Implementing a Simple Programming Language

Functional Programming (CS-210)

Viktor Kunčak

EPFL

Simple Untyped Functional Language

Example program:

```
(
  def fact = (n => (if n then (* n (fact (- n 1))) else 1))
  (fact 6)
)
```

evaluates to:

Simple Untyped Functional Language

Example program:

```
(
  def fact = (n => (if n then (* n (fact (- n 1))) else 1))
  (fact 6)
)
```

evaluates to: **720**

Simple Untyped Functional Language

Example program:

```
(
  def fact = (n => (if n then (* n (fact (- n 1))) else 1))
  (fact 6)
)
```

evaluates to: **720**

```
(
  def twice =  (f => x => (f (f x)))
  def square = (x => (* x x))
  (twice square 3)
)
```

evaluates to:

Simple Untyped Functional Language

Example program:

```
(
  def fact = (n => (if n then (* n (fact (- n 1))) else 1))
  (fact 6)
)
```

evaluates to: **720**

```
(
  def twice =  (f => x => (f (f x)))
  def square = (x => (* x x))
  (twice square 3)
)
```

evaluates to: **81**

Program Representation: Abstract Syntax Trees

```
(def twice =  (f => x => (f (f x)))
 def square = (x => (* x x))
 (twice square 3))
```

≈

```
val defs : DefEnv = Map[String, Expr](
  "twice" -> Fun("f", Fun("x",
                        Call(N("f"), Call(N("f"), N("x"))))),
  "square" -> Fun("x", BinOp(Times, N("x"), N("x"))))
val expr = Call(Call(N("twice"), N("square")), C(3))
```

- We represent a program using ***expression tree*** called Abstract Syntax Tree (AST)
- Our implementation is an ***interpreter***, which traverses AST to produce the result
- We discuss later briefly how to convert an input file into an abstract syntax tree; more on that in the course ***Computer Language Processing (CS-320)*** next year

Growing a Language and Its Interpreter

I01 Language of arithmetic and ***if*** expressions

I02 Absolute value and its ***desugaring***

I03 ***Recursive*** functions implemented using ***substitutions***

I04 ***Environment*** instead of substitutions

I05 ***Higher-order*** functions using substitutions

I06 Higher-order functions using environments

I07 ***Nested recursive*** definitions using environments

I01. Language of arithmetic and *if* expressions: Trees

Integer constants combined using arithmetic operations and the `if` conditional

```
val expr1 = BinOp(Times, C(6), C(7))          // 6 * 7
val cond1 = BinOp(LessEq, expr1, C(50))       // expr1 <= 50
val expr2 = IfNonzero(cond1, C(10), C(20))  // if (cond1) 10 else 20
```

How to describe such trees?

I01. Language of arithmetic and *if* expressions: Trees

Integer constants combined using arithmetic operations and the `if` conditional

```
val expr1 = BinOp(Times, C(6), C(7))         // 6 * 7
val cond1 = BinOp(LessEq, expr1, C(50))      // expr1 <= 50
val expr2 = IfNonzero(cond1, C(10), C(20))  // if (cond1) 10 else 20
```

How to describe such trees?

```
enum Expr
  case C(c: BigInt)                            // integer constant
  case BinOp(op: BinOps, e1: Expr, e2: Expr) // binary operation
  case IfNonzero(cond: Expr, trueE: Expr, falseE: Expr)

enum BinOps
  case Plus, Minus, Times, Power, LessEq
```

I01. Language of arithmetic and *if* expressions: Printing

```
def str(e: Expr): String = e match
  case C(c) => c.toString
  case BinOp(op, e1, e2) =>
    s"(${strOp(op)} ${str(e1)} ${str(e2)})" // string interpolation
  case IfNonzero(cond, trueE, falseE) =>
    s"(if ${str(cond)} then ${str(trueE)} else ${str(falseE)})"

def strOp(op: BinOps): String = op match
  case Plus   => "+"
  case Minus  => "-"
  case Times  => "*"
  case Power  => "^"
  case LessEq => "<="

> str(IfNonzero(BinOp(LessEq, C(4), C(50)), C(10), C(20)))
(if (<= 4 50) then 10 else 20)
```

I01. Language of arithmetic and *if* expressions: Interpreting

```
def eval(e: Expr): BigInt = e match
  case C(c) => c
  case BinOp(op, e1, e2) =>
    evalBinOp(op)(eval(e1), eval(e2))
  case IfNonzero(cond, trueE, falseE) =>
    if eval(cond) != 0 then eval(trueE) else eval(falseE)

def evalBinOp(op: BinOps)(x: BigInt, y: BigInt): BigInt = op match
  case Plus => x + y
  case Minus => x - y
  case Times => x * y
  case Power => x.pow(y.toInt)
  case LessEq => if (x <= y) 1 else 0

> eval(IfNonzero(BinOp(LessEq, C(4), C(50)), C(10), C(20)))
10
```

I02. Absolute Value and Its ***Desugaring***: Trees

```
enum Expr
  case C(c: BigInt)
  case BinOp(op: BinOps, e1: Expr, e2: Expr)
  case IfNonzero(cond: Expr, trueE: Expr, falseE: Expr)
  case AbsValue(arg: Expr)   // new case
```

How to extend evaluator to work with absolute value as well? Two approaches:

- add a case to the interpreter (exercise)
- transform (desugar) trees to reduce them to previous cases

Syntactic sugar = extra language constructs that are not strictly necessary because they can be expressed in terms of others (they make the language sweeter to use)

Desugaring = automatically eliminating syntactic sugar by expanding constructs

I02. Desugaring Absolute Value: Idea

By definition of absolute value, we would like this equality to hold:

`abs x` $\equiv$ **`if`** `(<= x 0)` **`then`** `(- 0 x)` **`else`** `x`

that is, at the level of AST,

```
AbsValue(x) ⇝
   IfNonzero(BinOp(LessEq, x, C(0)),
             BinOp(Minus, C(0), x),
             x)
```

How to write `desugar` function that eliminates all occurrences of `AbsValue`?

I02. Desugaring Absolute Value: Idea

By definition of absolute value, we would like this equality to hold:

`abs x` $\equiv$ **`if`** `(<= x 0)` **`then`** `(- 0 x)` **`else`** `x`

that is, at the level of AST,

```
AbsValue(x) ⇝
   IfNonzero(BinOp(LessEq, x, C(0)),
             BinOp(Minus, C(0), x),
             x)
```

How to write `desugar` function that eliminates all occurrences of `AbsValue`?

Replace (recursively) each subtree `AbsValue(x)` with its definition.

102. Desugaring Absolute Value: Code

```
def desugar(e: Expr): Expr = e match
  case C(c) => e
  case BinOp(op, e1, e2) =>
       BinOp(op, desugar(e1), desugar(e2))
  case IfNonzero(cond, trueE, falseE) =>
       IfNonzero(desugar(cond), desugar(trueE), desugar(falseE))
  case AbsValue(arg) =>
    val x = desugar(arg)
    IfNonzero(BinOp(LessEq, x, C(0)),
              BinOp(Minus, C(0), x),
              x)
```

I02. Desugaring Absolute Value: Example Run

```
def show(e: Expr): Unit =
  println("original:")
  println(str(e))
  val de = desugar(e)
  println("desugared:")
  println(str(de))
  println(" ~~> " + eval(de) + "\n")

show(AbsValue(BinOp(Plus,C(10),C(-50))))

original:
(abs (+ 10 -50))
desugared:
(if (<= (+ 10 -50) 0) then (- 0 (+ 10 -50)) else (+ 10 -50))
  ~~> 40
```