Week 2: Basic Interp

Concepts of Programming Languages

Course: TI2606 (https://weblab.tudelft.nl/ti2606/) Edition: 2017-2018 From February 10, 2018 until July 6, 2018 (https://weblab.tudelft.nl/ti2606/2017-2018/)

Course Information

Home (https://weblab.tudelft.nl/ti2606/2017-2018/)
All editions (https://weblab.tudelft.nl/ti2606/)
News archive (https://weblab.tudelft.nl/ti2606/2017-2018/news)
Course rules (https://weblab.tudelft.nl/ti2606/2017-2018/rules)
Lecture notes (https://weblab.tudelft.nl/ti2606/2017-2018/note/155)
Assignments (https://weblab.tudelft.nl/ti2606/2017-2018/assignment/14857/)

Your enrollment

2.2. Week 2: Basic Interp

(https://weblab.tudelft.nl/ti2606/2017-2018/note/162)
(https://weblab.tudelft.nl/ti2606/2017-2018/note/161)
(https://weblab.tudelft.nl/ti2606/2017-2018/note/164)

1 Interpreter

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

`parse` consumes an s-expression (`SExpr`), and produces an abstract syntax tree (`ExprExt`).

`desugar` consumes an abstract syntax tree (`ExprExt`), replaces all instances of `UnOpExt`, `BinOpExt`, and conditionals with their respective desugared counterparts, and returns the resulting core language syntax tree (`ExprC`).

`interp` consumes the desugared abstract syntax tree (`ExprC`) and returns a `Value`.

2 Features to Implement

2.1 Binary Operators

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.

Interpretation should raise an `InterpException` for non-numeric

- Your Course Dossier (https://weblab.tudelft.nl/ti2606/2017-2018/dossier/tbruyn)
- Your Submissions (https://weblab.tudelft.nl/ti2606/2017-2018/assignment/14857/submission/10127/)
- Unenroll

Course staff

Lecturers

- Casper Poulsen (https://weblab.tudelft.nl/profile/cbachpoulsen)
- Eelco Visser (https://weblab.tudelft.nl/profile/eelcovisser)

Assistants

- Taico Aerts (https://weblab.tudelft.nl/profile/taerts)
- Casper Boone (https://weblab.tudelft.nl/profile/cboone)
- Maarten Sijm (https://weblab.tudelft.nl/profile/msijm)
- Thomas Smith (https://weblab.tudelft.nl/profile/tsmith)
- Dominique van Cuilenborg (https://weblab.tudelft.nl/profile/dvancuilenborg)

values passed to binary number operators.

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`.
`if`-expressions should short-circuit and evaluation should raise an `InterpException` for non-boolean “test” values.

2.2.3 And/Or

When given two booleans, `and` evaluates to `true` if they are both `true`, or to `false` otherwise.
When given two booleans, `or` evaluates to `true` if either one is `true`, or to `false` otherwise.
The `desugar` function should convert `and` and `or` into equivalent expressions.
The `interp` function should evaluate expressions lazily and should short-circuit.

2.2.4 Not

The `desugar` function should convert a `not` expression into an equivalent expression.

2.3 Pairs

2.3.1 `pair`-Expressions and Values

Interpretation of a `(pair e1 e2)` expression should construct a pair by evaluating `e1` and `e2` isto values, and returning a `Value` that pairs these.

2.3.2 Operations on Pairs

The `fst` and `snd` expressions should return the first and second element of a pair, respectively.

Intperpretation should raise an `InterpException` for non-pair values passed to `fst` or `snd`.

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

2.3.3 Tuples (Nested Pairs)

A `tuple` expression should take a sequence of `n` expressions, where `n > 1`.
Tuples should `desugar` into a right-nested sequence of `n - 1` pairs.
E.g., `(tuple e1 e2 ... en)` should desugar into a core expression that is equivalent to `(pair e1 (pair e2 ... en))`.

A projection expression `(proj n e)` expects a number literal `n` and an expression `e` that computes a tuple.
Intepreting `(proj n e)` first evaluates `e` to a tuple, and then projects the `n`th element of the tuple where `0` denotes the initial element.
Projections should `desugar` into a sequence of `SndC` expressions, wrapped in a `CheckFstC` expression.
You should extend your interpreter to handle an `CheckFstC(_)` expression as follows:
1. Evaluate the argument that `CheckFstC` is given, and check if the resulting value is a pair.
2. If the value is a pair, return the first element of the pair.
3. Otherwise, return the resulting value directly.

3 Grammar

The concrete syntax of the Paret language with these additional features can be captured with the following scheme:

```
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.FST      = [fst]
  UnOp.SND      = [snd]
  UnOp.ISPAIR   = [is-pair]

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

  BinOp.PAIR    = [pair]
  BinOp.PROJ    = [proj]

  Expr.IfExt     = [(if [Expr] [Expr] [Expr])]

  Expr.TupleExt  = [(tuple [Expr] [Expr+])]
```

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

4 Classes

These classes should be used in your solution.

4.1 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: E
xprExt) extends ExprExt
case class UnOpExt(s: String, e: ExprExt) exten
ds ExprExt
case class IfExt(c: ExprExt, t: ExprExt, e: Exp
rExt) extends ExprExt
case class TupleExt(l: List[ExprExt]) extends E
xprExt

object ExprExt {
  val binOps = Set("+", "*", "-", "and", "or",
"num=", "num<", "num>", "pair", "proj")
  val unOps = Set("-", "not", "is-pair", "fst",
"snd")
}
```

4.2 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(n: Int) extends ExprC
case class PlusC(l: ExprC, r: ExprC) extends Ex
prC
case class MultC(l: ExprC, r: ExprC) extends Ex
prC
case class IfC(c: ExprC, t: ExprC, e: ExprC) ex
tends ExprC
case class EqNumC(l: ExprC, r: ExprC) extends E
xprC
case class LtC(l: ExprC, r: ExprC) extends Expr
C
case class PairC(l: ExprC, r: ExprC) extends Ex
prC
case class FstC(e: ExprC) extends ExprC
case class SndC(e: ExprC) extends ExprC
case class IsPairC(e: ExprC) extends ExprC
case class CheckFstC(e: ExprC) extends ExprC
```

Note that `LtC` is the less than operation.

4.3 Values

```
sealed abstract class Value
case class NumV(n: Int) extends Value
case class BoolV(b: Boolean) extends Value
case class PairV(l: Value, r: Value) extends Va
lue
```

4.4 Exceptions

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

```
abstract class ParseException(msg: String = nul
l)
abstract class DesugarException(msg: String = n
ull)
abstract class InterpException(msg: String = nu
ll)
```

5 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.

Assignments

2: Interp Basic (https://weblab.tudelft.nl/ti2606/2017-2018/assignment/14933/)