it from the AST terms when we are defining the translation. We use $p$ to denote primitive operators and $cf$ to denote runtime-system functions.

Programs in SimpleAST are just expressions (the structure of this expression is described in Section 5.2 below). Expressions consist of function bindings, let bindings, and several tail-expression forms (application, conditionals, cases, and returns).\footnote{Note that the use of the term “return” here just means that the result of the expression is the sequence of values; it only corresponds to a function return when it is in the tail position of a function body.} Notice that all arguments are values, which can be variables, nullary data constructors, or literals (integers and strings). The right-hand-side of a let binding can be an expression or one of several special forms: tuple creation, selection from a tuple, invocation of a primitive operator, a call to a runtime-system function, or application of a data constructor. The patterns in case rules are restricted to data constructors (or variables); tuple patterns are converted to tuple selection operators by the translation.

The SimpleAST representation has several important advantages over the AST IR:

1. The control-flow (i.e., order of evaluation) of the program is explicit.
2. Inlining (a.k.a., β-reduction) is semantically sound.

3. Simple use-def relations are directly represented in the IR.

4.1 SimpleAST Types

Unlike LangF and the AST IR, the SimpleAST IR is not strongly typed. We do, however, track the representation types of variables using a stripped down type system, which we call primitive types.  

\[
\begin{align*}
\text{ty} & ::= \text{Any} & & \text{An unknown type (the “top” of the type hierarchy)} \\
            & | \text{Obj} & & \text{A pointer to an object} \\
            & | \text{Int} & & \text{A tagged integer value} \\
            & | \text{String} & & \text{A string value value} \\
            & | \text{Ref} & & \text{A reference value} \\
            & | \{ty_1, \ldots, ty_n\} & & \text{A tuple type \( (n \geq 1) \)} \\
            & | (ty_1, \ldots, ty_n) \rightarrow ty' & & \text{A function type \( (n \geq 0) \)}
\end{align*}
\]

Runtime values in the LangF implementation have a uniform representation as 64-bit machine words. In order that the garbage collector can correctly identify pointers, we represent integer values as tagged 63-bit values (i.e., the integer value \( n \) is represented as \( 2^n + 1 \)). We use the Int as the type of these tagged integers.

Primitive types are monomorphic and do not support recursive types, but we have the Obj and Any types to represent polymorphism and recursion in the translation of types from AST.

\[
\begin{align*}
T[\alpha] & = \text{Any} \\
T[\tau' \rightarrow \tau] & = (T[\tau']) \rightarrow T[\tau] \\
T[\tau_1 \ast \cdots \ast \tau_n] & = \{T[\tau_1], \ldots, T[\tau_n]\} \\
T[\theta^{(k)}[\tau]] & = \text{the representation type of} \theta^{(k)}
\end{align*}
\]

We also define the kind of a type to be

- **Unboxed** – all values are represented by tagged integers.
- **Boxed** – all values are represented by pointers.
- **Mixed** – has both unboxed and boxed values.

The kind of Any is mixed and the kind of Int is unboxed; all other primitive types have boxed kind.

The representation of type and data constructors depends on an analysis that is implemented in the SimplifyType structure that is provided in the sample code. Let \( \theta^{(k)} \) be a datatype with \( n \) nullary constructors \( (C_1, \ldots, C_n) \) and \( k \) data-constructor functions \( F_1 \circ \tau_1, \ldots, F_k \circ \tau_k \). Then the following table gives the representation of the various constructors based on the number of constructors and the representations of the \( \tau \)'s.

\[\text{This type system is shared by the IR used for Project 4.}\]
<table>
<thead>
<tr>
<th>$n$</th>
<th>$k$</th>
<th>$C_i$</th>
<th>$F_j,v$</th>
<th>$\theta^{(k)}$'s representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&gt;0$</td>
<td>$0$</td>
<td>$i-1$</td>
<td>n.a.</td>
<td>Int</td>
</tr>
<tr>
<td>$0$</td>
<td>$1$</td>
<td>n.a.</td>
<td>$v$</td>
<td>$\tau_1$</td>
</tr>
<tr>
<td>$&gt;0$</td>
<td>$1$</td>
<td>$i-1$</td>
<td>$v$ if $\tau_j$ is boxed.</td>
<td>Any</td>
</tr>
<tr>
<td>$&gt;0$</td>
<td>$1$</td>
<td>$i-1$</td>
<td>${v}$ if $\tau_j$ is unboxed or mixed.</td>
<td>Any</td>
</tr>
<tr>
<td>$\geq0$</td>
<td>$&gt;1$</td>
<td>$i-1$</td>
<td>${j-1, v}$</td>
<td>Any</td>
</tr>
</tbody>
</table>

In this chart, $i$ means the immediate value $i$ and $\{\cdots\}$ means heap-allocated tuple. Applying this algorithm to the the builtin datatypes we get:

- False $\rightarrow 0$
- True $\rightarrow 1$
- Nil $\rightarrow 0$
- $a::b \rightarrow \{a, b\}$

### 4.2 Basis Functions and Operators

The LangF operators and basis functions are represented as variables in the AST IR. When we translate to SimpleAST, these operations are mapped to either applications of primitive operators ($p$) or calls to runtime-system functions ($cf$). The SimpleBasis structure provides a lookup function to support this translation. Note that because of the AST normalization described in Section 3, primitive functions will always occur as the left-hand-side of an application (possibly with an intervening type application).

### 5 Simplification

The process of simplification of the AST involves several aspects:

1. Restructuring the code so that every intermediate value has a name.
2. Replacing type abstractions and applications with value abstractions and applications.
3. Converting data-constructor functions that are not applied into functions.
4. Converting variables that denote primitive operators to primops.
5. Determining the representation of data constructors.

#### 5.1 The Translation

Simplification is applied to the AST IR, which is described in Figure 2. We use a red color scheme for the AST IR to distinguish it from the SimpleAST. We define simplification using a collection of
prog ::= dcl₁ ⋯ dclₙ  

dcl ::= data θ(k)  
| fun f param = exp  
| let x = exp  

param ::= [α₁, ..., αₙ]  
| (x)  

exp ::= if exp₁ then exp₂ else exp₃  
| exp₁ exp₂  
| exp [τ]  
| (exp₁, ..., expₙ)  
| dcl in exp  
| case exp of rule  
| x  
| C  
| F  
| lit  

rule ::= { pat => exp }  

pat ::= F x  
| (x₁, ..., xₙ)  
| C  
| x  

Figure 2: The AST intermediate representation produced by the type checker

translation functions:

\[
\begin{align*}
\mathcal{E} : & \quad \text{EXP} \rightarrow \text{EXP} \\
\mathcal{V} : & \quad \text{EXP} \rightarrow (\text{VAL} \rightarrow \text{EXP}) \rightarrow \text{EXP} \\
\mathcal{R} : & \quad \text{EXP} \rightarrow (\text{RHS} \rightarrow \text{EXP}) \rightarrow \text{EXP} \\
\mathcal{C} : & \quad \text{PAT} \times \text{EXP} \rightarrow \text{PAT} \times \text{EXP} \\
\mathcal{P} : & \quad \text{PARAM} \rightarrow \text{PARAM} \\
\mathcal{B} : & \quad \text{VAR} \rightarrow \text{PRIM} + \text{RTFUN} \\
\end{align*}
\]

where

\[
\begin{align*}
\text{exp} & \in \text{EXP} \\
\text{pat} & \in \text{PAT} \\
\text{rule} & \in \text{PAT} \times \text{EXP} \\
\text{param} & \in \text{PARAM} \\
\text{exp} & \in \text{EXP} \\
\text{val} & \in \text{VAL} \\
\text{rhs} & \in \text{RHS} \\
\text{pat} & \in \text{PAT} \\
\text{param} & \in \text{PARAM} \\
p & \in \text{PRIM} \\
f & \in \text{RTFUN} \\
\end{align*}
\]
Several of the translation functions take a functional representation of the context (i.e., $V$ and $R$).\footnote{This is an example of the continuation-passing style idiom that we saw in the lecture about interpreting regular expressions.} In the definition of the translation below, we supply a meta-level $\lambda$-abstraction (a.k.a., function) as the context. One should read the notation

$$V[exp] \lambda v(\cdots v \cdots)$$

as translate the AST expression $exp$ to a SimpleAST value and then build the SimpleAST expression "\cdots v \cdots", where $v$ is that value that represents $exp$. In your implementation of the translation, you can use SML functions to represent the meta-level functions.

5.1.1 The $E$ translation

The $E$ translation converts AST expressions to SimpleAST expressions. For certain function applications, the translation is conditioned on the status of the function variable being applied.

We start with rules for application of Basis operators and functions. Because of the AST normalization described in Section 3, all occurrences of Basis operators and functions are covered by these rules. We have two rules, depending on if the AST variable is polymorphic (for the polymorphic case, we erase the type application); the details of the translation are handled by the $R$ function.

$$f \in \text{dom}(B) \quad x \text{ is fresh}$$

$$E[ f [\tau] (exp_1, \ldots, exp_n) ] = R[ f [\tau] (exp_1, \ldots, exp_n) ] \lambda \text{rhs}(\text{let } x = \text{rhs in ret } (x))$$

$$f \in \text{dom}(B) \quad x \text{ is fresh}$$

$$E[ f (exp_1, \ldots, exp_n) ] = R[ f (exp_1, \ldots, exp_n) ] \lambda \text{rhs}(\text{let } x = \text{rhs in ret } (x))$$

The remaining cases are unconditional. For if-then-else expressions, we translate the argument expression to a value and the arms of the conditionals to expressions.

$$E[\text{if } exp_1 \text{ then } exp_2 \text{ else } exp_3] = V[exp_1] \lambda v(\text{if } v \text{ then } E[exp_2] \text{ else } E[exp_3])$$

We handle the case of data-constructor applications specially. For polymorphic constructor-function applications, we erase the type and translate the argument to a value.

$$E[ F [\tau] exp ] = V[exp] \lambda v(\text{let } x = Fv \text{ in ret } (x)) \quad x \text{ is fresh}$$

While for monomorphic constructor functions we just translate the argument to a value and apply the construct.

$$E[ F exp ] = V[exp] \lambda v(\text{let } x = Fv \text{ in ret } (x)) \quad x \text{ is fresh}$$

For the general case, we translate the function and argument to values and then build an SimpleAST application expression.

$$E[exp_1 exp_2] = V[exp_1] \lambda f(\text{let } x = Fv \text{ in ret } (f(x)))$$

As with applications, type applications have a special rule for handling polymorphic nullary data constructors, where we just erase the type application.

$$E[C [\tau]] = \text{ret } (C)$$
Other type applications are converted to applications to an empty argument list.

\[ \mathcal{E}[\text{exp } \tau] = \mathcal{V}[\text{exp}] \lambda f(f()) \]

The result of a tuple expression must be bound to a variable:

\[ \mathcal{E}[(\text{exp}_1, \ldots, \text{exp}_n)] = \mathcal{R}[(\text{exp}_1, \ldots, \text{exp}_n)] \lambda \text{rhs}(\text{let } x = \text{rhs in } \text{ret}(x)) \quad x \text{ is fresh} \]

The simplification of function bindings involves translating the parameters, function body, and its scope.

\[ \mathcal{E}[\text{fun } f \text{ param}_1 \cdots \text{param}_n = \text{exp in } \text{exp}'] =\]
\[ \begin{align*}
\text{fun } f \mathcal{P}[\text{param}_1] \cdots \mathcal{P}[\text{param}_n] &= \mathcal{E}[\text{exp}] \mathcal{V}[\text{exp}']
\end{align*} \]

The simplification of let bindings involves translating the right-hand-side expression and the body.

\[ \mathcal{E}[\text{let } x = \text{exp in } \text{exp}'] = \mathcal{R}[\text{exp}] \lambda \text{rhs}(\text{let } x = \text{rhs in } \mathcal{E}[\text{exp}']) \]

If a case is used to deconstruct a tuple value, then it will have only one rule, which we replace with select expressions:

\[ \mathcal{E}[\text{case exp of } \{ (x_0, \ldots, x_{n-1}) => \text{exp}' \}] =\]
\[ \mathcal{V}[\text{exp}] \lambda v(\text{let } x_0 = \#0(v) \text{ in } \cdots \text{let } x_{n-1} = \#(n-1)(v) \text{ in } \mathcal{E}[\text{exp}']) \]

Notice that tuple selection is zero-based! In the general form of a case expression, we translate each rule using the \( C \) translation function.

\[ \mathcal{E}[\text{case exp of } \text{rule}_1 \cdots \text{rule}_n] = \mathcal{V}[\text{exp}] \lambda v(\text{case } v \text{ of } C[\text{rule}_1] \cdots C[\text{rule}_n]) \]

The remaining AST forms all correspond to values, so we use \( V \) to translate them.

\[ \mathcal{E}[\text{exp}] = \mathcal{V}[\text{exp}] \lambda v(\text{ret}(v)) \]

### 5.1.2 The \( V \) translation

The \( V \) translation function translates AST expressions to SimpleAST values and passes the values to the supplied context (or continuation). For expressions that are already values, the translation is trivial; for all other expressions, we introduce a fresh variable that is bound to the translation of the expression. Notice that we discard the types for polymorphic nullary constructors.

\[ \begin{align*}
\mathcal{V}[x] k &= k(x) \\
\mathcal{V}[C \tau] k &= k(C) \\
\mathcal{V}[C] k &= k(C) \\
\mathcal{V}[\text{lit}] k &= k(\text{lit}) \\
\mathcal{V}[\text{exp}] k &= \mathcal{R}[\text{exp}] \lambda \text{rhs}(\text{let } x = \text{rhs in } k(x)) \quad x \text{ is fresh}
\end{align*} \]

### 5.1.3 The \( R \) translation

The \( R \) translation is used to translate AST expressions to SimpleAST right-hand-sides. The first four rules handle the various Basis operator and function cases. There are two rules for primitive operators and two rules for runtime-system functions.

\[ \begin{align*}
f \in \text{dom}(B) & \quad B(f) = p \\
\mathcal{R}[f \tau (\text{exp}_1, \ldots, \text{exp}_n)] k &= \mathcal{V}[\text{exp}_1] \lambda v_1(\cdots \mathcal{V}[\text{exp}_n] \lambda v_n(k(p(v_1, \ldots, v_n))) \cdots)
\end{align*} \]
\[
\begin{align*}
& f \in \text{dom}(B) \quad B(f) = p \\
& \mathcal{R}[f\ (exp_1, \ldots, exp_n)] k = \mathcal{V}[\text{[}\ exp_1 \text{]}] \lambda v_1 (\cdots \mathcal{V}[\text{[}\ exp_n \text{]}] \lambda v_n (k(p(v_1, \ldots, v_n))) \cdots) \\
& f \in \text{dom}(B) \quad B(f) = cf \\
& \mathcal{R}[f\ \{\{\ F x \Rightarrow exp \}\} \] k = \mathcal{V}[\text{[}\ exp \text{]}] \lambda v_1 (\cdots \mathcal{V}[\text{[}\ exp_n \text{]}] \lambda v_n (k(p(cf(v_1, \ldots, v_n)))) \cdots) \\
& f \in \text{dom}(B) \quad B(f) = cf \\
& \mathcal{R}[f\ (exp_1, \ldots, exp_n)] k = \mathcal{V}[\text{[}\ exp_1 \text{]}] \lambda v_1 (\cdots \mathcal{V}[\text{[}\ exp_n \text{]}] \lambda v_n (k(calls cf(v_1, \ldots, v_n))) \cdots) \\
\end{align*}
\]

For polymorphic constructor-function applications, we erase the type and translate the argument to a value.

\[
\mathcal{R}[\text{[}\ F \{\{\ F \} x \Rightarrow exp \}\} \] k = \mathcal{V}[\text{[}\ exp \text{]}] \lambda v(k(F \ v))
\]

While for monomorphic constructor functions we just translate the argument to a value and apply the construct.

\[
\mathcal{E}[F\ exp] = \mathcal{V}[\text{[}\ exp \text{]}] \lambda v(k(F \ v))
\]

Tuples require translating each element expression to a value and then building tuple.

\[
\mathcal{R}[\text{[}\ (exp_1, \ldots, exp_n) \] k = \mathcal{V}[\text{[}\ exp_1 \text{]}] \lambda v_1 (\cdots \mathcal{V}[\text{[}\ exp_n \text{]}] \lambda v_n (k\{v_1, \ldots, v_n\}) \cdots)
\]

All other AST expression forms are handled using the right-hand-side expression form of SimpleAST:

\[
\mathcal{R}[\text{[}\ exp \] k = k(\mathcal{E}[\text{[}\ exp \text{]}])
\]

### 5.1.4 The C translation

The C translation function handle the translation of case rules, where we know that the patterns can only be constructor applications, constructors, or variables.

\[
\begin{align*}
\mathcal{C}[\{\{\ F \} x \Rightarrow exp \} & = \{\{\ F \} x \Rightarrow \mathcal{E}[\text{[}\ exp \text{]}] \} \\
\mathcal{C}[\{\{\ C \} \Rightarrow exp \} & = \{\{\ C \} \Rightarrow \mathcal{E}[\text{[}\ exp \text{]}] \} \\
\mathcal{C}[\{\{\ x \Rightarrow exp \} & = \{\{\ x \Rightarrow \mathcal{E}[\text{[}\ exp \text{]}] \}
\end{align*}
\]

### 5.1.5 The P translation

The P translation function handles converting AST function parameters to Simple AST parameters. There are two cases:

\[
\begin{align*}
\mathcal{P}[\{\} & = () \\
\mathcal{P}[\{\ x \} & = \{\ x \}
\end{align*}
\]

### 5.2 Translation of Programs

A LangF program is represented as a sequence of declarations in the AST IR, where the last declaration is the main function. At runtime, the declarations are processed in order and then the main function is called on the command-line arguments. We capture this semantics in the SimpleAST
representation by creating a nested declaration structure (one declaration per top-level declaration) with a body that invokes the main function on the command-line arguments. For example, the LangF program

```plaintext
val n = 42;
fun main (args : List[String]) -> Int = n
```

will be simplified to

```plaintext
let n = 42 in
fun main (args) = 42 in
let args = call getArgs() in
main (args)
```

where getArgs is a runtime-system function that returns the list of command-line arguments.

We formalize this translation by defining one more translation function:

\[ D[\cdot] : \text{PROG} \rightarrow \text{EXP} \]

which translates an AST program to a SimpleAST expression. It is defined as follows:

\[ D[\text{data } \theta^{(k)}; \text{prog}] = D[\text{prog}] \]
\[ D[\text{fun } f \text{ param}_1 \cdots \text{param}_n = \text{exp}; \text{prog}] = \]
\[ \text{fun } f \ \text{P}[\text{param}_1] \cdots \text{P}[\text{param}_n] = E[\text{exp}] \ \text{in} \ D[\text{prog}] \]
\[ D[\text{let } x = \text{exp}; \text{prog}] = \text{R}[\text{exp}] \ \lambda \text{rhs} (\text{let } x = \text{rhs in } D[\text{prog}]) \]
\[ D[\text{fun } \text{main } x = \text{exp}] = \]
\[ \text{fun } \text{main } (x) = E[\text{exp}] \ \text{in} \]
\[ \text{let } t = \text{call getArgs()} \ \text{in} \]
\[ \text{main } (t) \]

Note that in an actual implementation of this translation function, we need to convert constructors of a data declaration to their SimpleAST equivalents as described in Section 4.1.

## 6 Optimization

The second part of the project is to implement optimization for the SimpleAST IR. In this section, we describe several simple optimizations; the first of these (shrinking) is required, while any of the others may be implemented for extra credit.

We will post a supplemental document that discusses optimization in more detail.

### 6.1 Shrinking

A very useful, but simple, optimization pass is called shrinking (or contraction). This pass combines constant folding (i.e., compile-time evaluation of arithmetic), dead-variable elimination, and denesting of let expressions.

The analysis required for shrinking is very simple; we perform a census that counts for every variable in the program the number of use occurrences of the variable, with the idea being that
variables with a use count of zero are unused and thus can be eliminated. The other piece of information that we need is a mapping from variables to information about what they are bound to. With this information, then we can perform several kinds of “constant folding” on expressions:

- Constant folding of arithmetic operations applied to known values.
- Constant folding of tuple selection when applied to a variable bound to a tuple.
- Constant folding of conditional expressions on known values.
- Constant folding of case expressions where the argument is a known data constructor.

6.2 Adding Contractive Inlining [Extra Credit]

If a function is only called once (but not recursively), inlining the function call will shrink the size of the program and (almost always) improve performance. For extra credit, you should add this optimization to the contraction pass.

6.3 Uncurrying [Extra Credit]

For extra credit, you may implement the uncurrying optimization. The idea of this optimization is to convert curried function definitions and applications to ones involving multiple parameters. For example, the expression

```haskell
{ fun f (a) (b) (c) =
    let t1 = IntMul(a, b) in
    let t2 = IntAdd(t1, c) in
    ret t2
  in
  let t3 = f 1 in
  let t4 = t3 2 in
  let t5 = t4 3 in
  ret t5
}
```

would be converted to

```haskell
{ fun f (a, b, c) = {
    let t1 = IntMul(a, b) in
    let t2 = IntAdd(t1, c) in
    ret t2
  in
  let t3 = f (1, 2, 3) in
  ret t3
}
```

Note that unused variables should only be eliminated when they are bound to a pure (or side-effect free) computation.
7 Implementation

You should start by implementing and testing the simplifier before you worry about optimization, but leave yourself time for the optimizer. As with the previous projects, the provided sample code supports dumping the SimpleAST IR to a text file (both before and after optimization).

The translation functions given above do not explicitly handle the translation of types, data constructors, or variables. In your implementation, you will need to add these parts of the translation. The sample code includes the necessary infrastructure for doing so.

8 Submission

We will collect the projects at 23:59 (Chicago time) on November 20, 2020 from the SVN repositories, so make sure that you have committed your final version before then.

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.

9 Document history

November 19, 2020 Fix type in last rule for \( R \) translation.

November 19, 2020 Extend deadline to November 20.

November 17, 2020 Fixed color of case expressions in the definition of the \( E \) translation function.

November 15, 2020 Added missing application of context in tuple case for the definition of \( R \).

November 15, 2020 Fixed the definition of \( D \) for \texttt{let} bindings.

November 15, 2020 Removed mention of “lowering case expressions to switches,” that will be part of Project 4.

November 15, 2020 Fixed a typo in the definition of the \( E \) and \( R \) translations.

November 13, 2020 Corrected the due date for the project.

November 11, 2020 Fixed a typo and added a sentence about processing \texttt{data} declarations in Section 5.2.

November 7, 2020 Original version.