UU AG System User Manual

Arthur Baars, Doaitse Swierstra and Andres Löh
Department of Computer Science
Utrecht University
`doaitse@cs.uu.nl`

May 24, 2007

Contents

The UUAG system is an attribute grammar system developed at the University of Utrecht.

After the introduction, this document contains a user guide. This guide is divided in two parts, the first consists of an example introducing most language features, the second part covers the language constructs and the UUAGC compiler in more detail.

Any bugs (or fixes!) can be reported to the author, Arthur Baars (`doaitse@cs.uu.nl`). Any feedback on:

- what modifications you are interested in
- what modifications you have made yourself

is greatly appreciated too. Besides that, we are also quite interested in any applications, that are created using this system.

1 Getting Started

1.1 Running the UUAGC system

We assume that UUAGC compiler is installed on your system. If you run the compiler without arguments it will show you a short help message, and a list of options.

```
> uuagc
Usage info:
  uuagc options file ...
List of options:
  -m                       generate default module header
          --module[=name]  generate module header, specify module name
  -d      --data           generate data type definitions
  ...
```

In this user manual all the compiler switches and language features are introduced and explained in the examples.

1.2 Simple Attribute Grammar

As a first example we take the well known RepMin problem. The input of the program is a binary tree, and it produces a binary tree of the same shape. In the new tree however all values in the leaves are equal to the minimum of the values in the leaves in the original tree.

A grammar is defined as a collection of `DATA` declarations. The types correspond to the nonterminals and the constructors to the productions of the grammar. The grammar of binary trees is defined as follows:

```
DATA Tree
 | Node left:Tree right:Tree
 | Leaf int:Int
```

As in Haskell the names of the types and constructors start with an uppercase letter. The difference with a Haskell data type definition is that the fields of a constructor are associated with a name, and not only by position.

1.3 Adding attributes

In this section we define attributes to solve the Repmin problem. We split the computation to be performed into three different aspects:

- computing the minimal value
- making the minimal value available at the leaves
- constructing the final result

For each of the aspects we introduce an attribute and attribute computation rules.

Firstly we introduce a synthesized attribute *minval* representing the minimum value of a *Tree* by an `ATTR` declaration.

```
ATTR Tree [ | | minval:Int]
```

That *minval* is a synthesized attribute follows from the fact that its declaration is located after the second vertical bar. In an `ATTR` declaration there are three places to put attributes declarations.

$$[\ \textit{inherited} \mid \textit{inherited/synthesized} \mid \textit{synthesized}\ ] \qquad (1)$$

Attributes in the first position are inherited attributes, attributes in the last position are synthesized attributes, and attributes in the middle are inherited as well as synthesized.

Next we specify the computation of the minimum value by providing semantic rules.

```
SEM Tree
  | Leaf lhs.minval = { @int }
  | Node lhs.minval = { min @left.minval @right.minval }
```

To compute the minimum value of a *Leaf* we simply return the value of the *Leaf*. For a *Node* the minimum value is the minimum of *left*'s *minval* and *right*'s *minval*. The right-hand side of a semantic rule is a Haskell expression between braces. The references to attribute and field values are all marked with an '@' symbol. The left-hand side of a semantic rule is a reference to an attribute. In this case the *minval* attribute of *Tree*, which is the left hand side of the productions *Leaf* and *Node*, hence the name *lhs*.

1.4 Compiling an attribute grammar

The example code developed thusfar is can be found in `examples/Repmin1.ag`. This simple attribute grammar is compiled into a `Haskell` source file as follows:

```
> uuagc --module --data --semfuns --catas --signatures Repmin1.ag
Repmin1.hs generated
```

Using the functions in the generated `Haskell` program we can compute the minimum of a *Tree* as is shown in the following example:

```
Repmin1> sem_Tree (Node (Node (Leaf 2 )(Leaf 3))(Node (Leaf 1)(Leaf 2)))
1
```

1.5 Generated code

In this section we explain the following compiler options a and take a brief look at the code generated by the UUAG compiler.

short option	long option	description
`-m`	`--module[=name]`	generate module header, specify module name
`-d`	`--data`	generate data type definitions
`-f`	`--semfuns`	generate semantic functions
`-c`	`--catas`	generate catamorphisms
`-s`	`--signatures`	generate type signatures for semantic functions

The option `--module` tells the UUAG compiler to generate a `Haskell` module header. If a name is specified this name is used as the module name. If no name is specified or when the short option (`-m`) is used the module name is the name of the UUAG source file, without its extension. Hence the generated code for `RepMin1.ag` code contains the following module header:

```
module Repmin1 where
```

The option `--data` tells the UUAG compiler to generate `Haskell` data type definitions corresponding to the `DATA` statements in the attribute grammar. The data type definition generated for `RepMin1.ag` is:

```
data Tree = Leaf Int
          | Node Tree Tree
```

The `SEM` rules are compiled into semantic functions that compute the output attributes from the input attributes. For each nonterminal a type synonym, named `T_`*Type*,is introduced for the type of its semantics. In our example there are no inherited attributes and only a single synthesized attribute, namely *minval* with type *Int*. Hence the type synonym for the nonterminal *Tree* is:

```
type T_Tree = Int
```

The option `semfuns` tells the compiler to generate a semantic function for each constructor. They are named as follows: `sem_`*Nonterminal*`_`*Constuctor*. A semantic function takes the semantics of the constructor's children as argument to compute the semantics of the nonterminal. By providing the `--catas` the UUAG compiler generates catamorphisms for each data type in the attribute grammar. A catamorphism takes a data type and computes its semantics. This is achieved by and applying the appropriate semantic functions. The generated catamorphisms are named as follows: `sem_`*Type*. The option `--signatures` tells the compiler to emit type signatures for all semantic functions and catamorphisms. For our example these signatures are:

```
sem_Tree_Node :: T_Tree -> T_Tree -> T_Tree
sem_Tree_Leaf :: Int -> T_Tree
sem_Tree :: Tree -> T_Tree
```

The semantics of a child with a type that is not defined using a `DATA` statement is simply its value. Hence the type *Int* for the semantics of the value in a *Leaf*.

The actual code generated for semantics functions and catamorphisms is discussed in section 4.

1.6 RepMin continued

The attribute grammar developed thusfar computes the minimum of a tree. This computation is done bottom-up using a single attribute *minval*. The global minimum of a tree is the value of *minval* at the root node. To solve the "repmin" problem we need to distribute the global minimum to all the leaves and, then reconstruct the tree with each value replaced by the global minimum.

1.6.1 Distribute global minimum

The global minimum is the minimum value of the root node of the tree. In order to make the global minimum available at all the leaves we need to push the minimum value of the root node down to all the leaves.

Firstly we declare an inherited attribute *gmin* that holds the global minimum.

```
ATTR Tree [ gmin:{Int} | | ]
```

At each *Node* the global minimum is distributed to both children. Their is no rule for the constructor *Leaf* because it does not have children.

```
SEM Tree
  | Node left.gmin  = { @lhs.gmin }
         right.gmin = { @lhs.gmin }
```

The global minimum is passed down from parent nodes to their children. The root node of a *Tree*, however, does not have a parent, so we cannot set its inherited attribute *gmin*. We introduce a data type *Root* that serves as the parent of a *Tree*. It uses the synthesized attribute *minval* of the *Tree* to define the inherited attribute *gmin*.

```
DATA Root | Root tree:Tree
SEM Root
  | Root tree.gmin = { @tree.minval }
```

1.6.2 Construct the result

Now the global minimum is available everywhere in the tree we can construct the final result, that is a tree with the same structure as the original, but with the value stored in each leaf replaced by the smallest integer stored in the entire tree.

First we declare a synthesized attribute *result* for both *Tree* and *Root*.

```
ATTR Tree [ | | result:Tree ]
ATTR Root [ | | result:Tree ]
```

In a *Node* the resulting trees of both children are combined into a new *Node*. For a *Leaf* a new *Leaf* is returned containing the minimum value.

```
SEM Tree
  | Node lhs.result  = { Node @left.result @right.result }
  | Leaf lhs.result  = { Leaf @lhs.gmin }
```

At a *Root* the resulting tree is returned.

```
SEM Root
  | Root lhs.result = { @tree.result }
```

1.6.3 Haskell code blocks

To finish the rep-min example we define a number of `Haskell` functions. These definitions are written between braces and are copied literally into the output of the UUAGC System. The following code block defines an instance of *Show* for *Tree*, a sample *Tree* and a *main* function.

```
{
instance Show Tree where
  show tree = case tree of
               Leaf val -> "Leaf " ++ show val
               Node l r -> "Node (" ++ show l ++ ") (" ++ show r ++ ")"
example :: Tree
example =  Node (Leaf 3)(Node (Leaf 6)(Leaf 2))
main :: IO ()
main = do putStrLn "input tree:"
          print example
          putStrLn "result tree:"
          print (sem_Root (Root example))
}
```

1.6.4 Compile and Run

The example code developed thusfar is can be found in `examples/Repmin2.ag`. This attribute grammar is compiled into a `Haskell` source file as follows:

```
> uuagc --module=Main --signatures --data --semfuns --catas Repmin2.ag
Repmin2.hs generated
```

The generated code is a module named *Main* containing the *Tree* datatype, semantic functions, catamorphisms, and some additional `Haskell` definitions. The program can be run using `runhugs` as follows:

```
> runhugs Repmin2.hs
input tree:
Node (Leaf 3) (Node (Leaf 6) (Leaf 2))
result tree:
Node (Leaf 2) (Node (Leaf 2) (Leaf 2))
```

2 Language Constructs

This section gives an overview of the UUAG language. Lines printed in bold are grammar rules and show what the language construct looks like in general. Subscripts and "…"-notation are used in the syntax rules. For example:

$$\textbf{constructor name}_\mathbf{1}\,\textbf{:}\,\textbf{type}_\mathbf{1}\ldots\textbf{name}_\mathbf{n}\,\textbf{:}\,\textbf{type}_\mathbf{n} \quad (n \geq 0)$$

This means that a constructor has zero or more fields. Valid instantiations are:

```
Leaf val:Int
Bin left:Tree right:Tree
Empty
```

The following sections show the syntax of each construct as a grammar rule, followed by an explanation of its semantics and a number of examples. The UUAG language provides many shorthand notations. These abbreviations are explained by example, as including them in the grammar rules would clutter the presentation. A complete reference in EBNF of the UUAG language can be found in Section 5.

2.1 DATA declaration

$$\begin{array}{ll} \texttt{DATA} & \textbf{nonterminal} \\ & \texttt{|}\,\textbf{constructor}_\mathbf{1}\textbf{field}_{\mathbf{1,1}}\,\textbf{:}\,\textbf{type}_{\mathbf{1,1}}\ldots\textbf{field}_{\mathbf{1,i}}\,\textbf{:}\,\textbf{type}_{\mathbf{1,i}} \\ & \texttt{|}\,\vdots \\ & \texttt{|}\,\textbf{constructor}_\mathbf{n}\textbf{field}_{\mathbf{n,1}}\,\textbf{:}\,\textbf{type}_{\mathbf{n,1}}\ldots\textbf{field}_{\mathbf{n,j}}\,\textbf{:}\,\textbf{type}_{\mathbf{n,j}} \\ & (i \geq 0, j \geq 0, n \geq 0) \end{array}$$

A `DATA` declares a number of productions for a nonterminal. Each production is labelled with a constructor name. In contrast to `Haskell` it is allowed to use the same constructor name for more than one nonterminal. However, the names of all constructors of the same nonterminal must be different. Giving multiple `DATA` declarations for the same nonterminal is allowed, provided that the constructor names in the declarations do not clash. The fields of each production all have a name and a **type**. The type can be a nonterminal or a `Haskell` type. All fields of the same constructor must have different names.

Valid `DATA` declarations:

```
DATA Tree | Bin left:Tree right:Tree
          | Leaf value:Int
DATA Decl | Fun name:String args:{[String]} body:Expr
```

Several abbreviations exist for `DATA` declarations. Fields with the same type can be declared by listing their names separated by commas. Also the field name can be left out, in which case the name is defaulted to the type name with the first letter converted to lowercase. It is only allowed to leave out the field name if the type is an uppercase type identifier. You also need to make sure that the default name does not clash with the name of another field. The following example show correct abbreviations:

```
DATA Tree | Bin left,right:Tree -- 'left' & 'right' have type 'Tree'
          | Leaf Int            -- field name is 'int'
```

The following `DATA` statement is wrong:

```
DATA Tree | Bin Tree Tree    -- duplicate field name
          | Leaf {(Int,Int)} -- type is not a single type identifier
```

2.2 ATTR declaration

$$\begin{array}{ll}
\texttt{ATTR} & \mathbf{nonterminal_1} \ldots \mathbf{nonterminal_n} \\
 & \texttt{[}\ \mathbf{attr_1} : \mathbf{type_1} \ldots \mathbf{attr_i} : \mathbf{type_i} \\
 & \texttt{|}\ \mathbf{attr_{(i+1)}} : \mathbf{type_{(i+1)}} \ldots \mathbf{attr_j} : \mathbf{type_j} \\
 & \texttt{|}\ \mathbf{attr_{(j+1)}} : \mathbf{type_{(j+1)}} \ldots \mathbf{attr_k} : \mathbf{type_k} \\
 & \texttt{]} \\
 & (n \geq 1, 0 \leq i \leq j \leq k)
\end{array}$$

An `ATTR` declaration declares attributes for one or more nonterminals. Each attribute has a name and a type. The position of an attribute in the declaration list (left of the bars, between the bars, or right of the bars) determines whether it is inherited, chained, or synthesized, respectivly. A chained attribute is just an abbreviation for an attribute that is both inherited and synthesized. The names of all inherited attributes declared by `ATTR` statements must be different. The same holds for synthesized attributes.

Valid `ATTR` declarations are:

```
ATTR Tree [ depth:Int  | minimum:Int | out:{[Bool]} ]
ATTR Tree [ count:Int  | | count:Int ]
ATTR Decl [ environment : {[(String,Type)]} | | ]
ATTR Decl [ | | code:Instructions ]
```

For attribute declarations the same abbreviations are permitted as for field in a `DATA` declaration. The name of an attribute can be left out, and attributes with the same type can be grouped. For example:

```
ATTR Tree [ | | min,max:Int ] -- 'min' and 'max' both have type 'Int'
ATTR Decl [ Environment | | ] -- attribute name is 'environment'
```

The following abbreviations are wrong:

```
ATTR Tree [ | | Int Int ]          -- duplicate attribute names
ATTR Decl [ {[(String,Type)]} | | ] -- complex type without name
```

A `USE` clause can be added to the declaration of a synthesized or chained attribute, to trigger a special kind of copy rule(see Section 3.3). The first expression must be an operator, and the second expression is a default value for the attribute.

$$\mathbf{attr}\ \texttt{USE}\ \mathbf{expr_1}\ \mathbf{expr_2}\ :\ \mathbf{type}$$

For example:

```
DATA Tree
   | Bin left,right:Tree
   | Leaf value:Int
ATTR Tree [ | | value USE {+} {0} : Int ] -- compute sum of values
```

An attribute can be declared to be of type `SELF`. The type `SELF` is a placeholder for the type of the nonterminal for which the attribute is declared. For example:

```
ATTR Tree Expr [ | | copy:SELF ]
```

The `ATTR` statement above declares an attribute *copy* of type *Tree* for nonterminal *Tree*, and an attribute *copy* of type *Expr* for nonterminal *Expr*. Declaring a synthesized attribute of type `SELF` triggers a special copy-rule, that constructs a copy of the tree. Section 3.4 explains this type of copy-rule.

Attribute declarations can also be given in `DATA` or `SEM` statements after the name of the nonterminal. For example:

```
DATA Tree | Bin left,right:Tree
          | Leaf Int
ATTR Tree [ | | min:Int ]
```

can be combined into:

```
DATA Tree [ | | min:Int ]
   | Bin left,right:Tree
   | Leaf Int
```

2.3 SEM

In a SEM construct one can specify semantic rules for attributes. For each production the synthesized attributes associated with its corresponding nonterminal and the inherited attributes of its children must be defined. If there is a rule for a certain attribute is missing, the system tries to derive a so called copy-rule. The SEM construct has the following form:

```
SEM nonterminal
    |constructor_1   fieldref_1 . attribute_1=expression_1
                     ⋮
    |constructor_n   fieldref_n . attribute_n=expression_n
    (n ≥ 0)
```

Semantic rules are organised per production. Semantic rules for the same production can be spread between multiple SEM statements. This has the same meaning as they were defined in a single SEM statement. A **fieldref** is lhs, or loc, or a **field** name. To refer to a synthesized attribute of the nonterminal associated with a production the special **fieldref** lhs is used together with the name of the attribute. To refer to an inherited attribute of a child of a production the **field** name of the child is used together with the attribute's name. The special **fieldref** loc is used to define a variable that is local to the production. It is in the scope of all semantic rules for the production.

The expressions in semantic rules are code blocks, i.e. Haskell expressions enclosed by { and }, see Section 2.6. They may contains references to values of attributes and fields. These references are all prefixed with an @-sign to distinguish them from Haskell identifiers. To refer to the value of a field one uses the name of the field. References to attributes are similar to the ones on the left-hand side of a semantic rule (**fieldref . attribute**). The difference is that they now refer to the synthesized attributes of the children and the inherited attributes of the nonterminal associated with the production. Local variables can be referenced using their name, optionally prefixed with the special **fieldref** loc.

Valid definitions:

```
ATTR Tree [ gmin:Int | | min:Int result:Tree ]
SEM Tree
   | Bin  left.gmin  = { @lhs.gmin }
     -- "left.gmin" refers to the inherited attribute "gmin"
     -- of the child "left"
   | Bin  right.gmin = { @lhs.gmin }
     -- "@lhs.gmin" refers to the inherited attribute "gmin"
     -- of nonterminal "Tree"
   | Bin  loc.min    = { min @left.min @right.min }
     -- "min" is a new local variable of the constructor "Bin"
SEM Tree
  | Bin  lhs.result = { Bin @left.result @right.result }
    -- "@left.result" refers to the synthesized attribute "result"
    -- of child "left"
  | Bin  lhs.min    = { @min }
    -- "@min" refers to the local variable "min"
  | Leaf lhs.result = { Leaf @lhs.gmin }
    -- "@lhs.gmin" refers to the inherited attribute "gmin"
    -- of nonterminal "Tree"
  | Leaf lhs.min    = { @int }
    -- "@int" refers to the value of field "int" of "Leaf"
```

For the SEM construct there exist a number of abbreviations. As for DATA statements one can write attribute declarations after the name of the nonterminal. Furthermore semantic rules for the same production can be grouped, mentioning the name of the production only once. For example:

```
SEM Tree
   | Bin  left.gmin  = { @lhs.gmin }
          right.gmin = { @lhs.gmin }
          loc.min    = { min @left.min @right.min }
```

In a similar way semantic rules for the same **fieldref** can be grouped. For example:

```
SEM Tree
  | Bin  lhs.result = { Bin @left.result @right.result }
            .min    = { @min }
```

When the same semantic rule is defined for two productions of the same nonterminal they can be combined by writing the names of both productions in front of the rule. For example:

```
SEM Tree
  | Node1 lhs.value = { @left.value + @right.value }
  | Node2 lhs.value = { @left.value + @right.value }
```

can be abbreviated as follows:

```
SEM Tree
  | Node1 Node2 lhs.value = { @left.value + @right.value }
```

Finally the curly braces around expressions may be left out. The layout of the code is then used to determine the end of the expression as follows. The column of the first non-whitespace symbol after the =-sign is the reference column. All subsequent lines that are indented the same or further to the right are considered to be part of the expression. The expression ends when a line is indented less than the reference column. An advantage of using layout is that problems with unbalanced braces, as described in Section 2.6 are avoided.

2.4 TYPE

The TYPE construct is convenient notation for defining list based types. It has the following form:

TYPE **nonterminal** = [**type**]

A TYPE construct is equivalent to:

```
DATA  nonterminal
      | Cons hd:type tl:nonterminal
      | Nil
```

Apart from a convenient notation the TYPE construct has effect on the code generated. Instead of generating data constructors Cons and Nil Haskell's list constructors :, and [] are used.

Examples of TYPE constructs:

```
TYPE IntList = [ Int ]
TYPE Trees   = [ Tree ]
```

2.5 INCLUDE

Other UUAG files can be included using the following construct:

INCLUDE **string**

The **string** is a file name, between double quotes. The suffix of the file (.ag , or .lag) should not be omitted. The file should contain valid UUAG statements. These statements are inlined in the place of the INCLUDE statement.

2.6 Code Block

A code block is a piece of `Haskell` code enclosed by curly braces.

{ **haskellcode** }

There exist three kinds of code blocks: top-level, type, and expression code blocks. A top-level code block contains `Haskell` declarations, such as `import` declarations, and function and type definitions. A name can be writen before a top-level code block. The code blocks are sorted by their names, and appended to the code generated by the UUAG system. A special name `imports` is used to mark code blocks containing `import` declarations. These are copied to the start of the generated code, as `Haskell` only allows `import` declarations at the beginning of a file.

An example of two code blocks, an import declaration and a function definition:

```
imports
{
import List
}
quicksort
{
-- simple implementation of quicksort:
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = let (l,r) = partition (<=x) xs
                in qsort l ++ [x] ++ qsort r
}
```

A type code block contains a `Haskell` type and may be used as **type**s in `DATA`, `TYPE`, and `ATTR` declarations. Examples:

```
DATA Module
   | Module name:{Maybe String} body:Declarations
TYPE Points = [ {(Int,Int)} ]
ATTR [ env:{[(String,Int)]} | | ]
```

Finally expression code blocks contain a `Haskell` expression and occur as the right-hand side of attribute definitions in `SEM` statements. Apart from normal `Haskell` code they may contain references to attributes. These references are prefixed with an `@`-symbol, to distinguish them from ordinary `Haskell` identifiers. Examples:

```
SEM Tree [ | | min:{Int} ]
  | Node lhs.min = { min @left.min @right.min } -- an expression code block
```

The contents of a block is the plain text between an open and a close brace. The text in a code block is not interpreted by the UUAG system.

any	::=	[“\0”..“\255”]	(*any character*)
codeblock	::=	“{”**codeblockcontent**∗“}”	
codeblockcontent	::=	**any**	*except* {, *and* }
	|	**codeblock**	

Curly braces occurring inside the `Haskell` code must be balanced. This includes curly braces in comments, and in string and character literals.

An example of a code block containing a nested pair of braces:

```
{
f a b c = let { d = b*b - 4*a*c
              ; result1 = (-b + sqrt d) / 2*a
              ; result2 = (-b - sqrt d) / 2*a
              ; result  | d >  0 = [result1, result2]
                        | d == 0 = [result1]
                        | d < 0  = []
              }
          in result
}
```

All curly braces `Haskell` constructs, such as `do`, `let` must be matched. However, curly braces in string, or character literals may cause problems. The balancing rule forbids code blocks such as:

```
{
openbrace = "{"
}
```

This problem can be fixed by inserting a matching brace in comments. In the following code the curly braces are balanced:

```
{
openbrace = "{"
-- }, just to balance braces
}
```

2.7 Comments

One-line comments start with two dashes (`--`) and end at the end of the line. Multi-line comments start with `{-` and end with `-}`. As in `Haskell` comments can be nested.

```
{-
Definition of a datatype for binary trees
-}
DATA Tree
   | Leaf val:Int
   | Node left:Tree right:Tree -- a node has two subtrees
```

2.8 Names

Names start with a letter followed by a (possibly empty) sequence of letters, digits, and the symbols `_` and `'`. A name for a **nonterminal** or **constructor** must start with an upper-case letter. A name of a **field** or **attribute** must start with a lower-case letter. The following words are reserved and cannot be used as names: `DATA`, `EXT`, `ATTR`, `SEMTYPE`, `USE`, `loc`, `lhs`, and `INCLUDE`.

Valid names:

```
-- nonterminals or constructors:
Node
Expression
Tree_Node
-- field names or attributes:
left
long_name
field2
```

2.9 Strings

A **string** in UUAGC is sequence of characters enclosed by double quotes (`"`). The structure of strings is similar to `Haskell` strings. The escape character is a backslash (\). Below a table with the most common escape sequences:

`\'`	single quote (`'`)
`\"`	double quote (`"`)
`\n`	newline
`\t`	tab
\nnn	character with ascii-code *nnn*

For a more detailed description of string and escape sequences see the Haskell Report[?]. Examples of valid strings:

```
"hello world"
"line 1\nline 2"
"hello\32world"
```

3 Copy Rule

When a definition for an attribute is missing, the UUAG can often derive a rule for it. These automatic rules, also known as copy rules, are based on name equality of attributes. They save a lot of otherwise trivial typing, thus making your programs easier to read by just leaving the essential parts in the code. If in the list of rules for a constructor a rule for an attribute *attr₁* is missing then UUAG system tries to derive a rule for this attribute. This is done by looking for an attribute *attr₂* with the same name as *attr₁* in the sets of synthesized attributes of the children of the constructor and in the set of inherited attributes of the nonterminal it belongs to. If such an attribute *attr₂* is found then the value of *attr₁* is set to the value of *attr₂*. This section firstly shows two examples and then defines a generalisation that captures both(and others). There are also two special copy rules, the `USE`, and `SELF` rules, which are explained at the end of this section.

3.1 Examples

Very often one needs to pass a value from a node to all its children. Consider for example the following code, in which a inherited attribute *gmin* is declared.

```
DATA Tree | Bin left,right:Tree
          | Leaf val:Int
ATTR Tree [ gmin:Int | | ]
```

In this example rules for the syntesized attribute *gmin* of children of the constructor *Bin* are missing. This is however no problem. The nonterminal *Tree* has an inherited attribute with the same name and the UUAG system automatically inserts the following rules:

```
SEM Tree
  | Bin left.gmin  = @lhs.gmin
        right.gmin = @lhs.gmin
```

This kind of copy-rule is very convenient for copying an inherited attribute to all nodes in a top-down fashion.

Another kind of copy-rule is a co-called chain-rule. For a chain rule an attribute that is both inherited as well as synthesized is chained from left to right through all children of a constructor. Consider for example the following code that numbers all leaves in a *Tree* from left to right.

```
ATTR Tree [ | label:Int | ]
SEM Tree
  | Leaf lhs.label = @lhs.label+1
```

Because the attribute *label* is declared inherited as well as synthesized the UUAG system derives the following rules for the constructor *Bin*:

```
SEM Tree
  | Bin left.label  = @lhs.label
        right.label = @left.label
        lhs.label   = @right.label
```

3.2 Generalised copy rule

The UUAG system implements a more general copy rule of which the examples above are instances. If a rule is missing for an inherited attribute *n* of a child *c* of constructor *con*, the UUAG system searches for an attribute with the same name(n). The UUAG system searches for a suitable candidate in the following lists:

1 local attributes
2 synthesized attributes of children on the left of c
3 inherited attributes
4 fields

The search takes place in the order defined above, and the first occurrence of n is copied. Thus local attributes have preference over others. When there are multiple occurrences of n in the list of synthesized attributes of the children the rightmost is taken.

When a rule for a synthesized attribute is missing the search for a candidate with the same name takes place in a similar fashion. In the second step all children are searched, again taking the rightmost candidate if more than one is found.

3.3 `USE` rules

A `USE` rule can be derived for a synthesized attribute whose declaration includes a `USE` clause. A `USE` clause consists of two expressions; the first is an operator, and the second is a default value. Suppose s is a synthesized attribute of n, that is declared with a `USE` clause. If for a constructor c of n a definition of s is missing, a rule is derived as follows. Collect all synthesized attributes of constructor c's children with the same name as s. If this collection is empty the default value declared in the `USE` clause is taken. If this collection contains only a single attribute, then the value of this attribute is copied. Otherwise the values of the attributes are combined using the operator and the result is used to define s.

For example:

```
DATA Tree
   | Bin left,right:Tree
   | Single val:Int
   | Empty
ATTR Tree [ || sum USE {+} {0} : Int]
SEM Tree
  | Single lhs.sum = @val
```

The UUAG system derives the following rules:

```
SEM Tree
  | Bin   lhs.sum = @left.sum + @right
  | Empty lhs.sum = 0
```

3.4 `SELF` rules

The type `SELF` in an attribute declaration is equivalent to the type of the nonterminal to which the attribute belongs. A synthesized `SELF` attribute can for example be used if one wants a local copy of a tree, or wants to transform it. The `SELF` attribute then holds the transformed version of the tree. A `SELF` attribute usually holds a copy of the tree, except for a few places where a transformation is done. The semantic rules required for constructing a copy of a tree call for each production the corresponding constructor function on the copies of the children. The UUAG system implements a special copy rule to avoid writing these trivial rules. For each production of a nonterminal with a synthesized `SELF` attribute n, the UUAG system generates a local attribute containing the application of the corresponding constructor to the `SELF` attributes of the children with the same name as n. The value of the synthesized attribute is set to this local attribute.

For example for:

```
DATA Tree
   | Bin left,right:Tree
   | Leaf val:Int
```

```
ATTR Tree [ | | copy : SELF ]
```

the following semantic rules are generated:

```
SEM Tree
  | Bin  loc.copy = Bin @left.copy @right.copy
         lhs.copy = @copy
  | Leaf loc.copy = Leaf @val
         lhs.copy = @copy
```

The default definitions for the local and sythesized `SELF` attributes can be overriden by the programmer.

The following program is a complete attribute grammer for the rep-min problem using as many copy rules as possible. For constructing the transformed a `SELF` attribute *result* is used. Note that only for the production *Leaf* an explicit definition of this attribute is given. The definition for *Bin* is provided by an automatic rule.

```
DATA Tree
   | Bin left,right:Tree
   | Leaf val:Int
DATA Root
   | Root Tree
ATTR Tree [ gmin:Int | | lmin USE {`min`} {0}:Int ]
ATTR Root Tree [ | | result:SELF ]
SEM Tree
  | Leaf lhs.lmin   = @val
            .result = Leaf @lhs.gmin
SEM Root
  | Root tree.gmin = @tree.lmin
```

4 Code Generation

4.1 Module header

If the option `-m` or `--module=[name]` is provided to the UUAG compiler then a module header will be generated. If a name is provided to the `--module` flag then this name is used as module name, otherwise the module name will be the filename without the suffix `.ag` or `.lag`.

4.2 Data types

When the flag `--data` or `-d` is passed to the UUAG compiler, then a data type definition is generated for each nonterminal introduced in a `DATA` declaration and a type synonym is generated for each nonterminal introduced in a `TYPE` declaration. The UUAG system allows different nonterminals to have constuctors with the same names. For `Haskell` data types this is not allowed. To prevent clashes between constuctors of different data types the flag `--rename` or `-r` can be specified. All constructors will then be prefixed with their corresponding nonterminal(and an underscore).

For example for this fragment of UUAG code:

```
DATA Expr
   | Var name:String
   | Apply fun:Expr arg:Expr
   | Tuple elems:Exprs
   | ...
TYPE Exprs = [Expr]
DATA Type
   | Var name:String
   | Apply fun:Type arg:Type
```

```
     | ...
```

the following Haskell code is generated when the flags `--data` and `--rename` are switched on:

```
data Expr
   = Expr_Var String
   | Expr_Apply Expr Expr
   | ...
type Exprs = [Expr]
data Type
   = Type_Var String
   | Type_Apply Type Type
   | ...
```

If the `--rename` flag is not provided it is the responsibility of the programmer to make sure that are constructors are uniquely named.

4.3 Semantic functions

The semantic domain of a nonterminal is a mapping from its inherited to its synthesized attributes. When the flag `--semfuns` or `-f` is switched on, the UUAG compiler generates for each nonterminal a type synonym representing its semantic domain, and for each constructor a semantic function. A semantic function takes the semantics of its children as arguments and returns the semantics of the corresponding nonterminal. A semantic function is named as follows:
prefix_nonterminal_constructor.
The default prefix is `sem`, another prefix can be supplied with the `--prefix=name` flag.

Consider the following code fragment:

```
DATA Tree
   | Bin left,right:Tree
   | Leaf val:Int
ATTR Tree [ lmin:Int gmin:Int | | lmin:Int result:Tree ]
SEM Tree
  | Bin  lhs.result = Bin @left.result @right.result
  | Leaf lhs.lmin   = min @lhs.lmin @val
         lhs.result = Leaf @lhs.gmin
```

The semantic domain of the nonterminal **Tree** is defined as follows:

```
type T_Tree = Int -> Int -> (Int,Tree)
```

The inherited attributes are arguments and the synthesized attributes are packed together in a tuple as result. The UUAG system lexicographically sorts the attributes, hence the first `Int` stands for the inherited attribute *gmin*, and the second for the inherited attribute *lmin*. If the flag `--newtypes` is switched on, a `newtype` declaration is generated for the semantic domain instead of a `type` synonym.

The types of the generated semantic functions for the constructors *Bin*, and the *Leaf* are the following:

```
sem_Tree_Bin  :: T_Tree -> T_Tree -> T_Tree
sem_Tree_Leaf :: Int              -> T_Tree
```

Note that the semantics of a child that has a Haskell type is simply the value of that child. When the flag `--signatures` or `-s` is switched on then the type signatures of the semantic functions are actually emmited in the generated code.

4.4 Catamorphisms

When the flag `--catas` or `-c` is supplied, the the UUAG compiler generates catamorphisms for every nonterminal. A catamorphism is a function that takes a (syntax) tree as argument and

computes the semantics of that tree. The catamorphism for a nonterminal **nt** is named as follows: **prefix_nonterminal**.
As for semantic functions the prefix is `sem` by default, and can be changed with the `--prefix=name` flag. For example the type of the catamorphism for the nonterminal **Tree** is:

```
sem_Tree :: Tree -> T_Tree
```

When the flag `--signatures` or `-s` is switched on then the type signatures of the catamorphisms are actually emmited in the generated code.

4.5 Wrappers

The result of a semantic function or a catamorphism is a function from inherited to synthesized attributes. To be able to use such a result, a programmer needs to know the order of all the attributes. Wrapper functions for the semantic domains can be generated to provide access to the attributes by their names. When the flag `--wrappers` or `-w` is switched on the following is generated for each semantic domain:

- a record type with named fields for the inherited attributes
- a record type with named fields for the synthesized attributes
- a wrapper function that transforms a semantic domain in a function from a record of inherited attributes to a record of synthesized attributes

The two record types for a nonterminal *nt* are called *nt*`_Inh`, and *nt*`_Syn`, for the inherited and synthesized attributes, respectively. The labels of the records are the names of the attributes suffixed with the name of the record type. The generated wrapper function is named `wrap_`*nt*.

For the nonterminal **Tree** in the example above the following record types are generated:

```
data Tree_Inh = Tree_Inh{ lmin_Tree_Inh :: Int
                        , gmin_Tree_Inh :: Int
                        }
data Tree_Syn = Tree_Syn{ lmin_Tree_Syn   :: Int
                        , result_Tree_Syn :: Tree
                        }
```

The generated wrapper function has the following type:

```
wrap_Tree :: T_Tree -> Tree_Inh -> Tree_Syn
```

Using the generated wrapper code the function *repmin* can be defined as follows:

```
repmin :: Tree -> Tree
repmin tree = let synthesized = wrap_Tree (sem_Tree tree) inherited
                  inherited   =
                        Tree_Inh
                        { lmin_Tree_Inh = infty
                        , gmin_Tree_Inh = lmin_Tree_Syn synthesized
                        }
                  infty       = 1000
              in result_Tree_Syn synthesized
```

5 Grammar

Normal UUAG system source files have `.ag` as suffix. The UUAG system also supports literate programming. Literate UUAG files have `.lag` as suffix. In literate mode all text in a file is considered to be comments, except for those blocks enclosed between: `\begincode`, and `\endcode`. The begin and end commands should be placed at the beginning of a line.

The remainder of this section presents the grammar of the UUAG system as EBNF production rules. Parenthesis are used for grouping, nonterminals are printed **boldface**, and terminal symbols are printed between "`quotes`". A rule of form $X*$ means zero or more occurrences of X, $X+$ means one or more occurrences of X, and X? is an optional occurrence of X. In the lexical syntax character ranges are written between square brackets. For example [“A”..“Z”] represents the range of uppercase letters.

5.1 Lexical Syntax

keywords	=	{ “DATA”, “EXT”, “ATTR”, “SEM”, “TYPE” , “USE”, “loc”, “lhs”, “INCLUDE” }
uppercase	::=	[“A” .. “Z”]
lowercase	::=	[“a” .. “z”]
any	::=	[“\0” .. “\255”] (any character)
conid	::=	**uppercase identletter*** except **keywords**
varid	::=	**lowercase identletter*** except **keywords**
identletter	::=	**uppercase**
	|	**lowercase**
	|	“’”
	|	“_”
string	::=	“"” **stringcontents** “"”
codeblock	::=	“{” **codeblockcontent*** “}”
codeblockcontent	::=	**any** except “{”, and “}”
	|	**codeblock**
layoutcodeblock	::=	**layoutcontent***
layoutcontent	::=	**any** (except letters that are less indented than reference column)

5.2 Context-free Grammar

ag	::=	**elem***
elem	::=	“DATA” **conid attrDecls**? **dataAlt***
	|	“ATTR” **conid**+ **attrDecls**
	|	“TYPE” **conid** “=” “[” **type*** “]”
	|	“SEM” **conid attrDecls**? **semAlt***
	|	**varid**? **codeblock**
	|	“INCLUDE” **string**
attrDecls	::=	“[” **inhAttrDecl*** “|” **synAttrDecl*** “|” **synAttrDecl*** “]”
type	::=	**conid**
	|	**codeBlock**
inhAttrDecl	::=	**varids** “:” **type**
varids	::=	**varid** (“,” **varid**)*
synAttrDecl	::=	**varids** (“USE” **codeBlock codeBlock**)? “:” **type**
dataAlt	::=	“|” **conid field***
field	::=	**varids** “:” **type**
	|	**conid**
semAlt	::=	“|” **conid**+ **semDef***
semDef	::=	(**varid** | “lhs”) **attrDef**+
	|	“loc” **locDef**+
attrDef	::=	“.” **varid assign expr**

locDef	::=	“`.`” **pattern assign expr**
expr	::=	**codeBlock**
	|	**layoutCodeBlock**
assign	::=	“`=`”
	|	“`:=`”
pattern	::=	**conid pattern**$_1$*
	|	**pattern**$_1$
pattern$_1$	::=	**varid** (“`@`” **pattern**$_1$)?
	|	“`(`” **patterns**? “`)`”
	|	“`_`”
patterns	::=	**pattern** (“`,`” **pattern**)*

6 Compiler flags

short option	long option	description
`-m`	`--module[=name]`	generate module header, specify module name
`-d`	`--data`	generate data type definitions
`-f`	`--semfuns`	generate semantic functions
`-c`	`--catas`	generate catamorphisms
`-s`	`--signatures`	generate type signatures for semantic functions
	`--newtypes`	use newtypes instead of type synonyms
`-p`	`--pretty`	generate pretty printed list of attributes
`-w`	`--wrappers`	generate wappers for semantic domains
`-r`	`--rename`	prefix data constructors with the name of corresponding type
	`--nest`	use nested pairs, instead of large tuples
`-o file`	`--output=file`	specify output file
`-v`	`--verbose`	verbose error message format
`-h`,`-?`	`--help`	get usage information
`-a`	`--all`	do everything (-dcfsprm)
	`--prefix=prefix`	set prefix for semantic functions, default is `sem_`
	`--self`	generate self attribute for all nonterminals
	`--cycle`	check for cyclic attribute definitions
	`--version`	get version information

The “Artistic License”

Preamble

The intent of this document is to state the conditions under which a Package may be copied, such that the Copyright Holder maintains some semblance of artistic control over the development of the package, while giving the users of the package the right to use and distribute the Package in a more-or-less customary fashion, plus the right to make reasonable modifications.

Definitions

- “Package” refers to the collection of files distributed by the Copyright Holder, and derivatives of that collection of files created through textual modification.
- “Standard Version” refers to such a Package if it has not been modified, or has been modified in accordance with the wishes of the Copyright Holder as specified below.
- “Copyright Holder” is whoever is named in the copyright or copyrights for the package.
- “You” is you, if you’re thinking about copying or distributing this Package.
- “Reasonable copying fee” is whatever you can justify on the basis of media cost, duplication charges, time of people involved, and so on. (You will not be required to justify it to the Copyright Holder, but only to the computing community at large as a market that must bear the fee.)
- “Freely Available” means that no fee is charged for the item itself, though there may be fees involved in handling the item . It also means that recipients of the item may redistribute it under the same conditions they received it.

1 You may make and give away verbatim copies of the source form of the Standard Version of this Package without restriction, provided that you duplicate all of the original copyright notices and associated disclaimers.

2 You may apply bug fixes, portability fixes and other modifications derived from the Public Domain or from the Copyright Holder. A Package modified in such a way shall still be considered the Standard Version.

3 You may otherwise modify your copy of this Package in any way, provided that you insert a prominent notice in each changed file stating how and when you changed that file, and provided that you do at least ONE of the following:
   - place your modifications in the Public Domain or otherwise make them Freely Available, such as by posting said modifications to Usenet or an equivalent medium, or placing the modifications on a major archive site such as uunet.uu.net, or by allowing the Copyright Holder to include your modifications in the Standard Version of the Package.
   - use the modified Package only within your corporation or organization.
   - rename any non-standard executables so the names do not conflict with standard executables, which must also be provided, and provide a separate manual page for each non-standard executable that clearly documents how it differs from the Standard Version.
   - make other distribution arrangements with the Copyright Holder.

4 You may distribute the programs of this Package in object code or executable form, provided that you do at least ONE of the following:
   - distribute a Standard Version of the executables and library files together with instructions (in the manual page or equivalent) on where to get the Standard Version.
   - accompany the distribution with the machine-readable source or the Package with your modifications.
   - give non-standard executables non-standard names, and clearly document the differences in manual pages (or equivalent), together with instructions on where to get the Standard Version.
   - make other distribution arrangements with the Copyright Holder.

5 You may charge a reasonable copying fee for any distribution of this Package. You may charge any fee you choose for support of this Package. You may not charge a fee for this Package itself. However, you may distribute this Package in aggregate with other (possibly