

Parsing + I/O: An Expression Evaluator

Goal: A program which evaluates infix expressions.

A very simple expression syntax:

```
Expr ::= Num | Var | App
```

```
App ::= '(' Expr Op Expr ')'
```

```
Num ::= [-] Digit { Digit }
```

```
Op ::= + | - | * | /
```

```
Var ::= Lower { Lower }
```

```
Lower ::= a | b | ... | z
```

```
Digit ::= 0 | 1 | ... | 9
```

Examples:

```
(42+a)
```

```
((6-2)*(a--2))
```

To provide a means of assigning values to variables, we add syntax for *commands*:

```
Comm ::= Var := Expr | Expr
```

That is, a command is one or the other of

- an assignment of an expression's value to a variable
- an expression.

```
> module Evaluator where
```

```
> import ParseLib -- Hugs Lib/Hugs/ParseLib.hs
```

```
> type Mem = [(String,Int)]
```

```
> store :: Mem -> String -> Int -> Mem
```

```
> store m v i = (v,i) : m
```

```
> fetch :: Mem -> String -> Int
```

```
> fetch []          var = 0
```

```
> fetch ((v,i):vis) var
```

```
>   | var == v      = i
```

```
>   | otherwise     = fetch vis var
```

```
> calc :: IO ()
```

```
> calc = do putStr "Expression evaluator"
```

```
>         run []
```

```
> run :: Mem -> IO ()
```

```
> run mem
```

```
>   = do putStr "\n? "
```

```
>         s <- getLine
```

```
>         if s == quitSignal
```

```
>           then do { putStr "...bye"; return () }
```

```
>           else
```

```
>             case parse s of
```

```
>               [((v,m),"")] -> do putStr (" = "++ show v)
```

```
>                               run m
```

```
>               _            -> do putStr " *** error"
```

```
>                               run mem
```

```
>       where
```

```
> comm :: Parser (Int,Mem)
```

```
> comm = do v <- name
```

```
>           char ':'
```

```
>           char '='
```

```
>           e <- expr
```

```
>           return (e, store mem v e)
```

```
>           +++
```

```
>           do e <- expr
```

```
>           return (e,mem)
```

```
> expr :: Parser Int
```

```
> expr = num +++ var +++ app
```

```
> var :: Parser Int
```

```
> var = do n <- name
```

```
>       return (fetch mem n)
```

```
> app :: Parser Int
```

```
> app = do char '('
```

```
>         e1 <- expr
```

```
>         op <- char '+' +++ char '-' +++
```

```
>           char '*' +++ char '/'
```

```
>         e2 <- expr
```

```
>         char ')'
```

```
>         case op of
```

```
>           '+' -> return (e1 + e2)
```

```
>           '-' -> return (e1 - e2)
```

```
>           '*' -> return (e1 * e2)
```

```
>           '/' -> if e2 /= 0
```

```
>                 then return (e1 `div` e2)
```

```
>                 else mzero
```

```
> nat :: Parser Int
```

```
> nat = do d <- digit
```

```
>         s <- many digit
```

```
>         return (read (d:s))
```

```
> num = do char '-'
```

```
>         n <- nat
```

```
>         return (-n)
```

```
>         +++ nat
```

```
> name :: Parser String
```

```
> name = do c <- lower
```

```
>         cs <- many lower
```

```
>         return (c:cs)
```

```
> quitSignal = "#"
```

Possible modifications and extensions

- add precedence and associativity (watch out for left-recursion)
- add syntax-error messages
- add provisions for whitespace (see ParseLib.hs)
- change parser's output to
 - * abstract syntax tree
 - * executable code

Formal methods: Just an “ivory-tower” exercise?

Real-life successes:

- Inmos Corp. (UK). Used formal methods to verify compliance of transputer™ processor chip's floating-point logic with IEEE 754 standard. Quicker and cheaper than testing, and covered *all* cases (impossible with testing). David May, “Use of Formal Methods by a Silicon Manufacturer”, in C.A.R. Hoare, ed., *Developments in Concurrency and Communication*, Addison-Wesley, 1990.
- IBM UK Labs. Used formal methods (Z) in development of transaction-processing software (CICS/ESA). Estimated savings: \$5.5 million. G. Jones, “Oxford Wins Queen's Award” in *Computing Research News* 4, 4 (Sept. 92).
- Praxis Critical Systems and Univ. of York (UK) developed a helicopter landing system under UK Interim Defence Standards 00-55 and 00-56, which *require* the use of formal methods in safety-critical applications.

Used Z for documenting the system specification and part of the design. Z proofs were substantially more efficient at finding faults than the most efficient testing phase. Given the importance of early fault detection, this helps to demonstrate the significant benefit and practicality of large-scale proof on projects of this kind.

(Paper in LNCS 1709 (1999), p. 1527 ff.)

Barrier to more widespread use of formal methods: not cost, but habit and ignorance. Programmers who are prepared will have an advantage.

The World Wide Web Virtual Library: Formal Methods

<http://www.comlab.ox.ac.uk/archive/formal-methods.html>

Formal reasoning about functional programs**Recursion and Induction****1. over natural numbers** (“mathematical induction”)Example: exponentiation (\wedge)

$$\begin{aligned} > z^0 &= 1 && \text{base} \\ > z^k \mid k > 0 &= z * (z^{k-1}) && \text{recursion} \end{aligned}$$

This gives a method for computing z^k by computing z^{k-1} .

The Inductive Proof schema:

To prove that $P(n)$ holds for all natural numbers n , you have to prove **both**

Case $P(0)$. That $P(0)$ is true. *base*

Case $P(n) \Rightarrow P(n+1)$.
That **if** $P(n)$ holds
then $P(n+1)$ holds also. *induction*

Note the correspondence between
the **definition** schema and
the **proof** schema.

Example: Given the definition of (\wedge) and the properties of (+) and (*), show that for all numbers x and natural numbers m and n ,

$$P(n): x^{\wedge(m+n)} = (x^{\wedge m}) * (x^{\wedge n})$$

Case $P(0)$: $x^{\wedge(m+0)} = (x^{\wedge m}) * (x^{\wedge 0})$

$$x^{\wedge(m+0)} = (x^{\wedge m}) * (x^{\wedge 0})$$

$$= \{ a+0 = a; \wedge.1 \}$$

$$x^{\wedge m} = (x^{\wedge m}) * 1$$

$$= \{ a * 1 = a \}$$

$$x^{\wedge m} = x^{\wedge m}$$

$$= \{ \text{reflexivity of } = \}$$

True

Case $P(n) \Rightarrow P(n+1)$:

$$(x^{\wedge(m+n)} = (x^{\wedge m}) * (x^{\wedge n}))$$

$$\Rightarrow (x^{\wedge(m+(n+1))} = (x^{\wedge m}) * (x^{\wedge(n+1)}))$$

$$x^{\wedge(m+(n+1))} = (x^{\wedge m}) * (x^{\wedge(n+1)})$$

$$= \{ + \text{ assoc.}, \wedge.2 \}$$

$$x^{\wedge((m+n)+1)} = (x^{\wedge m}) * (x * (x^{\wedge n}))$$

$$= \{ \wedge.2, * \text{ assoc.} \}$$

$$x * (x^{\wedge(m+n)}) = (x^{\wedge m}) * x * (x^{\wedge n})$$

$$= \{ * \text{ commut.} \}$$

$$x * (x^{\wedge(m+n)}) = x * ((x^{\wedge m}) * (x^{\wedge n}))$$

$$= \{ \text{induction hypothesis} \}$$

$$x * (x^{\wedge(m+n)}) = x * (x^{\wedge(m+n)})$$

$$= \{ \text{reflexivity of } = \}$$

True ■

Another class of recursively defined values: $[\alpha]$

Just as

$$x \in \mathbf{N} \text{ (natural numbers)} \Rightarrow (x = 0 \vee (\exists n \mid n \in \mathbf{N} : x = (n+1)))$$

so

$$x \in [\alpha] \text{ (lists over } \alpha) \Rightarrow (x = [] \vee (\exists a, as \mid a \in \alpha \wedge as \in [\alpha] : x = a:as))$$

i.e., every list can be expressed as either $[]$ or $(a:as)$.

Just as we defined primitive-recursive functions over natural numbers by specifying

$$f 0 = k \quad \text{and} \quad f(n+1) = \dots f(n) \dots$$

we define primitive-recursive functions over lists by specifying

$$f [] = k \quad \text{and} \quad f(x:xs) = \dots f(xs) \dots$$

Example: (length)

$$> \text{len } [] = 0$$

$$> \text{len}(r:rs) = 1 + \text{len } rs$$

Example: (++)

$$> [] ++ bs = bs$$

$$> (a:as) ++ bs = a : (as++bs)$$