The Term Processor Generator *Kimwitu*★

Peter van Eijk[1], Axel Belinfante[2], Henk Eertink[3], Henk Alblas[2]

[1] EDS, P.O. Box 2233, 3500 GE Utrecht, The Netherlands
[2] University of Twente, Department of Computer Science,
P.O. Box 217, 7500 AE Enschede, The Netherlands
[3] Telematics Research Centre, P.O. Box 589, 7500 AN Enschede, The Netherlands

Abstract

The Kimwitu system is a meta-tool that supports the construction of programs (tools) that operate on trees or terms. The system supports open multi-paradigm programming, in that it allows to express each part of an implementation in the most appropriate language. Terms can be implemented in a tool as well as exchanged between tools. In this way tool integration is facilitated. Experience has demonstrated that Kimwitu drastically speeds up development time, facilitates tool integration and generates production quality programs.

1 Introduction

Language-based tools, such as compilers, editors, debuggers, simulators, testers and verifiers, have in common that they operate on terms or trees. Operations of these tools deal directly or indirectly with trees, so that one may conclude that trees and algorithms to manipulate trees are the core of language-based software. Some tools have already gone a long way in the process of maturing. Compilation of high-level programming languages, for instance, is a well-understood process, in that all intermediate tree structures are well thought-out with respect to space and time. Apparently, the reason for this understanding is that high-level languages have much in common. This has stimulated people to develop compiler-generating systems that offer for each compilation phase a tool that generates an efficient implementation from a high-level specification. This does, however, not apply to other less understood tools, such as simulation, verification, test generation and test execution tools. These tools are typically implemented in high-level languages, based on, e.g. attribute grammars or functional languages. This works fine for prototypes, but does not always result in satisfactory space and time performance. Another approach is to use a programming language that allows total control over the space and time consumption of the software, such as C. The disadvantage of languages like C is, however, that the gap between the description of the functionality and its realization is large, implying a long, tedious, and error-prone implementation effort. This means that we actually need a system that bridges this gap, and allows, on one hand, a high-level specification of data structures and the operations on data structures, and on the other hand, low-level directions for the implementation.

Our *Kimwitu* system[vEB92] attempts to blend the advantages of both approaches, in that it supports open multi-paradigm programming. Multi-paradigm programming allows to express each part of the implementation in the most appropriate language. It is a 'best of both worlds' approach, where one uses a high-level language where possible, and a low-level language where necessary. Kimwitu allows one to specify rewrite rules, call them from within C functions, and arbitrarily mix advanced pattern-matching mechanisms over terms with ordinary C code. 'Open' in this context means that escape hatches are provided to other implementation techniques. In Kimwitu, this can be done through the mixing with C code. This allows one to integrate code generated by Kimwitu with, for instance,

★Kimwitu (pronounced 'kee-mweetu') is pidgin-Swahili for 'language of trees'

X-Windows based user interfaces[Eer94], Yacc/Lex parsers or socket-based services (the CLC system [Dub94]).

It has already been mentioned that trees or terms are a basic common concept in language-based software. Terms are, therefore, the basis of the Kimwitu system, in the sense that all formalisms operate on the same kind of structure. This facilitates tool integration, and allows, for instance, interfacing with Lex, Yacc and the Synthesizer Generator[RT89].

2 The Kimwitu System

The Kimwitu system is a *term processor* generator. The basis of its specification language is a notation to describe a term algebra, which defines a set of terms and operations to construct and manipulate terms. Computations over these terms can then be described through a variety of mechanisms, as explained below. Terms can be manipulated in a tool as well as exchanged between tools. In this way tool integration is facilitated because the same term algebra describes both the internal as well as the external representation of values. The description of terms and functions can be arbitrarily split into separate input files. This allows one to separate the specification of well-defined interchange formats from functions that operate over these interchange formats. Typically, a tool-environment for a specific language will use a single, shared, description of the abstract syntax of the language as a Kimwitu input file. This description is re-used for each tool, and therefore allows all tools to read/write files that contain terms according to that abstract syntax specification. This is the core functionality of two large tool environments that have been realized using Kimwitu: the LOTOS [ISO89] tool environment LITE[BvV95] and the SDL Tool environment OpenSITE[Hum]. Individual tools can subsequently define their own function definitions over these, common, abstract syntax terms. This is comparable to IDL specifications as used in the CORBA architecture [OMG95]. IDL, however, is more powerful in that it also supports the definition of method-invocations (which is, by definition, not supported by the file-interchange mechanisms used in Kimwitu), but is also less powerful in that it only supports the definition of interfaces, not of the implementations of these interfaces.

A specification that is input to the Kimwitu compiler consists of a description of terms, annotated with implementation directives, and a description of functions to manipulate terms. Examples of the latter are functions for pattern matching, term rewriting and unparsing. The output (see also section 4) consists of a number of C files that contain data-structure definitions for the terms, a translation of the function definitions, and a number of standard functions to create, compare and manipulate terms, and read and write them from and to files in various formats. Each term-type (called a 'phylum') is mapped onto a C type. This allows us to rely on the ANSI-C typing system for typechecking the C extensions that can be merged in the Kimwitu-specification. The Kimwitu-compiler itself will typecheck all rewrite-rules, patterns, etc., and implements warnings for incomplete pattern specifications (e.g. when partial functions are defined). The choice for C typing was pragmatic; a number of other systems are more powerful in this area as C typing does not, for instance, allow for subtyping. This could be improved by using C++ or Java instead of C, which we will experiment with in future versions of Kimwitu. Additionally, analysis and statistics collecting functions are generated. The generated C files are not intended to be edited by hand. In this sense the Kimwitu 'specifications' are really 'programs'.

3 Input Specifications

In this section we explain the input structure of Kimwitu. We first describe how terms and attributes of terms are defined, next, how the storage strategy of terms can be specified, and finally, how functions, rewrite and unparse rules can be written.

3.1 Defining Terms and Attributes

Terms in the Kimwitu system are specified by means of an abstract syntax or term algebra, in a similar way as in the specification language SSL of the Synthesizer Generator. Terms are defined by rules of the form:

$$x_0 : op\,(x_1\,x_2\,\ldots\,x_n);$$

where *op* is an *operator* name and x_i is a nonterminal, or (in the terminology of abstract algebra) the name of a *phylum*. The phylum associated with a nonterminal is a non-empty set of terms (i.e. a set of trees) that can be derived from it. An example is the following.

```
expr:     Plus(expr  expr)
|         Minus(expr  expr)
|         Neg(expr)
|         Zero()
;

exprlist: list  expr;
```

These rules define the phyla expr and exprlist, each of which denotes a set of terms. This example shows that there are two ways of constructing a phylum. One is by enumerating its variants, each of which is an *operator* applied to a list of phyla. It is possible to declare nullary operators, but it is not possible to define phyla that do not have operators. The other way is declaring a phylum as a list phylum. The definition of exprlist above is equivalent to the following right-recursive definition.

```
exprlist: Nilexprlist()
|         Consexprlist(expr  exprlist)
;
```

A list phylum, therefore, always has a nullary operator to construct an empty list, and a right-recursive binary operator to add an element to an already existing (possibly empty) list. The advantage of a list declaration, apart from its brevity, is that it automatically instructs the system to generate additional, list-specific, functions.

The aforementioned examples show how users can define their own phyla. Kimwitu generates for each phylum a C data type (a record) with the same name. Kimwitu also offers a number of predefined phyla, among them are casestring and nocasestring for case-sensitive and case-insensitive character strings, respectively.

Phyla can be declared to have *attributes* of a predefined type. This type can be any C type, e.g. int or float. It can also be a C type that is generated by Kimwitu, i.e., a phylum. An example phylum with an attribute is:

```
expr:    Plus(expr  expr)
|        Minus(expr  expr)
|        Neg(expr)
|        Zero()
{        float  value  =  0;}
;
```

Here the attribute value of type float is defined, and initialized with 0. Multiple attributes can be defined between the curly brackets. The initializations are optional. Attributes do not appear in structure files.

Attributes serve as a facility to decorate a tree with extra information. The decoration can be done in arbitrary user code. The attribute becomes a component of the record that is generated for the phylum. If x is a value of type expr, then the attribute can be referred to as x→value.

As the last item of the initialization a piece of arbitrary C code, enclosed in curly brackets, is allowed. The code is executed after the term has been built completely, and the other initializations have been performed. This can for example be used to update attribute values while a term is being built. It resembles the constructor functions in e.g. C++.

3.2 Storage Options and Life Time of Terms

The system provides a choice between two storage options, selectable per phylum. For both options a C data type is generated for each phylum, together with a 'create' function for each operator. In the default storage option each operator 'application' just yields a new 'memory cell' containing pointers to the arguments of the operator, with initialized attributes. The second storage option, called 'uniq', is more interesting. It will guarantee that if the operator is once called with a certain set of arguments, each additional call with the *same* arguments will yield a pointer to the cell that was created at the first call. The result is that common (sub)trees (including their attributes) are automatically shared. This technique is known as 'hashed-consing' (because consing is the LISP function to create new cells, and hashing is used to implement the uniqueness of the representation). In this storage option attributes will be initialized only at the first call. Obviously, side effects on subterms can jeopardize this scheme: terms maintained under unique storage should not be modified (though their attributes may be modified because they do not contribute to the uniqueness). An essential condition on phyla definitions is that all constituent phyla of a 'uniq' phylum are also 'uniq'. Kimwitu warns at generation time about violations of this condition.

The 'uniq' storage option has a number of interesting benefits. It gives us automatic sharing of common (sub)expressions. E.g., the Binary Decision Diagram (BDD) package which provides the primitives to implement a routine for solving Boolean equations. depends on this [Kar95].

As a bonus, the common subtree sharing is taken into account when terms are written to a structure file, which may greatly reduce the size of such a file: for the LOTOS abstract syntax tree that is the 'common object' in LITE[BvV95] we found that the difference in file size between no sharing and maximal sharing of common subtrees may be upto a factor 5.

Another benefit is cheap (constant-time) tree-matching. The LOTOS simulator Smile [Eer94] uses Kimwitu trees to represent 'states' in a simulation run, and the tree-matching is used a.o. to check if a certain state already has been analysed.

Finally, the sharing of attributes of common (sub)trees makes it easy to 'simulate' associative arrays that can be used to implement for example symbol tables and memo-functions. An example of such a symbol table is the following. It can be read as a mapping from casestring to short.

```
ID {uniq}: Str(casestring)
{          short type = UNDEF;};
```

Suppose that for each defining occurrence of an identifier a term is created with the attribute type appropriately set, then to check the type one merely has to 'create' it again, and look at the attribute. In the same way one can check if the identifier is already defined at a defining occurrence. A sketch of this code is as follows. It checks that an identifier is defined only once, and defined before use.

```
/* defining occurrence */
id = Str(mkcasestring("foo"));
if (id->type != UNDEF) error("doubly defined");
id->type = USED; /* set other attributes here as well */

/* applied occurrence */
id = Str(mkcasestring("foo"));
if (id->type == UNDEF) error("undefined");
```

Of course, this is not the most sophisticated application of a symbol table, but serves as an example. The LOTOS front-end LCR[Hof95a, Hof95b], a batch-oriented parser and static-semantics checker for LOTOS specifications that conforms strictly to the LOTOS standard, extensively uses this technique to implement its symbol tables. LCR generates the common abstract syntax terms used in LITE. Also the Kimwitu compiler itself implements all its symbol tables using these associative arrays, and most other Kimwitu-built programs use them.

Memo functions are functions that remember ('memorize') their results. If called again with the same arguments, they will return the remembered value. Memo functions are functional in their behaviour: a subsequent call with the same argument will yield the same result. In their performance they are not functional: the subsequent call will not need recomputation. Memo functions of course constitute a time/space trade off. Their performance comes at the expense of memory to store the results (and, in some schemes, memory to store the operands).

The mapping technique illustrated above can easily be used to implement a mapping from the function arguments to the result. Using Kimwitu, memo-functions of one argument can be implemented as an attribute of the phylum of the argument term. Memo-functions of more than one argument can be implemented as an attribute of a uniquely represented term that represents the function call. E.g. for a function F of two arguments one introduces a term F_memo(x,y) of which the function result is an attribute. In both approaches it is essential that the arguments of the function are represented uniquely.

The user is responsible for freeing the storage for trees (terms) that are no longer needed. For terms maintained under normal (non-uniq) storage, this can be done on a per-phylum basis. Terms maintained under uniq storage are stored through hash tables (the create routines use a hash table to guarantee the uniqueness of the representation) and can only be freed on the level of the (a) hash table as a whole, because the 'uniqueness of storage' property should not be violated.

Kimwitu implements default memory- and hash table management routines which are 'open' in the sense that they can be controlled and even overruled by the user, and for example offer the user control over the hash tables used. For normal use the default implementation is sufficient. More advanced memory management can be realized by selectively overriding parts of the default memory management routines. We have done this in Smile to implement a 'stacked heap' mechanism to efficiently clean up intermediate results of computations, both for unique and normal storage. This mechanism essentially does the following. Before a computation is started a new memory 'arena' is allocated. During the computation all results (including the final one) are allocated into this arena. After the end of the computation the final result is copied to the arena that was in use before the temporary one was allocated, and subsequently the temporary arena is freed. This scheme can be used recursively for 'sub-computations' of larger computations, which gives us something like a stack of heaps. We implemented it as a single input file for the Kimwitu system, which allowed easy reuse in other programs. The advantage of this technique is that we don't have to keep additional administration, as would be necessary if we used reference counting techniques, nor do we have to try to find all memory cells no longer in use, as garbage collectors do. Instead, we decide in advance where the intermediate results will go, and clean them up all together when we no longer need them. This technique itself is not particular for Kimwitu. The ability to seamlessly integrate such a user-defined memory management module gives the user much freedom in selecting the most suitable memory management strategy for his application. It made it relatively easy to get efficient memory management in Smile.

3.3 Function Definitions

The structure of the generated C data types (see Section 4) for the phyla is very regular. Nevertheless it appears tedious to write C functions over these data types. Therefore, there is a mechanism that allows easier expression of functions over phyla. This mechanism extends the normal C code with *with-statements* and and *foreach-statements* in which pattern matching can be expressed, which simplifies case analysis and subterm selection. The syntax of the *with-statements* is also borrowed from the language SSL. For example:

```
int len(exprlist el) {
    with(el) {
        Nilexprlist:                    { return 0; }
        tt = Consexprlist(*, t):        { return len(t) + 1; }
}   }
```

Here an integer-valued function len is defined with one argument of type exprlist (for exprlist see Section 3.1). The C code of this function body consists of a *with-statement*, which does pattern matching on its el argument. In the case where more than one pattern matches, the *most* specific (leftmost innermost, see Section 4) match is taken. The patterns can be arbitrary terms with variables, string-literals (double-quoted) and int-literals. Non-leaf variables can be denoted as *variable=subpattern*, as tt in the example above. The construct $*$ can be used to denote an 'anonymous' variable. As a degenerate pattern an operator name *not* followed by parentheses can be used when one is not interested in the (number of) subphyla. The Nilexprlist pattern above is an example of such a pattern. The 'pattern' default can be used to indicate a default case. In case there is no default, the default becomes to give a run-time error message.

For each pattern a piece of C code is given between curly brackets. If several patterns share the same piece of C code, the patterns can be grouped. In this C code, pattern variables denote the various components of the term. Attributes can be referred to as e.g. variable→value.

Another construct in function bodies and C code is the *foreach-statement*, which expresses the iteration over a list. Its components are the loop variable, which automatically gets the type of the list element, the list to loop over, and a body. Another example of the len function:

```
int len(exprlist el) {
    int length = 0;
    foreach( e; exprlist el ) {
        length++;
    }
    return length;
}
```

3.4 Rewrite Definitions

Functional languages are a convenient formalism for expressing functions over trees. Another convenient formalism is formed by *rewrite rules* [EM85]. For instance, if we have a certain equivalence over terms, then rewrite rules expressing this equivalence might define a procedure for computing a normal form of a term. Another use for term rewriting is as an alternative way of defining functions. For example to implement the function 'plus' on natural numbers one can define 'plus' as an operator and specify the rewrite rules such that the normal form does not contain a plus. The result of normalizing (term rewriting) then is that the function is 'evaluated'. The notation for term rewrite rules is simple. For example:

```
Neg(x) -> Minus(Zero(), x);
```

In this example x is a variable, used in the term in the right-hand side. The meaning of this example is that every occurrence of the operator Neg is replaced by an equivalent construct.

Rewrite *views* can be used to define different rewrite systems for the same terms, in the same way as in unparse rules (see Section 3.5).

For the collection of rewrite rules, the system generates a function rewrite_*phylum* for each phylum, which has the normalized form as its result. This function can be called in the same way as any other function. The currently implemented rewrite strategy is left-most inner-most. It is the responsibility of the user to guarantee that the rewrite systems always yields a normal form.

3.5 Unparse Rules

The Kimwitu system generates print functions that print a textual representation of terms in a fixed format to the standard output, but this representation is effectively only useful for debugging purposes. Unparse rules allow the user to describe textual representations of terms, by associating patterns with *unparse items*. Each unparse rule consists of a *pattern*, a list of *views* and a list of *unparse items*. The patterns are the same as those in function definitions and rewrite rules. Views can be used to specify different textual representations for the same term (e.g. a pre-order or a post-order representation of an expression). An unparse item can be any of the following: a string denotation, a piece of arbitrary C code in which pattern variables can be used, a pattern variable, or an attribute of a pattern variable. From the collection of unparse definitions, for each phylum a function unparse_*phylum* is generated. These functions take three arguments: the phylum that will be unparsed, a (void) printer function (to be supplied by the user) that will be applied to each string denotation, and the view to be used. Each unparse item defines a part of an unparse_*phylum* function.

The following example shows unparse rules that contain strings and pattern variables.

```
Plus(e1, e2)                     -> [ : e1 "+" e2 ];
Minus(e1, e2)                    -> [ : e1 "-" e2 ];
Neg(e1)                          -> [ : "-" e1 ];
Zero()                           -> [ : "0" ];

Nilexprlist()                    -> [ : ];
Consexprlist(ex, Nilexprlist())  -> [ : ex ];
Consexprlist(ex, rest)           -> [ : ex ", " rest ];
```

In the case of overlapping patterns, the *most* specific match is preferred. In the example this is used for the output of commas as list element separators. See the last line of the example where this is used to ensure that the number of separators is one less than the number of list elements. For each operator there is always a default pattern, in case none of the patterns match. The unparse rule associated with this default pattern simply unparses all its subphyla sequentially.

The possibility to include C code in unparse rules makes them usable for much more than only formatting the textual representation of a term. They can easily be used to describe arbitrary tree-walks to e.g. check or update the value of attributes. Views can be used to differentiate between different tree-walks. We demonstrate this in Section 5.2.

4 Output

Kimwitu generates a number of C files. They contain data types and functions on those data types.

For each phylum a C data type is generated. Its name is the same as the phylum so it can be arbitrarily used in a C program. Technically, it is a structure containing the attributes, a variant selector (cf. the operator) and a union of the alternatives. Note that this scheme allows type checking over C programs to check if a term is constructed from the correct phyla. An additional data type is YYSTYPE, which can be used in Yacc-generated parsers to construct terms. The generated C code for the example in Section 3.1 is given below. Note, it is rarely necessary to directly refer to these C structures, as function definitions are much more convenient.

```
typedef enum { ..., sel_Neg = 4, sel_Minus = 5, sel_Plus = 6, sel_Zero = 7,
                    sel_Nilexprlist = 8, sel_Consexprlist = 9, ... } kc_enum_operators;

typedef struct kc_tag_expr *expr; /* a 'expr' is a pointer to a 'struct kc_tag_expr' */
typedef struct kc_tag_exprlist *exprlist;

struct kc_tag_expr {
        kc_enum_operators prod_sel;
        union {
                struct { expr expr_1; } Neg;
                struct { expr expr_1; expr expr_2; } Minus;
                struct { expr expr_1; expr expr_2; } Plus;
        } u;
        float value;  /* an attribute */
};

struct kc_tag_exprlist {
        kc_enum_operators prod_sel;
```

```
        union {
                struct { expr expr_1; exprlist exprlist_1; } Consexprlist;
        } u;
};
```

The names of all the unparse views and rewrite views are collected in enumeration types uview respectively rview. Whether or not the user has used them, these types contains a value base_uview respectively base_rview.

For each user-provided function a corresponding C function is generated. Kimwitu also offers a number of C functions to manipulate hash tables, to construct phyla, to rewrite, unparse, and test terms for equality, and to read and write terms from and to a structure file.

The rewrite systems are compiled into C based on the approach described in [Heu88]. The patterns in rewrite rules, unparse rules and user-provided functions are all compiled in the same way. In case of overlapping patterns P and Q, the 'preorder most specific' one takes precedence: we say that P is more specific than Q if in a preorder treewalk of both patterns, at the point where the treewalks diverge, P contains (at least) one node more 'down' in the tree than Q. We have chosen this strategy because it (mostly) frees the user from thinking about the order in which patterns should appear. A disadvantage of this strategy is that it gives the user less control than for example a 'first matching pattern wins' strategy, which makes us infrequently write more pattern rules than is strictly necessary. However, our strategy turns out to work well in practice.

5 Some Special Features

In this section we explain some of the possibilities of rewrite systems, and the use of attributes grammars in Kimwitu.

5.1 Abstract Data Types and Rewrite Systems

The following example illustrates an abstract data type (ADT) style of programming functions. The data type defined here is the type of natural numbers. In ADT theory there is usually no difference between *constructors*, which make up a term in normal form, and *functions*, which can be applied to terms. The difference between these two is only a property of the rewrite system. In the phylum, both of them are operators.

```
/* the abstract data type of natural numbers */
nat:    zero()
|       s(nat)
|       plus(nat nat)
|       mul(nat nat)
|       ack(nat nat)
;

/* rewrite rules for addition, multiplication, and Ackermann's function */
plus(x, zero())  -> x;                      ack(zero(), x)     -> s(x);
plus(x, s(y))    -> s(plus(x, y));          ack(s(x), zero())  -> ack(x, s(zero()));
mul(x, zero())   -> zero();                 ack(s(x), s(y))    -> ack(x, ack(s(x),y));
mul(x, s(y))     -> plus(mul(x, y), x);
```

Here is the main routine to call the function. We build a term corresponding to the application of ack, and then rewrite this into normal form. Print_nat is a generated function that prints its argument term to standard output in a canonical format. Base_rview is the default rewrite view.

```
void main() {
    nat n2 = s(s(zero())), n3 = s(n2);
    print_nat(rewrite_nat(ack(n3, s(n3)), base_rview));
}
```

5.2 Attribute Grammars

Attribute grammars are a formalism where each node, or term, is decorated with a number of *attributes*, of which the value is computed from the values of the subterms of the node or from the encompassing node. In the literature a number of tree-walk evaluators are presented for different classes of attribute grammars[Alb91]. As the name suggests, a tree-walk evaluator is a procedure that traverses the tree. It starts from the root, walks from node to node, performing attribute evaluations until all attributes have been computed and the root is reached again. Most evaluation methods tacitly assume that all attributes are stored in the tree. However, most attributes carry only an intermediate result, and are only used to pass information. It is rarely necessary to keep attribute values in the tree.

In the current version of Kimwitu it is left to the user to specify his own attribute evaluator, which may be a reasonable simple thing to do. Unparse rules can be used to specify treewalks and attribute updates, as has been done in the implementation of the Kimwitu compiler itself. However, the following example shows that the design and implementation of an attribute evaluator can be a complicated task. The example is from the original paper on attribute grammars[Knu68], and computes the value of a fractional binary number, e.g. 1101.01.

The original attribute grammar, as presented by Knuth, is shown below. Note that the syntactic rules are on the left, and the associated evaluation rules on the right. The nonterminals B, L and N stand for *Bit*, *List* and *Number*, and the attributes v, s and l for *value*, *scale* and *length*, respectively.

$$
\begin{array}{lcll}
B & \to & 0 & v(B) = 0 \\
B & \to & 1 & v(B) = 2^{s(B)} \\
L & \to & B & v(L) = v(B), \quad s(B) = s(L), \quad l(L) = 1 \\
L_1 & \to & L_2\ B & v(L_1) = v(L_2) + v(B), \quad s(B) = s(L_1), \\
 & & & s(L_2) = s(L_1) + 1, \quad l(L_1) = l(L_2) + 1 \\
N & \to & L & v(N) = v(L), \quad s(L) = 0 \\
N & \to & L_1.L_2 & v(N) = v(L_1) + v(L_2), \quad s(L_1) = 0, \\
 & & & s(L_2) = -l(L_2)
\end{array}
$$

Below we give the abstract syntax.

```
/* From D. Knuth, Semantics of Context Free Languages */
/* The abstract syntax tree of fractional binary numbers, attributed */
number:     Nonfraction(bitstring)
|           Fraction(bitstring bitstring)
{           float value;        /* synthesized */}
;

bitstring:  Oneb(bit)
|           Moreb(bitstring bit)
```

```
{               float value;        /* synthesized */
                int length;         /* synthesized */
                int scale;          /* inherited */
};

bit:            One()
|               Zero()
{               float value;        /* synthesized */
                int scale;          /* inherited */
};
```

We first present a *demand-driven* evaluation scheme, in which we don't store attributes in the tree. Any synthesized attribute is connected to the root of some subtree. With each combination of a synthesized attribute and a subtree (phylum) we associate a function eval_*phylum_synthesized_attr*. This function takes as arguments the subtree concerned and (some) inherited attributes of the root, and returns the value of the synthesized attribute. The functions below do pattern matching on the function argument that is prefixed with $.

```
/* illustrating attribute evaluation without storing the attributes */
float eval_number_value(number $n) {
    Nonfraction(b):         { return eval_bitstring_value(b,0); }
    Fraction(b1, b2):       { return eval_bitstring_value(b1,0) +
                              eval_bitstring_value(b2, −eval_bitstring_length(b2));}
}

float eval_bitstring_value(bitstring $bs, int scale) {
    Oneb(b):                { return eval_bit_value(b, scale); }
    Moreb(bs_bs, bs_b):     { return eval_bitstring_value(bs_bs,scale+1) +
                              eval_bit_value(bs_b, scale); }
}

int eval_bitstring_length(bitstring $bs) {
    Oneb:                   { return 1; }
    Moreb(bs_bs, *):        { return eval_bitstring_length(bs_bs)+1; }
}

/* pow is a C math library function */
float eval_bit_value(bit $b, int scale) {
    One:                    { return pow(2,(double)scale); }
    Zero:                   { return 0.0; }
}
```

While it is simply enough for a number of cases, there can be some problems with this approach. First, an inherited attribute of a phylum may depend on a synthesized attribute of that phylum. For example bitstring_scale depends on bitstring_length, and the computation of bitstring_length therefore cannot have scale as an argument. An analysis of the attribute dependencies is necessary to prune the argument lists of the functions. Second, as each used occurrence of a synthesized attribute is represented as a call to the corresponding function, attributes may be evaluated more than once. This is of course the other side of not storing results in the tree.

We now present an evaluation scheme that visits the tree a number of times, computes at each visit of a node all the attributes that can be computed, and stores their values in the tree. In the implementation we use unparse rules and give each pass its own view. As unparse items (see Section 3.5) we use pattern variables (to recursively unparse, or visit, the corresponding subterms), and (between braces) C code (to update attributes). In the C code $0 represents the term being unparsed. In our example there are two passes. In the first pass the attribute length is computed, and in the second pass the other attributes. Again, this scheme has its disadvantages. The allocation of attributes to passes has to be derived from an analysis of the attribute dependencies. Second, in comparison with the previous scheme, this one represents the opposite time/space trade-off. No attribute is evaluated more than once, but at the expense of storing all intermediate results. Finally, this scheme does not coexist very well with unique storage of phyla that have inherited attributes. Two occurrences of a phylum cannot be shared if they have different inherited attributes.

```
/* illustrating a multi-pass evaluation, using unparse rules */
%uview pass1, pass2;           /* declare unparse views */

/* rules for phylum number, pass1: */
Nonfraction(b)      -> [pass1:   b ];
Fraction(b1, b2)    -> [pass1:   b1 b2 ];

/* rules for phylum bitstring, pass1: */
Oneb(*)             -> [pass1:   {$0->length=1;} ];
Moreb(bs, *)        -> [pass1:   bs {$0->length=bs->length+1;} ];

/* rules for phylum number, pass2: */
Nonfraction(b)      -> [pass2:   {b->scale=0;}  b  {$0->value=b->value;} ];
Fraction(b1, b2)    -> [pass2:   {b1->scale=0; b2->scale= -b2->length;}
                                 b1 b2
                                 {$0->value=b1->value+b2->value;} ];

/* rules for phylum bitstring, pass2: */
Oneb(b)             -> [pass2:   {b->scale=$0->scale;} b {$0->value=b->value;} ];
Moreb(bs, b)        -> [pass2:   {b->scale=$0->scale; bs->scale=$0->scale+1;}
                                 bs b
                                 {$0->value= bs->value + b->value;} ];

/* rules for phylum bit, pass2 (pow is a math library function): */
One()               -> [pass2:   {$0->value=pow(2,(double)$0->scale);} ];
Zero()              -> [pass2:   {$0->value=0.0;} ];
```

Here follows the C code to call the attribute evaluations.

```
number n = Fraction(Moreb(Moreb(Moreb(Oneb(One()),One()),Zero()),One()),
                    Moreb(Oneb(Zero()), One())); /* 1101.01 */
printf(" %f \n", eval_number_value(n));
unparse_number(n, 0 /*no strings to print*/, pass1); unparse_number(n, 0, pass2);
printf(" %f \n", n->value);
```

The current version of Kimwitu does not prescribe a particular evaluation scheme. The advantage is that schemes can be mixed at liberty, and can even be combined with non-attribute grammar paradigms. The disadvantage is of course that the evaluation order, e.g. the allocation of attributes to passes or visits, has to be determined manually, or by using some other tool.

A next version of Kimwitu will therefore provide an attribute evaluator function, based on an approach of Jourdan[Jou84], which meets the above-mentioned demands. It will be based on the class of absolutely non-circular attribute grammars, which is a class that seems to include every practical example. Attributes will not be stored in the tree, unless the user stipulates otherwise. Analysis of the attribute dependencies will be used to determine the argument list of the evaluation functions (i.e., the inherited attributes that are needed to compute a synthesized attribute). Evaluation by need will guarantee that only (or almost only) those attributes are computed that are needed to compute the synthesized attributes at the root of the tree. Memo-functions will be used to avoid that attributes are evaluated more than once.

6 Experiences

Kimwitu has proven to be a powerful tool that has been used to develop several language-based tools, such as compilers, simulators, testers and verifiers, and as 'glue' between tools in toolkits. The system has been in use now for more than 6 years, and has proven to be stable enough for production quality tools.

Common uses of Kimwitu exploit one or more of the following features: the easy interface with Yacc and Lex, to build abstract syntax (parse) trees; pattern matching and rewrite rules, to manipulate terms (trees); unparse rules, both to describe the textual representation of terms, and to specify tree-walks; the unique storage option, to get state matching almost for free, but also to be able to use associative array-like functionality in C programs; the ability to manipulate hash tables, to get efficient memory management with little effort; the ability to read and write from and to structure files, to interface between tools in toolkits. It is our experience that users need some time to 'digest' the features offered by Kimwitu to be able to select the most 'natural' way to describe their algorithms in Kimwitu, but in this Kimwitu does not differ from other advanced systems.

The first use of Kimwitu was to implement the Kimwitu compiler itself. It uses Yacc and Lex for parsing and scanning, and makes heavy use of unparse rules for type checking, and (C) code generation. The development of the Kimwitu compiler gave us early feedback about the practical usability of our ideas.

Our main experience with Kimwitu has been the work on LOTOS [ISO89] tools, which was part of the Lotosphere (Esprit II) project[BvV95]. In the Lotosphere project an integrated toolset, LITE, has been built for LOTOS. Every tool in this toolset works on a central object, which is a representation of a LOTOS specification. This object is formally described in 525 lines of Kimwitu input. Kimwitu generates data structures and I/O routines from this description. This makes changes to the structure of the interface object rather easy — in most cases programs only have to be recompiled. The fact that the specification of the central object is used for both the external and the internal representation simplifies the production of tools that work on the central object. In one case, a person with no experience in C or Kimwitu produced a conversion tool in one week.

A compiler for equational systems into code for specialized abstract term rewriting and narrowing machines [Wol91] has been produced using Kimwitu in three man-months, which was about half of the planned development time. This system is described in 2900 lines of Kimwitu input. In particular the automatic management of symbol tables proved very helpful. The speed of the resulting program is comparable with earlier versions, which were written in C. The interpreters for the abstract machines were written in 2600 lines of C.

The full LOTOS simulator Smile[Eer94] has been built in 6 man-months. This simulator does extensive manipulation of complex data structures. The size of Kimwitu code is about 4000 lines (112 Kb) with an additional 1200 lines of C code for the X-Window based user-interface. These numbers do not include the abstract data type part mentioned previously.

A previous system, Hippo [vEVD89], was implemented in 20,000 lines of Yacc, Lex and C, of which 5000 lines are devoted to the abstract data type part. Its development took 18 man-months. It is now hard to extend, and maintenance on it has been stopped.

Generally speaking, Smile has more functionality than Hippo: it is fully symbolic and its abstract data type part is much more advanced. The memory consumption of Smile is less and the execution speed is better (both by a factor of 2 or 3), on a comparable run.

The following gives an indication of the performance of the generated code. A 3195 line LOTOS specification results in a structure file of 780 Kb, containing approximately 200,000 operator applications. Reading this object, initializing the simulator, and compiling the abstract data types takes 18 seconds of cpu-time on a SparcStation 1.

7 Conclusions

The Kimwitu system improves productivity, is relatively easy to learn, and produces efficient code. The novelty of our approach is to allow a variety of formalisms to be used in the construction of language-based software. We do not claim novelty in the formalisms used, but rather in their combination. We believe that our system is a significant tool in the implementation of programming and specification languages.

Availability and Contact

Kimwitu is available from the internet, via URL <http://wwwtios.cs.utwente.nl/kimwitu/>.
The Kimwitu developers can be contacted at <kimwitu@cs.utwente.nl>.

References

[Alb91] Henk Alblas. Attribute evaluation methods. In H. Alblas and B. Melichar, editors, *International Summer School SAGA: Attribute Grammars, Applications and Systems*, number 545 in LNCS, pages 48–113, Prague, 1991. Springer-Verlag.

[BvV95] T. Bolognesi, J. van de Lagemaat, and C.A. Vissers, editors. *LOTOSphere: Software Development with LOTOS*. Kluwer Academic Publishers, 1995.

[Dub94] Eric Emile Dubuis. *Compiling the Behaviour Part of LOTOS*. PhD thesis, ETH Zurich, 1994. TIK-Schriftenreihe nr. 3.

[Eer94] E.H. Eertink. *Simulation Techniques for the Validation of LOTOS Specifications*. PhD thesis, University of Twente, 1994.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[Heu88] T. Heuillard. Compiling conditional rewriting systems. In Stéphane Kaplan and J.-P. Jouannaud, editors, *Conditional Term Rewriting Systems*, volume 308 of LNCS. Springer-Verlag, 1988.

[Hof95a] A. Hofkamp. The implementation of a compiler front-end for LOTOS. In *Gronics '95 Workshop*, pages 65–70, Groningen, The Netherlands, February 1995.

[Hof95b] A. Hofkamp. A static semantics checker for LOTOS. Master's thesis, University of Twente, Enschede, The Netherlands, 1995.

[Hum] OpenSITE: SDL Integrated Tool Environment.
URL: http://www.informatik.hu-berlin.de/Themen/SITE/.

[ISO89] ISO. Information processing systems — open systems interconnection — LOTOS — a formal description technique based on the temporal ordering of observational behaviour. International Standard 8807, ISO, Geneva, February 1989. 1st Edition.

[Jou84] M. Jourdan. Strongly non-circular attribute grammars and their recursive evaluation. In *SIGPLAN '84 Symposium of Compiler Construction*, pages 81–93. ACM, June 1984.

[Kar95] Pim Kars. Representation of process-gate nets in LOTOS and verification of LOTOS laws: The boolean algebra approach. In *Proc. FORTE '94*, 1995.

[Knu68] D.E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–145, 1968. A correction appears in vol. 5 pp 95-96.

[OMG95] OMG. The Common Object Request Broker Architecture and Specification. Object Managment Group, Framingham, MA, 1995. Revision 2.0.

[RT89] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator - A System for Constructing Language-Based Editors*. Springer-Verlag, New York, 1989.

[vEB92] Peter van Eijk and Axel Belinfante. The termprocessor *kimwitu*: manual and cookbook. Technical Report INF-92-67, University of Twente, Enschede Netherlands, 1992.

[vEVD89] P.H.J van Eijk, C.A. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS - results of the ESPRIT/SEDOS project*. North-Holland, Amsterdam, 1989.

[Wol91] D. Wolz. Design of a Compiler for lazy pattern driven narrowing. In *Recent Trends in Data Type Specification. Proceedings of the 7th international workshop on specifications of abstract data types*, number 534 in Lecture Notes in Computer Science, Wusterhausen-Dosse, 1991. Springer-Verlag.