Advanced Programming (I00032)
Semantics of Simple Functional Language

Assignment 9

Semantics of a first-order strict functional language

In this exercise you develop and test a semantics for a very simple functional programming language. This language is first order: no higher order functions. The only data types in this language are integers and booleans. These data types are hard-wired into the language. We assume that all functions are stored in the environment. Only the semantics of expressions with function applications has to be defined.

1 Language

As explained in the lecture, the language is represented by a family of data structures.

```
:: Var     = VI Ident
:: Fun     = FI Ident
:: Ident :== String

:: Expr    = Int   Int
           | Bool  Bool
           | Fun   Fun
           | Var   Var
           | Ap    Expr [Expr]
           | Infix Expr Prim Expr
           | Prim  Prim
:: Prim    = IF | +. | *. | -. | <. | NOT

:: Def     = Def Ident [Var] Expr
```

Note that the handling of primitive functions is slightly different from the primitive operations shown in the lecture. Also note that these data types are slightly more general than required for a first-order language. By using `Ap Expr [Expr]` rather than `Ap Fun [Expr]`, the expressions type can handle currying.

In this representation function definitions look like:

```
Def "id"  ["x"] (Var "x")
Def "inc" ["x"] (Infix (Var "x") +. (Int 1))
```

A simple expression that should obtain a semantics is

```
(Ap (Fun "id") [Int 7])
```

This should have the value `Int 7` as semantics.

2 Environments

The above definitions show that this language has two distinctive name spaces, one for functions and one for function arguments. Consequently, the semantics also has two environments, one for each name space. Fortunately, we can share all code for new environments, bindings and lookup:

```
:: State := Env Expr
:: Funs  := Env Def
:: Env e := Ident → e

(↦) infix 9 :: Ident v → (Env v) → (Env v)
(↦) v e = λenv x.if (x == v) e (env x)

newEnv :: Env e
newEnv = λv.abort ("No binding for " + v)
```

These environments are very similar to the one used in the course. The value of an identifier can be obtained by applying the environment (as a function) to the identifier. We choose to yield a run-time error for undefined identifiers rather than a value like `zero`.

3 Semantics for expressions

On Blackboard you find a skeleton file, `funSemSkeleton.icl`. This skeleton file contains a dummy semantics which does nothing at all. Extend this semantics to handle well-typed first-order strict functions.

```
E :: Expr State Funs → Expr
E e vf fs = e
```

Note that the semantics of a functional language yields an expression (in normal form), whereas the semantics of an imperative expression yields a state.

The fact that you only have to assign a semantics to well typed programs implies that ill typed programs do not need a semantics. Anything your semantics does for incorrect expressions is fine. Nevertheless it can be convenient for yourself during the development of the semantics to give some appropriate error message (perhaps by an `abort`) if an unexpected syntax tree is encountered.

The fact that the language is first order tells you that there is a strict separation between variables and functions. Moreover there is no currying. A function is always applied to the correct number of arguments.

Because of the strict semantics, function arguments are evaluated before the function body is evaluated. Although lazy semantics seems to be just as easy, it is slightly more complicated because you need to have the right environment available to evaluate the expression later on when it is actually needed. Note that a new function application can have formal parameters with the same name as existing identifiers and hence spoil the bindings during lazy evaluation.

4 Unit Tests

The provided skeleton contains some simple unit tests to give you a start. To the best of the author's knowledge a proper semantics should pass all given tests. Only the test that is given as a comment requires higher order functions. Your semantics might assign any value to this expression. It is just included to show that the data type is capable to handle those kind of expressions.

5 A functional language

The next step gives a semantics to function programs. Such a program consists of a list of declarations and evaluates the `Start` function.

```
Ds :: [Def] → Expr
Ds defs = abort "Ds not properly defined"
```

6 Model-based Testing

Try to state at least two non-trivial properties about the semantics and test them with G∀st. Most likely you have to change the given generic generation, `ggen`, of expressions and identifiers in order to ensure that only valid programs are generated.

A very simple property is for instance:

```
p1 :: Int Int → Bool  // equivalence of prefix/infix and (+) i j = j+i
p1 i j
 =     E (Ap (Prim PLUS) [Int i, Int j]) newEnv newEnv
   === E (Infix (Int j) PLUS (Int i))     newEnv newEnv
```

Of course we can state similar properties for the other primitive binary operators, but those are not very interesting. A slightly more interesting property states a similar property but quantifies over binary operators.

Deadline

The deadline for this exercise is November 25, 23:59h. Hand in your semantic description and the test report generated by G∀st.