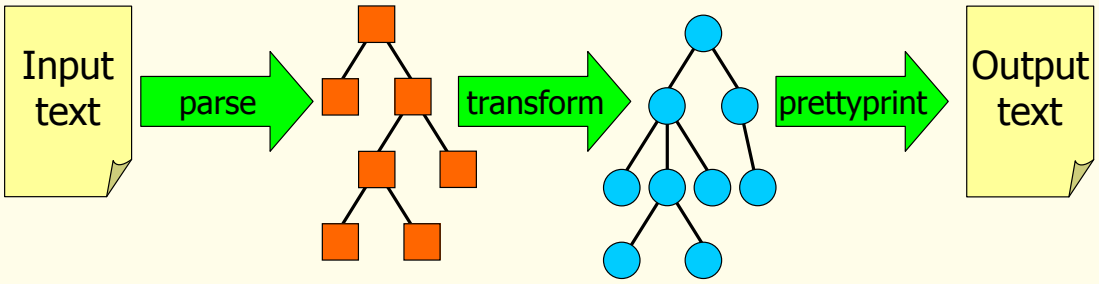
Tree Oriented Programming

the Haskell Utrecht Tools for compiler construction



Parse

```
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→c)→ Parser a b → Parser a c
(®)  :: (b→c)           → Parser a b → Parser a c
```

- Write parsers as grammars
- Add combinators beyond EBNF
- *Research: make an online, monadic, error recovering version*
- *Do analysis and transformation through Typed Abstract Syntax*

```
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
```

Transform

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

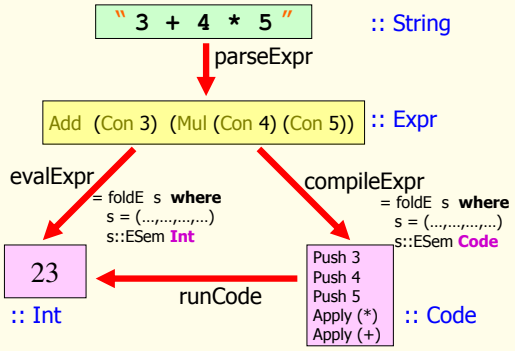
```
type ESem  b
= ( b → b → b
  , b → b → b
  , Int   → b  )
```

```
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
```

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

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

- Abstract from recursion in a 'fold' function
- Express semantics as a tuple of functions
- Every transformation is a call to 'fold'
- *Research: express a real compiler as 50 'fold's*




```
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

- Real semantics is large
- Attribute Grammar based preprocessor facilitates semantics definition
- *Research: optimise code generated by AG system*

Explicit names for fields and attributes

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

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

Attribute value equations instead of functions

```
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)]
```

generates

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

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

Pretty print

```
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
```

```
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
```

- Special case of transformation: String as target structure
- *Research: design class system which enables Typed Reflection*

NWO 'Hefboom' project 641.000.412,
NWO project 612.063.410,
Microsoft Research Scholarship

Prof. Dr. S. Doaitse Swierstra,
Dr. Atze Dijkstra, Drs. Jeroen Fokker,
Drs. Arie Middelkoop

http://www.cs.uu.nl/wiki/HUT