The UU_AG System

Programming with Functions, Aspects, Attributes, and Catamorphisms

Andres Löh

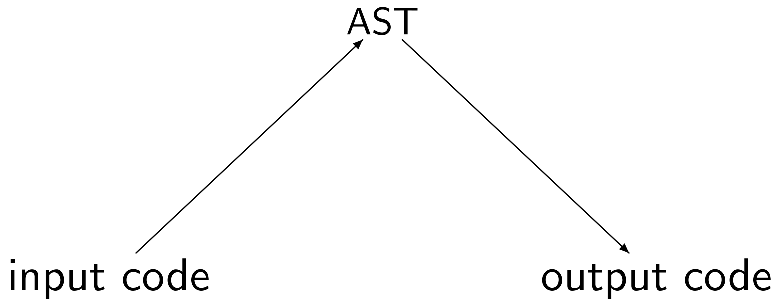
Institute of Information and Computing Science
Utrecht University
e-mail: andres@cs.uu.nl

June 27, 2001

(http://www.cs.uu.nl/~andres/talk3.ps)

A simplified view on compilers

- Input is transformed into output.
- Input and output language have little structure.
- During the process structure such as an Abstract Syntax Tree (AST) is created.



AST

input code

output code

Abstract syntax and grammars

- The structure in an abstract syntax tree is best described by a grammar.
- A concrete value (program) is then a word of the language defined by that grammar.

$$\begin{array}{lcl} Decimal & \rightarrow & Sign\ Digits \\ & \mid & Digits \end{array}$$

- The rules in a grammar are called **productions**. The right hand side of a rule is **derivable** from the left hand side.
- In each production a **nonterminal** is replaced by (**terminals** and/or) other nonterminals.
- A word is in the language defined by the grammar if it is derivable from the **root symbol** (or root nonterminal) in a finite number of steps.
- For convenience, we will always name the root symbol $Root$.

An example grammar

The following grammar describes the abstract syntax of a very simple language:

$$\begin{array}{rcl} \mathit{Root} & \longrightarrow & \mathit{Exprs} \\ \mathit{Exprs} & \longrightarrow & \mathit{Expr}\ \mathit{Exprs} \\ & | & \varepsilon \\ \mathit{Expr} & \longrightarrow & \mathit{Term} \\ \mathit{Term} & \longrightarrow & \mathit{String} \\ & | & \mathit{Term}\ \mathit{Term} \end{array}$$

- A program is a list of expressions.
- Each expression is a term.
- A term is either a string, or a concatenation of multiple strings.

Properties of Haskell I: Algebraic data types

- Haskell provides a powerful language construct to define own data types.
- Choice can be represented by introducing different **constructors**.
- Constructors may contain **fields**.
- It is possible to define **type constructors** by the introduction of type variables.
- It is possible to define **recursive types**.

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Bit} & = & \mathrm{Zero} \mid \mathrm{One} \\
\mathbf{data}\ \mathit{Complex} & = & \mathrm{Complex}\ \mathit{Real}\ \mathit{Real} \\
\mathbf{data}\ \mathit{Maybe}\ a & = & \mathit{Just}\ a \mid \mathit{Nothing} \\
\mathbf{data}\ \mathit{List}\ a & = & \mathrm{Nil} \mid \mathrm{Cons}\ a\ (\mathit{List}\ a)
\end{array}
$$

- There is a builtin list type with special syntax.

$$
\begin{array}{lcl}
\mathbf{data}\ [a] & = & [\,] \mid a : [a] \\
[1, 2, 3, 4, 5] & &
\end{array}
$$

Grammars correspond to datatypes

- Given this power, each nonterminal can be seen as a data type.
- The productions can be translated into definitions.
- Constructor names have to be invented.
- Abstraction is not needed, but recursion is.

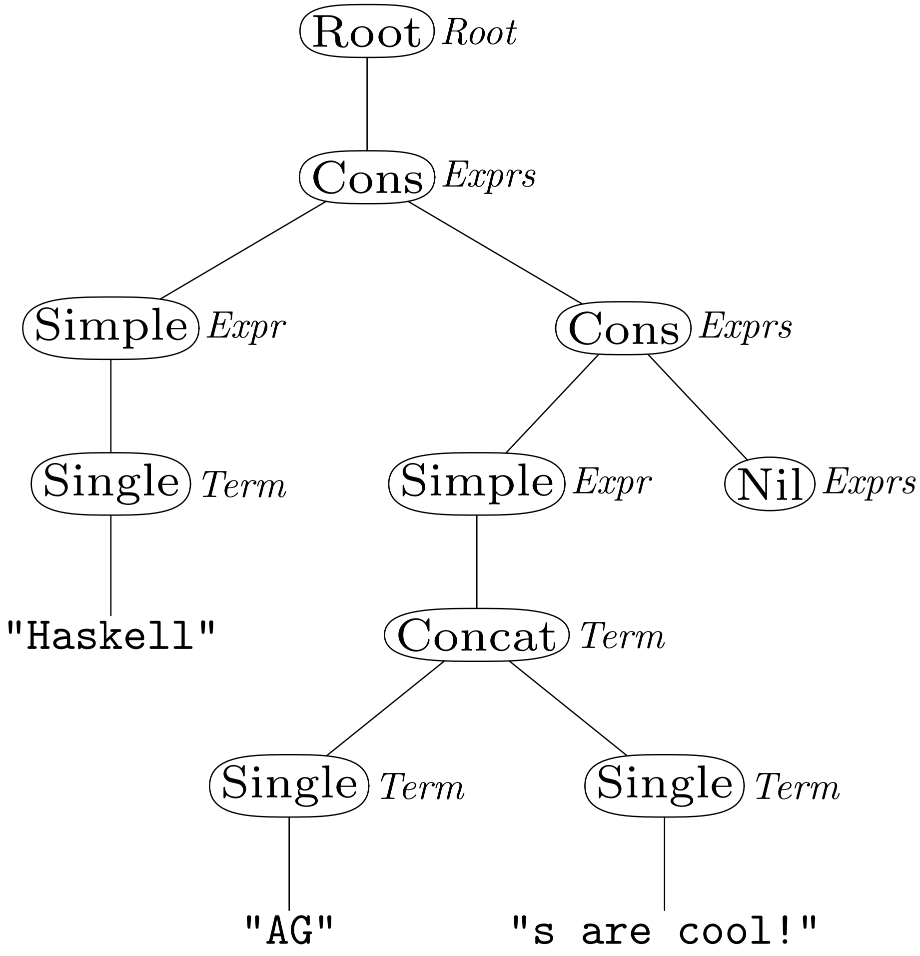
The example grammar translated

$$
\begin{array}{rcl}
\mathit{Root} & \rightarrow & \mathit{Exprs} \\
\mathit{Exprs} & \rightarrow & \mathit{Expr}\ \mathit{Exprs} \\
 & | & \varepsilon \\
\mathit{Expr} & \rightarrow & \mathit{Term} \\
\mathit{Term} & \rightarrow & \mathit{String} \\
 & | & \mathit{Term}\ \mathit{Term}
\end{array}
$$

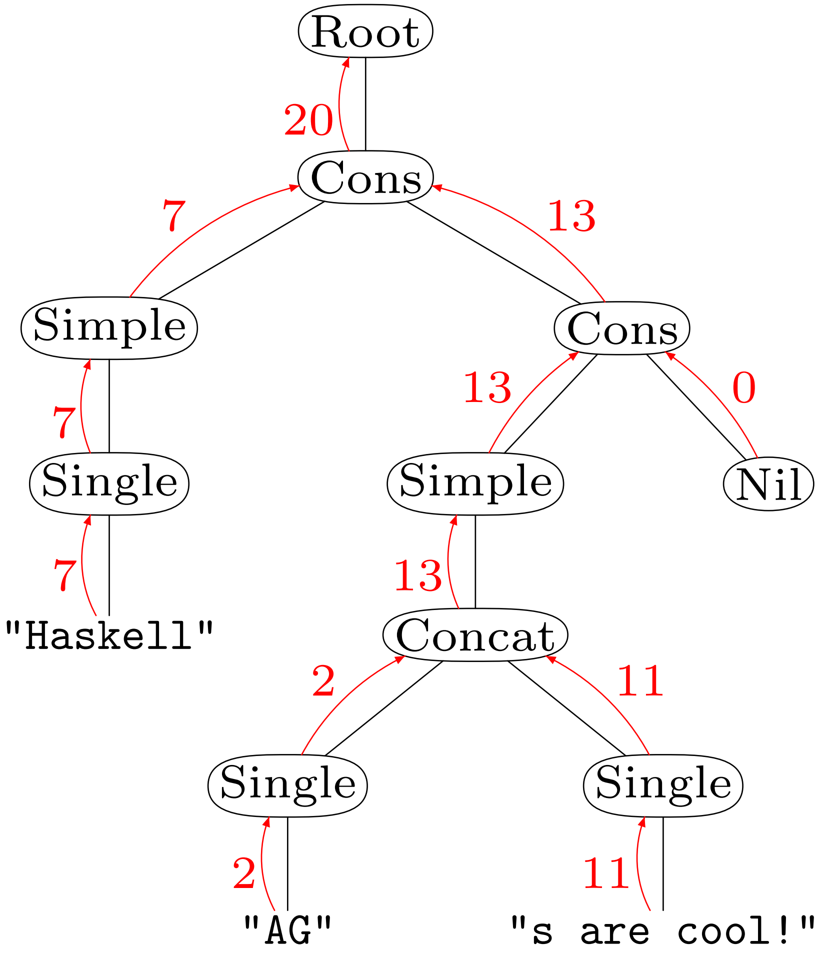
$$
\begin{array}{lcl}
\mathbf{DATA}\ \mathit{Root} & | & \mathrm{Root}\ \mathit{Exprs} \\
\mathbf{DATA}\ \mathit{Exprs} & | & \mathrm{Cons}\ \mathit{hd} : \mathit{Expr}\ \mathit{tl} : \mathit{Exprs} \\
 & | & \mathrm{Nil} \\
\mathbf{DATA}\ \mathit{Expr} & | & \mathrm{Simple}\ \mathit{Term} \\
\mathbf{DATA}\ \mathit{Term} & | & \mathrm{Single}\ \mathit{String} \\
 & | & \mathrm{Concat}\ \mathit{left} : \mathit{Term}\ \mathit{right} : \mathit{Term}
\end{array}
$$

- Data type definitions in UU_AG syntax are very similar (and, in fact, translated into) Haskell data type definitions.
- Fields may be given **field names**.
- ⟨Contrary to Haskell, UU_AG constructor names do not have to be unique.⟩

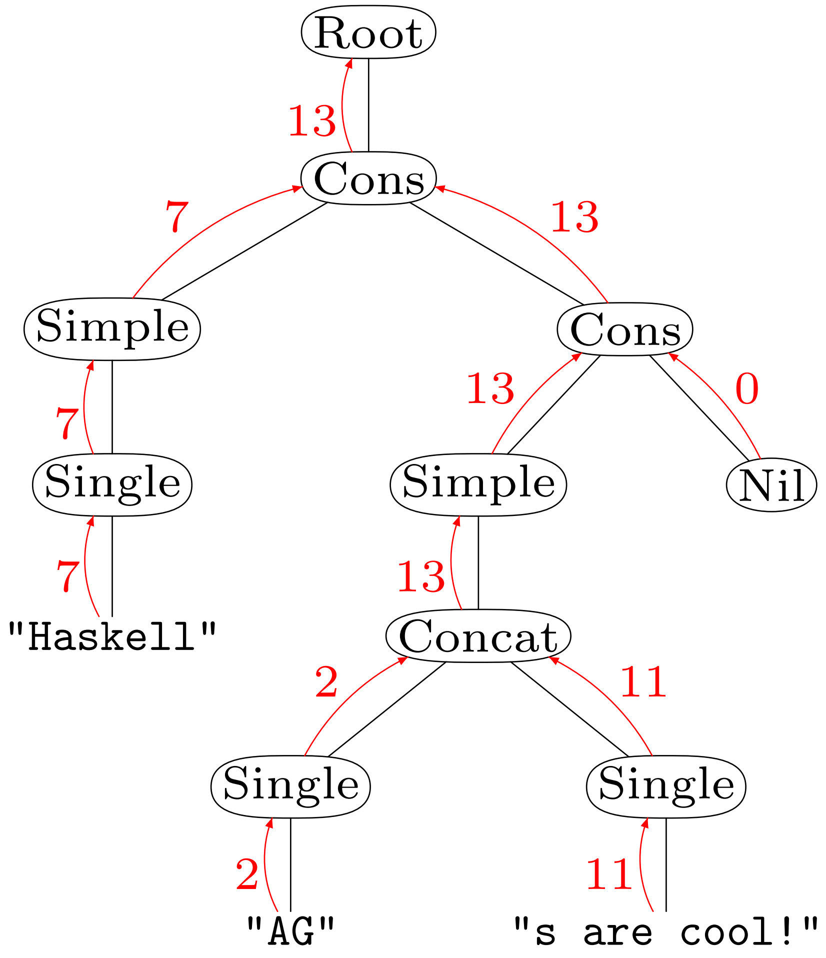
An example program



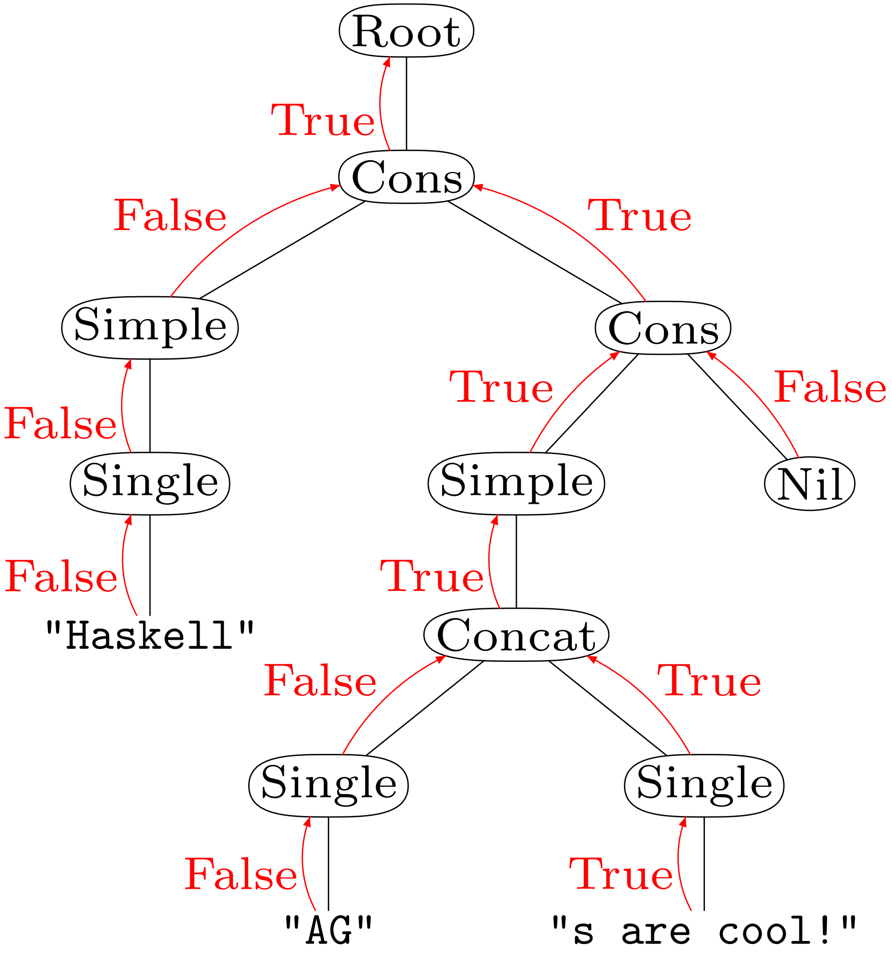
Computation follows structure I: Total length



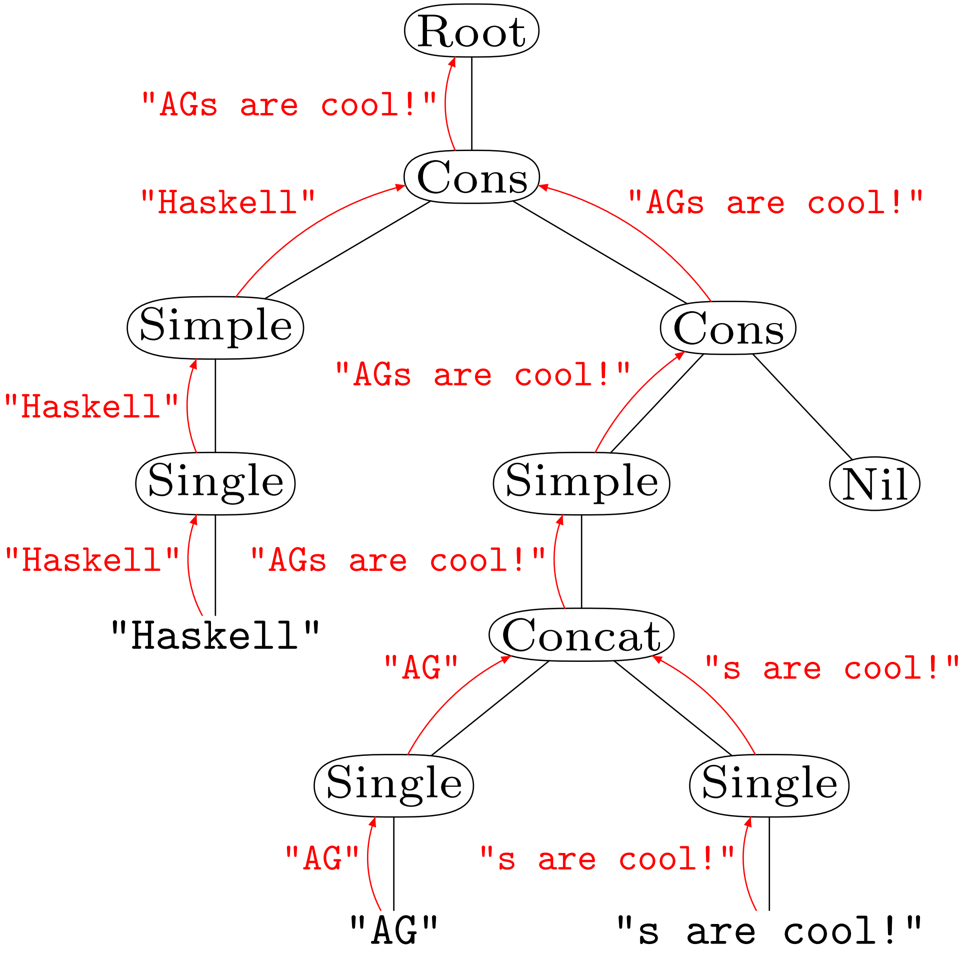
Computation follows structure II: Maximum length



Computation follows structure III: Spaces?



Computation follows structure IV: Value of last term



Computation follows structure — Observations

- Information is passed upwards.
- Constructors are replaced by operations.
- In many cases information is just copied unchanged.

Synthesised attributes

- In UU_AG, computations are modelled by **attributes**.
- Each of the examples defines an attribute.
- Attributes that are computed in a bottom-up fashion are called **synthesised attributes**.

```
ATTR Exprs Expr Term [|| maxlen : Int]
SEM Term
  | Single lhs.maxlen   =  length string
  | Concat lhs.maxlen   =  left.maxlen + right.maxlen
SEM Expr
  | Simple lhs.maxlen   =  term.maxlen
SEM Exprs
  | Cons lhs.maxlen     =  max hd.maxlen tl.maxlen
  | Nil lhs.maxlen      =  0
```

Synthesised attributes — continued

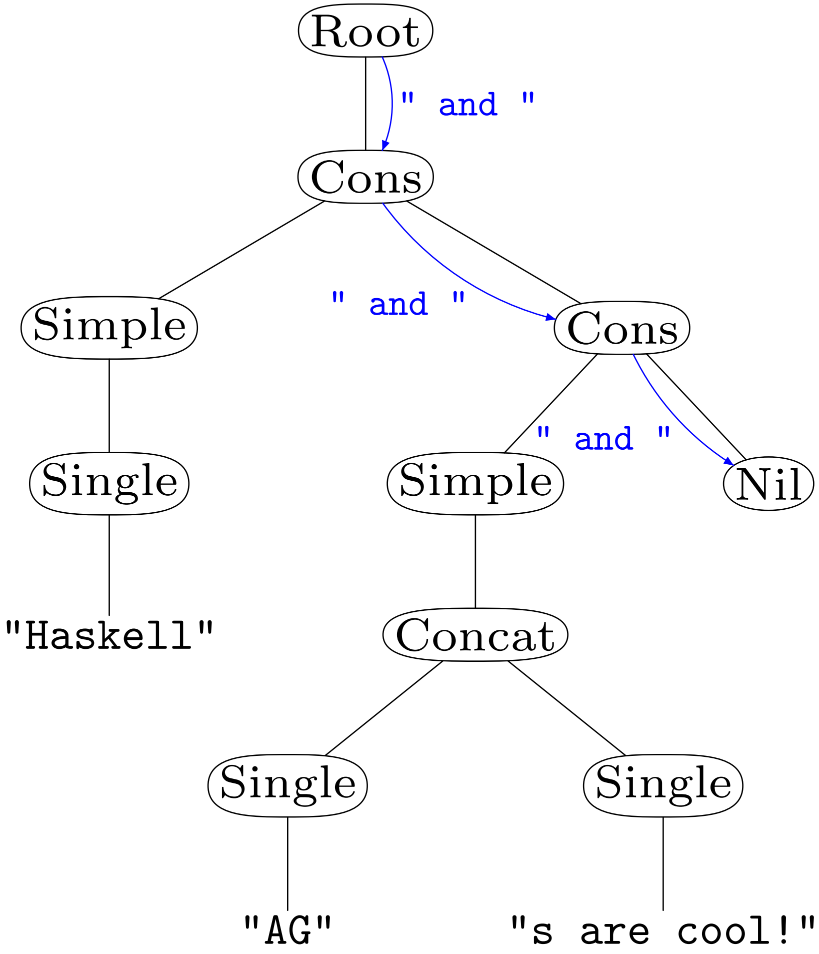
- Different attributes (and their semantics) can be defined separately, but can interact (be defined in terms of other attributes).
- The `UU_AG` system provides **copy rules** to eliminate trivial equations.

```
SEM Exprs [|| isEmpty : Bool]
  | Cons lhs.isEmpty    =  False
  | Nil lhs.isEmpty     =  True
ATTR Exprs Expr Term [|| lastval : String]
SEM Term
  | Single lhs.lastval  =  string
  | Concat lhs.lastval  =  left.lastval ⧺ right.lastval
SEM Exprs
  | Cons lhs.lastval    =  if tl.isEmpty then hd.lastval
                                         else tl.lastval
  | Nil lhs.lastval     =  error "no␣term␣in␣program"
```

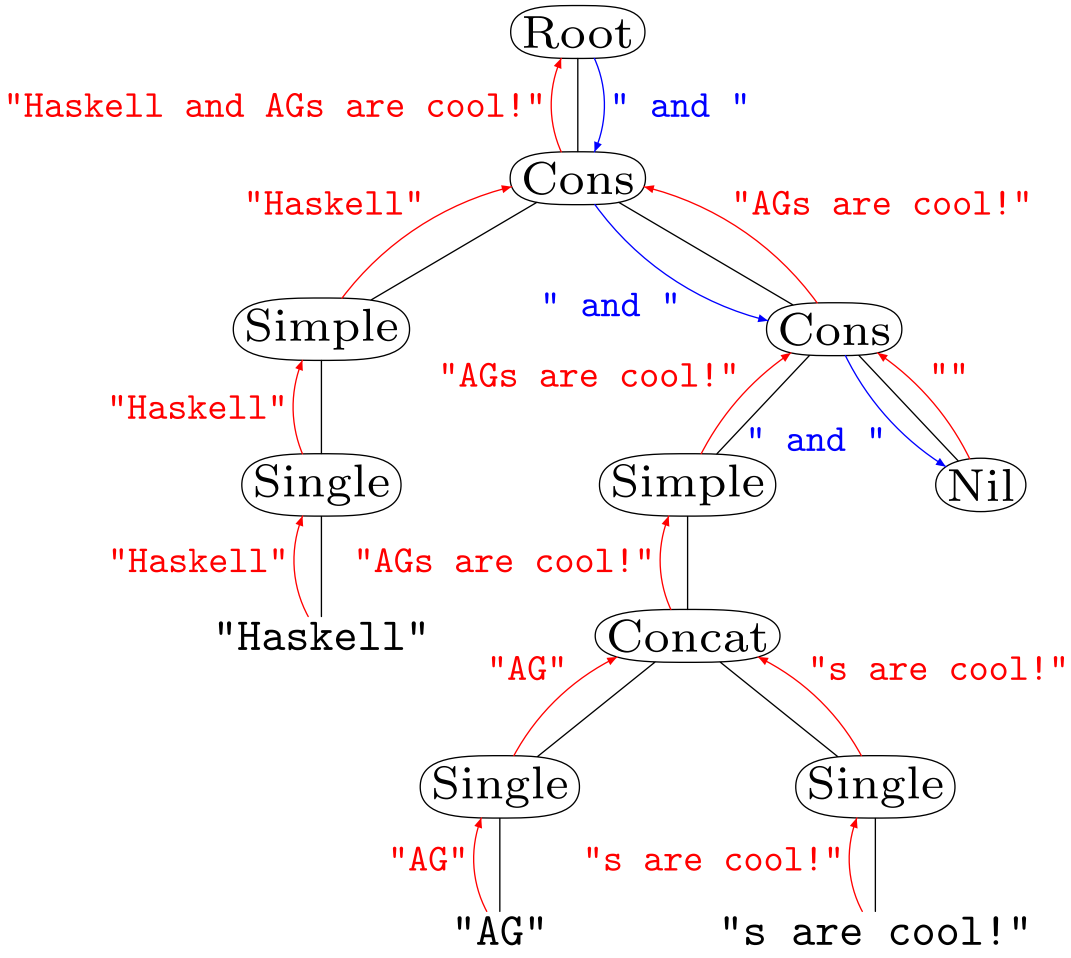
Distributing information

- Sometimes synthesised attributes depend on outside information.
- Examples: Options, parameters, results of other computations.
- In these cases it is not sufficient to pass information bottom-up. We need top-down attributes, too!

Example: Joining strings



Example: Joining strings — continued



Inherited attributes

- In attribute grammars, top-down attributes are called **inherited attributes**.
- In UU_AG, inherited attributes can be defined with the help of the **ATTR** and **SEM** statements, just like synthesised attributes.
- Again, for the downward distribution of inherited attributes there are copy rules that save some typing.
- Attributes can be inherited and synthesised at the same time. They are then called **chained attriutes**.

```
SEM Root
  | Root exprs.joinword   =  "␣and␣"
SEM Exprs [joinsep : String || joinval : String]
  | Cons tl.joinsep       =  lhs.joinsep
         lhs.joinval      =  if tl.isEmpty
                             then hd.lastval
                             else hd.lastval ++ lhs.joinsep ++ tl.joinval
  | Nil lhs.joinval       =  ""
```

Properties of Haskell II: Higher-order functions

- In functional languages functions are first-class values. In short: you can treat a function like any other value.
- Functions can be results of functions.

$$
\begin{array}{lcl}
(+) & :: & \mathit{Int} \rightarrow (\mathit{Int} \rightarrow \mathit{Int}) \\
(+)\ 2 & :: & \mathit{Int} \rightarrow \mathit{Int} \\
(+)\ 2\ 3 & :: & \mathit{Int}
\end{array}
$$

- Functions can be arguments of functions.

$$
\begin{array}{lcl}
\mathit{twice} & :: & (a \rightarrow a) \rightarrow (a \rightarrow a) \\
\mathit{twice}\ f\ x & = & f\ (f\ x)
\end{array}
$$

$$\mathit{twice}\ ((+)\ 17)\ 8 \equiv 42$$

$$
\begin{array}{lcl}
\mathit{map} & :: & (a \rightarrow b) \rightarrow ([a] \rightarrow [b]) \\
\mathit{map}\ f\ [\,] & = & [\,] \\
\mathit{map}\ f\ (x : xs) & = & f\ x : \mathit{map}\ f\ xs
\end{array}
$$

Catamorphisms

- A **catamorphism** is a function that computes a result out of a value of a data type by
  - replacing the constructors with operations
  - replacing recursive occurences by recursive calls to the catamorphism
- Since Haskell provides algebraic data types, catamorphisms can be written easily in Haskell.
- Sythesised attributes can be translated into catamorphisms in a straight-forward way.

Example translation

$$
\begin{array}{lcl}
\mathit{maxlen\_Root} & :: & \mathit{Root} \rightarrow \mathit{Int} \\
\mathit{maxlen\_Root}\ (\mathit{Root}\ \mathit{exprs}) & = & \mathit{maxlen\_Exprs}\ \mathit{exprs} \\
\mathit{maxlen\_Exprs} & :: & \mathit{Exprs} \rightarrow \mathit{Int} \\
\mathit{maxlen\_Exprs}\ (\mathrm{Cons}\ \mathit{hd}\ \mathit{tl}) & = & \mathbf{let}\ \mathit{hd\_maxlen} = \mathit{maxlen\_Expr} \\
 & & \qquad \mathit{tl\_maxlen} = \mathit{maxlen\_Exprs} \\
 & & \mathbf{in}\ \mathit{max}\ \mathit{hd\_maxlen}\ \mathit{tl\_maxlen} \\
\mathit{maxlen\_Exprs}\ \mathrm{Nil} & = & 0 \\
\mathit{maxlen\_Expr} & :: & \mathit{Expr} \rightarrow \mathit{Int} \\
\mathit{maxlen\_Expr}\ (\mathrm{Simple}\ \mathit{term}) & = & \mathit{maxlen\_Term}\ \mathit{term} \\
\mathit{maxlen\_Term} & :: & \mathit{Term} \rightarrow \mathit{Int} \\
\mathit{maxlen\_Term}\ (\mathrm{Single}\ \mathit{string}) & = & \mathit{length}\ \mathit{string} \\
\mathit{maxlen\_Term}\ (\mathrm{Concat}\ \mathit{left}\ \mathit{right}) & = & \mathbf{let}\ \mathit{left\_maxlen} = \mathit{maxlen\_Term} \\
 & & \qquad \mathit{right\_maxlen} = \mathit{maxlen\_Term} \\
 & & \mathbf{in}\ \mathit{left\_maxlen} + \mathit{right\_maxlen}
\end{array}
$$

Catamorphisms can be combined!

- Several attributes: Several catamorphisms?
- Better: Write one catamorphism computing a tuple!
  + only one traversal of the tree, attributes can depend on each other

```
SEM Exprs [|| isEmpty : Bool lastval : String]
  | Cons lhs.isEmpty      =  True
         lhs.lastval      =  if tl.isEmpty then hd.lastval
                                           else tl.lastval
sem_Exprs                 ::  Exprs → (Bool, String)
sem_Exprs (Cons hd tl)    =  let (tl_isEmpty, tl_lastval) = sem_Exprs tl
                                 hd_lastval = sem_Expr hd
                             in (False
                                ,if tl_isEmpty then hd_lastval
                                               else tl_lastval
                                )
```

Catamorphisms can compute functions!

- Inherited attributes can be realised by computing functional values.
- In fact, a group of inherited and synthesised attributes is isomorphic to one synthesised attribute with a functional value.
- The inherited attributes get mapped to the synthesised attributes.

Catamorphisms can compute functions! — continued

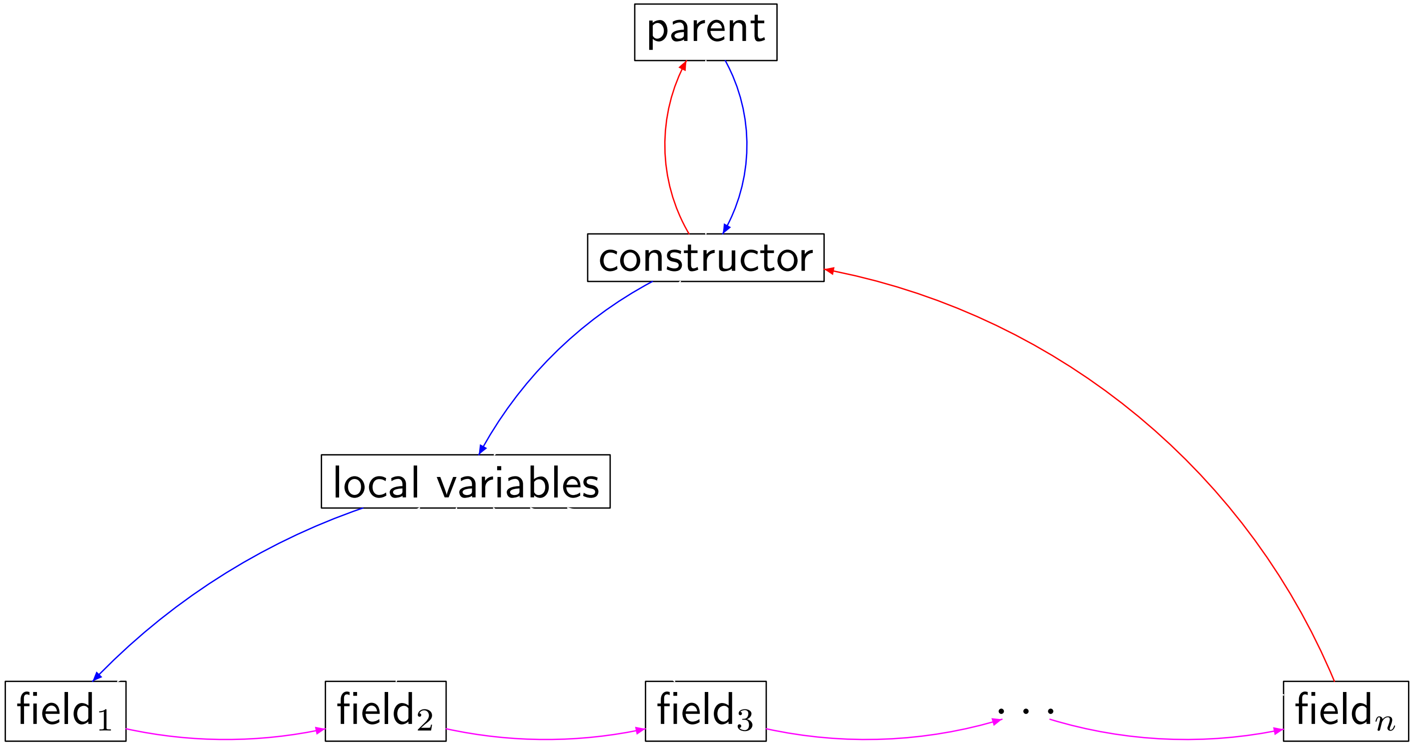
```
SEM Exprs [joinsep : String || joinval : String]
  | Cons tl.joinsep      =  lhs.joinsep
         lhs.joinval     =  if tl.isEmpty
                            then hd.lastval
                            else hd.lastval ++ lhs.joinsep ++ tl.joinval
sem_Exprs                ::  Exprs → (String → (Bool, String, String))
sem_Exprs (Cons hd tl)
          lhs_joinsep    =  let (tl_isEmpty
                                ,tl_lastval
                                ,tl_joinval
                                ) = sem_Exprs tl lhs_joinsep
                                hd_lastval = sem_Expr hd
                            in (False
                               ,if tl_isEmpty
                                 then hd_lastval
                                 else hd_lastval ++ lhs_joinsep ++ tl_joinval
                               )
```

Implementation of `UU_AG`

- Translates `UU_AG` source files into a Haskell module.
- Normal Haskell code can occur in `UU_AG` source files as well as in other modules.
- `UU_AG` data types are translated into Haskell data types.
- All attribute definitions for one data type are translated into one catamorphism on this data type, computing a function that maps the inherited attributes to the synthesised attributes of that particular data type.
- The catamorphism generated for the root symbol is the entry point to the computation.
- `UU_AG` copies the right-hand sides of rules almost literally and without interpretation.
  - \+ all Haskell constructs are available, system is lightweight
  - − no type check on `UU_AG` level, the generation process must be understood by the programmer

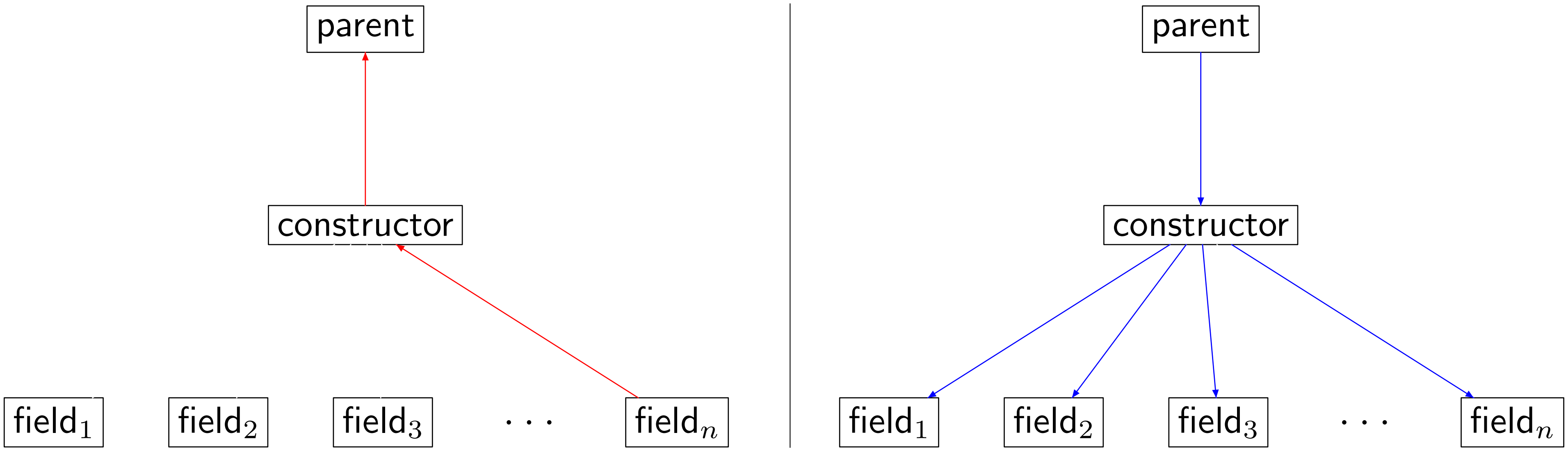
A closer look at copy rules

- There is just one (very general) copy rule.
- Attributes are identified by name.
- If an explicit rule for a specific attribute is missing, it is copied from the "nearest" node (in the picture) that provides that attribute.



Upward-copy, Downward-copy

- The copy rules for the distribution of inherited and the collection of synthesised attributes are special cases of the general copy rule.



parent

constructor

field$_1$ field$_2$ field$_3$ $\cdots$ field$_n$

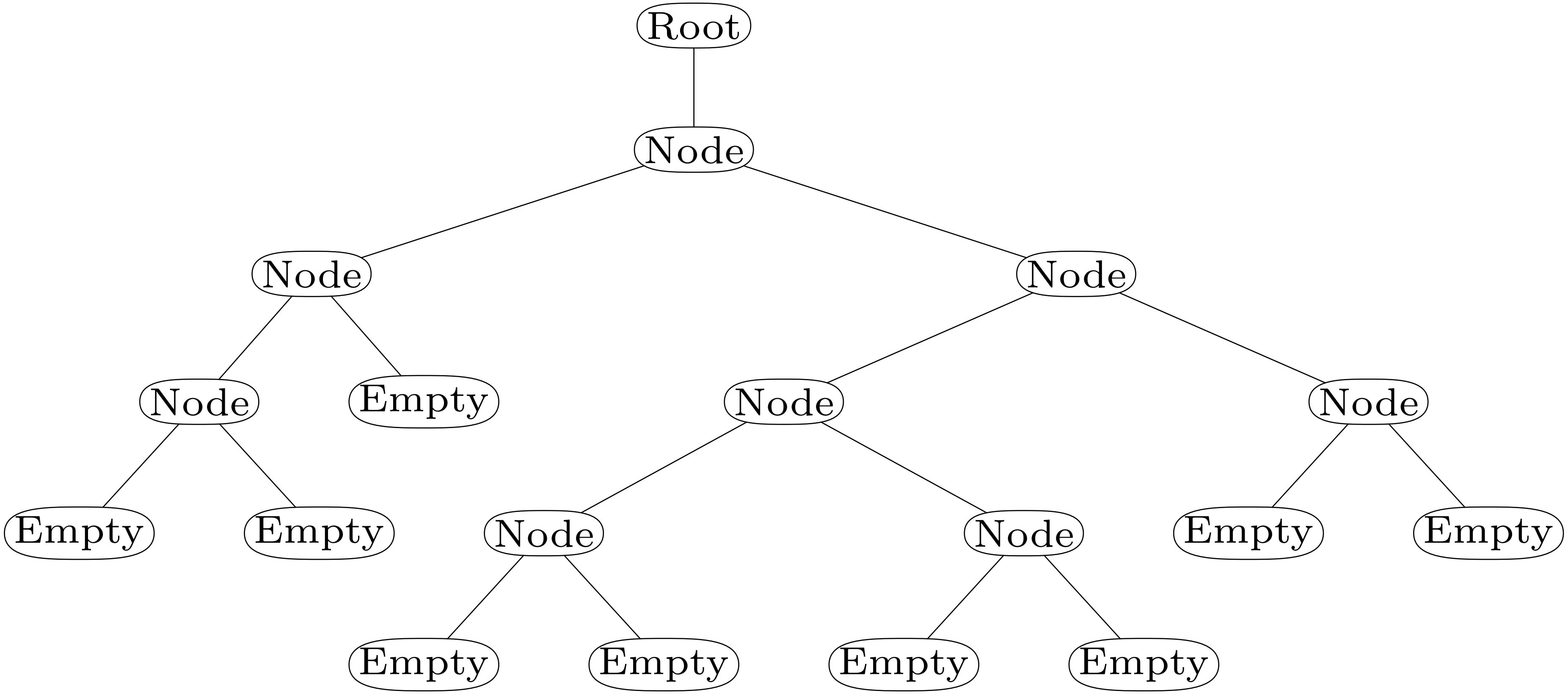
parent

constructor

field$_1$ field$_2$ field$_3$ $\cdots$ field$_n$

Tree traversals made easy I: Preliminaries

DATA *Root* | *Root Tree*
DATA *Tree* | Empty
| Node *left* : *Tree* *right* : *Tree*

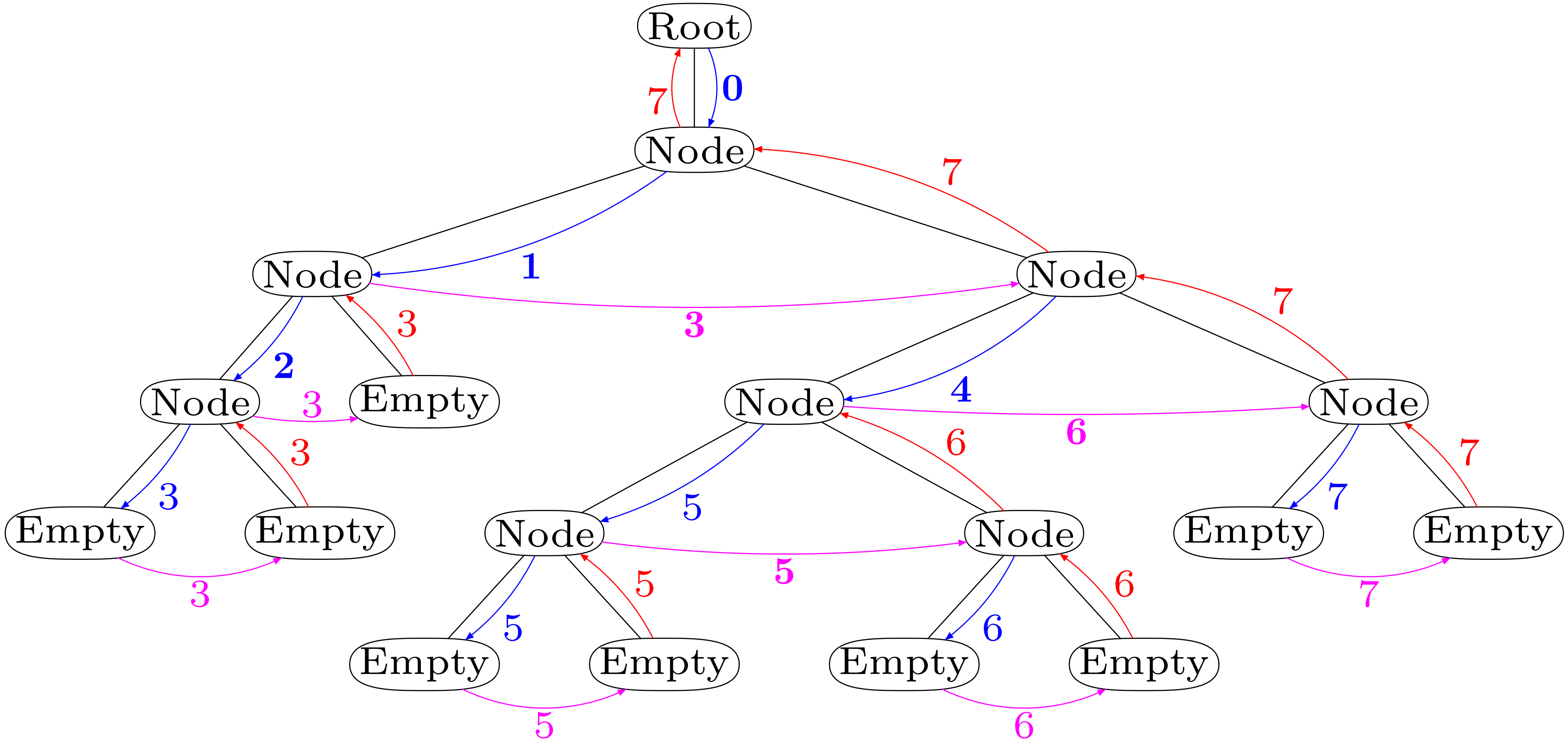


Tree traversals made easy II: DFL

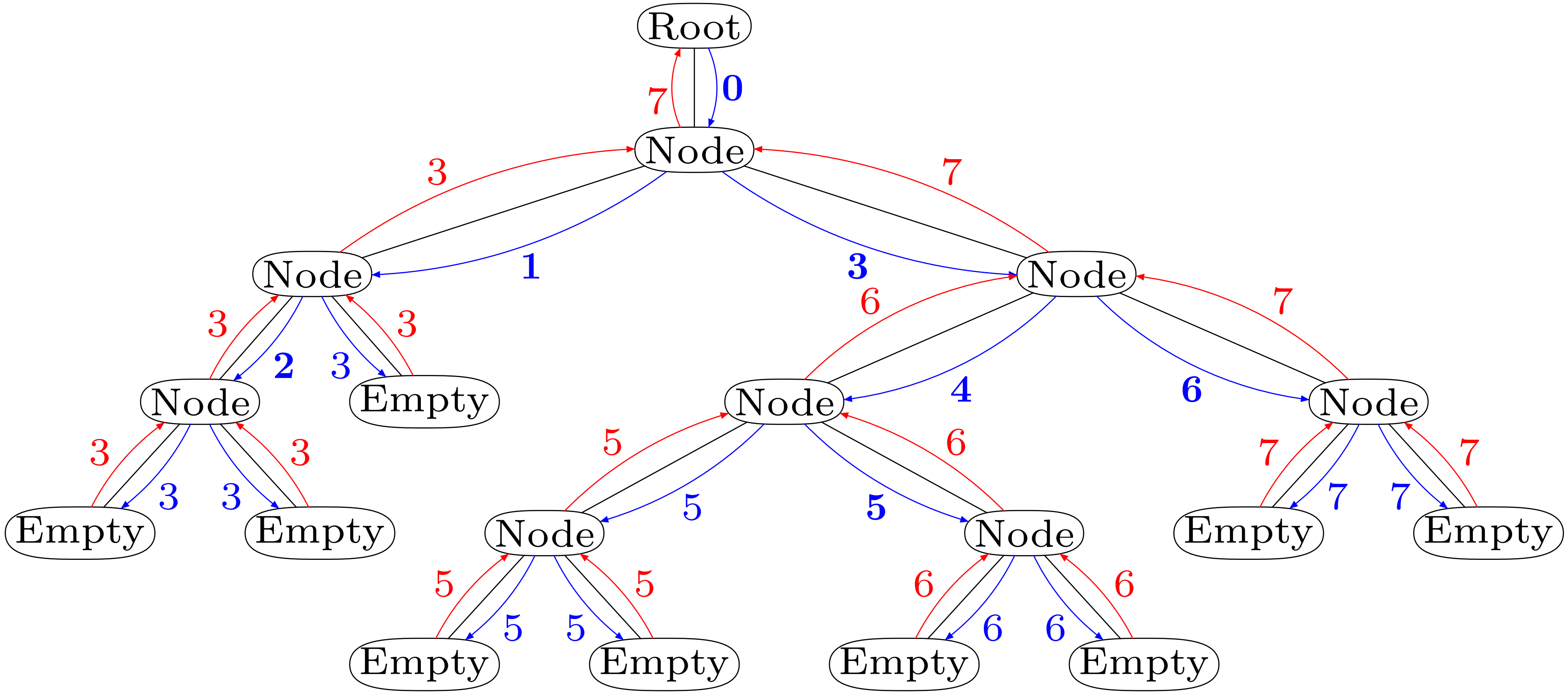
- The nodes should be uniquely labelled (in depth-first order).
- Useful for unique counters, building and changing environments corresponding to the order of the statements in the input code.

$$
\begin{array}{l}
\textbf{SEM}\ \mathit{Root} \\
\quad \mid \mathit{Root}\ \mathit{tree}.\mathit{label} \quad = \quad 0 \\
\textbf{SEM}\ \mathit{Tree}\ [\mid \mathit{label} : \mathit{Int} \mid] \\
\quad \mid \mathrm{Node}\ \mathit{left}.\mathit{label} \quad = \quad \mathbf{lhs}.\mathit{label} + 1
\end{array}
$$

Tree traversals made easy II: DFL example



Tree traversals made easy II: DFL example — continued



Properties of Haskell III: Lazy evaluation

- Function applications are reduced in "applicative order": First the function, then (and **only if needed**) the arguments.
- Lazy boolean "or" function: $\mathit{True} \vee \mathit{error}$ `"unreachable"`
- Lazy evaluation allows dealing with infinite data structures, as long as only a finite part is used in the end.

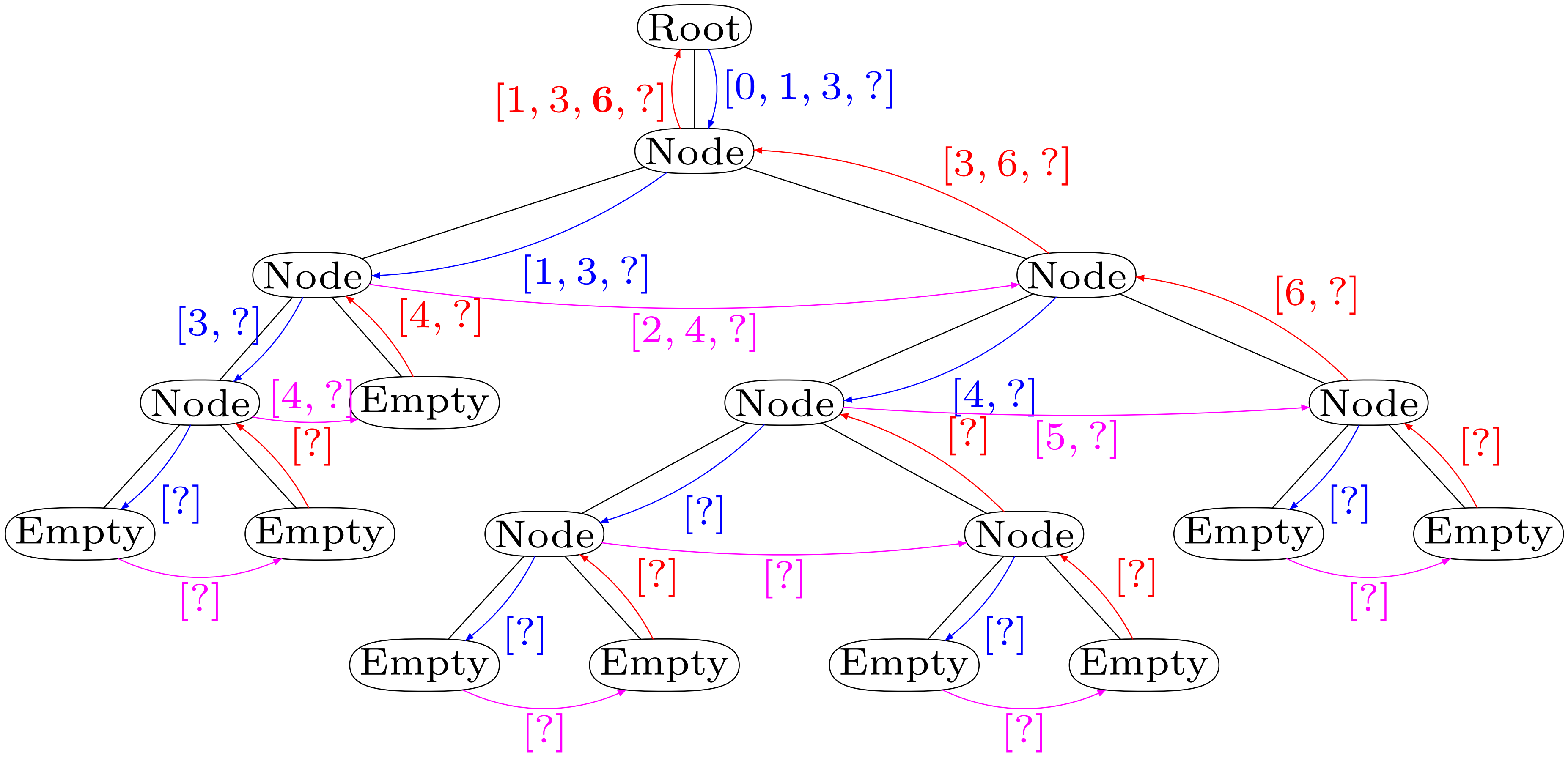
$$\begin{array}{lcl}
\mathit{primes} & :: & [\mathit{Int}] \\
\mathit{primes} & = & \mathit{sieve}\ [2\ ..] \\
\mathit{sieve} & :: & [\mathit{Int}] \rightarrow [\mathit{Int}] \\
\mathit{sieve}\ (x : \mathit{xs}) & = & x : \mathit{sieve}\ [y \mid y \leftarrow \mathit{xs}, y\ \text{`}\mathit{mod}\text{`}\ x \not\equiv 0] \\
\mathit{take}\ 100\ \mathit{primes} & &
\end{array}$$

Tree traversals made easy III: BFL

- A breadth-first traversal is not immediately covered by the copy rules.
- Nevertheless, it can be realised with only slightly more work (but making essential use of lazy evaluation!).
- Combinations of BF and DF traversal are often useful for scoping issues.
- Basic Idea: Provide a list with initial counter values for each level, return a list with final counter values for each level.

$$
\begin{array}{lcl}
\mathbf{SEM}\ \mathit{Root} & & \\
\quad \mid \mathit{Root}\ \mathit{tree}.\mathit{blabels} & = & 0 : \mathit{tree}.\mathit{blabels} \\
\mathbf{SEM}\ \mathit{Tree}\ [\mid \mathit{blabels} : [\mathit{Int}] \mid] & & \\
\quad \mid \mathrm{Node}\ \mathbf{loc}.\mathit{blabel} & = & \mathit{head}\ \mathbf{lhs}.\mathit{blabels} \\
\qquad\qquad \mathit{left}.\mathit{blabels} & = & \mathit{tail}\ \mathbf{lhs}.\mathit{blabels} \\
\qquad\qquad \mathbf{lhs}.\mathit{blabels} & = & (\mathbf{loc}.\mathit{blabel} + 1) : \mathit{right\_blabels}
\end{array}
$$

Tree traversals made easy III: BFL example



Extending the string example with variables

- Allow assignments to variables.
- Allow usage of variables.
- Variables should be visible gloabally.

$$\begin{array}{lll}
\mathbf{DATA}\ \mathit{Expr} & \mid & \mathit{Assign}\ \mathit{var} : \mathit{String}\ \mathit{Expr} \\
\mathbf{DATA}\ \mathit{Term} & \mid & \mathit{Var}\ \mathit{var} : \mathit{String}
\end{array}$$

$$\mathbf{ATTR}\ \mathit{Exprs}\ \mathit{Expr}\ \mathit{Term}\ [\,\mathit{vardist} : \mathit{Environment} \mid \mathit{varcollect} : \mathit{Environment} \mid]$$

- We store mappings of variables to string literals in an environment.
- Environments are given here as an abstract data type.

$$\begin{array}{lll}
\mathit{empty} & :: & \mathit{Environment} \\
\mathit{isDefined} & :: & \mathit{String} \rightarrow \mathit{Environment} \rightarrow \mathit{Bool} \\
\mathit{lookup} & :: & \mathit{String} \rightarrow \mathit{Environment} \rightarrow \mathit{Maybe}\ \mathit{String} \\
\mathit{add} & :: & (\mathit{String}, \mathit{String}) \rightarrow \mathit{Environment} \rightarrow \mathit{Environment} \\
\mathit{merge} & :: & \mathit{Environment} \rightarrow \mathit{Environment} \rightarrow \mathit{Environment}
\end{array}$$

Extending the string example with variables — continued

```
SEM Root
  | Root exprs.varcollect    =  empty
         exprs.vardist       =  exprs.varcollect
SEM Expr
  | Assign expr.varcollect   =  if isDefined var lhs.varcollect
                                then error "non-unique␣variable␣name"
                                else add (var, expr.lastval) lhs.varcollect
SEM Term
  | Var lhs.lastval          =  case lookup var lhs.vardist of
                                  Nothing → error "unknown␣variable"
                                  Just x → x
```

Extending the string example with groups

- Allow a list of expressions to be grouped.
- Outer variables can be used in a group, but inner variables are local.
- An inner variable can “shadow” an outer variable of the same name.

$$\begin{array}{lcl}
\mathbf{DATA}\ \mathit{Expr} & | & \mathit{Group}\ \mathit{Exprs} \\
\mathbf{SEM}\ \mathit{Expr} & & \\
\quad |\ \mathit{Group}\ \mathit{exprs}.\mathit{varcollect} & = & \mathit{empty} \\
\qquad\qquad\quad \mathbf{lhs}.\mathit{varcollect} & = & \mathbf{lhs}.\mathit{varcollect} \\
\qquad\qquad\quad \mathit{exprs}.\mathit{vardist} & = & \mathit{merge}\ \mathbf{lhs}.\mathit{varcollect}\ \mathit{exprs}.\mathit{varcollect}
\end{array}$$

Aspects can be separated

The `UU_AG` system allows to freely mix two styles of programming:

- Attribute (i.e. aspect) oriented: Define the semantics of an attribute in one place.
- Data oriented: Define the attributes of a data type in one place.

The first one is usually difficult to realise in ordinary programming languages.

Work in progress

- Static analysis: circularity, dependencies, strictification
- Language independency
- Higher-order attributes
- Type checking

Acknowledgements

- The UU_AG system has originally been designed by S. Doaitse Swierstra and Pablo Azero.
- The current implementation has been developed by Arthur Baars and Andres Löh. Further development is coordinated by Arthur Baars. This version will soon be available via `http://www.cs.uu.nl`.