Tree-walk put in a Nutshell

https://github.com/maxstrauch/sle-tree-walk

Based on: „Walk your tree any way you want“,
A. H. Bagge and R. Lämmel, June 2013

SLE Winter Term 2015/16, Assignment 03,

University of Koblenz-Landau

Maximilian Strauch

UNIVERSITÄT KOBLENZ · LANDAU

http://softlang.wikidot.com/course:sle1516

Source: [1]

A DSL for tree walks

Flashback

- [1] proposes a custom DSL to define tree walks *(only one simple example given here!)*

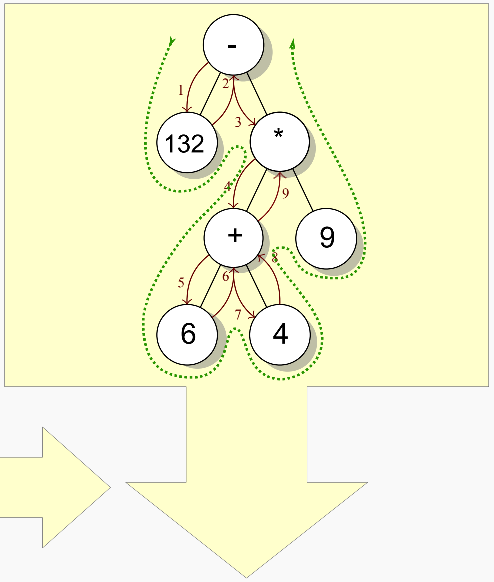
Stateful variable definition

```
walk toString {
  state s = "";
  if leaf {
    then s += data;
  } else {
     if down then s += name + "(";
     if up then s += ")";
     if from>=first && from<last then s += ", ";
  }
  walk to next;
}
```

Various Join Points reacting to different node “events“

Memory of current and last visited node

Define where to go next; here: default walk. Possible other expression:



```
-(132, *(+(6, 4), 9))
```

Intend for this assignment

- ***Objective:*** *develop a simple self-contained implementation of a Nuthatch DSL interpreter*
- Why?
  - Work out the core functionality of the Nuthatch tree walk idea
  - See the beauty and effectiveness of this idea at work
  - Get hands on technology!
- How?
  - Reduce the Nuthatch DSL to its bare minimum

Language & technology scope

Type	Name	Role
software language	Nuthatch DSL	Textual formulation or DSL definied in [1] as Nuthatch tree walk language (*see slide 2*).
	Tree walk language (*or* paradigm)	Paradigm of traversing a tree and choosing between nodes as proposed in [1]. (More abstract formulation of how the walk is performed.)
	Tree DSL	DSL to model a simple tree with an arbitray count of child nodes and a string value for every node.
software technology	Haskell	Programming language used to implement the simple self-contained Nuthatch “interpreter“.

The *mini* Nuthatch DSL in Haskell

```haskell
-- Very small subset of Nuthatch DSL
data Walk  = Walk String [Stmt]

data Stmt  = Print [Expr]
           | Println [Expr]
           | If Exprb [Stmt] [Stmt]
           | WalkTo Int

data Expr  = Str String
           | Boolean Exprb
           | Value

data Exprb = Eq Expr Expr
           | Leaf
           | Down
           | Up
```

Using the "grammer" one can recreate the simple stringify example from [1]

```haskell
toStringWalk :: Walk
toStringWalk =
    Walk "toString"
    [ (If Leaf
        [(Print [Value])]
        [ (If Down
            [(Print [Value, (Str "(")])]
            [ (If Up
                [(Print [(Str ")")])]
                [(Print [(Str ", ")])]
              )
            ]
          )
        ]
      )
    ]
```

`run :: Tree -> Walk -> IO ()`

Function

- Takes a tree and a walk and executes the walk *"over"* the tree
  - For every node the walk is interpreted: see `eval`
  - The result is printed on the console (using `putStr`)
- The **Tree** data structure:

```
data Tree = Node String [Tree]
```

  - Every node contains a string value
  - Every node can have as many children as possible

`eval :: Ctx -> Walk -> (String, int)`

Function

- Evaluates a walk for a given context `Ctx`
- The `Ctx` captures the join point conditions of the current tree node for which the walk is executed

```
-- (Ctx value isLeaf isDown isUp)
data Ctx = Ctx String Bool Bool Bool
```

- String value of the node *(payload)*
- Join point **isLeaf**: `arity == 0`
- Join point **isDown**: `from == 0`
- Join point **isUp**: `leaf || from == last`

A Peek into `eval`

Runs all statements provided in the walk; every statement execution returns the String (from Print) and a Maybe in a tuple. The Maybe is Nothing in most cases but for WalkTo it contains the number of the next branch to take

```
eval :: Ctx -> Walk -> (String, Int)
eval c (Walk _ stmts) = retmap (reduce (execs c stmts))
    where
        execs :: Ctx -> [Stmt] -> [(String, Maybe Int)]
        execs c [] = []
        execs c (stmt:stmts) = [(evals c stmt)] ++ execs c stmts

        evals :: Ctx -> Stmt -> (String, Maybe Int)
        evals _ (WalkTo i)      = ("", Just i)
        evals c (Print ex1)     = (foldl (++) "" (map (evale c) ex1), Nothing)
        evals c (If b st1 st2) = if evalb c b
                                  then reduce (execs c st1)
                                  else reduce (execs c st2)

...
        evalb :: Ctx -> Exprb -> Bool
        evalb (Ctx _ x _ _) (Leaf) = x
        evalb (Ctx _ _ x _) (Down) = x

...
```

WalkTo has no String output but the number of the next branch to walk to

Evaluate boolean "constants" by looking them up

A Peek into run

```
run :: Tree -> Walk -> IO ()
run tree w = putStr (foldr (++) "\n" (base tree))
    where
...
       children parent t i =
           if (i < length t) then
              if (test (t !! i) True (i-1)) < 0 then
                  (render (t !! i) True (i-1)) ++
                  (internal (t !! i)) ++
                  (render parent False (i+1)) ++
                  children parent t (i+1)
              else if (test (t !! i) True (i-1)) == 0 then
                  ...
              else
                  render (t !! (getindex  (t !! i) True (i-1))) True
 … (getindex (t !! i) True (i-1)) ++
                  internal (t !! (getindex  (t !! i) True (i-1)))
           else
               []
...
       render (Node v t) d u = [fst (eval (makectx
 … v t d u) w)]
...
```

Default tree walk

On 0 skip subtree

Requested nodes

1.) Render the current node
2.) Render all children
3.) Go back up
4.) Next parent node

*1.) Render the **requested** node*
*2.) Render all children of the **requested** node*

Thank you for your attention.

Any Questions?

https://github.com/maxstrauch/sle-tree-walk

SLE Winter Term 2015/16, University of Koblenz-Landau

Maximilian Strauch

UNIVERSITÄT KOBLENZ · LANDAU

http://softlang.wikidot.com/course:sle1516

FYI only!

BACKUP

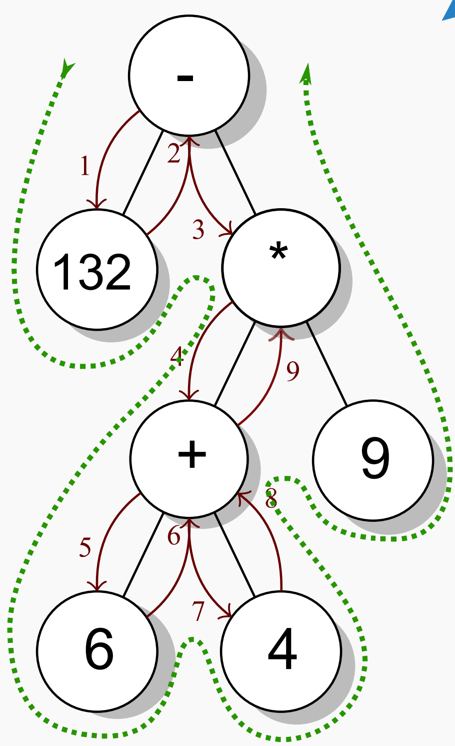
https://github.com/maxstrauch/sle-tree-walk

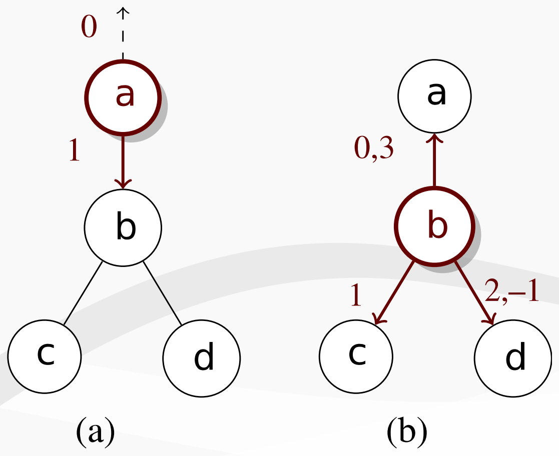
Source: [1]

What's a walk?

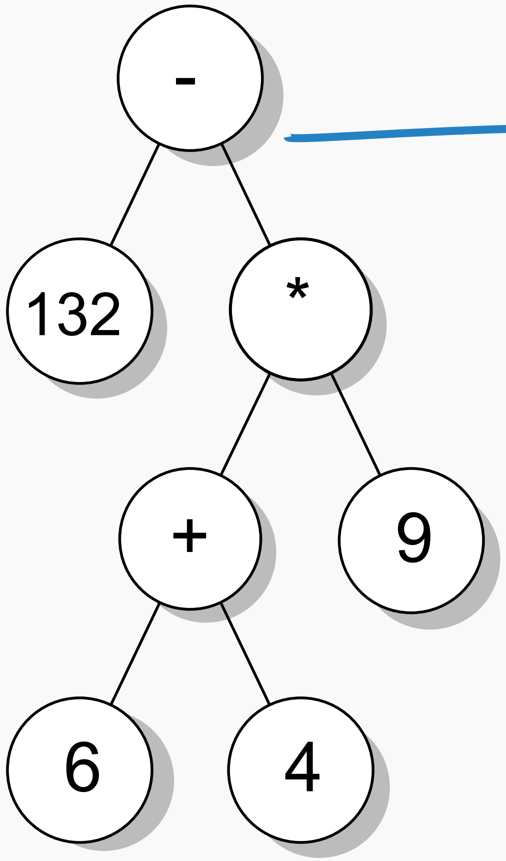
Flashback

- A **walk** *walks* along a tree, selects branches and mutates nodes (rewriting)
- Path: sequence of nodes; default Path: f–2–f–*–+–6–. . .
- If a walk comes to a node the inner statements of a walk are executed
  - Join point captures enter condition
  - Return value = next node





An example tree



```
data Tree = Node String [Tree]
```

… simply encode the tree as an instance of this data structure …

```
-- An excercise tree
atree :: Tree
atree =
    Node "-" [
        (Node "132" []),
        (Node "*" [
            (Node "+" [(Node "6" []), (Node "4" [])]),
            (Node "9" [])
        ])
    ]
```

`dump :: Walk -> IO ()`

Function

- Simple helper function to *pretty print* a walk in a more readable and bracket less style
- Invoking `dump toStringWalk` results in:

```
toStringWalk :: Walk
toStringWalk =
    Walk "toString"
    [ (If Leaf
        [(Print [Value])]
        [ (If Down
            [(Print [Value, (Str "(")])]
            [ (If Up
                [(Print [(Str ")")])]
                [(Print [(Str ", ")])]
              )
            ]
          )
        ]
      )
    ]
```

```
walk toString {
  if (leaf) {
    print value;
  } else {
    if (down) {
      print value + "(";
    } else {
      if (up) {
        print ")";
      } else {
        print ", ";
      }
    }
  }
}
```

References

- [1] A. H. Bagge, R. Lämmel: Walk Your Tree Any Way You Want. ICMT 2013. http://softlang.uni-koblenz.de/nuthatch/paper.pdf
- [2] A. H. Bagge: Analysis and transformation with the nuthatch tree-walking library. SLE Conference 2015. http://dl.acm.org/citation.cfm?doid=2814251.2814264
- [3] R. Lämmel: Language interpreters. Software Languages Team, CS Faculty, University of Koblenz-Landau. <*No URL available*>
- *Sitta Cashmirensis* imagery: https://commons.wikimedia.org/wiki/File:SittaCashmirensis.svg