APPLICATION PROGRAMMING AS THE CONSTRUCTION OF LANGUAGE INTERPRETERS

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

ABSTRACT

Application programming is often carried out in an ad hoc way compared to the construction of code for implementing operations in 'standard calculi' such as relational calculus, number theory, set theory, etc. The various programming techniques, such as 'abstraction' and 'transformation', which are often used in the latter are infrequently used in application programming. We believe that more structured approaches to application programming may facilitate the use of established programming techniques in application programming. In this paper, we explore one such approach that is based on the notion of 'application programming as the construction of language interpreters'. We have built a 'system' to support this approach. The system that we have built consists ofa number of higher order functions that can be used to 'glue together' parts of a specification of an attribute grammar such that the result is an executable interpreter. These functions facilitate the 'language processing' part of application programming. Our implementation language is a higher order, lazy, pure functional programming language; this facilitates the 'standard calculi' part of application programming. We claim that for certain types of application programming, our system not only helps the programmer to separate these two aspects of the problem, but that it also provides a framework in which both aspects can be addressed methodically, taking advantage of various programming techniques such as 'abstraction' and 'transformation', and subsequently re-integrated to obtain the required program. We have extended the basic notion of attribute grammars so that our approach has wider application. The extension is based upon a new notion of 'multi-input parsing'.

1. THE NEED FOR A MORE STRUCTURED APPROACH TO APPLICATION PROGRAMMING

1.1. Motivation

The major motivation for the work of which this paper is a part, is a belief that programmers would benefit by viewing certain types of application programming as the construction of language interpreters and structuring their solutions accordingly. We claim that such an approach will simplify the programmer's task in a number of ways. In particular, we believe that it will help application programmers to take advantage of more of the programming principles and techniques that have been developed during the evolution of programming as a discipline.

There would appear to be something fundamental and 'solid' about techniques such as 'abstraction' and 'transformation'; unfortunately it is not obvious how these techniques can be used in all aspects of application programming. We believe that more structured approaches to application programming should give new insight into how these techniques can be used. Such 'structured approaches' could be implemented as software development environments for application programming that include tools similar to those used by language engineers. That is, tools similar, but not necessarily identical to, for example, LEX, YACC, and The Cornell Synthesizer Generator (Reps and Teitlebaum 1988).

1.2. A structured approach to application programming

The approach to application programming that we propose in this paper, is based on the view of 'application programming as the construction of language interpreters'. The language to be interpreted is the language in which the input data is expressed.

The language processing parts of the problem are solved by specifying an appropriate executable attribute grammar for the 'input' language, and by identifying those computations on attributes that are required.

Those parts of the problem that involve computations in 'standard calculi' are solved by identifying appropriate existing code and by directly defining the required functions on attributes in terms of this code. If no appropriate code can be found, then either the application programmer will have to look for an 'expert', or he or she will have to switch mode and become a 'standard calculi' programmer.

According to the approach that we are proposing, application programming involves the following tasks :

(i) Definition of a grammar for the data input language.

(ii) Identification of attributes, of the components of the grammar, that are relevant to the problem.

(iii) Identification of the 'base type' of each attribute.

(iv) Specification of an appropriate attribute grammar.

(v) Definition of the required functions on the attributes in terms of operations from relevant standard calculi.

Similar approaches to programming have been proposed by Hehner and Silverberg (1983), Johnsson (1987), and Edupuganty and Bryant (1989).

1.3. The 'system' that we have built

To support the approach to application programming, we have developed a system that is based on the notion of 'attribute grammars' (Knuth 1968 and Bochmann 1976). The implementation that we present could be called an application generator. However, our 'software system' is different from most existing generators in that it enables users to construct executable specifications of interpreters, rather than specifications that need to be compiled.

The 'system' that we have built consists of a number of higher order functions that can be used to 'glue together' parts of a specification of an attribute grammar, such that the result is an executable interpreter in a higher order, lazy, pure functional programming language called Miranda * (Turner 1985). We claim that for certain types of application, our 'system' not only facilitates the separation of the language processing parts of the problem from those parts involving computations in standard calculi, but that it also provides a framework in which these components can be addressed methodically, taking advantage of various programming techniques, and subsequently re-integrated to obtain the required program. We also claim that our system facilitates the re-use of code for all parts of application programs.

1.4. Feasibility of the approach

Owing to limited availability of appropriate programming languages, many useful programming techniques have been restricted in their application. For example, the concept of 'partial application', which is particularly useful when used in conjunction with 'abstraction', has been restricted in its application until fairly recently, by limited availability of programming languages that support 'partial parameterization' directly. The concept of 'attribute grammar' has been similarly restricted in its application. Attribute grammars have been used primarily in 'language engineering'. For example :

- they have been used to describe the syntax and semantics of programming languages (eg Watt 1979),
- they have been used in compiler generators (eg Farrow 1982),
- they have been used as the basis for 'syntax directed' editors (eg Reps and Teitelbaum 1989).

Now that higher order, lazy, pure functional, programming languages such as Miranda are becoming more readily available, we believe that both of these restrictions are gradually easing up. Application programmers who have access to such languages can take advantage of them in two ways :

(i) They can use attribute grammars, in ways similar to those described in this paper, to help them structure their solution and to tackle the language processing parts methodically.

(ii) They can use more of the established programming techniques to solve the 'standard calculi' part of the problem, since higher order, lazy, pure functional languages support most of these techniques.

* Miranda is a trademark of Research Software Ltd.

2. AN EXAMPLE

As an example of the approach, consider the following problem : we are to define a function that takes a sequence of english words, denoting decimal digits, as input and which returns the length of the initial 'non-decreasing' sub-sequence of this sequence of words, together with the sum of the 'digits' and the decimal 'value' of this sub-sequence. For example, suppose the input were "one two two one.", the function should return a result such as (LENGTH 3, SUMDIGS 5, WDSEQVAL 122).

2.1. Task 1 : Definition of a grammar for the input language

According to the approach that we are proposing, we begin by defining a grammar of the language of the input data. We do this using standard BNF notation :

```
numword  = "one" | "two" | "three" , etc.

wordseq  = numword
         | numword wordseq
```

This means that a numerical word or 'numword' is either the word "one", or the word "two", or the word "three", etc., and that a sequence of words or 'wordseq' consists of either a single 'numword' or a 'numword' followed by a 'wordseq'. Examples of acceptable sequences of words according to this grammar are : "one", "two", "one two", and "three one two".

2.2. Task 2 : Identification of attributes that are relevant to the problem

The next step is to identify those attributes of the components of the input language that are relevant to the problem :

(i) The numeric values of 'numwords' such as "one", "two", etc. are relevant attributes that we shall refer to as NUMVAL attributes.
(ii) The length, sum of digits, and decimal 'value' of 'wordseqs' are relevant attributes. We refer to these as LENGTH, SUMDIGS and WDSEQVAL attributes, respectively.

If the problem had been to read in a sequence of words and to 'fit' these words into lines of some maximum length, then a relevant attribute would be the length of the 'numwords' : "one", "two", "three", etc.

2.3. Task 3 : Identify 'base type' of each attribute

We now identify the 'base type' of each relevant attribute. In this example, they all have base type 'num'. We represent the relationships between attributes and their base types in Miranda as shown below. We also introduce 'names' that are used later to pick out attributes :

```
attribute  ::= NUMVAL num | LENGTH num |
               SUMDIGS num | WDSEQVAL num
name (NUMVAL x)      = "numval"
name (LENGTH x)      = "length"
name (SUMDIGS x)     = "sumdigs"
name (WDSEQVAL x)    = "wdseqval"
```

This means that an object of type 'attribute' can be made from an object of type 'num' by applying any one of the four 'constructors', NUMVAL, LENGTH, SUMDIGS, or WDSEQVAL, to it.

Since the base type of all four attributes is 'num' the standard calculus of number theory can be used directly in the definitions of the required functions on the four attributes. However, before we do this, we specify an appropriate attribute grammar for the input language.

2.4. Task 4 : Specification of an attribute grammar

We begin by defining a 'dictionary' of the terminal symbols of the input grammar. The dictionary consists of a number of lists, one for each 'category' of terminal symbol. Each list consists of a number of pairs, one for each terminal symbol in the category. The first element of each pair is the terminal symbol, the second element is a list of attributes associated with that terminal symbol when used in that syntactic category.

In this example, the dictionary consists of only one list for the single category of terminal symbols 'numword'. This list consists of a number of pairs, one for each of the terminal symbols "one", "two", "three", etc. :

```
numword_list  = [("one", [NUMVAL 1]),
                 ("two", [NUMVAL 2])]
```

Now we create an interpreter for the syntactic category 'numword' using a higher order function 'mkint', that we have developed.

```
numword  = mkint  numword_list
```

The interpreter for 'wordseq' is obtained, as follows, by specifying an attribute grammar using a number of other higher order functions that we have developed. The Miranda code for these higher order functions can be found in Frost (1989).

```
wordseq = (consists_of [numword]
              with_att_syn_rules
                  [("length", "is", constant_length_of_1, [ ]),
                   ("sumdigs", "is", convert_numval_to_sumdigs, [("numval", fst_part)]),
                   ("wdseqval", "is", convert_numval_to_wdseqval, [("numval", fst_part)])]
              with_inh_att_calc_rules
                  [[ ]]
              with_failure_rules
                  [(decreasing_pair_of_numvals, [("numval", lhs), ("numval", fst_part)])])
          $orelse
          (consists_of [numword, wordseq]
              with_att_syn_rules
                  [("length", "is", add_one_to_length, [("length", snd_part),
                   ("sumdigs", "is", summ, [("numval", fst_part), ("sumdigs", snd_part)]),
                   ("wdseqval", "is", calc_wdseqval, [("numval", fst_part), ("wdseqval", snd_part), ("length", snd_part)])]
              with_inh_att_calc_rules
                  [[ ],
                   [("inh_numval_of_snd_part", "is", same_as, [("numval", fst_part)])]]
              with_failure_rules
                  [(decreasing_pair_of_numvals, [("numval", lhs), ("numval", fst_part)])])
```

We can explain this executable specification by giving a corresponding informal description in english :

A sequence of words, or 'wordseq', consists of a single 'numword' orelse it consists of a 'numword' followed by a sequence of words. If the 'wordseq' consists of a single 'numword', then three attributes are returned by the interpreter. Three 'attribute synthesis' rules are used to synthesize these attributes. The first rule is used to synthesize a 'LENGTH' attribute, the rule is that 'the length is a constant length of 1 and does not depend on any of the attributes of 'numword'. The second rule is used to synthesize a 'SUMDIGS' attribute, the rule is that 'the sum of the digits is obtained by converting the NUMVAL attribute of the first part of the wordseq (ie the single numword) to a SUMDIGS attribute'. The third rule is used to synthesize a WDSEQVAL attribute, the rule is that 'the wdseq value is obtained by converting the NUMVAL attribute of the 'numword' to a WDSEQVAL attribute.

If the 'wordseq' consists of a single component 'numword', there is no need to calculate any 'inherited attributes' and consequently, the set of 'inherited attribute calculation' rules for the single component 'numword' is empty. However, the set of 'interpretation failure' rules is not empty since the interpreter should fail if the 'wordseq' consisting of a single 'numword' is decreasing. This raises a question that we could have overlooked : In the original problem specification, we talk about a 'non-decreasing' sub-sequence of the input sequence. What do we mean by 'non-decreasing' if the input sequence consists of only one word? The answer to this question, that is implicit in the solution that we have given as example, is that the property of 'decreasingness' is established with respect to a 'context' or an inherited attribute that is given when the left hand side 'wordseq' interpreter is used. For example, if 'wordseq' is used in the context (NUMVAL 3) then the sequence of words "two" is decreasing and the interpreter will fail.

If the 'wordseq' consists of two components : a 'numword' followed by a 'wordseq' then the interpreter will return three attributes that are computed using three different synthesis rules as shown above. Also, in this case, the list of sets of 'inherited attribute calculation' rules has two elements, one for each of the components. The first set of rules, for computing the inherited attributes of the first component, ie the 'numword' part, is empty. However, the second set of rules, for computing the inherited attributes of the second component, ie the 'wordseq' part, is not empty. It consists of one rule which states that 'the 'numval' inherited attribute, that is passed as context to the second component, ie the interpreter 'wordseq', is the same as the synthesised 'numval' attribute that is returned from the first component, ie the 'numword' interpreter'. In general, inherited attributes for an interpreter p are calculated from inherited attributes passed in as context from the left hand side, if any, together with synthesized attributes that are returned from components, ie interpreters, to the left of p. Finally, the failure rule in the case in which the sequence of words consists of two components is the same as the failure rule when the sequence of words consists of one component only.

2.5. Task 5 : Definition of the required functions on attributes

The final task is to define the 'attribute calculation functions' that are used in the synthesis rules in the attribute grammar above. We define these functions in terms of operations from the standard calculus of number theory. In each case, the function is defined as a function which maps a list of attributes to a single attribute :

(i) If a wordseq consists of only one numword, then its LENGTH is 1, its SUMDIGS is obtained by converting the NUMVAL of the numword to a SUMDIGS value, and its WDSEQVAL is obtained by converting the NUMVAL of the numword to a WDSEQVAL :

```
constant_length_of_1 []                          = LENGTH 1

convert_numval_to_sumdigs [(NUMVAL x)] = (SUMDIGS x)

convert_numval_to_stringval [(NUMVAL x)] = (WDSEQVAL x)
```

(ii) If a wordseq consists of a numword n followed by a wordseq s, then its LENGTH will be the LENGTH of s plus 1, the SUMDIGS will be the 'summ' of the NUMVAL of n and the SUMDIGS of s, and its WDSEQVAL will be calculated from the NUMVAL of n and the LENGTH and WDSEQVAL of s. The required functions can be defined directly in terms of well known operations in the standard calculus of number theory as follows :

```
summ [(NUMVAL x), (SUMDIGS y)]   = (SUMDIGS (x + y))

add_one_to_length [(LENGTH x)]       = (LENGTH (x + 1))

calc_stringval [(NUMVAL x),(WDSEQVAL y),(LENGTH z)]  =
                                (WDSEQVAL ((x * (10^z)) + y)
```

(iii) Since we are also interested in testing to see if a wordseq is 'decreasing', the following definition is appropriate :

```
decreasing_pair_of_numvals [(NUMVAL x), (NUMVAL y)] = x > y
```

The resulting program, consisting of all of the text in bold-face in sub-sections 2.3, 2.4, and 2.2 together with the higher order library functions given in the appendix, constitutes the solution to the problem.

2.6. Sample output

We now give examples of the output returned when the interpreter 'wordseq' is applied in various contexts to various sequences of words. The function 'words' that is used in the following maps strings of characters to lists of words suitable for input to the interpreter. The word "Miranda" is the prompt from the Miranda interpreter, and the list below the Miranda prompt is the result returned by 'wordseq' :

```
Miranda wordseq [NUMVAL 0] (words "one.")
[([LENGTH 1, SUMDIGS 1, WDSEQVAL 1], ["."])]

Miranda wordseq [NUMVAL 2] (words "one.")
[ ]

Miranda wordseq [NUMVAL 0] (words "one two one.")
[([LENGTH 1, SUMDIGS 1, WDSEQVAL 1], ["two", "one", "."]),
 ([LENGTH 2, SUMDIGS 3, WDSEQVAL 12], ["one", "."])]
```

We can explain the first of these examples as follows : when the interpreter 'wordseq' is applied to the sequence "one." in the context [NUMVAL 0], a list of results is returned, one result for each interpretation. In this case, the list contains one result, a pair whose first component is the list of attributes [LENGTH 1, SUMDIGS 1,WDSEQVAL 1] and whose second component is the rest of the input after the interpreter has finished with it, ie the list ["."].

In the second example, the output is an empty list of results since the sequence "one.", in the context [NUMVAL 2], is decreasing.

In the third example, we see that the interpreter 'wordseq' that we have constructed is actually doing more than we required in the problem specification. It is giving us results for all of the non-decreasing sub-sequences of the input. For example, given the input "one two one.", in the context [NUMVAL 0], the interpreter returns results for the sub-sequence "one", and for the sub-sequence "one two". Clearly, we could select the required result from the two returned.

2.7. Comments on this example

In this example, we have illustrated how both 'synthesized' and 'inherited' attributes can be used to solve a problem. The inherited attribute was used to distinguish 'non-decreasing' sub-sequences of words from decreasing sub-sequences.

3. ANOTHER EXAMPLE

In this section, we present another example. The problem is to construct a program that converts arbitrary formulas of propositional calculus to clausal form. For example, given '(p implies (q and r)).' the program should return a result such as {{-p, q}, {-p, r}}.

3.1. Task 1 : Definition of a grammar for the input language

We begin by defining an appropriate grammar for the input language. For the purpose of this paper, we give an incomplete grammar that serves only as illustration :

```
wff          = expr  terminator

expr         = var | op_brack  compound  cl_brack
               | op_brack  implication  cl_brack

implication  = expr  implies_op  expr

compound     = conjunction  |  disjunction

conjunction  = expr | expr  conj_op  conjunction

disjunction  = expr | expr  disj_op  disjunction
```

3.2. Task 2 : Identification of relevant attributes

We now identify those attributes that are relevant to the problem :

(i) Since the problem is concerned with a 're-arrangement' of propositional variables, the only value of these variables that we are interested in is their 'name'. We shall refer to these values as PNAME attributes.
(ii) In our approach to this problem, we regard logical operators such as 'implies_op' only as 'syntactic markers', and consequently we do not associate any attributes with these operators.
(iii) For all other constructs in the language, we are interested in the associated clause set. We shall refer to such attributes as CLSET attributes.

3.3. Task 3 : Identification of base types

We now identify appropriate base types for these attributes :

```
attributes  ::= PNAME [char]  |  CLSET [[[char]]]

name (PNAME x)     = "pname"
name (CLSET x)     = "clset"
```

These statements mean that PNAME attributes can be made out of character strings using the constructor 'PNAME', and CLSET attributes can be made from lists of lists of character strings using the constructor 'CLSET'.

3.4. Task 4 : Specification of an attribute grammar

We now specify an appropriate attribute grammar. We begin by specifying the dictionary :

```
propvar_list     = [("p", [(PNAME "p")]),
                    ("q", [(PNAME "q")]),
                    ("r", [(PNAME "r")]),
                    ("s", [(PNAME "s")]),
                    ("t", [(PNAME "t")])]

implies_op_list  = [("implies", [ ]),
                    ("->", [ ])]

conj_op_list     = [("and", [ ])]

disj_op_list     = [("or", [ ])]

op_brack_list    = [("(", [ ])]

cl_brack_list    = [(")", [ ])]

terminator_list  = [(".", [ ])]
```

We now make interpreters for the basic syntactic categories :

```
var         = mkint  propvar_list

implies_op  = mkint  implies_op_list

conj_op     = mkint  conj_op_list

disj_op     = mkint  disj_op_list

op_brack    = mkint  op_brack_list

cl_brack    = mkint  cl_brack_list

termin      = mkint  terminator_list
```

We now specify the interpreters for the non-basic syntactic categories :

```
wff  = consists_of [expr, termin]
                with_att_syn_rules
                [("clset", "is", same_as, [("clset", fst_part)])]
                with_inh_att_calc_rules
                [[ ], [ ]]
                with_failure_rules
                [ ]
expr = (consists_of [var]
                with_att_syn_rules
                [("clset", "is", makeclset, [("pname", fst_part)])]
                with_inh_att_calc_rules
                [[ ]]
                with_failure_rules
                [ ])
          $orelse
          ((consists_of [op_brack, compound, cl_brack]
                with_att_syn_rules
                [("clset", "is", same_as, [("clset", snd_part)])]
                with_inh_att_calc_rules
                [[ ], [ ], [ ]]
                with_failure_rules
                [ ])
          $orelse
          (consists_of [op_brack, implication, cl_brack]
                with_att_syn_rules
                [("clset", "is", same_as, [("clset", snd_part)])]
                with_inh_att_calc_rules
                [[ ], [ ], [ ]]
                with_failure_rules
                [ ]))
implication = consists_of [expr, implies_op, expr]
                with_att_syn_rules
                [("clset", "is", att_cnf_implies, [("clset", fst_part),
                                                   ("clset", thrd_part)])]
                with_inh_att_calc_rules
                [[ ], [ ], [ ]]
                with_failure_rules
                [ ]
compound  = conjunction
                $orelse
                disjunction
conjunction = expr
              $orelse
              (consists_of [expr, conj_op, conjunction]
                with_att_syn_rules
                [("clset", "is", att_cnf_and, [("clset", fst_part),
                                               ("clset", thrd_part)])]
                with_inh_att_calc_rules
                [[ ], [ ], [ ]]
                with_failure_rules
                [ ])
disjunction  = expr
              $orelse
              (consists_of [expr, disj_op, disjunction]
                with_att_syn_rules
                [("clset", "is", att_cnf_or, [("clset", fst_part),
                                              ("clset", thrd_part)])]
                with_inh_att_calc_rules
                [[ ], [ ], [ ]]
                with_failure_rules
                [ ])
```

3.5. Task 5 : Definition of the 'attribute functions'

The following definitions of operations in clausal propositional calculus are appropriate for this problem :

```
|The function 'neglits' negates all literals in a clause

neglits clause     = [ neglit l | l <- clause]

neglit ('-' : x)   = x
neglit   y         = '-' : y

|The following function converts a clause set in disjunctive
|normal form to one in conjunctive normal form

conv_from_dnf_to_cnf  [ ]        = [[ ]]
conv_from_dnf_to_cnf  (cl:cls)   = [(l:clause) | l <- cl ;
                                    clause <- (conv_from_dnf_to_cnf cls)]

|The following function forms the negation of a clause set
|that is in conjunctive normal form

negate  c_set    = conv_from_dnf_to_cnf [ neglits c | c <- c_set]

|These functions compute the logical 'and', 'or', and 'implication'
|of two clause sets in conjunctive normal form

cnf_and  cnf_set1  cnf_set2 = cnf_set1 ++ cnf_set2

cnf_or   cnf_set1  cnf_set2  = [ c ++ k | c <- cnf_set1 ; k <- cnf_set2 ]

cnf_implies  cnf_set1  cnf_set2  = cnf_or (negate cnf_set1) cnf_set2
```

The final task is to define the attribute functions 'att_cnf_implies', 'att_cnf_and', etc. in terms of the operations above :

```
att_cnf_implies  [(CLSET x), (CLSET y)] = CLSET(cnf_implies x y)

att_cnf_and   [(CLSET x), (CLSET y)]    = CLSET (cnf_and x y)

att_cnf_or    [(CLSET x), (CLSET y)]    = CLSET (cnf_or x y)

makeclset     [(PNAME x)]               = CLSET [[x]]

same_as   [x]                           = x
```

3.6. Sample output

```
Miranda wff [ ] (words  "((p implies q) -> (r implies (s and t))).")

[([CLSET [["p","-r","s"],["p","-r","t"],["-q","-r","s"],["-q","-r","t"]]],[ ])]
```

3.7. Comments on this example

If code for the functions 'cnf_and', 'cnf_or', etc. had not been available, the application programmer would have to either look for an 'expert' or switch mode to become a programmer in clausal propositional calculus.

If the problem had been to 'evaluate' the input expressions, the assignment of truth values to the propositional variables 'p', 'q', etc. could have either been passed in as a context, ie. as an inherited attribute, or else the 'dictionary' could have included 'value' attributes in the 'propvar_list'. The only part of the interpreters for non-basic categories that would be different from the above, would be the attribute synthesis rules.

4. MULTI - INPUT PARSING

The two examples that we have given are well suited to the approach that we are proposing since they both have an obvious language processing component. However, the approach can also be used to advantage for other types of problem. For example, we can easily extend the approach to accommodate standard data processing applications such as 'updating sorted master files by running sorted transaction files against them'. The extension that we make is based upon a new concept of 'muiping', ie. multi-input parsing.

We illustrate the technique with a simple example : merging two sorted sequences of 'numwords'. The required function is to return a result such as (NUMLIST [1, 2, 2, 3, 4]) when applied to the input pair ("one two three.", "two four."). Since this is a simple example, we present the solution with little explanation. The function 'numword' is as above. The function 'numseq' is similar to 'wordseq' above but returns a list of numbers as result when applied to input such as "one two." The function 'wordsmerge' is a multi-input interpreter.

An example application of 'wordmerge' is :

```
Miranda wordmerge [] (words "one two.", words "one three.")

[([NUMLIST [1, 1]],        (["two", "."], ["three", "."]),
 ([NUMLIST [1, 1, 3]],     (["two", "."], ["."]),
 ([NUMLIST [1, 1, 2]],     (["."], ["three", "."]),
 ([NUMLIST [1, 1, 2, 3]],  (["."], ["."])]
```

Once again, we see that the function that we have constructed does more than we may have anticipated. The four results returned by 'wordmerge', when applied in the empty context (ie []) to the pair of inputs ("one two.", "one three."), correspond to the four ways in which these two inputs can be interpreted as 'numseqs' and subsequently merged. If we were only interested in the last of these results, we could change the interpreter 'numseq' such that it only returned an interpretation for sequences of words followed by a terminator such as '.'.

```
| AN EXAMPLE OF MUIPING

numseq =
 (consists_of [numword]
     with_att_syn_rules
     [("numlist","is",convert_to_numlist,[("numval",fst_part)])]
     with_inh_att_calc_rules
     [[]]
     with_failure_rules
     [])
 $orelse
 (consists_of [numword, numseq]
     with_att_syn_rules
     [("numlist","is",create_numlist_from,[("numval",fst_part),
           ("numlist",snd_part)])]
     with_inh_att_calc_rules
     [[],[]]
     with_failure_rules
     [])

wordmerge =
 mmuip [numseq, numseq]
     with_att_syn_rules
     [("numlist","is",att_merge,[("numlist",first_muip),
                                 ("numlist",second_muip)])]
     with_inh_att_calc_rules
     [[]]
     with_failure_rules
     []

|functions for the standard calculi part

convert_to_numlist  [NUMVAL x]              = NUMLIST [x]

create_numlist_from  [NUMVAL x, NUMLIST y] = NUMLIST (x : y)

att_merge  [NUMLIST x, NUMLIST y]      = NUMLIST (merge x y)

merge  []  y          = y
merge  x   []         = x
merge (x:xs) (y:ys)   = x: merge xs (y:ys)   , x < y
                      = y: merge (x:xs) ys   , otherwise
```

5. HOW THE PROPOSED APPROACH FACILITATES THE USE OF PROGRAMMING TECHNIQUES IN APPLICATION PROGRAMMING

5.1. Examples of programming techniques

The motivation for the work, of which this paper is a part, is that more structured approaches to application programming will facilitate the use of 'programming techniques'. Examples of the type of technique to which we are referring, are :

(i) Information hiding and use of modularity.
(ii) Use of data typing, as discussed in Danforth and Tomlinson (1988).
(iii) Declarative programming and use of stepwise refinement, as discussed in Mili, Desharnais, and Gagne (1986).

(iv) 'Pipeline' or 'unary function' programming, as advocated by Dijkstra (1975).
(v) Use of higher order functions and creation of 'generic' functions through 'abstraction'.
(vi) Program development through transformation, as discussed in Burstall and Darlington (1977), and Bauer et. al (1988).
(vii) Using 'lazy' evaluation to advantage, as discussed in Henderson (1982).
(viii) Derivation of programs from proofs as discussed in Manber (1988).

A detailed discussion of these techniques, examples of their use in Miranda, and their relevance to the concept of 'application programming as the construction of language interpreters' can be found in Frost (1989).

5.2. Use of techniques in the 'standard calculi' part of application programming

It is clear that many of the programming techniques listed above are relevant to those aspects of application programming that involve the construction of code to carry out operations in standard calculi. The advantage of using a higher order, lazy, pure functional programming language for this aspect of application programming is that such languages 'support' most of these programming techniques. Also, the attribute grammar structure helps the programmer to separate the language processing and the 'standard calculi' components of the problem, and thereby facilitates the use of programming techniques in the latter. For instance, consider the example in section 3. The functions of 'clausal propositional calculus' are quite clearly separated from the 'language processing' part of the problem. If functions such as 'conv_from_dnf_to_cnf' had not been available, then they could have been designed from scratch using techniques such as 'programs from proofs'. The programmer could have carried out this aspect of the task quite independently of the language processing part.

It should be noted that the approach that we have suggested in this paper does not require that the 'standard calculi' parts of the problem always be implemented in a language such as Miranda. There is nothing to prevent the attribute grammar interpreter from interacting with processes involving the execution of programs written in other languages. The implementation languages available may restrict the use of certain techniques to some extent, but in many cases programming techniques can be used in the design stage even though the implementation languages do not support them directly.

5.3. Use of concepts and techniques in the 'language processing' part of application programming

Our claim that the proposed approach will facilitate the use of programming techniques in all aspects of application programming, can only be put to the test through use of the approach in real applications. The following observations suggest that some such experimentation may be justified :

(i) The approach supports 'modular' programming in several respects : the language processing part is separated from the part involving computation in standard calculi, the attribute grammar formalism breaks the problem down into distinct parts corresponding to symbols in the grammar, and the 'pure functional' nature of the approach prevents 'modules' interacting other than through their interfaces.
(ii) The strong typing of Miranda is complemented by use of attribute constructors such as 'NUMVAL', 'WDSEQVAL', etc. The requirement to identify, name, and define functions on attributes provides additional constraints to guide programmers.
(iii) The approach is declarative. Programmers using the approach will derive all of the benefits of declarative programming.
(iv) The approach is 'higher order' and allows 'partial application'. Consequently, 'pipeline' programming is supported. For example, suppose that we have (a) a function 'first_res' that returns the first value from the first result returned by an interpreter, (b) a function 'trans_resolve' which forms the transitive closure of a clause set with respect to the 'resolution' rule of inference, and (c) a function 'empty_clause' which returned the value 'True' if a clause set given as argument contains the empty clause, and 'False' otherwise. We could use these functions, together with an enhanced function like 'wff', that also accommodated negation, to construct a function that checked arbitrary expressions of propositional calculus for consistency :

```
inconsistent = empty_clause . trans_res . first_res . (uncurry wff)
```

The function 'uncurry' is necessary since 'wff' expects two arguments, a context and a list of words. An example of the use of 'consistent' is :

```
inconsistent ([], (words "(p and (p -> q) and -q)."))  => True
```

(vi) The approach also supports 'abstraction'. For example, we could 'abstract out' the attribute rules from any interpreter. This would give us 'shell' interpreters for particular input languages. For example, we could do this for the example interpreters in section 3. We could then parameterize these shell in various ways to obtain a 'convert_to_clause_form' interpreter, a 'boolean evaluator' interpreter, and so on.
(vi) Owing to the fact that the implementation language is 'lazy', the interpreters that we build are also lazy. The fact that interpreters are able to return all possible interpretations does not mean that they will necessarily be inefficient if only one interpretation is required. Careful

design of grammars may help to make good use of the lazy evaluation strategy.
(vii) There is a relationship between the structure of an attribute grammar and the attribute synthesis rules used in it. For example, in section 2, the way in which 'WDSEQVAL's of complex 'wordseq's are computed is determined by the structure of the grammar. In fact, the specification of the function 'calc_wdseqval' could have been determined in conjunction with the specification of the initial grammar for 'numword' and 'wordseq' given in section 2.1. The specification could have been derived using the technique of 'programs from proofs'. Making the grammar of the input data explicit, simplifies the use of this technique.

6. CONCLUDING COMMENTS

The approach that we have presented would benefit greatly from a syntax-directed editor, that could both guide programmers and fill in parts of the syntax. We are currently building such an editor. We are also testing the approach by using it to construct a sophisticated knowledge base system that can answer complex queries expressed in a large non-modal, non-intensional, subset of english.

7. ACKNOWLEDGEMENTS

The author acknowledges the assistance of N.S.E.R.C. of Canada, and the support provided by The University of Windsor. Thanks also go to John Launchbury of The University of Glasgow, and Walid Saba of The University of Windsor, both of whom contributed to the design of the higher order functions used in the approach.

8. REFERENCES

Backus, J. W. (1978) 'Can programming be liberated from the von Neumann style? A functional style and its algebra of programs'. *CACM* **21 (8),** 613 - 641.

Bauer, F. L., Moller, B., Partsch, H. and Pepper, P. (1989) 'Formal program construction by transformation - Computer-aided, intuition-guided programming.' *IEEE Trans. Software Eng.*, **15 (2),** 165 - 179.

Bochmann, G. V. (1976) 'Semantic evaluation from left to right', *CACM* **19 (2),** 55 - 62.

Burstall, R. M. and Darlington J. (1977) 'A transformation system for developing recursive programs.' *JACM* **24 (1).**

Cardelli, L., and Wegner, P. (1985) 'On understanding types, data abstraction, and polymorphism', *ACM Computing Surveys,* **17 (4),** 471 - 522.

Danforth, S. and Tomlinson, C. (1988) 'Type theories and object-oriented programming', *ACM Computing Surveys,* **20 (1),** 29 - 72.

Dijkstra, E. W. (1975) 'Guarded commands, non-determinacy, and the formal derivation of programs', *CACM* **18 (8),** 453 - 457.

Edupuganty, B. and Bryant, B. R. (1989) 'Two-level grammar as a functional programming language', *The Computer Journal,* **32 (1),** 36 - 44.

Farrow, R. (1982) 'LINGUIST 86 : Yet another translator writing system based on attribute grammars', *ACM SIGPLAN NOTICES,* **17 (6),** 160 - 171.

Frost, R. A. (1989) 'The use of attribute grammars in application programming', *Technical Report 89-001,* School of Computer Science, University of Windsor, Canada.

Hehner, E. C. R., and Silverberg, B. A. (1983) 'Programming with grammars : an exercise in methodology-directed language design'. *The Computer Journal* **26 (3),** 227 - 281.

Henderson, P. (1980) *Functional Programming : Application and Implementation,* Prentice-Hall.

Johnsson, T. (1987) 'Attribute grammars as a functional programming paradigm', *Springer Lecture Notes 274,* 155 - 173.

Knuth, D. E. (1968) 'Semantics of context free languages', *Mathematical Systems Theory,* **2 (2),** 127 - 146.

Manber, U. (1988) 'Using induction to design algorithms', *CACM* **31 (11),** 1300 - 1313.

Manna, Z., and Waldinger, R. (1977) *Studies in Automatic Programming Logic.* North Holland, Amsterdam.

Mili, A., Desharnais, J. and Gagne, J. R. (1986) 'Formal models of stepwise refinement of programs.' *ACM Computing Surveys* **18 (3),** 231 - 276.

Reps, T. W. and Teitelbaum, T. (1988), *The Synthesizer Generator.* Springer-Verlag.

Turner, D. (1985) 'A non-strict functional language with polymorphic types', *Proc. IFIP Int. Conf. on Functional Programming Languages and Computer Architecture,* Nancy, France. Springer Lecture Notes in Computer Science vol. 201.

Watt, D. A. (1979) 'An extended attribute grammar for Pascal', *ACM SIGPLAN NOTICES,* **14 (2),** 60 - 74.