1 Introduction

The project for the course is to implement a small functional programming language, called Flang. (Students who have taken CMSC 22100 should recognize it as an enrichment of System F, the polymorphic \(\lambda\)-calculus.) The project will be divided into five parts:

1. Extending the front-end with non-local control flow, mutually recursive data type and function definitions, and GADTs.
2. Converting the typed AST to the LambdaIR representation.
4. LambdaIR contraction.
5. LambdaIR analysis and optimizations.

Each part of the project builds upon the previous parts, but we will provide reference solutions for previous parts. You will implement the project in the Standard ML programming language and submission of the project milestones will be managed using Phoenixforge.

1.1 Schedule

The tentative schedule for the project assignments is as follows:

<table>
<thead>
<tr>
<th>Assigned</th>
<th>Project description</th>
<th>Due date</th>
</tr>
</thead>
<tbody>
<tr>
<td>April 1</td>
<td>Extending the Flang Front End</td>
<td>Wednesday, April 15</td>
</tr>
<tr>
<td>April 16</td>
<td>Conversion to LambdaIR</td>
<td>Monday, April 27</td>
</tr>
<tr>
<td>April 28</td>
<td>Basic LLVM Code Generation</td>
<td>Monday, May 11</td>
</tr>
<tr>
<td>May 12</td>
<td>LambdaIR Contraction</td>
<td>Monday, May 25</td>
</tr>
<tr>
<td>May 21</td>
<td>More LambdaIR Optimizations</td>
<td>Wednesday, June 10</td>
</tr>
</tbody>
</table>

All project assignments will be due at 10pm.

2 Flang

Flang is a strongly-typed, call-by-value, higher-order, polymorphic, functional programming language. The syntax and semantics of Flang are similar to other functional programming languages
(e.g., Standard ML, Haskell), but with many simplifications and a more explicit type system. **Flang**
does not have type inference, exceptions, references, or a module system. **Flang** does have first-
class functions, datatypes, and first-class polymorphism.

This document specifies the concrete syntax of **Flang** and gives an informal description of its features. A companion document (*The Flang Type System*) gives a formal description of the language’s type system.

### 2.1 Types

**Flang** supports two primitive types of values: integers (type `Integer`) and strings (type `String`). In addition, **Flang** has datatype-constructed values, function values, and type-function values. The grammar of types is

\[
\text{Type} ::= \text{TypeParams Type} \\
| \text{Type} \rightarrow \text{Type} \\
| \text{Type} (\star \text{Type})^{opt} \\
| \text{TyId TypeArgs}^{opt} \\
| \text{TyVar} \\
| (\text{Type}) \\
\]

\[
\text{TypeParams} ::= [\text{TyVar} (, \text{TyVar})^*] \\
\]

\[
\text{TypeArgs} ::= [\text{Type} (, \text{Type})^*] \\
\]

**Flang** enforces the convention that type constructor names (TyId) begin with an upper-case letter and that type variable names (TyVar) begin with a lower-case letter. The \(\rightarrow\) constructor associates to the right and has lower precedence than the tuple-type constructor (\(\star\)), while type abstraction has the lowest precedence. Some examples:

- `Integer \rightarrow Integer`
- `[a] a \rightarrow a`
- `([a] a \rightarrow String) \rightarrow [a] List[a] \rightarrow String`
- `Integer \star Integer \rightarrow Bool`

In addition to `Integer` and `String`, **Flang** predefines several data types, such as `Bool`, `List`, and `Unit`.

### 2.2 Programs

A **Flang** program is a sequence of top-level definitions followed by an expression.

\[
\text{Program} ::= (\text{Definition ;})^* \text{ Exp} \\
\]

Executing a **Flang** program means evaluating each of the declarations (making their definitions available to the subsequent declarations and expression) and then evaluating the final expression. Here is a very simple **Flang** program that computes 5! by defining the `fact` function followed by
the expression `fact 5`:

```haskell
(* program declarations *)
fun fact (n : Integer) -> Integer =
    if n == 0 then 1 else n * fact (n - 1);
(* program expression *)
fact 5
```

Note that a `(*` is used to start a comment and `*)` is used to terminate a comment; as in SML, comments may be nested.

### 2.3 Top-level definitions

There two kinds of top-level definitions in Flang: definitions of types and definitions of values. Type definitions are further divided into simple type definitions that are used to define a synonym (or alias) for a type and data-type definitions that are used to define data structures, while value definitions include function definitions, let bindings, and expressions.

**Definition**

```
::= type TyId TypeParams^opt = Type
    | data DataDef (and DataDef)^*
    | ValBind
```

**ConDef**

```
::= con ConId (of Type)^opt
```

**ValBind**

```
::= fun FunDef (and FunDef)^* 
    | let SimplePat ( : Type)^opt = Exp 
    | Exp 
```

**FunParam**

```
::= TypeParams 
    | ( ValId : Type ) 
```

We describe these various forms below.

#### 2.3.1 Simple type definitions

A Flang type declaration introduces another name for a type; the new type name may be used in subsequent declarations and expressions. For example, we might wish to abbreviate the type of a curried integer comparison function (a function from two integers to a boolean):

```
type IntCmp = Integer -> Integer -> Bool
```

Note that a type declaration is introduced with the `type` keyword and that type names are written with a leading upper-case letter.

A Flang type declaration may also include type parameters, which must be instantiated at each use of the new type name. For example, we might wish to abbreviate the type of a general comparison function (a function from two values of the same (but any) type to a boolean) and then define the type of an integer comparison function in terms of the general comparison function:
type Cmp [a] = a -> a -> Bool
type IntCmp = Cmp [Integer]

Note that type parameters and type arguments are written in [ . . . ] brackets and, as in function parameters, that type variables are written with a leading lower-case letter. Multiple type parameters and type arguments are separated by , s:

```plaintext
type BinOp ['a, 'b] = 'a -> 'a -> 'b
type Cmp ['a] = BinOp ['a, Bool]
type IntCmp = Cmp [Integer]
```

Unlike polymorphic functions, a type name cannot be partially applied; at every use of the type name, all type parameters must be instantiated.

### 2.3.2 Data-type definitions

A Flang datatype declaration introduces a new type along with constructors; the constructors provide the means to create values of the new type and to take apart values of the new type. Each constructor is declared with the types of its argument(s). A very simple datatype declaration is one for defining the relationship between values in a total order:

```plaintext
data Order with
  con Less
  con Equal
  con Greater
```

This definition introduces both new type (Order) and three data constructors (Less, Equal, and Greater). Note that constructor names are written with a leading upper-case letter. A slightly more complicated datatype declaration is one that represents publications, which can be either a book (with an author and a title) or an article (with an author, a title, and a journal name):

```plaintext
data Publication with
  con Book of String * String
  con Article of String * String * String
```

A Flang datatype declaration may also include type parameters (yielding a polymorphic datatype), which must be instantiated at each use of the new type name. The types of a constructor’s arguments(s) may use the type parameters. For example, the Pair datatype takes two type parameters and introduces a constructor with two arguments of the types of the parameters:

```plaintext
data Pair [a, b] with con Pair of a * b
```

As in SML, data type definitions may be joined by the “and” keyword, which allows them to be mutually recursive. For example

```plaintext
data Tree [a] with
  con EmptyT
  con Forest of (a * Forest[a])
and Forest [a] with
  con EmptyF
  con Tree of Tree[a] * Forest[a];
```
2.3.3 Function definitions

Function definitions introduce functions that are parameterized over types and values. Functions may be recursive, but Flang does not support mutually recursive functions directly. For example, here is a recursive function that computes the length of a list:

```plaintext
fun length [a] (xs : List[a]) -> Integer =
  case xs of
  { _::r => 1 + length [a] r }
  { Nil => 0 }
end
```

A defining characteristic of Flang (taken from System F, the polymorphic λ-calculus) is polymorphism or type abstraction. The prototypical example of a polymorphic function is the identity function, which simply returns its argument (without performing any computation on it). Thus, the behavior of the function is the same for all possible types of its argument (and result). The function declaration for the identity function introduces one function parameter (a type variable) to be used as the type of the second function parameter and the result type:

```plaintext
fun id [a] (x : a) -> a = x;
```

Note that type variables are written with a leading lower-case letter.

Like (ordinary) functions, polymorphic functions in Flang are first-class: they may be nested, taken as arguments, and returned as results. To use a polymorphic function, it must be applied to a type, rather than to an expression. The result of applying a polymorphic function to a type is a value having the type produced by instantiating the type variable with the applied type. For example, the result of applying the identity function to the integer type is a function having the type `Integer -> Integer`:

```plaintext
fun id [a] (x : a) -> a = x;
let _ : [a] a -> a = id;
let _ : Integer -> Integer = id [Integer];
let zero : Integer = id [Integer] 0;
```

Note that the polymorphic function type is written using the syntax

```plaintext
[ TyVar, ... , TyVar ] Type
```

Also note that the type variable in a function parameter and in a polymorphic function type is a binding occurrence of the type variable; two polymorphic function types are equal if each of the bound type variables in one can be renamed to match the bound type variables in the other:

```plaintext
fun id [a] (x : a) -> a = x;
let _ : [b] a -> b = id;
let _ : [c] -> c -> c = id;
```

In function declarations, type variable and value parameters may be mixed, but a type variable parameter must occur before any use of the type variable in the types of value parameters.

```plaintext
fun revApp [a] (x : a) [b] (f : a -> b) -> b = f x;
let _ : [b] (Integer -> b) -> b = revApp [Integer] 1;
fun double (y : Integer) -> Integer = 2 * y;
let two = revApp [Integer] 1 [Integer] double;
```

The above examples also demonstrate that a function with more than one parameter (either type
variable parameters or value parameters) is a *curried* function and can be partially applied to types or expression arguments.

As in SML, mutually recursive functions are joined by the “**and**” keyword. For example,

```haskell
fun isEven (n : Integer) -> Bool =
  n == 0 || isOdd (n - 1)
and isOdd (n : Integer) -> Bool =
  if n == 0 then False else isEven (n - 1);
```

### 2.3.4 Value definitions

In addition to function definitions, **Flang** allows let binding of value identifiers and expressions\(^1\) as top-level definitions. Let bindings introduce new value identifiers that are bound to the result of evaluating the right-hand-side expression.

### 2.4 Expressions

**Flang** is an *expression* language, which means that all computation is done by expressions (there are no statements). Furthermore, **Flang** is a *call-by-value* language, which means that (almost) all sub-expressions are evaluated to values before the expression itself is evaluated.

#### 2.4.1 Conditionals

**Flang** provides a conditional expression with the syntax

```
if Exp then Exp else Exp
```

and the expected semantics. The conditional must have the builtin type **Bool** and the arms of the conditional must have the same type. The conditional expression is the lowest-precedence expression form.

#### 2.4.2 Binary expressions

**Flang** defines a small collection of infix binary operators as described in the following table:

---

\(^1\)Expressions can be thought of as a degenerate form of value binding.
### Operator Associativity Description

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Left</td>
<td>and-also conditional expression</td>
</tr>
<tr>
<td>==</td>
<td>Left</td>
<td>integer equality relation</td>
</tr>
<tr>
<td>&lt;</td>
<td>Left</td>
<td>integer less-than relation</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Left</td>
<td>integer less-than-or-equal relation</td>
</tr>
<tr>
<td>::</td>
<td>Right</td>
<td>list cons operator</td>
</tr>
<tr>
<td>@</td>
<td>Left</td>
<td>String concatenation operator</td>
</tr>
<tr>
<td>+</td>
<td>Left</td>
<td>Integer addition operator</td>
</tr>
<tr>
<td>-</td>
<td>Left</td>
<td>Integer subtraction operator</td>
</tr>
<tr>
<td>*</td>
<td>Left</td>
<td>Integer multiplication operator</td>
</tr>
<tr>
<td>/</td>
<td>Left</td>
<td>Integer division operator</td>
</tr>
<tr>
<td>%</td>
<td>Left</td>
<td>Integer modulo operator</td>
</tr>
</tbody>
</table>

The operators are listed in order of increasing precedence, with horizontal lines separating the difference precedence levels.

Note that the two conditional operators (|| and &&) are not strict in their second argument. In fact, they may be viewed as defining the following syntactic sugar:

\[
\begin{align*}
\text{e}_1 || \text{e}_2 & \Rightarrow \text{if } \text{e}_1 \text{ then True else e}_2 \\
\text{e}_1 && \text{e}_2 & \Rightarrow \text{if } \text{e}_1 \text{ then e}_2 \text{ else False}
\end{align*}
\]

#### 2.4.3 Application

There are two forms of application expressions in Flang: value application and type application. Both of these forms associate to the left and have higher precedence than binary operators. For example:

```flang
id [Integer] 0
foldl [Integer, Integer]
fact (n-1)
reverse [Bool * Bool] ((True, False) :: (False, True) :: Nil[Bool * Bool])
```

#### 2.4.4 Variables and constants

Variables, data constructors, numbers, and string literals are all expressions in Flang.

#### 2.4.5 Tuple expressions

Flang supports tuples of values using the syntax

\[
( (Exp , Exp)^* )^{opt}
\]

The expression () is shorthand for the Unit constructor; the expression (e) is just the expression e, where the parentheses have been used to make precedence and associativity explicit; and the expression \((e_1, e_2, \ldots, e_n)\) defines an n-ary tuple.
2.4.6 Blocks

A block introduces a nested scope that can include function and value bindings and has the syntax

\[
\{ \text{Scope} \}
\]

where a Scope is a sequence of value bindings followed by an expression

\[
(\text{ValBind} ; )^+ \text{Exp}
\]

Flang follows standard lexical scoping rules: bound identifiers have a scope that consists of the rest of the block (the scope of a function includes its body), but subsequent definitions of the same identifier will override (or shadow) the outer definition.

2.4.7 Case expressions

Flang provides case expressions with simple (one-level) patterns. For example, the body of the reverse function from above is a case on a list:

```flang
  case xs of
  { _::_:r => 1 + length [a] r }
  { Nil => 0 }
  end
```

Note that polymorphic constructors in patterns are not applied to types, since one can use the argument type of the case to determine how to instantiate the polymorphism. Each rule of the case consists of a pattern, which may bind variables, and a Scope consisting of bindings and an expression.

2.4.8 Non-local control flow

Flang supports non-local control via two constructs: the catch expression, which encloses a computation with a handler, and the throw expression, which transfers control to the dynamically innermost handler. The catch expression has the syntax

```flang
  try \text{Exp} catch \{ \text{SimplePat} => \text{Scope} \}
```

and wraps the evaluation of the expression with a handler.

A non-local transfer of control is initiated by the throw expression, which has the syntax

```flang
  throw [ \text{Type} ] \text{Exp}
```

and which evaluates the string-valued expression Exp and then transfers control to the innermost handler. The argument is matched against the pattern of the handler. Since the throw expression does not return to its context, we must specify its type, which is the rôle of the type argument.

A non-local transfer of control can also be caused by the operators / and \%, when their second argument is 0.
3 The collected grammar of Flang

3.1 Lexical issues

There are five classes of tokens in Flang:

1. lower-case identifiers: a, b, toString, y23, etc.
2. upper-case identifiers: X, Foo, SOME_VAL, etc.
3. numbers: 0, 42, etc.
4. strings: "hello\world", "some\ntext", etc.
5. delimiters and operators: (, ), =, <=, +, etc.

Tokens can be separated by whitespace and/or comments.

Type-variable, type-constructor, data-constructor, and value identifiers in Flang can be any string of letters, digits, underscores, and quote marks, beginning with a letter. Identifiers are case-sensitive (e.g., foo is different from Foo). We use distinguish between identifiers that begin with an upper-case letter and those that begin with lower-case letters. We use upper-case identifiers for type and data constructors, and lower-case identifiers for type and value variables. The following lower-case identifiers are reserved as keywords:

    and case catch con data
de else end fun if let
of then throw try type
with

Flang also has a collection of delimiters and operators, which are the following:

    (  ) [  ] {  }
    = || && == <= <
    : :: @ + - *
    / % , ; -> =>

Numbers in Flang are integers and their literals are written using decimal notation (without a sign).

String literals are delimited by matching double quotes and can contain the following C-like escape sequences:

    \a — bell (ASCII code 7)
    \b — backspace (ASCII code 8)
    \f — form feed (ASCII code 12)
    \n — newline (ASCII code 10)
    \r — carriage return (ASCII code 13)
    \t — horizontal tab (ASCII code 8)
    \v — vertical tab (ASCII code 11)
    \ — backslash
    \" — quotation mark
A character in a string literal may also be specified by its numerical value using the escape sequence `'\ddd'`, where `ddd` is a sequence of three decimal digits. Strings in Flang may contain any 8-bit value, including embedded zeros, which can be specified as `'\000'`.

Comments may start anywhere outside a string with “(*” and are terminated with a matching “*)”. As in SML, comments may be nested.

Whitespace is any non-empty sequence of spaces (ASCII code 32), horizontal or vertical tabs, form feeds, newlines, or carriage returns. Any other non-printable character is treated as an error.

3.2 Grammar

The collected syntax of Flang given below using an extended-BNF format. Literal symbols, such as keywords and punctuation, are written in a **bold fixed-width font**, other terminal symbols are written in roman font, and non-terminal symbols are written in italic font. We use the following terminal symbols in the grammar:

<table>
<thead>
<tr>
<th>Terminal</th>
<th>Lexical class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TyId</td>
<td>upper-case identifier</td>
<td>type constructor and data types</td>
</tr>
<tr>
<td>TyVar</td>
<td>lower-case identifier</td>
<td>type variable</td>
</tr>
<tr>
<td>ConId</td>
<td>upper-case identifier</td>
<td>data constructor</td>
</tr>
<tr>
<td>ValId</td>
<td>lower-case identifier</td>
<td>value identifier</td>
</tr>
</tbody>
</table>

The collected syntax follows:

```
Program ::= (Definition ;)* Exp

Definition ::= type TyId TypeParams*opt = Type | data DataDef (and DataDef)* | ValBind

DataDef ::= TyId TypeParams*opt with ConDef+ |

ConDef ::= con ConId (of Type)*opt

TypeParams ::= [ TyVar (, TyVar)* ]

Type ::= TypeParams Type | Type -> Type | Type (* Type)* | TyId TypeArgs*opt | TyVar | ( Type )

TypeArgs ::= [ Type (, Type)* ]
```

10
ValBind ::= fun FunDef (and FunDef)*
         | let SimplePat (: Type)* = Exp
         | Exp

FunDef ::= ValId FunParam* --> Type = Exp

FunParam ::= TypeParams
         | (ValId : Type)

Exp ::= if Exp then Exp else Exp
       | Exp || Exp
       | Exp && Exp
       | Exp == Exp
       | Exp < Exp
       | Exp <= Exp
       | Exp ::: Exp
       | Exp @ Exp
       | Exp + Exp
       | Exp - Exp
       | Exp * Exp
       | Exp / Exp
       | Exp % Exp
       | ApplyExp

ApplyExp ::= AtomicExp
           | ApplyExp AtomicExp
           | ApplyExp TypeArgs
           | throw [Type] ApplyExp

AtomicExp ::= ValId
             | ConId
             | Int
             | String
             | (Exp , Exp)*
             | {Scope}
             | case Exp of MatchCase* end
             | try Exp catch {SimplePat => Scope}

Scope ::= (ValBind ;)* Exp

MatchCase ::= {Pat => Scope}
```
Pat ::= SimplePat
    | ConId SimplePat opt
    | SimplePat :: SimplePat
    | ( SimplePat (: SimplePat)* )

SimplePat ::= ValId
             | _
```

### 3.3 Syntactic conventions

The grammar as written has some ambiguities, which are resolved by specifying the precedence and associativity of operators.

For types, the \( \rightarrow \) constructor associates to the right and has lower precedence than the tuple-type constructor \( \ast \). Type abstraction has the lowest precedence.

For binary expressions, all operators are left associative, except the infix list cons operator \( :: \), which is right associative. Binary operations are grouped into seven precedence levels from lowest to highest as follows:

- or-else operator: \( || \)
- and-also operator: \( && \)
- relational operators: \( ==, <, \) and \( <= \)
- list operator: \( :: \)
- string operator: \( @ \)
- addition operators: \( + \) and \( - \)
- multiplication operators: \( \ast, /, \) and \( \% \)

### Document History

March 31, 2015  Original version