SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 23(10), 1139–1156 (OCTOBER 1993)

Guarded Attribute Grammars

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

SUMMARY

Contrary to a widely-held belief, it is possible to construct executable specifications of language processors that use a top-down parsing strategy and which have structures that directly reflect the structure of grammars containing left-recursive productions. A novel technique has been discovered by which the non-termination that would otherwise occur is avoided by 'guarding' top-down left-recursive language processors by non-left-recursive recognizers. The use of a top-down parsing strategy increases modularity and the use of left-recursive productions facilitates specification of semantic equations. A combination of the two is of significant practical value because it results in modular and expressively clear executable specifications of language processors. The new approach has been tested in an attribute grammar programming environment that has been used in a number of projects including the development of natural language interfaces, SQL processors and circuit design transformers within a VLSI design package.

KEY WORDS: Top-down parsing Left-recursive grammar Attribute grammar Functional language

INTRODUCTION

It is widely believed that it is impossible to build top-down parsers with structures that directly reflect the structure of grammars containing left-recursive productions. If this were true, it would result in an undesirable constraint on the construction of executable specifications of language processors for the following reasons:

1. As discussed by Koskimies,[1] use of a top-down parsing strategy results in more modular language processors than can be obtained using alternative methods.
2. In many cases, language specifications that are defined with respect to left-recursive grammars are more natural to construct and easy to reason about than are their non-left-recursive counterparts.

If top-down parsing and left-recursive productions were incompatible, one would have to forfeit modularity for naturalness of expression, or vice versa, when constructing executable specifications of language processors. Fortunately, as we shall show in this paper, top-down parsing and left-recursive productions can co-exist.

The technique that we have discovered involves an extension to a method of building executable specifications of language processors first described by Burge[2] in 1975. One limitation of the method proposed by Burge, and to modifications that have been made to it since 1975, is that it cannot be used to create executable specifications of language processors which have structures that directly reflect the structure of grammars containing left-recursive productions. Our extension overcomes

0038–0644/93/101139–18$14.00
Received 21 April 1992
Revised 14 March 1993

this limitation. We have discovered a novel technique by which the non-termination that would otherwise occur is avoided by 'guarding' top-down left-recursive language processors by non-left-recursive recognizers. Our technique enables the construction of executable specifications of language processors that use a top-down parsing strategy and which have structures that directly reflect the structure of grammars containing left-recursive productions. The use of a top-down parsing strategy increases modularity and the use of left-recursive productions facilitates specification of semantic equations. A combination of the two is of significant practical value because it results in modular and expressively clear executable specifications of language processors. Our technique can be described informally as follows:

1. Language processors are implemented as executable specifications of attribute grammars. Each production in the attribute grammar is implemented directly as a syntax-directed evaluator using classical top-down parsing with full back-tracking.
2. Each executable production is a function: given some input the function returns some sort of value as its result. The value is paired with the tail of the input stream so that subsequent language processors can be applied at the point at which the first left off. If the grammar is ambiguous, more than one value may need to be returned, and there must be a mechanism for returning no value if the input is not recognized. We can satisfy these requirements as follows: each language processor returns a list of results. The list may be empty indicating failure (this notion was first described by Wadler[3] in 1985), or else it may contain one or more successful interpretations.
3. Each left-recursive production p is 'guarded' by a non-left-recursive recognizer r that recognizes the same language as p. This guarding is denoted by r $enables p. In very general terms, the guarding recognizer r enables or disables application of the executable left-recursive production p.

We begin by describing the technique formally with respect to recognizers rather than executable attribute grammars. Readers should note that the use of top-down left-recursive recognizers would serve no purpose whatsoever. We introduce them only as an aid to explanation. The technique that we describe with respect to recognizers can be extended readily to the domain of executable attribute grammars. Later in the paper, we describe an attribute-grammar programming environment that incorporates the technique and which has been used to build various language processors. We conclude with a brief analysis of the approach and an overview of current work.

RECOGNIZERS

In this section, we provide a formal definition of the set of recognizers. The formal definition is used in the third section to define the set of guarded recognizers.

In order to define precisely what we mean by a recognizer, we introduce the set type in Figure 1.

We can now define the type recognizer as follows:

```
terminal == char
recognizer == [([terminal], [terminal])] -< [([terminal], [terminal])]
```

The set `type` is defined as follows :

```
char ∈ type, where char is the type of characters.
if t ∈ type then so is [t], the type of lists whose elements are of type t.
if t1,..,tk ∈ type then so is (t1,..,tk), the type of tuples with elements of type t1,..,tk.
if t1, t2 ∈ type then so is t1 -> t2, the type of functions with arguments in t1 and results in t2.
```

`x == y` **introduces** `x` **as an acronym for the type name** `y`.

`x :: y` **declares** `x` **to be of type** `y`.

Figure 1. Definition of the set type

That is, a recognizer is a function mapping lists of pairs to lists of pairs. We shall see later that the first component of each input pair is a list of terminals that have been consumed so far by the recognition process, and the second component of each input pair is a list of terminals yet to be processed. For each pair (done, rest) in its input list, a recognizer returns zero or more pairs depending on the number of ways in which it can recognize initial segments of the terminal list rest.

We define the set of recognizers formally in terms of the evaluation of a set of recursive definitions. We begin by defining the set of recognizer expressions rec_expressions in Figure 2.

1) `terminal c ∈ rec_expressions` **for all** `c ∈ characters`, **where** `characters` **is the set of characters.**
2) **The Kleene closure of** `characters` **is a subset of** `rec_expressions` **(character strings are used as variables).**
3) **If** `x,y ∈ rec_expressions`, **then so are** `x $orelse y` **and** `x $then y`.

Figure 2. Definition of the set rec_expressions

No expressions other than those that can be constructed by means of a finite number of applications of rules 1 to 3 above are in the set of rec_expressions.

We now define the set of *recursive definitions of recognizers* in Figure 3.

If `e1, . . . , ek` **are rec_expressions containing no variables other than those in the set** `{f1, . . . , fk}`,
then the set of definitions : `{f1 = e1, . . . , fk = ek}`
is a set of simultaneous recursive definitions of the recognizers `f1, . . . . , fk`.

Figure 3. Definition of the set of recursive definitions of recognizers

No definitions, other than those that can be constructed according to the above rule, are in the set of recursive definitions of recognizers. Examples are given in Figure 4. The definition of silly_ex in Figure 4 is included for reasons that will become apparent in the third section.

We now define the set of *recognizers* as that set containing only those functions induced by recursive definitions of recognizers under normal-order reduction with the functions recognize, orelse and then defined as in Figure 5.

Normal-order reduction is a strategy in which no argument to a function, or subexpression of an expression, is evaluated unless its value is known to be required. Readers are referred to Reference 4 and Reference 5 (Appendix C) for formal descriptions of normal-order reduction.

```
a            = recognize 'a'
b            = recognize 'b'
c            = recognize 'c'
plus         = recognize '+'
times        = recognize '*'

var          = a $orelse b $orelse c
ex           = var $orelse summ $orelse productt
summ         = var $then plus $then ex
productt     = var $then times $then ex

ex'          = var
               $orelse (ex' $then plus  $then ex')
               $orelse (ex' $then times $then ex')

silly_ex     = silly_ex
```

Figure 4. Examples of recursive definitions of recognizers

```
recognize           :: terminal -> recognizer
recognize c input   = concat (map test_for_c input)
                      where
                      test_for_c (any, t:ts) = [([t],ts)], if t = c
                      test_for_c      any    = []

orelse              :: recognizer -> recognizer -> recognizer
(x $orelse y) input   = x input ++ y input

then               :: recognizer -> recognizer -> recognizer
(x $then y) inp = [(v ++ v',inp2) | (v,inp1) <- x inp; (v',inp2) <- y [([],inp1)]]
```

Figure 5. Definitions of the functions recognize, orelse *and* then

The notation of the functional programming language Miranda* is used throughout this paper. This allows the new technique to be described concisely and formally. It also provides readers with executable specifications with which they can experiment (all definitions of functions given in this paper are executable). In the following, we provide a description of the semantics of the Miranda notation that should be sufficient for the reader to understand the novel technique. A more complete description of the semantics of Miranda can be found in References 6 and 7.

Miranda is a higher-order language—functions are first-class citizens and can be passed as parameters and returned as results. Every function of two or more arguments is a higher-order function: if one argument is supplied, the result is a function of the remaining arguments. In Miranda notation f x denotes application of the function f to the value x, function application has higher precedence than any operator, ++ is an infix operator that appends lists, concat is a unary function that takes a list x of lists as argument and returns a list obtained by appending all the elements of x together, and map is a higher-order binary function that applies the function given as first argument to each element of the list given as second argument. Lists are denoted by square brackets.

In Miranda notation, functions are defined by equations:

1. In Figure 5, we have defined the function recognize using an equation that

* Miranda is a trademark of Research Software Ltd.

includes a local definition of a supplementary function test_for_c that is introduced through a where clause.

2. The definition of test_for_c uses pattern matching: the two defining equations are distinguished by the use of different patterns in the formal parameters. The pattern (any, t:ts) matches any binary tuple with an arbitrary first element and whose second element is a non-empty list. The pattern t:ts matches any non-empty list l letting t stand for the first element of l and ts stand for l minus the first element. The pattern any matches any value.
3. The definition of test_for_c also uses a guarded equation—the guard is written on the right following the comma and the word if. Use of pattern matching and guards allows one to define functions by cases. In order to determine which equation is applicable when a function is applied to some argument(s), the equations are examined in order from top to bottom until one is found whose pattern and guards, if any, are satisfied. That equation is then used as the defining equation for the application of the function.
4. The $ sign in $f indicates that the function f is being used as an infix operator.
5. The construct used on the right-hand side of the defining equation of the function then in Figure 5, is called a list comprehension and is modelled on Zermelo–Frankel set comprehensions. List comprehensions provide a concise syntax for iterations over lists. The general form of a list comprehension is [body | qualifiers], where each qualifier is either a generator of the form x >– y denoting that the variable x ranges over all elements of the list y, or a boolean expression used to restrict the ranges of the variables introduced by the generators. Qualifiers are separated by semicolons.

The following informal descriptions of the higher-order functions recognize, orelse and then are given to aid understanding of the formal definitions in Figure 5:

1. The function recognize takes two arguments, a character and a list of pairs. The result returned is computed by appending the results obtained by mapping the function test_for_c over the list of pairs and concatenating the results. The function test_for_c when applied to a pair returns a non-empty list if the second element of the pair is recognized, and an empty list otherwise.
2. The function orelse is used as an infix operator that takes two recognizers x and y as operands and returns an ‘alternative recognizer’ as result. When the alternative recognizer is applied to some input, it returns a result that is obtained by appending together the results returned by separate application of x and y to the input.
3. The function then is used as an infix operator that takes two recognizers x and y as operands and returns a ‘composite recognizer’ as result. The composite recognizer when applied to some input returns the list of results obtained by applying y to each of the results returned by application of x to the input. then can be thought of as a recognizer composition operator.

Recognizers that are built using recognize, orelse and then use a top-down fully-backtracking search strategy and return lists of results, one result for each way in which each element of the input can be recognized. For example, consider the following applications of the recognizers from Figure 4, where =< denotes normal-order reduction:

a	[([], “a+b”)]	=>	[(“a”, “+b”)]
a	[([], “a+b”), ([], “a*b”)]	=>	[(“a”, “+b”), (“a”, “*b”)]
a	[([], “b+c”)]	=>	[]
summ	[([], “a+b”), ([], “a*b”)]	=>	[(“a+b”, “”)]
ex	[([], “a*b+c”)]	=>	[(“a”, “*b+c”), (“a*b”, “+c”), (“a*b+c”, “”)]]
ex′	[([], “a+b”)]	=>	[(“a”, “+b”), (“a”, “*b”), (“a+b”, “”), ⊥,
ex′	[([], “xxx”)]	=>	[⊥

Note that when a is applied to the input [([], “b+c”)] the result is an empty list. This is because the string “b+c” does not commence with an ‘a’, Note also that application of the ‘left-recursively defined’ recognizer ex′ enters a non-terminating recursive descent, denoted by ⊥, both for input that it recognizes and for input that it fails to recognize. In the next section, we formalize the notion of left-recursion and describe how non-termination that results from it can be avoided.

GUARDED RECOGNIZERS

We now formalize the notion of guarded recognizer. We begin by defining the relation direct derivative in Figure 6.

```
rec_expressions, as follows :


For all α,β,γ,ρ,χ ∈ rec_expressions,

   ρ is a direct derivative of χ, denoted by χ --> ρ
           1) if χ = ρ ∈ G
        or 2) if ρ is_an_alternative_in χ
        or 3) if α = γ ∈ G
               and  β is_an_alternative_in γ
               and there exist rec_expressions φ1, φ2 such that :
                    χ = φ1 $then α  and ρ = φ1 $then β
                 or χ = α  $then φ1 and ρ = β  $then φ1
                 or χ = φ1 $then α $then φ2 and ρ = φ1 $then β $then φ2
where
        x is_an_alternative_in x               = True


        x is_an_alternative_in (y $orelse z)  = (x = y) or (x is_an_alternative_in z)
```

Figure 6. Definition of the relation direct derivative

We say that ρ is a *derivative* of χ, denoted by χ --<* ρ, if (χ, ρ) is in the reflexive transitive closure of --<. For example, with respect to the definitions given in Figure 4:

ex --<* var $then plus $then var

A definition f = x is a *left-recursive definition* if f --<* f or f --<* f $then α for any α. A recognizer that is defined by a left-recursive definition, is a *left-recursive recognizer*. A rec_expression ρ is a *left-recursive alternative* of f = x if ρ is_an_alternative_in x and ρ --<* f α for some α.

ρ is a *fully-expanded derivative* of a recognizer χ if χ --<* ρ and ρ contains no variables or instances of the word $orelse. For example, recognize ‘a’ $then recognize

'+' $then recognize 'a' is a fully-expanded derivative of the recognizer ex from Figure 4.

The *language recognized* by a recognizer χ, denoted by L(χ), is the set of character strings obtained by removing the words recognize and $then from the fully-expanded derivatives of χ. For example, with respect to the definitions in Figure 4, "a+a" $\in$ L(ex), and "a*b"$\in$ L(var $then times $then var).

```
The set of recursive definitions of guarded recognizers is obtained from the set of recursive definitions of recognizers :

1)  All non left-recursive definitions remain the same.
2)  All definitions, f = e, where f -->* f, are removed.
3)  Each left-recursive definition f = e is modified as follows :
           f = e is replaced by a guarded left-recursive definition f = r $enables e'
                 where
                 r is a non left-recursive recognizer such that L(r) = L(e)
                 and e' is obtained from e by replacing each left-recursive alternative ρ in e by :
                                 p $enables ρ
                                 where
                                 p is a non left-recursive recognizer such that L(p) = L(ρ)
```

Figure 7. Definition of the set of recursive definitions of guarded recognizers

The set of *recursive definitions of guarded recognizers* is defined in Figure 7.

Figure 8 contains recursive definitions of guarded recognizers that have been obtained from those in Figure 4 using the procedure described in Figure 7. In particular, (1) all non-left-recursive definitions remain the same, (2) the definition of silly_ex has been removed because silly_ex --<* silly_ex, and (3) the definition of g_ex′ has been obtained from the left-recursive definition of ex′ in Figure 4 using the modification process described in Figure 7. This modification required two additional non-left-recursive recognizers to be defined, summ′ and productt′.

We now define the set of *guarded recognizers* as that set containing only those functions induced by recursive definitions of guarded recognizers under normal-order reduction with the functions recognize, orelse and then defined as in Figure 5, and

```
a          = recognize 'a'
b          = recognize 'b'
c          = recognize 'c'
plus       = recognize '+'
times      = recognize '*'

var        = a $orelse b $orelse c

ex         = var $orelse summ $orelse productt

summ       = var $then plus  $then ex

productt   = var $then times $then ex

summ'      = ex  $then plus  $then ex

productt'  = ex  $then times $then ex

g_ex'      = ex $enables
                (var
                 $orelse (summ'     $enables (g_ex' $then plus  $then g_ex'))
                 $orelse (productt' $enables (g_ex' $then times $then g_ex')))
```

Figure 8. Examples of recursive definitions of guarded recognizers

the function enables defined as in Figure 9. Note that -- is list subtraction, and # is a unary function that returns the length of the list given as argument.

The following informal description of the function enables is given to aid understanding of the formal definition in Figure 9.

The function enables is used as an infix operator that takes two recognizers r and i as operands and returns a new recognizer as result. The new recognizer works as follows: when applied to some input, it begins by applying the 'guarding' recognizer r. The results returned by r are sorted into ascending order by length of recognized segment of the input. This ordering is done by the function set_sort, which also removes duplicate results. The ordered sequence of results is then processed in order by a recognizer i $then recognize '@'. This composite recognizer is applied to an input [([], done ++ ['@'])] for each result (done, rest) in the ordered sequence. Each application of i $then recognize '@' returns a list of results. If the list is empty, an error is reported that the guarding recognizer r is wrong in the sense that it has recognized some input that is not recognized by i. If the recognizer is not in error, a list of results is returned. The first element of each of these lists is then processed by the function fix to remove the character '@' and to add the rest of the input back to the sequence of terminals yet to be processed. The purpose of adding the character '@' to the end of each result done returned by r and then using the recognizer i $then recognize '@', is to ensure that each of the results returned has been produced by consuming all of the terminals in done. This, together with the ordering of the results returned by r, ensures that the results returned by r $enables i are a non-redundant sequence ordered by length of terminals consumed. This is required to guarantee termination when i is a left-recursive recognizer.

The following are examples of applications of the left-recursive guarded recognizer g_ex′ from Figure 8.

```
g_ex′  [([],  "a+b")]     =<  [("a", "+b"), ("a", "*b"), ("a+b", "")]
g_ex′  [([],  "xxx")]     =<  []
```

Experimentation with the technique suggests that normal-order reduction of any application of a guarded recognizer will terminate if the input is finite. However, we have not yet established a formal proof of this termination property. We anticipate that the establishment of such a proof will be difficult owing to the context-sensitive

```
enables            :: recognizer -> recognizer -> recognizer

(r $enables i) inp = [fix (newint [([],done ++ ['@'])]) rest
                                     |(done,rest) <- set_sort (r inp)]
                      where
                      fix     []             rest    = error ("wrong recognizer")
                      fix   ((d,r) : more)   rest    = (d -- ['@'],r ++ rest)
                      newint = i $then recognize '@'

set_sort   []       = []
set_sort  (a:as)    = set_sort [e | e <- as ; (# (fst e)) < (# (fst a))]
                      ++ [a] ++
                      set_sort [e | e <- as ; (# (fst e)) > (# (fst a))]
                      where
                      fst (a, b) = a
```

Figure 9. Definition of the function enables

aspect of the definition of the set of recursive definitions of guarded recognizers given in Figure 7.

AN APPLICATION OF THE TECHNIQUE

The technique that we have described can be extended readily to the domain of executable specification of attribute grammars. In order to do this, the functions orelse etc. are replaced by functions that carry out attribute computation in addition to parsing. The concept of 'guarding' is now of practical value. The language processor r in an expression r $enables i is still a recognizer, but the processor i is an interpreter constructed as an executable specification of an attribute grammar. The use of guarding allows i to have left-recursive productions. Therefore, although we have to convert left-recursive grammars to equivalent non-left-recursive form (in order to obtain definitions of the guarding recognizers) if we wish to use a top-down parsing strategy, we do not have to convert the attribute rules associated with these left-recursive grammars. This is of significant practical value owing to the fact that in many cases it is considerably easier to specify semantic rules associated with a left-recursive grammar than it is to specify the semantic rules associated with an equivalent non-left-recursive grammar. We now describe a programming environment that makes use of the new technique in this way.

W/AGE

The Windsor Attribute Grammar programming Environment[8] (W/AGE) is an environment that allows language processors to be implemented as executable specification of attribute grammars. To construct a language processor one simply specifies the syntax and semantics of the language using a slight variant of a commonly-used notation for attribute grammars. Fully general attribute dependencies are allowed with the exception that 'inheritance from the right' may only be used with unambiguous grammars. The most recent version of W/AGE accommodates left-recursive productions using the technique described in this paper.

Readers who are unfamiliar with attribute grammars and notions such as 'inheritance from the right' should be able to read the following and obtain a sufficient understanding of how W/AGE can be used to build executable specifications. Those who would prefer more background on attribute grammars are referred to Reference 9.

Each attribute grammar production that is specified in W/AGE is a function that maps a list of inputs to a list of results. Each of the inputs consists of a tuple containing a list of terminals together with a list of context attributes. Each result consists of a tuple containing a list of terminals not yet consumed together with a list of result attributes.

W/AGE consists of a number of definitions of higher-order functions including definitions of the functions uninterpreted, orelse, recognized, structure and enables, that are derived from the functions recognize, orelse, then and enables described earlier in this paper. W/AGE uses normal-order evaluation. The most recent version of W/AGE is constructed as a set of functions that have been added to the standard environment of Miranda.

An example use of left-recursive productions in W/AGE

The example given in Figure 10 is a complete W/AGE program except for the definitions of the functions orelse, etc. which are included through the first insert command at the top of the Figure, and the definition of the executable attribute grammar number which is included through the second insert command. A detailed explanation of the program is as follows.

The W/AGE program consists of five components:

1. PRELIMINARIES. In addition to the insert commands, this component of the program contains a type declaration for attributes that are to be computed. In this example, there is only one type of attribute—VAL attributes. The declaration states that attributes of type VAL (this is a minor misuse of terminology) are constructed from values of the base type num by applying the 'constructor' VAL to them. The two equations at the end of the preliminaries section are used by W/AGE's lexical scanner.
2. BASIC ATTRIBUTE GRAMMARS. This component of the program contains executable specifications of attribute grammars for the terminal symbols of the language. For example, the first equation states that **plus** is a language processor that recognizes the special symbol terminal "+" but does not compute any attribute for it, i.e. it is uninterpreted. In other applications, an alternative form of definition might be used that allows specification of attributes associated with terminal symbols. W/AGE also contains functions that enable various kinds of attribute grammars to be specified for set of non-terminals such as the set of identifiers, reals, integers, etc. The attribute grammar number that is included in the example program makes use of such functions.
3. ATTRIBUTE GRAMMARS FOR THE NON-TERMINALS. This component of the program contains executable specifications of attribute grammars for the non-terminals of the language. That part of the specification given in bold font in Figure 10 is simply a BNF-like definition of the syntax of the language:

```
plus    : : =   "+"
ext     : : =   ex          period
ex      : : =   number
              | opbr        ex          clbr
              | ex          plus        ex
              | ex          times       ex
              | minus       ex
```

The W/AGE notation is to be read as, for example, 'an ex is a number or else it is a structure consisting of a substructure s1 that is an opbr followed by a substructure s2 that is an ex followed by a substructure s3 that is a clbr', or else etc. Left-recursive productions are guarded in a similar manner to that introduced in the third section. The guarding recognizers are defined in the lower half of Figure 10. That part of the specification given in italic font consists of the semantic rules that describe how attributes are related to each other. For example the rule *c_rule 1 (VAL $u lhs) EQ (VAL $u s1)* is a copy rule stating that the VAL attribute of the lhs, i.e. of the ext on the left-hand side of the equation, is a copy of the VAL attribute of the substructure s1, i.e.

```
|| PRELIMINARIES

%insert <local/header_for_WAGE_VERSION_2_RELEASE_0.m>
%insert <local/number_interpreter_for_WAGE_VERSION_2_RELEASE_0.m>
attribute     ::=  VAL num
reserved_words  = []
special_symbols = [".","+","*","-","(",")"]

||*****************************************************************************************
||  BASIC ATTRIBUTE GRAMMARS

plus        = uninterpreted   (SPECIAL_SYMBOL_TERM "+")
times       = uninterpreted   (SPECIAL_SYMBOL_TERM "*")
minus       = uninterpreted   (SPECIAL_SYMBOL_TERM "-")
period      = uninterpreted   (SPECIAL_SYMBOL_TERM ".")
opbr        = uninterpreted   (SPECIAL_SYMBOL_TERM "(")
clbr        = uninterpreted   (SPECIAL_SYMBOL_TERM ")")

||*****************************************************************************************
||  ATTRIBUTE GRAMMARS FOR THE NON-TERMINALS

ext = structure (s1 ex ++ s2 period)
      [c_rule 1 (VAL $of lhs) EQ (VAL $of s1)]

ex  = rec_ex
      $enables
         (number
          $orelse structure (s1 opbr    ++ s2 ex ++ s3 clbr)
                               [c_rule 2 (VAL $of lhs) EQ (VAL $of s2)]
          $orelse (rec_sum
                   $enables structure (s1 ex ++ s2 plus    ++ s3 ex)
                               [a_rule 3 (VAL $of lhs) EQ   add[VAL $of s1, VAL $of s3]])
          $orelse (rec_sub
                   $enables structure (s1 ex ++ s2 minus   ++ s3 ex)
                               [a_rule 4 (VAL $of lhs) EQ subtr[VAL $of s1,VAL $of s3]])
          $orelse (rec_product
                   $enables structure (s1 ex ++ s2 times   ++ s3 ex)
                               [a_rule 5 (VAL $of lhs) EQ mult[VAL $of s1, VAL $of s3]])
          $orelse structure (s1 minus   ++ s2 ex)
                               [a_rule 6 (VAL $of lhs) EQ negate[VAL $of s2]])

add       [VAL x, VAL y] = VAL (x + y)
subtr     [VAL x, VAL y] = VAL (x - y)
mult      [VAL x, VAL y] = VAL (x * y)
negate    [VAL x]        = VAL (- x)
||*****************************************************************************************
||  NON left-recursive RECOGNISERS THAT ARE USED AS GUARDS ABOVE  .

rec_ex          = recognised (s1 number)
                  $orelse  recognized (s1 opbr    ++ s2 rec_ex ++ s3 clbr)
                  $orelse  recognized (s1 number ++ s2 sum_sub_prod)
                  $orelse  recognized (s1 opbr    ++ s2 rec_ex ++ s3 clbr   ++ s4 sum_sub_prod)
                  $orelse  recognized (s1 minus  ++ s2 rec_ex)

sum_sub_prod  = recognized (s1 plus  ++ s2 rec_ex)
                  $orelse   recognized (s1 minus  ++ s2 rec_ex)
                  $orelse   recognized (s1 times  ++ s2 rec_ex)

rec_sum         = recognized  (s1 rec_ex ++ s2 plus  ++ s3 rec_ex)
rec_sub         = recognized  (s1 rec_ex ++ s2 minus ++ s3 rec_ex)
rec_product     = recognized  (s1 rec_ex ++ s2 times ++ s3 rec_ex)

||*****************************************************************************************
|| EXAMPLE APPLICATIONS

|| apply_interpreter ext "(((1 + 2) * 3) + 22)."   => VAL  31
|| apply_interpreter ext "5 + 2 * 4 + -2 - 3 * 4." => VAL  -1
|| apply_interpreter ext "1 - 2 - 3."              => VAL   2
```

Figure 10. An example of a complete program written in W/AGE

the ex part of ext. The semantic rule *a_rule 3 (VAL $of lhs) EQ add[VAL $of s1, VAL $of s3]* states that the VAL attribute of an ex with structure (s1 ex ++ s2 plus ++ S3 ex) is equal to the result obtained when the attribute function add is applied to the list of attributes containing the VAL attributes of the substructures s1 and s3. The equations that follow the definitions of the attribute grammars, define the attribute functions using predefined functions in the host language.

4. NON LEFT-RECURSIVE RECOGNIZERS. The fourth component of the W/AGE program in Figure 10 contains definitions of the non-left-recursive recognizers that are used to guard the left-recursive executable attribute grammars. It is always possible, although sometimes difficult, to transform a left-recursive grammar to an equivalent non-left-recursive form. Algorithms to do this are available in many of the standard textbooks on programming language processing. Ideally the W/AGE environment would perform this transformation automatically.
5. EXAMPLE APPLICATIONS. The final component of the program contains example applications of the interpreter ext. These examples illustrate a feature of the new technique that we have not yet mentioned: operator precedence is enforced through the ordering of the alternatives in a production. For example, in Figure 10, the alternatives for the attribute grammar ex are listed in the following order: bracketed ex, summation, subtraction, product and negation. The guarding technique results in an operator binding precedence which is the reverse of this ordering, as illustrated in the two example applications at the bottom of the Figure.

VIABILITY OF THE APPROACH

Functional programming languages that use normal-order evaluation are frequently criticized as being too inefficient to be of value in practical applications. In addition, executable specifications are generally less efficient than are implementations that meet specifications. Consequently, it is natural to question the viability of building language processors as executable specifications written in such languages. It may appear even less plausible that the approach which we have proposed for accommodating left-recursive productions could have any value whatsoever, owing to its exponential behaviour as discussed below. We address these concerns in this section.

Advantages of using a normal-order evaluation

Three advantages derive from the normal-order evaluation strategy used by functional languages such as Miranda: (1) it enables the construction of guarded attribute grammars, (2) it enables the construction of executable attribute grammars which include inheritance from the right, (3) it prevents unnecessary computation of attributes that would otherwise occur when backtracking parsers follow unproductive derivation paths. Not only is normal-order evaluation necessary for the approach described in this paper, it contributes significantly to its viability.

Applications of W/AGE

In addition to use as a teaching aid for a second year university 'Grammars and Translators' course, W/AGE has been used in three research projects involving the construction of non-trivial language processors: (1) a natural-language front-end to a database system, (2) an SQL query processor covering all of the standard features of the language including aggregates, and (3) a VLSI design transformer which transforms mathematical specifications of finite impulse response (FIR) filters into executable specifications of systolic circuits based upon a standard VLSI cell, and subsequently transforms these executable specifications into net-list layouts in an industry-standard format. In all three applications the response time was sufficient to support the investigation being carried out.

The most complex language processor that has been constructed using W/AGE is the natural-language front-end. This processor is capable of interpreting questions expressed in a first-order subset of English. The interpreter can accommodate conjunctions and disjunctions, quantifier scoping, pronoun resolution, and various other language features. The grammar of the executable specification contains 132 terminals, 38 production rules and 53 attribute equations. Table I gives response times for four arbitrary queries evaluated on a Sun SPARC 2 with the current implementation of W/AGE, which uses Miranda as host language. The fourth column is the time spent on parsing the query. Owing to the fact that Miranda uses normal-order evaluation, the data for the fourth column was obtained by simply monitoring the time taken when the interpreter were asked whether the query was successful, rather than the time taken to return the result. Note that two answers are returned for the last query owing to its ambiguity.

Recent advances in implementation techniques have resulted in functional languages which are ten times faster than Miranda. In addition, advances in computer hardware have resulted in workstations with processors that are many times faster than a Sun SPARC station 2. In combination these developments have a significant effect on the applicability of the approach discussed in this paper. For example, a thirtyfold increase in speed of the natural language interpreter described above would result in response times that would be acceptable to most database users. However, owing to the fact that the processing time depends on the length of the input, several more orders of magnitude increase in processing speed would be required before the

Table I.

Query	Result	Response time (s)	Parsing time (s)
Which planet is orbited by Phobos?	Mars	1·00	0·55
Io was discovered by Galileo and it is a moon.	true	2·05	1·07
Every moon that orbits Mars was discovered by Hall.	true	4·67	3·82
Which planets are orbited by a moon that was discovered by Hall or Kuiper?	Mars or Mars, Uranus and Neptune	8·92	7·72

approach could be of value in applications where the input to be parsed is long, such as the construction of executable specifications of compilers. Even then, this presupposes that the processing time is linearly dependent on the length of the input. We discuss this further in the next subsection.

Use of left-recursive productions

In the current implementation, the time complexity of the parsing process used by guarded attribute grammars is exponential in the depth of left-recursive calls. It is not yet clear whether this efficiency problem is inherent in our technique, or is a property of the implementation. If it is inherent in the technique, it will severely limit application to prototyping language processors or to cases where the potential depth of left recursion is known to be very small. It is possible that there are many applications where the potential depth of left recursion is very small. For example, consider the example language processor ext from Figure 10. Only three left-recursive calls are made in the first example application given at the bottom of the Figure. More left-recursive calls are required as the number of brackets is reduced and more reliance is put on operator-binding precedence. In practice, users do not rely on operator precedence to the extent assumed in the second example application in Figure 10.

Where the depth of left recursion is not small, the use of guarded attribute grammars will be restricted to language prototyping. After the prototype has been developed, the language designer will have a number of choices. Should he or she wish to continue to implement the final version as a modular executable specification, it will be necessary to convert the attribute grammar to an equivalent attribute grammar that does not contain left-recursive productions. Transformation of the syntactic part of the grammar will always be possible using standard textbook techniques, but in many cases it will be a non-trivial process to transform the attribute equations.

The construction of efficient language processors for LL(1) grammars

In some applications it may be desirable to eliminate all backtracking in the parsing process and still implement the language processor as a modular executable specification. In many cases this can be achieved by transforming the attribute grammar to LL(1) form and subsequently implementing it using an additional extension to Burge's method that we shall discuss in this subsection.

Once again, for simplicity, we describe the extension to Burge's method with respect to recognizers rather than executable attribute grammars. The extension involves the use of two new higher-order functions: directs and excel_orelse defined in Figure 11.

An example use of these functions is given in Figure 12, which contains an executable specification of a non-backtracking recognizer for a language defined by the following LL(1) grammar:

```
t   ::=  s  '.'
s   ::=  a  b  'v'
a   ::=  'w'  b
```

```
directs :: [char] -> interpreter -> interpreter

(terms $directs i) inps   = i ((concat . map test) inps)
                            where
                            test (done,[]     )  = []
                            test (done,(u:us))  = [(done,(u:us))],member  terms  u
                                                = [], otherwise


excl_orelse :: interpreter -> interpreter -> interpreter

(p1 $excl_orelse p2) input  = result, result ~= []
                            = p2 input, otherwise
                              where
                              result = p1 input
```

Figure 11. Definitions of the functions directs *and* excl_orelse

```
period   = terminal '.'
v        = terminal 'v'
w        = terminal 'w'
x        = terminal 'x'
y        = terminal 'y'
z        = terminal 'z'

t =  s $then  period

s =  a $then b $then v

a = (['w'] $directs (w $then b))
    $excl_orelse
    (['x'] $directs (x $then s))
    $excl_orelse
    y

b = (['w','x','y'] $directs (a $then s))
    $excl_orelse
    z
```

Figure 12. An efficient executable specification

```
     |   'x'  s
     |   'y'
b   ::=  a   s
     |   'z'
```

The function directs causes the parsing process to examine the next character of the input. If the character is in the list of characters which constitutes the first argument to directs, then the recognizer given as second argument to directs is applied, otherwise the next alternative in the production is tried. The function excel_orelse is similar to orelse defined earlier but differs in that when an alternative in a production is successful, no other alternatives are tried.

Executable specifications that are implemented in this way have complexity that is linear in the length of the input to be processed. Table II gives times for the recognizer in Figure 12 to recognize five inputs successfully on a Sun SPARC station 2.

Table II.

Length of input in characters	341	715	1089	1446	1837
Seconds cpu	0·57	1·30	1·78	2·57	2·95

RELATIONSHIP TO OTHER WORK

Use of higher-order functions to create executable specifications of language processors was first described by Burge[2] in 1975. The technique was made significantly more elegant in 1985 through Wadler's notion of 'replacing failure by a list of successes'[3]. Subsequent to Wadler's work, Johnsson[10] suggested that a new case-like structure be added to lazy functional languages in order to express attribute dependencies over data structures. The following year, 1988, Uddeborg[11] built an LR(1) functional parser generator FPG which could accept a very general class of attribute grammars.

There would appear to be no work, other than that reported in this paper, that attempts to tackle the problem of constructing executable specifications of language processors that use a top-down parsing strategy and which have structures that directly reflect the structure of grammars containing left-recursive productions. The simple extension of Burge's approach to accommodate the construction of efficient executable specifications of LL(1) grammars would also appear to be novel.

CONCLUSION

We have described an approach by which it is possible to construct executable specifications of language processors that use a top-down parsing strategy and which have structures that directly reflect the structure of grammars containing left-recursive productions. The specifications are in a form that is similar to a commonly used notation for specifying attribute grammars. The specifications are declarative, with the exception that operator precedence is determined by the order of alternatives within a production. The use of a top-down parsing strategy, together with the declarative nature of the host language, results in modular specifications in the sense that individual productions can be designed, compiled and tested independently of all other productions apart from those that are referred to explicitly in the defining equation. The extension of a specification to cover a larger language is straightforward; new productions and/or alternatives are added as required. For example, suppose that the function ex in Figure 10 is to be extended to include 'powers'. We would simply add the following production together with an appropriately-defined attribute function raise and a non-left-recursive recognizer rec_power:

```
$orelse(rec_power
        $enables structure(s1 ex++s2(uninterpreted(SPECIAL_SYMBOL "^")++s3 ex)
                          [a_rule 7(VAL $of lhs)EQ raise[VAL $of s1, VAL $of s3]])
```

A number of advantages derive from the declarative, modular and expressively clear nature of the executable programs written in this way. One of the most important advantages is that these features facilitate proof of properties. Because the programs are executable specifications, the question of the correctness of the implementation with respect to the language specification does not arise. The language specification is the language processor. The declarative recursive nature of the specification facilitates proof by induction. The referential transparency of the host language Miranda, together with the limited use of variables in the specifications, facilitates use of equational proof techniques. Assuming that our current work establishes a strong termination property for the parsing algorithm that we have proposed, proof of termination of W/AGE programs will reduce to proof of termination of attribute computations.

The technique proposed in this paper is such that only the leftmost derivation is returned by a guarded attribute grammar for each segment of terminals recognized. There are two consequences of this. First, the technique imposes a right-associativity on operators (and on juxtapositions). For example, consider the last example application of the attribute grammar ext given at the bottom of Figure 10. Secondly, in some cases, only one result is returned for ambiguous grammars. In applications such as natural-language front-ends to database systems, such 'disambiguation' is inappropriate. Left-associativity of operators can be obtained by restructuring a grammar, but this may defeat the purpose of using guarded attribute grammars if the resulting grammar is descriptively unnatural.

Current work is directed at a number of issues that have arisen directly from the work reported in this paper. In particular, we are (i) developing a formal proof of termination properties, (ii) determining whether or not the complexity of the approach can be improved without affecting modularity, and (iii) looking at ways in which we might allow left-recursive grammars to return a result for each way in which a segment of terminals can be parsed. It would then be possible to choose the left- or right-associative result as appropriate. However, this problem is closely related to termination properties, and it is not clear if a solution exists.

We are also working on improving the speed, availability and user-friendliness of the W/AGE environment. We already have a prototype structure editor, and hope to have various implementations of W/AGE in other widely-available functional programming languages within a year.

ACKNOWLEDGEMENT

The author acknowledges the support of the Natural Science and Engineering Research Council of Canada and of Wadia Farook, School of Computer Science, University of Windsor.

REFERENCES

1. K. Koskimies, 'Lazy recursive descent parsing for modular language implementation', *Software—Practice and Experience*, **20**, 749–772 (1990).
2. W. H. Burge, *Recursive Programming Techniques*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1975.
3. P. Wadler, 'How to replace failure by a list of successes', in J. P. Jouannaud (ed.) *Functional Programming Languages and Computer Architectures, Lecture Notes in Computer Science 201*, Springer-Verlag, Heidelberg, 1985 p. 113.
4. J. E. Stoy, *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*, MIT Press, 1977.
5. A. J. Field and P. G. Harrison, *Functional Programming*, Addison-Wesley Publishing Company, 1988.
6. D. Turner, 'Functional programs as executable specifications', in C. A. Hoare and J. C. Shepardson

(eds), *Mathematical Logic and Programming Languages*, Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

7. D. Turner, ‘A non-strict functional programming language with polymorphic types’, *Proc. IFIP Int. Conf. on Functional Programming Language and Computer Architecture*, Nancy, France, *Springer Lecture Notes 201*, 1985.
8. R. A. Frost, ‘Constructing programs as executable attribute grammars’, *The Computer Journal,* **35** (4), 376–389 (1992).
9. P. Deranshart, M. Jourdan and B. Lorho, ‘A survey of attribute grammars. Part I—main results on attribute grammars’, *Report 485*, INRIA, 1986.
10. T. Johnsson, *Attribute Grammars as a Functional Programming Paradigm, Springer Lecture Notes 274*, 155–173 (1987).
11. G. Uddeborg, ‘A functional parser generator’, *Licentiate Thesis*, Chalmers University of Technology, Goteborg, 1988.