CMSC 22600 **Autumn 2016**	**Compilers for Computer Languages**	**Handout 1** **October 6, 2016**

Project Overview

1 Introduction

The project for the course is to implement a small object-oriented programming language, called SOOL. The project will consist of four parts:

1. the *parser and scanner*, which coverts a textual representation of the program into a *parse tree* representation.
2. the *type checker*, which checks the parse tree for type correctness and produces a *typed abstract syntax tree* (AST)
3. the *normalizer*, which converts the typed AST representation into a static single-assignment (SSA) representation.
4. the *code generator*, which takes the SSA representation and generates 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.

> **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.

2 A quick introduction to SOOL

SOOL is a statically-typed object-oriented language. It supports single inheritance with nominal subtyping and interfaces with structural subtyping.

The SOOL version of the classic *Hello World* program is

```
class main () {
  meth run () -> void { system.print ("hello world\n"); }
}
```

Here we are using the predefined `system` object to print the message. Every SOOL program has a `main` class that must have a `run` function. The SOOL runtime system will execute the program by essentially executing the statement

```
main().run();
```

which creates an instance of the `main` class and then invokes its `run` function.

2.1 Primitive types and operations

While SOOL is an object-oriented language, it also has a small collection of primitive types, values, and operations. The types are **`bool`**, with values **`true`** and **`false`**; **`int`**, with decimal integer literals; and strings of 8-bit characters. The primitive operators include integer comparisons (`<` and `<=`), equality (`==` and `!=`), integer arithmetic (`+`, `-`, `*`, `/`, and unary `-`), and string concatenation (`@`).

2.2 Classes

SOOL is a class-based language; classes serve three very important rôles in the language:

1. they provide containers for program state
2. they host the computational code of the program
3. they define types

As an example, let us consider the representation of 2D points in SOOL. We might start with a simple class that contains two integer-valued *member variables* (or *fields*) (`x` and `y`):

```
class point () {
  var x : int = 0
  var y : int = 0
}
```

Member variables must be given an initial value, so we set them to zero.

This definition is not very useful, however, because while we can create new point values using the expression "`point()`," there is very little useful computation that we can do with them, since `x` and `y` are not visible outside the class (or its subclasses). We can address this restriction by adding a `move` member function (or *method*) that allows one to change the position of a point

```
meth move (dx : int, dy : int) -> point {
  self.x := self.x + dx;
  self.y := self.y + dy;
  return self;
}
```

and we add two functions to allow other classes to read the point's coordinates:

```
  meth x () -> int { return self.x; }
  meth y () -> int { return self.y; }
```

Although it may appear that we have two definitions of the names `x` and `y`, they are in different namespaces (variables versus functions) and thus cannot be confused. Notice also that we use the keyword **`self`** to access the variable values and to return the object as the result of `move`. Class-member variables are not visible outside of the class (or its subclasses), while member functions are always visible.[1]

We can now write some code that uses these features. The following expression creates a point object, then calls the object's `move` function, which returns the object, and then calls the `y` function:

```
point().move(17, 42).y()  // returns 42
```

If we want to be able to create points positioned at somewhere other than the origin, we can add parameters to the class:

```
class point (x : int, y : int) {
  var x : int = x
  var y : int = y
  ...
}
```

We will now need to supply two integer arguments when creating new objects (*e.g.*, `point (17, 42)`).

As mentioned above, variables can only be accessed by the class's functions, but it is legal for a function to access the variables of *other* objects of the same class. For example, we might add an `add` function to our code, which takes another `point` object and uses its coordinates to move the function's position:

```
  meth add (pt : point) -> point {
    return self.move (pt.x, pt.y);
  }
```

Putting these pieces all together gives us the following complete class definition for 2D points:

```
class point (x : int, y : int) {
  var x : int = x
  var y : int = y
  meth x () -> int { return self.x; }
  meth y () -> int { return self.y; }
  meth move (dx : int, dy : int) -> point {
    self.x := self.x + dx;
    self.y := self.y + dy;
    return self;
  }
  meth add (pt : point) -> point {
    return self.move (pt.x, pt.y);
  }
}
```

[1] In C++ teminology, functions are *public* and variables are *protected*.

The variable definitions of a class must come before the functions and they may not use the class's functions in their definitions. The functions may be given in any order, however, and may be mutually recursive.

2.3 Inheritance

We can extend the definition of points by adding a variable that holds the point's color (represented as a string):

```
class colorPoint (x : int, y : int, c : string)
  extends point (x, y)
{
  var c : string = c
  meth color () -> string { return self.c; }
}
```

The `colorPoint` class *inherits* the variables and functions from the `point` class and so, for example, we can write the expression

```
colorPoint(17, 42, "red").y()  // returns 42
```

In addition to defining `colorPoint` as an extension of the `point` class, the above definition establishes a *nominal subtyping relationship* between the types of objects generated by the two classes. Recall that class names also serve as types in SOOL. We say that a value of type `colorPoint` is a *subtype* of `point`. If a value has type σ and σ is a subtype of τ, then we can use the value in any context that expects a value of type τ. For example, we can pass a `colorPoint` to the `add` function.

```
point(0,0).add(colorPoint(17, 42, "red")).y()  // returns 42
```

2.4 Overriding functions

A subclass is allowed to replace the implementation of functions that it inherits with its own code. This mechanism is called *function overriding*. For example, we might define a subclass of points that move more quickly than regular points:

```
class fastPoint (x : int, y : int)
  extends point (x, y)
{
  override meth move (dx : int, dy : int) -> point {
    self.x := self.x + 2 * dx;
    self.y := self.y + 2 * dy;
    return self;
  }
}
```

Function dispatch is a *dynamic* mechanism, so if we use a `fastPoint`'s `add` function, it will use the overriden definition of the `move` function.

```
fastPoint(0,0).add(point(17, 42)).y()  // returns 84
```

2.5 Interfaces

In addition to the nominal subtyping introduced by inheritance, SOOL also supports a form of *structural subtyping* based on *interfaces*. For example, both `point` and `colorPoint` objects match the following interface:

```
interface pointI {
  meth x () -> int
  meth y () -> int
  meth move (int, int) -> void
}
```

For example, the type of the `system.print` function is

```
meth print : toStringI -> void
```

where `toStringI` is the following interface:

```
interface toStringI {
  meth toString () -> string
}
```

This means that we can use `system.print` on any object generated by a class that implements a `toString` function with the appropriate type.

2.6 Options

In addition to builtin and user-defined class types, SOOL has one type constructor for representing optional values (similar to the `'a option` in SML). If T is a primitive type, class, or interface, then "T**?**" is the type of optional values of type T. These values include all of the values of type T plus the special constant "**nil**," which represents no value. Unlike the `'a option` type constructor, optional types in SOOL are not first-class; *i.e.*, there can only be a single layer of options.

There are six expression forms to support working with optional values. The first two are variations of member-variable selection; the third and fourth are variations of function dispatch on optional arguments, and the last allows the option to be stripped off an optional value after a runtime check.

- The expression "**nil** T" is the **nil** value with type "T**?**."
- The expression "*Exp***?**v" evaluates the expression *Exp* to an optional object value; if the value is **nil**, then the result is **nil**. Otherwise, the result is the contents of the member variable "v."
- The expression "*Exp***!**v" evaluates the expression *Exp* to an optional object value; if the value is **nil**, then a runtime error is signaled. Otherwise, the result is the contents of the member variable "v."

- The expression "*Exp*`?`f`(` ... `)`" evaluates the expression *Exp* to an optional object value; if the value is **nil**, then the function is *not* invoked and the whole expression returns **nil**. Otherwise, if *Exp* evaluates to an object, then the function f is invoked on the object and its result is returned. If the result type of the function is **void**, then the type of the whole expression is **void**.
- The expression "*Exp*`!`f`(` ... `)`" evaluates the expression *Exp* to an optional object value; if the value is **nil**, then a runtime error is signaled. Otherwise, if *Exp* evaluates to an object, then the function f is invoked on the object and its result is returned.
- The expression "*Exp*`!`" evaluates the expression *Exp* to an optional value. If the result is **nil**, then there is a runtime error, otherwise the value is returned.

Here is an example of the use of optional values to implement a binary search tree class.

```
// binary search tree
class BST (v : int) {
  var value : int = v         // node's value
  var left : BST? = nil BST   // left child
  var right : BST? = nil BST // right child
  // test if 'x' is in the tree
  meth member (int x) {
    if (self.value == x) {
      return true;
    }
    else if (x < self.value) {
      return (self.left != nil BST) && self.left!member(x);
    }
    else {
      return (self.right != nil BST) && self.right!member(x);
    }
  // insert 'x' into the tree
  meth insert (int x) {
    if (self.value != x) {
      if (x < self.value) {
        if (self.left == nil BST) { self.left = BST(x); }
        else { self.left!insert(x); }
      } else {
        if (self.right == nil BST) { self.right = BST(x); }
        else { self.right!insert(x); }
      }
    }
  }
}
```

2.7 Recursion

We have already noted that the methods in a class definition may be mutually recursive. SOOL also allows mutual recursion between class and interface declarations. Specifically, the declarations that make up a SOOL program are all considered to be mutually recursive with the restriction that super classes must be declared before subclasses (*i.e.*, the inheritance hierarchy must be acyclic).

The following example illustrates the use of recursion to implement down-cast methods. It is an example of a common way to implement datatypes in object oriented languages by defining a base class for the type and subclasses for the constructors.

```
// base class of trees
class BT () {
  meth asNode () -> BTNode? { return nil BTNode; }
  meth asLeaf () -> BTLeaf? { return nil BTLeaf; }
}
// leaves
class BTLeaf extends BT() {
  override meth asLeaf () -> BTLeaf? { return self; }
}
// interior nodes
class BTNode(x : int) extends BT() {
  var value : int = x
  var left : BT = BTLeaf()
  var right : BT = BTLeaf()
  override meth asNode () -> BTNode? { return self; }
  ...
}
```

3 The SOOL basis

There are a small set of pre-defined interfaces defined in the SOOL basis, as well as one predefined global variable.

3.1 The `toStringI` interface

As mentioned above, the `toStringI` interface describes objects that provide a `toString` method for rendering their value as a string.

```
interface toStringI {
  // return the string representation of the object
  meth toString () -> string
}
```

3.2 The `systemI` interface

The `systemI` interface is the type of the special global variable `system`, which is used to access system services. For SOOL, these are printing, exiting the program, and signaling a runtime error.

```
interface systemI {
  // 'system.print(obj)' prints 'obj.toString()'
  meth print (toStringI) -> void
  // 'system.input()' reads a line of text from the program's input stream.
  // It returns nil on end-of-file.
  meth input () -> string?
  // exit the program
  meth exit () -> void
  // signal failure with the given message
  meth fail (string) -> void
}

// global system object
var system : systemI;
```

3.3 Interfaces for primitive types

For each of the primitive types (**bool**, **int**, and **string**), there is a corresponding predefined interface that the type is a subtype of (*e.g.*, **bool** is a subtype of boolI). All of these interfaces extend the toStringI interface with additional operations.

3.3.1 The boolI interface

The boolI interface adds a logical negation operator on booleans.

```
// bool <: boolI
interface boolI extends toStringI {
  // logical negation
  meth not () -> bool
}
```

3.3.2 The intI interface

The intI interface adds a function for converting character codes to length-one strings.

```
// int <: intI
interface intI extends toStringI {
  // for values from 1..255, return the corresponding length-one string
  meth char () -> string
}
```

3.3.3 The stringI interface

Although there is not a string class in SOOL, the **string** type is a subtype of the stringI interface type, which has the following definition:

```
// string <: stringI
interface stringI extends toStringI {
  // return length of string
  meth length () -> int
  // substring(i, len) returns the substring from i to i+n-1
  meth substring (int, int) -> string
  // return the integer value of the character at the given position
  meth charAt (int) -> int
  // convert a string representing a number
  meth toInt () -> int?
}
```

4 Project schedule

The tentative schedule for the project assignments is as follows:

Assigned	Project description	Due date
October 6	Parser & Scanner	Friday, October 21
October 20	Type checker	Friday, November 4
November 3	Normalizer	Friday, November 18
November 17	Code generator	Monday, December 5

All project assignments will be due at 11pm.

Document history

October 23, 2016 Fixed a couple of inconsistencies in the `point` example.

October 12, 2016 Add `input` method to `systemI` interface.

October 9, 2016 Add type constructor to **`nil`** expression to simplify Project 2. Also fixed a couple of typos.

October 7, 2016 Fix syntax error the BST example

October 6, 2016 Original version