Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Compiler Construction

WWW: `http://www.cs.uu.nl/wiki/Cco`

Edition 2011/2012

Agenda

Attribute grammars

- Catamorphisms
- Syntax-directed computation

5. Attribute grammars

5.1 Catamorphisms

Running example: analysing the stock market

Given: the changes in the exchange rate of some share on a day-to-day basis for some period in time:

```
+2, -3, +2, +1, +2, +4, -2, 0, +3, ...
```

Problem: calculate the maximum profit per share one could have made by buying the share at some point during this period and selling it at a later point.

☞ In a more general form, this problem is known as calculating a *maximum segment sum*.

Stategy

Underlying every single solution to the problem is the simple but effective investment strategy **"buy low, sell high"**.

Hence, we need to figure out when the share's index was low and when it was high.

But, for each day, we are only given a rate relative to the rate of the previous day.

Hence, we first have to accumulate the given differences.

Representing the index

In Haskell, we represent the index in terms of a custom list type *Index*:

```
infixl 5 `Next`
data Index α = Start | Next (Index α) α
```

Changes with respect to the previous day are simply represented by integer values:

```
type Delta = Int
```

For example:

```
fortis :: Index Delta
fortis = Start `Next` (−82)  `Next` 99
               `Next` (−162) `Next` 110
```

☞ $\mathit{Index}\ \mathit{Delta} \cong [\mathit{Int}]$.

Accumulating the differences

Fix the rate at the start of the period at zero and compute the rate for each day:

```
type Rate = Int
```

```
  -- computes the most recent rate together with all previous rates
historyR :: Index Delta                 → (Rate, Index Rate)
historyR   Start                        = (0, Start)
historyR   (Next prevD todaysD) =
  let (yesterdaysR, prevR) = historyR prevD
      todaysR              = yesterdaysR + todaysD
  in  (todaysR, prevR `Next` todaysR)
```

```
  -- computes all a historic development of the rate
rates :: (Rate, Index Rate) → Index Rate
rates    (_, allR)          = allR
```

Running minimum values

Next, we compute, for each day, which was the lowest rate observed until (and including) that day:

$\textbf{type}\ \textit{Lowest} = \textit{Int}$

```
-- computes the historic minimum rate together with all previous lows
historyL :: Index Rate                 → (Lowest, Index Lowest)
historyL    Start                      = (0, Start)
historyL    (Next prevR todaysR) =
  let (yesterdaysL, prevL) = historyL prevR
      todaysL              = yesterdaysL `min` todaysR
  in  (todaysL, prevL `Next` todaysL)
```

```
-- computes a historic development of the minimum rate
lowests :: (Lowest, Index Lowest) → Index Lowest
lowests    (_, allL)              = allL
```


Selling profits

Now we can calculate, for each day, the highest profit that could have been made *would we have sold at that day*.

To do so, we combine the historic development of the exchange rate with the development of the minimum rate:

$\textbf{type}\ \textit{Profit} = \textit{Int}$

```
-- tuples historic developments of the rate and the minimum rate
zipRL :: Index Rate → Index Lowest → Index (Rate, Lowest)
zipRL Start                Start                = Start
zipRL (Next prevR todaysR) (Next prevL todaysL) =
                    zipRL prevR prevL `Next` (todaysR, todaysL)
```

```
-- computes the historic development of the selling profit
profits :: Index (Rate, Lowest)               → Index Profit
profits    Start                              = Start
profits    (Next prevRL (todaysR, todaysL)) =
                    profits prevRL `Next` (todaysR − todaysL)
```


The maximum profit

We can now compute the maximum profit:

$\textbf{type}\ \textit{Highest} = \textit{Int}$

```
-- computes the highest profit that could have been made
highest :: Index Profit          → Highest
highest    Start                 = 0
highest    (Next prevP todaysP) =
  let yesterdaysH = highest prevP
      todaysH     = yesterdaysH `max` todaysP
  in  todaysH
```


Combining all functions

Finally, we combine all functions into a single function that computes the maximum profit from an index of changes in rate with respect to the previous day:

```
-- computes the maximum profit
maxProfit :: Index Delta → Highest
maxProfit    allD        =
  let allR  = rates (historyR allD)
      allL  = lowests (historyL allR)
      allRL = zipRL allR allL
      allP  = profits allRL
  in  highest allP
```

☞ *maxProfit* performs 6 index traversals!

Towards a more efficient solution

It is not so hard to produce a more efficient solution for maximum segment sum.

We shall now systematically construct such a solution.

The key idea is to seperate the traversal code from the actual computations and to define these computations algebraically, allowing for a straightforward means of combining traversals.

Catamorphisms

A catamorphism is a function that consumes objects X of some algebraic data type τ and produces objects of some type ρ by

- *destructing* X according to the structure of τ,
- *calling itself recursively* on any components of X that are themselves also of type τ, and
- combining the recursively obtained results of type ρ with any remaining components of X to *construct* the final result of type ρ.

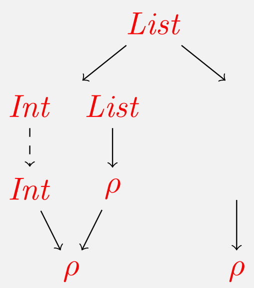
A catamorphism that consumes objects of type τ is called a τ-catamorphism.

☞ For a given τ, the destruction and recursion steps can be defined once and for all, while the construction step can be specified algebraically.

List-catamorphisms: structure

Consider a custom algebraic data type for lists of integers:

$$\textbf{data}\ \mathit{List} = \mathit{Cons}\ \mathit{Int}\ \mathit{List} \mid \mathit{Nil}$$



A List-catamorphism proceeds as follows:

A Cons-object is destructed into an Int-component and a List-component. From the List-component we obtain a recursive result of type ρ. The Int-component and the recursive result are *somehow* combined to form a final result of type ρ.

A Nil-object does not have any components. So, we have to *somehow* directly procuce a final result of type ρ.

☞ We only have to specify *how* the final nonrecursive components and recursive results are to be combined to form the final result.

List-algebras

To specify the combination step of a List-catamorphism, we employ a so-called List-algebra.

A List-algebra provides, for some choice of a type ρ, a semantics to the following signature:

$$\begin{aligned} &\mathit{cons} :: \mathit{Int} \to \rho \to \rho \\ &\mathit{nil} \quad :: \rho \end{aligned}$$

The type ρ is called the carrier of the algebra.

In Haskell, we represent List-algebras by values of the type $\mathit{Algebra}_{\mathit{List}}$:

$$\textbf{type}\ \mathit{Algebra}_{\mathit{List}}\ \rho = (\ \mathit{Int} \to \rho \to \rho\ ,\ \rho\ )$$

☞ Note how the type $\mathit{Algebra}_{\mathit{List}}$ can be systematically derived from the structure of the data type List.

List-catamorphisms: implementation

Given a List-algebra with some carrier ρ, we can produce List-catamorphisms by means of a generic function $\mathit{cata}_{\mathit{List}}$:

$$
\begin{aligned}
&\mathit{cata}_{\mathit{List}} :: \mathit{Algebra}_{\mathit{List}}\ \rho \to (\mathit{List} \to \rho)\\
&\mathit{cata}_{\mathit{List}}\quad (\mathit{cons}, \mathit{nil}) \quad = \mathit{cata}\\
&\quad \textbf{where}\\
&\qquad \mathit{cata}\ (\mathit{Cons}\ n\ \mathit{ns}) = \mathit{cons}\ n\ (\mathit{cata}\ \mathit{ns})\\
&\qquad \mathit{cata}\ \mathit{Nil} = \mathit{nil}
\end{aligned}
$$

☞ Note that $\mathit{List} \cong [\mathit{Int}]$ and that $\mathit{cata}_{\mathit{List}}$ is essentially an uncurried variation on the $\mathit{Prelude}$-function $\mathit{foldr} :: (\alpha \to \rho \to \rho) \to \rho \to [\alpha] \to [\rho]$ on built-in lists.

List-catamorphisms: length

As an example of a List-algebra, consider the following algebra for computing the length of a List:

$$
\begin{aligned}
&\mathit{alg}_{\mathit{length}} :: \mathit{Algebra}_{\mathit{List}}\ \mathit{Int}\\
&\mathit{alg}_{\mathit{length}} = (\mathit{cons}, \mathit{nil})\\
&\quad \textbf{where}\\
&\qquad \mathit{cons}\ \_\ \mathit{len} = \mathit{len} + 1\\
&\qquad \mathit{nil} \qquad\quad\ = 0
\end{aligned}
$$

The corresponding List-catamorphism is obtained by using the algebra as an argument to $\mathit{cata}_{\mathit{List}}$:

$$
\begin{aligned}
&\mathit{length} :: \mathit{List} \to \mathit{Int}\\
&\mathit{length} = \mathit{cata}_{\mathit{List}}\ \mathit{alg}_{\mathit{length}}
\end{aligned}
$$

List-catamorphism: summing

The algebra $\mathit{alg}_{\mathit{sum}}$ for summing the elements of a List is given by

$$
\begin{aligned}
&\mathit{alg}_{\mathit{sum}} :: \mathit{Algebra}_{\mathit{List}}\ \mathit{Int}\\
&\mathit{alg}_{\mathit{sum}} = ((+), 0)
\end{aligned}
$$

The actual summing function then reads

$$
\begin{aligned}
&\mathit{sum} :: \mathit{List} \to \mathit{Int}\\
&\mathit{sum} = \mathit{cata}_{\mathit{List}}\ \mathit{alg}_{\mathit{sum}}
\end{aligned}
$$

Tree-catamorphisms: structure

Of course, we can define catamorphic computations for other types as well.

Consider the type $\mathit{BinTree}$ of binary trees:

$$\textbf{data}\ \mathit{BinTree}\ \alpha = \mathit{Leaf}\ \alpha \mid \mathit{Node}\ (\mathit{BinTree}\ \alpha)\ (\mathit{BinTree}\ \alpha)$$

The structure of a $\mathit{BinTree}$-catamorphism follows from the structure of the data type:

$$
\begin{tikzcd}
 & & \mathit{BinTree}\ \alpha \arrow[dll] \arrow[dr] & & & \\
\alpha \arrow[d, dashed] & & & \mathit{BinTree}\ \alpha \arrow[d] & & \mathit{BinTree}\ \alpha \arrow[d] \\
\alpha \arrow[d] & & & \rho \arrow[dr] & & \rho \arrow[dl] \\
\rho & & & & \rho &
\end{tikzcd}
$$

☞ A $\mathit{BinTree}$-algebra needs to specify how we construct a ρ-value from an α-value and how we can construct a ρ-value from two recursively obtained ρ-values.

Tree-catamorphisms: implementation

The type $Algebra_{BinTree}$ of $BinTree$-algebras takes two type parameters: one for the values stored in the leaves of a $BinTree$, one for the carrier of the algebra.

$$\textbf{type}\ Algebra_{BinTree}\ \alpha\ \rho = (\ \alpha \to \rho\ ,\ \rho \to \rho \to \rho\ )$$

$BinTree$-catamorphisms can be obtained from calls to the function $cata_{BinTree}$:

$$\begin{array}{l}
cata_{BinTree} :: Algebra_{BinTree}\ \alpha\ \rho \to (BinTree\ \alpha \to \rho) \\
cata_{BinTree}\ \ (leaf, node) \qquad = cata \\
\quad \textbf{where} \\
\qquad cata\ (Leaf\ x) \quad = leaf\ x \\
\qquad cata\ (Node\ l\ r) = node\ (cata\ l)\ (cata\ r)
\end{array}$$

Tree-catamorphisms: size

A catamorphism for retrieving the number of leaves in a $BinTree$:

$$\begin{array}{l}
alg_{size}\ :: Algebra_{BinTree}\ \alpha\ Int \\
alg_{size}\ = (const\ 1, (+))
\end{array}$$

$$\begin{array}{l}
size :: BinTree\ \alpha \to Int \\
size = cata_{BinTree}\ alg_{size}
\end{array}$$

Tree-catamorphisms: product

A catamorphism for computing the product of all Int-values stored in a $BinTree$:

$$\begin{array}{l}
alg_{product}\ :: Algebra_{BinTree}\ Int\ Int \\
alg_{product}\ = (id, (*))
\end{array}$$

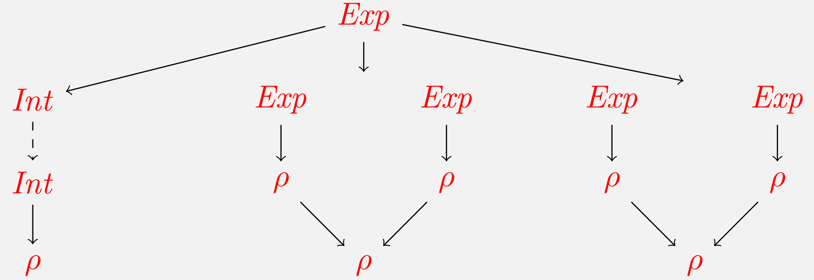
$$\begin{array}{l}
product :: BinTree\ \alpha \to Int \\
product = cata_{BinTree}\ alg_{product}
\end{array}$$

Expression-catamorphisms: structure

Now consider a data type Exp of simple arithmetic expressions:

$$\textbf{data}\ Exp = Const\ Int \mid Add\ Exp\ Exp \mid Mul\ Exp\ Exp$$

The structure of an Exp-catamorphism follows the structure of the data type:



Expression-catamorphisms: implementation

The type of Exp-algebras:

$$\textbf{type}\ \mathit{Algebra}_{\mathit{Exp}}\ \rho = (\ \mathit{Int} \to \rho\ ,\ \rho \to \rho \to \rho\ ,\ \rho \to \rho \to \rho\ )$$

The function $\mathit{cata}_{\mathit{Exp}}$ for constructing Exp-catamorphisms from Exp-algebras:

$$\begin{array}{l}
\mathit{cata}_{\mathit{Exp}} :: \mathit{Algebra}_{\mathit{Exp}}\ \rho \quad\to (\mathit{Exp} \to \rho) \\
\mathit{cata}_{\mathit{Exp}}\quad (\mathit{const}, \mathit{add}, \mathit{mul}) = \mathit{cata} \\
\quad \textbf{where} \\
\qquad \mathit{cata}\ (\mathit{Const}\ n) \quad = \mathit{const}\ n \\
\qquad \mathit{cata}\ (\mathit{Add}\ e_1\ e_2) = \mathit{add}\ (\mathit{cata}\ e_1)\ (\mathit{cata}\ e_2) \\
\qquad \mathit{cata}\ (\mathit{Mul}\ e_1\ e_2) = \mathit{mul}\ (\mathit{cata}\ e_1)\ (\mathit{cata}\ e_2)
\end{array}$$

Expression-catamorphisms: evaluation

A catamorphism for evaluating expressions:

$$\begin{array}{l}
\mathit{alg}_{\mathit{eval}} :: \mathit{Algebra}_{\mathit{Exp}}\ \mathit{Int} \\
\mathit{alg}_{\mathit{eval}} = (\mathit{id}, (+), (*))
\end{array}$$

$$\begin{array}{l}
\mathit{eval} :: \mathit{Exp} \to \mathit{Int} \\
\mathit{eval} = \mathit{cata}_{\mathit{Exp}}\ \mathit{alg}_{\mathit{eval}}
\end{array}$$

Index-catamorphisms: structure

Returning to our running example, recall the definition of the custom list type Index:

$$\textbf{data}\ \mathit{Index}\ \alpha = \mathit{Start} \mid \mathit{Next}\ (\mathit{Index}\ \alpha)\ \alpha$$

The structure of an Index-catamorphism follows the structure of the data type:

$$\begin{tikzcd}
& \mathit{Index}\ \alpha \arrow[dl] \arrow[dr] & & & \\
{} \arrow[d] & & \mathit{Index}\ \alpha \arrow[d] & & \alpha \arrow[d, dashed] \\
\rho & & \rho \arrow[dr] & & \alpha \arrow[dl] \\
& & & \rho &
\end{tikzcd}$$

Index-catamorphims: implementation

The type $\mathit{Algebra}_{\mathit{Index}}$ of Index-algebras:

$$\textbf{type}\ \mathit{Algebra}_{\mathit{Index}}\ \alpha\ \rho = (\ \rho\ ,\ \rho \to \alpha \to \rho\ )$$

The function $\mathit{cata}_{\mathit{Index}}$ for constructing Index-catamorphisms from Index-algebras:

$$\begin{array}{l}
\mathit{cata}_{\mathit{Index}} :: \mathit{Algebra}_{\mathit{Index}}\ \alpha\ \rho \to (\mathit{Index}\ \alpha \to \rho) \\
\mathit{cata}_{\mathit{Index}} = (\mathit{start}, \mathit{next}) \qquad = \mathit{cata} \\
\quad \textbf{where} \\
\qquad \mathit{cata}\ \mathit{Start} \qquad\qquad\quad = \mathit{start} \\
\qquad \mathit{cata}\ (\mathit{Next}\ \mathit{prev}\ \mathit{today}) = \mathit{next}\ (\mathit{cata}\ \mathit{prev})\ \mathit{today}
\end{array}$$

Accumulating differences, catamorphically

Now we can define our function $historyR$ as an $Index$-catamorphism:

```
-- computes the most recent rate together with all previous rates
historyR :: Index Delta → (Rate, Index Rate)
historyR = cata_Index (start, next)
  where
    start                                = (0, Start)
    next (yesterdaysR, prevR) todaysD =
      let todaysR = yesterdaysR + todaysD
      in  (todaysR, prevR `Next` todaysR)
```


Running minimum values, catamorphically

In a similar fashion, $historyL$ can be defined as a catamorphism:

```
-- computes the historic minimum rate together with all previous lows
historyL :: Index Rate → (Lowest, Index Lowest)
historyL = cata_Index (start, next)
  where
    start                                = (0, Start)
    next (yesterdaysL, prevL) todaysR =
      let todaysL = yesterdaysL `min` todaysR
      in  (todaysL, prevL `Next` todaysL)
```


Combining the traversals

Now we have two $Index$-catamorphisms, $historyR$ and $historyL$:

```
historyR :: Index Delta → (Rate   , Index Rate  )
historyL :: Index Rate  → (Lowest, Index Lowest)
```

These can easily be combined into a single catamorphism $historyRL$ by simply combining the underlying $Index$-algebras:

```
historyRL   ::  Index Delta →
                (Rate, Index Rate, Lowest, Index Lowest)
historyRL   =   cata_Index (start, next)
  where
    start = (0, Start, 0, Start)
    next (yesterdaysR, prevR, yesterdaysL, prevL) todaysD =
      let todaysR = yesterdaysR + todaysD
          todaysL = yesterdaysL `min` todaysR
      in      (   todaysR, prevR `Next` todaysR
              ,   todaysL, prevL `Next` todaysL
              )
```


Cleaning the traversal

By now, the historic developments of the rate and minimum produced by $historyRL$ are redundant and we can simply define:

```
computeRL :: Index Delta → (Rate, Lowest)
computeRL = cata_Index (start, next)
  where
    start                                      = (0, 0)
    next (yesterdaysR, yesterdaysL) todaysD =
      let todaysR = yesterdaysR + todaysD
          todaysL = yesterdaysL `min` todaysR
      in  (todaysR, todaysL)
```

Profits, catamorphically

Continuing in this fashion, we can extend the result of the catamorphism with a component that holds the profit associated with selling on a certain day and a component that holds the maximum profit:

$$
\begin{array}{l}
\mathit{computeRLPH} :: \mathit{Index}\ \mathit{Delta} \to (\mathit{Rate}, \mathit{Lowest}, \mathit{Profit}, \mathit{Highest}) \\
\mathit{computeRLPH} = \mathit{cata}_{\mathit{Index}}\ (\mathit{start}, \mathit{next}) \\
\quad \mathbf{where} \\
\qquad \mathit{start} = (0, 0, 0, 0) \\
\qquad \mathit{next} \\
\qquad\quad (\mathit{yesterdaysR}, \mathit{yesterdaysL}, \mathit{yesterdaysP}, \mathit{yesterdaysH})\ \mathit{todaysD} = \\
\qquad\qquad \mathbf{let}\ \ \mathit{todaysR} = \mathit{yesterdaysR} + \mathit{todaysD} \\
\qquad\qquad\qquad \mathit{todaysL} = \mathit{yesterdaysL}\ \text{`}\mathit{min}\text{`}\ \mathit{todaysR} \\
\qquad\qquad\qquad \mathit{todaysP} = \mathit{todaysR} - \mathit{todaysL} \\
\qquad\qquad\qquad \mathit{todaysH} = \mathit{yesterdaysH}\ \text{`}\mathit{max}\text{`}\ \mathit{todaysP} \\
\qquad\qquad \mathbf{in}\ \ (\mathit{todaysR}, \mathit{todaysL}, \mathit{todaysP}, \mathit{todaysH})
\end{array}
$$

Simplifying

We can further simplify this by dropping the *Profit*-component and calculating the *Highest*-component directly:

$$
\begin{array}{l}
\mathit{computeRLH} :: \mathit{Index}\ \mathit{Delta} \to (\mathit{Rate}, \mathit{Lowest}, \mathit{Highest}) \\
\mathit{computeRLH} = \mathit{cata}_{\mathit{Index}}\ (\mathit{start}, \mathit{next}) \\
\quad \mathbf{where} \\
\qquad \mathit{start} = (0, 0, 0) \\
\qquad \mathit{next}\ (\mathit{yesterdaysR}, \mathit{yesterdaysL}, \mathit{yesterdaysH})\ \mathit{todaysD} = \\
\qquad\quad \mathbf{let}\ \mathit{todaysR} = \mathit{yesterdaysR} + \mathit{todaysD} \\
\qquad\qquad \mathit{todaysL} = \mathit{yesterdaysL}\ \text{`}\mathit{min}\text{`}\ \mathit{todaysR} \\
\qquad\qquad \mathit{todaysH} = \mathit{yesterdaysH}\ \text{`}\mathit{max}\text{`}\ (\mathit{todaysR} - \mathit{todaysL}) \\
\qquad\quad \mathbf{in}\ \ (\mathit{todaysR}, \mathit{todaysL}, \mathit{todaysH})
\end{array}
$$

The final solution

The final solution now simply reads:

$$
\begin{array}{l}
\mathit{maxProfit} :: \mathit{Index}\ \mathit{Delta} \to \mathit{Highest} \\
\mathit{maxProfit}\ \ \mathit{allD} = \\
\quad \mathbf{let}\ (\_, \_, \mathit{highest}) = \mathit{computeRLH}\ \mathit{allD} \\
\quad \mathbf{in}\ \mathit{highest}
\end{array}
$$

☞ This version of *maxProfit* computes the maximum profit by performing just a single iteration over the argument index.

Maximum segment sum

$$
\begin{array}{l}
\mathbf{type}\ \mathit{Delta} \qquad\quad = \mathit{Int} \\
\mathbf{data}\ \mathit{Index} \qquad\quad = \mathit{Start} \mid \mathit{Next}\ \mathit{Index}\ \mathit{Delta} \\
\mathbf{type}\ \mathit{Algebra}_{\mathit{Index}}\ \rho = (\rho, \rho \to \mathit{Delta} \to \rho) \\
\mathit{cata}_{\mathit{Index}} :: \mathit{Algebra}_{\mathit{Index}}\ \rho \to \mathit{Index} \to \rho \\
\mathit{cata}_{\mathit{Index}}\ (\mathit{start}, \mathit{next}) = \mathit{cata} \\
\quad \mathbf{where} \\
\qquad \mathit{cata}\ \mathit{Start} \qquad\qquad\quad = \mathit{start} \\
\qquad \mathit{cata}\ (\mathit{Next}\ \mathit{prev}\ \mathit{today}) = \mathit{next}\ (\mathit{cata}\ \mathit{prev})\ \mathit{today} \\
\mathit{mss} :: \mathit{Index}\ \mathit{Delta} \to \mathit{Int} \\
\mathit{mss}\ \mathit{allD} = \mathbf{let}\ (\_, \_, \mathit{highest}) = \mathit{cata}_{\mathit{Index}}\ (\mathit{start}, \mathit{next})\ \mathit{allD} \\
\qquad\qquad\quad \mathbf{in}\ \ \mathit{highest} \\
\quad \mathbf{where} \\
\qquad \mathit{start} = (0, 0, 0) \\
\qquad \mathit{next}\ (\mathit{yesterdaysR}, \mathit{yesterdaysL}, \mathit{yesterdaysH})\ \mathit{today} = \\
\qquad\quad \mathbf{let}\ \mathit{todaysR} = \mathit{yesterdaysR} + \mathit{today} \\
\qquad\qquad \mathit{todaysL} = \mathit{yesterdaysL}\ \text{`}\mathit{min}\text{`}\ \mathit{todaysR} \\
\qquad\qquad \mathit{todaysH} = \mathit{yesterdaysH}\ \text{`}\mathit{max}\text{`}\ (\mathit{todaysR} - \mathit{todaysL}) \\
\qquad\quad \mathbf{in}\ \ (\mathit{todaysR}, \mathit{todaysL}, \mathit{todaysH})
\end{array}
$$

5.2 Syntax-directed computation

The UU Attribute Grammar system

The UU Attribute Grammar system essentially facilitates the definition and composition of algebras.

It it implemented as a preprocessor to Haskell:

Attribute grammars

An attribute grammar is a means to associate attributes (semantics) with the productions of a grammar (syntax).

Defining an attribute grammar proceeds in three steps:

1. Define a grammar.
2. Declare attributes.
3. Define attribute equations.

☞ As an example, we consider an attribute grammar for maximum segment sum.

Synthesised and inherited attributes

We distinguish between two sorts of attributes:

Synthesised attributes: these “travel” upward through a syntax tree, i.e., from child to parent.

Inherited attributes: these “travel” downward through a syntax tree, i.e., from parent to child.

Declaring grammars

A grammar consists of a set of Haskell-like data-type definitions.

$\{\textbf{type}\ \mathit{Delta} = \mathit{Int}\}$

$\textbf{data}\ \mathit{Index}$
$\quad |\ \mathit{Start}$
$\quad |\ \mathit{Next}\ \mathit{prev} :: \mathit{Index}\ \mathit{today} :: \{\mathit{Delta}\}$

- Each constructor field has a label.
- Code between curly braces is just Haskell-code and will show up in the generated Haskell-file without further processing.

Compiling grammars

We invoke the AG compiler with the flags `H` and `d` for, respectively, enabling Haskell-like syntax and generating a set of algebraic data types for the grammar:

```
uuagc -Hd MSS.ag
```

This produces a file `MSS.hs` containing:

$\textbf{type}\ \mathit{Delta} = \mathit{Int}$

$\textbf{data}\ \mathit{Index} = \mathit{Start}\ |\ \mathit{Next}\ \mathit{Index}\ \mathit{Delta}$

Declaring synthesised attributes

We declare a synthesised attribute rate:

$\textbf{attr}\ \mathit{Index}$
$\quad \textbf{syn}\ \mathit{rate} :: \{\mathit{Int}\}$

This introduces the obligation to show, for each production of the grammar Index, how to synthesise an Int-value rate.

Defining synthesised attributes

Attributes are defined by associating semantic actions with grammar productions:

$\textbf{sem}\ \mathit{Index}$
$\quad |\ \mathit{Start}\ \textbf{lhs}.\mathit{rate} = 0$
$\quad |\ \mathit{Next}\ \ \textbf{lhs}.\mathit{rate} = @\mathit{prev}.\mathit{rate} + @\mathit{today}$

- $@\mathit{today}$ refers to the value stored in the field today.
- $@\mathit{prev}.\mathit{rate}$ refers to the synthesised attribute rate for the child prev.

Compiling attributes: semantic functions

We generate code for semantic actions by issueing the compiler flag `f`:

```
uuagc -Hdf MSS.ag
```

This results in:

$$\begin{aligned} &sem_{Index|Start} &&= 0 \\ &sem_{Index|Next}\ rate_{prev}\ today &&= rate_{prev} + today \end{aligned}$$

☞ $sem_{Index|Start}$ and $sem_{Index|Next}$ constitute an $Index$-algebra.

Compiling attributes: catamorphisms

To emit a catamorphism for the defined algebra, we issue the compiler flag `c`:

```
uuagc -Hdfc MSS.ag
```

This produces:

$$\begin{aligned} &sem_{Index}\ Start &&= sem_{Index|Start} \\ &sem_{Index}\ (Next\ prev\ today) &&= sem_{Index|Next} \\ & && \quad (sem_{Index}\ prev)\ today \end{aligned}$$

More synthesised attributes

$$\begin{aligned} &\textbf{attr}\ Index \\ &\quad \textbf{syn}\ lowest :: \{Int\} \\ &\textbf{sem}\ Index \\ &\quad |\ Start\ \textbf{lhs}.lowest = 0 \\ &\quad |\ Next\ \ \textbf{lhs}.lowest = @prev.lowest\ \text{`}min\text{`} \\ &\qquad\qquad\qquad (@prev.rate + @today) \end{aligned}$$

Compiling multiple attributes

If we have multiple synthesised attributes, i.e., multiple algebras, for a grammar, these show up as tuples in the generated Haskell-code:

$$\begin{aligned} &sem_{Index|Start} = (0, 0) \\ &sem_{Index|Next}\ (rate_{prev}, lowest_{prev})\ today = \\ &\quad (rate_{prev} + today\ ,\ lowest_{prev}\ \text{`}min\text{`}\ (rate_{prev} + today)) \end{aligned}$$

☞ Note we have some code duplication here.

Local attributes

When defining an algebra, we can introduce attributes that are local to a given production, i.e., these flow neither upward nor downward:

$$
\begin{aligned}
&\textbf{sem}\ \mathit{Index}\\
&\quad |\ \mathit{Next}\ \textbf{loc}.\mathit{rate} \quad = @\mathit{prev}.\mathit{rate} + @\mathit{today}\\
&\qquad\qquad \textbf{lhs}.\mathit{rate} \quad = @\textbf{loc}.\mathit{rate}\\
&\qquad\qquad \textbf{lhs}.\mathit{lowest} = @\mathit{prev}.\mathit{lowest}\ \text{`}\mathit{min}\text{`}\ @\textbf{loc}.\mathit{rate}
\end{aligned}
$$

Compiling local attributes

Local attributes are compiled into local definitions in the generated algebras:

$$
\begin{aligned}
&\mathit{sem}_{\mathit{Index}|\mathit{Start}} = (0,0)\\
&\mathit{sem}_{\mathit{Index}|\mathit{Next}}\ (\mathit{rate}_{\mathit{prev}}, \mathit{lowest}_{\mathit{prev}})\ \mathit{today} =\\
&\quad \textbf{let}\ \mathit{rate} = \mathit{rate}_{\mathit{prev}} + \mathit{today}\\
&\quad \textbf{in}\ \ (\mathit{rate}\ ,\ \mathit{lowest}_{\mathit{prev}}\ \text{`}\mathit{min}\text{`}\ \mathit{rate})
\end{aligned}
$$

Copy rule

If we need to produce a synthesised attribute and already have a local attribute of the same name, the AG compiler can automatically produce code for the synthesised attribute by simply copying the value of the local attribute into the synthesised attribute:

$$
\begin{aligned}
&\textbf{sem}\ \mathit{Index}\\
&\quad |\ \mathit{Next}\ \textbf{loc}.\mathit{rate} \quad = @\mathit{prev}.\mathit{rate} + @\mathit{today}\\
&\qquad\qquad \textbf{loc}.\mathit{lowest} = @\mathit{prev}.\mathit{lowest}\ \text{`}\mathit{min}\text{`}\ @\textbf{loc}.\mathit{rate}
\end{aligned}
$$

This is an instance of the so-called copy rule.

Generating type signatures

With the compiler flag `s` we can generate type signatures for algebras and catamorphisms:

```
uuagc -Hdfcs MSS.ag
```

$$
\begin{aligned}
&\textbf{type}\ T_{\mathit{Index}} = (\mathit{Int}, \mathit{Int})\\
&\mathit{sem}_{\mathit{Index}|\mathit{Start}} :: T_{\mathit{Index}}\\
&\mathit{sem}_{\mathit{Index}|\mathit{Start}} = (0,0)\\
&\mathit{sem}_{\mathit{Index}|\mathit{Next}} :: T_{\mathit{Index}} \to \mathit{Delta} \to T_{\mathit{Index}}\\
&\mathit{sem}_{\mathit{Index}|\mathit{Next}}\ (\mathit{rate}_{\mathit{prev}}, \mathit{lowest}_{\mathit{prev}})\ \mathit{today} =\\
&\quad \textbf{let}\ \mathit{rate} \quad = \mathit{rate}_{\mathit{prev}} + \mathit{today}\\
&\qquad\ \ \mathit{lowest} = \mathit{lowest}_{\mathit{prev}}\ \text{`}\mathit{min}\text{`}\ \mathit{rate}\\
&\quad \textbf{in}\ \ (\mathit{rate}, \mathit{lowest})\\
&\mathit{sem}_{\mathit{Index}} :: \mathit{Index} \to T_{\mathit{Index}}\\
&\mathit{sem}_{\mathit{Index}}\ \mathit{Start} \qquad\qquad\quad = \mathit{sem}_{\mathit{Index}|\mathit{Start}}\\
&\mathit{sem}_{\mathit{Index}}\ (\mathit{Next}\ \mathit{prev}\ \mathit{today}) = \mathit{sem}_{\mathit{Index}|\mathit{Next}}\\
&\qquad\qquad\qquad\qquad\qquad\quad (\mathit{sem}_{\mathit{Index}}\ \mathit{prev})\ \mathit{today}
\end{aligned}
$$

A complete attribute grammar for MSS

$$
\begin{array}{l}
\{\mathbf{type}\ \mathit{Delta} = \mathit{Int}\} \\
\mathbf{data}\ \mathit{Index} \\
\quad \mid \mathit{Start} \\
\quad \mid \mathit{Next}\ \mathit{prev} :: \mathit{Index}\ \mathit{today} :: \{\mathit{Delta}\} \\
\mathbf{attr}\ \mathit{Index} \\
\quad \mathbf{syn}\ \mathit{rate} \quad :: \{\mathit{Int}\} \\
\quad \mathbf{syn}\ \mathit{lowest} \ :: \{\mathit{Int}\} \\
\quad \mathbf{syn}\ \mathit{highest} :: \{\mathit{Int}\} \\
\mathbf{sem}\ \mathit{Index} \\
\quad \mid \mathit{Start}\ \mathbf{lhs}.(\mathit{rate}, \mathit{lowest}, \mathit{highest}) = (0, 0, 0) \\
\quad \mid \mathit{Next}\ \mathbf{loc}.\mathit{rate} \quad = @\mathit{prev}.\mathit{rate} + @\mathit{today} \\
\qquad\qquad \mathbf{loc}.\mathit{lowest} \ = @\mathit{prev}.\mathit{lowest}\ `\mathit{min}`\ @\mathbf{loc}.\mathit{rate} \\
\qquad\qquad \mathbf{loc}.\mathit{profit} \ = @\mathbf{loc}.\mathit{rate} - @\mathbf{loc}.\mathit{lowest} \\
\qquad\qquad \mathbf{lhs}.\mathit{highest} = @\mathit{prev}.\mathit{highest}\ `\mathit{max}`\ @\mathbf{loc}.\mathit{profit}
\end{array}
$$

Declaring inherited attributes

Assume we are given the actual exchange rate at the start of the period and want to compute the rate at the end of the period.

For this, we can employ an inherited attribute:

$$
\begin{array}{l}
\mathbf{attr}\ \mathit{Index} \\
\quad \mathbf{inh}\ \mathit{start} :: \{\mathit{Int}\} \\
\quad \mathbf{syn}\ \mathit{end} \ \ :: \{\mathit{Int}\}
\end{array}
$$

This introduces the obligation to show, for each production that has an Index-field, how an Int-attribute start can be passed to the field.

Defining inherited attributes

$$
\begin{array}{l}
\mathbf{sem}\ \mathit{Index} \\
\quad \mid \mathit{Start}\ \mathbf{lhs}.\mathit{end} \quad = @\mathbf{lhs}.\mathit{start} \\
\quad \mid \mathit{Next}\ \ \mathit{prev}.\mathit{start} = @\mathbf{lhs}.\mathit{start} \\
\qquad\qquad \mathbf{lhs}.\mathit{end} \quad = @\mathit{prev}.\mathit{end} + @\mathit{today}
\end{array}
$$

☞ $@\mathbf{lhs}.\mathit{start}$ accesses the inherited attribute start.

Compiling inherited attributes


```
uuagc -Hdfcs Rate.ag
```

$$
\begin{array}{l}
\mathbf{type}\ T_{\mathit{Index}} = \mathit{Int} \to \mathit{Int} \\
\mathit{sem}_{\mathit{Index}|\mathit{Start}} :: T_{\mathit{Index}} \\
\mathit{sem}_{\mathit{Index}|\mathit{Start}} = \lambda \mathit{start}_{\mathit{lhs}} \to \mathit{start}_{\mathit{lhs}} \\
\mathit{sem}_{\mathit{Index}|\mathit{Next}} :: T_{\mathit{Index}} \to \mathit{Delta} \to T_{\mathit{Index}} \\
\mathit{sem}_{\mathit{Index}|\mathit{Next}}\ \mathit{prev}\ \mathit{today} = \lambda \mathit{start}_{\mathit{lhs}} \to \\
\quad \mathbf{let}\ \mathit{end}_{\mathit{prev}} = \mathit{prev}\ \mathit{start}_{\mathit{lhs}} \\
\quad \mathbf{in}\ \ \mathit{end}_{\mathit{prev}} + \mathit{today}
\end{array}
$$

Generating wrappers


```
uuagc -Hdfcsw Rate.ag
```

$$
\begin{array}{l}
\textbf{data}\ \mathit{Inh}_{\mathit{Index}} = \mathit{Inh}_{\mathit{Index}}\ \{\mathit{start}_{\mathit{Inh}|\mathit{Index}} :: \mathit{Int}\} \\
\textbf{data}\ \mathit{Syn}_{\mathit{Index}} = \mathit{Syn}_{\mathit{Index}}\ \{\mathit{end}_{\mathit{Syn}|\mathit{Index}} :: \mathit{Int}\} \\
\mathit{wrap}_{\mathit{Index}} :: T_{\mathit{Index}} \to \mathit{Inh}_{\mathit{Index}} \to \mathit{Syn}_{\mathit{Index}} \\
\mathit{wrap}_{\mathit{Index}}\ \mathit{sem}\ (\mathit{Inh}_{\mathit{Index}}\ \mathit{start}_{\mathit{lhs}}) = \\
\quad \textbf{let}\ \mathit{end}_{\mathit{lhs}} = \mathit{sem}\ \mathit{start}_{\mathit{lhs}} \\
\quad \textbf{in}\ \ \mathit{Syn}_{\mathit{Index}}\ \mathit{end}_{\mathit{lhs}}
\end{array}
$$

An expression grammar

$$
\begin{array}{l}
\{\textbf{type}\ \mathit{Num}\_ = \mathit{Int}\} \\
\textbf{data}\ \mathit{Tm} \\
\quad |\ \mathit{Tm}\quad \mathit{pos} :: \{\mathit{SourcePos}\}\ t :: \mathit{Tm}\_ \\
\textbf{data}\ \mathit{Tm}\_ \\
\quad |\ \mathit{Num}\ \ n :: \{\mathit{Num}\_\} \\
\quad |\ \mathit{False}\_ \\
\quad |\ \mathit{True}\_ \\
\quad |\ \mathit{If}\quad t_1 :: \mathit{Tm}\ t_2 :: \mathit{Tm}\ t_3 :: \mathit{Tm} \\
\quad |\ \mathit{Add}\quad t_1 :: \mathit{Tm}\ t_2 :: \mathit{Tm} \\
\quad |\ \mathit{Mul}\quad t_1 :: \mathit{Tm}\ t_2 :: \mathit{Tm} \\
\quad |\ \mathit{Lt}\quad t_1 :: \mathit{Tm}\ t_2 :: \mathit{Tm} \\
\quad |\ \mathit{Eq}\quad t_1 :: \mathit{Tm}\ t_2 :: \mathit{Tm} \\
\quad |\ \mathit{Gt}\quad t_1 :: \mathit{Tm}\ t_2 :: \mathit{Tm}
\end{array}
$$

An evaluation attribute

$$
\begin{array}{llll}
\multicolumn{4}{l}{\{\textbf{data}\ \mathit{Val} = \mathit{VNum}\ \mathit{Num}\_ \mid \mathit{VFalse} \mid \mathit{VTrue}\}} \\
\multicolumn{4}{l}{\textbf{attr}\ \mathit{Tm}\ \mathit{Tm}\_} \\
\multicolumn{4}{l}{\quad \textbf{syn}\ \mathit{val} :: \{\mathit{Val}\}} \\
\multicolumn{4}{l}{\textbf{sem}\ \mathit{Tm}\_} \\
\quad |\ \mathit{Num} & \textbf{lhs}.\mathit{val} & = & \mathit{VNum}\ @n \\
\quad |\ \mathit{False}\_ & \textbf{lhs}.\mathit{val} & = & \mathit{VFalse} \\
\quad |\ \mathit{True}\_ & \textbf{lhs}.\mathit{val} & = & \mathit{VTrue} \\
\quad |\ \mathit{If} & \textbf{lhs}.\mathit{val} & = & \textbf{case}\ @t_1.\mathit{val}\ \textbf{of}\ \mathit{VTrue} \to @t_2.\mathit{val} \\
 & & & \qquad\qquad\qquad\qquad \mathit{VFalse} \to @t_3.\mathit{val} \\
\quad |\ \mathit{Add} & \textbf{lhs}.\mathit{val} & = & \textbf{let}\ (\mathit{VNum}\ n_1, \mathit{VNum}\ n_2) = (@t_1.\mathit{val}, @t_2.\mathit{val}) \\
 & & & \textbf{in}\ \ \mathit{VNum}\ (n_1 + n_2) \\
\quad |\ \mathit{Mul} & \textbf{lhs}.\mathit{val} & = & \textbf{let}\ (\mathit{VNum}\ n_1, \mathit{VNum}\ n_2) = (@t_1.\mathit{val}, @t_2.\mathit{val}) \\
 & & & \textbf{in}\ \ \mathit{VNum}\ (n_1 * n_2) \\
\quad |\ \mathit{Lt} & \textbf{lhs}.\mathit{val} & = & \textbf{let}\ (\mathit{VNum}\ n_1, \mathit{VNum}\ n_2) = (@t_1.\mathit{val}, @t_2.\mathit{val}) \\
 & & & \textbf{in}\ \ \textbf{if}\ n_1 < n_2\ \textbf{then}\ \mathit{VTrue}\ \textbf{else}\ \mathit{VTrue} \\
\quad |\ \mathit{Eq} & \textbf{lhs}.\mathit{val} & = & \textbf{let}\ (\mathit{VNum}\ n_1, \mathit{VNum}\ n_2) = (@t_1.\mathit{val}, @t_2.\mathit{val}) \\
 & & & \textbf{in}\ \ \textbf{if}\ n_1 \equiv n_2\ \textbf{then}\ \mathit{VTrue}\ \textbf{else}\ \mathit{VFalse} \\
\quad |\ \mathit{Gt} & \textbf{lhs}.\mathit{val} & = & \textbf{let}\ (\mathit{VNum}\ n_1, \mathit{VNum}\ n_2) = (@t_1.\mathit{val}, @t_2.\mathit{val}) \\
 & & & \textbf{in}\ \ \textbf{if}\ n_1 > n_2\ \textbf{then}\ \mathit{VTrue}\ \textbf{else}\ \mathit{VFalse}
\end{array}
$$