An Attribute Grammar Evaluator in Haskell

David Rushall

Department of Computer Science,

Victoria University of Manchester,

Oxford Road, Manchester, UK. M13 9PL.

drushall@uk.ac.man.cs

September 1992

Abstract. A tool that enables Haskell functions to be automatically derived from attribute grammar definitions is presented. Based on work in [Joh87], it is able to deal with general attribute grammars, provided the data dependencies can be resolved under lazy evaluation. The tool takes the form of a pre-processor, taking attribute grammar specifications and producing Haskell that can be compiled through the preferred compilation route. The paper includes a brief introduction to attribute grammars, why they are useful in the implementation of certain types of problems, and how the evaluator can generate native Haskell code from their specifications. Examples are used throughout, and executable scripts are provided as an appendix.

1. Introduction

When implementing a compiler, or some similar complex program, it is common to build some internal data structure and make a number of passes over it. These passes may gather information from the nodes, annotate nodes with information gathered or replace parts of the data structure with some new value.

Often it is theoretically possible to convert these many pass algorithms into a single pass over the structure. Doing so would probably be more efficient as the data is only traversed once and intermediate structures do not need to be built. This is especially important in function languages as, in the absence of side effect, it is not possible to implement annotations by making slight ‘in place’ modifications to the original structure. Instead it is necessary to completely rebuild the structure, in the worse case. This is both expensive in terms of computation and, if the original structure is also required later, in terms of store.

Unfortunately, such single pass algorithms can prove to be extremely complex. The programs produced may depend on subtleties in evaluation order and side effect making them difficult to write, prove correct, understand and maintain. It is therefore common that the many pass algorithms are used at the expense of efficiency.

By allowing the definition of single pass algorithms in a systematic, easily understood form attribute grammars may provide a solution. They are particularly suitable for implementation in a lazy functional language, such as Haskell[Has91]. Although translating attribute grammars into the lazy language syntax is relatively straight forward, it is a tedious task. Whilst being relatively easy to derive, the resulting program is cluttered by the overheads necessary for the propagation of attributes, and once again becomes difficult to understand and prove correct.

The attribute grammar evaluator presented in this paper provides a method by which Haskell functions may be derived from a simple attribute grammar definition. It takes the from of a pre-processor which reads in

a file which may contain a number of attribute grammar definitions embedded in a Haskell module, and produces an equivalent module in native Haskell. This can then be compiled via a normal compilation route.

In the following sections, it is assumed that the reader is familiar with the concept of lazy functional programming languages, and preferably with the syntax of Haskell.

2. An Introduction to Attribute Grammars

An *attribute grammar* is a context free grammar augmented with attributes and semantic rules. Each non-terminal symbol in the grammar has set of attributes associated to it. Attributes are one of two kinds:

- *synthesised* attributes are passed up from child symbols to their parent and tend to propagate up towards the root of the grammar;
- *inherited* attributes are obtained by a symbol from its parent and tend to propagate down towards the leaves.

For a given production,

$$X_0 \text{ -> } X_1, X_2, \ldots X_n$$

a set of semantic rules define the attributes of the non-terminals symbols. These definitions take the form,

$$a := f(c_1, c_2, \ldots, c_k)$$

where f is an arbitrary function; $c_1 \ldots c_k$ are the attributes of $X_0 \ldots X_n$; and either

- b is a synthesised attribute of X_0, or
- b is an inherited attribute of one of $X_1 \ldots X_n$.

An attribute grammar that contains only synthesised attributes is said to be *S-attributed*. An *L-attributed* grammar is defined by the fact that each inherited attribute of X_i $(1 <= i <= n)$ depends only on,

- the attributes of the symbols $X_1 \ldots X_{(i-1)}$, or
- the inherited attributes of X_0.

It can be seen that S-attributed grammars are a subset of L-attributed grammars. For many problems an L-attributed grammar may be sufficient, however a larger set of attribute grammars can be useful in some cases.

2.1 A Brief History

Attribute grammars were originally designed as a method for specifying the semantics of programming languages [Knu68] but are more often used today for the specification of syntax directed translations, such as in compilers. An example of this can be found in [Tar88] which describes a compiler specified as an attribute grammar in a single pass.

In [Joh87], Johnsson presents attribute grammars as a tool by which a single pass algorithm can be derived for many arbitrary complex problems. This is achieved by making the observation that decorating the nodes of a tree with attributes is not only applicable to parse trees, but to any data structure. By implementing example problems through an attribute grammar, Johnsson shows that a program can be formed which is more efficient than the naive approach in a at least one of the following ways:

- the program should require only one pass over the data structure;

- if only one pass is necessary it is no longer necessary to build intermediate data structures, a saving in both execution time, storage and type definitions required;
- in some cases, it may be possible to reduce the complexity of the algorithm, by reducing the exponential order.

Johnsson refers to [Bir84], in which another technique for transforming many pass functional programs into single pass ones is presented. In order to define circular dependencies with this technique it is essential that the language is lazy and provides local recursion. The technique makes use of simple program transformations but has the disadvantage of requiring the programmer to find a 'eureka' step. This can be extremely difficult and can result in rather a contorted program, making the derivation and maintenance of such programs available only to more sophisticated programmers.

By specifying the problems as attribute grammars, Johnsson is able to derive the same programs as Bird, but without the need for a 'eureka' step. He also presents a language construct suitable for the specification of attribute grammars and details how this can be dealt with at compile time.

In [Fro92], an alternative method of implementing executable attribute grammar specifications under a lazy functional language is presented. This differs from Johnsson's method in that does not operate on an abstract syntax tree, rather it interprets the problem expressed in a 'language' by means of an attribute grammar specified in a 'passage'. The specification is by means of four combinators which return interpreters for the problem. The passage can be easily translated into a language such as Haskell by simple textual replacement.

When implementing a program specified as an attribute grammar in a traditional imperative language, one must be aware of the interdependencies of the various attributes. It is necessary to derive an evaluation order which ensures no attribute is used until is is defined, which may be a complex task if the grammar is not L-attributed. Some tools, such as *Ag* [Gro91] exist to aid in this process, although they may be only operate on a restricted set of attribute grammars.

Functional languages provide a natural way of implementing attribute grammars since the language can mirror their declarative style in a succinct manner. Attributes can be a value of any type, allowing them to be integers, lists, trees or functions, for example. If a lazy functional language is used, it is not necessary to consider the evaluation order of the attributes as it will be determined at run-time by the lazy evaluation mechanism. In addition it is possible to express those attribute grammars prohibited by other evaluation mechanisms due to complex dependencies, without further complication. It is likely that an efficient lazy functional implementation of an attribute grammar will be less efficient that an implementation in 'C', for example. However, a lazy implementation has the advantage that the set of attribute grammars that can be implemented easily is not restricted to L-attributed grammars.

3. Expressing a Simple Problem as an Attribute Grammar

As an example of the power of an attribute grammar, consider the following problem. A non-terminal, *tree*, produces two non-terminal symbols *Node* and *Leaf*. *Node* contains two further non-terminals, forming a left and a right branch of the tree, whilst a *Leaf* contains a single integer value:

tree -> Leaf val

tree -> Node $tree_1$ $tree_2$

The task set is this. Given a *tree*, produce a tree of the same shape as the original, but with each *Leaf* in the output tree containing the sum of the leaves of the input tree.

A simple solution to the problem would be to perform this process in two passes. The first pass would wander over the tree, calculating the sum of the leaves. Then a second pass could create a tree of the same shape, but with the sum previously calculated being placed in each of the leaves. This might be achieved with the following Haskell code:

```
summedTree tree = mkNew (calcSum tree) tree
        where
        calcSum (Leaf n)           = n
        calcSum (Node left right) = calcSum left + calcSum right
        mkNew sum (Leaf _)         = Leaf sum
        mkNew sum (Node left right)
                                   = Node (mkNew sum left) (mkNew sum right)
```

With the use of an attribute grammar, it becomes possible to specify this operation with a single pass. Our grammar has a single inherited attribute, *globalSumIn*. This contains the sum of the leaves of the entire tree. The grammar also has two synthesised attributes: *localSumOut* contains the sum of all leaves in the sub-tree from which the symbol is the root; whilst *newTree* is a sub-tree of the final result.

We can now present the attribute grammar. We write the grammar productions as above, but now we also define values for the attributes. Attributes are named as the name of the symbol to which the attribute belongs and the name of the attribute itself, separated by an apostrophe (’). For example, *tree’globalSumIn* is the attribute *globalSumIn* of the symbol *tree*.

First we shall consider the simpler of the two productions, that for a *Leaf*. This does not produce any child non-terminals so the only attributes to be defined are the synthesised ones for the symbol. The value of *localSumOut* is simply the value contained in the original leaf, and the value of *newTree* is a new *Leaf* containing the inherited attribute *globalSumIn*. This can be seen in the following production:

tree -> Leaf val
tree’localSumOut := *val*
tree’newTree := *Leaf tree’globalSumIn*

It is now necessary to look at the more complicated production for *Node*. Here, we encounter two further non-terminal symbols, $tree_1$ and $tree_2$. It is therefore necessary to define the inherited attributes for these symbols, of which there is only the one, *globalSumIn*. Since the task of *globalSumIn* is to carry the sum of all the nodes in the tree down to the leaves it is simply defined as being equal to *tree’globalSumIn*. We still have to consider the values of the synthesised attributes of *tree*: *localSumOut* is the sum of the local sums of the child symbols; and *newTree* is a *Node* containing the new trees created by the children.

tree -> Node $tree_1$ $tree_2$
$tree_1$*’globalSumIn* := *tree’globalSumIn*
$tree_2$*’globalSumIn* := *tree’globalSumIn*
tree’localSumOut := $tree_1$*’localSumOut* + $tree_2$*’localSumOut*
tree’newTree := *Node* $tree_1$*’newTree* $tree_2$*’newTree*

All that remains is to define a top level for the attribute grammar. This takes the form of a production that will only ever be used to match the root symbol of the grammar and defines the inherited attributes of this symbol. For this example, the *localSumOut*, which contains the sum of all the leaves in the tree, is passed

back down as *globalSumIn*. Notice that since *localSumOut* is a synthesised attribute, we are using it before it has been evaluated. As a consequence, this grammar is not L-attributed.

```
tree -> tree
        tree'globalSumIn        := tree'localSumOut
```

It can be seen that the attribute grammar specifies the problem with a single pass over the data structure. How this specification can be transformed into an executable program in a lazy functional language will be described in the next section. It is worth pointing out at this stage, that whilst the above example is a fairly trivial one, a similar technique can be applied to obtain single pass algorithms for much more complex tasks, such as lambda lifting [Joh85] and code generation.

4. Transforming Attribute Grammars into Lazy Functional Languages

In [Joh87], Johnsson gives a simple transformation by which an attribute grammar can be expressed in a lazy functional language. This technique is applied to a number of example problems from [Bir84], deriving exactly the same code without the need to find a 'eureka' step.

Johnsson's transformation is based on the idea of translating an attribute grammar with arbitrary inherited and synthesised attributes, into an S-attributed grammar with a single synthesised attribute. This is attribute takes the form of a function which when applied to the inherited attributes of the original grammar, returns the synthesised ones. For example one could rewrite the *Node* production in the example in section 3 as such an S-attributed grammar as follows:

```
tree -> Node tree1 tree2
        tree'fn := \ tree'globalSumIn . letrec
                        (tree1'localSumOut, tree1'newTree) = tree1'fn tree'globalSumIn
                        (tree2'localSumOut, tree2'newTree) = tree2'fn tree'globalSumIn
                in (tree1'localSumOut + tree2'localSumOut,
                                Node tree1'newTree tree2'newTree)
```

The first phase of the translation is to simply rewrite the attribute definitions in the syntax of the functional language. Then the new definitions are inserted into the body of a function that will handle the 'plumbing' overhead of passing the attribute values between nodes during evaluation. The function takes the symbol and a tuple of the inherited attributes as arguments, and returns the synthesised attributes as a further tuple. For example, the $_{\text{Node}}$ production might be placed in the following shell:

```
eval (Node tree1 tree2) tree'globalSumIn =

        (tree'localSumOut, tree'newTree)

        where

        (tree1'localSumOut, tree1'newTree) = eval tree1 tree1'globalSumIn

        (tree2'localSumOut, tree2'newTree) = eval tree2 tree2'globalSumIn

        ... attribute definitions ...
```

This simple transformation makes the derivation of a single pass version of the *summedTree* function in section 3 from the attribute grammar definition straight forward:

```
summedTree tree = tree'newTree
      where
      (tree'localSumOut, newTree)    = eval tree tree'globalSumIn
      tree'globalSumIn               = tree'localSumOut
      eval (Leaf val) tree'globalSumIn
            = (tree'localSumOut, tree'newTree)
            where
            tree'localSumOut         = val
            tree'newTree             = Leaf tree'globalSumIn
      eval (Node tree1 tree2) tree'globalSumIn
            = (tree'localSumOut, tree'newTree)
            where
            tree'localSumOut         = tree1'localSumOut
                                           + tree2'localSumOut
            tree'newTree             = Node tree1'newTree tree2'newTree
            tree1'globalSumIn        = tree'globalSumIn
            tree2'globalSumIn        = tree'globalSumIn
            (tree1'localSumOut, tree1'newTree)
                                     = eval tree1 tree1'globalSumIn
            (tree2'localSumOut, tree2'newTree)
                                     = eval tree2 tree2'globalSumIn
```

Further simple rewrites of the program can then be performed to remove the unnecessary bindings of variables.

In addition to this transformation, Johnsson also presents an an extension to the lazy functional language LML. By providing a new construct, `case rec`, attribute grammars may be expressed directly as LML expressions. During compilation, the attribute grammar is compiled into native LML, in much the same way that pattern matching and list comprehension are dealt with in other languages.

Thus the `case rec` construct provides a means by which a traditionally many pass algorithm can be written as an efficient single pass one, whilst relieving the programmer of the task of performing both tedious attribute grammar transformations and difficult 'eureka' steps. Since the source code remains in the form of an attribute grammar, it should be more straight forward to prove correct, understand and maintain. It is in this spirit that an attribute grammar evaluator for Haskell has been developed.

5. Designing an Attribute Grammar Evaluator

The attribute grammar evaluator presented here takes the form of a pre-processor, before an existing Haskell compilation route. A script attribute grammar definitions is processed to form a native Haskell module.

The evaluator is designed to be as general as is practical. Not only should it be able to evaluate an arbitrary attribute grammar, it should also be able to use as many appropriate Haskell constructs as possible. To this end, the design has a number of useful features.

- Non-attribute values can be defined in production rules, and shared between the evaluation of the attributes.
- All variables in scope at the top-level of the attribute grammar definition are accessible inside all attribute bindings in the production rules. These include both function definitions local to the attribute grammar and parameters to the top-level function.
- The patterns representing production rules may be arbitrary Haskell patterns, allowing the use of tuples and the *cons* (:) operator in addition to user defined data constructors.
- The arguments to the top-level attribute grammar are explicitly specified and can be arbitrary Haskell patterns, allowing the programmer to specify a single clause of a top level function with an attribute grammar. Any other clauses could be defined either in native Haskell, or with further attribute grammar specifications if desired.

5.1 Attribute Sorts

In order that non-terminal symbols can be easily assigned attributes by the programmer the evaluator uses a concept taken from Johnsson's `case rec` statements. An *attribute sort* is a set of inherited and synthesised attributes that are to be associated with a variable and is explicitly defined prior to the production rule declarations. By annotating a variable in a production rule with the name of the attribute sort that is to be associated with it, the programmer instructs the attribute grammar processor to arrange for the synthesised attributes of the symbol to be evaluated. By allowing more than one attribute sort to be defined, and making the annotation of all non terminal symbols, including the left hand side, in a production mandatory, the use of attribute sorts provides other advantages:

- One is able to define production rules for more than one Haskell data type in an attribute grammar specification.
- The attributes in a production rule are restricted to those of interest in cases when a particular Haskell data type appears in more than one context in the grammar.

In the *summedTree* example in section 3, one might define an attribute sort, *Tree*. This would have the inherited attribute *globalSumIn* and the synthesised attributes *localSumOut* and *newTree*. In the production rules, the variables *tree*, $tree_1$ and $tree_2$ would all be annotated as having sort *Tree*.

It is important that one should not confuse attribute sort with Haskell type. The name of an attribute sort may, or may not, correspond to a data type. It is possible that a single data type may have more that one attribute sort associated with it, in different contexts, within a single grammar. However, it is normal that an individual attribute sort is associated with a single, possibly parameterised, Haskell type. This avoids typing problems in the output program.

5.2 Making Use of the Evaluator Practical

A primary design consideration of the attribute grammar evaluator has been ease of use. A programmer familiar with attribute grammars and Haskell should be able to use the system with the minimal of effort.

- Attribute grammar specifications form Haskell functions that are defined at the top level within a module. Function definitions within a module can be an arbitrary mix of attribute grammar specifications and native Haskell.
- The interface between an attribute grammar derived function and the native Haskell code is totally user defined: the function name, argument pattern and result expression being explicitly and intuitively specified.
- Each attribute grammar specification produces only one Haskell function: all definitions within the attribute grammar definition are placed in a single `where` block. This ensures the internal workings of the attribute grammar are hidden from the user program, enabling the implementation of the attribute grammar to be changed without effecting the rest of the program.
- Boundaries between attribute grammar and native Haskell are clearly marked with begin and end tokens.
- In order to make the Haskell programmer feel at home, the syntax of the attribute grammar looks like Haskell, as far as possible. The patterns used to match productions in the grammar are arbitrary Haskell *as* (@) patterns, and the attribute bindings are to almost native Haskell expressions.

In addition to providing an easy language, the evaluator aims to allow the elimination of compilation errors to be as simple as possible. However, it is also designed so as not to require a full Haskell parser: it is only necessary to examine the relevant parts of the source text and naively copy the native Haskell text to the output program. Whilst this makes the implementation of the evaluator quite straight forward, it means that any errors in the Haskell text will not be detected until the output is fed into the compiler. In order that the errors then reported are easily related to the source text, all output text occurs at the same line number as the source text that generated it. This means that:

- the native Haskell functions appear in exactly the same form, on the same line in the output file, as in the input;
- Haskell text generated by the evaluator occurs on the line that the source text caused it to be generated;
- all comments and blank lines are retained in the output text.

Some errors will be detected by the attribute grammar pre-processor and many of the remaining ones can be quickly spotted in the attribute grammar source, from the error and line number generated by the compiler. However, resolving some errors may require examination of the output Haskell text, in conjunction with the attribute grammar definition. By retaining the structure and the comments of the input text, the evaluator makes this process relatively painless.

6. A Basic Evaluator

Bearing the above design decisions in mind, we can now present a basic attribute grammar evaluator. As already stated, the evaluator takes the form of a pre-processor that reads the Haskell module containing the attribute grammar definition and produces a second, equivalent module written in native Haskell. There may be any number of attribute grammar specifications within the module provided they do not occur within function definitions, or within other attribute grammar definitions. Each attribute grammar is defined in a number of sections. These specify, in turn:

- the name of the Haskell function to be derived;
- the names of each attribute sort, the names of their attributes, and whether the each attribute is inherited or synthesised;
- the arguments and result of the Haskell function that is to be derived from this grammar, and the bindings for the attributes at this level;
- the productions rules for each case of each attribute sort, including the attribute bindings.

The various sections of the specification are separated by *directive* lines. These all start with a '*%*' character, which *mus*t be the first character on the line, and a keyword:

- `%begin` denotes the end of native Haskell, and the start of an attribute grammar definition block;
- `%top` defines the arguments to the derived function;
- `%result` gives the result of the derived function, and denotes the start of the attribute definitions and production rules; and
- `%end` marks the end of the attribute grammar, and a return to native Haskell.

6.1 Defining the Top Level of the Derived Function

The name of the Haskell function being derived is simply declared as part of the first line of the attribute grammar definition. For example,

```
%begin summedTree
```

declares a new attribute grammar definition defining the Haskell function `summedTree`. The arguments of the function, and the result expression are also defined in a similar way. For the *summedTree* example in section 3 these are defined by:

```
%top tree

%result tree'newTree
```

The directive `%top` may be followed by any number of arguments, which may arbitrary Haskell patterns, provided they occur on a single line. Similarly, the result expression may be of any type and complexity, provided it occurs on a single line.

The last line of the attribute grammar is always an `%end` directive. This serves no other purpose than to provide a clear break between the end of the attribute grammar specification and the start of native Haskell, for both the evaluator and the programmer. If always has the following form:

```
%end
```

The above `%begin`, `%top`, `%result` and `%end` lines generate the following Haskell in the output file. The `where` clause will eventually contain the top level attribute bindings and all the rule definitions for the attribute grammar.

```
summedTree tree =

      tree'newTree where { ... }
```

6.2 Declaring Attribute Sorts

Attribute sort declarations occur between the `%begin` and `%top` lines, that is before any attributes are used. A series of lines declare inherited and synthesised attributes separately. To declare a sort, `Tree` with an inherited attribute `globalSumIn` and two synthesised attributes `localSumOut` and `newTree` one might use:

```
Tree inherits      globalSumIn

Tree synths        localSumOut newTree
```

6.3 Adding Production Rules

The production rules appear after the definition of the top-level function, that is between the `%result` and `%end` lines. Each starts with a production which is written as a Haskel *as* (@) pattern. For example, the production for a *Node* in the example grammar for *summedTree* in section 3 would be written as:

```
tree@(Node tree1 tree2)
```

This states that the identifier `tree` produces a Node containing the identifiers `tree1` and `tree2`. We have not, yet, named the attribute sorts associated with these identifiers, if any.

Each rule pattern produces a Haskell function clause inside the top-level where block. Assuming for the moment that the identifier `tree` had attribute sort `Tree`, the above production would create the Haskell code,

```
eval'Tree tree@(Node tree1 tree2) tree'globalSumIn =

      (tree'localSumOut,tree'newTree)

      where { ... } ;
```

This, in turn, includes a second `where` block into which the attribute bindings will be placed.

6.4 Attribute Bindings

The body of the top level and each production rule definition is split into two subsections. The first lists the identifiers at that level that represent a non-terminal symbol in the grammar, and so require evaluation of their attributes. Each such identifier is associated with an attribute sort. For example, to declare that the identifier `tree` is of sort `Tree` one might use:

```
Tree : tree
```

The name of the sort should appear, starting in the first column of the line, and any number of identifiers may occur on the right of the colon, provided they are separated only by white space and appear on one line.

The behaviour of the evaluator with respect to a declaration of this form depends on where the listed identifiers originate from. If the identifier is the left hand side of the production rule, the sort name declared is used to select the name of the function of which the current production will form a clause.

Any other identifiers occurring in these declarations cause code to be planted which arranges for the synthesised attributes of these symbols to be evaluated. By the nature of attribute grammars, inherited attributes of these symbols must be defined by the production rule. For example, the sort declaration

```
Tree : tree1 tree2
```

might produce the following code within the `where` clause of the function body being locally defined:

```
(tree1'localSumOut,tree1'newTree)
      = eval'Tree tree1 (tree1'globalSumIn) ;
(tree2'localSumOut,tree2'newTree)
      = eval'Tree tree2 (tree2'globalSumIn) ;
```

The second part of the top-level or production rule definition defines all attributes that must be defined locally: that is any synthesised attributes of the production symbol, and any inherited attributes of child symbols. Attribute definitions take the form of an attribute name and Haskell text to form the right hand side of the definition, contained in braces. For example,

```
tree'newTree { = Leaf tree'globalSumIn }
```

binds the `newTree` attribute of `tree` with the given Haskell expression. The text within the the braces is copied to the output file, almost, verbatim and is used to form the binding:

```
tree'newTree = Leaf tree'globalSumIn ;
```

A few points should be noted, however, when writing the attribute binding text:

- The attribute definition may take place over a number of lines provided each line, including the first, starts with a white space character.
- Nested braces are permitted within the Haskell text contained in the outer ones, but single braces must be preceded by an escape, '\' character; even if they occur within quotes.
- With '\' being treated as an escape character, the string '*c*' within the braces, where '*c*' is any character, is replaced simply by '*c*'. This means that a single '\' character requires that '\\' be entered in the input text. This includes lambda abstractions and escape characters in the output Haskell module.
- It is possible to include the definition of a number of non-attribute values as part of an attribute definition. These bindings occur within the braces and are separated by ';' characters, as usual within a *free-form* `where` block. The attribute bindings will appear in the output program at the same level, within the production rule, and so additional values declared as part of one attribute binding can be also used in any of the others.

6.5 A Note About 1:1 Attribute Bindings

It is quite common that an attribute may be passed from one symbol to the next without any change. The only way to express this in this syntax is to write an attribute definition of the form:

```
tree1'globalSumIn { = tree'globalSumIn }
```

Unfortunately this binding is translated directly into a Haskell binding so one often gets code of the form:

```
eval'Tree tree@(Node tree1 tree2) tree'globalSumIn =
      ( ... )
      where {
            ( ... )              = eval'Tree tree1 tree1'globalSumIn;
            ...
```

```
tree1'globalSumIn = tree'globalSumIn;
...
}
```

Compiled naively, this code would be inefficient as the binding to `tree1'globalSumIn` is unnecessary; all occurrences of it could be replaced with `tree'globalSumIn`. The attribute grammar processor makes no attempt to optimise such situations as it is hoped that the Haskell compiler should be able to do this for us.

7. Extending the Basic Evaluator

The basic attribute grammar evaluator described above is adequate for general problems, but there is scope for improvement in ease of use. For example, a large proportion of the attribute bindings in a production rule can be trivial, passing attributes between parent and child symbols with little or no transformation. It would relieve the programmer from yet more overhead if these bindings could be made automatically, as defaults when no other local binding is made.

7.1 Default Attribute Bindings

Consider the propagation of inherited attributes. It is quite common that an inherited attribute be passed directly from the parent to one or more of its child symbols exactly as it is. This is the case for the *globalSumIn* attribute in the example in section 3, and perhaps for an 'environment' attribute within an expression tree: *application* nodes may simply pass the environment down; whilst *letrec* nodes may augment the environment with new information, and pass this on the children. By having a default 'just pass it on' binding for inherited attributes, the programmer only has to specify bindings for the attribute explicitly when a binding other than the default is required.

Such a scheme is quite simple to implement and is included in the attribute grammar evaluator. Each inherited attribute of each child in the production is examined in turn. If the programmer has not supplied a binding for an inherited attribute, and the parent symbol has an inherited attribute of the same name, the child's attribute is bound to the parent's.

It is a harder problem to consider how a default binding for a synthesised attribute might be constructed. In a similar way to inherited attributes, synthesised attributes are often a combination of the synthesised attributes of the same name, of the child symbols. However, it is not clear what would be a suitable default combination. At present the evaluator does not support any form of default synthesised attribute bindings, but there are plans to do so (see section 10.3).

7.2 'Threaded' Attributes

Default bindings for synthesised attributes may be difficult to define in general, but there is a further set of useful default bindings that have been implemented in the current evaluator. For certain classes of problems, it might be desirable to pass a single attribute to all the symbols in the tree sequentially. An example of this might be a name supply in a compiler. Each non-terminal symbol inherits a name supply, and passes it to its first child. The child then returns it, after perhaps altering it, as a synthesised attribute and is is passed to the next child and so on. In this way the attribute is *threaded* through all the non-terminals in the tree.

The attribute grammar evaluator supports automatic routing of such threaded attributes. The default bindings route the attribute through all children that *thread* an attribute of the same name. The order in which the attribute visits the children is determined by the order that the children and named in the sort declarations

for the production rule.

In order for this to work, a new attribute type is used: a threaded attribute. This is, in fact, is a combination of an inherited and synthesised attribute. For example, the sort attribute declaration,

```
Expr threads nameSupply
```

would associate the inherited attribute `nameSupply'In` and the synthesised attribute `nameSupply'Out` to the attribute sort `Expr`. By using such a declaration, the programmer has not only declared the two attributes, but also informed the evaluator with the information that they are associated and what their relationship is. It is then possible for the automatic routing to take place, for example if the production rule,

```
e@Ap e1 e2

Expr : e e1 e2
```

was not followed by any definitions for the *nameSupply* threaded attribute, the following bindings would be generated automatically, in the `where` block:

```
e1'nameSupply'In  = e'nameSupply'In ;

e2'nameSupply'In  = e1'nameSupply'Out ;

e'nameSupply'Out  = e2'nameSupply'Out ;
```

8. The Advantages of the Evaluator

Whilst being developed in the spirit of Johnsson's `case rec` expressions, the attribute grammar evaluator presented here has a number differences. Although some of these make the attribute grammar slightly less convenient to use, they are on the whole outweighed by the advantages of the system.

8.1 Separate Compilation Phase

Johnsson's `case rec` expressions were proposed as a new construct in the LML language. This would obviously require that to use it one would need a compiler supporting the new construct. Although many languages support 'syntactic sugar' such as list comprehension and pattern matching, attribute grammars may only be suitable for a small set of problems so the presence of a syntactic sugar for them may be seen as confusing to a new programmer.

Taking the form of a pre-processing phase in the compilation of a program, the attribute grammar evaluator remains separate from the main Haskell language. You only need to be concerned with it if you are actually using it. Meanwhile, the evaluator should be as powerful and almost as easy to use as Johnsson's. In addition, the resulting code should also be as efficient, as the evaluator carries out a similar program transformation to the `case rec` expression.

Johnsson speculated that the presence of the `case rec` in the language might both make programmers aware of attribute grammars and encourage them to use them in their programs. Although a pre-processor phase is less convenient than having the evaluator built in to the compiler, the power of attribute grammars should outweigh the disadvantages.

8.2 Intuitive Notation

The `case rec` expression has a rather obscure method of defining the top level attribute. This was achieved by a combination of a 'magic' production rule and associated attribute sort, which were only ever applied to the root symbol in the abstract syntax tree. This production rule defines the inherited attributes at

the top level, whilst the result of the `case rec` expression is given as being the synthesised attributes of the attribute sort. Although this may be seen as being mathematically elegant, it is not easy to read.

In contrast, the evaluator presented in this paper describes the name, arguments and result of the top level function clearly and explicitly, through the `%begin`, `%top` and `%result` clauses. This makes the code easier to understand, and the interface between functions produced by the attribute grammars and those written by the programmer directly is more obvious.

Whilst making the interface between the functions derived from the attribute grammar and those written in native Haskell more natural, the evaluator retains generality. Not only can one express arbitrary attribute grammars, but one is able to chose from many Haskell constructs when applicable. This includes the ability to locally define non-attribute values, including functions, and using any Haskell *as* pattern to select production rules.

8.3 Extensions to the Basic Evaluator

By providing default bindings for attributes, following simple rules, the evaluator is able to relieve the programmer of the overhead of propagating some attributes throughout the grammar definition. This allows the programmer to concentrate on the points at which values are significant.

The attribute grammar syntax should support a number of improvements to the system, whilst retaining source code compatibility. Unfortunately, a major restriction is that attribute grammar definitions must occur at the top level within a Haskell module. This rules out the possibility of defining functions with attribute grammars locally within another function. It is hoped that this should not prove to be a serious restriction.

9. Performance of Attribute Grammar Derived Programs

Implementing a program by means of an attribute grammar evaluator, such as the one described here, may prove an efficient option in terms of programmer productivity in producing a single pass algorithm. However, it is not necessarily true that such a single pass algorithm is the most efficient, in terms of store and processing time required to execute it.

The functions derived from attribute grammar specifications contain a considerable amount of code building tuples of attribute values, passing them to the next level of computation and then unpacking the tuples to retrieve the value. This may, at first, sound very expensive, but in this sort of code is common in many functional programs. In addition, a multiple pass algorithm may require the construction, and subsequent dismemberment, of complex, intermediate data structures. The attribute grammar removes the need for this, at the expense of requiring the construction of the tuples.

At present, only a small number of attribute grammar derived programs have been compared to their naive, multiple pass equivalents. Since it is difficult to assess what constitutes a ‘naive’ program such that a ‘fair trial’ can be set up, only small trails have been conducted. Unfortunately, the results of these have been somewhat disappointing.

Consider, for example, the two versions of the *summedTree* function given in sections 3 and 4. A trial was conducted to evaluate and print the results of the *summedTree* function applied to a symmetrical tree with 2^{10} *Leaf* nodes. The two programs where identical, excepting that the *summedTree* function was the naive, two pass algorithm in the first, and the attribute grammar derived version in the second. The input tree was being lazily generated by a tree-growing function.

Both programs produced the correct answer, with the attribute grammar version taking marginally longer. However, the naive version used only $^1/_8$ of the heap space used by the former, in terms of maximum store

occupancy. A reason for this can be found if one considers the behaviour of the printing mechanism, which visits each node in the tree in a depth first manner, discarding nodes as they are passed.

For the naive program, the nodes can be created for printing, almost immediately, by the `mkNew` function. When the contents of the first `Leaf` node are required, the evaluation of the entire input tree is invoked, as the result of `calcSum tree` is demanded. The result of this behaviour is that the largest data structure that is ever stored is the input tree.

In the case of the attribute grammar derived version of `summedTree`, however, the story is a little different. Once again, the printing mechanism chases down the left hand spine of the input tree, and reaches a `Leaf` node. In this case, the evaluation of the contents of this node requires the evaluation of the attribute `globalSumIn`, which in turn requires the evaluation all the `localSumOut` attributes. Since the evaluation of one attribute at a node means that the expressions for each of the remaining one must be constructed and, in this case, retained, this results in the large amount of heap being required.

This might appear an overwhelming case against the use of attribute grammars in the development of lazy functional programs. One can make, however, a two observations in their favour. Firstly, when writing function programs one can often trade execution performance for ease of implementation and elegance of code: this is not something unique to attribute grammars. Secondly, this is a simple problem: the naive approach has only two passes and does not require an intermediate data structure. One might expect that a more complex problem, such as a lambda lifter, would perform better than a many pass one. It is also worth noting that the attribute grammar `summedTree` function did not appear to require significantly more execution time. A similar test comparing functions renaming an expression tree suggested that for that problem, the performance of the naive and attribute grammar versions was almost identical in terms of both cell claims and reductions performed. The attribute grammar for the expression renaming is given in appendix 2.

10. Future Extensions to the Attribute Grammar Processor

The attribute grammar evaluator is still in its infancy. It has only just begun use in applications other than the test programs. Whilst it is already a useful tool in writing programs of a suitable class, there are a number of improvements that can be made.

10.1 Extended Error Checking

The useability of the attribute grammar could be greatly increased by an improvement on the number of checks and error reports made on the source text. Many errors may not be detected until the Haskell compiler is invoked. At present the processor aids the programmer in sorting out such errors by maintaining the line numbers and comments in the output file, and occasionally inserting a comment of its own. It would be preferable to trap more errors during attribute grammar pre-processing to give the programmer more meaningful error messages.

Some of these error checks would be quite simple to implement. For example, in order that a 1:1 correspondence between the lines of the input and output files be maintained, the function clauses generated by the production rules appear in the same order in the output file as they were declared in the input file. It is therefore necessary to group the production rules for each attribute sort together, to ensure the Haskell function definitions remain separate. If this is not done, an error will be thrown back from the compiler, perhaps complaining that a function has already been defined. It would be quite straight forward to implement an error check that detected such situations and warn the programmer.

Some of the more complex error detection mechanisms that might be advantageous may require a large

investment in the form of a full parser for the expressions on the right hand side of the attribute definitions. This extension is not being considered for the moment as it is unclear that the benefits would make the investment worthwhile.

10.2 Increased Convenience and Generality

The attribute grammar pre-processor scans the input file on a line by line basis. Although easier to implement, this tends to restrict the input at some stages to a single line. The attribute sort declarations in the preamble and in the production rules suffer from this to a degree, but one is able to specify additional attributes by starting again on the next line. For example, the code

```
Tree synths newTree

Tree synths localSumOut
```

is valid in the preamble between the `%begin` and `%top` directives. However, it is not currently possible to specify the text following a `%top` or `%result` directive on additional lines. A future extension should resolve this situation.

10.3 Further Default Attribute Bindings

The evaluator is able to relieve the programmer of the task of defining the attribute bindings for inherited and threaded attributes in some cases by providing defaults. There is scope for more advanced information to be gathered that enable further declarations to be derived.

It would be useful to be able to specify a default binding for synthesised attributes. A simple scheme for this would be to synthesise the value returned from the first child. However such a strategy may prove to be not particularly useful in general, if there is more than one child. An alternative strategy might be to allow the specification of a binary function that will allow the combination of two synthesised attributes from child symbols. It is likely that such a function would have to be both commutative and associative so that the child attributes can be combined in an arbitrary order. Even so, this strategy would be useful when the synthesised attributes are some sum or set union of the attributes of child symbols.

If one wishes to use an attribute grammar to describe an operation on a data structure which returns a second structure of the same type, one would often use a synthesised attribute to propagate the new structure back to the root. In this situation is is common for the bindings for this attribute to be a simple constructor application of structures returned from the children. For example, in the code,

```
tree@(Node left right)

Tree: tree left right

        tree'newTree = Node left'newTree right'newTree
```

the `newTree` attribute simple returns a similar constructor to the one in the production, but with the non-terminal symbols replaced with their respective `newTree` attributes. It might be possible to implement a third rule for deriving default synthesised attribute bindings that make use of this observation.

Since they are all useful in different situations, it would be desirable to be able to use a combination of all three default synthesised attribute bindings in a given grammar specification. The attribute grammar evaluator will probably be enhanced to permit this.

Whilst default attribute bindings save the programmer from an amount of unnecessary detail, they might cause confusion if they are applied unexpectedly. It would therefore be desirable to permit the programmer to specify whether a default binding was required for individual attributes in the initial sort declaration preamble.

If we observe that this is already necessary to implement a choice of default synthesised bindings under the above scheme, and that threaded attributes can always be expected to have default bindings, it remains only to allow the specification that some inherited attributes may not support a default binding.

10.4 Automatic Derivations Requiring More Complex Analysis

In addition to providing default bindings for attributes, it may also be possible for the attribute grammar pre-processor to derive the list of attributes for each attribute sort, or even the sort of a non-terminal symbol from type information. Unfortunately, these are a little more complicated to implement, in particular the latter. Also, it is unclear that such sophistication would prove advantageous to the programmer if implemented, as:

- the system becomes unable to spot invalid attribute names, for example a misspelling would simple create a new attribute;
- if default bindings for synthesised attributes are to be implemented the programmer would have to specify these and the attribute names anyway;
- if sort name is to be derived from type information, one looses the ability to apply a number of sorts over the same Haskell type.

10.5 Command Line Interface

At present, the pre-processor acts as a simple filter of standard input to standard output. It is an obvious extension to allow the use of command line options to specify alternative input and output files. In addition it would be desirable to permit a number of processing options, such as printing warnings when default bindings are applied for example.

11. Conclusion

A useful aid to the implementation of certain classes problems in Haskell has been presented. A number of example programs have been implemented demonstrating that it is general enough to express many complex problems quite succinctly. As larger programs are written through the evaluator, many possible extensions have been conceived and in some cases implemented. Although attribute grammars may be utilised to derive single pass algorithms, they may not necessarily be more efficient than equivalent multiple pass ones. However, the attribute grammar specification can often be much easier to derive, understand and prove correct than other forms of efficient executable code, for many problems.

Acknowledgement

I would like to thank Dr D. Lester and J. Seward for their feedback on a number of points in the design of this evaluator.

References

[Bir84] R. S. Bird, Using Circular Programs to Eliminate Multiple Traversals of Data, Acta Informatica, 21:239-250, 1984

[Fro92] R. A. Frost, Constructing Programs as Executable Attribute Grammars, The Computing Journal, Vol.35 4:376-389, 1992

[Gro91] J. Grosch, Ag - An Attribute Evaluator Generator, GMD Forschungsstelle an der Universität Karlsruhe 16, 1991

[Has91] P. Hudak, S. Peyton Jones, P. Wadler (editors), Report on the Programming Language Haskell,

Version 1.1, 1991

[Joh85] T. Johnsson, Lambda Lifting: Transforming Programs to Recursive Equations, Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture, Nancy, France, 1985

[Joh87] T. Johnsson, Attribute Grammars as a Functional Programming Paradigm, Springer Lecture Notes in Computer Science, 274:155-173, 1987

[Knu68] D. E. Knuth, Semantics of Context-Free Languages, Mathematical Systems Theory, 2:127-145, 1968

[Tar88] J. Tarhio, Attribute Grammars for One-Pass Compilation, Department of Computer Science, University of Helsinki, A-1988-11

Appendix I : Syntax of the Attribute Grammar Evaluator Input

The attribute grammar definition is layout sensitive, and line based. In the following, rather informal, syntax description, line breaks are not allowed, unless explicitly stated otherwise. In addition the '<' symbol has been given special meaning, indicating a new line must immediately precede the next symbol. The definition is fairly informal, however study of it and the examples in appendix 2 should make the permissible input clear.

AgModule = [*ModuleDefinitionBlock*]

ModuleDefinitionBlock = *HaskellDefinitionBlock*
| *AGDefinitionBlock*

AGDefinitionBlock = *BeginDeclaration* [*SortDeclaration*] *TopDeclaration ResultDeclaration* [*RuleDeclaration*] *EndDeclaration*

BeginDeclaration = < `%begin` *Identifier*

SortDeclaraion = < *SortName* `inherits` [*Identifier*]
| < *SortName* `synths` [*Identifier*]
| < *SortName* `threads` [*Identifier*]

TopDeclaration = < `%top` [*HaskellFunctionArg*]

ResultDeclaration = < `%result` *SimpleHaskellExpr UserDefinitions*

RuleDeclaration = < *Identifier* `@` *HaskellFunctionArg UserDefintions*

EndDeclaration = < `%end`

UserDefinitions = [*SortsOfIdentifiers*] [*AttributeBinding*]

AttributeBinding = *AttributeName* { *HaskellExpressionRHS*
[**;** *HaskellFreeFormDefinition*] }
(*AttributeBinding* may occur over a number of lines, provided each line, including the first in starts with white space. In addition, single braces and '\' characters in the *HaskellExpressionRHS* and *HaskellFreeFormDefinition* must be preceded with an escape, '\', character.)

SortsOfIdentifiers = < *SortName* **:** [*Identifier*]

HaskellDefinitionBlock = arbitrary native Haskell top-level definitions

HaskellExpressionRHS = arbitrary text forming the right hand side of a 'free form' Haskell binding

HaskellFreeFormDefinition = arbitrary 'free form' Haskell definition

HaskellFunctionArg = an arbitrary pattern suitable for a function argument

SimpleHaskellExpr = a Haskell expression that is no more complicated than a tuple

Identifier = identifier beginning with lower-case letter

SortName = identifier beginning with upper-case letter

In addition, the attribute grammar definition block (the lines between `%begin` and `%end`) may also contain blank lines, or `--` comment lines at any point. (*NB* the pre-processor does not know about `{-...-}` comments, so these should not be used within this block)

Appendix II : Example Executable Attribute Grammar Specifications

This appendix contains a number of example programs written as attribute grammars, in the syntax of the evaluator. When processed, each program forms a script suitable for compilation under *Gofer*, a Haskell-like language interpreter written by Mark Jones, and an example of function in use is given in a function called `test`.

1. The simple 'summedTree' example

Name: summedTree

Origin: Example from section 3, similar to an example to label tree leaves with minimum value, in [Joh87].

Purpose: Produces a tree, the same shape as the input tree, except that each leaf contains the sum of all the leaves in the original tree.

Features: Shows simple use of attribute grammar in the derivation of a single pass program, for a traditionally two-pass operation.

```
data BinTree = Leaf Int
             | Node BinTree BinTree
%begin       summedTree
Tree inherits      globalSumIn
Tree synths        localSumOut newTree
%top         tree
%result      tree'newTree
Tree : tree
      tree'globalSumIn  { = tree'localSumOut }
tree@(Leaf val)
Tree : tree
```

```
        tree'localSumOut   { = val }
        tree'newTree       { = Leaf tree'globalSumIn }
tree@(Node tree1 tree2)
Tree : tree tree1 tree2
        tree1'globalSumIn { = tree'globalSumIn }
        tree2'globalSumIn { = tree'globalSumIn }
        tree'localSumOut   { = tree1'localSumOut
                                 + tree2'localSumOut }
        tree'newTree       { = Node tree1'newTree
                                 tree2'newTree }
%end
test = summedTree
        (Node (Leaf 5) (Node (Leaf 6)
                (Node (Leaf 3) (Node (Node (Leaf 2) (Leaf 1))
                (Node (Node (Leaf 8) (Leaf 7)) (Leaf 4))))))
```

2. An extension to the 'summedTree' example

Name: summedTree II

Origin: An extension of summedTree.

Purpose: Takes a tree of containing values of an arbitrary type, and produces a tree the same shape, but with each leaf value being the *sum* of the input leaf values. *sum* is a function of the type *a->a->a*, where *a* is the type of the leaves in the tree.

Features: Demonstrates how a higher-order function may be generated through the attribute grammar evaluator, and how additional arguments to the derived function can be specified. Also demonstrates the use of default, inherited attribute bindings.

```
data BinTree a      = Leaf a
                    | Node (BinTree a) (BinTree a)
%begin      summedTree
Tree inherits       globalSumIn
Tree synths         localSumOut newTree
%top        sumFunction tree
%result     tree'newTree
Tree : tree
```

```
      tree'globalSumIn   { = tree'localSumOut }
tree@(Leaf val)
Tree : tree
      tree'localSumOut   { = val }
      tree'newTree       { = Leaf tree'globalSumIn }
tree@(Node tree1 tree2)
Tree : tree tree1 tree2
      tree'localSumOut   { = tree1'localSumOut
                         `sumFunction` tree2'localSumOut }
      tree'newTree       { = Node tree1'newTree tree2'newTree }
%end
test = summedTree (+)
      (Node (Leaf 5) (Node (Leaf 6)
            (Node (Leaf 3) (Node (Node (Leaf 2) (Leaf 1))
            (Node (Node (Leaf 8) (Leaf 7)) (Leaf 4))))))
```

3. Expression tree renaming pass

Name: rename

Origin: Translated from example in [Joh87].

Purpose: Takes an abstract syntax tree for a small language and performs scopeing analysis on the program, producing a new, equivalent program in which the identifiers are unique.

Features: A more complex example featuring the use of 'threaded' attributes to automatically pass a name supply around the syntax tree.

```
data Expr    = Id Var
             | Ap Expr Expr
             | Letrec [Bind] Expr
type Bind    = (Var,Expr)
type Var     = Int
%begin       rename
Expr inherits      map
Expr synths        ren
Expr threads       ns
```

```
Defn inherits      map
Defn synths        ren newmap
Defn threads       ns
%top         ex
%result      ex'ren
Expr : ex
       ex'map              { = [] }
       ex'ns'In            { = 0 }
ex@(Ap e1 e2)
Expr : ex e1 e2
       ex'ren              { = Ap e1'ren e2'ren }
       e1'map              { = ex'map }
       e2'map              { = ex'map }
ex@(Id id)
Expr : ex
       ex'ren              { = Id newid;
                           newid = lookup id ex'map }
ex@(Letrec df e1)
Defn : df
Expr : ex e1
       ex'ren              { = Letrec df'ren e1'ren }
       e1'map              { = extendedmap }
       df'map              { = extendedmap;
                           extendedmap = ex'map ++ df'newmap }
df@( (id,ex) : d1 )
Expr : ex
Defn : df d1
       df'ren              { = (newid,ex'ren):d1'ren;
       newid               { = df'ns'In }
       ex'map              { = df'map }
       d1'map              { = df'map }
       df'newmap           { = (id,newid):d1'newmap }
```

```
        ex'ns'In     { = df'ns'In+1 }
        d1'ns'In     { = ex'ns'Out }
        df'ns'Out    { = d1'ns'Out }
df@([])
Defn : df
        df'ren       { = [] }
        df'newmap    { = [] }
%end
lookup n []          = error "Item not found in lookup"
lookup n ((n1,n2):ns)
        | n == n1    = n2
        | otherwise  = lookup n ns
test = rename
        (Letrec [
        (7263,Id 74),
        (74, Ap (Letrec [(64,Id 7263)] (Ap (Id 64) (Id 5)))
                (Id 7263)),
                (5, Id 5)
        ]
        (Ap (Id 5) (Ap (Id 74) (Id 5)))
        )
```

4. Sorting the leaves of a tree

Name: sort

Origin: Translated from example in [Joh87].

Purpose: Takes a tree of integers and returns a tree in which all the values in the leaves are in sequence when passed from left to right. The output tree is of the same shape as the input tree.

Features: Demonstrates how a program can be sometimes be derived that is an order of magnitude of complexity simpler than the 'naive' approach, see [Joh87].

```
data Tree     = Tip Int
              | Fork Tree Tree
%begin        leafSort
Tree threads        tips sorted
Tree synths         result
%top          t
%result       t'result
Tree : t
       t'tips'In { = [] } t'sorted'In { = sort t'tips'Out }
t@(Tip i)
Tree : t
       t'tips'Out   { = i : t'tips'In }
       t'sorted'Out { = tail t'sorted'In }
       t'result     { = Tip (head t'sorted'In) }
t@(Fork l r)
Tree : t l r
       t'result{ = Fork l'result r'result }
%end
test = leafSort
       (Fork (Fork (Tip 65) (Fork
              (Fork (Tip 54) (Tip 4)) (Tip 87))) (Tip 1))
```

5. Johnsson's lambda lifter

Name: lambda

Origin: Translated from specification in [Joh85].

Purpose: Performs lambda lifting on a small language.

Features: Shows how a complex, useful program can be expressed in the attribute grammar syntax.

```
data Expr     = Letrec [Defn] Expr
              | Appl Id [Expr]
data Defn     = Var Id Expr
              | Fun Id [Id] Expr
type Id= [Char]
-------------------------------
-- Lambda lifting definition.
--
%begin lambdaLift
-------------------------------
--  Attribute sort definitions
-- there are three of these - expressions, argument lists (idenitifers)
-- and definitions lists. All three share the same inherited attributes
-- and three common synthesised ones. In addition, Defs have three
-- extra synthesied attributes to make the set equations easier.
--
Expr inherits       vars funs sol
Defs inherits       vars funs sol
Args inherits       vars funs sol
Expr synths         ids lifted expr
Defs synths         ids lifted expr dvars dfuns ss
Args synths         ids lifted expr
-------------------------------
-- Top level function definition
-- the top-level lambda lifting function takes an expression and returns
-- a new expression
--
%top          expr
%result       (Letrec expr'lifted expr'expr)
Expr : expr
       -- the vars, funs and sol sets are empty initially...
```

```
        expr'vars { = [] } expr'funs { = [] } expr'sol { = [] ;
        -- (some general functions defined as part of expr'sol...)
        -- locally defined utility functions for set equations
        xSet sol s z =
              itlist (\\f -> \\p -> setUnion (assocdef f sol []) p) s z;
        --
        solve []                  = [];
        solve ((f,s,e):[])        = [(f,s)];
        solve ((f,s,e):l) =
              let sol = solve (map
                    (\\t@(f1,s1,e1) -> case setElementOf f e1 of
                          True -> (f1, setUnion s s1, setUnion e e1)
                          False -> t
                    ) l)
              in (f, itlist
                    (\\x p -> setUnion (assocdef x sol []) p) e s) : sol
        --
        }
-------------------------------
-- Rules for Expressions
--
ex@(Letrec d e)
Expr : ex e
Defs : d
        ex'ids       { = d'ids `setUnion` e'ids }
        ex'lifted    { = d'lifted ++ e'lifted }
        ex'expr      { | d'expr == []     = e'expr
                     | otherwise          = Letrec d'expr e'expr }
        --
        d'vars       { = ex'vars `setUnion` d'dvars }
        d'funs       { = ex'funs `setUnion` d'dfuns }
```

```
      d'sol        { = solve (map (\\ (f,s) ->
                           (f, (xSet ex'sol (s `setIntersection` ex'funs)
                           (s `setIntersection` (ex'vars `setUnion`
                           d'dvars))), (s `setIntersection` d'dfuns) ))
                           d'ss ) ++ ex'sol }
       --
      e'vars { = d'vars } e'funs { = d'funs } e'sol { = d'sol }
ex@(Appl f a)
Expr : ex
Args : a
      ex'ids       { = [f] `setUnion` a'ids }
      ex'lifted    { = a'lifted }
      ex'expr      { = (let toExpr = map (\\i -> Appl i [])
                   in Appl f (toExpr (lookup f ex'sol) ++ a'expr) ) }
---------------------------------
-- Rules for arguement lists
--
ag@(e:a)
Expr : e
Args : ag a
      ag'ids       { = e'ids `setUnion` a'ids }
      ag'lifted    { = e'lifted ++ a'lifted }
      ag'expr      { = e'expr : a'expr }
ag@([])
Args : ag
      ag'ids { = [] } ag'lifted { = [] } ag'expr { = [] }
---------------------------------
-- Rules for definitions lists
--
df@((Var i e) : d)
Defs : df d
```

```
Expr : e
      df'ids       { = e'ids `setUnion` d'ids }
      df'lifted    { = e'lifted ++ d'lifted }
      df'expr      { = (Var i e'expr) : d'expr }
      df'dvars     { = [i] `setUnion` d'dvars }
      df'dfuns     { = d'dfuns }
      df'ss        { = d'ss }
df@((Fun f i e) : d)
Expr : e
Defs : df d
      df'ids       { = e'ids `setUnion` d'ids }
      df'lifted    { = (Fun f (lookup f df'sol ++ i) e'expr)
                   : (e'lifted ++ d'lifted) }
      df'expr { = d'expr } df'dvars { = d'dvars }
      df'dfuns     { = [f] `setUnion` d'dfuns }
      df'ss        { = (f,e'ids) : d'ss }
      --
      e'vars { = df'vars `setUnion` (setFromList i) }
df@([])
Defs : df
      df'ids { = [] } df'lifted { = [] } df'expr { = [] }
      df'dvars { = [] } df'dfuns { = [] } df'ss { = [] }
--------------------------------
-- End of lambda lifter definition
--
%end
--------------------------------
-- Utility functions
--
itlist f [] z       = z
itlist f (x:l) z    = f x (itlist f l z)
```

```
assocdef x [] def          = def
assocdef k ((x,y):l) def   | k == x = y
                           | otherwise = assocdef k l def
lookup i ((f,s):sol)       | f == i = s
                           | otherwise = lookup i sol
lookup i [] = []
------------------------
-- Set stuff
--
type Set a = [a] -- Ordered by the sort function
{- IMPLEMENTATION OF SETS OMMITED HERE -}
test = lambdaLift
       (Letrec [Var "i" (Appl "" []), Var "+" (Appl "" []) ]
       (Letrec [Fun "f" ["x"]
              (Appl "f" [Appl "+" [Appl "i" [],Appl "i" []] ] ) ]
        (Appl "f" [Appl "*" [Appl "i" [],Appl "i" []]]) ))
```