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

1.2. Week 2: Basic Interp

1 Interpreter

To complete this week's assignments, the `interp`, `desugar` and `parse` functions must be implemented:

- `parse` takes an s-expression (`SExpr`) as input, and returns an abstract syntax tree (`ExprExt`).
- `desugar` takes an abstract syntax tree (`ExprExt`) as input, and returns a core syntax tree (`ExprC`).
- `interp` takes a core abstract syntax tree (`ExprC`) as input, and returns a `Value`. Interpretation should raise an `InterpException` for undefined behavior, such as `(+ true 1)`.

2 Features to Implement

2.1 Binary Operators

The language you will be implementing, called Paret, includes binary addition (`+`), binary subtraction (`-`), binary multiplication (`*`), unary negation (`-`), and number comparison operations (`num=`, `num<`, `num>`). These operations should be defined in terms of their counterparts in Scala.

Instead of having separate rules (and syntactic forms) for `+`, `-`, `*`, `num=`, `num<`, and `num>`, we will define a single syntactic rule for all binary operators. `desugar` converts these operators into the desugared datatype variant, shown in the data definition below.

2.2 Conditionals

2.2.1 `if`-Expressions

`if`-expressions in Paret are composed of three parts:

A "test" expression that evaluates to a `BoolV`. A "then" expression that evaluates if the test expression evaluates to `true`. An "else" expression that evaluates if the test expression evaluates to `false`.

2.2.2 Multi-Armed Conditional

A multi-armed conditional, or a `cond` expression, consist of a list of `(condition expression)` pairs, where both `condition` and `expression` are expressions. If the `condition` of the pair evaluates to true, then the result of the multi-armed conditional expression is the result of evaluating `expression`. Otherwise, if the `condition` of a pair evaluates to false, the next pair is tried.

If each pair's condition evaluates to false and the condition of the last pair is not an `else` expression, a InterpException should be raised.

For example, the following program evaluates to `1`:

```
(cond
  ((num< 1 0) 0)
  (else 1))
```

Whereas the following program results in an `InterpException`:

```
(cond
  ((num> 0 1) 1)
  ((num< 1 0) 2))
```

Similarly, the following program should result in an `InterpException` (hint: see the undefined construct below):

```
(cond
  ((and true false) 3))
```

Your `desugar` function should desugar `cond` expressions into (possibly-)nested `if` expressions.

2.2.3 And/Or

When given two operands, `and` evaluates to the value of the second operand if the first operand evaluates to `true`, and `false` otherwise. When given two operands, `or` evaluates to `true` if the first operand evaluates to `true`, and the value of the second operand otherwise. The `desugar` function should convert `and` and `or` into equivalent expressions. The `interp` function should evaluate expressions lazily and should short-circuit according to the rules of a loosely typed language. For example, `(and true 9)` should evaluate to `9`.

2.2.4 Not

The `desugar` function should convert a `(not e)` expression into another expression that evaluates to true when `e` evaluates to false, and to false when `e` evaluates to true.

2.3 Lists

2.3.1 `nil` and `cons`

A `nil` expression should yield a `NilV` value, whereas `(cons e1 e2)` should yield a `ConsV` value that "conses" the result of `e1` with the result of `e2`, similar to Scala cons lists.

2.3.2 Operations on Lists: `head`, `tail`, `is-nil`, and `is-list`

The `head` and `tail` expressions should return the head and tail elements of a cons.

The `is-nil` expression should return a Boolean indicating whether a given list is empty or not. `is-nil` is only well-defined for list values.

The `is-list` expression should test whether a given value is a list value or not.

2.3.3 Syntactic sugar for lists

`list` is syntactic sugar for cons lists in Paret. `(list 1 2 3)` should desugar into a cons list: `(cons 1 (cons 2 (cons 3 nil)))`.

2.4 Undefined Behavior

To deal with undefined behavior, the core language classes have an `UndefinedC` construct, which should give rise to an `InterpException` instance when interpreted.

3 Testing

Extend your test suite with tests in order to validate the behaviour of your `parse`, `desugar`, and `interp` functions. Your tests will be graded in the test suite assignment.

4 Grammar

The concrete syntax of the Paret language is given by the following grammar:

```
module conditionals

imports Common

context-free syntax

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

  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*])]
```

Note that `[Expr*]` denotes zero or more of `[Expr]`.

5 Classes

These classes should be used in your solution.

5.1 S-Expressions

The s-expression constructors are the same as the ones you implemented in week 1.

5.2 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

object ExprExt {
  val binOps = Set("+", "*", "-", "and", "or", "num=", "num<", "num>", "cons")
  val unOps = Set("-", "not", "head", "tail", "is-nil", "is-list")
}
```

5.3 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(n: 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
```

Note that `LtC` is the less than operation.

5.4 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
```

5.5 Exceptions

For erroneous grammar and abstract syntax trees, the correct `Exception`s should be thrown. The library provides three exceptions that you should extend:

```
abstract class ParseException(msg: String = null)
abstract class DesugarException(msg: String = null)
abstract class InterpException(msg: String = null)
```