Concepts of Programming Languages

Course: CSE2120 Edition: 2019-2020

Available from January 26, 2020 until July 3, 2020

Your Enrollment

You're taking this course for a grade

Your Course Dossier

Your Submissions

Unenroll

Course Information

Home
All editions
News archive
Course rules
Lecture notes
Assignments

Course staff

Lecturers
- Casper Poulsen
- Eelco Visser

Assistants
- Ali Al-Kaswan
- Yana Angelova
- Wesley Baartman
- Kirti Bihanie
- Philippos Boon Alexaki
- Luc Everse
- Boris Janssen
- Rembrandt Klazinga
- Mirco Kroon
- Chris Lemaire
- Sterre Lutz
- Yaniv Oren
- Wouter Polet
- Thijs Raijmakers
- Jim van Vliet
- Yoshi van den Akker
- Paul van der Stel
- Eric van der Toorn

1.4. Week 4: Func + Env

In this week's assignments, you will implement an interpreter that uses *environments* to delay substitutions. The Paret language has also been extended with a new construct `rec-lam` for recursive functions. Your interpreter should be *eager* (also known as *call-by-value* or *strict*), and should use left-to-right evaluation order.

This assignment is based on Chapter 6 in PLAI.

1 Features to Implement

1.1 Environment-Based Interpreter

An environment should be used in the interpreter to keep track of current values of identifiers in scope. An `Environment` is a list of `Bind` objects, representing a binding of a name (`String`) to a `Value` (see the classes below).

The interpreter should allow identifier shadowing, meaning that a new binding takes precedence if an identifier with the same name is already bound.

DO NOT USE SUBSTITUTION

Although the substitution-based interpreter from week 3 will pass this lab, your solution will be manually checked to see if you are using environments rather than substitution. If you use substitution, you will fail the assignment, and thus the lab.

1.2 Multi-Argument Functions

Functions should declare a sequence of parameters; e.g.:

```
(lambda (x1 ... xn) e)
```

You should also support functions with an empty sequence of parameters.

Function application should accept a sequence of arguments; e.g.:

```
(e e1 ... en)
```

Each function should always be fully applied. That is, in the example above, `e` should evaluate to a function `f` that takes a `n` parameters. Otherwise, the interpreter should fail by throwing an appropriate exception.

1.3 Recursive Lambdas

`rec-lam` is a new piece of syntactic sugar for recursive functions. It is very similar to a `lambda` , except that it also takes a function name which can be used in its body. More precisely,

```
(rec-lam f (v) b)
```

constructs a function with name `f` , argument `v` , and body `b` , where the function name `f` is bound inside the body.

For example, we can define a function `sum` that sums the numbers from `0` to `n` by writing:

```
(rec-lam sum (n)
  (if (num= n 0)
    0
    (+ n (sum (- n 1)))))
```

The above simply defines the function. A `rec-lam` function can be passed around as a value and applied like any other function. For example, we can apply the `sum` function to 3:

```
((rec-lam sum (n)
   (if (num= n 0)
     0
     (+ n (sum (- n 1)))))
 3)
```

This should evaluate to 3+2+1 = 6.

You can also write infinite loops using `rec-lam` :

```
((rec-lam forever (x) (forever x)) 0)
```

This week's programming assignment asks you to implement recursive functions using the Y combinator (for call-by-value languages, the combinator is sometimes known as the Z combinator). `rec-lam` can be defined by desugaring into an application of the Y combinator. For example, the `sum` function above can be desugared as follows (where `<Y>` is a placeholder for the eager equivalent of the Y combinator):

```
(<Y>
  (lambda (sum)
    (lambda (n)
      (if (num= n 0)
        0
        (+ n (sum (- n 1)))))))
```

Implement this desugaring.

`rec-lam` can only be used to define recursive functions with a single parameter. Your parser should reject `rec-lam` expressions with multiple parameters, like `(rec-lam bad (x y) ...)` . Think about what it would take to extend `rec-lam` to support multiple parameters. (You do not have to implement such an extension.)

2 Grammar

```
module functions

imports Common

context-free syntax

  Expr.NumExt       = INT      // integer literals
  Expr.TrueExt      = [true]
  Expr.FalseExt     = [false]
  Expr.IdExt        = ID

  Expr.UnOpExt      = [([UnOp] [Expr])]
  Expr.BinOpExt     = [([BinOp] [Expr] [Expr])]

  UnOp.MIN          = [-]
  UnOp.NOT          = [not]
  UnOp.HEAD         = [head]
  UnOp.TAIL         = [tail]
  UnOp.ISNIL        = [is-nil]
  UnOp.ISLIST       = [is-list]

  BinOp.PLUS        = [+]
  BinOp.MULT        = [*]
  BinOp.MINUS       = [-]
  BinOp.AND         = [and]
  BinOp.OR          = [or]
  BinOp.NUMEQ       = [num=]
  BinOp.NUMLT       = [num<]
  BinOp.NUMGT       = [num>]
  BinOp.CONS        = [cons]

  Expr.IfExt        = [(if [Expr] [Expr] [Expr])]
  Expr.NilExt       = [nil]
  Expr.ListExt      = [(list [Expr*])]

  Expr.FdExt        = [(lambda ([ID*]) [Expr])]
  Expr.AppExt       = [([Expr] [Expr*])]
  Expr.LetExt       = [(let ([LetBind+]) [Expr])]
  LetBind.LetBind   = [([ID] [Expr])]

  Expr.RecLamExt    = [(rec-lam [ID] ([ID]) [Expr])]
```

Note that `[ID*]` denotes a sequence of zero or more `[ID]` s, and `[LetBind+]` denotes one or more `[LetBind]` s.

3 Classes

These classes should be used in your solution.

3.1 Abstract Syntax

The abstract syntax is postfixed with `Ext` for extended syntax.

```
sealed abstract class ExprExt
case class TrueExt() extends ExprExt
case class FalseExt() extends ExprExt
case class NumExt(num: Int) extends ExprExt
case class BinOpExt(s: String, l: ExprExt, r: ExprExt) extends ExprExt
case class UnOpExt(s: String, e: ExprExt) extends ExprExt
case class IfExt(c: ExprExt, t: ExprExt, e: ExprExt) extends ExprExt
case class ListExt(l: List[ExprExt]) extends ExprExt
case class NilExt() extends ExprExt
case class AppExt(f: ExprExt, args: List[ExprExt]) extends ExprExt
case class IdExt(c: String) extends ExprExt
case class FdExt(params: List[String], body: ExprExt) extends ExprExt
case class LetExt(binds: List[LetBindExt], body: ExprExt) extends ExprExt
case class RecLamExt(name: String,
                     param: String,
                     body: ExprExt) extends ExprExt

case class LetBindExt(name: String, value: ExprExt)

object ExprExt {
  val binOps = Set("+", "*", "-", "and", "or", "num=", "num<", "num>",
    "cons")
  val unOps = Set("-", "not", "head", "tail", "is-nil", "is-list")
  val reservedWords = binOps ++ unOps ++ Set("list", "nil", "if", "lambda",
    "let", "true", "false", "rec-lam")
}
```

3.2 Desugared Syntax

The desugared syntax is postfixed with `C` for core syntax.

```
sealed abstract class ExprC
case class TrueC() extends ExprC
case class FalseC() extends ExprC
case class NumC(num: Int) extends ExprC
case class PlusC(l: ExprC, r: ExprC) extends ExprC
case class MultC(l: ExprC, r: ExprC) extends ExprC
case class IfC(c: ExprC, t: ExprC, e: ExprC) extends ExprC
case class EqNumC(l: ExprC, r: ExprC) extends ExprC
case class LtCC(l: ExprC, r: ExprC) extends ExprC
case class NilC() extends ExprC
case class ConsC(l: ExprC, r: ExprC) extends ExprC
case class HeadC(e: ExprC) extends ExprC
case class TailC(e: ExprC) extends ExprC
case class IsNilC(e: ExprC) extends ExprC
case class IsListC(e: ExprC) extends ExprC
case class UndefinedC() extends ExprC
case class AppC(f: ExprC, args: List[ExprC]) extends ExprC
case class IdC(c: String) extends ExprC
case class FdC(params: List[String], body: ExprC) extends ExprC
```

3.3 Values

```
sealed abstract class Value
case class NumV(n: Int) extends Value
case class BoolV(b: Boolean) extends Value
case class NilV() extends Value
case class ConsV(hd: Value, tl: Value) extends Value
case class ClosV(f: FdC, env: Environment) extends Value
```

3.4 Other

```
case class Bind(name: String, value: Value)
type Environment = List[Bind]
```

4 Exceptions

Specific exceptions should be created that *inherit* from the given abstract exceptions.
Creating specific exceptions for each case makes debugging a lot easier and are more informative.
Throw only exceptions derived from `DesugarException` in the desugarer and only exceptions derived from `InterpException` in the interpreter.

```
abstract class DesugarException extends RuntimeException
abstract class InterpException  extends RuntimeException
```