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

1.3. Week 3: Func + Subst

In this week we extend the Paret language with multi-argument higher-order functions and `let` expressions.

You will implement an interpreter that uses *substitution*. Your interpreter should be *eager* (also known as *call-by-value* or *strict*), and it should use left-to-right evaluation order.

1 Features to Implement

The interpreter in this week extends the language from the previous week. Thus you can reuse parts of your interpreter from the previous week's lab assignment. If your interpreter was not perfect last week, try to improve that this week.

Below we summarize the new features you need to implement.

1.1 Multi-Argument Substitution

Your interpreter should implement an auxiliary function `subst` for performing substitution.
Your `subst` function should take as input:

1. an `ExprC` in which substitution needs to happen;
2. a list of `Bind`ings (defined below) where each binding is given by an identifier `String` and a `Value`.

`subst` should return the `ExprC` expression resulting from applying the substitution given by the list of bindings inside the input expression.
You should use the `ValC` class given below to represent values inside `ExprC` expressions.

Your substitution function should only traverse the input term *once*.

1.2 Multi-Argument Functions

Implement an interpreter with multi-argument `lambda`s (i.e., higher-order functions) and multi-argument lambda application by using your `subst` function.

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)
```

Function application should:
1. evaluate the first expression `e` to a function value `f`;
2. evaluate each argument expression `e1`, `e2`, ..., `en` to a list of argument values `v1`, `v2`, ..., `vn`;
3. zip the argument names of `f` with the argument values into a list of `Bind`ings; and
4. evaluate the expression resulting from `subst`ituting the argument bindings in the body of `f`.

It is undefined behavior to apply a function that expects `m` arguments to a list of `n` argument values when `m` is different from `n`.

Note: As summarized in the book, `lambda`s should be *capture-avoiding*.

1.3 Multi-Binder `let` Expressions

Paret is extended with `let` as syntactic sugar.
`let` should accept a list of one or more *binder expressions*, where a binder expression is a pair `(x e)` of an identifier `x` and an expression `e`. For example, the following expression should evaluate to 3: `(let ((x 1) (y 2)) (+ x y))`.

Your implementation of multi-binder `let` expressions should behave similarly to the single-binder `let` expressions described in the book, and should not support recursive definitions such as `(let ((ones (cons 1 ones))) ones)`. In other words, in `(let ((<id> <expr>)) <body>)`, `<id>` should be bound in `<body>`, but not in `<expr>`. In `(let ((<id1> <expr1>) (<id2> <expr2>)) <body>)`, `<id1>` should be bound in `<body>`, but not in `<expr1>` or `<expr2>`.

1.4 Reserved Words

Your parser should not allow operators such as `+` and keywords of the language such as `not`, `if`, and `lambda` to be used as identifiers. See the list of reserved words below.

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.CondExt  = [(cond [Branch+])]
  Expr.CondEExt = [(cond [Branch+] (else [Expr]))]
  Branch.Branch = [([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])]
```

Note that `[ID*]` denotes a sequence of zero or more `[ID]`s; and `[Expr+]` denotes one or more `[Expr]`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 CondExt(cs: List[(ExprExt, ExprExt)]) extends ExprExt
case class CondEExt(cs: List[(ExprExt, ExprExt)], e: ExprExt) 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 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")
}
```

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 LtC(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

case class ValC(v: Value) extends ExprC // note: no corresponding surface syntax
```

3.3 Values

```
sealed abstract class Value
case class NumV(v: Int) extends Value
case class BoolV(v: Boolean) extends Value
case class NilV() extends Value
case class ConsV(head: Value, tail: Value) extends Value
case class FunV(f: FdC) extends Value
```

3.4 Other

A binding is a pair of a name and a value:

```
case class Bind(name: String, value: Value)
```

4 Exceptions

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

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