Constructing Programs as Executable Attribute Grammars

R. A. FROST
School of Computer Science, University of Windsor, Windsor, Ontario N9B 3P4. Canada

Attribute grammars provide a formal yet intuitive notation for specifying the static semantics of programming languages and consequently have been used in various compiler generation systems. Their use, however, need not be limited to this. With a little change in perspective, many programs may be regarded as interpreters and constructed as executable attribute grammars. The major advantage is that the resulting modular declarative structure facilitates various aspects of the software development process.

In this paper, we show how the attribute grammar programming paradigm can be readily supported by adding four combinators to the standard environment of a lazy functional programming language. We give examples of the use of these combinators and discuss the advantages that derive from integration of the attribute grammar and functional programming paradigms.

Received April 1989, revised December 1991

1. INTRODUCTION

Attribute grammars were introduced by Knuth[20,21] in 1968 as a notation for specifying the static semantics of programming languages. In 1971 Knuth[22] suggested that the attribute grammar formalism might lead to viable declarative programming languages, in which problems are solved in terms of relevant structures.

Since their introduction, a good deal of theory has been developed[7,12] and attribute grammars have been used in various language-processor generation systems.[6,8] However, use of the attribute grammar formalism as a general-purpose programming paradigm has received attention from only a few researchers.[9,14,17,19,30,32]

In this paper we show how the attribute grammar programming paradigm can be readily provided by adding four combinators (higher-order functions) to the standard environment of a lazy, functional programming language. We present a number of examples of the use of these combinators to illustrate various features of the style of programming that they support.

We discuss the advantages and disadvantages of a functional implementation of the attribute grammar paradigm. In particular, we consider the advantages that derive from lazy evaluation and from the higher-order nature of the host language.

We conclude with a brief overview of related work and suggestions for future research.

2. THE ATTRIBUTE GRAMMAR FORMALISM AND ITS USE IN SOFTWARE ENGINEERING

An *attribute grammar* is a context-free grammar, each production of which is augmented with a set of *semantic rules*. Each semantic rule states how the value of an *attribute* associated with a syntactic construct in the production is derived by applying a *semantic function* to values of attributes associated with other syntactic constructs in the production.

The set of attributes associated with a particular syntactic construct can be partitioned into two disjoint sets: *synthesised* attributes and *inherited* attributes. Each semantic rule associated with a production rule P either defines a synthesised attribute of the syntactic construct named on the left-hand side (lhs) of P or defines an inherited attribute of a syntactic construct on the right-hand side (rhs) of P. Synthesised attributes may be regarded as passing semantic data upwards towards the root of the derivation tree. Inherited attributes may be regarded as passing semantic data down the derivation tree.

An example of a simple attribute grammar, involving only synthesised attributes, is given in Fig. 1.

```
numb     ::= "one"
             VAL↑numb = VAL 1
           | etc

summ     ::= numb
             VAL↑summ = VAL↑numb
           | numb "plus" summ'
             VAL↑summ = VAL↑numb + VAL↑summ'

subtr    ::= numb "minus" numb'
             VAL↑subtr = VAL↑numb - VAL↑numb'

comp     ::=   subtr | summ

br_comp  ::= "(" comp ")"
             VAL↑br_comp = VAL↑comp

expr     ::= br_comp
             VAL↑expr = VAL↑br_comp
           | "minus" br_comp
             VAL↑expr = - VAL↑br_comp
```

Figure 1. A simple attribute grammar.

- The bold text constitutes a context-free grammar for a simple language of expressions. The notation used is a variant of Backus–Naur form, in which terminal symbols appear inside double quotes.
- An upward arrow signifies that the attribute is synthesised. For example, `VAL↑expr` should be read as 'the `VAL` attribute that is synthesised for the expression'.
- Each semantic rule indicates how the value of an attribute associated with the syntactic construct on the lhs of a production is obtained from attributes of

syntactic constructs on the rhs. For example, the fourth rule states that the VAL attribute of a syntactic construct of type subtr is obtained by subtracting the VAL attribute of the numb construct that is its third component from the VAL attribute of the numb construct that is its first component.

2.1 Use of the attribute grammar formalism in language engineering

Attribute grammars provide a formal yet intuitive notation for specifying the static semantics of programming languages and, consequently, they have been used in various compiler generation systems. A comprehensive survey of this work is given in Deransart, Jourdan and Lorho.[8] In most of these systems, non-executable specifications of attribute grammars are compiled into code in some conventional host programming language.

Attribute grammars have also been used for the specification and automatic construction of language-based editors. For example, the 'Synthesiser Generator'[28] constructs language-based editors from attribute grammar descriptions of target languages.

2.2 Use of the attribute grammar formalism as a general-purpose programming paradigm

Despite the substantial use of attribute grammars in the automatic construction of language-processors, surprisingly few attempts have been made to follow up on Knuth's suggestion to use the attribute grammar formalism as a general-purpose programming paradigm. It is not clear why this is. The following may be partial reasons.

- Compilers are clearly language-processors and use of the attribute grammar formalism in this problem domain is quite natural. The fact that many other programs are also language-processors may not be so obvious.
- In most of the language-processor generators that use attribute grammars, the specifications are pre-processed and translated into code in some conventional host language. The result is an increased indirection in the programming environment. This may be tolerated less in problem domains where the use, and the advantages, of the attribute grammar formalism are not so obvious.
- To avoid the pre-processing of attribute grammar specifications, an appropriate host programming language could be extended to include new 'attribute grammar' constructs. However, adding new constructs to most conventional programming languages is a non-trivial task, and two additional problems derive from the strict (i.e. non-lazy) evaluation order used in most conventional languages:

  (*a*) Executable attribute grammars that are driven by top-down backtracking parsers are more modular than those driven by alternative parsing strategies. However, a good deal of unnecessary computation of attributes occurs during backtracking if strict evaluation is used.

  (*b*) Implementation, in a strict language, of executable attribute grammars that allow fully general attribute dependencies requires either substantial transformation or multi-pass evaluation.

The difficulties that are met in extending conventional programming languages to accommodate executable specifications of attribute grammars do not arise with lazy, functional languages. New programming constructs may be introduced as combinators (i.e. higher-order functions) that are added to the language's standard environment. Also, with lazy evaluation, no value is computed until it is needed. Consequently, no unnecessary attribute computation occurs when the top-down parser of an executable attribute grammar is backtracking. In addition, neither transformations nor multiple evaluation passes are required to handle fully general attribute dependencies.

In the next section we describe four combinators that can be used by application programmers to glue together parts of a specification of an attribute grammar such that the result is a modular executable interpreter. One of the combinators is expressed as an infix operator, so that the visual appearance of the executable attribute grammar is similar to the conventional formulation. The notation that we use in discussion of the combinators, and in examples of their use, requires only minor changes to run in concrete lazy functional languages such as Miranda*[31] or Haskell.[15] The few textual replacements required to convert the notation to executable Miranda code are given at the end of the paper.

3. LAZY FUNCTIONAL PROGRAMMING LANGUAGES AND EXECUTABLE ATTRIBUTE GRAMMARS

A *functional language* is a language in which functions are first-class objects with all the privileges and uses that any other object has, for example they can be put in lists, given as arguments, returned as results, etc. A *pure functional language* is a language in which functions provide the only control structure and no side-effects are allowed (i.e. a function call can have no effect other than to return a value).

There are two types of pure functional programming language: *strict* languages in which all arguments to a function are evaluated before the function body is invoked, and *lazy* languages in which the evaluation of arguments is delayed until required.

Interest in lazy functional languages has grown in recent years, and they may now be regarded as an established addition to the programmer's repertoire. A discussion of the advantages and disadvantages of lazy functional languages, together with descriptions of applications and implementations, can be found in the April 1989 special issue of *The Computer Journal*.[34]

3.1 Supporting the attribute grammar programming paradigm in a lazy functional language

The benefits of lazy evaluation of attributed derivation trees were recognized in 1987.[17] The led to the development of an LR(1) parser generator, similar to Yacc, producing programs in a lazy functional language.[32] Functional languages have also been suggested as being appropriate hosts for attribute transformation systems.[23]

* Miranda is a trademark of Research Software Ltd.

```
attribute ::= VAL num

numb        =  term ("one",[VAL 1])
            |  term ("two",[VAL 2])
            |  etc

summ        =  ψ(s1 numb)
                 [rule 1.1 (VAL↑lhs) <-  same[VAL↑s1]]
            |  ψ(s1 numb...s2 !"plus"...s3 summ)
                 [rule 1.2 (VAL↑lhs) <-   add[VAL↑s1,VAL↑s3]]

subtraction =  ψ(s1 numb...s2 !"minus"...s3 numb)
                 [rule 1.3 (VAL↑lhs) <-   sub[VAL↑s1,VAL↑s3]]

compound    =  subtraction  |  summ

br_compound =  ψ(s1 !"("...s2 compound...s3 !")")
                 [rule 1.4 (VAL↑lhs) <-   same[VAL↑s2]]

expr        =  ψ(s1 br_compound)
                 [rule 1.5 (VAL↑lhs) <-   same[VAL↑s1]]
            |  ψ(s1 !"minus"...s2 br_compound)
                 [rule 1.6 (VAL↑lhs) <- negate[VAL↑s2]]

same     [x]              = x
add      [VAL x, VAL y]   = VAL (x + y)
sub      [VAL x, VAL y]   = VAL (x - y)
negate   [VAL x]          = VAL (- x)


Example Use
expr [([],["minus","(","one","plus","two",")","."])]
                                    => [([VAL -3],["."])]
```

Note : lists are written with square brackets and commas, tuples are written using parentheses and commas, and function application is denoted by juxtaposition and is left associative. The combinator | is defined as an infix operator with a lower precedence than function application. The detailed syntax of productions and attribute rules are discussed later.

Figure 2. Passage 1: a program constructed as an executable specification of an attribute grammar.

Despite these related developments, no one would appear to have considered supporting the attribute grammar programming paradigm in a lazy functional language. This is surprising, since it is relatively easy to do so.

The attribute grammar paradigm can be supported in a lazy functional programming language by the introduction of four combinators (described in more detail later) denoted by `term`, !, |, and ψ. Fig. 2 illustrates how these combinators can be used to construct a program as an executable attribute grammar. (We shall refer to such programs as *passages* from now on).

As can be seen, passage 1 is very similar in structure to the attribute grammar given in Fig. 1; the difference is that it is executable. Each 'production' defines an interpreter that is driven by a top-down, recursive syntax-directed parser.

3.2 Interpreters as functions

We choose to regard *interpreters*, such as `expr`, as functions with the following type:

```
[([attribute], [terminal])]
          →[([attribute], [terminal])]
```

That is, an interpreter is a function which maps a list of pairs of type `([attribute], [terminal])` to a list of pairs of the same type.

- Each pair `(as, ts)` that is in the list that is input to an interpreter is such that the list of attributes `as` may be regarded as a context, and `ts` as a sequence of terminal symbols to be interpreted in that context. We shall refer to such pairs as *att_term_pairs*.
- Each att_term_pair `(as', ts')` in the list that is output by an interpreter is related to exactly one att_term_pair `(as, ts)` in the input list such that: (i) `as'` is a subset of the union of `as` and the interpretation of some initial segment of `ts`, and (ii) `ts'` is a list of the remaining uninterpreted terminal symbols in `ts`.
- Interpreters return lists of att_term_pairs because a sequence of terminals may have more than one interpretation.
- Interpreters are regarded as accepting lists of att_term_pairs for a number of reasons, one being that it simplifies their composition.

An example application of an interpreter is given at the

COMBINATOR	TYPE
term	(terminal, [attribute]) -> interpreter
!	terminal -> interpreter
|	interpreter -> interpreter -> interpreter
ψ	[(identifier, interpreter)] -> [attribute_rule] -> interpreter
NOTES	If T is type, then [T] is the type of lists whose elements are of type T, if T1 to Tn are types, then (T1,...,Tn) is the type of tuples with objects of these types as components, and if T1 and T2 are types, then T1->T2 is the type of functions with arguments in T1 and results in T2.

Figure 3. The types of the attribute grammar combinators.

bottom of Fig. 2: applying `expr` to a list containing a single att_term_pair whose first component is an empty context[ ], and whose second component is the list of terminals ["minus","(", etc., results in a list containing a single att_term_pair whose first component is [VAL −3] and whose second component is the list of uninterpreted terminals [“.”].

3.3 The attribute grammar combinators

Formal definitions of the combinators `term`, `!`, `|`, and ψ are given later. Their types are given in Fig. 3.

3.3.1 The combinator term

The combinator `term` takes as argument a pair, consisting of a single terminal symbol followed by a list of attributes (the meaning of the terminal symbol), and returns an interpreter for that terminal symbol.

Example use of **term**

```
us_bill=term ("billion", [VAL 10^9,
                          DERIV "USA"])
uk_bill=term ("billion", [VAL 10^12,
                          DERIV "UK"])
Example application of interpreters
us_bill [([],["billion", "."])]
          => [([VAL 10^9, DERIV "USA"],
                                  ["."])]
uk_bill [([], ["two", "."])] => []
```

3.3.2 The combinator !

the combinator ! takes a single terminal symbol as argument and returns an interpreter that recognises and removes that terminal symbol but does not interpret it. The attributes in the input to the interpreter, i.e. the 'inherited' context, are simply copied into the output from the interpreter.

Example use of !

```
clbr=!")"
Example application of the interpreter
clbr [([VAL 10],[")", "xx"])]
       => [([VAL 10]. ["xx"])]
```

Notice that contexts such as [VAL 10] 'pass unchanged' through interpreters that are constructed using !.

3.3.3 The combinator |

The combinator | takes two interpreters as argument and returns an 'alternate' interpreter as result. This alternate interpreter applies both of the component interpreters to the input and appends their results.

Example use of |

```
us_or_uk_bill = us_bill |br_bill
Example application of the interpreter
us_or_uk_bill [([],["billion","xx"])]
     => [([VAL 10^9, DERIV "USA"],
                                  ["xx"]),
         ([VAL 10^12, DERIV "UK"],
                                  ["xx"])]
```

3.3.4 The combinator ψ

The combinator ψ takes two arguments: (i) a 'production' that denotes the identity and order of application of the component interpreters, and (ii) a list of 'attribute rules' that state how to compute the synthesised attributes of the interpreter on the lhs of the production and the inherited attributes that are to be used as context by the 'component' interpreters on the rhs of the production. Use of ψ is illustrated in Fig. 4.

```
add_two_numbs =
 ψ (s1 numb ... s2 numb)
    [rule 2.1 (VAL↑lhs)  <- add [VAL↑s1,
                                 VAL↑s2]]
context_numb =
 ψ (s1 numb)
    [rule 2.2 (VAL↑lhs)  <- add [VAL↑s1,
                                 VAL↓lhs]]
add_two_numbs' =
 ψ (s1 numb ... s2 context_numb)
    [rule 2.3 (VAL↑lhs)  <- same [VAL↑s2],
     rule 2.4 (VAL↓s2)   <- same [VAL↑s1]]
add  [VAL x, VAL y] = VAL (x + y)
same [x]            = x
Example application of interpreters
add_two_numbs [([],["1","2","."])]
                    => [([VAL 3],["."])]

context_numb [([VAL 10],["2","."])]
                    => [([VAL 12],["."])]
```

Figure 4. Passage 2: example use of ψ.

The definitions in Fig. 4 may be read as follows:

- The interpreter `add_two_numbs` recognises a structure that consists of a sub-structure `s1` of 'type' `numb` followed by a sub-structure `s2` of 'type' `numb`. By semantic rule 2.1, the `VAL` attribute that is synthesised (↑) for the lhs (i.e. returned by `add_two_numbs`), is equal to the result obtained by applying the attribute function `add` to a list of attributes containing the `VAL` attributes that were synthesised for the `numbs` `s1` and `s2`.
- The interpreter `context_numb` recognises a structure that consists of a structure `s1` of type `numb`. By semantic rule 2.2, the `VAL` attribute that is synthesised for the lhs (i.e. returned by `context_numb`), is equal to the result of applying the attribute function `add` to a list of attributes containing:
  (*a*) the `VAL` attribute that was synthesised (↑) for the `numb s1`,
  (*b*) the `VAL` attribute that was passed down as context (↓) to the interpreter `context_numb`.
- The interpreter `add_two_numbs'` recognises a structure that consists of a structure `s1` of type `numb` followed by a structure `s2` of type `context_numb`:
  (*a*) By semantic rule 2.3, the `VAL` synthesised for the lhs `lhs` (i.e. returned by `add_two_numbs'`), is equal to the result of applying the attribute function `same_as` to a list of attributes containing the `VAL` attribute that was synthesised for the `context_numb s2`.
  (*b*) By semantic rule 2.4, the `VAL` attribute inherited as context by the `numb s2` is equal to the result of applying the attribute function `same` to a list of attributes containing the `VAL` attribute that was synthesised for the `numb s1`.

Note that in the use of the ψ combinator: (i) the productions must not be left recursive, (ii) all attribute functions are of type `[attribute]` → `attribute`, i.e. they are all functions from lists of attributes to a single attribute, (iii) the order of the attribute rules within a definition is irrelevant, and (iv) if the 'names' of the synthesised and inherited attributes are disjoint, then ↑ and ↓ can both be replaced by ‡, which is to be read as 'of'.

4. EXAMPLES OF PROGRAMS CONSTRUCTED AS EXECUTABLE ATTRIBUTE GRAMMARS

Database query processors, natural language interpreters, theorem provers, specification transformers, and expression evaluators are immediate candidates for application of the attribute grammar programming paradigm. Examples of passages in each of these categories have been constructed using the combinators described in this paper.[10,11,18]

With a little change in perspective, many other types of programs may be regarded as interpreters and constructed as attribute grammars. In this section, we consider three problems that may not generally be recognised as language-processing problems. The first is concerned with a simple numeric computation, the second with file-processing, and the third with tree-manipulation. These examples are presented for two reasons: (i) to illustrate the wider applicability of the attribute grammar paradigm and the notion of solving problems in terms of relevant structures; and (ii) to illustrate certain aspects of the approach that we discuss further in Section 5.

4.1 A simple numeric example

An initial segment of the fibonacci sequence is 1 1 2 3 5 8 13 21. A function for calculating the *n*th fibonacci number, where *n* is given as argument, can be defined intuitively in a functional language, as follows:

```
fib n=1                          , if n<2
fib n=fib (n-1)+fib (n-2), if n>=2
```

This function has exponential complexity, but can be transformed to a function with complexity $O(n)$ (i.e. linear complexity) by introducing a 'generalising' function `g`:

```
fib n=first (g n)
      where
      first (a, b)=a
      g       n   =  (1, 1)   ,  if n<2
      g       n   =  (x+y, x),   if n>=2
                     where
                     (x, y)=g (n-1)
```

An attribute grammar solution, with complexity $O(n)$, can be obtained by making the primitive recursive structure of the input explicit, for example the natural number 4 is input as 'succ succ succ one'. Two attributes are associated with each such number: `FIB` standing for fibonacci value, and `PFIB` for the fibonacci value of the predecessor of the number. The resulting passage is given in Fig. 5.

4.2 A simple data-processing example

Construction of passages involves refinement through five steps. We illustrate these steps with a simple data-processing example.

Suppose that a program is required to calculate the average number of entries per record in a file in which each record consists of an initial field containing a numeric identifier followed by one or more alphanumeric

```
attribute ::= FIB num | PFIB num
numb =
  ψ(s1 !"one")
  [rule 3.1 (FIB↑lhs) <- fib_1[],
   rule 3.2 (PFIB↑lhs)<- pfib_0[]]
| ψ(s1 !"succ" ... s2 numb)
  [rule 3.3 (FIB↑lhs) <- add[FIB↑s2,PFIB↑s2],
   rule 3.4 (PFIB↑lhs)<- make_prev[FIB↑s2]]
     where
     fib_1     []              = FIB 1
     pfib_0    []              = PFIB 0
     add       [FIB x, PFIB y] = FIB (x + y)
     make_prev [FIB x]         = PFIB x

example application of interpreter
numb [([],["succ","succ","succ","one"])]
                 => [([FIB 3,PFIB 2],[])]
```

Figure 5. Passage 3: calculating fibonacci numbers.

string entries. Records are separated by semicolons, fields by commas, and end-of-file is signified by a period. A solution to this problem, shown in Fig. 6, can be obtained as follows.

Step 1. A grammar is defined for a language whose expressions include the specified input structures. This grammar is given in bold text in Fig. 6.

Step 2. Attributes that are relevant to the problem are identified and their types specified in terms of the base types `num`, `bool`, etc. The resulting formal introduction of the attributes is shown at the top of Fig. 6.

Step 3. Appropriate attribute rules are specified. These rules are numbered 4.1–4.10 in Fig. 6. Notice that rule 4.3 defines the average number of entries per record in a file in terms of two other attributes of the file. The fact that the file consists of a list of records is irrelevant to the specification of this rule (we discuss this further in the next section).

Step 4. Appropriate attribute functions are defined. These include `calc_average`, etc.

An example application of the interpreter is given at the bottom of Fig. 6.

4.3 A one-pass tree processor

The problem, in this example, is to construct a program that accepts a binary tree as input and which returns as result an equivalent tree except that all node values are equal to the maximum value in the input tree.

If the attribute grammar combinators are defined to allow inherited (context) attributes for an interpreter to be calculated from synthesised attributes returned by interpreters to its right in the production, then a passage can be constructed which returns the required result in a single pass over the input tree. Such a passage is given in Fig. 7.

- Rule 5.1 states that the RESULT attribute that is synthesised for a rooted tree is the same as the RESULT attribute that is synthesised for the tree that comprises it.
- Rule 5.2 states that the REPVAL attribute (i.e. the replacement value) that is inherited by a tree which comprises a rooted tree is obtained by converting the type of the MAX attribute that is synthesised for the tree.

```
attribute ::= NUM_RECS num | NUM_ENTS num | AV_ENTS num

file          = ψ (s1 list_of_recs ... s2 ! ".")
                  [rule 4.1 (NUM_ENTS↑lhs) <-       same_as [NUM_ENTS↑s1],
                   rule 4.2 (NUM_RECS↑lhs) <-       same_as [NUM_RECS↑s1],
                   rule 4.3 (AV_ENTS ↑lhs) <-  calc_average [NUM_ENTS↑lhs, NUM_RECS↑lhs]]

list_of_recs = ψ (s1 record)
                  [rule 4.4 (NUM_RECS↑lhs) <- const_one_rec [],
                   rule 4.5 (NUM_ENTS↑lhs) <-       same_as [NUM_ENTS↑s1]]
               | ψ (s1 record ... s2 ! ";" ... s3 list_of_recs)
                  [rule 4.6 (NUM_RECS↑lhs) <- incre_num_recs[NUM_RECS↑s3],
                   rule 4.7 (NUM_ENTS↑lhs) <-  add_num_ents [NUM_ENTS↑s1, NUM_ENTS↑s3]]

record        = ψ (s1 identifier ... s2 !"," ... s3 list_of_ents)
                  [rule 4.8 (NUM_ENTS↑lhs) <-       same_as [NUM_ENTS]↑s3]]

list_of_ents = ψ (s1 entry)
                  [rule 4.9 (NUM_ENTS↑lhs) <- const_one_ent []]
               | ψ (s1 entry ... s2 ! "," ... s3 list_of_ents)
                  [rule 4.10 (NUM_ENTS↑lhs)<- incre_num_ents[NUM_ENTS↑s3]]

calc_average          [NUM_ENTS x, NUM_RECS y] = AV_ENTS  (x/y)
const_one_rec         []                       = NUM_RECS 1
same_as               [x]                      = x
incre_num_recs        [NUM_RECS x]             = NUM_RECS (1 + x)
add_num_ents          [NUM_ENTS x, NUM_ENTS y] = NUM_ENTS (x + y)
const_one_ent         []                       = NUM_ENTS 1
incre_num_ents        [NUM_ENTS x]             = NUM_ENTS (1 + x)

process d = file [([],tokenise d)]

Example application
process  "1234,hesselink,hensen,jones;2345,bauer,partsch,sharir,morgan;5678,heath,li."
                        =>[([NUM_ENTS 9,NUM_RECS 3,AV_ENTS 3],[])]
```

Note : We assume that the interpreters `identifier` and `entry` have been defined elsewhere. The function `tokenise` converts the input string to a list of terminals.

Figure 6. Passage 4: calculating average number of entries in records in a file.

```
attribute ::= VAL num | RESULT [char] | MAX num | REPVAL num
numb        = as in passage 1
rooted_tree = ψ (s1 tree)
                [rule 5.1 (RESULT↑lhs) <-    same_as [RESULT↑s1],
                 rule 5.2 (REPVAL↓s1)  <-    convert [MAX↑s1]]
tree        = ψ (s1 !"-")
                [rule 5.3 (MAX↑lhs)    <- const_zero [],
                 rule 5.4 (RESULT↑lhs) <-  null_tree []]
              | ψ (s1 !"(" ... s2 numb ... s3 tree ... s4 tree ... s5 !")")
                [rule 5.5 (MAX↑lhs)    <-    maximum [   VAL↑s2,   MAX↑s3,   MAX↑s4],
                 rule 5.6 (RESULT↑lhs) <-  make_tree [REPVAL↓s2,RESULT↑s2,RESULT↑s3],
                 rule 5.7 (REPVAL↓s3)  <-    same_as [REPVAL↓lhs],
                 rule 5.8 (REPVAL↓s4)  <-    same_as [REPVAL↓lhs]]

same_as      [x]                          = x
convert      [MAX x]                      = REPVAL x
const_zero   []                           = VAL 0
null_tree    []                           = RESULT "-"
maximum      [REPVAL x,MAX y,MAX z]       = MAX (maximum_of [x,y,z])
make_tree    [REPVAL x,RESULT y,RESULT z] = RESULT ("("++(show x)++" "++y++" "++z++")")

Example application
rooted_tree [([],tokenise"(three (one (four - -)(five - -))(three (four - -)(two  - -)))")]
   => [(RESULT "(five (five (five - -) (five - -)) (five (five - -) (five - -)))", [])]
```

Note : We assume that the interpreters `maximum_of` and `show` have been defined appropriately elsewhere. The function `maximum_of` picks the maximum from a list, and the function `show` converts a number to character format.
Note : The definitions of the attribute grammar combinators given in Appendix II do not allow inherited (context) attributes for an interpreter to be defined in terms of synthesized attributes returned by interpreters to its right in the production. Definitions of combinators that do allow such 'right inheritance' can be constructed in a lazy functional language but are more complex than those given in Appendix II.

Figure 7. Passage 5: a one-pass tree processor.

- Rules 5.3 and 5.4 state that if a tree is a null tree then two attributes are synthesised for it: MAX 0 and RESULT "−".
- Rule 5.5 states how the MAX attribute is to be synthesised for a non-null tree.
- Rule 5.6 states how the RESULT attribute for a non-null tree is to be synthesised from the REPVAL attribute that is inherited from its context and the RESULT attributes that are synthesised for its two subtrees.
- Rules 5.7 and 5.8 state that the REPVAL attribute that is inherited by a tree is also inherited by its two subtrees.

Passage 5 illustrates how a lazy functional implementation of executable attribute grammars can be used to obtain a nearly completely declarative program for a problem that in many other programming environments would require a procedural solution.

5. ADVANTAGES OF CONSTRUCTING PROGRAMS AS EXECUTABLE ATTRIBUTE GRAMMARS

The major advantage of constructing programs as executable attribute grammars is the reasoned modular declarative structure that results. This structure facilitates various aspects of the software development process. In particular, it facilitates the use of formal methods in the analysis and construction of programs. It also facilitates prototyping and requirements analysis. We discuss these issues further in this and the next section.

5.1 Modularity of passages

Modular design is crucial to the development of 'good' software. There are two important types of modularity. The first is concerned with the decomposition of programs into cleanly separated interacting components. The second is concerned with the separation of structure from content.

5.1.1 Lazy functional languages and modularity

Strong arguments have been made[16] claiming that lazy functional languages not only support the two types of modularity mentioned above but also provide new types of 'glue' (higher-order functions and lazy evaluation) that allow programs to be modularised in ways that are impossible using conventional programming languages. We do not reiterate the arguments here, but refer interested readers to Hughes,[16] where many examples are given as illustration of the claims made.

5.1.2 Attribute grammars and modularity

The attribute grammar formalism is itself extremely modular. Syntactic issues are cleanly separated from semantic concerns, and the computation of attributes is well structured. Executable attribute grammars reflect this modularity – interpreters are distinct modules that interact through well-defined interfaces.

Use of the attribute grammar programming paradigm results in modular passages that are structured according

to the structure of the input data. The paradigm, thereby provides guidance to programmers on how to structure their programs. This reduction in the design space not only facilitates subsequent modification of programs to accommodate changes in requirements, but also the re-use of components in other applications (we discuss this further in Section 6).

If the attribute grammar programming paradigm is supported in a language that allows user-defined higher-order functions, the 'pattern of computation' that is represented by the productions of the grammar can be abstracted out and stored for re-use. (We also discuss this further in Section 6.)

5.1.3 Top-down parsing, lazy evaluation and modularity

Top-down, fully backtracking parsers are more modular than bottom-up parsers in the sense that parsers for individual syntactic structures can be compiled, tested, and used separately (provided that component interpreters on which they depend are available). Executable attribute grammars that are driven by top-down, fully backtracking parsers are similarly more modular than those driven by alternate parsing strategies. However, there is a price to pay: parsing is less efficient and, if the implementation language uses strict (non-lazy) evaluation, a good deal of unnecessary recomputation of attributes occurs during backtracking. The second problem does not occur if the executable attribute grammars are built in a lazy (non-strict) language owing to the fact that attribute values are only computed if required, and then only to the extent required.

5.2 Declarativeness of passages

Declarative programs are more easy to analyse, transform and modify than their procedural counterparts. As illustrated by the examples in Section 4, the lazy functional implementation of the attribute grammar programming paradigm can lead to programs that are nearly completely declarative.

In general, the order of productions within a passage, and the order of attribute rules within a production, is irrelevant. If the names of the synthesised and inherited attributes are disjoint, then the upward and downward arrows in a passage can both be replaced by a single symbol. The only procedurality that cannot be removed from a passage is that which results from (i) the fact that interpreters are applied in an order which is given on the rhs of productions, and (ii) the fact that attribute rules are used as right to left re-writes. However, it could be argued that this procedurality is inherent in the problem and cannot be avoided, but can be hidden to some extent by encouraging programmers to think declaratively in terms of data dependencies rather than procedurally in terms of the order in which interpreters are applied and attributes computed.

5.3 Passages and formal methods

Various formal methods have been proposed to facilitate the construction of provably correct efficient programs. These methods may be classified as being either 'analytic' or 'constructive'. *Analytic methods* include those in which programs are annotated with conditions stated in some formal language. The correctness of the final product is assured by verifying through formal proof that the conditions remain satisfied after each stage of refinement. *Constructive methods* include those in which correct but inefficient programs are (i) derived from proofs, or (ii) written as executable specifications and then transformed to more efficient forms using correctness-preserving transformations. Programs developed in this way are 'correct through construction'.

Papers presented and discussions held at the 1990 ACM International Workshop on Formal Methods in Software Development[27] suggest that the acceptance of formal methods by the non-academic programming community continues to be slower than anticipated. It appears that one problem with analytic methods is that the separation of verification from construction is not conducive to the acceptance of such methods by the majority of programmers, who find constructing programs more rewarding than proving them correct. A problem with constructive methods is that program transformation is extremely difficult and theories to support this activity are relatively immature.

These problems are compounded by the fact that conventional (i.e. procedural, imperative) languages facilitate neither proof nor program transformation. Recognition of this has stimulated interest in alternate programming paradigms which better support these activities. The lazy functional paradigm is one example that is finding growing support for the following reasons:

- Lazy functional languages (LFLs) are referentially transparent. The result of evaluating an expression is independent of the order in which the evaluation is carried out, since a variable or expression always denotes the same value within a given scope. A consequence of this is that we can always replace an expression by one that is equal to it without changing the value of the whole expression that contains it. This property facilitates reasoning about programs and helps in proofs of correctness and in program transformation.
- LFLs that support user-defined higher-order functions enable many related functions to be realised as specific instances of one 'generic' function, resulting in shorter programs, improved readability and re-use of proofs and patterns of transformation.
- LFLs that allow partial application enable functions to be defined as compositions of other functions with no need to make parameters explicit. The result is more natural and compact programs that are easier to manipulate for the purpose of proof and transformation.

These features of lazy functional languages are complemented by those of the attribute grammar formalism. In the following, we discuss how the functional implementation of the attribute grammar programming paradigm can facilitate application of particular formal methods in the software development process.

5.3.1 Passages and 'program analysis'

Not only does the referential transparency of the functional host language facilitate proof, but also the structure of passages is clearly suited to structural inductive proof by cases.

5.3.2 Passages and 'the derivation of programs from proofs'

A comprehensive discussion of the technique of 'deriving programs from proofs' is given in Manber.[25] For the purpose of this paper, it is sufficient to illustrate the technique with a simple example. Suppose that we want to define a function `reverse` that reverses the order of elements in a list. The required definition may be derived from an 'inverted' structural inductive proof as follows: the empty list, denoted by `[]`, is identified as an appropriate base case and the first line of the definition immediately follows. The inductive hypothesis is that the recursive call of `reverse` applied to the tail of the input list does what is required. The problem is thereby reduced to defining a non-recursive function which we name *g* in this particular example. The specification of *g* follows immediately from an intuitive understanding of what `reverse` is required to do. The resulting definition is given below, where `(e:es)` denotes a list with head `e` and tail `es`:

```
reverse  []    =[]
reverse (e:es)=g e (reverse es)
                 where
                 g e l = l ++ [e]
```

The technique described above can be generalised to arbitrary inductive data structures and therefore to the development of passages. The inductive hypotheses would be that recursively defined interpreters return the attributes that are required.

5.3.3 Passages and 'program transformation'

Various methods have been developed by which programs can be transformed to alternate forms that are extensionally equivalent but more efficient.[3,5] The following are examples of methods that have been developed for transforming functional programs:

- Equations are derived from the original function definitions through application of the Burstall and Darlington (B&D)[5] transformation rules of *folding*, *unfolding*, *instantiation*, *abstraction*, and *law application*. New definitions are obtained by selecting adequate subsets of the derived equations.
- The original definition of a function `f` is modified to obtain a definition of a more general function `g`. A new definition of `f`, written in terms of the definition of `g`, is then obtained through application of B&D rules. An example of the result of applying this technique was given in Section 4.1.
- One method of generalising a function definition is to introduce an additional argument called an accumulating parameter. As example, consider the definition of `reverse` given above which has time complexity $O(n^2)$ if the operator `++` is linearly dependent on the length of its left argument. This definition can be generalized to a definition of a function `rev` by the introduction of an accumulating parameter `a` which accumulates the reversed list on the recursive descent. A new definition of `reverse` with linear time complexity can be written in terms of the definition of `rev`:

```
reverse=rev []
        where
        rev a  []   =a
        rev a (e:es)=rev (e:a) es
```

A major difficulty with program transformation is that it is often not at all obvious which method to apply when given a new problem. Program transformation is non-deterministic. Any heuristic that helps the programmer to recognise when a method might be appropriate is useful. An advantage of the functional attribute grammar paradigm is that passages are structured in a highly modular way to reflect the structure of the problem domain and not some arbitrary structure that is imposed by the programmer. It is possible that this structure will facilitate the identification of appropriate transformation strategies. At the very least, it will provide the programmer with a new way of looking at problems, and with some new insights into the transformation strategies. For example: (i) generalisation may be regarded as increasing the number of attributes that are returned by an interpreter, and (ii) the introduction of an accumulating parameter may be regarded as the introduction of an inherited attribute.

5.3.4 Passages and 'program calculation'

A number of constructive formal methods focus attention on small collections of higher-order functions that capture common patterns of computation. Programs are expressed as combinations of partial applications of these functions. Some of the higher-order functions may be defined recursively, but once that is done further explicit use of recursion and induction is avoided. The approach results in variable-free or nearly variable-free programs that may be transformed to more efficient forms using algebraic identities associated with the higher-order functions. Bird[4] refers to this style of program development as 'program calculation', and Backus[2] as 'function-level programming'. The following 5-step program development process illustrates the notion of program calculation.

(1) An initial definition of a required function is obtained using some formal method.

(2) Parameterisation is used to obtain a higher-order function that captures the pattern of the specific function from which it was derived.

(3) Algebraic identities associated with the higher-order functions are proven and documented.

(4) Definitions of the original and of other functions are obtained by partial application of the higher-order functions.

(5) Programs that involve explicit or implicit use of the higher-order functions are transformed using algebraic identities that have been proven and documented for the higher-order functions, together with other identities such as those introduced in definitions.

The functional implementation of the attribute grammar programming paradigm is well suited to program calculation. Not only does the higher-order nature of the host language support parameterisation and partial application, the structure of passages lends itself well to the notion of abstracting the 'pattern of computation' out from function definitions – the application-dependent pattern of computation is explicit in the 'grammar'

part of a passage. We illustrate this in the next section, where we discuss the use of abstraction further.

6. PASSAGES AND PROTOTYPING

Application of formal methods, as discussed in Section 5, assumes the existence of some comprehensive set of requirements against which a passage is judged (in analytic methods), or from which a passage is derived (in constructive methods). It is often the case, however, that a potential user of software is unable to give a complete and accurate specification of requirements until she or he has had an opportunity to experiment with a prototype implementation. Such experimentation frequently leads to a more comprehensive specification of what is required. In addition, the experience gained in building the prototype often helps in the construction of the 'real thing'. This role of prototyping has been recognised for some time and there is a growing acceptance that it is an essential part of the requirements analysis process.[24]

The prototyping activity is greatly facilitated if the prototype is constructed in a modular form. In Section 5, we discussed the question of modularity with respect to the functional implementation of the attribute grammar programming paradigm. We now expand on one type of modularity supported by the paradigm that is particularly relevant to prototyping.

6.1 Abstracting out patterns of computation

Functional languages that allow user-defined higher-order functions and partial application enable 'patterns of computation' to be abstracted out of functions and 'stored' in higher-order functions for subsequent re-use. The abstraction process is clerical: constants that appear on the rhs of a function definition may be replaced by parameters whose values are passed in as arguments to the new function. For example, parameterising [ ] and g in the definition of `reverse` given earlier results in a higher-order function `foldr`:

```
foldr u op []     =u
foldr u op (a:as)=op a (foldr u op as)
```

A new definition of `reverse` and of other functions with the same pattern can now be obtained by partial application of `foldr`, e.g.:

```
sum_of_list     =foldr 0 (+)
product_of_list =foldr 1 (*)
reverse         =foldr [] put_on_end
                 where
                 put_on_end e l=l++[e]
```

This technique can also be used to derive higher-order functions from passages. For example, parameterisation of the definitions of the interpreters in passage 1 gives the higher-order functions shown in Fig. 8.

The higher-order function `seqq` takes four arguments: two interpreters, a two_argument function and an attribute constructor v. An interpreter is returned as result. The original interpreter `summ` as well as new interpreters with similar structure can be defined by supplying appropriate arguments to `seqq`.

In addition to deriving higher-order functions from complete interpreters, we can also derive them from parts of interpreters. For example, the function `ps`

```
seqq int1 int2 f v =
  ψ(s1 int1)
    [rule 100.1 (v↑lhs) <-    same[v↑s1]]
 |ψ(s1 int1...s2 int2
                ...s3 (seqq int1 int2 f v))
    [rule 100.2 (v↑lhs) <- f[v↑s1,v↑s3]]

pair int1 int2 g v =
  ψ(s1 int1...s2 int2...s3 int1)
    [rule 100.3 (v↑lhs) <- g[v↑s1,v↑s3]]

bracketed int v =
  ψ(s1 !"("...s2 int...s3 !")")
    [rule 100.4 (v↑lhs) <- same[v↑s2]]

s_or_ps int1 int2 h v =
  ψ(s1 int1)
    [rule 100.5 (v↑lhs) <-     same[v↑s1]]
 |ψ(s1 int2...s2 int1)
    [rule 100.6 (v↑lhs) <-       h[v↑s2]]
```

Figure 8. Higher-order functions derived from passage 1.

standing for 'prefixed interpreter' can be derived from the second alternative in the production for `expr`:

```
ps int1 int2 j v =
  ψ(s int1...s2 int2)
    [rule 100.7( v↑lhs) ←j[v↑s2]]
```

The higher-order functions that are derived from inadequate prototype passages can be used to implement solutions to more comprehensive requirements that arise from experimentation. For example, the higher-order functions in Fig. 8 can be used to define an evaluator of a larger class of numeric expressions as illustrated in Fig. 9. The higher-order functions may also be used to define interpreters in other domains, as illustrated in Fig. 10.

The role of abstraction in the software development process, as illustrated above, is related to a number of techniques that have been proposed for use with other programming paradigms, e.g. 'generic programming',[13] use of 'clichés',[29] and the use of analogy.[26] An advantage of the functional attribute grammar paradigm is that passages are structured in a highly modular way to reflect the structure of the problem domain and not some arbitrary structure that is imposed by the programmer. This facilitates re-use of the patterns of computation that are abstracted out into higher-order functions and re-use of components that are held together by these patterns.

7. RELATED WORK

Hehner and Silverberg[14] have described a method of programming in which solutions to problems are structured around several grammars interacting through a communication graph. The approach differs from that described in this paper in several respects. In particular, it is proposed as an extension to the imperative programming paradigm.

Johnsson[17] has suggested that a new case-like construct be added to lazy functional languages in order to express attribute dependencies over data structures. No restrictions are placed on these dependencies except that they

```
nexpr =
 numb
 | bracketed (summ|product|subtr|power)VAL
 | negation
 where
 summ     = seqq nexpr (!"plus")   add VAL
 product  = seqq nexpr (!"times") mult VAL
 subtr    = pair nexpr (!"minus")  sub VAL
 power    = pair nexpr (!"^")    raise VAL
 negation = ps (!"minus") nexpr negate VAL

 mult [VAL x,VAL y] = VAL (x * y)
 raise[VAL x,VAL y] = VAL (x ^ y)

Example use of the interpreter
nexpr[([],tokenise"(one plus (two ^ two)).")]
                 =>  [([VAL 5],["."])]
```

Note : the attribute functions add, sub, and negate are defined as in passage#1:

Figure 9. Passage 6: an evaluator for arithmetic expressions.

```
bexpr =
 boolean
 | bracketed (conj | disj | implic) VAL
 | complement
 where
 boolean = term ("t",[VAL 1])
         | term ("f",[VAL 0])
 conj    = seqq bexpr (!"&")     l_and VAL
 disj    = seqq bexpr (!"or")     l_or VAL
 implic = pair bexpr (!">") l_implies VAL
 complement = ps (!"-") bexpr   l_neg VAL

 l_and    [VAL 1,VAL 1] = VAL 1
 l_and    [VAL x,VAL y] = VAL 0

 l_or     [VAL 0,VAL 0] = VAL 0
 l_or     [VAL x,VAL y] = VAL 1
 etc

Example use of the interpreters
bexpr [([],tokenise"-(t & (f > f) & - f).")]
                 =>  [([VAL 0],["."])]
```

Figure 10. Passage 7: an evaluator for boolean expressions.

should not be circular. However, Johnsson's approach assumes the existence of a parse tree and is therefore addressing a related but somewhat different problem from that in this paper.

Uddeborg[32] has built an LR(1) functional parser generator 'FPG' which accepts a very general class of attribute grammars. In doing so, Uddeborg has demonstrated how lazy evaluation combines elegantly with attribute grammar evaluation. The input to FPG is an attribute grammar written in a syntax taken from Yacc, the output is a program coded in LML,[1] a lazy and purely functional variant of ML developed at Chalmers University of Technology.

Edupuganty and Bryant[9] have shown how a two-level grammar can be used as a programming language. The objectives of their work would appear to be similar to those expressed in this paper. However, the solution proposed is disadvantaged on two counts: the TLG language proposed is not referentially transparent, and the approach would not appear to support an equivalent notion to inherited attributes.

In 1981 Katayama[19] presented an approach to programming based on attribute grammars, in which programs are hierarchically decomposed into modules that are characterised by their inputs and outputs. A set of equations, associated with each decomposition, specifies the relationships between the inputs and outputs of the modules that participate in the decomposition. Subsequently, Shinoda and Katayama[30] developed this work further and constructed an environment called SAGE to support what they refer to as *attribute grammar-based programming* in a language called AG. The programming paradigm supported by SAGE is substantially different from that suggested here. In particular, in SAGE there is no notion of parsing the input to a program, neither is attribute evaluation lazy. SAGE compiles programs written in AG into procedures in C. One advantage of the SAGE approach is that it allows recursive application of modules within attribute-evaluation computations. Equivalent expressive power can be achieved in the approach described in this paper by applying an interpreter recursively to attributes synthesised by a higher-level call of the interpreter. However, doing this is not as elegant as it is in SAGE. One disadvantage of SAGE is that the separation of syntactic and semantic aspect of a problem is not as clean as it is in the approach described in this paper.

There is a growing interest in the use of attribute grammars in software engineering in East Europe. In particular, Riedewald, Forbrig and Lämmel of the Information Section at the University of Rostock, and Simon of the Research Group on Theory of Automata at the Hungarian Acadamy of Sciences are working on projects that have similar objectives to the work described in this paper. A major difference is the programming 'styles' that are being integrated. Riedewald *et al.* are investigating the relationship between attribute grammars and logic programming, whereas Simon has developed a language that combines the attribute grammar paradigm with the imperative procedural programming paradigm.

8. CONCLUDING COMMENTS

We have shown how the attribute grammar programming paradigm may be readily supported by adding four combinators to the standard environment of a lazy functional programming langauge. We have given examples of the use of these combinators and have discussed the advantages of the resulting programming style. Perhaps the most promising future development would be an investigation of the role of passages in program transformation.

Acknowledgements

The author acknowledges the assistance of N.S.E.R.C. of Canada, and of Stephen Karamatos, Dimitris Phoukas, and Walid Saba of the University of Windsor.

REFERENCES

1. L. Augustsson and T. Johnsson, The Chalmers Lazy ML Compiler. *The Computer Journal* **32** (2), 127–141 (1989).
2. J. W. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM* **21** (8), 613–641 (1978).
3. F. L. Bauer, B. Moller, H. Partsch and P. Pepper, Formal program construction by transformation – computer-aided, intuition-guided programming. *IEEE Trans. Software Engineering* **15** (2), 165–179 (1989).
4. R. S. Bird, Algebraic identities for program calculation. *The Computer Journal* **32** (2), 123–126 (1989).
5. R. M. Burstall and J. Darlington, A transformation system for developing recursive programs. *JACM* **24**, 44–67 (1977).
6. P. Deransart, M. Jourdan and B. Lorho, *A Survey of Attribute Grammars. Part III – Classified Biography*. Report 417, INRIA (1985).
7. P. Deransart, M. Jourdan and B. Lorho, *A Survey of Attribute Grammars. Part I – Main Results on Attribute Grammars*. Report 485, INRIA (1986).
8. P. Deransart, M. Jourdan and B. Lorho, *A Survey of Attribute Grammars. Part II – Review of Existing Systems*. Report 510, INRIA (1986).
9. B. Edupuganty and B. R. Bryant. Two-level grammar as a functional programming language. *The Computer Journal* **32** (1), 36–44 (1989).
10. R. A. Frost, Constructing programs in a calculus of interpreters. *Proceedings of the 1990 ACM International Workshop on Formal Methods in Software Development* (1990).
11. R. A. Frost and W. Saba, A database interface based on Montague's approach to the interpretation of natural language. *International Journal of Man–Machine Studies* **33**, 149–176 (1990).
12. R. Giegerich, Composition and evaluation of attribute coupled grammars. *Acta Informatica* **25**, 355–423 (1988).
13. R. Gupta, W. H. Cheng, R. Gupta, I. Hardonag and A. A. Breuer. An object oriented VLSI CAD Framework. *IEEE Computer* **22** (5), 28–38 (1989).
14. E. C. R. Hehner and B. A. Silverberg, Programming with grammars: an exercise in methodology–directed language design. *The Computer Journal* **26** (3), 227–281 (1983).
15. P. Hudak and P. Wadler, *Report on the Programming Language Haskell. Version* 1.0. Available from the Computing Science Department at the University of Glasgow (1990).
16. J. Hughes, Why functional programming matters. *The Computer Journal* **32** (2), 123–126 (1989).
17. T. Johnsson, *Attribute Grammars as a Functional Programming Paradigm*. Springer Lecture Notes in Computer Science **274**, 155–173 (1987).
18. G. A. Jullien, S. Bandyopadhyay, W. C. Miller and R. A. Frost, A modulo bit level systolic compiler. *Proceedings IEEE International Symposium on Circuits and Systems*, Portland, Oregon, pp. 457–460 (May 1989).
19. T. Katayama, HFP: a hierarchical and functional programming based on attribute grammars. *Proceedings of the 5th International Conference on Software Engineering*, pp. 343–353 (1981).
20. D. E. Knuth, Semantics of context-free languages. *Mathematical Systems Theory* **2** (2), 127–145 (1968).
21. D. E. Knuth, Semantics of context-free languages: correction. *Mathematical Systems Theory* **5**, 95–96 (1971).
22. D. E. Knuth, *Examples of Formal Semantics*. Springer Lecture Notes in Computer Science **188**, 212–235 (1971).
23. P. Lipps, U. Moncke and R. Wilhelm, *OPTRAN – a Language/System for the Specification of Program Transformations: System Overview and Experiences* (Proceedings of the 2nd Workshop on Compiler Compilers and High Speed Compilation, Berlin, 10–14 Oct. 1988), ed. D. Hammer. Springer Lecture Notes in Computer Science, **371**, 52–65 (1988).
24. Luqi, Software evolution through rapid prototyping. *IEEE Computer* **22** (5), 13–27 (1989).
25. U. Manber, Using induction to design algorithms. *Comm. ACM* **31** (11), 1300–1313 (1988).
26. K. Miriyala and M. T. Harandi, Analogical Approach to specification derivation. *Proceedings ACM/IEEE Fifth International Workshop on Software Specification and Design*, pp. 203–210 (1989).
27. M. Moriconi, 1990 ACM International Workshop on Formal Methods in Software Development. *ACM SIGSOFT* (in the Press).
28. T. Reps and T. Teitelbaum, The Cornell program synthesizer: a syntax-directed programming environment. *Comm. ACM* **24** (9) 563–573 (1981).
29. H. B. Rubenstein and R. C. Waters, The requirements apprentice: an initial scenario. *Proceedings ACM/IEEE Fifth International Workshop on Software Specification and Design*, pp. 211–218 (1989).
30. Y. Shinoda and T. Katayama, Attribute grammar based programming and its environment. *Proceedings, 21st Hawaii International Conference on System Sciences, Kailu-Kona, Hawaii*, pp. 612–620 (1988).
31. D. Turner, A non-strict functional language with polymorphic types. *Proceedings, IFIP International Conference on Functional Programming Languages and Computer Architecture, Nancy, France*. Springer Lecture Notes in Computer Science **201** (1985).
32. G. Uddeborg, A functional parser generator. *Licentiate Thesis*, Chalmers University of Technology, Göteborg (1988).
33. J. L. A. van de Snepscheut, *Mathematics of Program Construction*. Springer Lecture Notes in Computer Science **375** (1989).
34. P. Wadler, *The Computer Journal Special Issue on Lazy Functional Programming* **32** (2) (1989).

APPENDIX I Textual replacements to convert definitions in paper to executable Miranda code

The following textual replacements will convert all definitions given in the paper to executable Miranda code provided that the definitions of the combinators given in Appendix II are available.

sx !"y"	sx (uterm "y")	eg s1 !"&" becomes s1 (uterm "&")
|	$orelse	
ψ	rstc	
↑	$u	
↓	$d	
<-	EQ	

Note : The introduction of brackets when converting ! is owing to the fact that operators have a lower priority than function application.

Figure 11. Conversion to Miranda.

APPENDIX II Formal definitions of the combinators and ancillary functions in the programming language Miranda

```
terminal    == [char]
interpreter == [([attribute],[terminal])]->[([attribute],[terminal])]
att_direction   ::= ATT_DIR [char]
att_id          ::= ATID [char]
syntactic_sugar ::= EQ

fail :: interpreter
fail input = []

succeed :: interpreter
succeed input = input

term :: (terminal,[attribute]) -> interpreter
term pair  = concat . (map (interp_term pair) )
             where
             interp_term (t,atts) (inh, []    ) = []
             interp_term (t,atts) (inh,(u:us)) = [(atts,us)], if u = t
                                               = []         , otherwise
uterm x                    [] = []
uterm x ((atts,[]):rest)      = (uterm x rest)
uterm x ((atts,(t:ts)):rest) = (atts,ts):(uterm x rest), if t = x
                             = (uterm x rest)          , otherwise

orelse :: interpreter -> interpreter -> interpreter
(p1 $orelse p2) input = p1 input ++ p2 input
```

Note : `x == y` introduces `x` as a synonym for the type `y`, `id :: t` declares `id` to be of type `t`, the `$` symbol indicates that the function is defined as an infix operator, and '.' denotes function composition, ie (f . g) x = f (g x).

Figure 12. Formal definitions of the combinators `term`, `uterm` and `orelse`.

```
make_complex_interpreter  name interp_ids syn_rules inh_rules (inh_atts,inp) =
 res
 where
 res = [(applyrulestoself syn_rules inh_atts vvs, rest) |
         (vvs,rest) <- hassparts interp_ids inh_rules ([inh_atts] ++ [[]]) inp ]
    where
    apply_rules rules item = [apprule (syna,r,latts) item | (syna,r,latts) <- rules ]
    apprule (syna,r,latts) item   = r (getitems latts item)
    getitems attlist v            = [pick attn (v ! int_pos) | (int_pos, attn) <- attlist]
    pick attn list                = [v|v <- list ; name v = attn] ! 0
    hassparts interp_ids inhrs  atts inp = apply_interps interp_ids inhrs atts inp
      where
      apply_interps     [ ]        x          y     inp                = [([ ],inp)]
      apply_interps ((obj,interp_id):interp_ids) (inhr:inhrs) atts inp =
            [(v1:v2,res) | (v1,inp1)<- interp_id [((apply_rules inhr atts), inp)];
                           (v2,res) <- apply_interps interp_ids inhrs (atts ++ [v1]) inp1]
      applyrulestoself syn_rules inh_atts vvs =
                  g
                  where
                  g = apply_rules syn_rules ([inh_atts ++ g] ++ [g] ++ vvs)

sort_out_rules_and_make_complex_interpreter  name  interp_ids rules =
  concat . (map (make_complex_interpreter  name interp_ids syn_rules inh_rules))
  where
  syn_rules = [((int_pos,attn),c,d) | ((int_pos,attn),c,d)<- newrules;
                                                    (int_pos = 1) \/ (int_pos = 0)]
  (x:y:inh_rules)  = update empty_inh_rules newrules
  newrules         = [((pos_of p ((lhs succeed)++interp_ids),attn),attf,change_all latts)
                                                    | ((p,attn),attf,latts)<-rules]
  change_all latts = [(pos_of p ((lhs succeed)++interp_ids),attn)|(p,attn)<-latts]
  pos_of(p,updown)((obj,int):interp_ids) = 0,   if (p = obj) & (updown = ATT_DIR "down")
                                         = 1,   if (p = obj) & (updown = ATT_DIR "up")
                                         = 0,   if (p = obj) & (updown = ATT_DIR "of")
                                         = -100,if ~ (member (map first interp_ids) p)
                                         = (pos p interp_ids) + 1,otherwise
                                           where
                                           pos x ((obj,int):ys) = 1             ,if x = obj
                                                                = 1 + (pos x ys),otherwise
  first (a,b)                            = a
  empty_inh_rules                        = [] : empty_inh_rules
  update x []                            = x
  update x (r:rs)                        = update (addrule x r) rs
  addrule x ((int_pos,attn),c,d)         = (upto int_pos x)
                                           ++ [((int_pos,attn),c,d) : (x!int_pos)]
                                           ++ (after int_pos x)
  upto 0 x                               = []
  upto n (x:xs)                          = x : (upto (n-1) xs)
  after 0 (x:xs)                         = xs
  after n (x:xs)                         = after (n-1) xs
rule num (pid,attn) eq f latts      = ((pid,attn),f,latts)
(constructor $u  obj)               = ((pid,ATT_DIR "up"),   name (constructor undef))
                                       where [(pid,succeed)] = obj succeed
(constructor $d obj)                = ((pid,ATT_DIR "down"), name (constructor undef))
                                       where [(pid,succeed)] = obj succeed
lhs       x                         = [(-10,x)]
s0        x                         = [(1,x)]
s1        x                         = [(2,x)]
etc
rstc                                = sort_out_rules_and_make_complex_interpreter  name
```

Note : The function name is defined independently for each passage. For example, for passage#2, name would be defined as follows :

```
name (FIB x)  = ATID "fib"
name (PFIB x) = ATID "pfib"
```

The function name is used by the attribute grammar combinators to identify attributes by type.

Figure 13. Formal definition of the combinator `rstc` and of the associated operators `rule`, `u`, `d`, `lhs` `s0`, `s0`, etc.