The Term Processor *Kimwitu*
Manual and Cookbook

Peter van Eijk Axel Belinfante

April 17, 2000

License

Kimwitu, a system that supports the construction of programs that use trees or terms as their main data structure.

Contact

Feedback about *Kimwitu* is welcome at:

Kimwitu Distribution
Formal Methods and Tools Group
Department of Computer Science
University of Twente
P.O. BOX 217
NL-7500 AE ENSCHEDE
THE NETHERLANDS

email: <kimwitu@cs.utwente.nl>

More information about *Kimwitu* can be found on the web at:

<http://purl.oclc.org/net/kimwitu>

About this document

The Term Processor *Kimwitu* Manual and Cookbook, Version 10a.
This version is identical to Version 10, except for this page (the license text), a few bugfixes, and a change from LaTeX to LaTeX2e, and the use of the hyperref package.
This manual documents *Kimwitu* version 4.5.

Contents

Introduction

When all else fails, read the manual.
Anonymous.

In this document we describe a system that supports the construction of programs that use *trees* or *terms* as their main data structure. This system is a 'meta'-tool in the development process of tools. Its *input* is an abstract description of terms, annotated with implementation directives, plus a description of functions on these terms. The *output* consists of a number of C-files that contain data structure definitions for the terms, a number of standard functions on those terms, and a translation (in C) of the function definitions in the input (e.g. term rewriting). The standard functions can be used to create terms, compare them for equality, read and write them on files in various formats and do manipulations like list concatenation.

This document is organised as follows. In Section 1 we describe the input format of the term processor[1]. In Section 2 we describe the output of the system: the C types and functions, the files, etc. Section 3 goes into the details of running the system, and the relation with other software development tools, such as the Unix tools make, yacc, lex, lint and the Synthesizer Generator (SG)[TR84, TR89b, TR89a]. Section 4 discusses various techniques in which programs can be written over terms, and thus gives examples of the concept of *multi-paradigm programming.* Finally, Section 5 motivates some of the design decisions of our system.

Some reading knowledge of the C programming language and associated tools is assumed.

1 Input

This section gives the input structure of *Kimwitu*. We describe how terms are defined, how attributes of terms are defined, how the storage strategy of terms can be specified, and how functions and rewrite rules on terms can be written.

1.1 Defining Terms

The input structure of terms is borrowed from the Synthesizer's Specification Language SSL. An example is the following.

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

exprlist: list expr;
```

This declares two *phyla*, or term types, or nonterminals, depending on your viewpoint. Each of these denotes a set of terms. As shown, 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

[1] There is no need to distinguish programming languages from their processors. Yet, calling our notation and tool a 'term language' would allow unintended alternative interpretations. We see our notation and tool as only one of a class of term processors, it therefore has its own name. Colloquially we refer to it either as 'the term processor' (being our only one) or as 'Kimwitu' (pronounced 'kee-mweetu', stress on the second syllable).

declare nullary operators, but it is not possible to define phyla that do not have operators. The other way is declaring it as a list phylum. This is effectively equivalent to declaring the following right recursive phylum:

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

A list phylum therefore always has an empty list constructor, and a prefix operator. The advantage of a list declaration, apart from its brevity, is that it instructs the system to generate additional, list-specific, functions.

There are a number of predefined phyla, among them are casestring and nocasestring for case-sensitive and case-insensitive character strings respectively. The full list of predefined phyla is in Section 2.3.

Phyla can be defined more than once, and at each occurrence operators, attributes, and storage options (see below) can be added.

For each phylum, the term processor generates a C data type (a record) with the same name. This is discussed in more detail in Section 2.1.

1.2 Attributes of Terms

Phyla can be declared to have *attributes* of a predeclared type. This type can be any C type, e.g. int, or float. Of course, it can also be a C type that is generated by *Kimwitu*. An example phylum with attributes is:

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

Here the attribute value of type float is defined, and initialised with 0. Multiple attributes can be defined between the curly brackets. The initialisations are optional. The type of an attribute can also be a type generated by the term processor (such as a phylum).

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 initialisation a piece of arbitrary C, enclosed in curly brackets, is allowed. In this code the expression $0 denotes the term that is being created. The code is executed after the term has been built completely, and the other initialisations have been performed. An alternative way of expressing the above initialisation therefore is:

```
{          float value;
           { $0->value=0;}}
```

1.3 Storage Options

The system (currently) provides two storage options, selectable on a per phylum basis. For both of them 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 initialised 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 by the first call. The result is that common (sub)trees 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 this uniqueness of representation property). In this storage option attributes will be initialised 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 example application is as follows.

```
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, e.g. through Str(yourstring) 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 its 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 example of a symbol table, but serves as an example.

An essential condition on phyla definitions is that all constituent phyla of a 'uniq' phylum are also 'uniq'. The term processor warns at generation time about violations of this condition.

As will be explained in Section 4.3, in some cases a phylum that has inherited attributes should *not* be stored 'uniq'. The following example declares ID as explicitly 'non-uniq'.

```
ID {! uniq}:;
```

Phyla may not be declared both {uniq} and {!uniq}.

1.4 Life Time of Terms

Terms maintained under non-unique storage can be freed using a free_*phylum* call. Terms maintained under unique storage cannot be freed individually. They are stored via hashtables, and their lifetime can only be controlled by manipulation of their hashtables. By default, Kimwitu stores all unique terms in the same hashtable. The following manipulations on hashtables are defined: creation, assignment, deletion and reuse. New, initially empty, hashtables of arbitrary size can be created and used instead of the default hashtable. A newly created hashtable is not used directly after creation, but only after it has been 'assigned' - from that moment on it will be used instead of the previously assigned hashtable, until another hashtable gets assigned. A 'de-assigned' hashtable is not freed; all its memory cells remain alive, so pointers to those cells remain valid. However, they are invisible to the create routines, and thus the unique storage guarantee no longer holds in the following sequence of events: an (uniquely maintained) operator is created with a certain set of arguments, and a memory cell for it is created in the hashtable, and the initialization code for its phylum is executed; a new hashtable is created and assigned; the same operator is called with the same set of arguments, a new memory cell for it is created in the new hashtable, and the initialization code for its phylum is executed again. A 'de-assigned' hashtable can be freed when its memory cells are no longer needed.

This can be used to control the lifetime of unique terms quite effectively, eg. to free intermediate results of a computation, as follows. First, a 'temporary' hashtable is created to hold the intermediate results. Then, just before the computation is started, replace the default hashtable with the temporary one. When the computation is finished, the intermediate results and the final result are in the 'temporary' hashtable. Re-assign the 'old' hashtable, and copy the final result[2], to re-create the memory cells belonging to the final result in the 'old' hashtable. Finally, free the 'temporary' hashtable.

Often only a limited number of phyla appears in the intermediate results. To allow more precise control, the phyla maintained under unique storage can be partitioned over multiple *storage-classes*, that each can have its own hashtable assigned. The memory cells created for a phylum are created in the hashtable assigned to the phylum's storage-class. Usually, phyla with a different 'lifetime' will be in different storage classes. There is one predefined storage-class: *uniq*. Other storage-classes can be defined and used as follows:

```
%storageclass intermediate wholeprogram;   /* the '%' is part of the keyword */

ID2 { intermediate }: Operator1( ... ) | Operator2( ... ) | ... ;
ID3 { wholeprogram }: Operator3( ... );
```

Here ID2 and ID3 belong to different storage-classes; if different hashtables are assigned to intermediate and wholeprogram, then the memory cells for Operator1 and Operator2 will be in another hashtable than those for Operator3.

A phylum can be in only *one* storage class. The term processor reports an error if this condition is violated. The %storageclass declaration is optional; if *Kimwitu* input does contain a storage class declaration, errors will be reported for all storage classes that were not explicitly declared.

For each storage class, Kimwitu creates statically (at compile time) a hashtable. Dynamically, hashtables can be created, (re)assigned to one or more storage classes, cleared and freed. The hashtable functions are listed in Section 2.2.

If the default (statically allocated) hashtable is not used, ie. if all storage classes are reassigned to dynamically created hashtables *before they are used*, it is useful to redefine LARGEPRIME :

```
%{ KC_TYPES_HEADER
#define LARGEPRIME 0
%}
```

[2]Special care has to be taken of attributed terms, because the copy_*phylum* functions do at most copy the values of the attributes - if this value is a pointer to a term, make sure that this term is copied as well, if it is needed...

or use the -DLARGEPRIME=0 flag with cc during compilation of file k.c (see Section 2.5), to avoid the creation of the statically allocated hashtable.

1.5 Function Definitions

The structure of the generated C data types (see Section 2.1) 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. In this way case analysis and subterm selection is simplified. For example:

```
int len(exprlist $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 1.1). The function does pattern matching on the argument that is prefixed with $. In the case where more than one pattern matches, the *most* specific 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 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 (with separating commas). In this C code, pattern variables denote the various components of the term. The values $1 etc. denote the subphyla of the term: in the example above len(t) could also be written as len($1). The value $0 denotes the term itself. Attributes can be referred to as e.g. variable→value.

Alternatively, function bodies can be an arbitrary piece of C code. This code can contain *with-statements*, in which the same pattern matching can be expressed. For example, an alternative description of the function len is:

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

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. Yet another example of the len function:

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

The *foreach-with-statement* is useful if the body of the foreach loop consists of only a with-statement for the loop variable. Then the same syntactical shorthand as for the function definitions can be used:

```
expr sum(exprlist el) {
    expr sub_total = Zero();
    foreach( $e; exprlist el ) {
        Add( x ):          { sub_total = Plus( sub_total, x ); }
        Subtract( x ):     { sub_total = Minus( sub_total, x ); }
    }
    return sub_total;
}
```

The following *foreach-pattern-statement* is useful if there is only one interesting pattern. Instead of a loop variable it takes a pattern. The body is only executed for those list elements that match the pattern.

```
expr add_Adds(exprlist el) {
    expr all_Adds = Zero();
    foreach( Add( x ); exprlist el ) {
        all_Adds = Plus( all_Adds, x );
    }
    return all_Adds;
}
```

It is also possible to do pattern matching over multiple expressions in one with-statement and loop over several lists in a foreach-statement. The syntax for it is a straightforward extension of the 'singular' case. For example:

```
boolean equiv(expr $a, expr $b) {
    Add( asub ) & Add( bsub ), Subtract( asub ) & Subtract( bsub ), Const( * ) & Const( * ):
            { return equiv( asub, bsub ); }
    Plus( asub1, asub2 ) & Plus( bsub1, bsub2 ), Minus( asub1, asub2 ) & Minus( bsub1, bsub2 ):
            { return equiv( asub1, bsub1 ) && equiv( asub2, bsub2 ); }
    default: { return False; }
}
```

Here we compare two expr trees: if they have the same tree structure (form) we say that they are equivalent (and we don't care about the constant values in the leaves). For each dollar-prefixed argument we have a pattern; the patterns are grouped together with separating ampersand (&). Pattern groups that share the same piece of C code are grouped together with separating commas.

Pattern-variables may appear multiple times in the patterns of an ampersand-linked pattern group, to indicate that subtrees have to be (structurally) equivalent. We can even use that to 'parameterize' a pattern:

```
boolean has_sub(expr $a, expr $b) {
    Add( asub ) & asub = * : {
        /* code here will be executed if 'a' has top-operator 'Add', and asub == b */
    }
    default: { return False; }
}
```

If comma-separated pattern groups share a common pattern or ampersand-linked pattern group, it can be factored out. For example, the two pattern groups below are equivalent:

```
Add( asub ) &   Add ( bsub ) , Add( asub ) & Subtract ( bsub )   : { /* C-code */ }

Add( asub ) & ( Add ( bsub ) ,                 Subtract ( bsub ) )  : { /* C-code */ }
```

The foreach-statement over multiple lists is actually a combination of the foreach-statement, the foreach-with-statement and the foreach-pattern-statement. It loops over all lists at the same time, as long as each list still contains elements. For example:

```
boolean equiv_lists(exprlist el1, exprlist el2) {
    boolean result = True;
    foreach( e1 & e2; exprlist el1, explist el2 ) {
        /* this body is executed as long as both lists have elements */
        result = result && equiv( e1, e2 );
    }
    /* we don't know if one list is longer than the other; we only */
    /* know the 'result' for the elements that we compared with 'equiv' */
    return result;
}
```

Here we have a function in which we check that the elements of two lists are pairwise equivalent. At the right of the semicolon two list expressions appear (separated by commas, and each prefixed with its type). At the left of the semicolon a corresponding number of foreach-items appears (seperated by ampersands), where each foreach-item is either a variable (as in a foreach-statement), a variable prefixed with a dollar (as in a foreach-with-statement) or a pattern (as in a foreach-pattern-statement). The body contains either C-code, or patterns with C-code (if one or more foreach-items was a dollar-prefixed variable).

Our equiv_lists function has one disadvantage: the body of the foreach is executed for each pair of elements, as long as all (both) lists are non-empty, so, to know whether the two lists have the same length, we would have to test explicitly for the length. There is an experimental feature to get around this: it is possible to specify a clause that will be executed after the end of the foreach body, in which variables are bound to (or patterns are matched with) the remaining sub-lists:

```
boolean equiv_lists(exprlist el1, exprlist el2) {
    boolean result = True;
    foreach( e1 & e2; exprlist el1, explist el2 ) {
        /* this body is executed as long as both lists have elements */
        result = result && equiv( e1, e2 );
    } afterforeach( $re1 & $re2 /* same number of items here as in foreach */ ) {
        Nilexprlist() & Nilexprlist() : { /* both lists same length: result unchanged */ }
        default: { /* lists have different length: result changed */ result = False; }
    }
    return result;
}
```

The argument of the afterforeach has the same number of items as the corresponding foreach. The items can be patterns (over the listtype, not the listelement type), dollar-prefixed variables and ordinary variables. The body, which can contain C-code, or patterns (if there are dollar-prefixed variables), is exectuted exactely one time.

Actually, the use of patterns and dollar-prefixed variables are just syntactical sugar for a foreach statement with only variables, but with nested with-statements in its body. For example, the two foreach-statements below are equivalent:

```
/* here we combine all kinds of items: patterns, dollar-prefixed variables
 * and ordinary variables. Of course, the items can appear in arbitrary order.
 */
foreach( pattern1 & ... & patternk & $dvar1 & ... & $dvarn & var1 & ... & varm ; ... ) {
          /* body, in which we can refer to
           * pattern variables, dollar-prefixed variables (dvar*)
           * and ordinary variables (var*)
           * but (most often) not to dollar-variables ($i, i >= 0)
           */
}

/* here we have the same statement, in which we only use ordinary variables,
 * (and we introduced 'anonymous' variables for the patterns pat* ),
 * together with nested with statements.
 */
foreach( var_pat1 & ... & var_patk & dvar1 & ... & dvarn & var1 & ... & varm ; ... ) {
          with( var_pat1, ..., var_patk ) {
                    pattern1 & ... & patternk : {
                              with( dvar1, ..., dvarn ) {
                                        /* body, in which we can refer to
                                         * pattern variables, dollar-prefixed variables (dvar*)
                                         * and ordinary variables (var*)
                                         * but (most often) not to dollar-variables ($i, i >= 0)
                                         */
}         }         }         }
```

Most C code can contain so-called dollar-variables: expressions of the form $i where i is 0, 1, 2, ... The context in which the dollar-variables are used defines whether dollar-variables may be used, and to which values they are bound. The dollar-variables may *not* be used in a with-statement over more than one expression, or in a foreach-statements that induce such a with-statement: foreach-statements with more than one dollar-prefixed variable, and foreach-statements without dollar-prefixed variables but with more than one pattern. The dollar-variable $0 is bound to one of the following: the C-expression that is the argument of a with-statement, the function- or foreach-parameter that is preceded by a $, the pattern of a foreach-pattern-statement, or the phylum that is being created, when used in phylum-initialisation code. A dollar-variable $i, i > 0 is bound to the i-th subterm of the enclosing with-, foreach-with- or foreach-pattern-statement. Note that a foreach-statement does *not* introduce dollar-variables. In case of multiple candidates for bindings, the one from the smallest enclosing scope is chosen.

It is seldom necessary to use dollar-variables, in most cases pattern variables can be used; the only exception is $0 used in phylum-initialisation code. We strongly advise the use of patterns and pattern-variables, instead of degenerate patterns and dollar-variables, if only because it allows the term processor to warn against inconsistencies in its input. For example, if the number of subphyla of an operator is changed, the term processor will report an error if a pattern was not updated accordingly; however, it will not notice it when a dollar-variable was not updated to reflect the same change (lint may catch this, but there are cases that lint cannot catch).

There is one use of dollar-variables that cannot be achieved in another way: in-place modification of terms. Try to avoid it, because it is quite easy to mess things up. Do not try to in-place modify terms maintained under unique storage. If you really need to in-place modify a term, you have the following options: In phylum initialization code (see Section 1.2), an assignment to $0 will modify the term under construction. In all other C texts an assignment to $0 is equivalent to a no-op. In those contexts, it *is* possible to assign to $$i$ ($i > 0$), or to do something like: ... replacement_term = ...; *$0 = *replacement_term; ... , but this last solution may not be completely portable (it assigns a struct to a dereferenced (*) struct pointer).

1.6 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 normalising (term rewriting) then is that the function is ‘evaluated’. (This is all abstract data type theory.) The notation for term rewrite rules is very 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.

In general both sides are terms with variables, where all variables of the right-hand side also appear on the left-hand side. Actually, the left-hand side can be the same kind of pattern that is allowed in function definitions. To allow ‘greater control’ (a euphemism for hacking...) we allow arbitrary C functions and string- and int-denotations in the right-hand side (see below). For the collection of rewrite rules, the system generates a function rewrite_*phylum* for each phylum, which has the normalised form as its result. This function can be called in the same way as any other function.

The rewrite functions have an additional argument of type rview. Rewrite views can be used to group rewrite rules in separate rewrite systems. Rewrite rules can be made specific for a set of rewrite *views* (or, added to one or more rewrite systems) by naming these views between the angle open bracket and the colon of a rewrite rule. For example:

```
Neg(x) -> < view1 view2: Minus(Zero(), x) >;
```

An omitted view list defaults to all rewrite views. If no views are specified, the angle brackets and colon may be omitted (as we usually do), as we show below for the rewrite rule for Neg(x) that we gave above.

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

This is equivalent to a view list that only contains the view name base_rview. The type rview is an enumeration type of all view names occurring in the rewrite rules. It always contains the view name base_rview.

Rewrite views can be declared as in the following example:

```
%rview view1 view2; /* the ‘%’ is part of the keyword */
```

Rewrite views should be declared. If the termprocessor input contains one or more rewrite view declarations it will report errors for all rewrite views that are used and not declared. The use of view declarations may help to find typing errors in view names – in term processor input that does not contain view declarations, the misspelling of a view name may just introduce a new view (with the misspelled name).

If several rewrite rules share the same left-hand side, they may be combined by grouping the right-hand sides (with seperating commas), provided that all the angle brackets haven't been omitted from the right-hand sides. If several rewrite rules share the same right-hand side, they may be combined by grouping the left-hand sides (with seperating commas). The general form of a rewrite rule is shown below:

```
pattern1, pattern2, ... -> < v1 v2 ... : ... >, ..., < vn ... : ... > ;
```

How we can prove that a rewrite system always yields a normal form is beyond the scope of this manual, but informally stated, each rule should bring the term 'closer' to its normal form, and the order in which the rules are applied (the *rewrite strategy*) should not matter. Two simple warnings: never ever use a rule in which the right-hand side is equal to the left-hand side, or a rule that directly expresses a commutative property.

The default rewrite strategy is leftmost innermost. In the case of overlapping patterns, where more than one left-hand side matches, the *most* specific match is taken.

Rewrite rules with functional right-hand sides are a bit harder to prove correct. The first requirement of course is that the arguments and the results of those functions should have the correct types, which lint can check. The second requirement is that the rule is meaningful, but that is hard to check automatically. The writer of such rules should convince herself of the correctness of the rewrite rules. Here we give some considerations that can be used in doing so. The 'purpose' of a rewrite system is to convert a term to its *normal form*. Now let us look closer at a rule with a function in it:

```
Operator(x) -> Function(x);
```

The rewrite system uses this rule when it has a subterm that matches the left-hand side, and 'assumes' that substituting it by the right-hand side brings it closer to the normal form. It then continues applying rules on the new term. The function Function should therefore bring the expression closer to the normal form. Now what about the argument x of Function? It is not known whether or not it is in normal form already (although the default strategy, which is leftmost innermost, does guarantee this). Therefore, Function cannot assume that its argument is in normal form. This is not usually a problem, but if it is, the function rewrite_*phylum* can be used in the function definition to guarantee this. The other side of the picture is that the *result* of Function does not have to be in normal form, which is convenient for the writer of Function.

1.7 Unparsing Definitions

Unparsing definitions describe textual representations of terms. The definitions describe a tree walk over terms by associating *unparsing rules* with patterns. For the collection of unparsing definitions *Kimwitu* generates a function unparse_*phylum* for each phylum. The patterns are the same patterns as can occur in rewrite definitions and with-statements. An unparsing rule contains a list of unparse *views* and a list of *unparse items*. We'll discuss the views later. An unparse item defines an action for an unparse_*phylum* function and can be any of the following.

`"a string"` Each string denotation will be printed as such.

{ ... } Between curly brackets, arbitrary C code can be used, in which the pattern variables and `$0` (which is a reference to the entire left hand term) can be used. This C text is inserted in the unparse function. The C text can contain curly brackets, nested in matching pairs. If a non matching bracket needs to be included it can be escaped with a `$`, e.g. `${` and `$}`.

variable An occurrence of a pattern variable is translated into an invocation of the unparse function for the phylum of the pattern variable.

variable→attribute An occurrence of an attribute is translated into an invocation of the unparse function for the phylum or type of the attribute. An attribute of an attribute is denoted by *variable→attribute→attribute* etc. The unparsing function for a type that is not a phylum has to be provided by the user in a separate piece of C code.

(typename)variable A prefixed cast has the effect that the unparse function for that type will be called for the variable (or its attribute). This is useful for the unparsing of non-pattern variables, eg. variables that are defined in C code.

variable:view An appended unparse view name (see below) has the effect that the unparse function for the variable will be called with this view rather than the current view. This can also be done with attributes, casted variables and their attributes, and even strings.

`${` *unparse items* `$}` A list of unparse items enclosed by escaped bracket is translated into a curly open-bracket followed by the translation of the *unparse items* followed by a curly close-bracket.

How an unparse item is actually printed will be discussed later. First an example, containing 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. This is used in the example for the introduction of list element separators, see the last line of the example. 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 unparsing rule associated with this default pattern unparses all its subphyla.

The use of the escaped brackets is illustrated in the example below.

```
Divideby(e1, e2)           -> [ : { if (eq_expr(e2, Zero()) } ${
                                          e1 "/ /* <-- division by zero --> */" e2
                                    $} { else } ${
                                          e1 "/" e2
                                    $}
                              ];
```

The unparse functions have an additional argument of type `uview`. Unparsing rules can be made specific for a set of unparse views by naming these views between the square open bracket and the colon of an unparsing rule. For example:

```
Plus(e1 e2)          -> [ view1 view2: e1 "+" e2 ];
```

An omitted view list defaults to all unparse views. This is equivalent to a view list that only contains the view name base_uview. The type uview is an enumeration type of all view names occurring in the unparsing rules. It always contains the view name base_uview.

Views can be declared as in the following example:

```
%uview view1 view2; /* the '%' is part of the keyword */
```

Unparse views should be declared. If the termprocessor input contains one or more unparse view declarations it will report errors for all unparse views that are used and not declared. The use of view declarations may help to find typing errors in view names – in term processor input that does not contain view declarations, the misspelling of a view name may just introduce a new view (with the misspelled name).

If several unparse rules share the same left-hand side, they may be combined by grouping the right-hand sides (with seperating commas). If several unparse rules share the same right-hand side, they may be combined by grouping the left-hand sides (with seperating commas). The general form of an unparse rule is shown below:

```
pattern1, pattern2, ... -> [ v1 v2 ... : ... ], ..., [ vn ... : ... ] ;
```

The unparse_*phylum* functions that are generated have three arguments: the term to be unparsed, a (void) function, the *printer*, to be applied to each string (including string denotations of the predefined phyla) and an argument of type uview, which is passed to the printer function. The C code in the unparsing rules can refer to the latter two arguments directly by the names kc_printer and kc_current_view respectively. The user provides the printer function. The simplest example of such a function and its use is as follows.

```
void printer(char *s, uview v) {
    printf("%s", s);
}

{ /* example of usage */
    unparse_exprlist(expression, printer, base_uview);
}
```

There are two features that are notably missing from unparsing rules. A feature for handling list element separators is not necessary, as is shown above. A feature for specifying indentation in the output can be emulated in the printer function. How to do that is explained in Section 4.2.

1.8 Including Other Definitions

As will be presented later, *Kimwitu* generates a number of .h files, and a number of .c files that include those .h files. It may be necessary to include in those files additional C code or definitions, for instance because a type is defined that is used in the attribute declarations, or a function from a system library, such as math.h, is used in C code. The mechanism for this is similar to that of yacc, and the syntax for it is illustrated in the following example.

```
%{ /* these brackets should be at the beginning of a line */
#include <math.h>
%}
```

Inclusions like these can appear anywhere between declarations. They go to the beginning of the .c file that is being generated from the .k file in which this construct appears.

In case it is necessary to make an inclusion in *another* file, the distinguished symbol of that file can be used to redirect the inclusion, as in the following example. To make the same inclusion in several files, the redirection symbols of those files, separated by whitespace, can be given on the same line as the %{.

```
%{ KC_TYPES_HEADER
    /* Include this in k.h, and thus in every generated file. */
%}
```

For further information consult Section 2.4.

2 Output

Kimwitu generates a number of C files. They contain data types and functions on those data types. In this section we present the details of the term processor output.

2.1 Generated 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 for the example in Section 1.1 and Section 1.2 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; /* note that 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 term processor generates a C type definition boolean with constants True and False that have the usual C interpretation (namely int, 1, and 0).

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

2.2 Generated Functions

In this section we describe the various functions that are generated. Every function can be used like any other C function. In the names, the following 'meta' notation is used:

- *phylum* is a phylum name (user-defined or predefined), in the list functions *phylum* corresponds to the list elements.
- *list* is the name of a list phylum.
- *Operator* is an operator name.
- *result function*(); is the definition of a function found in the term processor input.

```
phylum Operator( phylum_sub1 sub1, ..., phylum_subn subn );
```

For each operator a C function is declared, with the same name. This function implements the storage option and returns (a pointer to) the term constructed from its arguments.

```
void *emalloc( kc_size_t n );
void *ecalloc( kc_size_t n, kc_size_t m );
void *erealloc( void *r, kc_size_t s );
```

The emalloc and friends functions are used for allocating memory, they call the system memory allocation routines and check the return status. The emalloc and friends functions' interfaces use the void* type[3], and hide the fact that some malloc libraries use the char* type from the user.

[3]actually, for ease of portability to systems that don't have the void* type it uses kc_voidptr_t, which is typedef'ed to void*.

The uniqmalloc and friends routines are used to implement the hashtables' storage. They claim large chunks from the system and pass out small blocks. These blocks cannot be freed by themselves, it is only possible to free a complete collection of chunks as a whole. Each collection of chunks can be identified by its kc_memory_info_t*.

```
kc_memory_info_t *uniqmallocinit( kc_size_t n );
```

Initialize the allocation information of a new set-of-chunks and return a pointer to it.

```
void *uniqmalloc( kc_size_t n, kc_memory_info_t *mi );
```

Returns a pointer to a block of size n from the chunks administrated in mi.

```
boolean isinuniqmalloccedblock( void *p, kc_memory_info_t *mi );
```

Report whether or not p was allocated via mi.

```
void uniqfreeall( kc_memory_info_t *mi );
```

Free all chunks allocated (managed) via mi .

```
void uniqfreeelement( void *, kc_memory_info_t *mi );
```

Free one block allocated by uniqmalloc. However, its current implementation is a no-op.

In the generated code these allocation routines are used via the following macros: MALLOC and friends are used for all allocation, except for the allocation of memory by the *Operator* functions. UNIQMALLOC2 is used to allocate memory for the phyla that have the 'uniq' storage option. NONUNIQMALLOC is used for the remaining phyla. If you want different ones, redefine MALLOC and friends, in your included code. For example, the default definition of MALLOC and NONUNIQMALLOC is emalloc, the default definition of UNIQMALLOC2 is uniqmalloc.

kc_size_t is by default typedef'ed to unsigned.

The generated code contains the definition of the enumerated type kc_storageclass_t , which contains all storage classes, including the predefined storage class uniq.

```
kc_hashtable_t kc_ht_assign( kc_hashtable_t ht, kc_storageclass_t sc );
```

Makes ht the hashtable for storage class sc, and return the previously assigned hashtable.

```
kc_hashtable_t kc_ht_assigned( kc_storageclass_t sc );
```

Return the hashtable assigned to storage class sc.

```
kc_hashtable_t kc_ht_create_simple( kc_size_t n );
```

Creates a hashtable of size n, using the kc_ht_create_bucketmanagement routine with default allocation routines (uniqmalloc and friends). For the bucket_alloc routines ecalloc and friends are used.

```
kc_hashtable_t kc_ht_create(
                    kc_size_t n,
                    void *(*uniq_malloc_init)(),
                    void *(*uniq_malloc)(kc_size_t n, void *mi),
                    void (*uniq_free_element)(void *p, void *mi),
                    void (*uniq_free_all)(void *p, void *mi),
                    boolean (*in_block)(void *p, void *mi) );
```

Creates a hashtable of size n with the given allocation routines. A 0 argument can be given for uniq_free_all, uniq_free_element and/or for in_block, if those routines are not implemented. This routine uses the kc_ht_create_bucketmanagement routine with default bucket allocation routines (ecalloc and friends).

```
kc_hashtable_t kc_ht_create_bucketmanagement(
                    kc_size_t n,
                    void *(*uniq_malloc_init)(),
                    void *(*uniq_malloc)(kc_size_t n, void *mi),
                    void (*uniq_free_element)(void *p, void *mi),
                    void (*uniq_free_all)(void *p, void *mi),
                    boolean (*in_block)(void *p, void *mi),
                    void (*uniq_info)(void *mi),
                    void *(*bucket_alloc_init)(),
                    void *(*bucket_calloc)(kc_size_t n, kc_size_t s, void *bi),
                    void *(*bucket_realloc)(void *b, kc_size_t old, kc_size_t new, void *bi),
                    void (*bucket_free)(void *b, kc_size_t s, void *bi),
                    void (*bucket_free_all)(void *bi),
                    int bucket_increment,
                    void (*bucket_info)(void *mi) );
```

Creates a hashtable of size n with the given allocation routines. A 0 argument can be given for uniq_free_all, uniq_free_element and/or for in_block, if those routines are not implemented, as well as for the info routines, and either the bucket_free or the bucket_free_all routines.

```
void kc_ht_clear( kc_hashtable_t ht);
```

Clears hashtable ht, ie. frees its elements (either using uniq_free_elements or, preferably, uniq_free_all), and its buckets. The result of this routine should be equivalent with calling kc_ht_delete followed by kc_ht_create.

```
void kc_ht_delete( kc_hashtable_t ht);
```

First kc_ht_clears the hashtable, and then frees the hashtable itself.

```
void kc_ht_reuse( kc_hashtable_t ht);
```

Prepares hashtable ht for reuse, ie. frees its elements, but not its buckets.

```
casestring mkcasestring( char *s );
nocasestring mknocasestring( char *s );
```

The basic phylum constructors. They convert a value of a C string into the corresponding phylum value. From a casestring or nocasestring value the string can be referenced as cs->name. In the case of case insensitive strings, this field will contain the string as it is capitalized at its first occurrence.

```
result function( args );
```

The template of user-provided functions.

```
boolean eq_phylum( phylum p1, phylum p2 );
```

The function that tests for structural equality of terms. Attribute values do not influence the comparison. If the phylum is uniquely represented, the test always takes constant time. Note that the comparison of a casestring with a nocasestring is not type correct.

```
void free_phylum( phylum p, boolean recursive);
```

The node p is zeroed and freed. If recursive is True, the subterms are freed as well. The body of a free function for a phylum with storage class uniq is empty.

```
void freelist_list( list p );
```

Free the list, but not the elements of the list. In other words, free the spine.

```
list concat_list( list l1, list l2);
```

Produce a new list that is the concatenation of the two arguments. If the second argument is the empty list, and the list phylum is not uniquely represented, this is equivalent to a copy of the list nodes (the 'spine'). The list elements are never copied.

```
phylum copy_phylum( phylum p, boolean copy_attributes );
```

Return a copy of p. If copy_attributes is False attribute values are initialised as defined in the definition of *phylum*. If copy_attributes is True the values of the attributes are copied. Please note that this does not imply that the attributes themselves are copied. This function is ineffective on uniquely represented phyla.

```
list reverse_list( list l);
```

Produce a list that is the reverse of the argument list.

```
int length_list( list l);
```

Yield the number of elements in a list, a nil list has zero elements.

```
phylum last_list( list l);
```

Yield the last element of a list, or an error if there isn't one.

```
list map_list( list l, phylum(*f)(phylum));
```

Yield the list constructed from the element-wise application of f. It 'lifts' f to a function on lists.

```
list filter_list( list l, boolean(*f)(phylum));
```

Yield the list of elements of which f is True.

```
void fprint_phylum( FILE *f, phylum p );
void print_phylum( phylum p );
```

Print the argument term on a given file (f = 0 means standard output – print_*phylum*(*p*) is identical to fprint_*phylum*(0, *p*)) in a canonical format. In this format each operator appears on a separate line. It is mainly intended for debugging.

```
void fprintdot_phylum(
                FILE *f,
                phylum p,
                char *root_label_prefix,
                char *edge_label_prefix,
                char *edge_attributes,
                boolean print_node_labels,
                boolean use_context_when_sharing_leaves,
                boolean print_prologue_and_epilogue );
void fprintdotprologue( FILE *f );
void fprintdotepilogue( FILE *f );
```

Print the argument term p, with its attributes, on a given file (f = 0 means standard output) in *dot* input format. *Dot* is a program that draws directed acyclic graphs in various output formats, among which are postscript and gif. It is available as part of the *GraphViz* graph visualisation package[EKN]. If print_node_labels is True, the nodes are labeled with the names of the operators, the types of the subtrees, and the attribute names and types (or values, for boolean attributes), and the root node is labeled with root_label_prefix followed by *phylum*. The edges are numbered; the edge numbers show a depth-first left-to-right treewalk. The numbers are prefixed with the contents of edge_label_prefix, which can make it easier to see to which tree an edge belongs when several trees are combined in the same picture, as discussed below. Attributes of the edges (colors, fonts, etc.) can be given with edge_attributes. Shared non-leaf tree nodes appear as such in the drawing. For leaf nodes (like integers, (no)casestrings) the amount of sharing can be influenced with the use_context_when_sharing_leaves argument: if it is False leaf nodes with the same value will be shared (which resembles the internal program structure, but may give pictures that are a bit confusing, especially if no non-leaf nodes are shared), if it is True then the ancestor of the leaf node is taken into account for the sharing, i.e. leaf nodes with the same value but different ancestors will not be shared. Finally, it is possible to combine several trees in one drawing. To do so, start by invoking fprintdotprologue. Next, invoke fprintdot_*phylum* for each tree, with argument print_prologue_and_epilogue set to False. You likely want to use a different edge_label_prefix for each tree. Finally, invoke fprintdotepilogue. If only one tree is to be drawn, then it is sufficient to invoke fprintdot_*phylum* with print_prologue_and_epilogue set to True (and fprintdotprologue and fprintdotepilogue need not be invoked 'by hand').

```
int kc_set_fprintdot_hashtablesize( int s );
```

Set the size of the hashtable that the fprintdot_*phylum* routines use to detect node-sharing. Return the previous value.

```
phylum rewrite_phylum( phylum p, rview v );
```

Yield the normal form of the first argument with respect to the rewrite system and view.

```
void unparse_phylum( phylum p, void(*kc_printer)(char *s, uview v), uview kc_current_view );
```

Unparses the argument p and its descendants, with view and printer as given.

```
char *CSGIOread_phylum( FILE *f, phylum *ptr );
char *CSGIOwrite_phylum( FILE *f, phylum p );
```

To read and write a term from a structure file (see Section 3.5). FILE is the standard file i/o type defined in <stdio.h>. The functions return a null pointer if all went well. If not, they return a pointer to a string containing an error message.

`int kc_set_csgio_hashtablesize( int s );`

Set the size of the hashtable that the CSGIOwrite_*phylum* routines use to detect node-sharing. Return the previous value.

`boolean kc_set_csgio_sharing( boolean b );`

Disable (b = False) or re-enable the use of node-sharing while *writing* structure files. This does not affect the ability to *read* structure files that contain node-sharing! By default (if this routine is not called) node-sharing in structure files is enabled. Return the previous value.

`void kc_print_operator_statistics( FILE *f );`

Print for each operator its size in bytes, the number of create calls, the number of actually created elements (usually much lower if its phylum has the uniq storage option), the number of free calls (recursive and non-recursive), the number of freed elements, the number of remaining nodes and the amount of space that these take, in bytes. The routine is equivalent to a no-op, unless the preprocessor symbol KC_STATISTICS is defined.

`void kc_print_hash_statistics( FILE *f );`

Print the number of hashtable buckets that contain zero, one, two, three, four, five, six, seven, eight or more elements.

`void kc_print_hashtable_memory_statistics( FILE *f, kc_hashtable_t h );`

Print information about the memory management of h. (This will only be possible if node- and bucket memory-management routines have been specified.)

2.3 Predefined Phyla and Operators

The following phyla and operators are predefined. They cannot be redefined by the user's input. These predefined phyla have the property of unique representation. The predefined operators are only used in conjunction with the Synthesizer Generator, see Section 3.5.

Phyla	Operators
int	_Int *mapped directly to C*
casestring	_Str
nocasestring	NoCaseStr

The operators mentioned here should not be used. Instead see the description of the functions mkcasestring, and mknocasestring in Section 2.2, which create values of these phyla.

2.4 File Names, Preprocessor Symbols and Redirection

The system generates at least 5 .c files and 5 .h files from its input. In particular, for each input *file*.k, a *file*.c and a *file*.h is generated that contain the functions defined in that input file. If there is no file argument, *Kimwitu* reads from standard input and *file* then takes the value stdin. The names of the generated files, and their include structure are given in the following table. Each .c file also defines a distinguished symbol.

File	Includes	Distinguished Symbol
k.h		KC_TYPES_HEADER
k.c	k.h	KC_TYPES
file.h		KC_FUNCTIONS_*file*_HEADER
file.c	k.h *file*.h	KC_FUNCTIONS_*file*
rk.c	k.h rk.h	KC_REWRITE
unpk.c	k.h unpk.h	KC_UNPARSE
csgiok.c	k.h csgiok.h	KC_CSGIO

The distinguished symbols can also be used as redirection symbols, as in:

```
%{ KC_REWRITE
#include "my_fns.h"
%}
```

Additionally one can use the shorthands from the following table.

Redirection Symbol	File
HEADER	*file*.h
CODE	*file*.c
*/*empty*/*	*file*.c

The use of distinguished symbols is further illustrated in Section 1.8.

2.5 Overview of Generated Names

This section contains an limited overview of the generated names. In the names, the ‘meta’ notation of Section 2.2 is used, with the following extension:

- *file* is the basename of the input file that corresponds to the generated thing (e.g. function).

As is indicated in the table, some of the macros can be redefined, to cater for specific applications.

Name	File	Type	See also Section
KC_TYPES_HEADER	k.h	macro	2.4
KC_TYPES	k.c	macro	2.4
LARGEPRIME	k.c	redefinable macro	1.3
UNIQMALLOC2	k.c	redefinable macro	2.2
NONUNIQMALLOC	k.c	redefinable macro	2.2
MALLOC	k.c, csgiok.c	redefinable macro	2.2
CALLOC	k.c, csgiok.c	redefinable macro	2.2
REALLOC	k.c, csgiok.c	redefinable macro	2.2
UNIQFREE	k.c	redefinable macro	2.2
NONUNIQFREE	k.c	redefinable macro	2.2
FREE	k.c, csgiok.c	redefinable macro	2.2
uniqmalloc	k.[ch]	void * function	2.2
emalloc	k.[ch]	void * function	2.2
ecalloc	k.[ch]	void * function	2.2
erealloc	k.[ch]	void * function	2.2
kc_ht_create_bucketmanagement	k.[ch]	kc_hashtable_t function	2.2
kc_ht_create	k.[ch]	kc_hashtable_t function	2.2
kc_ht_create_simple	k.[ch]	kc_hashtable_t function	2.2
kc_ht_assign	k.[ch]	kc_hashtable_t function	2.2
kc_ht_assigned	k.[ch]	kc_hashtable_t function	2.2
kc_ht_clear	k.[ch]	void function	2.2
kc_ht_delete	k.[ch]	void function	2.2
kc_ht_reuse	k.[ch]	void function	2.2
mkcasestring	k.[ch]	casestring function	2.2
mknocasestring	k.[ch]	nocasestring function	2.2
Operator	k.[ch]	*phylum* function	2.2
eq_*phylum*	k.[ch]	boolean function	2.2
fprint_*phylum*	k.[ch]	void function	2.2
print_*phylum*	k.[ch]	void function	2.2
fprintdot_*phylum*	k.[ch]	void function	2.2
fprintdotprologue	k.[ch]	void function	2.2
fprintdotepilogue	k.[ch]	void function	2.2
kc_set_fprintdot_hashtablesize	k.[ch]	int function	2.2
free_*phylum*	k.[ch]	void function	2.2
copy_*phylum*	k.[ch]	*phylum* function	2.2
concat_*list*	k.[ch]	*list* function	2.2
reverse_*list*	k.[ch]	*list* function	2.2
length_*list*	k.[ch]	int function	2.2
last_*list*	k.[ch]	*phylum* function	2.2
map_*list*	k.[ch]	*list* function	2.2
filter_*list*	k.[ch]	*list* function	2.2
phylum	k.h	datatype	2.1
Cons*list*	k.h	*Operator*	1.1
Nil*list*	k.h	*Operator*	1.1
sel_*Operator*	k.h	enumeration constant	2.1
YYSTYPE	k.h	datatype	2.1, 3.4
yt_*phylum*	k.h	union selector for Yacc	2.1, 3.4
kc_size_t	k.h	datatype	2.2
KC_FUNCTIONS_*file*	*file*.c	macro	2.4
function	*file*.[ch]	*result* function	

Name	File	Type	See also Section
KC_REWRITE	rk.c	macro	2.4
rewrite_*phylum*	rk.[ch]	*phylum* function	2.2
rview	rk.h	datatype	2.1
KC_UNPARSE	unpk.c	macro	2.4
unparse_*phylum*	unpk.[ch]	void function	2.2, 1.7
uview	unpk.h	datatype	2.1
KC_CSGIO	csgiok.c	macro	2.4
CSGIOread_*phylum*	csgiok.[ch]	char* function	2.2, 3.5
CSGIOwrite_*phylum*	csgiok.[ch]	char* function	2.2, 3.5
kc_set_csgio_hashtablesize	csgiok.[ch]	int function	2.2
kc_set_csgio_sharing	csgiok.[ch]	boolean function	2.2

2.6 Debugging Support

The generated code is organised in such a way that a symbolic debugger will be a useful tool for exploring data structures. Furthermore, by default the code is instrumented with assert statements that check e.g. for references through nil pointers. In the same way as the Unix assert, compiling with -DNDEBUG effectively deletes the asserts from the code. (There are other macros whose redefinition can influence the behaviour of assert.)

A typical run time error message is 'no default action defined'. This occurs when a with- or foreach-with-statement does not cover all cases, as explained in Section 1.5. One of the situations in which this can occur is when such a statement is applied to an argument of the wrong phylum. Running lint should have uncovered this, see Section 3.3.

3 Running It

In the previous sections we presented the input and output of the term processor. This section discusses some aspects of using the system in a software-development process. In particular, we discuss the relation with other tools (such as yacc) that are useful for this kind of software.

3.1 The Program and the Files

Kimwitu is invoked by, for example, the command kc file.k. It then generates a number of files. Multiple file.k arguments are permitted. To avoid superfluous recompilations when using make, kc will not overwrite a file with an identical file. See the discussion below (Section 3.2) for an example makefile. The program kc can give some error messages and warnings. Depending on the user-provided code, cc and lint can give additional error messages. In particular, lint will typecheck all expressions involving phyla and operators.

3.2 A Makefile

For the sake of convenience, we give here a typical makefile for use with the term processor, yacc and lex. This example illustrates a naming convention: the input files are called file1.k and file2.k, all user-provided C code is in the file examplemain.c, the yacc input is in exampley.y, and the lex input is in examplel.l. Note that the makefile makes extensive use of make's defaults, and that it attempts to avoid superfluous recompilations.

```
# /* Makefile for the term processor */
# /* 2 input .k-files plus yacc and lex usage. */
IT = example
KFILES = file1.k file2.k
YOURFILES = ${KFILES} ${IT}y.y ${IT}l.l ${IT}main.c
ALLOBJS = k.o rk.o csgiok.o unpk.o\
        ${KFILES:k=o} ${IT}y.o ${IT}l.o ${IT}main.o
GENERATED_BY_KC = k.c rk.c csgiok.c unpk.c ${KFILES:k=c}\
        k.h rk.h csgiok.h unpk.h ${KFILES:k=h}
YFLAGS = -d

${IT}:                          ${ALLOBJS}
        ${CC} ${CFLAGS} ${ALLOBJS} -ll -o $@

${GENERATED_BY_KC}:             kctimestamp

kctimestamp:                    ${KFILES}
        kc ${KFILES}; touch kctimestamp

${ALLOBJS}:                     k.h
${IT}main.o ${IT}l.o:           x.tab.h
${IT}main.o ${KFILES:k=o}:      ${KFILES:k=h}
${IT}main.o rk.o:               rk.h
${IT}main.o csgiok.o:           csgiok.h
${IT}main.o unpk.o:             unpk.h

# /* making copies to prevent unnecessary recompilation after yacc run */
x.tab.h:                        y.tab.h
        -cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h

# /* if you clean up, don't forget to remove the file kctimestamp */
```

This makefile is rather complicated, for the following reason. The target (${IT}) depends on a number of object files, which depend on a number of .c and .h files, some of which depend on the .k files. However, if example.k is changed the term processor does not always change all .c and .h files, and we want to avoid recompilations of unchanged files. For this reason, an intermediate target kctimestamp is introduced that 'remembers' when kc was last executed successfully.

The last rule (for x.tab.h) helps to avoid recompilations when yacc overwrites the file y.tab.h but doesn't change it.

There is still a problem with this makefile. It assumes that if kctimestamp is up to date the generated files are also accurate. The user can mess this up.

How to adapt this to your situation? If you have a different number of .k files, you will have to adapt the definition of the KFILES macro. If you do not use yacc you can remove the lines containing x.tab.h (2 lines) and the filenames ${IT}y.[yo]. If you do not use lex you can remove the filenames ${IT}l.[lo] and the loader option -ll. In either case it is sufficient to remove the object file names from the definition of the macro ALLOBJS. This is also true for other files that need not be included in the final program. For example, if no rewriting functions are used you can delete rk.o from ALLOBJS and it will not be compiled.

3.3 Using lint

Some static errors can be discovered by running lint on the C files, whether they are written by the user or generated by *Kimwitu*. Most notably this will catch argument type mismatches and such. The most convenient way is to use the option -u to suppress messages on functions not used.

3.4 Interfacing with Yacc and Lex

The first piece of advice is: try to avoid it, because the SG (SSL) structure file format is much more convenient. Notwithstanding that, it is fairly simple, and the term processor handles a lot of the machinery involved, such as defining a data type for the yacc stack. The two important considerations in using yacc are the difference between abstract syntax and concrete syntax, and handling terminal symbols with values, such as identifiers. Abstract syntax should be the input of the term processor and concrete syntax goes into yacc. First a simple abstract syntax.

```
/* Abstract syntax */
funnytree:      Str(casestring)
|               Cons(funnytree funnytree)
;
```

In the yacc input file, concrete syntax is translated into abstract syntax using the *operator* functions. Note also the type annotation of concrete symbols, as in %token <yt_casestring> ID. This type is a union selector of the generated YYSTYPE.

```
/* Concrete syntax */
%{
#include "k.h"

funnytree thetree;
%}

%token <yt_casestring> ID

%type <yt_funnytree> tree
%%

theroot: tree                  { thetree = $1;};

tree:    ID                    { $$ = Str($1);}
|        '(' tree tree ')'     { $$ = Cons($2, $3);}
;
```

The typical way of using lex is to have the lexical analyser generate tokens for the parser and build values of the phylum casestring. The following is the complete lex input.

```
/* Lexemes */
%{
#include "k.h"
#include "y.tab.h"
%}
%%
[a-zA-Z0-9]+    { yylval.yt_casestring = mkcasestring(yytext); return ID;}
[\t\n ]         { ; }  /* skip the white space */
```

```
.                { return yytext[0]; }
```

3.5 Interfacing with Structure Files and the Synthesizer Generator

The functions CSGIOread_*phylum* and CSGIOwrite_*phylum* read and write structure files that are fully compatible with the structure files produced by Synthesizer Generator editors[4]. The correspondence between the primitive phyla is indicated in the following table.

SSL primitive phylum	Kimwitu primitive phylum
INT	int
STR	casestring

To convert an SSL abstract syntax to a Kimwitu abstract syntax, one has to replace the occurrences of primitive phyla. It is convenient, but not essential, to also change SSL list phyla into Kimwitu list phyla, which can involve a renaming of SSL operators.

Since structure files encode a term directly on a file, a concrete syntax is not necessary. They have a number of properties that make them particularly suitable for interfacing separate programs, generated by either the SG or Kimwitu. These programs do not have to be generated from exactly the same term description in order to be able to communicate through structure files. In the following we call the structure file format (format for short) *insensitive* for such differences in the term description if this is the case. The format is insensitive for attributes because attribute values do not appear in the file. The format is insensitive for renaming of phyla, but not for renaming of operators. It is also sensitive for changes in the definition of operators such as adding or deleting component phyla. The format is insensitive for the definition or existence of operators that do not appear in the term to be written. This is essential because a tool would want to have 'local' phyla definitions. The format is also insensitive for storage options. Whether or not a term is shared internally is dependent only on the definition of the phyla at the reader program.

Below we give an example of usage of structure files. The abstract syntax is as follows:

```
/* Abstract syntax */
funnytree:      Str(casestring)
|               Cons(funnytree funnytree)
;
```

A structure file on this phylum could be written with the following code.

```
/* an example of structure file i/o */
#include <stdio.h>
#include "csgiok.h"
#include "k.h"

void main() {
    char *io;
    funnytree ft;

    ft = Str(mkcasestring("foo"));
    ft = Cons(ft, ft);
    ft = Cons(ft, ft);
    io = CSGIOwrite_funnytree(stdout, ft);
    if (io != (char *)0)          printf("%s\n", io);
```

[4] This is the, unattributed, ASCII SSL V3 format, which is the structure file format from SG version 2 onwards.

```
}
```

The reading of such a structure file then looks like this.

```
/* an example of structure file i/o */
#include <stdio.h>
#include "csgiok.h"
#include "k.h"

void main() {
    char *io;
    funnytree ft;

    io = CSGIOread_funnytree(stdin, &ft);
    if (io== (char *)0) print_funnytree(ft);
    else         printf("%s\n", io);
}
```

4 Cookbook

The term processor *Kimwitu* supports a number of styles of programming functions over trees. Such styles are sometimes called *paradigms*, and we can therefore say that the term processor supports *multi-paradigm programming*. In this section we try to substantiate this claim by presenting some examples. Each of these examples highlights a particular paradigm, some also show how paradigms can be mixed.

The most fundamental paradigm of course is that of trees, and in particular trees in which nodes of various types appear. This structure appears in a number of ways in computer science. *Parse tree* nodes correspond to the terminal and non-terminal symbols of a context-free language, or more precisely, production instances of those symbols. Normally an abstraction of parse trees is used, in which irrelevant terminal symbols and non-terminal symbols are eliminated. Such trees are usually called *abstract syntax trees*. In abstract data type theory, trees denote *expressions* and a normal form of an expression denotes its *value* in a certain sense. Trees can also represent a prescription of the computation of a value, or more general, a *program*.

The reason for mixing paradigms is that we want to exploit the strong points of each paradigm, while at the same time avoiding their weak points. Examples of such weak points are the following. Attribute grammars allow only attributes to be computed over trees, where the computation can not have a circular dependency. Functional programming languages do not allow side effects to be expressed. Abstract data type rewrite systems can only rewrite terms to their normal form. Conventional programming languages, such as C, usually force the programmer to be fairly aware of the representation of data types.

4.1 Structural Induction

Perhaps the most straightforward style of computing a value over a tree is by *structural induction*. The result of a tree is computed as a function of the results of its subtrees. The simplest example of such a function is equality of trees. Two trees are equal if the top nodes have the same structure and all the corresponding subtrees are equal. This function is so common that it is always generated by the term processor. Only one click less trivial is the function to compute the number of leaves of a tree. An example of that functions follows. The base case of nroftips simply returns 1, and the structural induction case just adds the results of the components.

```
/* A very simple tree structure */
funnytree:          Str(casestring)
|                   Cons(funnytree funnytree)
;

int nroftips(funnytree $f)
{
    Str:            { return 1; }
    Cons(l, r):     { return nroftips(l) + nroftips(r); }
}
```

In general one writes a function for every phylum that occurs in the tree.

4.2 Unparsing

Unparsing is in some ways a special case of structural induction. An example of an unparsing definition has been given in Section 1.7. In this subsection we show how a printer function can look that implements indentation. The idea is to define special control sequences to indicate the increment or decrement of indentation. We have chosen these control sequences in accordance with the SSL style. For example, the string

```
"start%t%nlevel1%nlevel1%b%nend"
```

should be printed as follows.

```
start
        level1
        level1
end
```

An example of a printer function to do that is the following:

```
#include <stdio.h>
#include "unpk.h"
static indent=0;

void printer(char *s, uview v) {
    char c; int j;

    while(c=*s++) {
        if (c!='%') putchar(c);
        else switch(c=*s++) {
        case 'b':       indent--; break;
        case 't':       indent++; break;
        case 'n':       putchar('\n');
                        for (j=indent; j>0; j--) putchar('\t');
                        break;
        case '\0':      return;
        default:        putchar(c);
```

```
}}}
```

The approach above has an advantage that is at the same time a disadvantage. The printer function also prints casestring and nocasestring terms, which has the advantage that it is possible to dynamically adapt the indentation, simply by assigning an unparsing control sequence to a casestring and nocasestring variable before it will be unparsed. The disadvantage is that one has to be careful that casestring and nocasestring terms do not contain unexpected unparsing control sequences - it may be necessary to quote the 'escape character' (the character % in the example above) in casestring and nocasestring terms, unless the 'escape character' does not normally appear in the casestring and nocasestring terms.

The following alternative does not have this problem - but neither has it the advantage of 'dynamic' control sequences. The idea is to use unparse views instead of control sequences - remember that the printer function was passed both a string and a view argument. Instead of a single string, a number of strings are unparsed, some of which have *:view* postfix - these strings are only meant to invoke the printer function with a 'control view'. To reduce the number of printer calls, the 'control views' can be combined with the 'normal' strings. This is left as an exercise for the reader. In the following example the printer function recognizes newline characters ('\n'):

```
"start" "":v_right "\nlevel1\nlevel1" "":v_left "\nend"
```

The printer function:

```
#include <stdio.h>
#include "unpk.h"

void printer(char *s, uview v) {
    char c; int j;
    static indent=0;        /* static here, or static at file level */

    switch(v) {
    case v_left:            indent--; break;
    case v_right:           indent++; break;
    default:                while(c=*s++) {
                                switch(c){
                                case '\n':  putchar(c);
                                            for (j=indent; j>0; j--) putchar('\t');
                                            break;
                                default:    putchar(c); break;
}}}}
```

If instead of newline characters a newline view would be used (eg. "":v_nl) the printer function could be simplified even more: the case '\n': code in the switch(c) can then be placed as case v_nl: in the switch(v), and the code for the default: case in that switch(v) can be replaced by a simple printf("%s",s); .

4.3 Attribute Grammars

Structural induction is a special case of attribute grammars, where there is only one pass, computing only synthesized attributes. This is not the place to give an overview of attribute grammars. Suffice it to say that 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 (synthesized attributes) or from the encompassing node (inherited attributes). An evaluation scheme walks a tree a number of times to compute all the attributes. Two schemes are demonstrated here. The example is from the original paper on attribute grammars[Knu68], and computes the value of a fractional binary number, e.g 1101.01. First 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 */
}
;
```

The first example of an evaluation scheme derives from the observation that each phylum occurrence can be viewed as a set of functions, each of which computes a synthesized attribute from the inherited attributes. So, there is a set of functions eval_*phylum_synthesized_attr*. Each rule for a synthesized attribute corresponds to a case in one of these functions, and each rule for an inherited attribute appears as a parameter of one of the calls to these functions. The function invocation structure is isomorphic to the attribute dependency graph. This scheme can also be characterised as a *demand-driven* scheme.

There are at least two problems with this approach. First, an inherited attribute of a phylum can be dependent on a synthesized attribute of that phylum. For example bitstring_scale depends on bitstring_length, and the computation of bitstring_length can therefore not have the 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.

```
/* 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; }
}

%{
#include <math.h>
%}

float eval_bit_value(bit $b, int scale) {
    One:                { return exp2((float)scale); }
    Zero:               { return 0.0; }
}
```

The second example of an evaluation scheme visits the tree a number of times and computes at each visit of a node all the attributes that can be computed. In the implementation there are procedures called pass*number_phylum*. 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 */
void pass1_number(number $n) {
    Nonfraction(b):     { pass1_bitstring(b); }
    Fraction(b1, b2):   { pass1_bitstring(b1);
                          pass1_bitstring(b2); }
}

void pass1_bitstring(bitstring $b) {
    Oneb:               { b->length=1;}
    Moreb(bs, *):       { pass1_bitstring(bs);
                          b->length=bs->length+1; }
}

/* pass1_bit omitted, it does nothing */

void pass2_number(number $n) {
    Nonfraction(b):     { b->scale=0;
                          pass2_bitstring(b);
                          n->value= b->value; }
    Fraction(b1, b2):   { b1->scale=0;
                          b2->scale= -b2->length;
                          pass2_bitstring(b1);
                          pass2_bitstring(b2);
                          n->value= b1->value+b2->value; }
}

void pass2_bitstring(bitstring $bs) {
    Oneb(b):            { b->scale= bs->scale;
                          pass2_bit(b);
                          bs->value= b->value; }
    Moreb(b1, b2):      { b2->scale= bs->scale;
                          b1->scale= bs->scale+1;
```

```
                    pass2_bitstring(b1);
                    pass2_bit(b2);
                    bs->value= b1->value + b2->value; }
}

void pass2_bit(bit $b) {
    One:            { b->value= exp2((float)b->scale); }
    Zero:           { b->value= 0.0; }
}
```

Here follows for the sake of completeness the main program to call the attribute evaluations.

```
/* the main program to call the evaluations */
void main() {
    number n;
    n = Fraction(Moreb(Moreb(Moreb(Oneb(One()), One()), Zero()), One()),
            Moreb(Oneb(Zero()), One())); /* 1101.01 */
    printf(" %f \n", eval_number_value(n));
    pass1_number(n); pass2_number(n);
    printf(" %f \n", n->value);

    n = Nonfraction(Moreb(Moreb(Moreb(Oneb(One()), One()), Zero()), One()));
    printf(" %f \n", eval_number_value(n)); /* 1101 */
    pass1_number(n); pass2_number(n);
    printf(" %f \n", n->value);
}
```

The examples illustrate that the term processor 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 evaluation order, e.g. in pass allocation, has to be determined manually, or by using some other tool. Conceivably a system can be made to generate term processor input from an attribute grammar.

4.4 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 program to call the function. We build a term corresponding to the application of ack, and then rewrite this into normal form.

```
#include "k.h"
#include "rk.h"
nat n2, n3;

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

4.5 Rewrite Systems and Functions

Combinators are a technique for implementing functional languages. This example, combinator reduction, illustrates the use of function calls in the right-hand side of an equation, as well as a few small yacc and lex techniques. The abstract syntax and the rewrite rules are as follows.

```
/* SKI combinator reduction */
%{ KC_REWRITE
int cplus();
%}

exp:    S()
|       K()
|       I()
|       ap(exp exp)
|       num(int)
|       plus()
;

ap(I(), x)                          -> x;
ap(ap(K(), x), y)                   -> x;
ap(ap(ap(S(), x), y), z)            -> ap(ap(x, z), ap(y, z));
ap(ap(plus(), num(x)), num(y))      -> num(cplus(x, y));
```

Note that the operator num refers to a built-in type int, which the term processor maps to C. In the last rewrite rule, a C function cplus is called on values of this type. This function is defined in the main program, as follows.

```
/* SKI expression reduction, main */
#include "k.h"
#include "rk.h"
extern exp x;

void main() {
    yyparse();
    print_exp(x);
    print_exp(rewrite_exp(x), base_rview);
}

int cplus(int i, int j) {
```

```
    return i+j;
}
```

In the yacc input some of the more mundane details of forcing the 'right' associativity are illustrated, watch the lines that start with %left.

```
/* yacc input SKI expression */
%{
#include "k.h"

exp x;

yyerror(char *s) {
    printf("%s\n", s); exit(1);
}
%}

%token <yt_int> NUM
/* the next 2 lines force left associativity */
%left '(' 'i' 's' 'k' '+' NUM
%left LEFT

%type <yt_exp> exp
%%

theroot:        exp                     { x = $1;};

exp:            '(' exp ')'             { $$ = $2;}
|               exp exp  %prec LEFT     { $$ = ap($1, $2);}
|               'i'                     { $$ = I();}
|               's'                     { $$ = S();}
|               'k'                     { $$ = K();}
|               NUM                     { $$ = num($1);}
|               '+'                     { $$ = plus();}
;
```

Finally, the minimal lex input, which shows how to read and convert an integer properly, and how to make the keywords case insensitive.

```
/* lex input for numbers */
%{
#include "k.h"
#include "y.tab.h"
#include <stdio.h>
#include <ctype.h>
%}
%%
[0-9]+          { sscanf(yytext, "%d", &yylval.yt_int); return NUM;}
[\t\n ]         { ; }  /* skip the white space */
.               { return (isupper(yytext[0])?tolower(yytext[0]):yytext[0]); }
```

The program that is generated from this, reads and reduces combinator expressions. For example, the input s k i 1, which is really the identity operation applied to 1, yields num(1). The input s(s(k+)i)(k1)8, which illustrates the increment operation, yields num(9).

4.6 Memo Functions

Memo functions 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).

Using the term processor, 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.

An example to illustrate memo functions is the Fibonacci function. This is a good example because the standard recursive definition recomputes the same result over and over again. For example, fib(5) uses fib(4) and fib(3), but for fib(4), fib(3) is also computed. It is also a silly example, because the best solution is a simple iteration. Furthermore we use abstract data type natural numbers, and the cost of the rewrite functions outweighs the costs of Fibonacci. The non-memo solution looks as follows, the phylum and rewrite definitions are from the previously discussed natural numbers example.

```
/* Fibonacci */
%{
#include "rk.h"
%}

nat fib(nat $n) {
    zero():        { return s(zero()); }
    s(zero()):     { return s(zero()); }
    s(s(x)):       { return rewrite_nat( plus(fib(x), fib(s(x)))); }
}
```

The memo version looks as follows, the natural number phylum is made unique and has an attribute fib to store the result.

```
/* Fibonacci with memo function */
nat{uniq}:     zero()
|              s(nat)
|              plus(nat nat)
{              nat fib = (nat)0; }
;

/* rewrite rules omitted */

%{
#include "rk.h"
%}

nat fibm(nat n) {
    nat result;
    if (n->fib != (nat)0) return n->fib;
    with(n){
        zero():          { result = s(zero()); }
```

```
        s(zero()):          { result = s(zero()); }
        s(s(x)):            { result = rewrite_nat( plus(fibm(x), fibm(s(x)))); }
    }
    n->fib = result;
    return result;
}
```

Note the initialisation of the attribute fib. We take the nil pointer to mean 'no value known yet'. In the second line of the function body the test is made for this, and in the second last line the result is stored. Measurements show that computing fib(15) (which is 987) takes 1973 calls on fib. In fibm the with-statement is entered only 16 times. However, as stated, using rewriting to compute addition makes this hardly noticeable on total run time. Both functions compute plus(377, 610) exactly once, and this takes most of the time.

4.7 Beyond Symbol Tables

Attributes of uniquely stored terms can be used to implement symbol tables, or more exactly, the contents of symbol tables. Looking up translates to newly creating a term (which is represented uniquely) and then inspecting its attributes. One can view this as making the look up function a memo function.

The nice thing is that entire terms can be used as the key in the 'symbol table'. This is useful for e.g. nested scopes. A key can then be a term composed of an identifier and a scope indication. This tuple should be unique.

As an example, we consider the detection of common subexpressions. Every subexpression is a key in the symbol table, and one pass over the expression computes the attribute ocs, which represents the number of occurrences of each subexpression.

```
/* A very simple tree structure */
funnytree {uniq}:           Str(casestring)
|                           Cons(funnytree funnytree)
{                           int ocs = 0;
                            funnytree next ;
                            { $0->next = alltrees;
                              alltrees = $0; }
}
;

void occurs(funnytree $f) {
    Str:                    { f->ocs++; }
    Cons(f1, f2):           { f->ocs++; occurs(f1); occurs(f2); }
}

%{ KC_TYPES_HEADER
funnytree alltrees;
%}

void main() {
    funnytree ft, it;

    alltrees = (funnytree)0;
    ft = Str(mkcasestring("foo"));
    ft = Cons(ft, ft);
    ft = Cons(ft, Str(mkcasestring("bar")));
```

```
    ft = Cons(ft, ft);
    it = alltrees;
    occurs(it);
    for(; it!= (funnytree)0; it= it->next) {
        if (it->ocs>1) {
            printf("occurs %d times:\n", it->ocs);
            print_funnytree(it);
}   }   }
```

The example also illustrates a technique through which all values of a particular phylum in the symbol table can be accessed. The idea is that they are strung together using an attribute (here next) and a global variable (here alltrees), which are manipulated in the initialisation part of a term. The other essential components of this technique are the initialisation of the global variable, and the inclusion of its definition in all the files through KC_TYPES_HEADER. This technique also works for non unique phyla. Unique phyla should not be used when some of the attributes depend on the context of their nodes.

5 Design Considerations for *Kimwitu*

In this section we discuss some of the considerations that have shaped the design of *Kimwitu*. This should provide the reader with some background information on the 'philosophy' of the system.

5.1 Why a Type per Phylum?

A fundamental choice is between a node type per phylum, as is done in *Kimwitu*, or a generic node type (the term structure to describe all term structures). This difference is comparable to that between compiled code and interpreted code. The first representation option is more efficient to use, it seems, but results in much more C-code (it might be different if assembly is generated directly). The second option is more suitable for language independent or polymorphic operations (as is used in the kernel of the Synthesizer Generator). An additional benefit of the first option is that it allows compile-time type checking of user-added code, as each phylum will correspond to a different type.

5.2 What is in a Name?

How are the various objects generated by the system named? The problem is that for each input name (e.g. of a phylum or operator) a number of identifiers in the output are generated (e.g. for an operator a name is generated to distinguish it from other operators, and a name for creating a term with that operator etc.). The basic idea is that related concepts have related names. In natural languages a comparable situation exists. For example, in English the words norm, normal, normalcy, normality, normalization, normalize, normalized, normalizes, normalizeth, normalizing, normally, and normalness, denote different but related forms of one word. These other words are called inflections, and are constructed, in most western languages, by changing suffixes. In other languages, e.g. Swahili, prefix changing is also used. In Swahili, the word *witu* means tree, its plural, meaning forest or jungle, is *mwitu* . The prefix *ki-* indicates a likeness of being, so that the name of our system reads as tree-s-ish. (This may not sound like English to you. Well, Swahili speakers don't count *Kimwitu* as a legal word either...)

The same scheme is employed in programming languages. For example, in Algol-68 and C, 'proc()' denotes the result of calling a parameterless function and 'proc' denotes the function itself. An example from the term processor: the function to rewrite a phylum foo is called rewrite_foo.

5.3 What is the Place in Software Engineering?

In the engineering of language based software, there are two extreme approaches. One is to just use a regular programming language (assembly language is not used very much any more these days), the other is to use a very high-level formalism, such as attribute grammars. *Kimwitu* was designed to fill a place in between these extremes, and to attempt to bridge the gap. The main theme is that by supporting *multi-paradigm programming* one can avoid being locked in by one formalism. In each formalism there are things that cannot be easily or efficiently expressed. Abstract data types can allow easy expression of certain functions, but the implementations are not necessarily efficient. In C every detail that can possibly influence performance is under the control of the programmer, but that does not necessarily allow easy expression of conceptually simple functions.

6 Acknowledgements

Helpful comments on previous versions of this manual where made by Bart Botma, Henk Eertink, Robert Elbrink, Rob Gansevles, Eric van Hengstum, Gerrit van der Hoeven, Anneke Kleppe, Matthijs Kuiper, Albert Nijmeijer, and Jan Tretmans.

A Syntax of the *Kimwitu* input

In this section we present the syntax of the *Kimwitu* input. The notation is based on yacc, with the following extensions. Single quotes are used for string denotations, ordinary parentheses () denote grouping, square brackets [] denote zero or one instance, curly brackets {} denote zero or more instances.

Comments are in C style, e.g. enclosed between /* and */, but can be nested.

Some of the rules and conditions are expressed in prose, rather than by formal means. An identifier (ID) is a sequence of letters digits and underscores, not starting with a digit.

```
specification:        { phylumdeclaration | includedeclaration | functiondeclaration |
                        rwdeclaration | unparsedeclaration |
                        rviewdeclaration | uviewdeclaration | storageclassdeclaration } ;

phylumdeclaration:    ID [storage_option] ':' [productionblock] [Ccode] ';' ;

storage_option:       '{' ['!'] ID '}' ;

productionblock:      'list' ID | alternative_list ;

alternative_list:     [alternative_list '|'] ID '(' arguments ')' ;

arguments:            {ID} ;

Ccode:                '{' [Attributes] [Cbody] '}' ;
```

Attributes: [Attributes] ID ID ['=' Cexpression] ';' ;

Cexpression: /* *arbitrary C expression without ';' and ',' with '$0'* */

includedeclaration: '%{' {ID} includes '%}' ; /* *the tokens are at the beginning of a line* */
/* *ID's in {ID} are at the same line as '%{'* */

includes: /* *arbitrary text* */ ;

rwdeclaration: outmostpatterns '->' rwclauses_or_term ';' ;

rwclauses_or_term: {rewriteclause} | outmostterm ;

rewriteclause: '<' rviewnames ':' outmostterm '>' ;

rviewnames: {ID} ;

patternchains: patternchain [',' patternchains] ;

patternchain: patternchainitem ['&' patternchain] ;

patternchainitem: outmostpattern
| '(' patternchains ')'
| '$' ID /* *this rule is to be used only in 'patternchain' in foreach_statement* */
;

outmostpatterns: outmostpattern [',' outmostpatterns] ;

outmostpattern: [ID '='] ID ['(' [patterns] ')'] | '*' | 'default' ;

patterns: pattern [',' patterns] ;

pattern: ID '=' pattern
| ID ['(' [patterns] ')']
| '"' /* *any string of characters* */ '"'
| /* *a number* */
| '*' | 'default'
;

outmostterm: ID '(' [terms] ')' ;

term: ID ['(' [terms] ')']
| '"' /* *any string of characters* */ '"'
| /* *a number* */
;

terms: term [',' terms] ;

functiondeclaration: decl_specifiers fn_declarator {declaration} MainCbody ;

decl_specifiers: [stor_class_specifier] [type_qualifier] ID ;

stor_class_specifier: 'auto' | 'register' | 'static' | 'extern' | 'typedef' ;

type_qualifier: 'const' | 'volatile' ;

fn_declarator: [pointer] direct_fn_declarator ;

```
direct_fn_declarator:        ID '(' [fnarguments] ')'
|                            ID '(' parameter_type_list ')'
;

fnarguments:                 [fnarguments ','] ['$'] ID ;

parameter_type_list:         parameter_list [',' '...'] ;

parameter_list:              [parameter_list ','] parameter_decl ;

parameter_decl:              decl_specifiers declarator
|                            decl_specifiers abstract_declarator
;

declarator:                  [pointer] direct_declarator ;

pointer:                     '*' {type_qualifier} [pointer] ;

direct_declarator:           ['$'] ID
|                            '(' pointer direct_declarator ')'
|                            direct_declarator '[' [Cexpression] ']'
|                            direct_declarator '(' param_type_list ')'
|                            direct_declarator '(' [fnarguments] ')'
;

abstract_declarator:         pointer
|                            [pointer] direct_abstract_declarator
;

direct_abstract_declarator:  '(' abstract_declarator ')'
|                            [direct_abstract_declarator] '[' [Cexpression] ']'
|                            [direct_abstract_declarator] '(' [parameter_type_list] ')'
;

declaration:                 decl_specifiers     [declarator_list] ;

declarator_list:             [declarator_list ','] declarator ;

MainCbody:                   Cbody
|                            '{' with_clause '}' /* a with_clause if a fn argument has a $ prefix */
;

Cbody:                       '{' Ctext '}' ;

Ctext:                       /* arbitrary C text, with $0 through $n,
                                with_statement and foreach_statement */ ;

foreach_statement:           'foreach' '(' patternchain ';' IDCexpressions ')' MainCbody
                             [ 'afterforeach' '(' patternchain ')' MainCbodyinC ]
;

IDCexpressions:              ID Cexpression [',' IDCexpressions] ;

with_statement:              'with' '(' Cexpressions ')' '{' with_clause '}' ;

Cexpressions:                Cexpression [',' Cexpressions] ;

with_clause:                 patternchains ':' Cbody [with_clause] ;
```

unparsedeclaration:	outmostpatterns '->' {unparseclause} ';' ;
unparseclause:	'[' uviewnames ':' {unparseitem} ']' ;
uviewnames:	{ID} ;
unparseitem:	'"' /* *any string of characters* */ '"' [':' ID]
|	['(' ID ')'] ID {'->' ID} [':' ID]
|	Cbody
|	'${' {unparseitem} '$}'
;	
rviewdeclaration:	'%rview' rviewnames ';' ;
uviewdeclaration:	'%uview' uviewnames ';' ;
storageclassdeclaration:	'%storageclass' {ID} ';' ;

B Structure File Encoding

This is a description of the ASCII SSL V3 format as used by Grammatech's Synthesizer Generator and the term processor Kimwitu (University of Twente). (Actually, both of these are generators of programs using this representation.)

Not all features are described, e.g. other atomic phyla are possible in Synthesizers, but not in combination with Kimwitu.

The structure file format is an ASCII encoding of a term. It is a prefix representation of this term. A file representation has two major components. The first component is a table of the operators that are used in the term, operators not appearing in the term do not have to be contained in this table. The operators in this table are numbered, starting from 0. The second component of the file is a representation of the term (or object as it is called). The begin markers of these components, $operators and $object, must be followed by one space at the end of the line.

Each line in the operator table describes the properties of one operator, and has 4 fields. The fields in the line are separated by one space. They are: the operator name, the number of its operands, the number of attributes (in term processor generated files this is always 0), and an indication of whether the operator belongs to an atomic phylum (1=yes, 0=no).

In the object section each line represents one node in the term. There are four types of these nodes: operator applications, string constants, integer constants, and pointers to shared nodes.

A line beginning with a number denotes an operator application, it is an index in the operator table. The rest of this line, if any, can be ignored (it contains an alternative unparsing indication for some tools). An operator of an atomic phylum is followed, on the next line, by a representation of a value.

A plus sign (+) denotes a string, and is followed by a number indicating the number of characters in the string, a separating space, and the string. In the string, all printable characters except the backslash (\) are represented as such. A backslash is doubled, and a non printable character is represented as a backslash followed by a hexadecimal representation of the ascii number, e.g. newline is \oa. An integer is represented as the string encoding of its decimal representation.

Pointers serve to share trees, and strings (but not integers). Such a pointer is encoded in a base-64 representation, in which the character `:` represents the 'digit' 0, and the character `y` represents the 'digit' 63. Intermediate 'digits' are represented by the intermediate ascii characters. For example, the string `;=` denotes the value 67. Conceptually at least, the result of each operator application is stored in one table, and each string is stored in a second table. The two numbers on the line below the begin marker `$object` give the size of the operator application resp. the string table. The pointer value is then a reverse index (counting from the end back) in the appropriate table of values. E.g. the last value of the appropriate kind in the file before the pointer has number 1.

The following two examples illustrates the structure file format. Both examples show the same term; the first example contains no sharing, the second everything possible is shared. Also, in the second example the operator table in the operators section has been sorted (decreasingly) on the number of operator applications. As a result, the references to this table should be smaller (lower numbers), taking less characters for their representation.

A#S#C#S#S#L#V#3 ← *magic word indicating file type*
$operators ← *begin marker operator table*
CR_Spec 12 0 0 ← *operator 0*
CR_Label 1 0 0
_Str 0 0 1 ← *operator of an atomic phylum*
Nilcr_comment_list 0 0 0
CR_Specification_id 1 0 0
CR_Identifier 2 0 0
NoCaseStr 0 0 1
CR_DefExtension 1 0 0
Nilcr_gate_identifier_list 0 0 0
Nilcr_identifier_declaration_list 0 0 0
CR_Noexit_part 0 0 0
Nilcr_data_type_definition_list 0 0 0
CR_Definition_block 3 0 0
CR_Stop_expression 2 0 0
Nilcr_annotation_list 0 0 0
Nilcr_process_definition_list 0 0 0
CR_Booleans_NotChecked 0 0 0
CR_IS8807 0 0 0
$object ← *begin marker object*
24 4 ← *number of operator applications; number of strings*
0 ← *operator application, index in table above*
1
2
+10 specname_1 ← *string representation*
3
4
5
6
+8 specname
7
6
+1 0
8
9
10
11
3
12
13
1
2
+6 stop_0
14
11
15
16
14
17

A#S#C#S#S#L#V#3 ← *magic word indicating file type*
$operators ← *begin marker operator table*
CR_Label 1 0 0
NoCaseStr 0 0 1 ← *operator of an atomic phylum*
_Str 0 0 1 ← *operator of an atomic phylum*
Nilcr_identifier_declaration_list 0 0 0
CR_Spec 12 0 0
CR_IS8807 0 0 0
Nilcr_comment_list 0 0 0
CR_Booleans_NotChecked 0 0 0
CR_DefExtension 1 0 0
CR_Specification_id 1 0 0
CR_Definition_block 3 0 0
Nilcr_process_definition_list 0 0 0
CR_Identifier 2 0 0
Nilcr_gate_identifier_list 0 0 0
Nilcr_annotation_list 0 0 0
CR_Noexit_part 0 0 0
Nilcr_data_type_definition_list 0 0 0
CR_Stop_expression 2 0 0
$object ← *begin marker object*
21 4 ← *number of operator applications; number of strings*
4 ← *operator application, index in table above*
0
2
+10 specname_1 ← *string representation*
6 ← *shared node referenced below as label_1*
9
12
1
+8 specname
8
1
+1 0
13
3
15
16 ← *shared node referenced below as label_2*
D ← *reference to label_1 above*
10
17
0
2
+6 stop_0
14 ← *shared node referenced below as label_3*
@ ← *reference to label_2 above*
11
7
= ← *reference to label_3 above*
5

C Compatibility with previous versions of Kimwitu

During the (ongoing) development of Kimwitu new ideas lead to changes in the software. Most of those changes are invisible to the Kimwitu user, some of them are not. This section lists a number of visible changes in Kimwitu.

C.1 Define-before-use constraints

In older versions of Kimwitu, both *phyla* and *operators* had to be defined before their use in a pattern such as appears in function definitions and rewrite rules. More recent versions of Kimwitu no longer impose this constraint (version V3_0 or newer).

C.2 Node-sharing in Structure Files

From version V3_4 on the CSGIOwrite routines use node-sharing features. Older kc-generated programs can not read files that use node-sharing; of course, new kc-generated programs can read files without it. For compatibility with older programs, the use of node-sharing for *writing* files can be turned off. The routine kc_set_csgio_sharing can be called with a boolean argument False to make the CSGIOwrite routines do *not* use the node-sharing. The routine returns the current csgio node-sharing setting.

In addition, if the preprocessor symbol KC_CSGIO_NO_SHARING is defined when the generated csgiok.c file is compiled, the CSGIOwrite routines do *not* use the node-sharing. Defining this macro does *not* turn off the ability to *read* files that contain node-sharing.

C.3 The UNIQMALLOC2 Macro

This macro is named UNIQMALLOC2 for backward compatibility with previous versions of kc that defined and used a macro UNIQMALLOC that referred to a *unary* function. UNIQMALLOC2 refers to a *binary* function.

C.4 The KIMW_ Redirection Symbols and Macros

In previous versions of Kimwitu the redirection symbols and macros had a KIMW_ prefix. Because these names did not fit in the Kimwitu naming scheme, the redirection symbols and macros now have a KC_ prefix. The 'old' redirection keywords are still recognized by Kimwitu, but a warning is given.

C.5 The view Enumerated Type

In previous versions of Kimwitu only unparse rules had *views*, which were collected in a enumerated type view. For backwards compatibility, view is now defined as a *typedef* for the enumerated type uview.

D Future

This section contains some of the ideas for future changes, enhancements of Kimwitu. The purpose of this section is twofold: it can be seen as a sort of 'advance warning' for potential future incompatible changes, and, because it contains material that is not completely mature, it can be seen as a 'request for feedback' from *You*, the Kimwitu users.

D.1 CSGIO Structure File IO Routines

Possible future change: changing the names of the structure file functions (because the uppercase CSGIO prefix doesn't really fit in the naming scheme) and supporting more formats.

D.2 Conditional Rewrite Rules

In future versions, it will be possible to use conditional rewrite rules. The exact Kimwitu input syntax has not yet been decided.

D.3 User Defined Atomic Phyla

In future versions of Kimwitu, it will be possibly to specify your own atomic phyla. The exact Kimwitu input syntax has not yet been decided.

D.4 Generation of C++ code

Future versions of Kimwitu will be able to generate C++ code, and accept a more C++ like syntax for function definitions. The move towards C++ will be gradually, with the following likely milestones: generation of C code that can be compiled as C++ code, generation of overloaded C++ functions for the standard functions, generation of CSGIO routines that support the C++ file io mechanisms,

D.5 Support for Sets, Queues, Stacks, Arrays, etc.

Once Kimwitu has the power of C++, it may be possible to support more 'basic' data types.

D.6 Polymorfism

Extend the Kimwitu input syntax with a mechanism to define some kind of template functions over lists.

D.7 Hash Management

In addition to the routines described in 2.2 the routines kc_ht_static, kc_ht_dynamic, kc_ht_inc_level, kc_ht_dec_level, and kc_ht_free_level have been added as an experiment. Right now it is unclear whether they are useful or not, or whether this sort of features should be offered in Kimwitu, or are better left to the individual programmer. A compromise might be to supply such routines as a kind of library in the form of a Kimwitu input (.k) file.

Controlling the lifetime of unique memory cells by the creation, freeing and (re-) assignment of hashtables has one disadvantage: the uniqueness of storage property is no longer guaranteed. The following alternative does not have this flaw, but may be slower in execution time. The idea is to treat the hashtable as a stack of life-times, called segments, or memory-levels. New memory cells will always be created in the memory segment at the top of the stack. A new segment can be pushed onto the stack, eg. to store intermediate results of a computation. When the computation is finished, the segment can be popped from the stack; its memory cells remain alive. When now the final result of the computation is copied, its memory cells in the popped segment are re-created in the now topmost segment. Finally, the popped segment can be deleted. For each hashtable such a stack of segments can be used, and pushing and popping the segments of one hashtable does not influence the segments of another one.

In addition to the segments on the stack, there is one 'heap' segment, that is meant for memory cells that never will be reclaimed. In Kimwitu, we call the heap segment 'static', and the stack segments 'dynamic'. Kimwitu offers two routines to switch between the use of the 'static' segment and the 'dynamic' segments. By default, all unique memory cells are created in the 'heap' segments of the hashtables, ie. by default the 'static' segments are used.

The routines that 'manage' the segments (pushing, popping, freeing, switching between 'static' (heap) and 'dynamic' (stack)), and the routines that manage the hashtables (creation, assignment, freeing, deletion, reuse) can be freely intermixed, to use the best of both approaches.

Below follows a description of the memory level routines.

```
void kc_ht_static( kc_storageclass_t sc );
```

Allocate everything using the 'static' scheme, until kc_ht_dynamic is called for this storage class. This is the default.

```
void kc_ht_dynamic( kc_storageclass_t sc );
```

Allocate everything using the 'dynamic' scheme, until kc_ht_dynamic is called for this storage class, or until the hashtable is changed (by assignment, deletion, clearing or reuse).

```
void kc_ht_inc_level( kc_storageclass_t sc );
```

Increment the memory level: everything allocated in hashtable ht using the 'dynamic' scheme will be freed during subsequent kc_ht_dec_level, kc_ht_free_level calls.

```
void kc_ht_dec_level( kc_storageclass_t sc );
```

Decrements the memory level: everything allocated using the 'dynamic' scheme in the previous, higher, memory level is still available, but can no longer be found using the hashtable, ie. copying a node that was created in the previous, higher, memory level will yield a new copy of that node (and the initialization code of its phylum will be executed).

```
void kc_ht_free_level( kc_storageclass_t sc );
```

Free all nodes/elements allocated during previous, higher, memory levels.

References

[EKN] John Ellson, Eleftherios Koutsofios, and Stephen North. graphviz – tools for viewing and interacting with graph diagrams.
URL: http://www.research.att.com/sw/tools/graphviz/. 3

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*. Springer-Verlag, Berlin, 1985. 1.6

[Knu68] Donald E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2:127–145, 1968. A correction appears in vol. 5 pp95-96. 4.3

[TR84] T. Teitelbaum and T. W. Reps. The synthesizer generator. *SIGPLAN*, 19(5):42–48, May 1984. 1

[TR89a] T. Teitelbaum and T. W. Reps. *The Synthesizer Generator - A System for Constructing Language-Based Editors*. Springer-Verlag, New York, 1989. 1

[TR89b] T. Teitelbaum and T. W. Reps. *The Synthesizer Generator Reference Manual: Third Edition*. Springer-Verlag, New York, 1989. 1

Index