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 Biharie
- 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

1.5. Week 5: Mutation

In this week's assignments you will add constructs for state and recursion to the language. You will add boxes, allow variable mutation, and extend the language with syntactic sugar for recursion. The approach for implementing state and recursion is discussed in Chapter 8 and Chapter 9 of the book.

1 New Features

The grammar and the abstract syntax have been extended with `box`, `unbox`, and `setbox` for boxes, with `set` for variables, with `seq` for sequential composition of mutation operations,and with `letrec` for recursion as syntactic sugar.

Boxes and variable mutation are two different ways to do mutation in the language. Make sure you understand both: try to write a program that uses boxes and then rewrite it to use variables.

Notes:

- Assume left-to-right evaluation order.
- Memory locations should start at zero, like how memory is managed in most systems.

1.1 Boxes

See Chapter 8 of the book for the implementation details of `box`, `unbox`, `setbox` and `seq`. Note that `(setbox e1 e2)` should evaluate to the value of `e2`.

1.2 `set`

`set` takes an identifier and an expression, evaluates the expression to a value, and assigns that value to the identifier by updating the appropriate store location. The result of a `(set x e)` expression is the value of `e`. For instance, the Paret program

```
(let ((x 1)) (set x 2))
```

should evaluate to 2.

If the identifier `x` in a `(set x e)` expression is not in scope, you should raise an exception.

1.3 `letrec`

The language should also be extended with `letrec` for recursive `let` bindings.
Like `let`, `letrec` should support multiple binders, where each binding is surrounded by parentheses; e.g.:

```
(letrec ((x 1) (y 2)) (+ x y))
```

A `letrec` can be desugared into a `let` using the "create, update, use" approach described in the book; e.g.:

```
(letrec ((sum
          (lambda (n)
            (if (num= n 0)
              0
              (+ n (sum (- n 1)))))))
  (sum 3))
```

can desugar into:

```
(let ((sum <dummy>))
  (seq
    (set sum
      (lambda (n)
        (if (num= n 0)
          0
          (+ n (sum (- n 1))))))
    (sum 3)))
```

There are two issues with this desugaring:

1. We need a suitable `<dummy>` value.
   As discussed in the book, a good choice of value is one that can never be used in a meaningful context.
   The classes below include an `UninitializedC` core construct and `UninitializedV` value that do not exist in the surface language.
   You should use `UninitializedC` as dummy value in your desugaring of `letrec`.
2. The desugaring involves a `let`, which is itself syntactic sugar.
   You can get around this by directly desugaring `letrec` into a core expression involving an application, similarly to how `let` expressions desugar.
   (One might also have attempted to use *generative recursion* to recursively call `desugar` to desugar the `let`. This would be challenging, since `UninitializedC` cannot be constructed in the surface language.)

Your desugaring should be such that `letrec` accepts a sequence of *possibly-mutually-recursive* binders.
That is, each identifier bound in a `letrec` expression should be in scope in each binder expression.
For example, in `(letrec ((<id1> <expr1>) (<id2> <expr2>)) <body>)`, `<id1>` and `<id2>` should be bound in all expressions, i.e., `<expr1>`, `<expr2>`, and `<body>`.

2 Examples

Here are some examples that your interpreter should accept.

2.1 Summing values, imperatively

```
(let
 ((sumto  (box 0))
  (countv (box 0))
  (sumv   (box 0))
  (nop    0)
  (runsum (box 0))
 )
 (seq
  (setbox
   runsum
   (lambda ()
     (if (num= (unbox countv) (unbox sumto))
       nop
       (seq (seq (setbox countv (+ (unbox countv) 1))
                 (setbox sumv   (+ (unbox sumv  ) (unbox countv))))
            ((unbox runsum)))
     )
   )
  )
  (seq (setbox sumto 5)
       (seq ((unbox runsum))
            (unbox sumv)
       )
  )
 )
)
```

With boxes, imperative code can be written.

2.2 Functional Fibonacci

```
(letrec
  ((fib
    (lambda (n)
      (if (num= n 0)
        0
        (if (num= n 1)
          1
          (+ (fib (- n 1)) (fib (- n 2))))))))
  (fib 5))
```

A naive function for computing the fibonacci numbers.

2.3 Imperative Fibonacci

```
(let ((a 0) (b 1) (sum 0))
  (letrec
    ((fib
      (lambda (n)
        (if (or (num= n 0) (num= n 1))
          sum
          (seq (set sum (+ a b))
          (seq (set a b)
          (seq (set b sum)
              (fib (- n 1)))))))))
      (fib 5)))
```

A function for computing fibonacci numbers that uses `set` on `let`-bound variables to store intermediate results.

3 Grammar

Here is the extended grammar:

```
module mutation

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]

  UnOp.BOX          = [box]
  UnOp.UNBOX        = [unbox]

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

  BinOp.SETBOX      = [setbox]
  BinOp.SEQ         = [seq]

  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])]
  Expr.Set          = [(set [ID] [Expr])]
  LetBind.LetBindExt = [([ID] [Expr])]

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

  Expr.LetRecExt     = [(letrec ([LetBind+]) [Expr])]
```

Note that `[Expr+]` denotes one or more of `[Expr]`.

4 Classes

4.1 Abstract Syntax

The ExprExt case classes are already defined and imported via `import Parser._` ***. You should not put these in your solution!***

```
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 SetExt(id: String, e: ExprExt) extends ExprExt
case class RecLamExt(name: String,
                     param: String,
                     body: ExprExt) extends ExprExt
case class LetRecExt(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", "setbox", "seq")
  val unOps = Set("-", "not", "head", "tail", "is-nil", "is-list", "box", "unbox")
  val reservedWords = binOps ++ unOps ++ Set("list", "if", "lambda",
    "let", "true", "false", "rec-lam", "set", "letrec")
}
```

4.2 Desugared Syntax

These case classes are also provided! No import is needed. Do not copy these to your own solution.

```
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 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 BoxC(v: ExprC) extends ExprC
case class UnboxC(b: ExprC) extends ExprC
case class SetboxC(b: ExprC, v: ExprC) extends ExprC
case class SetC(v: String, b: ExprC) extends ExprC
case class SeqC(b1: ExprC, b2: ExprC) extends ExprC
case class UninitializedC() extends ExprC
```

4.3 Values

These case classes are also provided! No import is needed. Do not copy these to your own solution.

```
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(hd: Value, tl: Value) extends Value
case class PointerClosV(f: FdC, env: List[Pointer]) extends Value
case class BoxV(l: Int) extends Value
case class UninitializedV() extends Value
```

4.4 Other

These case classes are also provided! Do not copy these to your own solution.

```
case class Pointer(name: String, location: Int)
case class Cell(location: Int, value: Value)
```

Note that `Bind` no longer stores a map from name to `Value`. This is replaced with the `Pointer` case class.
A pointer maps a name to an integer location. The values are stored in a `Cell` which maps locations to `Value`s.

4.5 Exceptions

Define your own *specific* exceptions that *inherit* from the given abstract exceptions. Throw only exceptions derived from `DesugarException` in the desugarer and `InterpException` in the interpreter.
The abstract classes are already provided for you. Do not copy these to your own solution.

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

Think about writing a program using your own interpreter: what exception names are useful from the view of the programmer?
What exception parameters are useful from the view of the programmer?