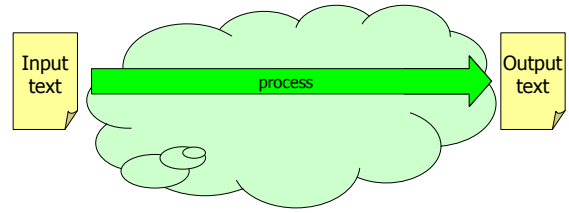
Tree Oriented Programming

Jeroen Fokker

Universiteit Utrecht

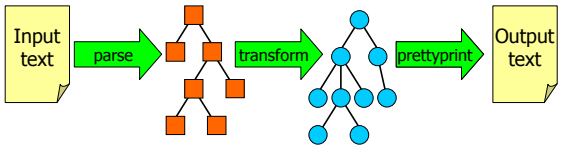
Tree oriented programming

- Many problems are like:



Tree oriented programming

- Many problems are like:



internal tree representation

Tree oriented programming tools

should facilitate:

- Defining trees
- Parsing
- Transforming
- Prettyprinting

Mainstream approach to tree oriented programming

- Defining trees — *OO programming language*
- Parsing — *preprocessor*
- Transforming — *clever hacking*
- Prettyprinting — *library*

Our approach to tree oriented programming

- Defining trees — *functional language* Haskell
- Parsing — *library*
- Transforming — *preprocessor*
- Prettyprinting — *library*

This morning's programme

- A crash course in
  Functional programming using Haskell
- Defining trees in Haskell
- The parsing library
- Transforming trees
  using the UU Attribute Grammar Compiler
- Prettyprinting
- Epilogue: Research opportunities

Language evolution: Imperative & Functional

- 50 years ago

- Now

Haskell

Part I

A crash course in
Functional programming
using Haskell

Function definition

```
fac :: Int → Int
fac  n  =  product [1..n]
```

Haskell

```
static int fac (int n)
{   int count, res;
    res = 1;
    for (count=1; count<=n; count++)
        res *= count;
    return res;
}
```

JAVA

Definition forms

- Function

```
fac :: Int → Int
fac  n  =  product [1..n]
```

- Constant

```
pi :: Float
pi  =  3.1415926535
```

- Operator

```
( !^! ) :: Int → Int → Int
n !^! k  = fac n
           / (fac k * fac (n-k))
```

Case distinction with guards

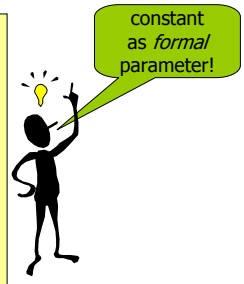
```
abs :: Int → Int

abs  x  | x>=0   =  x
        | x<0    =  -x
```

"guards"

Case distinction with patterns

```
day :: Int → String
day  1  =  "Monday"
day  2  =  "Tuesday"
day  3  =  "Wednesday"
day  4  =  "Thursday"
day  5  =  "Friday"
day  6  =  "Saturday"
day  7  =  "Sunday"
```



Iteration

```
fac :: Int → Int

fac  n  | n==0   =  1
        | n>0    =  n * fac (n-1)
```

without using standard function *product*

recursion

List: a built-in data structure

List: 0 or more values of the same type

- "empty list" constant: **[ ]**
- "put in front" operator: **:**

Shorthand notation for lists

- enumeration: [1, 3, 8, 2, 5]

```
> 1 : [2, 3, 4]
[1, 2, 3, 4]
```

- range: [4 .. 9]

```
> 1 : [4..6]
[1, 4, 5, 6]
```

Functions on lists

```
sum :: [Int] → Int
sum  [ ]        =  0
sum  (x:xs)     =  x + sum xs
```

```
length :: [Int] → Int
length  [ ]      =  0
length  (x:xs)   =  1 + length xs
```

patterns

recursion

Standard library of functions on lists

- null

```
> null [ ]
True
```

- ++

```
> [1,2] ++ [3,4,5]
[1, 2, 3, 4, 5]
```

- take

```
> take 3  [2..10]
[2, 3, 4]
```

challenge: Define these functions, using pattern matching and recursion

Functions on lists

```
null  ::  [a] → Bool
null  [ ]        =  True
null  (x:xs)     =  False
```

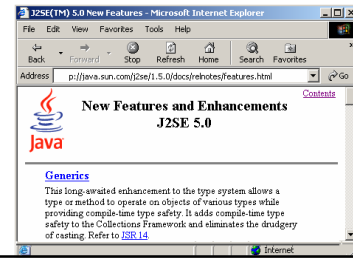
```
(++)  ::  [a] → [a] → [a]
[ ]      ++ ys   =  ys
(x:xs) ++ ys     =  x : (xs++ys)
```

```
take ::  Int → [a] → [a]
take  0  xs       = [ ]
take  n  [ ]      = [ ]
take  n  (x:xs)   = x : take (n-1) xs
```

Polymorphic type

- Type involving *type variables*

```
take ::  Int → [a] → [a]
```



Why did it take 10 years and 5 versions to put this in Java?

Functions as parameter

- Apply a function to all elements of a list

```
map
```

```
> map  fac  [1, 2, 3, 4, 5]
[1, 2, 6, 24, 120]
```

```
> map  sqrt  [1.0, 2.0, 3.0, 4.0]
[1.0, 1.41421, 1.73205, 2.0]
```

```
> map  even  [1 .. 6]
[False, True, False, True, False, True]
```

Challenge

- What is the type of *map* ?

```
map ::  (a→b) →  [a]  →  [b]
```

- What is the definition of *map* ?

```
map  f  [ ]      =  [ ]
map  f  (x:xs)   =  f  x  :  map  f  xs
```

Another list function: *filter*

- Selects list elements that fulfill a given predicate

```
> filter  even  [1 .. 10]
[2, 4, 6, 8, 10]
```

```
filter ::  (a→Bool) → [a] → [a]
filter  p  [ ]                =  [ ]
filter  p  (x:xs)  | p x      =  x : filter p xs
                   | True     =  filter p xs
```

Higher order functions: repetitive pattern? Parameterize!

```
product ::  [Int] → Int
product  [ ]      =  1
product  (x:xs) =  x  *  product xs
```

```
and      ::  [Bool] → Bool
and      [ ]      =  True
and      (x:xs)   =  x  &&  and xs
```

```
sum      ::  [Int] → Int
sum      [ ]      =  0
sum      (x:xs)   =  x  +  sum xs
```

Universal list traversal: *foldr*

```
foldr :: (a→b→b) → b → [a] → b
```

combining function

start value

```
foldr (#) e [ ]     = e
foldr (#) e (x:xs)= x # foldr (#) e xs
```

Partial parameterization

- *foldr* is a generalization of *sum*, *product*, and *and* ....
- ...thus *sum*, *product*, and *and* are special cases of *foldr*

```
product  = foldr (*)  1
and      = foldr (&&) True
sum      = foldr (+)  0
or       = foldr (||) False
```

Example: sorting (1/2)

```
insert :: Ord a ⇒ a → [a] → [a]
insert e [ ]      = [ e ]
insert e (x:xs)
      | e ≤ x     = e : x : xs
      | e ≥ x     = x : insert e xs
```

```
isort :: Ord a ⇒ [a] → [a]
isort [ ]         = [ ]
isort (x:xs)      = insert x (isort xs)
```

```
isort             = foldr insert [ ]
```

Example: sorting (2/2)

```
qsort :: Ord a ⇒ [a] → [a] → [a]
qsort [ ]         = [ ]
qsort (x:xs)      = qsort (filter (<x) xs)
                    ++ [x] ++
                    qsort (filter (≥x) xs)
```

(Why don't they teach it like that in the algorithms course?)

Infinite lists

```
repeat    :: a → [a]
repeat  x  =  x : repeat x
```

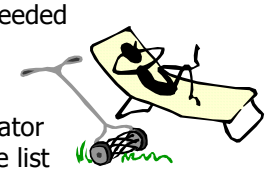
```
> repeat  3
[3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,
```

```
replicate  :: Int → a → [a]
replicate  n  x  =  take n (repeat x)
```

```
> concat (replicate  5  "IPA " )
"IPA IPA IPA IPA IPA "
```

Lazy evaluation

- Parameter evaluation is postponed until they are really needed



- Also for the (:) operator so only the part of the list that is needed is evaluated

Generic iteration

```
iterate     ::  (a→a) → a → [a]
iterate  f  x  =   x : iterate  f  (f x)
```

```
> iterate  (+1)  3
[3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,2
```

Convenient notations (borrowed from mathematics)

- Lambda abstraction

```
\x → x*x
```

for creating anonymous functions

- List comprehension

```
[ x*y  |  x ← [1..10]
       ,  even x
       ,  y ← [1..x]
]
```

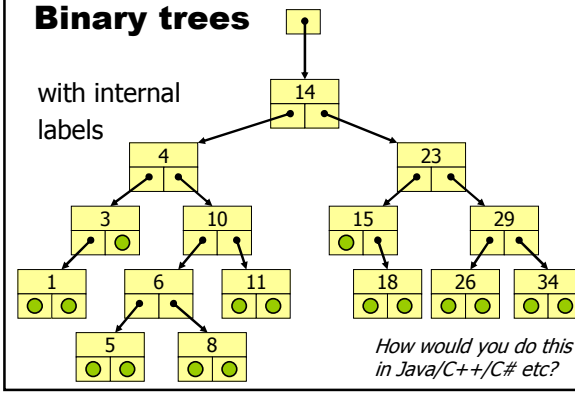
more intuitive than equivalent expression using *map* , *filter* & *concat*

Part II

Defining trees in Haskell

Binary trees

with internal labels



How would you do this in Java/C++/C# etc?

The OO approach to trees

```
class Tree
{     private Tree left, right;
      private int value;

      // constructor
      public Tree(Tree al, Tree ar, int av)
      {     left = al; right=ar; value=av;  }

      // leafs are represented as null
}
```

The OO approach to trees: binary trees with external labels

```
class Tree {
      // empty superclass
}
class Leaf extends Tree {
      int value
}
class Node extends Tree {
      Tree left,right
}
```

Functional approach to trees

- I need a polymorphic type Tree a
- and constructor functions

```
Leaf  ::    a                  → Tree a
Node :: Tree a → Tree a → Tree a
```

- Haskell notation:

```
data Tree a
= Leaf     a
 | Node  (Tree a)  (Tree a)
```

Example

- Data types needed in a compiler for a simple imperative language

```
data Stat
= Assign    Name Expr
 | Call     Name [Expr]
 | If       Expr Stat
 | While    Expr Stat
 | Block    [Stat]
```

```
type Name = String
```

```
data Expr
= Const    Int
 | Var     Name
 | Form    Expr Op Expr
```

```
data Op
= Plus | Min | Mul | Div
```

Functions on trees

- In analogy to functions on lists

```
length :: [a] → Int
length  [ ]      =  0
length  (x:xs)   =  1 + length xs
```

- we can define functions on trees

```
size :: Tree a → Int
size  (Leaf v)       =  1
size  (Node lef rit) =  size lef + size rit
```

Challenge: write tree functions

- *elem* tests element occurrence in tree

```
elem ::   Eq a ⇒   a → Tree a → Bool
elem  x  (Leaf y)        =  x==y
elem  x  (Node lef rit)  =  elem x lef || elem x rit
```

- *front* collects all values in a list

```
front ::   Tree a → [a]
front  (Leaf y)          =  [ y ]
front  (Node lef rit)    =  front lef ++ front rit
```

A generic tree traversal

- In analogy to *foldr* on lists

```
foldr ::    (a→b→b)  →      -- for (:)
            b        →      -- for [ ]
            [a]     →  b
```

- we can define *foldT* on trees

```
foldT ::    (a→b)    →      -- for Leaf
            (b→b→b)  →      -- for Node
            Tree a  →  b
```

Challenge: rewrite *elem* and *front* using *foldT*

```
foldT ::    (a→b)    →      -- for Leaf
            (b→b→b)  →      -- for Node
            Tree a  →  b
```

```
elem  x  (Leaf y)        =  x==y
elem  x  (Node lef rit)  =  elem x lef || elem x rit
```

```
elem  x                  =  foldT  (==x)  (||)
```

```
front  (Leaf y)          =  [ y ]
front  (Node lef rit)    =  front lef ++ front rit
```

```
front                    =  foldT  ( :[] )  (++)
```

Part III

A Haskell
Parsing library

Approaches to parsing

- Mainstream approach (imperative)
  - Special notation for grammars
  - Preprocessor translates grammar to C/Java/…
    - -YACC *(Yet Another Compiler Compiler)*
    - -ANTLR *(ANother Tool for Language Recognition)*
- Our approach (functional)
  - Library of grammar-manipulating functions

ANTLR generates Java from grammar

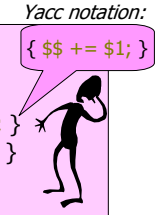
```
Expr  :  Term
         ( PLUS   Term
         | MINUS Term
         ) *
      ;
Term  :  NUMBER
      |  OPEN
         Expr
         CLOSE
      ;
```

```
public void expr ()
{ term ();
  loop1: while (true) {
    switch(sym) {
    case PLUS:
      match(PLUS); term (); break;
    case MINUS:
      match(MINUS); term (); break;
    default:
      break loop1;
    }
  }
}
public void term()
{  switch(sym) {
    case INT: match(NUMBER); break;
    case LPAREN: match(OPEN); expr ();
                 match(CLOSE); break;
    default: throw new ParseError();
  }
}
```

ANTLR: adding semantics

Yacc notation:

```
Expr   returns [int x=0]
       { int y; }
:  x= Term
       (  PLUS   y=Term    { x += y; }
       |  MINUS  y=Term    { x –= y; }
       ) *
;
Term  returns [int x=0]
:   n: NUMBER  { x = str2int(n.getText(); }
|      OPEN   x= Expr    CLOSE
;
```



A Haskell parsing library

type Parser

- Building blocks

epsilon :: Parser

symbol :: a → Parser
satisfy :: (a→Bool) → Parser

- Combinators

(⊕) :: Parser → Parser → Parser
(⊗) :: Parser → Parser → Parser

A Haskell parsing library

type Parser a b

start :: Parser a b → [a]→ b

- Building blocks

epsilon :: Parser a ()

symbol :: a → Parser a a
satisfy :: (a→Bool) → Parser a a

- Combinators

(⊕) :: Parser a b → Parser a b → Parser a b
(⊗) :: Parser a b → Parser a c → Parser a (b,c)
(®) :: (b→c) → Parser a b → Parser a c

Domainspecific Language vs. Combinator Library

- New notation and semantics
- Preprocessing phase
- What you got is all you get

- Familiar syntax, just new functions
- 'Link & go'
- Extensible at will using existing function abstraction mechnism

Expression parser

```
open   = symbol '('
close  = symbol ')'
plus   = symbol '+'
minus  = symbol '–'
```

```
data Tree
= Leaf  Int
| Node Tree Op Tree
type Op = Char
```

```
expr, term :: Parser Char Tree
expr   =  Node  ® term ⊗ (plus⊕minus) ⊗ expr
       ⊕        term
term   =  Leaf  ® number
       ⊕middle ® open ⊗ expr ⊗ close
          where middle (x,(y,z)) = y
```

Example of extensibility

- Shorthand

```
open   = symbol '('
close  = symbol ')'
```

- Parameterized shorthand

```
pack   :: Parser a b → Parser a b
pack p = middle ®  open ⊗ p ⊗ close
```

- New combinators

```
many   :: Parser a b → Parser a [b]
```

The real type of (⊗)

How to combine b and c ?

```
(⊕)   :: Parser a b → Parser a b → Parser a b
(⊗)   :: Parser a b → Parser a c → Parser a (b,c)
(®)   :: (b→c)      → Parser a b → Parser a c
```

```
(⊗)   :: Parser a b → Parser a c →
                      (b→c→d) → Parser a d
```

```
(⊗)   :: Parser a (c→d)
                  → Parser a c  → Parser a d
```

```
pack p  =  middle  ®  open ⊗ p ⊗ close
        where middle x y z = y
```

Another parser example; design of a new combinator

```
many  ::  Parser a b → Parser a [b]

many p  =    (\b bs→ b:bs)  ®
             p ⊗ many p
        ⊕    (\e→ [ ])  ®
             epsilon

many p  =  (:) ®  p ⊗ many p  ⊕  succeed [ ]
```

Challenge: parser combinator design

EBNF * | EBNF + | Beyond EBNF

```
many     :   Parser a b    → Parser a [b]
many1    ::  Parser a b    → Parser a [b]
sequence :: [ Parser a b ] → Parser a [b]

many1 p  =   (:)  ®
             p ⊗ many p

sequence [ ]      =   succeed  [ ]
sequence (p:ps)   =   (:)  ®  p  ⊗  sequence ps

sequence  =  foldr  f  (succeed [])
      where f p r  =  (:) ® p ⊗ r
```

More parser combinators

separator

```
sequence  :: [ Parser a b ]  → Parser a [b]
choice    :: [ Parser a b ]  → Parser a [b]
listOf  :: Parser a b → Parser a s → Parser a [b]
chain  :: Parser a b → Parser a (b→b→b)
                                → Parser a b
choice     = foldr (⊕) fail
listOf  p s  =     (:)  ®
                   p
              ⊗    many  ( (\s b → b)  ®
                           s ⊗ p
                         )
```

Example: Expressions with precedence

```
data  Expr  =    Con   Int
             |   Var   String
             |   Fun   String  [Expr]
             |   Expr   :+:   Expr
             |   Expr   :–:   Expr
             |   Expr   :*:   Expr
             |   Expr   :/:   Expr
```

Method call

Parser should resolve precedences

Parser for Expressions (with precedence)

```
expr  = chain  term   (  (\o→(:+:)) ®(symbol '+')
                      ⊕ (\o→(:–:)) ® (symbol '–')
                      )
term  = chain  fact   (   (\o→(:*:)) ®(symbol '*')
                      ⊕  (\o→(:/:)) ® (symbol '/')
                      )
fact   =   Con ®  number
       ⊕          pack expr
       ⊕   Var ®  name
       ⊕   Fun ®  name ⊗ pack (listOf expr
                                  (symbol ',') )
```

A programmers' reflex: Generalize!

```
expr  = chain  term    (      ... (:+:)...'+' ...
                       ⊕      ... (:–:)...'–' ...
                       )
term  = chain  fact    (      ... (:*:)...'*' ...
                       ⊕      ... (:/:)...'/' ...
                       )
gen ops next
       = chain  next    (  choice ...ops... )

fact   =          basicCases
       ⊕          pack expr
```

Expression parser (many precedence levels)

```
expr   = gen ops1   term1
term1= gen ops2   term2
term2= gen ops3   term3
term3= gen ops4   term4
term4= gen ops5   fact
fact   =          basicCases
       ⊕          pack expr
```

expr = foldr gen fact [ops5,ops4,ops3,ops2,ops1]

```
gen ops next
       = chain  next    (  choice ...ops... )
```

Library implementation

```
type Parser      =  String → X
type Parser b    =  String → b
type Parser b    =  String → (b, String)
type Parser a b  =  [a] → (b, [a])
type Parser a b  =  [a] → [ (b, [a]) ]
```

polymorphic result type

rest string

polymorphic alfabet

list of successes for ambiguity

Library implementation

```
(⊕) :: Parser a b → Parser a b → Parser a b
(p ⊕ q) xs = p xs ++ q xs
```

```
(⊗) :: Parser a (c→d) → Parser a c → Parser a d
(p ⊗ q) xs = [ ( f c , zs )
             | (f,ys) ← p xs
             , (c,zs) ← q ys
             ]
```

```
(®) :: (b→c) → Parser a b → Parser a c
(f ® p) xs = [ ( f b , ys )
             | (b,ys) ← p xs
             ]
```

Part IV

Techniques for Transforming trees

Data structure traversal

- In analogy to *foldr* on lists

```
foldr :: (a→b→b) →        -- for (:)
         b       →        -- for [ ]
         [a]     → b
```

- we can define *foldT* on binary trees

```
foldT :: (a→b)     →      -- for Leaf
         (b→b→b)   →      -- for Node
         Tree a    → b
```

Traversal of Expressions

```
data Expr
= Add Expr Expr
| Mul Expr Expr
| Con Int
```

```
type ESem b
= ( b → b → b
  , b → b → b
  , Int   → b )
```

```
foldE :: (b→b→b) →      -- for Add
         (b→b→b) →      -- for Mul
         (Int →b) →     -- for Con
         Expr → b
```

Traversal of Expressions

```
data Expr
= Add Expr Expr
| Mul Expr Expr
| Con Int
```

```
type ESem b
= ( b → b → b
  , b → b → b
  , Int   → b )
```

```
foldE :: ESem b → Expr → b
foldE (a,m,c)     = f where
  f (Add e1 e2)   = a (f e1) (f e2)
  f (Mul e1 e2)   = m (f e1) (f e2)
  f (Con n)       = c n
```

Using and defining Semantics

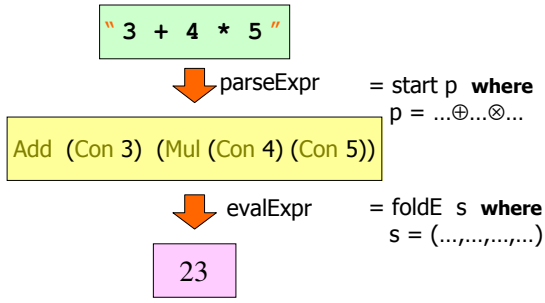
```
data Expr
= Add Expr Expr
| Mul Expr Expr
| Con Int
```

```
type ESem b
= ( b → b → b
  , b → b → b
  , Int   → b )
```

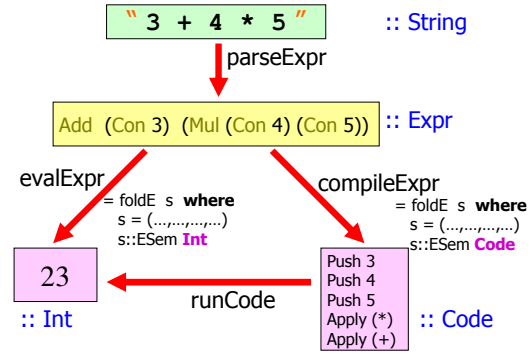
```
evalExpr :: Expr → Int
evalExpr = foldE evalSem
```

```
evalSem :: ESem Int
evalSem = ( (+) , (*) , id )
```

Syntax and Semantics



Multiple Semantics



A virtual machine

- What is "machine code" ?

```
type Code = [ Instr ]
```

- What is an "instruction" ?

```
data Instr = Push Int
           | Apply (Int→Int→Int)
```

Compiler generates Code

```
data Expr
= Add Expr Expr
| Mul Expr Expr
| Con Int
```

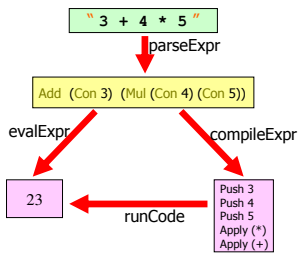
```
type ESem b
= ( b → b → b
  , b → b → b
  , Int     → b )
```

```
compExpr :: Expr → Code
compExpr = foldE compSem where
  compSem :: ESem Code
  compSem = ( add , mul , con )
```

```
mul :: Code → Code → Code
mul c1 c2 = c1 ++ c2 ++ [Apply (*)]
```

```
con n = [ Push n ]
```

Compiler correctness



runCode (compileExpr e)
=
evalExpr e

runCode: virtual machine specification

```
run :: Code → Stack → Stack
run     [ ]     stack = stack
run (instr:rest) stack = run rest ( exec instr stack )
```

```
exec :: Instr → Stack → Stack
exec (Push x)        stack     = x     : stack
exec (Apply f) (x:y:stack)     = f x y : stack
```

```
runCode :: Code → Int
runCode prog = hd ( run prog [ ])
```

Extending the example: variables and local def's

```
data Expr
= Add Expr Expr
| Mul Expr Expr
| Con Int
| Var String
| Def String Expr Expr
```

```
type ESem b
= ( b → b → b
  , b → b → b
  , Int → b
  , String → b
  , String →b→b→b )
```

```
evalExpr  :: Expr → Int
evalExpr  = foldE evalSem  where
  evalSem :: ESem Int
  evalSem = ( add , mul , con , var, def )
```

Any semantics for Expression

```
add :: b → b → b
add x y =
```

```
mul :: b → b → b
mul x y =
```

```
con :: Int → b
con n =
```

```
var :: String → b
var x =
```

```
def :: String → b → b → b
def x d b =
```

Evaluation semantics for Expression

```
add :: Int → Int → Int
add x y = x + y
```

```
mul :: Int → Int → Int
mul x y = x * y
```

```
con :: Int → Int
con n = n
```

```
var :: String → Int
var x =
```

```
def :: String → Int → Int → Int
def x d b =
```

Evaluation semantics for Expression

```
add :: Int → Int → (Env→Int)
add x y = x + y
```

```
mul :: Int → Int → (Env→Int)
mul x y = x * y
```

```
con :: Int → (Env→Int)
con n = n
```

```
var :: String → (Env→Int)
var x = \e → lookup e x
```

```
def :: String → Int → Int → (EnvInt
def x d b =
```

Evaluation semantics for Expression

```
add :: (Env→Int) → (Env→Int) → (Env→Int)
add x y = \e → x e + y e
```

```
mul :: (Env→Int) → (Env→Int) → (Env→Int)
mul x y = \e → x e * y e
```

```
con :: Int → (Env→Int)
con n = \e → n
```

```
var :: String → (Env→Int)
var x = \e → lookup e x
```

```
def :: String →(Env→Int)→ (Env→Int) → (Env→Int)
def x d b = \e → b ((x,d e) : e )
```

Extending the virtual machine

- What is "machine code" ?

```
type Code = [ Instr ]
```

- What is an "instruction" ?

```
data Instr = Push Int
           | Apply (Int→Int→Int)
           | Load Adress
           | Store Adress
```

Compilation semantics for Expression

```
add  :: (Env→Code)→(Env→Code)→ Env → Code
add  x  y   =   \e → x e ++ y e ++ [Apply (+)]

mul  :: (Env→Code)→(Env→Code)→ Env → Code
mul  x  y   =   \e → x e ++ y e ++ [Apply (*)]

con  ::                Int     → Env → Code
con  n      =   \e → [Push n]

var  ::                String  → Env → Code
var  x      =   \e → [Load (lookup e x)]

def  :: String →(Env→Code)→(Env→Code)→Env → Code
def  x  d  b =   \e → d e++[Store a]++b ((x,a) : e )
```

where a = length e

Language: syntax and semantics

```
data  Expr
=  Add  Expr  Expr
|  Mul  Expr  Expr
|  Con  Int
|  Var  String
|  Def  String Expr Expr
```

```
type ESem  b
=  (  b → b          → b
   ,  b → b          → b
   ,  Int            → b
   ,  String         → b
   ,  String →b→b→   b
   )
```

```
compSem ::  ESem   (Env→Code)
compSem =  (f1, f2, f3, f4, f5) where ……
compile t  =  foldE  compSem  t  [ ]
```

Language: syntax and semantics

```
data  Expr
=  Add  Expr  Expr
|  Mul  Expr  Expr
|  Con  Int
|  Var  String
data Stat
=  Assign  String Expr
|  While   Expr   Stat
|  If      Expr   Stat  Stat
|  Block   [Stat]
```

```
type ESem  b  c
= (  b → b          → b
   ,  b → b          → b
   ,  Int            → b
   ,  String         → b
   )
  ,(  String → b     → c
   ,  b  → c         → c
   ,  b  → c → c     → c
   ,  [ c ]          → c
   )
  )
```

```
compSem ::  ESem   (Env→Code)  (Env→Code)
compSem =  ((f1, f2, f3, f4), (f5, f6, f7, f8)) ……
compile t  =  foldE  compSem  t  [ ]
```

Real-size example

```
data Module = ……
data Class  = ……
data Method = ……
data Stat   = ……
data Expr   = ……
data Decl   = ……
data Type   = ……
```

```
type ESem  a b c d e f
=  (  (…,…,…)
   ,  (…,…)
   ,  (…,…,…,…,…,…)
   ,  …
```

```
compSem ::  ESem   (…… → ……)
                   (…… → ……)
                   (…… → ……)
                   (…… → ……)
                   (…… → ……)
                   (…… → ……)
                   (…… → ……)
compSem =  (…dozens of functions…) ……
```

Attributes that are passed top-down

Attributes that are generated bottom-up

Tree semantics generated by Attribute Grammar

```
data Expr
=  Add    Expr    Expr
|  Var    String
|  …
```

```
codeSem =
( \ a b  →  \ e → a e ++ b e ++ [Apply (+)]
, \ x    →  \ e → [Load (lookup e x)]
, ……
```

```
DATA Expr
=  Add   a: Expr   b: Expr
|  Var   x: String
|  …
```

```
ATTR Expr  inh  e: Env
           syn  c: Code
```

```
SEM Expr
| Add    this.code = a.code ++ b.code
                    ++ [Apply (+)]
         a.e  =  this.e
         b.e  =  this.e

| Var    this.code = [Load (lookup e x)]
```

Explicit names for fields and attributes

Attribute value equations instead of functions

UU-AGC Attribute Grammar Compiler

- Preprocessor to Haskell
- Takes:
  - Attribute grammar
  - Attribute value definitions
- Generates:
  - **data**type, *fold* function and *Sem* type
  - Semantic function (many-tuple of functions)
  - Automatically inserts trival def's

a.e = **this**.e

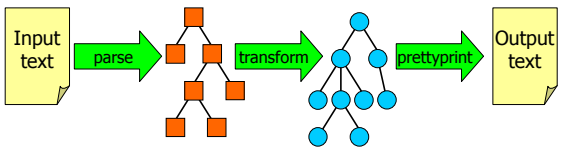
UU-AGC
Attribute Grammar Compiler

- Advantages:
  - Very intuitive view on trees
    no need to handle 27-tuples of functions
  - Still full Haskell power in attribute def's
  - Attribute def's can be arranged modularly
  - No need to write trivial attribute def's
- Disadvantages:
  - Separate preprocessing phase

Part IV

Pretty printing

Tree oriented programming



internal tree representation

Prettyprinting is just another tree transformation

- Example:
  transformation from Stat to String

```
DATA Stat
= Assign  a: Expr  b: Expr
| While   e: Expr  s: Stat
| Block   body: [Stat]
```

```
ATTR Expr Stat [Stat]
     syn  code: String
     inh  indent: Int
```

But how to handle newlines & indentation?

```
SEM Stat
| Assign  this.code = x.code ++ "=" ++ e.code ++ ";"
| While   this.code = "while (" ++ e.code ++ ")"++ s.code
| Block   this.code = "{" ++ body.code ++ "}"
```

```
SEM Stat
| While   s.indent  =  this.indent + 4
```

A combinator library for prettyprinting

- Type
- Building block
- Combinators
- Observer

```
type PPDoc
```

```
text :: String → PPDoc
```

```
(>|<)   ::  PPDoc → PPDoc → PPDoc
(>–<)   ::  PPDoc → PPDoc → PPDoc
indent  ::  Int   → PPDoc → PPDoc
```

```
render  ::  Int   →  PPDoc → String
```

Epilogue

Research opportunities

Universiteit Utrecht

Research opportunities (1/4)

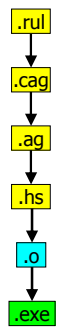
- Parsing library:
  - API-compatible to naïve library, but
  - With error-recovery etc.
  - Optimized
  - Implemented using the "Attribute Grammar" way of thinking

Research opportunities (2/4)

- UU - Attribute Grammar Compiler
  - More automatical insertions
  - Pass analysis ⇒ optimisation

Research opportunities (3/4)

- A real large compiler (for Haskell)
  - 6 intermediate datatypes
  - 5 transformations + many more
  - Learn about software engineering aspects of our methodology

Reasearch opportunities (4/4)

- Generate as much as possible with preprocessors
  - Attribute Grammar Compiler
  - Shuffle
    extract multiple views & docs from the same source
  - Ruler
    generate proof rules checked & executable