CMSC 22620
Spring 2009

Implementation of Computer Languages

Handout 2
March 31, 2009

Project overview

1 Introduction

The project for the course is to implement code generation and optimizations for a small functional programming language, called *LangF*.

Each part of the project builds upon the previous parts (but reference solutions for previous parts will be available). The implementation will be undertaken using the Standard ML programming language and submission of the project milestones will be managed using the course GForge server. Programming projects will be individual efforts (no group submissions).

2 LangF

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

For those of you familiar with the version of LangF used in CMSC 22610, we have added two new features:

1. A mutable array type constructor (`Array['a]`), with supporting operations.
2. Support for non-local control-flow with the **`try`** and **`escape`** expressions.

In addition, you will be generating code to run on the x86-64 architecture, so the `Integer` type will span the range -2^{62} to $2^{62} - 1$.

2.1 Types and Values

LangF supports two primitive types of values, integers and strings, and the primitive array type constructor. In addition, LangF has datatype-constructed values (including predefined datatypes for booleans and units), function values, and polymorphic-function values.

2.2 Programs

A LangF program is a sequence of declarations, which are either type, datatype, value, or function definitions, followed by an expression. Executing a LangF program means evaluating each of the declarations (making their definitions available to the subsequent declarations and expression) and then evaluating the expression. Here is a very simple LangF program that computes 5! by defining the `fact` function followed by the expression `fact 5`:

```
(* 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 separate the declarations from the expression (but multiple declarations are not separated by any delimiter) and a **`(*`** is used to start a comment and **`*)`** is used to terminate a comment; comments may be nested.

2.3 Simple Expressions

LangF is an *expression* language, which means that all computation is done by expressions (there are no statements). Furthermore, LangF is a call-by-value language, which means that (almost) all sub-expressions are evaluated to values before the expression itself is evaluated. The body of the `fact` function in the example program above has introduced many of the simple LangF expressions:

- variables: `n`
- integer constants: `0`, `1`
- integer comparisons: `n == 0`
- integer operations: `n - 1`
- array indexing: `a!i`
- array updating: `a!i := x`
- function applications: `fact (n - 1)`
- conditionals: **`if`** `n == 0` **`then`** `1` **`else`** `n * fact (n - 1)`

Thus, many of the simple LangF expressions deal with values of the primitive integer and string types and the built-in boolean datatype. For reference, the following is the full collection of integer-centric expressions:

expression form	meaning
integer	integer constant
Exp **==** *Exp*	integer equality
Exp **<>** *Exp*	integer inequality
Exp **<** *Exp*	integer less-than
Exp **<=** *Exp*	integer less-than-or-equal
Exp **>** *Exp*	integer greater-than
Exp **>=** *Exp*	integer greater-than-or-equal
Exp **+** *Exp*	integer addition
Exp **-** *Exp*	integer subtraction
Exp ***** *Exp*	integer multiplication
Exp **/** *Exp*	integer division
Exp **%** *Exp*	integer modulus
Exp **!** *Exp*	array subscript
~ *Exp*	unary integer negation

the following is the full collection of string-centric expressions:

expression form	meaning
string	string constant
Exp **^** *Exp*	string concatenation

and the following is the full collection of boolean-centric expressions:

expression form	meaning
if *Exp* **then** *Exp* **else** *Exp*	conditional
Exp **orelse** *Exp*	short-circuiting disjunction
Exp **andalso** *Exp*	short-circuiting conjunction

2.4 Value Declarations

A LangF value declaration simply binds a name to an expression; the name may be used in subsequent declarations and expressions. When a value declaration is evaluated, the expression is evaluated to a value and the value is bound to the name (making the value available to subsequent declarations and expressions through the name). The following simple value declaration binds the name `two` to the integer expression `1 + 1`:

```
val two = 1 + 1
```

Note that a value declaration is introduced with the **val** keyword and that (value variable) names are written with a leading lower-case letter. A LangF value declaration may optionally assert the type of the bound expression:

```
val four : Integer = two + two
```

Such type assertions can be useful to understand why a program has (or does not have) type errors.

2.5 Function Declarations and Anonymous Function Expressions

A LangF function declaration defines a function and introduces a name for the function; the function name may be used in subsequent declarations and expressions. For example, we might wish to define a function to add two integers:

```
fun add (a: Integer) (b: Integer) : Integer = a + b
```

Note that a function declaration is introduced with the **fun** keyword, followed by the function name, one or more function parameters (each a variable with its type), the function result type, and an expression for the function body. Functions in LangF are first-class: they may be nested, taken as arguments, and returned as results. A function with more than one parameter is a *curried* function; thus the following program evaluates to 4:

```
fun add (a: Integer) (b: Integer) : Integer = a + b
val _ : Integer -> Integer -> Integer = add
val inc : Integer -> Integer = add 1
;
inc (add 1 2)
```

Note that the function type is introduced with the syntax *Type* **->** *Type*; the `add` function has the type `Integer -> Integer -> Integer`; applying it to the argument 1 returns a function having the type `Integer -> Integer` (which is bound to the name `inc`).

A LangF function declaration may be recursive and a collection of function declarations may be mutually recursive. Thus, the function body may use the function names introduced by the function declaration(s). For example, in the factorial function, the function name `fact` is used in the function body:

```
fun fact (n: Integer) : Integer =
  if n == 0 then 1 else n * fact (n - 1)
```

As an example of mutually recursive functions, consider the `isEven` and `isOdd` functions:

```
fun isEven (n: Integer) : Bool =
  if n == 0 then True else isOdd (n - 1)
and isOdd (n: Integer) : Bool =
  if n == 0 then False else isEven (n - 1)
```

Note that functions declarations in a collection of mutually recursive function declarations are separated with the **and** keyword. Also note that within a collection of mutually recursive function declarations, all of the introduced function names must be distinct.

In addition to function declarations, LangF has anonymous function expressions. For example, we might wish to return a function to multiply two integers (without introducing a name for the function):

```
        fn (x: Integer) (y: Integer) => x * y
```

Note that an anonymous function expression is introduced with the `fn` keyword, followed by one or more function parameters (each a variable with its type) and an expression for the function body. A LangF anonymous function expression cannot be recursive (as there is no name for the function).

2.6 Polymorphic Functions

A defining characteristic of LangF (taken from *System F*, the polymorphic λ-calculus) is *polymorphism* or *type abstraction*. The prototypical example of a polymorphic function is the identity function. The identity function 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:

```
        fun id ['a] (x: 'a) : 'a = x
```

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

Like (ordinary) functions, polymorphic functions in LangF 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`:

```
        fun id ['a] (x: 'a) : 'a = x
        val _ : ['a] -> 'a -> 'a = id
        val _ : Integer -> Integer = id [Integer]
        val zero : Integer = id [Integer] 0
```

Note that the polymorphic function type is introduced with the syntax **[** *tyvarid* **] ->** *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:

```
        fun id ['a] (x: 'a) : 'a = x
        val _ : ['b] -> 'b -> 'b = id
        val _ : ['c] -> 'c -> 'c = id
```

Anonymous function expressions may also be used to return polymorphic functions:

```
      val twice : ['a] -> ('a -> 'a) -> 'a -> 'a =
        fn ['a] (f: 'a -> 'a) (x: 'a) => f (f x)
      val two : Integer = twice [Integer] (fn (x: Integer) => x + 1) 0
      val zzzz : String = twice [String] (fn (s: String) => s ^ s) "z"
```

In function declarations and anonymous function expressions, type variable and (value) variable parameters may be mixed; however, a type variable parameter must occur before any use of the type variable in the types of (value) variable parameters:

```
val revApp = fn ['a] (x: 'a) ['b] (f: 'a -> 'b) => f x
val _ : ['b] -> (Integer -> 'b) -> 'b = revApp [Integer] 1
val two = revApp [Integer] 1 [Integer] (fn (x: Integer) => x * 2)
```

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

2.7 Non-local control flow

While LangF does not have ML-style exceptions, it does provide a non-local control-flow mechanism for error handling. The mechanism is similar to, but more restricted than, C's `setjmp`/`longjmp` operations. For example, the following code uses this mechanism to short-circuit evaluation when a zero-valued leaf is encountered:

```
datatype Tree = ND { Tree, Tree } | LF { Integer };
fun treeMul (t : Tree) : Integer = let
      fun mul (t : Tree) : Integer = case t
           of LF{i} => if i == 0 then escape[Integer] else i
            | ND{t1, t2} => treeMul t1 * treeMul t2
          end
      in
        try mul t catch 0 end
      end
```

Note that the **`escape`** operator takes a type argument; this type is the type of the expression.

2.8 Type Declarations

A LangF 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 an integer comparison function (a function from two integers to a boolean):

```
type IntCmp = Integer -> Integer -> Bool
val intEq : IntCmp = fn (x: Integer) (y: Integer) = x == y
```

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

A LangF 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 **[** ... **]** and, as in function parameters, that type variables are written with a leading `'` and lower-case letter. Multiple type parameters and type arguments are separated by **,**s and **[ ]** can be used to be explicit about the absence of type parameters and type arguments:

```
        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.

2.9 Datatype Declarations and Constructor and Case Expressions

A LangF 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 the one for booleans that introduces two constructors (`True` and `False`), each with no arguments:

```
        datatype Bool = True | False
```

Note that a datatype declaration is introduced with the **`datatype`** keyword and 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):

```
        datatype Publication = Book {String, String}
                             | Article {String, String, String}
```

Note that the types of a constructor's argument(s) are written in { ... } separated by **,**s and { } can be used to be explicit about the absence of arguments.

To create a value having a datatype type, a constructor name is applied to argument(s):

```
    val pub1 : Publication =
      Book {"Lawrence Paulson", "ML for the Working Programmer"}
    val pub2 : Publication =
      Article {"Andrew Appel", "A Critique of Standard ML",
               "Journal of Functional Programming"}
```

As in constructor declarations, the arguments of a constructor are written in { ... } separated by **,**s and { } can be used to be explicit about the absence of arguments. Unlike functions, a constructor name cannot be partially applied; at every use of the constructor name, all arguments must be given.

To take apart a value having a datatype type, a collection of *match rules* (each with a *pattern* involving the constructor names and an expression) is checked successively against an argument; when a pattern matches the argument, its expression is returned as the result:

```
fun not (b: Bool) : Bool =
  case b of True => False | False => True end
fun author (p: Publication) : String =
  case p of
     Book {author, title} => author
   | Article {author, title, journal} => author
  end
val lawrence : String = author pub1
val andrew : String = author pub2
```

Note that a case expression is introduced with the **case** keyword, followed by the case-expression argument, the **of** keyword, one or more match rules, and terminated by the **end** keyword. Each match rule is introduced with the syntax *Pat* **=>** *Exp* and separated by **|**s. In a match rule, the pattern is either a constructor name applied to simple pattern(s) or a simple pattern; a simple pattern is either a variable name or an _. When a pattern matches a value, pattern variable names are bound to the corresponding value components and may be used in the match rule expression. Thus, in the evaluation of `author pub1` in the example above, the pattern `Book {author, title}` matches the value `Book {"Lawrence Paulson", "ML for the Working Programmer"}`, making `author` bound to `"Lawrence Paulson"`. The following example uses _ patterns:

```
fun isBook (p: Publication) : Bool =
  case p
   of Book {_, _} => True
    | _ => False
  end
fun isJournal (p: Publication) : Bool =
  case p
   of Journal {_, _, _} => True
    | _ => False
  end
```

A LangF 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:

```
datatype Pair ['a, 'b] = Pair {'a, 'b}
```

To create a value having a polymorphic-datatype type, a constructor name is applied to type argument(s) and argument(s):

```
val one_hello = Pair [Integer, String] {1, "hello"}
```

The type arguments of a constructor are written in **[** . . . **]** separated by **,**s and **[ ]** can be used to be explicit about the absence of type arguments. Unlike polymorphic functions, a polymorphic-constructor name cannot be partially applied; at every use of the polymorphic-constructor name, all type arguments and arguments must be given.

Similarly, to match a value having a polymorphic-datatype type, a constructor name is applied to type argument(s) and simple pattern(s) in a match rule:

```
fun fst ['a] ['b] (p: Pair ['a, 'b]) : 'a =
  case p of Pair ['a, 'b] {x, y} => x end
fun snd ['a] ['b] (p: Pair ['a, 'b]) : 'b =
  case p of Pair ['a, 'b] {x, y} => y end

val one : Integer = fst [Integer] [String] one_hello
val hello : String = snd [Integer] [String] one_hello
```

A LangF datatype declaration may be recursive and a collection of datatype declarations may be mutually recursive. Thus, the types of a constructor's argument(s) may use the type names introduced by the datatype declaration(s). For example, in the list datatype, the type of the second argument of the `Cons` constructor is itself the list datatype:

```
datatype List ['a] = Nil | Cons {'a, List ['a]}
```

As an example of mutually recursive datatypes, consider variadic trees that are either empty or a node with an element and a forest and forests that are either empty or a node with a tree and a forest:

```
datatype Tree ['a] = EmptyT | Forest {'a, Forest ['a]}
and Forest ['a] = EmptyF | Tree {Tree ['a], Forest ['a]}
```

Note that datatype declarations in a collection of mutually recursive datatype declarations are separated with the **and** keyword. Also note that within a collection of mutually recursive datatype declarations, all of the introduced type names must be distinct and all of the introduced constructor names must be distinct; however, type names and constructor names need not be distinct (as in the pair datatype declaration and in the tree/forest datatype declarations).

The following mutually recursive functions compute the height of variadic trees and forests:

```
fun max (x: Integer) (y: Integer) : Integer =
  if x > y then x else y
fun heightTree ['a] (t: Tree 'a) : Integer =
  case t of
     EmptyT ['a] => 0
   | Forest ['a] {_, f} => 1 + (heightForest ['a] f)
  end
and heightForest ['a] (f: Forest 'a) : Integer =
  case f of
     EmtpyF ['a] => 0
   | Tree ['a] {t, f} =>
       max (heightTree ['a] t) (heightForest ['a] f)
```

2.10 Miscellaneous Expressions

LangF include a few more expressions that have not been introduced in the previous examples.

A let expression introduces declarations with limited scope; that is, the names (type names, constructor names, function names) introduced by the declarations can only be used in the body of the let expression:

```
val sixteen =
  let
    fun square (n: Integer): Integer = n * n
  in
    square (square 2)
  end
(* cannot use 'square' in later declarations or expressions. *)
```

Note that a let expression is introduced with the **`let`** keyword, followed by one or more declarations, the **`in`** keyword, followed by the let-expression body, and terminated by the **`end`** keyword.

A type-constraint expression asserts the type of an expression:

```
val three = ((id [Integer] : Integer -> Integer) (3 : Integer)
             : Integer)
```

Note that a type-constraint expression is introduced with the syntax *Exp* : *Type*. Such type assertions can be useful to understand why a program has (or does not have) type errors.

Finally, a sequence expression evaluates multiple expressions, but only returns the result of the final expression:

```
val zero =
  (print "Hello " ; print "world!\n" ; 0)
val one = inc zero
```

Note that a sequence expression is written in `(...)` separated by `;`. Evaluating these declarations first prints the string `"Hello "`, then prints string `"world!\n"`, then binds `zero` to the value `0`, and finally binds `one` to the value `1`. As a convenience, a sequence expression appearing in the body of a let expression can be written without the delimiting parentheses:

```
val nine =
  let
    fun square (n: Integer) : Integer = n * n
  in
    print "Hello " ; print "world!\n" ; square 3
  end
```

Evaluating this declaration first prints the string `"Hello "`, then prints string `"world!\n"`, then binds `nine` to the value `9`.

3 Predefined Types, Constructors, and Functions

LangF provides the following predefined types and constructors:

```
type Integer = ...
type String = ...
type Array ['a] = ...
datatype Unit = Unit
datatype Bool = True | False
datatype ['a] Option = Some {'a} | None
```

The `Integer` and `String` types correspond to the primitive types of integers and strings; we have seen many examples that use these types. The `Array` type constructor generates the type of mutable arrays.

The `Bool` datatype corresponds to the type returned by the integer-comparison expression forms and the type expected by the boolean-centric expression forms. The `Unit` datatype corresponds to a type with exactly one constructor `Unit`; it is useful as the return type of functions that should be evaluated for a side-effect, without returning a value.

LangF also provides the following predefined functions:

```
val argc       : Unit -> Integer
val arg        : Integer -> String
val print      : String -> Unit
val fail       : ['a] -> String -> 'a
val size       : String -> Integer
val sub        : String -> Integer -> Integer
val toString   : Integer -> String
val fromString : String -> Option[Integer]
val array      : ['a] -> Integer -> 'a -> Array['a]
val length     : ['a] -> Array['a] -> Integer
```

4 LangF syntax

For reference, the following is the collected syntax of LangF. We assume the following kinds of terminal symbols: type variable identifiers (*tyvarid*, with a leading `'` and lower-case letter), type constructor identifiers (*tyconid*, with a leading upper-case letter) type; data constructor identifiers (*daconid*, with a leading upper-case letter); variable identifiers (*varid*, with a leading lower-case letter); integer literals (*integer*); and string literals (*string*).

Prog
 ::= *Exp*
 | (*Decl*)* `;` *Exp*

Decl
 ::= `type` *tyconid TypeParams* `=` *Type*
 | `datatype` *DataDecl* (`and` *DataDecl*)*
 | `val` *SimplePat* (`:` *Type*)^{opt} `=` *Exp*
 | `fun` *FunDecl* (`and` *FunDecl*)*

TypeParams
 ::=
 | `[ ]`
 | `[` *tyvarid* (`,` *tyvarid*)* `]`

Type
 ::= *Type* `->` *Type*
 | `[` *tyvarid* `]` `->` *Type*
 | *tyconid TypeArgs*
 | *tyvarid*
 | `(` *Type* `)`

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

DataDecl
 ::= *tyconid TypeParams* `=` *DaConDecl* (`|` *DaConDecl*)*

DaConDecl
 ::= *daconid DaConArgTys*

DaConArgTys
 ::=
 | { }
 | { *Type* (`,` *Type*)* }

FunDecl
 ::= *varid Param*$^+$ `:` *Type* `=` *Exp*

Param
  ::= **(** *varid* **:** *Type* **)**
  | **[** *tyvarid* **]**

Exp
  ::= **let** *Decl*$^+$ **in** *Exp* (**;** *Exp*)* **end**
  | **fn** *Param*$^+$ **=>** *Exp*
  | **if** *Exp* **then** *Exp* **else** *Exp*
  | **case** *Exp* **of** *MatchRule* (**|** *MatchRule*)* **end**
  | **try** *Exp* **catch** *Exp* **end**
  | **escape** **[** *Type* **]**
  | *Exp* **orelse** *Exp*
  | *Exp* **andalso** *Exp*
  | *Exp* **:** *Type*
  | *Exp* **!** *Exp* **:=** *Exp*
  | *Exp Operator Exp*
  | **~** *Exp*
  | *daconid TypeArgs DaConArgs*
  | *Exp ApplyArg*
  | *varid*
  | **(** *Exp* (**;** *Exp*)* **)**
  | *integer*
  | *string*

MatchRule
  ::= *Pat* **=>** *Exp*

Pat
  ::= *daconid TypeArgs DaConPats*
  | *SimplePat*

DaConPats
  ::=
  | { }
  | { *SimplePat* (**,** *SimplePat*)* }

SimplePat
  ::= *varid*
  | _

DaConArgs
  ::=
  | { }
  | { *Exp* (**,** *Exp*)* }

ApplyArg
  ::= *Exp*
  | **[** *Type* **]**

Operator

::= **==** | **<>** | **<** | **<=** | **>** | **>=** | **+** | **-** | ***** | **/** | **%** | **^** | **!**