CMSC 22600	**Compilers for Computer Languages**	**Handout 1**
Autumn 2020		**Sept. 30, 2020**

Project Overview

1 Introduction

The project for the course is to implement a small functional programming language, called **LangF**. This language is an enrichment of *System F* (the polymorphic λ-calculus).

The project grade will comprise the bulk of your course grade, and will involve a significant effort to complete, so it is important that you give it sufficient time and effort.

The project consists of four parts:

1. **Scanning and Parsing**, which consists of implementing a scanner and parser for the language that builds a *parse-tree* representation of the input program.
2. **Type Checking**, which consists of implementing a type checker for the parse tree that produces a *typed abstract syntax tree* (AST).
3. **Optimizer**, which consists of implementing a conversion from the typed AST to a *normalized* representation that is suitable for optimization, and the implementation of some simple optimizations.
4. **Optimization and Code Generation**, which consists of implementing a closure converter that lowers the higher-order program to a first-order representation and a code generator that produces LLVM assembly code.

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.

You will have two weeks to complete each project (not counting Thanksgiving break). We will grade whatever code you have submitted at the deadline; late assignments will **not** be accepted, so do not leave your work to the last minute.

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 Haskell), but with many simplifications and a more explicit type system. **LangF** does not have type inference, exceptions, or a module system, but it does include mutable references. Furthermore, **LangF** has first-class functions, datatypes, and first-class polymorphism.

2.1 Types

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

Type
::= *TypeParams Type*
| *Type* `->` *Type*
| *Type* (`*` *Type*)$^{+}$
| tyid *TypeArgs*^{opt}
| tyvar
| `(` *Type* `)`

TypeParams
::= `[` tyvar (`,` tyvar)* `]`

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

LangF 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 `->` constructor associates to the right and has lower precedence than the tuple-type constructor (`*`), while type abstraction has the lowest precedence. Some examples:

```
Int -> Int
[a] a -> a
([a] a -> String) -> [a] List[a] -> String
Int * Int -> Bool
```

In addition to `Int` and `String`, **LangF** predefines several data types, such as `Bool`, `List`, and `Unit`. These will be discussed in the Project 2 description.

2.2 Programs

A **LangF** program is a sequence of top-level definitions.

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

The convention is that the last definition is a function named `main` that has the type

```
List[String] -> Int
```

For example, here is the classic "hello world" program in **LangF**:

```
fun main (args : List[String]) -> Int = { print "hello world\n"; 0 }
```

Executing a **LangF** program means evaluating each of the declarations (making their definitions available to the subsequent declarations and expression) and then applying the `main` function to

the command-line arguments. The result of the program (an integer) is used as the exit status; traditionally a non-zero value is used to signal an error.

Here is a slightly more interesting program that computes 5! by defining the `fact` function followed by the expression `fact 5`, which is returned as the exit status.

```
// compute factorial of n
fun fact (n : Int) -> Int =
      if n == 0 then 1 else n * fact (n - 1);
/* main function */
fun main (args : List[String]) -> Int = fact 5
```

Note that comments are either single-line comments that start with // or are block comments bracketed by /* and */. Block comments may be nested (as in SML).

2.3 Top-level definitions

There two kinds of top-level definitions in **LangF**: 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** tyid *TypeParams*^{opt} **=** *ConDef* (**|** *ConDef*)*
| *ValBind*

ConDef
::= conid (**of** *Type*)^{opt}

ValBind
::= **fun** varid *FunParam*$^{+}$ **->** *Type* **=** *Exp*
| **let** *SimplePat* (**:** *Type*)^{opt} **=** *Exp*
| *Exp*

FunParam
::= *TypeParams*
| **(** varid **:** *Type* **)**

We describe these various forms below.

2.3.1 Simple type definitions

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 a curried integer comparison function (a function from two integers to a boolean):

```
type IntCmp = Int -> Int -> 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 **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 [Int]
```

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:

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

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 **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 one for defining the relationship between values in a total order:

```
data Order = Less | Equal | 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):

```
data Publication
  = Book of String * String
  | Article of String * String * String
```

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:

```
data Pair [a, b] = Pair of a * b
```

2.3.3 Function definitions

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

```
fun length [a] (xs : List[a]) -> Int =
      case xs of
```

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

A defining characteristic of **LangF** (taken from *System F*) 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:

```
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 **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 `Int ->Int`:

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

Note that the polymorphic function type is written using the syntax

`[` 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:

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

Project 2 will discuss this aspect in more detail.

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.

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

2.3.4 Value definitions

In addition to function definitions, **LangF** 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

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.

2.4.1 Conditionals

LangF 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.

LangF also has two infix conditional operators: "`||`" and "`&&`".

2.4.2 Binary expressions

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

Operator	Associativity	Description
`:=`	Left	Assignment
`||`	Left	Conditional "or-else"
`&&`	Left	Conditional "and-also"
`==`	Left	Integer equality relation
`!=`	Left	Integer inequality relation
`<`	Left	Integer less-than relation
`<=`	Left	Integer less-than-or-equal relation
`::`	Right	List cons operator
`^`	Left	String concatenation operator
`+`	Left	Integer addition operator
`-`	Left	Integer subtraction operator
`*`	Left	Integer multiplication operator
`/`	Left	Integer division operator
`%`	Left	Integer modulo operator

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

[1] Expressions can be thought of as a degenerate form of value binding.

2.4.3 Unary expressions

LangF has two unary operators: negation ("`-`") and dereferencing ("`!`"). Unary expressions bind more tightly than binary operators, but less than function application (*e.g.*, the expression "`- f x`" is parsed as "`- (f x)`").

2.4.4 Application

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

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

2.4.5 Variables and constants

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

2.4.6 Tuple expressions

LangF 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.

2.4.7 Blocks

A block introduces a nested scope that can include function and value bindings. The

{ (*ValBind* `;`)$^+$ *Exp* }

LangF 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.

2.4.8 Case expressions

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

```
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.

2.4.9 References

Similar to SML, **LangF** provides mutable references. The builtin function `newRef` creates a reference cell, the `:=` operator can be used to assign to a reference, and `!` is used to extract a reference's value.

For example, here is an imperative implementation of the factorial function:

```
fun fact (n : Int) -> Int = {
      let res : Ref[Int] = newRef [Int] 1;
      fun lp (i : Int) -> Unit = if (i < n)
            then { res := !res * i; lp (i+1) }
            else ();
      lp (2);
      !res
    }
```

There are some subtleties with respect to the interaction between references and polymorphism. For example, consider the following expression:

```
{ let r : [a] Ref[List[a]] = newRef [a] (Nil [a]);
  r[Bool] := true :: !r[Bool];
  r[Int] := 1 :: !r[Int];
  r
}
```

As will be discussed in Project 2, this program type checks, which might lead one believe that it has a runtime type error (*i.e.*, mixing booleans and integers in the same list). In fact, it does not, because the *type abstraction* in the binding of `r` behaves like a λ-abstraction at runtime. Thus, the expressions "`r [Bool]`" and "`r [Int]`" evaluate to **different** reference cells.

The semantically equivalent program in SML would be

```
let r = fn () => ref nil
in
  r() := true :: !(r());
  r() := 1 :: !(r());
  r
end
```

where we have used unit for the type parameters and arguments.

3 Project schedule

Important note: You are expected to submit code that **compiles** and that is well documented. 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.

The tentative schedule for the project assignments is as follows:

Assigned	Project description	Due date
October 7	Parser & Scanner	Tuesday, October 20
October 21	Type Checker	Tuesday, November 3
November 4	Optimizer	Tuesday, November 17
November 18	Code Generator	Tuesday, December 8

All project assignments will be due at 23:59 Chicago Time. Late assignments are **not** accepted, so do not leave your work to the last minute.

Document history

October 18, 2020 Added integer inequality (`!=`) to the binary-operator table.

October 18, 2020 Fixed inconsistent use of valid (should be varid).

October 13, 2020 Fixed syntax of programs to require a semicolon between definitions.

October 5, 2020 Changed string concatenation operator to "`^`" and added missing conditional operators to precedence table.

October 1, 2020 Fixed grammar rule for *Program* non-terminal and added hello-world example.

September 30, 2020 Original version