Supporting the Attribute Grammar Programming Paradigm in a Lazy Functional Programming Language

R. A. Frost and S. Karamatos

School of Computer Science, University of Windsor, Ontario, Canada N9B 3P4

Abstract. Attribute grammars were introduced in the late 60's. In the 70's they found use in compiler work, a use that is continuing to grow. A more recent development is that of the 'attribute grammar programming paradigm'. A number of environments have been built to support this paradigm. W/AGE is one such environment. It consists of several functions that extend the standard environment of the pure lazy functional programming language Miranda. W/AGE has been used in the construction of various types of program including natural language interpreters, database front-ends, file-processors, theorem provers, and VLSI specification transformers.

1 Introduction

Over the last several years researchers at the University of Windsor have been involved in various projects involving the investigation of new theories and techniques in the areas of database management, VLSI design and natural language processing. These investigations have all required the construction of special purpose language processers. The design and construction of these processors required a good deal of effort. It became evident that substantial resources were being used for this purpose and that this was having a deleterious effect on our research. We decided, therefore, to construct a programming environment that would enable researchers to produce language processors with minimum effort. The environment that we have built is called the Windsor Attribute Grammar Programming Environment W/AGE. This environment allows language processors to be constructed as executable specifications of the syntax and semantics of the languages required.

W/AGE was initially used in the construction of natural language interpreters, database front-ends, and specification transformers. Subsequently, it was recognised that other types of program could be profitably constructed as language processors and W/AGE has since been used to build theorem provers, tree procesors and even file processors.

The purpose of this paper is twofold: firstly to introduce readers to W/AGE and secondly to illustrate the wide applicability of the technique of constructing programs as executable attribute grammars.

2 Attribute Grammars

Attribute grammars were introduced by Knuth in 1968[10] as a means for specifying the semantics of context free languages. Since then, attribute grammars have been used extensively in compiler work.

3.1 Notation Used in this Section

The following Miranda notation is used in this section:

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

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

Where the set **type** is defined inductively as follows:

`num`, `char`, `bool` ∈ **type**.

If `t` ∈ **type** then so is `[t]`,
ie. the type of lists whose elements are of type `t`.

If `t1..tn` ∈ **type** then so is `(t1,..,tn)`,
ie. the type of tuples with elements of type `t1` to `tn`.

If `t1, t2` ∈ **type** then so is `t1 -> t2`,
ie. the type of functions with arguments in `t1` and results in `t2`.

If `y` and `z` ∈ **type** then so is `x`
where `x ::= C1 y |..| Cn z` and `C1` to `Cn` are
user defined *constructors*.

3.2 The Type of Terminals

The type `terminal` is predefined in W/AGE as follows:

```
terminal ::= INT_TERM              [char]   | REAL_TERM            [char]
           | IDENTIFIER_TERM       [char]   | SPECIAL_SYMBOL_TERM [char]
           | RESERVED_WORD_TERM [char]   | UNCATEGORISED_TERM  [char]
           | ANY_TERM              [char]
```

In addition to defining the type `terminal` this introduces seven new identifiers: `INT_TERM`, `REAL_TERM`, etc. as constructors for terminals. Note that each of these constructors is of type `[char] -> terminal`. We introduce an acronym for this type:

```
terminal_constructor == [char] -> terminal
```

3.3 The Type of the Lexical Scanning Function `tokenise`

```
tokenise :: string_to_be_processed -> [terminal]
            where
            string_to_be_processed == [char]
```

3.4 The Type of Attributes

The type `attribute` is defined by W/AGE users according to the application. For example:

```
attribute ::= LITERAL_VAL terminal | VAL num
            | OP num -> num -> num | PAIR num [char]
```

Note that constructors can have any number of fields. For example, the constructor `PAIR` has two fields. A field may be any Miranda type as defined in 3.1. To analyse an attribute, we use Miranda pattern matching as illustrated later.

With a little change in perspective, many other types of program can be constructed as executable attribute grammars. We refer to such programs as *passages*. This style of programming was first suggested by Knuth in 1971[12], and subsequently developed by Katayama[9], Hehner and Silverberg[7], Simon[16], Johnsson[8], Panayiotopoulos, Papakonstantinou, and Stamatopoulos[14], Forbig and Lammel[3], Frost[4][5] and others. Several environments have been built to support the attribute grammar programming paradigm, *eg.* PLASTIC [16], SAGE[15], AGILP[14], FLR[3], and W/AGE.

W/AGE consists of a several functions that extend the standard environment of the pure lazy programming language Miranda[17]. The resulting combination of programming paradigms facilitates software development in several ways:

1. Programs are completely declarative, extremely modular, and are largely variable free. This simplifies reasoning about them for the purpose of verification, complexity analysis, transformation, etc.
2. The inductive program structure that results from the combined paradigm lends itself well to the technique of deriving 'programs from proofs'.
3. The structure of a program that is built in this way is closely related to the structure of the data that it is to process. This results in code that is easier to maintain and easier to modify.

This paper will introduce readers to the attribute grammar programming paradigm, show how this paradigm can be readily supported in a pure functional programming language, and briefly discuss some of the advantages that derive from this approach.

3 An Overview of W/AGE

We use the notation of the Miranda[1] functional programming language throughout the paper. We give brief explanation of this notation where appropriate. Readers who are unfamiliar with functional notation are referred to Turner[17].

W/AGE currently consists of five components:

A lexical scanning function:	`tokenize`
A set of functions for applying interpreters:	`{apply_recogniser, apply_interpreter}`
A set of functions for building basic interpreters:	`{literal, interpreted, uninterpreted}`
A set of interpreter combinators:	`{$orelse, $excl_orelse, structure}`
A function for creating attribute lists:	`meaning_of`

[1] Miranda is a trademark of Research Software Ltd.

3.5 The Type of Interpreters

We have chosen to define the type **interpreter** as follows:

```
interpreter == [([attribute], [terminal])]  ->  [([attribute], [terminal])]
```

That is, an interpreter is a function that maps a list of pairs of type **([attribute], [terminal])** to a list of pairs of the same type, such that:

1. Each pair **(as,ts)** that is in the list that is input to an interpreter is such that the list of attributes **as** may be regarded as a context in which the list of terminals **ts** is to be interpreted.
2. Each pair **(as',ts')** in the list that is output by an interpreter is related to exactly one pair **(as,ts)** in the input list such that: (i) **as'** is a subset of the union of **as** and the interpretation of some initial segment of **ts**, and (ii) **ts'** is the list of remaining uninterpreted terminals in **ts**.
3. Interpreters return lists of pairs because each pair in the input may have more than one interpretation.
4. Interpreters are regarded as accepting lists of pairs for a number of reasons, such as it simplifies composition.

3.6 The Type of Functions for Top-level Application of Interpreters

The types of the functions **apply_recognizer** and **apply_interpreter** are as follows:

```
apply_recognizer :: interpreter -> string_to_be_recognized -> message
                    where   string_to_be_recognized  == [char]
                            message                  == [char]

apply_interpreter :: interpreter -> string_to_be_interpreted -> [attribute]
                     where  string_to_be_interpreted  == [char]
```

Throughout the paper, we use the notation **x => y** to indicate that output **y** is returned by the Miranda interpreter when **x** is evaluated. For example:

```
apply_interpreter number "12"  =>  [VAL 12]
```

3.7 The Type of Functions for Building Basic Interpreters

There are three functions in W/AGE that may be used to build interpreters for single terminals:

```
literal       :: terminal_constructor    -> interpreter
uninterpreted :: terminal                -> interpreter
interpreted   :: (terminal, [attribute]) -> interpreter
```

3.8 The Type of Interpreter Combinators

There are three functions in W/AGE that may be used to define new interpreters in terms of other interpreters:

```
$orelse      :: interpreter  -> interpreter  -> interpreter
$excl_orelse :: interpreter  -> interpreter  -> interpreter
structure    :: list_of_tagged_interpreters -> list_of_attribute_rules
                -> interpreter
  where
  list_of_tagged_interpreters == [(tag, interpreter)]
  list_of_attribute_rules     == [(rule_num, att_id, att_function, [att_id])]
                                 where
                                 att_id == ((tag, att_direction), att_type)
                                 att_function  == [attribute] -> attribute
                                 att_type      == [char]
                                 att_direction ::= UP | DOWN
```

3.9 The Type of the meaning_of Function

The type of the function **meaning_of** is as follows:

```
meaning_of :: interpreter -> string_to_be_interpreted -> [attribute]
              where string_to_be_interpreted  == [char]
```

In the next few sections, we give examples of various passages that have been built using W/AGE. All of the passages are complete and can be executed as Miranda programs just as they appear provided that the W/AGE script is available in the Miranda 'local' directory.

4 Lexical Analysis

Passage #1 illustrates how the function **tokenise** can be tailored for particular applications through definition of the reserved words and special symbols.

```
|| Passage #1:
%insert <local/header_for_WAGE_VERSION_1_RELEASE_0.m>
||-----------------------------------------------------------------------
attribute     :: type
||-----------------------------------------------------------------------
reserved_words  = ["begin", "end", "one"]
special_symbols = ['(', ')', '-', '/', '+', '-']
||-----------------------------------------------------------------------
|| EXAMPLE APPLICATION
|| tokenise "123 begin sas ddd a234 b3.2 3.4 (ff%"  =>
|| [INT_TERM "123", RESERVED_WORD_TERM "begin",IDENTIFIER_TERM "sas",
|| IDENTIFIER_TERM "ddd",IDENTIFIER_TERM "a234", UNCATEGORISED_TERM "b3.2",
|| REAL_TERM  "3.4", SPECIAL_SYMBOL_TERM "(", UNCATEGORISED_TERM "ff%"]
```

5 Constructing Basic Recognisers and Interpreters

Passage #2 illustrates how the W/AGE functions **literal**, **interpreted**, and **uninterpreted** can be used to build basic recognisers and interpreters, *ie.* recognisers and interpreters for single terminals.

```
|| Passage #2:
%insert <local/header_for_WAGE_VERSION_1_RELEASE_0.m>

||-----------------------------------------------------------------------

attribute     ::= LITERAL_VAL terminal     | VAL    num
                | ENGLISH [char]           | GENDER char

||-----------------------------------------------------------------------

reserved_words  = ["begin", "one"]

special_symbols = []

||-----------------------------------------------------------------------

int         = literal       INT_TERM
anything    = literal       ANY_TERM
key         = uninterpreted (IDENTIFIER_TERM any)
begin       = uninterpreted (RESERVED_WORD_TERM "begin")
one         = interpreted   (RESERVED_WORD_TERM "one", [VAL 1])
salaire     = interpreted   (IDENTIFIER_TERM "salaire",[ENGLISH "wage",
                             GENDER 'm'])
||-----------------------------------------------------------------------
|| EXAMPLE APPLICATIONS
||
|| apply_recognizer  int        "64"     => input is recognized successfully
|| apply_interpreter int        "106"    => [LITERAL_VAL (INT_TERM "106")]
|| apply_interpreter anything   "3.21"   => [LITERAL_VAL (REAL_TERM "3.21")]
|| apply_interpreter key        "sas"    => []
|| apply_interpreter begin      "begin"  => []
|| apply_interpreter begin      "sas"    => input not recognized
|| apply_interpreter one        "one"    => [VAL 1]
|| apply_interpreter salaire "salaire"   => [ENGLISH "wage", GENDER 'm']
```

6 Constructing Non-Basic Recognizers

Passage #3 illustrates how non-basic recognizers can be built by 'gluing' other recognisers together using the combinators **$orelse** and **$structure**.

```
|| Passage #3: A recogniser of arithmetic expressions
%insert <local/header_for_WAGE_VERSION_1_RELEASE_0.m>
||-----------------------------------------------------------------------
attribute     ::= LITERAL_VAL terminal
||-----------------------------------------------------------------------
reserved_words  = []
special_symbols = ['(', ')', '*', '/', '+', '-']
||-----------------------------------------------------------------------
op                    = uninterpreted   (SPECIAL_SYMBOL_TERM "+")
                        $orelse
                        uninterpreted   (SPECIAL_SYMBOL_TERM "*")
                        $orelse
                        uninterpreted   (SPECIAL_SYMBOL_TERM "/")

negate                = uninterpreted   (SPECIAL_SYMBOL_TERM "-")
opbr                  = uninterpreted   (SPECIAL_SYMBOL_TERM "(")
clbr                  = uninterpreted   (SPECIAL_SYMBOL_TERM ")")
uninterpreted_number  = uninterpreted   (INT_TERM any)
                        $orelse
                        uninterpreted   (REAL_TERM any)
||-----------------------------------------------------------------------
rec_expr = structure (s1 uninterpreted_number)
           []
           $orelse
           structure (s1 opbr ++ s2 rec_expr ++ s3 op ++ s4 rec_expr ++ s5 clbr
           []
           $orelse
           structure (s1 negate ++ s2 rec_expr)
           []
||-----------------------------------------------------------------------
|| EXAMPLE APPLICATIONS
|| apply_recognizer op   "+"             => input is recognized successfully
|| apply_recognizer rec_expr "(12 + 45)" => input is recognized successfully
|| apply_recognizer rec_expr "12 + 5"    => end of input not recognized
||                              ie [SPECIAL_SYMBOL_TERM "+",INT_TERM "5"]
|| apply_recognizer rec_expr "(12 + 45) + 3"
||                             => end of input not recognized
||                              ie [SPECIAL_SYMBOL_TERM "+",INT_TERM "3"]
|| apply_recognizer rec_expr "((one + 45) + 3)"=> input not recognized
```

Notice that the lists of attribute rules in the definition of **rec_expr** are empty. This is because **rec_expr** is a recognizer and not an interpreter, therefore no attributes are to be computed.

7 Constructing Non-Basic Interpreters

Passage #4 illustrates how non-basic interpreters can be built by 'gluing' other interpreters together using the combinators **orelse** and **structure**.

```
|| Passage #4: An arithmetic evaluator
%insert <local/header_for_WAGE_VERSION_1_RELEASE_0.m>
%insert <local/number_interpeter_for_WAGE_VERSION_1_RELEASE_0.m>
||-----------------------------------------------------------------------
attribute ::= LITERAL_VAL terminal | VAL num  | OP num -> num -> num
||-----------------------------------------------------------------------
reserved_words  = []
special_symbols = ['(', ')', '*', '/', '+', '-']
||-----------------------------------------------------------------------

op    = interpreted   (SPECIAL_SYMBOL_TERM "+", [OP (+)])
        $orelse
        interpreted   (SPECIAL_SYMBOL_TERM "*", [OP (*)])
        $orelse
        interpreted   (SPECIAL_SYMBOL_TERM "/", [OP (/)])
negop = uninterpreted (SPECIAL_SYMBOL_TERM "-")
opbr  = uninterpreted (SPECIAL_SYMBOL_TERM "(")
clbr  = uninterpreted (SPECIAL_SYMBOL_TERM ")")

||-----------------------------------------------------------------------

expr = structure (s1 number)
       [c_rule 1 (VAL $u lhs) EQ           (VAL $u s1)]
       $orelse
       structure (s1 opbr ++ s2 expr ++ s3 op ++ s4 expr ++ s5 clbr)
       [a_rule 2 (VAL $u lhs) EQ  apply_op [VAL $u s2,OP $u s3,VAL $u s4]]
       $orelse
       structure (s1 negop ++ s2 expr)
       [a_rule 3 (VAL $u lhs) EQ    negate [VAL $u s2]]

apply_op        [VAL x, OP y, VAL z]       = VAL (y x z)
negate          [VAL x]                    = VAL (-x)

||-----------------------------------------------------------------------
|| EXAMPLE APPLICATIONS
||
|| apply_interpreter expr "(12 + 4.5)"                  => [VAL 16.5]
|| apply_interpreter expr "((4 + (4 * 3))/-2)"         => [VAL (-8.0)]
```

The syntax used for the attribute rules is a variant of standard BNF notation. The following provides an informal semantics for our notation:

`v $u s`	stands for "the synthesized `v` attribute passed *up* by the structure `s`"
`v $d s`	stands for "the inherited `v` attribute passed *down* to the structure `s`"
`c_rule n x EQ y`	indicates that the attribute `x` is to be *copied* from the attribute `y`
`a_rule n x EQ f l`	indicates that the attribute `x` is obtained by *applying* the attribute function `f` to the list of attributes `l`
`i_rule n x EQ y`	indicates that the attribute `x` is to be *initialised* to the value `y`

8 Examples of Passages

8.1 A Simple Data Processing Example

Passage #5 calculates the average number of entries per record in a file in which each record consists of an integer key followed by one or more alphanumeric string entries. Records are separated by semicolons, fields by commas, and end-of-file is signified by a period.

```
|| Passage #5: Calculating average number of entries of records in a file
%insert <local/header_for_WAGE_VERSION_1_RELEASE_0.m>
||-----------------------------------------------------------------------
attribute ::= LITERAL_VAL terminal | NUM_RECS num | NUM_ENTS num
              | AV_ENTS num
||-----------------------------------------------------------------------
reserved_words = []
special_symbols = ['.', ';', ',']
||-----------------------------------------------------------------------
key        = uninterpreted (INT_TERM any)
entry      = uninterpreted (IDENTIFIER_TERM any)
period     = uninterpreted (SPECIAL_SYMBOL_TERM ".")
semicolon  = uninterpreted (SPECIAL_SYMBOL_TERM ";")
comma      = uninterpreted (SPECIAL_SYMBOL_TERM ",")
||-----------------------------------------------------------------------
file    = structure (s1 records ++ s2 period)
          [c_rule 1 (NUM_ENTS $u lhs) EQ (NUM_ENTS $u s1),
           c_rule 2 (NUM_RECS $u lhs) EQ (NUM_RECS $u s1),
           a_rule 3 (AV_ENTS  $u lhs) EQ    calc_average[NUM_ENTS $u lhs,
                                                         NUM_RECS $u lhs]]

records = structure (s1 record ++ s2 semicolon ++ s3 records)
          [a_rule 4 (NUM_RECS $u lhs) EQ add_one_to_num_recs[NUM_RECS $u s3],
           a_rule 5 (NUM_ENTS $u lhs) EQ       add_num_ents [NUM_ENTS $u s1,
                                                             NUM_ENTS $u s3]]
          $excl_orelse
          structure (s1 record)
          [i_rule 6 (NUM_RECS $u lhs) EQ (NUM_RECS 1),
           c_rule 7 (NUM_ENTS $u lhs) EQ (NUM_ENTS $u s1)]
```

```
record  = structure (s1 key ++ s2 comma ++ s3 entries)
          [c_rule 8 (NUM_ENTS $u lhs) EQ  (NUM_ENTS $u s3)]

entries = structure (s1 entry ++ s2 comma ++ s3 entries)
          [a_rule 9 (NUM_ENTS $u lhs) EQ add_one_to_num_ents[NUM_ENTS $u s3]]
          $excl_orelse
          structure (s1 entry)
          [i_rule 10 (NUM_ENTS $u lhs)  TO (NUM_ENTS 1)]

calc_average         [NUM_ENTS x, NUM_RECS y] = AV_ENTS  (x/y)
add_one_to_num_recs  [NUM_RECS x]             = NUM_RECS (1 + x)
add_num_ents         [NUM_ENTS x, NUM_ENTS y] = NUM_ENTS (x + y)
add_one_to_num_ents  [NUM_ENTS x]             = NUM_ENTS (1 + x)
||-------------------------------------------------------------------------
|| EXAMPLE APPLICATION
|| apply_interpreter  file "1234,hesselink,hensen,jones;2345,
||                           bauer,partsch,sharir,morgan;5678,heath,lin."
||              => [NUM_ENTS 9,NUM_RECS 3,AV_ENTS 3.0]
```

8.2 A Passage to Reverse a List

Passage #6 reverses its input. The combinator **$excl_orelse** avoids unnecessary backtracking and ensures that the list is parsed in only one way (*ie.* the parse should include all elements). Reverse here has $O(n^2)$ complexity, owing to the fact that the append operator ++ is O(m) where m is the length of its left operand. It is relatively straightforward to transform this passage to one with O(n) complexity.

```
|| Passage #6: Reversing a list
%insert <local/header_for_WAGE_VERSION_1_RELEASE_0.m>
||-------------------------------------------------------------------------
attribute   ::= LITERAL_VAL terminal | RES list_of_terminals
||-------------------------------------------------------------------------
reserved_words = []
special_symbols = []
||-------------------------------------------------------------------------
elem  = literal ANY_TERM

list = structure (s1 elem ++ s2 list)
       [a_rule 1 (RES $u lhs) EQ stick_on_end [LITERAL_VAL $u s1,
                                               RES $u s2]]
       $excl_orelse
       structure (s1 elem)
       [a_rule 2 (RES $u lhs) EQ make_list    [LITERAL_VAL $u s1]]

make_list    [LITERAL_VAL x]        = RES [x]
stick_on_end [LITERAL_VAL x, RES y] = RES (y ++ [x])
||-------------------------------------------------------------------------
|| EXAMPLE APPLICATION
|| apply_interpreter list "5 six 7 8"
||    => [RES [INT_TERM "8",INT_TERM "7",IDENTIFIER_TERM "six",
||             INT_TERM "5"]]
```

8.3 A Passage to Calculate Fibonacci Numbers

Passage #7 recognizes numbers and returns their Fibonacci values. It would appear that the attribute grammar paradigm is not appropriate for this problem. However, an alternative passage for calculating fibonacci numbers in linear time can be obtained by transforming this passage as discussed in Frost[5].

```
|| Passage #7: Calculating Fibonacci numbers
%insert <local/header_for_WAGE_VERSION_1_RELEASE_0.m>
%insert <local/number_interpreter_for_WAGE_VERSION_1_RELEASE_0.m>
||-------------------------------------------------------------------------
attribute   ::= LITERAL_VAL terminal | FIB num | VAL num
||-------------------------------------------------------------------------
reserved_words  = []
special_symbols = []
||-------------------------------------------------------------------------
fibnumb = structure (s1 number)
          [a_rule 1 (FIB $u lhs) EQ calc_fib [VAL $u s1]]

calc_fib [VAL  x] = FIB (fib  x)
                    where   fib n = 1                          , n <= 2
                            fib n = fib (n - 1) + fib (n - 2), otherwise
||-------------------------------------------------------------------------
|| EXAMPLE APPLICATION
||                  apply_interpreter fibnumb "5" => [FIB 5]
```

9 Use of the Function meaning_of

The function **meaning_of** allows lists of attributes to be defined in terms of the application of an interpreter to an expression. The following passage illustrates such use.

Notice that in this passage, the proper noun "john" does not denote an entity. Rather, it denotes a function defined in terms of the entity denoted by the number 1. This idea is loosely based on a notion from Richard Montague's approach to the interpretation of natural language.

```
|| Passage #8:
%insert <local/header_for_WAGE_VERSION_1_RELEASE_0.m>
||-------------------------------------------------------------------------
attribute ::= ANS bool              | DETVAL   ([entity] -> [entity] -> bool)
            | CNOUNVAL  [entity] | T_PHRASEVAL ([entity] -> bool)
            | INTRVBVAL [entity] | LITERAL_VAL terminal
entity == num
||-------------------------------------------------------------------------
reserved_words    = []
special_symbols   = []
||-------------------------------------------------------------------------
cnoun  = interpreted          (IDENTIFIER_TERM "person",[CNOUNVAL  [1..10]])
       $orelse interpreted (IDENTIFIER_TERM "woman", [CNOUNVAL  [6..10]])
intrvb = interpreted          (IDENTIFIER_TERM "runs",  [INTRVBVAL [2..7]])
```

```
det    = interpreted         (IDENTIFIER_TERM "every", [DETVAL    f_every])
         $orelse interpreted (IDENTIFIER_TERM "a",     [DETVAL    f_a])
pnoun  = interpreted         (IDENTIFIER_TERM "john",  [T_PHRASEVAL f_john])
         $orelse interpreted (IDENTIFIER_TERM "someone",
                                          meaning_of detphrase "a person")
||-----------------------------------------------------------------------

sent       = structure (s1 termphrase ++ s2 intrvb)
             [a_rule 1 (ANS $u lhs) EQ   apply1[T_PHRASEVAL $u s1,
                                                INTRVBVAL $u s2]]
termphrase = structure (s1  pnoun)
             [c_rule 2 (T_PHRASEVAL $u lhs) EQ (T_PHRASEVAL $u s1)]
             $orelse
             structure (s1 detphrase)
             [c_rule 3 (T_PHRASEVAL $u lhs) EQ (T_PHRASEVAL $u s1)]

detphrase = structure (s1 det ++ s2 cnoun)
            [a_rule 4 (T_PHRASEVAL $u lhs)  EQ   apply2[  DETVAL $u s1,
                                                    CNOUNVAL $u s2]]
apply1   [T_PHRASEVAL f, INTRVBVAL s] = ANS (f s)
apply2   [DETVAL f, CNOUNVAL s]       = T_PHRASEVAL (f s)

f_every x  y   = (x --- y) = []
f_a     x  y   = (intersect  x  y) ~= []
                   where intersect  x  y = (x --- (x --- y))

f_john   x     = member x  1
||-----------------------------------------------------------------------
|| EXAMPLE APPLICATIONS
||
|| apply_interpreter sent "every woman runs" => [ANS False]
|| apply_interpreter sent "someone runs"    => [ANS True]
|| apply_interpreter sent "john runs"       => [ANS False]
```

10 Left Recursion

W/AGE has recently been extended to accommodate attribute grammars with left recursive productions. The technique that allows left recursive productions to co-exist with top down parsing is described in Frost [6].

11 Interpreting Ambiguous Input

In no example given so far, have we applied an interpreter directly to an input. We have always used the higher order function **apply_interpreter**. This approach is only appropriate when at most one parse of the input is anticipated. However, there are applications in which it is necessary to return multiple interpretations, one for each way in which the input can be parsed. Recall, from section 3.5, that the type **interpreter** is defined as follows:

```
interpreter == [([attribute], [terminal])] -> [([attribute], [terminal])]
```

This means that if an interpreter is applied directly to a 'suitably packaged' input string, several results may be returned. For example, suppose that the interpreter **list** of passage#6 had been defined using **$orelse** in place of **$excl_orelse**. Direct application of the modified interpreter to any input with more than one terminal, will give multiple results. For example:

```
list [([], tokenise "5 six 7 8")]
=>
  [([RES [INT_TERM "8",INT_TERM "7",IDENTIFIER_TERM "six",INT_TERM "5"]],[]),
   ([RES [INT_TERM "7",IDENTIFIER_TERM "six",INT_TERM "5"]],[INT_TERM "8"]),
   ([RES [IDENTIFIER_TERM "six",INT_TERM "5"]],[INT_TERM "7",INT_TERM "8"]),
   ([RES [INT_TERM "5"]],[IDENTIFIER_TERM "six",INT_TERM "7",INT_TERM "8"])]
```

Each of these four results is related to a parse of the input as a list. The first result corresponds to a parse of the whole of the input as a list. The last result corresponds to a parse of the input as a singleton list followed by three uninterpreted terminals.

The ability to handle ambiguous input is useful in many applications including natural language processing. However, it is also a fundamental property of the W/AGE system. All interpreters are implemented as top down, fully backtracking, syntax directed, lazy evaluators. There is one major advantage of this: passages are modular. A different, but somewhat related, approach to lazy recursive descent parsing for modular language implementation is described in Koskimies[13]. Many of the arguments given there apply to the parsing strategy that we have adopted in W/AGE.

12 An Example of a Complex Passage

Passage #9 converts expressions of propositional logic to clausal form. If the expression is valid, the interpreter **wff** returns the empty clause set, and may therefore be regarded as a decision procedure for propositional logic.

```
|| Passage #9:
%insert <local/header_for_WAGE_VERSION_1_RELEASE_0.m>
||-----------------------------------------------------------------------
attribute  ::= LITERAL_VAL terminal | CCFSET [disjclause] | CONTEXT [char]
disjclause ::= DISJCL [[char]]
||-----------------------------------------------------------------------
reserved_words  = ["and", "or", "implies"]
special_symbols = ['.', '(', ')', '-']
||-----------------------------------------------------------------------
|| opbr, clbr, period and negate as defined in earlier passages
orr      = uninterpreted (RESERVED_WORD_TERM  "or")
aand     = uninterpreted (RESERVED_WORD_TERM  "and")
implies  = uninterpreted (RESERVED_WORD_TERM  "implies")
var      = literal IDENTIFIER_TERM
||-----------------------------------------------------------------------
wff      =  structure     (s1 expr   ++   s2 period)
              [c_rule 1 (CCFSET  $u lhs)      EQ      (CCFSET $u s1),
               i_rule 2 (CONTEXT $d s1)       EQ      (CONTEXT "pos") ]
```

```
expr        =  structure (s1 var)
                 [a_rule 3  (CCFSET $u lhs)   EQ  make_ccfset
                                                    [LITERAL_VAL $u s1,
                                                     CONTEXT $d lhs]]
               $orelse
               structure (s1 opbr ++ s2 (conjunction $orelse  disjunction
                                         $orelse  implication) ++ s3 clbr)
                 [c_rule 4   (CCFSET  $u lhs) EQ    (CCFSET  $u s2),
                  c_rule 5   (CONTEXT $d s2)  EQ    (CONTEXT $d lhs)]
               $orelse  structure   (s1 negate   ++   s2 expr)
                 [c_rule 6   (CCFSET  $u lhs) EQ    (CCFSET  $u s2),
                  a_rule 7   (CONTEXT $d s2)  EQ  opposite [CONTEXT $d lhs]]

conjunction =  structure (s1 expr   ++   s2 aand   ++   s3 conjunction)
                 [a_rule 8  (CCFSET  $u lhs) EQ  context_and
                                                    [CONTEXT $d lhs,
                                                     CCFSET  $u s1,
                                                     CCFSET  $u s3],
                  c_rule 9  (CONTEXT $d s1)  EQ    (CONTEXT $d lhs),
                  c_rule 10 (CONTEXT $d s3)  EQ    (CONTEXT $d lhs)]
               $orelse structure    (s1 expr)
                 [c_rule 11 (CCFSET  $u lhs) EQ    (CCFSET  $u s1),
                  c_rule 12 (CONTEXT $d s1)  EQ    (CONTEXT $d lhs)]

disjunction =  structure  (s1 expr   ++   s2 orr   ++   s3 disjunction)
                 [a_rule 13 (CCFSET  $u lhs) EQ  context_or
                                                    [CONTEXT $d lhs,
                                                     CCFSET  $u s1,
                                                     CCFSET  $u s3],
                  c_rule 14 (CONTEXT $d s1)  EQ    (CONTEXT $d lhs),
                  c_rule 15 (CONTEXT $d s3)  EQ    (CONTEXT $d lhs)]
               $orelse structure    (s1 expr)
                 [c_rule 16 (CCFSET  $u lhs) EQ    (CCFSET  $u s1),
                  c_rule 17 (CONTEXT $d s1)  EQ    (CONTEXT $d lhs)]
```

The function **sort** is required in the definition of **unite_clauses** in order that **mkset** performs as required.

```
implication =  structure  (s1 expr   ++   s2 implies   ++   s3 expr)
                 [a_rule 18 (CCFSET  $u lhs) EQ   context_or
                                                    [CONTEXT $d lhs,
                                                     CCFSET  $u s1,
                                                     CCFSET  $u s3],
                  a_rule 19 (CONTEXT $d s1)  EQ   opposite
                                                    [CONTEXT $d lhs],
                  c_rule 20 (CONTEXT $d s3)  EQ    (CONTEXT $d lhs)]
||-------------------------------------------------------------------------
```

```
context_and [CONTEXT "pos", x, y]  =  ccf_and  x  y
context_and [CONTEXT "neg", x, y]  =  ccf_or   x  y

context_or  [CONTEXT "pos", x, y]  =  ccf_or   x  y
context_or  [CONTEXT "neg", x, y]  =  ccf_and  x  y

opposite    [CONTEXT "pos"]        =  CONTEXT "neg"
opposite    [CONTEXT "neg"]        =  CONTEXT "pos"

make_ccfset [LITERAL_VAL (IDENTIFIER_TERM v),CONTEXT "pos"]
                                   = CCFSET[DISJCL [v]]
make_ccfset [LITERAL_VAL (IDENTIFIER_TERM v),CONTEXT "neg"]
                                   = CCFSET[DISJCL [negate_lit v]]
||-----------------------------------------------------------------------
|| FUNCTIONS FROM CLAUSE FORM LOGIC
|| The function ccf_and forms the clausal conjunction of two conjunctive
|| clause sets

ccf_and (CCFSET dcs) (CCFSET dcs') = CCFSET (mkset (dcs ++ dcs'))

|| The function ccf_or forms the clausal disjunction of two clause sets
|| tautologous clauses are removed when produced. mkset makes a set
|| from a list

ccf_or (CCFSET dcs) (CCFSET dcs') =
   CCFSET (mkset [newclause | ( DISJCL c1 )- dcs; ( DISJCL c2 )- dcs';]
           newclause - [DISJCL (unite_clauses c1 c2)]; not_taut newclause])

not_taut (DISJCL c) =  [l1 | l1 - c; l2 - c; l1 = (negate_lit l2)] = []

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

unite_clauses c1 c2 = (sort . mkset) (c1 ++ c2)
||-----------------------------------------------------------------------
|| EXAMPLE APPLICATIONS
||
|| apply_interpreter wff "---((p implies q) implies (r implies (s and t)))."
||
||   =>  [[CCFSET [DISJCL ["-r","p","s"], DISJCL ["-r","p","t"],
||                 DISJCL ["-q","-r","s"],DISJCL ["-q","-r","t"]]]]
||
|| apply_interpreter wff "((john_has_money and (john_has_money implies
||                              john_could_pay)) implies john_could_pay)."
||   =>  [CCFSET []]
|| The second example shows how a valid formula is converted to an empty
|| clause form set.Therefore, to see if a formula F is a theorem of a set of
|| formulas S, you simply apply convert to "(S implies F).". If an empty
|| clause set is returned, then F is a theorem of S, otherwise it is not.
```

13 Concluding Comments

13.1 Experimentation with W/AGE

During the last twelve months, W/AGE has been used extensively in a number of application areas. In particular, it has been used in the construction of a sophisticated experimental natural language interface to a database, in the transformation component of a VLSI designer's assistant, and as a teaching aid in a third year 'Grammars and Translators' course. The natural language interface that was constructed using W/AGE can handle both syntactic and semantic ambiguity and provides 'dialogue' answers to user's questions. The interpreter was built as part of an investigation into the feasibility of extending Montague's compositional semantics to accommodate semantic ambiguity. The VLSI project involved the construction of (i) programs to translate mathematical specifications of finite impulse response filters to executable specifications of systolic circuits based on a standard VLSI cell, and (ii) programs to translate the executable specifications to EDIF netlist representations suitable for input to a VLSI layout package. The viability of the approach was confirmed by testing the translators on a real FIR filter design comprising 3 moduli and 64 coefficients. A detailed description of this work is given in Master's theses available from the University of Windsor. W/AGE has also been used in the construction of various other programs in a separate study into the use of the attribute grammar paradigm in constructive (transformational) programming[4].

13.2 Findings

We have found that the integration of the lazy functional programming and attribute grammar paradigms is straightforward. Construction of a programming environment to support this combined paradigm was helped significantly by the declarative nature of the host language Miranda. Some of the more difficult aspects of the W/AGE were constructed using the method of 'programs from proofs' in which induction is used 'in reverse' to design a complex recursive function definition. We chose to implement the syntax analyzers as top-down fully backtracking parsers. It has been argued elsewhere[13] that such parsers are more modular than those built using other strategies. The lazy evaluation order allows attribute evaluation to be closely related to syntax analysis carried out by top-down fully backtracking parsers without incurring the redundant computation that would occur if a strict evaluation order were used.

Our experimentation with W/AGE has convinced us that application of the new combined programming paradigm results in extremely clear and modular executable specifications of language interpreters. However, the actual construction of the interpreters was hindered by the poor debugging facilities of both W/AGE and Miranda. In particular, the absence of trace facilities in W/AGE was a very noticeable shortcoming. Adding a trace facility to W/AGE is not a simple task. The fact that pure functional programming languages do not allow any kind of side effects requires one to adopt a completely different approach to the provision of de-bugging facilities. We hope to overcome this problem in the next few months.

During our investigation, we found that it would have been useful if W/AGE could have supported the construction of language transformers (*ie.* syntactic rewriters) as well as language interpreters. If these transformers are of the same type as interpreters, the two could be combined in various ways enabling wider experimentation in language processing. Such extension of both the programming paradigm and the environment to support it is the subject of our current work.

W/AGE is currently undergoing experimental use at a number of university sites. Potential users can obtain a copy of the W/AGE code through request to `richard@cs.uwindsor.ca`.

The authors acknowledge the assistance of N.S.E.R.C. of Canada, and of Subir Bandyopadhyay and Walid Saba of the School of Computer Science at the University of Windsor.

References

1. B. Edupuganty and B. R. Bryant, Two-level grammar as a functional programming language. The Computer Journal, 32 (1), 36 - 44 (1989).
2. M. S. Feather, A survey and Classification of some program transformation approaches and techniques. In L. G. L. T. Meertens (Editor) Program Specification and Transformation. IFIP 1987. Elsevier Science Publishers B. V. North-Holland.
3. P. Forbig, and U. Lammel, Knowledge based program generation using attribute grammars, in: Grabowski, J (ed), Proc. of the Berliner Informatik Tage bit '89. Akademie d. Wissenschaften der DDR, iir–Report, 114–123, (1989).
4. R. A. Frost, Constructing Programs in a Calculus of Interpreters, Proceedings of the 1990 ACM International Workshop on Formal Methods in Software Development, (1990).
5. R. A. Frost, Constructing programs as executable attribute grammars, The Computer Journal (to appear in the August 1992 issue).
6. R. A Frost, Guarded Attribute Grammars: Top Down Parsing and Left Recursive Productions, ACM SIGPLAN 27(6), 72-76, (1992).
7. E. C. R. Hehner and B. A. Silverberg, Programming with grammars: an exercise in methodology-directed language design. The Computer Journal 26 (3), 227 - 281 (1983).
8. T. Johnsson, Attribute grammars as a functional programming paradigm. Springer Lecture Notes 274, 155 - 173 (1987).
9. T. Katayama, HFP : A hierarchical and functional programming based on attribute grammars, Proceedings of 5th International Conf. on Software Engineering, 343–353, (1981).
10. D. E. Knuth, Semantics of context-free languages. Math. Syst. Theory. 2(2), 127–145, (1968).
11. D. E. Knuth, Semantics of context-free languages: correction. Math. Syst. Theory. 5, 95–96, (1971).
12. D. E. Knuth, Examples of Formal Semantics. Springer Lecture Notes in Computer Science Vol 188, 212–235 (1971).
13. K. Koskimies, Lazy recursive descent parsing for modular language implementation. Software Practice and Experience, 20 (8), 749–772 (1990).
14. T. Panayiotopoulos, G. Papakonstantinou, and G. Stamatopoulos, Attribute grammars and logic programming, Agnew. Inf. No 5 (1988) 227.
15. Y. Shinoda and T. Katayama, Attribute grammar based programming and its environment, Proceedings of 21st Hawaii International Conference on System Sciences, Kailu-Kona, Hawaii, 612–620, (1988).

16. E. Simon, A new programming methodology using attribute grammars, Acta Cybernetica, 7 (4), 425–436 (1986).
17. D. Turner, 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 201. (1985).