Programming Paradigms and Formal Semantics

Lab 6: More Steps in Haskell (Basic Language Processing)

© Vadim Zaytsev, Ralf Lämmel
(@grammarware & @notquiteabba),
Software Languages Team,
Universität Koblenz-Landau

Presenting teams

- 42
  - Hahn, Neumann, Klass
- delta
  - Bauer, Theisen, Künster
- funkymonkey
  - Lellmann, Schröter, Schauß
- Jigsaw
  - Brenk, Hück, Klauer
- kbd
  - Klein, Borth, Daudrich
- ssp
  - Saal, Schmorleiz, Polster
- tekkan
  - Altgeld, Spies, Lackner
- tthesing — Thesing

Lookup & update

```
mylookup :: Identifier -> LookupTable -> Maybe Int
mylookup e [] = Nothing
mylookup e ((x,v):xvs) =
    if e == x
    then Just v
    else mylookup e xvs

myupdate :: LookupTable -> Identifier -> Int ->
LookupTable
myupdate [] x v = [(x, v)]
myupdate ((x1,v1):xvs) x2 v2 =
    if x1 == x2
    then ((x1,v2):xvs)
    else ((x1,v1):(myupdate xvs x2 v2))
```

Boolean expr evaluation

```
evalb :: BExpression -> LookupTable -> Bool
evalb BTrue _ = True
evalb BFalse _ = False
evalb (Equals a1 a2) e =
    (evala a1 e) == (evala a2 e)
evalb (LessThanOrEqual a1 a2) e =
    (evala a1 e) <= (evala a2 e)
evalb (Not b) e =
    not (evalb b e)
evalb (And b1 b2) e =
    (evalb b1 e) && (evalb b2 e)
```

Statement evaluation (1)

```
evals :: Statement -> LookupTable -> LookupTable

evals (SList s1 s2) e =
    evals s2 (evals s1 e)
evals Skip e =
    e
evals (Assign i a) e =
    myupdate e i (evala a e)
```

Statement evaluation (2)

```
evals :: Statement -> LookupTable -> LookupTable

evals (IfThenElse b st se) e =
    if evalb b e
    then evals st e
    else evals se e

evals (While b s) e =
    if evalb b e
    then evals (While b s) (evals s e)
    else e
```

Remember B and NB?

B (slide 108)

```
true
false
if T then T else T
```

NB (slide 109)

```
true
false
if T then T else T
0
succ T
pred T
iszero T
```

Recall the syntax of (N)B

```
% B in Prolog
term(true).
term(false).
term(if(T1,T2,T3)) :-
    term(T1), term(T2), term(T3).
% ----------------------------------------

% Peano numbers
term(zero).
term(succ(T)) :- term(T).
term(pred(T)) :- term(T).

% iszero :: Nat -> Bool
term(iszero(T)) :- term(T).
```

The syntax of B in Haskell

```
module B where

data B
    = TrueB
    | FalseB
    | IfB B B B
    deriving Show
```

slide 420 for Recursive Types

Parsing B in Haskell (1)

```
import B
import Parsing

b = true +++ false +++ cond
true =
 do
    token (string "1")
    return TrueB
false =
 do
    token (string "0")
    return FalseB
```

Parsing B in Haskell (2)

```
cond =
 do
    token (string "if")
    x <- b
    token (string "then")
    y <- b
    token (string "else")
    z <- b
    token (string "fi")
    return (IfB x y z)
```

B syntax as a Haskell data type

```
data B
    = TrueB
    | FalseB
    | IfB B B B
```

Constructors for B values

```
TrueB :: B

FalseB :: B

IfB :: B -> B -> B -> B
```

Constructors for B values

```
TrueB :: B

FalseB :: B

IfB :: B -> B -> B -> B



foldB :: r -> r -> (r->r->r->r) -> (B -> r)
```

Folding for B

```
foldB :: r -> r -> (r->r->r->r) -> (B -> r)

foldB r _ _ TrueB = r

foldB _ r _ FalseB = r

foldB r1 r2 f (IfB x y z) =
        f (fold x) (fold y) (fold y)
        where fold = foldB r1 r2 f
```

see slides 426-429

Using foldB (1)

```
foldB :: r -> r -> (r->r->r->r) -> (B -> r)

% count the parse tree maximal depth
depth :: B -> Int
depth = foldB
     1
     1
     (\ x y z -> 1 + (maximum [x, y, z]))
```

Using foldB (2)

```
foldB :: r -> r -> (r->r->r->r) -> (B -> r)

% count how many times TrueB occurs
countT :: B -> Int
countT = foldB
    1
    0
    (\ x y z -> x + y + z)
```

Using foldB (3)

```
foldB :: r -> r -> (r->r->r->r) -> (B -> r)

% count how many times FalseB occurs
countF :: B -> Int
countF = foldB
    0
    1
    (\ x y z -> x + y + z)
```

Using foldB (4)

```
foldB :: r -> r -> (r->r->r->r) -> (B -> r)

% evaluate a B expression
eval :: B -> Bool
eval = foldB
    True
    False
    (\ x y z -> if x then y else z)
```

Assignment 6

- Complete the data type B to cover the abstract syntax of NB
- Complete the B parser to cover all NB expressions and product NB abstract syntax trees
  - concrete syntax is up to you

- Develop a folding operator for NB
- Demonstrate its use with evaluation function

- Write a test case that:
  - is parsed from text into NB AST
  - evaluated using the fold-based function

Conclusion

- Complete the last assignment
- Prepare for the midterm exam
- Questions can be tweeted to @grammarware or @notquiteabba or @yaplcourse or emailed
- The working code/tests/makefiles are available via SLPS: http://slps.sf.net, use SVN to check out.

```
svn co https://slps.svn.sourceforge.net/svnroot/slps/topics/exercises/b2
svn co https://slps.svn.sourceforge.net/svnroot/slps/topics/exercises/b3
```