Kimwitu++
A Term Processor

Toby Neumann, Michael Piefel

Kimwitu++

A Term Processor

Toby Neumann
Humboldt-University Berlin, Institute for Informatics

Michael Piefel
Humboldt-University Berlin, Institute for Informatics

Kimwitu++: A Term Processor

Toby Neumann, Michael Piefel

User's guide 1.0, published 2002

The program Kimwitu++ is based on Kimwitu (with its own web site (http://purl.oclc.org/net/kimwitu)), written by Axel Belinfante (belinfan@utwente.nl). Kimwitu is also free software, licensed under the GPL, since its 4.6.1 release.

This paper was written in XML using the DocBook DTD.

Output is produced with the typesetting system TEX using the KOMA Script package after conversion to LATEX with the XSLT program docbook2tex.

Contents

Examples

Part I

Introduction to Kimwitu++

This part explains what essentially Kimwitu++ is and what advantages it provides. It gives an overview of the functionality of Kimwitu++, which is demonstrated with example. Detailed discussion of concepts is left to the reference part later in this book.

1 What is Kimwitu++?

This chapter sketches out the field on which Kimwitu++ is usefully employed and explains important terms. It develops an example to outline the advantages of this tool compared to conventional techniques. The example will be used throughout the following chapters to introduce concept by concept. The complete code to make it work can be found in appendix A.

1.1 What for is Kimwitu++ used?

To illustratively explain what we can do with the help of Kimwitu++ we call upon an example. Let us imagine we want to write a computer game. Neat example. Respectable programmers as we are we calculate the overall costs of the new project in advance. It will take us, say, 30 days, on each we need to have a pizza at 7 € and 3 beer at 2 € each, but luckily meanwhile our grandma supports us with 100 €. We get the expression $30(8 + 3 \times 2) - 100$ and type it in postfix notation into our RPN-calculator. You know, one of the antiquated devices knowing nothing about precedence nor parentheses and expecting input in Reverse Polish Notation, where operands precede their operator.

So we type `30 8 3 2 * + * 100 -`. Er, we cannot. I remember. The calculator gave up the ghost last week. But don't despair. We quickly program a new one.

1.2 A Simple Example

What exactly should the new calculator be able to do? Anyhow, it better would accept also variables. Why? Just imagine a drastic shortage of pizza and beer what would urge us to setup a new expression again. We better use a flexible one which can be seen in example 1.1, in which x is the price of a beer and we know the price of a pizza always to be 4 times a beer.

We now list the requirements on the calculator in proper order.

1. We want it to analyse an input expression (term) of digits, arithmetical operators, and variables.

2. It should calculate the expression if possible or simplify it at least.

Example 1.1 Sample Expression

$$30(4 \times x + 3 \times x) - 100$$

(and resulting input in RPN: `30 4 x * 3 x * + * 100 -`)

3. It then should output a result.

We assume the existence of a code component (a scanner) providing us with syntactical tokens like operators, numbers and identifiers (variables). We then need a description of correct input, a grammar. Our example demands a valid arithmetical term only to consist of numbers, identifiers, and the four arithmetical operators. A formal notation as used by the parser generator Yacc (see 8.2) would look like in example 1.2.

Example 1.2 Yacc Grammar for Sample

```
aritherm:    simpleterm
           | aritherm aritherm '+'
           | aritherm aritherm '–'
           | aritherm aritherm '*'
           | aritherm aritherm '/';

simpleterm:  NUMBER
           | IDENT;
```

Such a grammar is made up of rules, each having two sides separated by a colon. These rules describe how the left-hand side called nonterminal can be composed of syntactical tokens called terminals, which are typed in upper case letters or as single quoted characters. Different alternatives are separated by bar. The right-hand side may also contain nonterminals which there can be regarded to be an application of their respective composition rule.

The first rule of this grammar defines an aritherm to be a simpleterm or a composition of two aritherms and an operator sign. These aritherms in turn may be composed according to the aritherm rule. The second rule describes what an aritherm looks like, if it is a simpleterm.

A common way to hold an input term internally is to keep it as a syntax tree, that is a hierarchical structure of nodes (upside down tree). A nonterminal can be regarded as a node type and every alternative of the assigned right-hand side as one kind of that type. Every actual node thus is a kind of a node type with a specific number of child nodes, which depends on the kind. For example '+' is a kind of aritherm and it has 2 child nodes, while NUMBER is a kind of simpleterm and it has none. Figure 1.1 shows a syntax tree for our sample input term. Since every node is a term itself such a tree is more generally called a term tree.

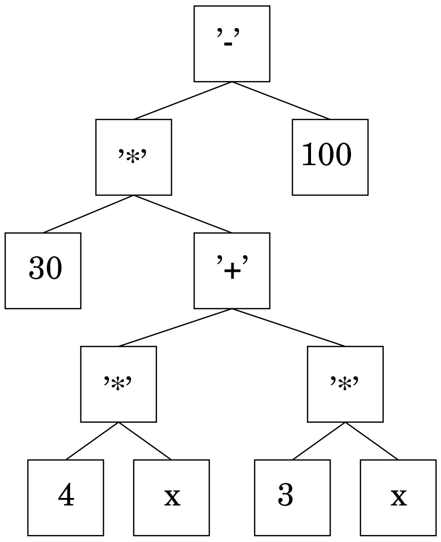

Figure 1.1: Syntax tree representing sample term

To summarise the task identified: we want to build a tree from the term which has been typed in, walk over the tree nodes, and perform appropriate actions like calculating, simplifying or printing some results.

1.3 Conventional Approach

In a programming language a node type is usually represented as a structured data type, that is, as a class in object oriented languages and as a sort of record in others. A kind, then, may be a class variant or a subclass, and a variant record respectively. The code fragment in example 1.3 illustrates a possible implementation in C++ (the kind as a subclass). The classes left out are to define similar.

From these classes we can instantiate C++ objects to represent our sample tree. We can navigate through it by accessing the child nodes of nodes. Not really yet, if you look closely at it. The child nodes are private members and thus they can not be accessed. We have to make them public or to add methods for their access. But what about the next step, simplifying parts of the tree? The subtree $4 \times x + 3 \times x$ could be transformed to, right, $(4 + 3) \times x$, by putting x outside the parentheses, as illustrated in figure 1.2. For C++ this may look like listed in example 1.4.

That seems quite complicated and it is even more so! Not only have we to cast for every child node access, to avoid memory leaks we had to free memory of unused nodes as old A and its child nodes. Furthermore the equality check between

Example 1.3 Node Types as Classes with C++

```
class Aritherm                      { /* ... */ };
class Simpleterm : Aritherm         { /* ... */ };

class Plus          : public Aritherm
{
  public :
    Plus( Aritherm *a, Aritherm *b ) : t1 ( a ), t2 ( b )
    { /* ... */ };
  private:
    Aritherm *t1, *t2;
};

class Number        : public Simpleterm
{
  public :
    Number( int a ) : n ( a )
    { /* ... */ };
  private:
    int n;
};
```

t1 and *t2* will merely compare pointer values, instead of checking whether the subtrees are structurally equal. Thus we additionally need to overload the equality operators. What a lot of trouble for such a simple example!

1.4 Kimwitu++ Approach

Kimwitu++'s name is formed after Swahili while the '++' reminds on C++.

witu	:	'tree'
m-	:	plural prefix
ki-	:	adjectival prefix: 'being like'
kimwitu	=	'tree-s-ish'

Thus the name indicates affiliation to trees and to C++. You guessed it before, didn't you? More strictly spoken Kimwitu++ is a tool which allows to describe in an easy way how to manipulate and to evaluate a given term tree. From these descriptions C++ code is generated and compiled to a program which processes terms. That is why Kimwitu++ itself is called a term processor.

The code for building a tree, we have to write ourselves, or we let preferably other tools generate it. We use Yacc to call term creation routines which are generated from a Kimwitu++ abstract grammar. This we have to specify first. It is similar to the Yacc grammar, but its right-hand side alternatives are operators

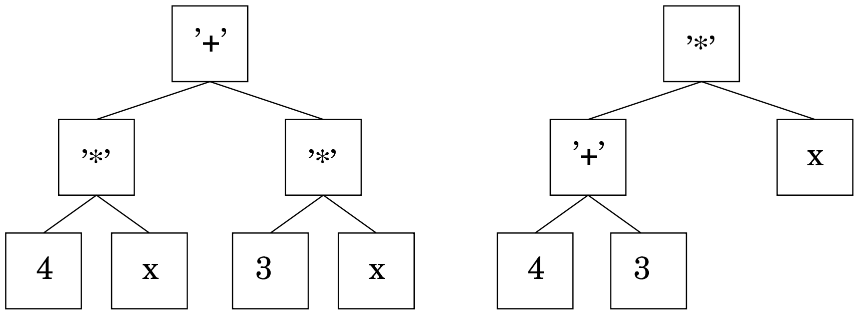

Figure 1.2: Simplification of a samples subtree

Example 1.4 Term Substitution with C++

```
if( dynamic_cast<Sum> ( A ) !=0 &&
    dynamic_cast<Mul> ( dynamic_cast<Sum>( A ) –> t1 ) !=0 &&
    dynamic_cast<Mul> ( dynamic_cast<Sum>( A ) –> t2 ) !=0 &&
    dynamic_cast<Mul> ( dynamic_cast<Sum>( A ) –> t1 ) –> t2 ==
    dynamic_cast<Mul> ( dynamic_cast<Sum>( A ) –> t2 ) –> t2 )
{

   A = new Mul ( new Sum (
    dynamic_cast<Mul> ( dynamic_cast<Sum>( A ) –> t1 ) –> t1,
    dynamic_cast<Mul> ( dynamic_cast<Sum>( A ) –> t2 ) –> t1 ),
    dynamic_cast<Mul> ( dynamic_cast<Sum>( A ) –> t1 ) –> t2 );
};
```

applied to nonterminals. An abstract grammar for our sample can be seen in example 1.5. The nonterminals integer and casestring are predefined in Kimwitu++.

Next we have to complete the Yacc grammar by adding semantic actions in braces to every alternative. These recursively create a term tree which has its root element assigned to the variable *root_term*. Example 1.6 shows this grammar, in which *$$* denotes the term under construction and *$1* and *$2* its first and its second subterm (child node).

That is it for building a tree. Everything else is left to the automatic code generation. Modifying or evaluating the tree needs its own rules (see chapter 4).

Example 1.5 Abstract Grammar for Sample

```
aritherm:     SimpleTerm ( simpleterm )
            | Plus       ( aritherm aritherm )
            | Minus      ( aritherm aritherm )
            | Mul        ( aritherm aritherm )
            | Div        ( aritherm aritherm );

simpleterm:   Number     ( integer )
            | Ident      ( casestring );
```

Example 1.6 Completed Yacc Grammar for Sample

```
aritherm:    simpleterm
               { root_term = $$ = SimpleTerm( $1 ); }
            | aritherm aritherm '+'
               { root_term = $$ = Plus( $1, $2 ); }
            | aritherm aritherm '–'
               { root_term = $$ = Minus( $1, $2 ); }
            | aritherm aritherm '*'
               { root_term = $$ = Mul( $1, $2 ); }
            | aritherm aritherm '/'
               { root_term = $$ = Div( $1, $2 ); };

simpleterm:   NUMBER
               { $$ = Number( $1 ); }
            | IDENT
               { $$ = Ident( $1 ); };
```

1.5 Summary

The language Kimwitu++ is an extension of C++ for handling of term trees. It allows the definition of term types, creation of terms, and provides mechanisms for transforming and traversing trees as well as saving and restoring them. Besides creating it in static code a tree can dynamically be obtained by interfacing with compiler generator C++ code (as from Yacc/Bison). The Kimwitu++ processor generates C++ code from the contents of Kimwitu++ input (`.k`-files). Compilation of that code yields the term processing program.

2 How to Define Term Types

This chapter describes possibilities to define term types, which make up the Kimwitu++ input. In Kimwitu++ they are called phyla (singular phylum), and that is what we will call them from now on. A phylum instance is called a term.

2.1 Definition

How are phyla defined? Example 1.5 shows that each phylum is defined as an enumeration of its kinds, each being an operator applied to a number of phyla, maybe zero. So the operator SimpleTerm takes one simpleterm phylum, Plus takes two aritherm phyla. There are several predefined phyla, of which integer and casestring already have been mentioned. The latter denotes a case sensitive character string. If a phylum is defined more than once, all occurrences contribute to the first one. For each phylum, a C++ class is generated.

2.2 Lists

We may want to define a phylum as a list of phyla. Imagine we wanted not only to type one expression into our calculator but several ones at once, separated in input by, say, a semicolon. The main phylum, representing whole the input, would be a list of aritherms. This is a right-recursive definition of a list which may be a nil (empty) list. The name of the list phylum prefixed by Nil and Cons make up common names for the two list operators. The other way to define a list phylum is to use the built-in list operator. This not only looks more simple but causes the generation of additional list functions. Example 2.1 shows both definitions.

Example 2.1 Recursive and built-in List Definition

```
arithermlist:    Nilarithermlist ( )
               | Consarithermlist( aritherm arithermlist );

arithermlist: list  aritherm;
```

2.3 Attributes

Each phylum definition can contain declarations of attributes of phylum or arbitrary C++ types. They follow the operator enumeration as a block enclosed in braces. What purpose do they serve? With them, we can attach additional information to nodes, which otherwise could only unfavourably be represented in the tree. In our example, we may take advantage of attributes by saving intermediate results to support the calculation. Therefore we extend the definition of aritherm from example 1.5 to example 2.2.

Example 2.2 Phylum Definition with Attributes

```
aritherm:     SimpleTerm  ( simpleterm )
            | Plus        ( aritherm aritherm )
            | Minus       ( aritherm aritherm )
            | Mul         ( aritherm aritherm )
            | Div         ( aritherm aritherm )
            { int result        = 0;
              bool evaluated    = false;
              bool computable = true;
            };
```

Attribute *result* should hold the intermediate result of an aritherm, if it already has been evaluated (*evaluated*==true) and found computable during the evaluation (*computable*==true), that is the subterms contain no variables. The attributes can be initialized at the declaration or inside an additional braces block which may follow the declarations and can contain arbitrary C++ code. Example 2.3 shows an alternative to the initialization from example 2.2.

Example 2.3 Alternative Attributes Initialization

```
            { int result;
              bool evaluated;
              bool computable;
               { $0–>result       = 0;
                 $0–>evaluated = false;
                 $0–>computable = true; }
            };
```

That C++ code is executed when a term of that phylum has been created. It can be referred to as *$0* and its attributes can be accessed via the operator ->.

3 Aid with Term Handling

This chapter describes techniques necessary and useful to traverse a term structure: the application of term patterns and the use of special Kimwitu++ language constructs for term handling.

3.1 Patterns

Patterns are a means to select specific terms which can be associated with a desired action. Example 3.1 shows some patterns and explains what terms they match. Patterns are used in special statements and in rewrite and unparse rules (see 3.2 and 4 respectively).

Example 3.1 Patterns

```
Plus ( *, * )
// matches a term Plus with two subterms

c=Plus( a, b )
// matches a term Plus with two subterms, which are assigned to
// the variables a and b, while the term itself is assigned to c

Plus( Mul( a, b ), * )
// matches a term Plus with two subterms, whereof the first one
// is a Mul term with the subterms assigned to the variables a and b

Plus( a, b ), Minus( a, b )
// matches if the term in question is either a Plus or a Minus; the
// two subterms then are assigned to the variables a and b
```

3.2 Special Statements

Kimwitu++ provides two statements as extensions to C++ which make it more comfortable to deal with terms. These are the with-statement and the foreach-statement. They can be used in functions and in the C++ parts of unparse rules (see 4.2).

The with-statement can be considered as a switch-statement for a phylum. It contains an enumeration of patterns which must describe kinds of the specified phylum. From these the one is chosen which matches a given term best. Then the C++ code is executed, which was assigned to that pattern.

Example 3.2 takes a term of the phylum aritherm and calculates the attribute *result*, if the term is a sum or a difference, by adding or subtracting the results of the subterms. The keyword default serves as a special pattern in with, which matches when none of the others does.

Example 3.2 with-statement

```
aritherm a;
 ...
with( a ) {
  c = Plus  ( a, b  ) :   { c -> result = a -> result + b -> result; }
  c = Minus ( a, b  ) :   { c -> result = a -> result - b -> result; }
  default:                { }
}
```

The second special construct is the foreach-statement, which iterates over a term of a list phylum and performs the specified actions for every list element. Example 3.3 determines the greatest *result* from the terms in the list *a*.

Example 3.3 foreach-statement

```
int max = 0;
foreach( a; arithermlist A ){
  if ( a -> result > max ) max = a -> result;
}
```

4 Modifying and Evaluating Term Trees

This chapter describes how the tree of terms can be processed once it has been created. On one hand we can change its structure and hopefully simplify it by applying rewrite rules. On the other hand we can step through it and create a formatted output by applying unparse rules.

4.1 Transforming

Rewriting denotes the process of stepping through the tree, seeking the terms that match a pattern and substituting them by the appropriate substitution term. Rewrite rules consist of two parts, where the left-hand side is a pattern (as described in 3.1) and the right-hand side is a substitution term enclosed by angle brackets. A term must always be substituted by a simpler one, that is by one that is nearer to the desired form of result. Otherwise the substitution process may never stop.

Let us try simplifying according to figure 1.2 to demonstrate the usage of rewrite rules. What would the rules look like in Kimwitu++ to achieve a simplification of that kind? Example 4.1 shows a solution using rewriting. It is quite short in comparison with example 1.4, isn't it?

Example 4.1 Term Substitution using Kimwitu++

```
Mul( Plus( a, b  ),  Plus( c ,  b  ) ) –> <:  Plus( Mul( a, c  ),  b  )> ;
```

An equivalent part would cover the case that *b* takes the first position in both subterms. The meaning of the colon will be explained in 4.3.

4.2 Traversing

Originally unparsing was meant to be a reverse parse, used to give a formatted output of the tree built. In general it should better be recognized as a way to traverse the tree. Unparse rules have a structure similar to that of rewrite rules. The left-hand side consists of a pattern, the right-hand side of a list of unparse items enclosed by square brackets. Some more common unparse items are strings, pattern variables, attributes, and blocks of arbitrary C++ code enclosed in braces.

Unparsing starts by calling the unparse-method of a term, usually the root term, and when a rule matches a term the specified items are 'printed'. Only string items are really delivered to the current printer. This printer has to be defined by the user, and it usually writes to the standard output or into a file. Variable items and attribute items are further unparsed, code fragments are executed. If no pattern matches then the default rule is used, which exists for every phylum operator and which simply unparses the subterms.

We could do quite a lot of different things with the information saved in the tree. It just depends on the rules we use. For example we choose to print the input term in infix notation, because it is better readable to humans.

```
Plus( a, b ) -> [ infix : "(" a "+" b ")" ];
```

For every Plus term this rule prints an opening parenthesis, unparses the first subterm, prints a plus sign, unparses the second subterm, and then prints the closing parenthesis. The other operators are handled by similar rules.

We also may want to eventually compute the result of expressions. This can be achieved with rules like this.

```
c = Plus( a, b ) -> [: a b { c -> result = a -> result + b -> result ; } ];
```

Here the subterms are unparsed and then an attribute of the term gets assigned the sum of the subterm results. This will work only if these do not contain Idents. The meaning of the colon will be explained in 4.3.

The C++ code is enclosed by braces, but can itself contain braces in matching pairs. If a single brace is needed, as when mixing code and variable items, it has to be escaped with the dollar sign. Example 4.2 shows an application. If the function yields true the first branch is taken, and *b* is unparsed before *a*.

Example 4.2 Escaped Braces in Unparse Rule

```
Plus( a, b  ) -> [ : {  if( smaller_than( a, b  ) ) }  ${
                           b "+" a $}
                        { else } ${
                           a "+" b $ } ];
```

4.3 Views

The whole process of rewriting and unparsing as well as parts of it can be executed under different views. Each rule contains a view list between the respective opening brace and the colon, and it is used only if the current view appears in the list. This allows to specify different rules for a pattern. If a term is visited twice under different views, different rules are applied. These rules can be merged into one by listing all right-hand sides after one left-hand side, separating them by a comma.

Example 4.3 defines two rewrite views (simplify and canonify) and two rules, each of which will only be applied to a matching term if the current view is among the specified ones. The first rule replaces the quotient of the same two identifiers by the number 1, the second expresses the associativity of addition. Both change the tree.

Example 4.3 Views in Rewrite Rules

```
%rview simplify, canonify;

Div( SimpleTerm( Ident( a ) ), SimpleTerm( Ident( a ) ) )
     -> < simplify: SimpleTerm( Number( mkinteger( 1 ) ) ) >;

Plus( Plus( a , b  ),  c )
     -> < canonify: Plus( a, Plus( b , c  ) ) >;
```

Example 4.4 defines two unparse views (infix and postfix) and two rules for the same pattern, the one of which is used which matches both the term and the current view. It is possible to force a variable item to be unparsed under a desired view by specifying it after the variable and an intermediate colon. Subterm *b* is further unparsed under the view check_zero instead of the view infix.

```
%uview check_zero;

Div( a , b  ) -> [  infix  :  a "/"  b  :  check_zero ];
```

Example 4.4 Views in Unparse Rules

```
%uview infix, postfix;

Plus( a , b  ) -> [  infix     :  a "+" b  ],
                  [  postfix  :  a b "+"  ];
```

Part II

Reference Manual

This part lists all concepts of Kimwitu++ and explains them in detail. It is meant as a complete documentation of all features and contains advices of do's and don'ts. To the advanced user it should serve as a programming reference.

5 Definition of Phyla

5.1 Basic Definition

A phylum definition consists of two sides. The left-hand side specifies the phylum name, the right-hand side, behind a colon, a set of alternative operators, which are separated by bars. An operator is followed by a matching pair of parentheses, which may enclose a list of phyla as arguments. A nullary operator has an empty list. Example 5.1 presents a basic definition with operators of different arity. Before the semicolon closing the definition an attribute block may appear. The definition of a phylum follows the general form:

```
phylum  := phylum_name ":" operator { " | " operator } [ attributes  ] ";"
operator := operator_name "(" [ phylum_name ] { " " phylum_name } ")"
```

For the names of phyla and operators, the same restrictions hold as for C++ identifiers. Multiple definitions of of a phylum with the same name are interpreted as additional alternative branches of the first definition.

There are some simple predefined phyla: integer, real, casestring, and nocasestring, representing integer values, real values, case sensitive strings, and case insensitive strings respectively. Terms of these are not created by calls of operators but of special generated functions: mkinteger(), mkreal(), mkcasestring() and mknocasestring().

The phylum abstract_phylum represents the direct base of all phyla. Adding properties, like attributes or methods, to it makes them available for all phyla. Other direct bases may be specified for a phylum by using the keyword %base. One of them has to be derived from abstract_phylum, not necessarily directly. Example 5.2 makes phylum math_expression to be derived from a phylum expression and a user defined C++ class counter_class.

Example 5.1 Definition of a Phylum

```
expression:
    Plus    ( expression expression )
  | Neg     ( expression )
  | Null     ( ) ;
```

Example 5.2 Changing base of a phylum

```
%base math_expression : expression, counter_class;

%{ KC_TYPES_HEADER /* code redirection */
  class counter_class {
     ...
  };
%}
```

5.2 List Phyla

A list phylum can be defined by using the basic form of phylum definitions in a (right-) recursive way. The nullary operator constructs a list which is empty, and the binary, a list which is concatenated of a single list element and a list. The other variant uses the predefined operator list after which is specified the phylum of the list elements. The latter notation is to prefer because it is concise and causes the generation of additional list functions. That includes two operators which are named according to the scheme which is used in the right-recursive definition (prefixes Nil and Const). Other names could be chosen as well but this may cause some confusion. With either variants, a term is created by calling one of the two operators. The two definitions in example 5.3 yield the same list of elements of a phylum expression.

Example 5.3 Alternative Definitions of one List

```
expression_list:
     Nilexpression_list   ( )
   | Consexpression_list ( expression expression_list )
   ;

expression_list:  list   expression;
```

5.3 Attributes of Phyla

A phylum definition may have an attribute part attached which is enclosed in braces. It contains declarations of attributes of phylum types or other C++ types and optionally their initial value assignment. After that arbitrary C++ code can follow again enclosed in braces. This is the general form of the attribute part:

```
attributes  := "{" { attr_type  " "  attr_name [ "=" init_value   ] ";"} [   cpp_part  ] "}"
```

```
cpp_part    := "{"  arbitrary_cpp_code "}"
```

In particular, the initialization of attributes can be done in the C++ part too; though technically then it is not initialization, but assignment. Attributes may also be defined outside the phylum definition according to the general form beneath.

```
attributes := "%attr  "  attr_type phylum_name "::" attr_name ";"
members := "%member " attr_type phylum_name "::" attr_name ";"
```

Only attributes of phylum types can be defined with %attr. This is because they are considered part of the enclosing phylum by Kimwitu++. When a term is written to a file using the CSGIO functions (see 7.2.2), its attributes are saved and can thus be restored.

Using the keyword %member instead allows also C++ types to be used. The values of attributes defined in this way will get lost when their term is written to a file. Restoring a saved term will assign the initial values to all %member attributes.

The C++ code is executed after the term creation and the attribute initializations. Inside that code the newly created term can be referred to as *$0*. Attributes are accessed via the operator ->. Each predefined phylum has an attribute which holds its value, which is *value* for integer and real, and *name* for casestring and nocasestring. Example 5.4 shows three alternative ways to define and initialize one attribute (of C++ type).

Example 5.4 Alternative Definitions of one Attribute

```
expression: Div ( expression expression )
  { bool is_valid = true; };

expression: Div ( expression expression )
  { bool is_valid;
    { $0 –> is_valid = true ; } };

expression: Div ( expression expression );
%member bool expression::is_valid = true;
```

5.4 Supplemental Definitions

It is possible to supplement the classes which will be generated from phylum definitions with additional methods. These are defined as usual but have as qualifier the name of either a phylum or of an operator. Such a method is known for all terms of the phylum or only for those constructed by the specified operator. Predefined phyla can get additional methods too.

Example 5.5 Additional Methods Definitions

```
bool aritherm::equals( aritherm a ) {...}

int Number::get_int ( ) {...}

nocasestring casestring::convert_to_nocase ( ) {...}
```

It may appear desirable to initialize attributes of a phylum with non-default values immediately at term creation. This can be realized by using the Kimwitu++ keyword %ctor to define own constructors which will replace the default ones. Additional constructors are possible for phyla as well as for operators, but not for predefined phyla.

If these constructors are defined to have arguments then of some points are to take note. First, when used with operators the arguments will be added to those in the operator definition. Second, since the default constructors are replaced but relied on by some internal mechanism, the user has to define new ones or alternatively just provide default values for all new arguments. The latter may cause performance loss in the case of many or non-simple arguments.

The keyword %dtor allows own destructor definitions for freeing memory of attributes or something similar. It is applicable to phyla and operators but not to predefined phyla.

Example 5.6 User provided Constructors and Destructors

```
%ctor simpleterm( int i ) {...}
%ctor simpleterm ( ) {...}

%ctor Minus( bool neg = true ) {...}
// Minus now has 3 arguments, but the third is optional

%dtor virtual term ( ) {...}
```

5.5 Phylum Storage Options

For every phylum a C++ class is generated, with one create-method for every operator. A term is created by calling the appropriate method, which returns a pointer to the object representing the term. As a default for every such object a new cell is allocated in memory. But the user may influence the memory management for optimization purposes. At phylum definition time the phylum can be declared to use a special storage class. There is one predefined storage class:

uniq. It is allowed to specify !uniq what is the same as specifying nothing and results in usage of the default storage. Subphyla of a phylum with storage class can not use the default storage, but must be defined with a storage class too. The completed general form of a phylum definition is the following one.

```
phylum := phylum_name [ "{" storage_class "}" ] ":" operator
          { "|" operator } [ attributes ] ";"
```

Example 5.7 Phylum with Storage Class

```
aritherm    {! uniq } : Plus    ( aritherm aritherm );

simpleterm {uniq}   : Number ( integer );
```

The first phylum in example 5.7 declares explicitly to use the default storage. All other phyla defined until now got that implicitly. Memory of such terms can be freed individually, terms of the second phylum can not. They are kept under uniq storage. What does this mean? Each storage class has a hashtable assigned. All terms of phyla with that class are stored there. If a term is to be created the create routine does conditionally allocate new storage. It checks first whether there is already stored an object created with the same arguments. If found true, the routine will return a pointer to that object in memory. Such every term is created only once. All predefined phyla, such as integer, are declared uniq. So the example has the effect that if two simpleterms are created from the same int value they will both point to the same memory cell.

It is possible to define additional storage classes, which each get their own table. Tables also can be explicitly created and assigned to storage classes, as well as cleared or freed (see 7.2.4). Example 5.8 declares two additional storage classes and defines two phyla using them.

Example 5.8 Storage Class Definition and Application

```
%storageclass table1 table2;

intermediate {table1}: Div    ( aritherm aritherm );

final        {table2 }: Ident  ( casestring );
```

6 Processing phyla

6.1 Pattern Matching

Patterns make it easier to select terms and subterms and to distinguish cases. They appear in rules for rewriting and unparsing and in with- and foreach-statements. Here there are explained common features while the slight differences will be mentioned in the appropriate place.

The term 'pattern' can be defined through induction. Each of the following is a pattern in the context of Kimwitu++:

1. the literal of a predefined phylum or the asterisk sign,
2. the phylum operator with zero or more patterns as arguments,
3. the assignment of a pattern to a variable, and
4. the enumeration of patterns delimited by commas.

Additionally some restictions hold regarding the use of patterns. The patterns of item 1 are not allowed as the outermost pattern, while these of item 4 are allowed only as the outermost pattern. The assignment of an asterisk to a variable can be abbreviated by stating only the variable.

If more than one pattern matches a term, the most specific pattern is chosen. If there is no most specific one, the first of the matches is chosen. The matched term can be accessed as *$0*, its first subterm as *$1*, the second as *$2* etc. (not in rewrite rules). Table 6.1 lists pattern examples, which are in each group equally specific. Kimwitu++ is not yet able to decide between more complex patterns (maybe partly overlapping each other) which to be most specific, but chooses the first found.

6.2 Kimwitu++ Control Structures

Kimwitu++ provides two control structures which help dealing with terms: the with-statement and the foreach-statement. They appear in different variants, fitting slightly different purposes.

Table 6.1 Pattern groups increasingly specific

pattern	**matches**
*	any term; not allowed as the outermost pattern.
—	—
SimpleTerm	term SimpleTerm with an unspecified number of subterms; only allowed as the outermost pattern.
SimpleTerm(*)	term SimpleTerm with a subterm.
a=SimpleTerm(b)	term SimpleTerm with a subterm; the term is assigned to a variable *a*, the subterm, to *b*.
Number(7)	term Number with an integer subterm of value the 7.
SimpleTerm(*),Mul(b,b)	either term SimpleTerm with a subterm or term Mul with two equal subterms assigned to *b*.
—	—
SimpleTerm(Number(*))	term SimpleTerm with a subterm being a Number.
a=SimpleTerm(Number(b))	term SimpleTerm with a subterm being a Number; the term is assigned to *a* and the sub-subterm to *b*.
—	—
SimpleTerm(Number(b)) provided (b->value!=0)	term SimpleTerm with a subterm being a Number, which is assigned to *b*, but matches only if the integer *b* is not zero.

6.2.1 Explicit with

The with-statement is similar to a C++ switch and decides for a given term which alternative branch to choose. The alternatives are patterns describing kinds of one phylum and have assigned a C++ block, maybe an empty one.

The example 6.1 lists a code piece containing an explicitly stated with. It decides whether the term *a* is an identifier or a number and executes its code. The special pattern default matches when the preceding patterns do not. Here this would never occur, because a simpleterm is defined to be one of the two specified. If no default case is specified and none of the patterns matches the term a run-time exception is released (program execution aborted). The pattern default is allowed in all with-variants (implicit-, explicit-, and foreach-with).

6.2.2 Implicit with

If a function is defined to have a phylum argument whose variable name begins with a dollar sign, the function body is assumed to be the body of an implicitly given with-statement. Example 6.2 presents a function which returns `true` if the given aritherm matches one of the specified patterns, that is represents an

Example 6.1 Explicit with

```
simpleterm a;
 ...
with( a ){
  Ident( * )          : {  $0 −> computable = false; }
  c=Number( a )  : {  c −> computable = true; c −> result = a −> value; }
  default              : {  // never reached, because simpleterm has only
                                //  the above two kinds
                              }
}
```

arithmetic term. If the term has the kind SimpleTerm, the default case would catch.

Example 6.2 Implicit with

```
bool is_arithmetic_term( aritherm $a ) {
  Plus, Minus, Mul, Div:    { return true; }
  default:                          { return false; }
}
```

6.2.3 Simple foreach

The foreach-statement is a loop which runs over all elements of a term which is a list phylum and executes at every run the code which is specified in the statement body.

The foreach in example 6.3 counts the appearances of arithmetic terms by calling the function is_arithmetic_term for *a* during each run. This variable holds the current list element.

Example 6.3 Simple foreach

```
int count = 0;
foreach( a ; arithermlist A ) {
  if ( is_arithmetic_term( a ) ) count++;
}
```

6.2.4 foreach-with

This foreach variant also steps through a list, but performs a with for every element. A dollar prefixed list variable is used instead of a simple variable. The statement body contains patterns with a C++ block assigned to each. Example 6.4 demonstrates the usage. It counts the number of sums and differences within the term list.

Example 6.4 foreach-with

```
int num_plus = 0;
int num_minus = 0;
foreach( $a; arithermlist A ) {
  Plus:    { num_plus++; }
  Minus:  { num_minus++; }
}
```

6.2.5 foreach with Pattern

This third variant of foreach allows to specify a pattern instead of a list variable. The action is executed only for those list elements which match the pattern. Thus it combines foreach and an implicit with containing only one pattern of interest. Example 6.5 is similar to example 6.4 but it counts only the number of sums in the list.

Example 6.5 foreach with Pattern

```
int num_plus = 0;
foreach( Plus ( *, * );  arithermlist A ) {
  num_plus++;
}
```

6.2.6 Multiple Patterns

For every one of the preceding statements it is possible to specify not only one but multiple variables or patterns respectively. Used with foreach, multiple lists can be iterated over at one time. The variables or patterns are separated by ampersand signs (&), the list specifications by commas. Used with with, multiple terms can be checked at one time whether matching complex patterns. Here, the variables are separated by commas.

The complex patterns are made up by concatenating single patterns by ampersand signs (&), where the first pattern has to match the first term, the second the second term (and so on) to have the complex pattern to match. A complex pattern can also be a grouping of patterns, but then it must be enclosed in parentheses.

Example 6.6 shows a with over two variables. The statement simply prints out whether the two terms have the same kind.

Example 6.6 Multiple Patterns in with

```
simpleterm a, b;
 ...
with ( a, b ) {
  Number & Number:           { std::cout << "numbers"     << std::endl; }
  Ident & Ident:             { std::cout << "identifiers" << std::endl; }
  Number & (Number, Ident):  { std::cout << "first␣num"   << std::endl; }
  default:                   { std::cout << "first␣ident" << std::endl; }
}
```

6.2.7 afterforeach

When a foreach iterates over more than one list at one time, the execution will stop if one of the lists reaches its end, while the others may still contain elements. The afterforeach-statement is useful if it is desired to iterate further over the remainders of the lists. The variables of the afterforeach refer to the list remainders. Their phyla are already known from the preceding foreach. The code fragment in example 6.7 uses foreach-with and afterforeach to decide whether list *A* is longer than list *B*, returning `true` if it is.

Example 6.7 afterforeach in Length Test

```
foreach( $a & $b; arithermlist A, arithermlist B ) {
  default: { /* do nothing */ }
}
// at least one list ran out of elements
afterforeach( $a & $b ) {
  Consarithermlist & Nilarithermlist: { return true;
                                  /* A has elements, B doesn't */ }
  default: { return false; }
}
```

6.3 Rewriting

The process of rewriting transforms a term tree. The left-hand side of each rewrite rule is a pattern specifying which term to substitute. The right-hand side denotes the term to substitute by, which has to be of the same phylum as the original one. This is done by calls of operators and term returning functions. Variables from the left-hand side may be used. Example 6.8 simplifies terms by replacing every sum of numbers by its evaluated result. A helper function is necessary since it is not possible directly to use C++ operators in a rewrite rule (except method calls, see 6.6).

Example 6.8 Rewriting

```
Plus( SimpleTerm( Number( a ) ), SimpleTerm( Number( b ) ) ) ->
  <: SimpleTerm( Number( plus( a, b ) ) ) >;

%{ KC_REWRITE /* code redirection */
integer plus( integer a, integer b ){
    return mkinteger( a -> value + b -> value );
  }
%}
```

To allow rewriting to end, each rule has to replace the term in question by one which is really simpler, reduced, or closer to a normal form. Since rewriting searches depth first, the subterms are usually already the result of a transformation. Calling the rewrite method of the root term starts the transforming process for the tree. Choosing an arbitrary term instead rewrites the subtree beneath.

6.4 Unparsing

The process of unparsing traverses a term tree and executes the instructions for every term matching a pattern as specified in the unparse rules. The left-hand side of a rule denotes a term pattern, the right-hand side a list of unparse items which are evaluated for matching terms. The various items allowed are listed below and appear all in example 6.9, which is completely nonsensically.

string Text strings in double quotes are delivered to the printer unchanged.

variable The term denoted by this term variable will be unparsed by calling its unparse-method.

attribute The attribute of a term will be unparsed by calling its unparse-method. If it is of non-phylum type the user has to provide such a method.

C++ code Inside a pair of braces arbitrary C++ code may be placed.

escaped braces If a non-matching brace is needed it has to be escaped by a dollar sign.

variable with view A view can be specified if a variable should be unparsed under other than the current view (see 6.5).

unparse view variable definition Variables of user defined unparse view classes can be defined inside a rule (see 6.5).

Example 6.9 Unparse Items

```
Mul( a, b ) -> [: "zero"
                  a
                  b->result
                  { if (a->value == 0) } ${
                      a:infix
                    $}
                  %uviewvar prefix p1;
                  a:p1
               ];
```

For every operator, there is a default pattern which matches if no other pattern does. Its associated rule unparses all subterms. The unparse-method generated for every phylum can also be called explicitly. It has 3 arguments: the term to be unparsed, a printer and an unparse view. The names *kc_printer* and *kc_current_view* respectively refer to the printer and the view which are currently in use.

6.4.1 Printer

The user himself has to provide a printer function which satisfies his needs. Usually it prints to the standard output or into some file, and may take actions dependent on the view.

```
void printer( const char* the_string, uview the_view ) { ... };
```

Since several printer instances may be needed also a printer functor can be specified as the printer. The printer functor class must be derived public from the class printer_functor_class.

```
%{ HEADER /* redirection since no class definitions allowed in .k */
class example_printer : public printer_functor_class {
  public:
    void operator( ) ( const char* the_string, uview the_view ) { ... };
}
%}
```

6.4.2 Language Options

Often a term tree has to be pretty printed into different but similar destination languages, which sometimes require only slightly different output to be generated. To avoid whole rule sets to be multiplied and to allow a more flexible choice concerning the destination language, the concept of language options has been introduced. Every unparse item can be preceded by a language option, which is a language name in angle brackets followed by a colon. That item will be unparsed only if the language specified is active. Languages are declared using the keyword %language and they are set by calling set_language(...). The active language can be checked by calling is_language(...). The language names must satisfy the requirements for C++ identifiers. Example 6.10 demonstrates the application of language options.

Example 6.10 Language Options

```
%language CPP, JAVA;

ClassDefinition( name, base_name, class_body ) –>
   [:  <JAVA>: "public␣" "class␣" name
       <JAVA>: "␣extends␣" <CPP>: "␣:␣"
       base_name "␣{\n" class_body "}"
       <CPP>: ";"
       "\n"
   ];
```

6.5 View Classes

Rewriting and unparsing of each term is done under a certain view. The view serves as a means to further differentiate between rules when choosing one to apply. To be a match a rule must have the current view to appear in its view list, which is the left part of the right-hand side between the bracket and the colon. If no rule matches the rules with empty list are considered. Below is shown the general form of rules with views.

```
rewrite_rule   := pattern "–>"  "<" rview_list " :"  {operator | function} ">;"
unparse_rule   := pattern "–>"  "["  uview_list " :"  unparse_items          " ];"
```

Views are declared by enumerating them after the keyword %rview and %uview for rewriting and unparsing respectively, separated by a space or a comma. These declarations enable Kimwitu++ to check for view consistency, although it is possible to leave them out entirely. But that should be avoided, because then even simple misspellings in a view list cause the implicit introduction of new views.

One view is predefined for rewriting and unparsing respectively, *base_rview* and *base_uview*, which is implicitly included in the empty view list.

The view can be changed for a term by calling its rewrite/unparse-method with a new view argument. In unparse rules there the same can also be achieved by appending a colon and a new view name to a variable unparse item. Thus a whole subtree can be rewritten/unparsed under a different view, or even multiple times under changing views. Changing views allows to interlock several tasks on a certain subtree.

```
%rview demo1 ;

Plus( a , SimpleTerm( Number( 0 ) ) ) —> <: a —> rewrite( demo1 ) >;

%uview demo2 ;

Plus( a , b  ) —> [: a:demo2 { b—>unparse( kc_printer, demo2 ); } ];
// both subterms of Plus are further unparsed under view demo2;
```

Every view introduced by the user actually causes the generation of a view class and one view variable of the same name. Since the user cannot distinguish them, the generalizing term 'view' is used. For unparse views that may matter since the user can define his own unparse view classes. These are declared by enclosing the view name in parentheses. The user has to provide a class *view*_class derived from a generated class *view*_baseclass. In particular, that class may contain member variables to hold information persistent over several rules. The base class provides an equality operator (==) deciding whether two view variables are of the same class and a name-method returning the name of the view.

No global view variable of the same name is generated for a user defined unparse view class. Variables of such a view are instantiated inside of unparse rules by %uviewvar and may bear the same name as their class. They can be used like the implicitly created view variables, but additionally provide all features of their class. Example 6.11 shows the definition of an unparse view class and demonstrates its usage.

View lists contain names of view classes, all other occurrences of views actually are view variables. The scope of an unparse view variable ends with its defining rule. Since its name is not known inside other rules there it can be accessed only by means of the name *kc_current_view*, which always refers to the view variable currently used.

6.6 Restrictions on C++ in Kimwitu++

There are many places in a .k-file where C++ code can be used. But for some of them, the Kimwitu++ processor allows only restricted use of C++ constructs. These places are listed in the following along with the restrictions they impose.

Example 6.11 User defined Unparse View Class

```
%uview ( number_count ) ;

%{ KC_UNPARSE /* code redirection to where it is needed */
class number_count_class : public number_count_baseclass {
  public:
    number_count_class() : counter( 0 ) { };
    int counter;
};
%}

c=Plus, c=Minus, c=Mul, c=Div, c=SimpleTerm -> [:
      %uviewvar number_count nc; //instantiate view variable
      c:nc                       // and unparse c with it
      { std::cout << "Numbers␣counted:␣" <<nc.counter <<std::endl; }
];
Plus( a, b ), Minus( a, b ), Mul( a, b ), Div( a, b )
                -> [ number_count: a b ];
SimpleTerm( a ) -> [ number_count: a ];
Number          -> [ number_count: { kc_current_view.counter++; } ];
Ident           -> [ number_count: ];
```

.k-file Only function definitions are allowed. These must have no types as arguments which have compound names (for example no long int). C++ comments are allowed everywhere in .k-files.

C++ unparse item Almost arbitrary C++ code is allowed, that is, everything which is allowed inside a local C++ block.

rewrite rule Only simple function calls are allowed, that is, calls which have as arguments only term variables and term literals, phylum operators and other simple function calls; in particular no C++ operators, except access of member functions.

code redirection Arbitrary C++ code is allowed, but it has to be pure C++ since redirection code is not evaluated by Kimwitu++.

There is a way to get around the restrictions of the Kimwitu++ processor using macros. A macro is defined inside a code redirection, which the processor does not evaluate. Therefore it can be as complex as necessary, while the macro call inside the rewrite rule looks as simple as Kimwitu++ wishes. Example 6.8 defines a function for addition, which can be avoided when using a macro as in example 6.12.

Example 6.12 Macro Application in Rewriting

```
Plus( SimpleTerm( Number( a ) ), SimpleTerm( Number( b ) ) ) ->
   <: SimpleTerm( Number( PLUS( a, b ) ) ) >;

%{ KC_REWRITE /* code redirection */
#define PLUS( a, b )     mkinteger( ( a ) -> value + ( b ) -> value )
%}
```

Some generated functions return terms of the phylum abstract_phylum which have to be cast to the actual phylum. The C++ cast operators may be used also for phylum conversion but Kimwitu++ provides phylum_cast, a cast operator for phyla, which is better to use.

7 Generated Code

From the Kimwitu++ definitions, rules and C++ code pieces, several classes and functions in pure C++ are generated and distributed over multiple files. Compiled, they will perform the desired tree handling. Additional code is needed to create the trees, probably created by scanner and parser generators, for instance Flex and Bison.

7.1 Generated Classes and Types

The definition of phyla and operators result in generated C++ classes. But these should be of no further interest for the user since the phylum names can be used in C++ code as if being pointer types of these classes, the operators as if being C++ constructors. Every phylum has a const counterpart of the same name prefixed by c_, which is the only means to get a const phylum variable.

Just for the sake of completeness, be it mentioned that every phylum corresponds to a class impl_*phylum* and every operator to a subclass impl_*phylum_operator*. All the classes are derived from a common base class which can be referred to as abstract_phylum. By adding constructors, methods or attributes to it, all phyla will be changed in that way.

The interworking with Yacc/Bison requires a type YYSTYPE which will be generated by Kimwitu++ when the option yystype is specified (see 8.1)

7.1.1 Smart-pointer

Memory often leaks when phylum operators are used in expressions, and that is sometimes hard to detect. The option smart-pointer enables a smart memory management which avoids unnecessary copying of terms and automatically frees memory of unused terms. This is achieved by using so called smart-pointers which do reference counting and allow to free a term if it is no longer referenced.

An additional type is generated for every phylum with the suffix _ptr. Variables of such types are unnecessary ever to be freed. Avoid mixing them with variables of the usual types, especially never assign between them, because that is likely to cause memory access errors.

7.1.2 Weak-pointer

The option weak-pointer extends the smart-pointer technique and supports a third type for every phylum. It gets prefix weak_ and suffix _ptr. Weak-pointer variables of a term will not contribute to the reference counting, such that the term already is freed if merely weak-pointers reference it yet. That is why they are only usefully employed in conjunction with smart-pointers. In contrast to usual variables, weak-pointers have their own reference counting, which allows to determine whether such a pointer dangles, that is points to a term already freed and thus is no longer valid.

7.2 Generated Functions

Kimwitu++ generates a number of functions which are available wherever C++ code is allowed in .k-files. The table 7.1 lists all these functions and the sections which contain a more detailed description.

Table 7.1 Generated Functions

function	see section	function	see section
append	7.2.5	last	7.2.5
concat	7.2.5	length	7.2.5
eq	7.2.1	map	7.2.5
CSGIOread	7.2.2	merge	7.2.5
CSGIOwrite	7.2.2	mkcasestring	7.2.3
filter	7.2.5	mkinteger	7.2.3
fprint	7.2.1	mknocasestring	7.2.3
fprintdot	7.2.1	mkreal	7.2.3
fprintdotepilogue	7.2.1	op_name	7.2.1
fprintdotprologue	7.2.1	phylum_name	7.2.1
free	7.2.4	print	7.2.1
freelist	7.2.4	reduce	7.2.5
ht_create_simple	7.2.4	reverse	7.2.5
ht_assign	7.2.4	rewrite	7.2.1
ht_assigned	7.2.4	set_subphylum	7.2.1
ht_clear	7.2.4	subphylum	7.2.1
ht_delete	7.2.4	unparse	7.2.1
is_nil	7.2.5		

7.2.1 Common Functions

Since abstract_phylum has an unparse-method defined and all phyla are derived from abstract_phylum all phyla have it. The same holds for some other methods.

copy

```
abstract_phylum copy( bool copy_attributes ) const;
```

The method copies this term completely, including its subterms. Since the result is always abstract_phylum it has to be casted to the phylum of this term. If `true` is specified as argument, the attributes are copied too. But beware! Merely the addresses are copied if the attributes are phyla or C++ pointers, that is, the new term references the attributes of the old one.

eq

```
bool eq( c_abstract_phylum c_p ) const;
```

The method returns `true` if this term is structurally equal to the argument, that is, both terms have equal subtrees.

fprint

```
void fprint( FILE* file );
```

The method prints a textual presentation of this term to the specified file. This simple example produces the output underneath.

```
simpleterm a_number = SimpleTerm( mkinteger( 23 ) );
a_number −> print( );

SimpleTerm(
  Number(
    23
  )
)
```

fprintdot

```
void fprintdot( FILE *f,
                const char *root_label_prefix,
                const char *edge_label_prefix,
                const char *edge_attributes,
                bool print_node_labels,
                bool use_context_when_sharing_leaves,
                bool print_prologue_and_epilogue
               ) const;
```

The method creates a presentation of this term in a format understood by the program dot. It is part of the graphic package graphviz and draws directed acyclic graphs in various output formats like PostScript or GIF. The presentation is written to file *f*, while the other arguments control the details of the graphs appearance.

root_label_prefix Adds a label to the graph denoting the root term. The label has the name of the phylum of that term prefixed by this string argument.

edge_label_prefix Every edge in the graph is labelled with a number. This string argument appears as the prefix of these labels.

edge_attributes For dot, the edges can have attributes which specify additional features like font name or edge colour (see dot manual for attribute names and values). This string argument is a list of attribute/value pairs (*attribute=value*), separated by commas.

print_node_labels If this argument is set to `true`, the names of the subterms phyla appear in the graph. Otherwise they are suppressed and only the term names (operator names) are printed.

use_context_when_sharing_leaves Terms which are shared in the tree usually appear only once in the graph (terms of uniq phyla). In particular, terms of predefined phyla are shared if they are equal. They are always leaves since they have no subphyla. If this argument is set to `true`, the leaves appear shared in the graph only if they are subterms of shared (uniq) terms.

print_prologue_and_epilogue This argument is usually set to `true` since a certain prologue and epilogue are necessary to frame the graph. This is set to `false` if multiple graphs are to be grouped into one figure. In that case the prologue function has to be called explicitly, then some fprintdot calls follow, and finally the epilogue call finishes the figure creation.

The following call of fprintdot writes a a presentation of the term *t* to a file *exa*. From that file dot creates a graph like that in figure 7.1.

```
aterm t = Plus( SimpleTerm( Number( mkinteger( 7 ) ) ),
                SimpleTerm( Number( mkinteger( 7 ) ) ) );
t −> fprintdot(exa, "root_", "edge", "style=dashed", true, false, true);
```

fprintdotepilogue

```
void fprintdotepilogue ( FILE *f );
```

The function writes an instruction to *f* which finishes the figure. Usually the figure contains only one graph and this function is called implicitly by fprintdot.

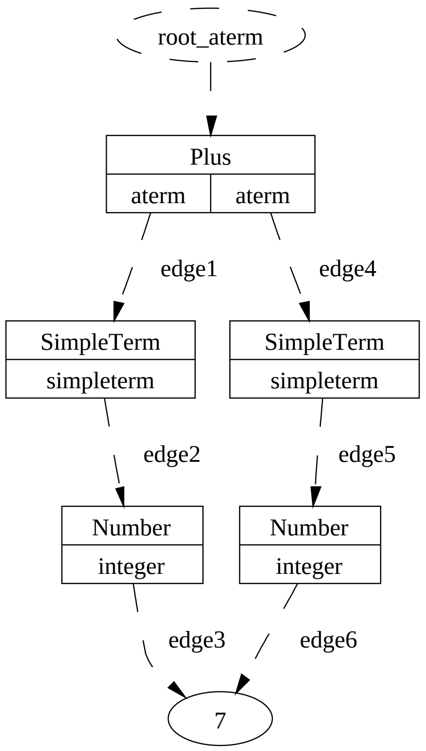

Figure 7.1: Dot Created Graph of an Example Term

fprintdotprologue

void fprintdotprologue (FILE ∗f);

The function writes an instruction to *f* which starts the figure. Usually the figure contains only one graph and this function is called implicitly by fprintdot.

op_name

const char∗ op_name( ) **const**;

The method returns the name of the phylum operator which has been used to create this term.

phylum_name

const char∗ phylum_name( ) **const**;

The method returns the name of the phylum of this term.

print

void print( );

The method prints a textual presentation of this term to the standard output. It is similar to the output of fprint.

set_subphylum

void set_subphylum(**int** n, abstract_phylum p, **bool=false**);

The method replaces the *n*th subterm of this term by term *p*, which must be of a phylum castable to the phylum of the appropriate subterm. Numbering starts with `0`.

subphylum

abstract_phylum subphylum(**int** n, **bool=false**) **const**;

The method returns the *n*th subterm of this term. Numbering starts with `0`.

unparse

void unparse(printer_functor pf, uview uv);
void unparse(printer_function opf, uview uv);

The method starts unparsing for this term. It is recursively called for every subterm. Unparsing is processed under the specified unparse view, and the strings to output are delivered to the printer functor or function respectively.

rewrite

<actual phylum> rewrite(rview rv);

The methods starts rewriting for this term. It returns a new term of the actual phylum. Usually it is called at the root term whereupon the entire tree is searched under the specified view.

7.2.2 CSGIO Functions

The generated files `csgiok.h` and `csgiok.cc` provide means to write terms to files and to reconstruct terms from such files. Whole term trees thus can be saved and exchanged between different applications. Reading and writing is performed by two functions.

The format of the files has once been designed to be compatible to the structure files of the commercial tool Synthesizer Generator. The format written now by Kimwitu++ is somewhat extended so that they are not compatible any more, but old structure files are expected to be still understood.

CSGIOwrite

void CSGIOwrite(FILE ∗f) **const**;

The methods writes this term to *f*, that is, the entire subterm tree. The attributes are ignored except they are phyla which have been defined using the keyword %member.

CSGIOread

template<typename P> **void**
CSGIOread(FILE ∗f, P &p)

The function reads from *f* the presentation of a term. The term is constructed by successively calling the appropriate operators of the subterms. The operators initialize the attributes according to the phylum definition; except the %member-attributes which get their values from the saved term. The created term is assigned to *p* which has to be a variable of the correct phylum.

7.2.3 Creation Functions

Terms of predefined phyla are created by functions.

mkcasestring

casestring mkcasestring(**const char** ∗str);
casestring mkcasestring(**const char** ∗str, **unsigned int** length);

The function creates a term of the phylum casestring from the specified string. Upper and lower case characters are distinguished. The second variant uses only the first *length* characters of the specified string.

mkinteger

```
integer mkinteger( const INTEGER i );
```

The function creates a term of the phylum integer from the specified value. INTEGER is a macro which can be defined by the user as needed but defaults to int.

mknocasestring

```
nocasestring mknocasestring( const char *str );
nocasestring mknocasestring( const char *str, unsigned int length );
```

The function creates a term of the phylum nocasestring from the specified string. Upper and lower case characters are not distinguished. The second variant uses only the first *length* characters of the specified string.

mkreal

```
real  mkreal( const REAL r );
```

The function creates a term of the phylum real from the specified value. REAL is a macro which can be defined by the user as needed but defaults to double.

7.2.4 Memory Management Functions

When terms, once constructed, are no longer needed it is usually reasonable to free the memory they allocate, especially when dealing with large numbers of terms.

The same does not hold not for the use of smart-pointers, because these keep track of allocated memory by their own. Never apply free or freelist to smart-pointers. The C++ delete should never be applied to any term, since that would get around some Kimwitu++ mechanisms.

free

```
void free( bool recursive=true );
```

The method frees the memory allocated by this term and by default it frees also the subterms recursively. When it is applied to a list term, the whole list and all its elements are freed. The non-recursive form only separates the list into its first element and the remainder of the list. Terms of phyla under non-default storage management can not be freed individually, calling free on them has no effect.

freelist

```
void freelist ( );
```

The method frees the spine of this list term and leaves the list elements untouched.

Hashtable Functions

The memory management of terms of storage class uniq or a user defined one can only be influenced by hashtable operations.

ht_create_simple

```
hashtable_t ht_create_simple ( int size );
```

The function creates a new hashtable and returns it. The current implementation ignores the *size* argument.

ht_assign

```
hashtable_t ht_assign ( hashtable_t ht, storageclass_t sc,
  bool still_unique=false );
```

The function assigns the hashtable *ht* to the storage class *sc* and returns the hashtable which has previously been assigned to *sc*.

ht_assigned

```
hashtable_t ht_assigned ( storageclass_t sc );
```

The function returns the hashtable which is assigned to the storageclass *sc*.

ht_clear

```
void ht_clear ( hashtable_t ht );
```

The function removes all entries from the hashtable *ht*.

ht_delete

```
void ht_delete ( hashtable_t ht );
```

The function deletes the hashtable *ht* entirely.

7.2.5 List Functions

List phyla which have been defined using the list keyword get some methods performing convenient tasks. In the function signatures, the name <phylum list> denotes the actual list phylum, phylum> denotes the phylum of the list elements.

append

```
<phylum list> append( <phylum> p );
```

The method appends the specified term to this list and returns this list.

concat

```
friend <phylum list> concat( c_<phylum list> l1, c_<phylum list> l2 );
```

The function constructs a new list from the terms of *l1* followed by the terms of *l2* and returns that list.

filter

```
<phylum list> filter( bool (∗fp) (<phylum>) );
```

The method constructs a new list from the terms of this list for which the function fp yields `true`.

is_nil

```
bool is_nil ( ) const;
```

The method returns `true` if this list is empty.

last

```
<phylum list> last( ) const;
```

The method returns the remainder of this list which contains only one, the last, element. If this list is empty the empty list is returned.

length

```
int length ( ) const;
```

The method returns the number of elements in this list.

map

```
<phylum list> map( <phylum> (∗fp) (<phylum>) );
```

The method constructs a new list containing the terms which are returned by the fp which is called for every element of this list. The new list is returned.

merge

```
<phylum list> merge( <phylum list> l, <phylum> (*fp) (<phylum>, <phylum>) );
```

The method constructs a new list containing the terms which are returned by the fp which is called for every element of this list taking the second argument from the specified list. The new list is returned.

reduce

```
<phylum> reduce( <phylum> p, <phylum> (*fp) (<phylum>, <phylum>) );
```

The method successively applies the function fp to each element of this list and fps last result which initially is the term *p*. The final result is returned.

reverse

```
<phylum list> reverse( ) const;
```

The method constructs a new list which contains the elements of this list in reverse order. The new list is returned.

7.3 Generated Files

The generated code is spread over several files. The table 7.2 lists these files and a description of their contents. Every file defines a macro symbol which can be used in preprocessor instructions and in code redirections. These symbols are listed as well. From every Kimwitu++ file a C++ file and a header file are generated. The name *file* in the table refers to such files.

7.3.1 Code Redirection

A .k-file can contain pieces of arbitrary C++ enclosed between a line starting with %{ and one starting with %}. Since it will not be parsed by Kimwitu++ but copied directly into generated code, it can not contain special Kimwitu++ constructs, but merely pure C++. It will go to the matching .cc-file, if no redirection is specified. Giving a list of file symbols after %{ will copy the code each of the specified files instead. The available redirection symbols are listed in table 7.2.

Table 7.2 Generated files

file	**symbol**	**contents**
`csgiok.cc`	KC_CSGIO	functions for saving and restoring of terms
`csgiok.h`	KC_CSGIO_HEADER	some definitions for saving and restoring of terms
`k.cc`	KC_TYPES	implementation of all classes generated from phylum definitions
`k.h`	KC_TYPES_HEADER	all class declarations generated from phylum definitions; included by all implicitly generated files
`rk.cc`	KC_REWRITE	rewrite methods for all phyla
`rk.h`	KC_REWRITE_HEADER	rewrite view class definitions
`unpk.cc`	KC_UNPARSE	unparse methods for all phyla
`unpk.h`	KC_UNPARSE_HEADER	unparse view class definitions
`file``.cc`	KC_FUNCTIONS_*file* or CODE	function definitions from *`file`*`.k`-file
`file``.h`	KC_FUNCTIONS_*file*_HEADER or HEADER	declarations of functions from *`file`*`.k`-file

```
// this be a file example.k

%{
// everything between the brace lines will be copied to example.cc
%}

%{ HEADER KC_UNPARSE /* beware of //-comments here */
// everything between the brace lines will be copied to example.h and unpk.cc
%}
```

8 Running Kimwitu++

The Kimwitu++ processor is invoked with a command like kc++. It can be invoked on a number of .k-files. When used together with other tools (see 8.2) a makefile would help. But every source file may influence every generated file, because of the code redirections. Thus multiple destination files depend on multiple source files. That means the makefile becomes more complicated in order to handle these dependencies. That is why a makefile is added to the appendix (see A.6). It suffices the RPN example and with file names adapted surely many more.

8.1 Options

Kimwitu++ recognizes a number of command line options which affect the process of parsing and code generation, some rather drastically. Table 8.1 presents, in alphabetical order, all available options and their explanation. Two forms are provided, short and GNU style long options.

Some vital options can be specified directly in Kimwitu++ using the keyword %option. Such specified options have higher priority than command line options and are thus enforced to be observed. Table 8.2 lists them in alphabetical order. They behave like their command line counterparts. A line like this could be specified in a Kimwitu++ file:

```
%option yystype smart−pointer
```

8.2 Yacc/Bison

Interfacing with a compiler generator is useful when a tree should be build from some kind of input. Kimwitu++ provides the yystype-option (see 8.1) which causes the generation of a header file needed by Yacc to cooperate. For every token found, the desired Kimwitu++ operator is called to create a term. If lex/flex is used too, Yacc has to be run with the option which causes the generation of an other header file needed by lex/flex (-d for Bison). Appropriate files for the example can be found in appendix A. The makefile uses implicit rules for flex and Bison.

Table 8.1 Command line options

	option	**explanation**
–b	– –yystype[=FILE]	generate file (default `yystype.h`) containing YYSTYPE, for Yacc and Bison
–c	– –no–csgio	do not generate phylum read/write functions (`csgiok.{h,cc}`)
–d	– –no–printdot	no fprintdot functions are generated
–e	– –dllexport=STRING	generates string between keyword class and the class name of all operators and phyla
–f	– –file–prefix=PREF	prefix all generated files
–h	– –help	display the help and exit
–l	– –no–linedirec	changes line directives (#line) to mere comments
–m	– –smart–pointer	generates code for smart pointers (reference counting)
–n	– –covariant=C	use covariant return types: `y|n|p` (yes, no or generate both and decide per preprocessor macro NO_COVARIANT_RETURN)
–o	– –overwrite	always write generated files even if not changed
–p	– –pipe=CMD	process all files while piping them through CMD
–q	– –quiet	quiet operation (is default)
–r	– –no–rewrite	do not generate code for rewrite rules (`rk.{h,cc}`)
–s	– –suffix=EXT	extension for generated source files (default `.cc`)
–t	– –no–hashtables	do not generate code for hashtable operations
–u	– –no–unparse	do not generate code for unparse rules (`unpk.{h,cc}`)
–v	– –verbose	print additional status information while processing
	– –stdafx[=FILE]	generate include for Microsoft precompiled header files (default `stdafx.h`)
–V	– –version	output version information and exit

Table 8.2 Built-in options

no–csgio
no–hashtables
no–printdot
no–rewrite
no–unparse
smart–pointer
weak–pointer
yystype

A Complete Code of RPN Example

A.1 main.k

```
%{
#include <iostream>
#include "k.h"
#include "rk.h"
#include "unpk.h"
#include "csgiok.h"

int yyparse( );
aritherm root_term;
%}

%{ KC_TYPES_HEADER
extern aritherm root_term;
%}

%option yystype

void
printer( const char *s, uview v ) {
  std::cout << s;
}

int
main( int argc, char** argv ) {
  std::cout << "rpn␣calculator" <<std::endl;
  yyparse( );
  aritherm canon_term = root_term −> rewrite( canonify );
  aritherm simpl_term = canon_term −> rewrite( simplify );
  std::cout << "simplified␣term␣in␣infix␣notation" <<endl;
  simpl_term −> unparse( printer, infix );
  std::cout << std::endl;
}
```

A.2 abstr.k

```
aritherm:   SimpleTerm ( simpleterm )
          | Plus       ( aritherm aritherm )
          | Minus      ( aritherm aritherm )
          | Mul        ( aritherm aritherm )
          | Div        ( aritherm aritherm );

simpleterm:  Number  ( integer )
           | Ident   ( casestring );
```

A.3 rpn.k

```
%rview simplify, canonify;

// 1 + 2 –> 3
Plus(SimpleTerm(Number(a)),SimpleTerm(Number(b)))–>
  <simplify:SimpleTerm(Number(plus(a,b)))>;
// 1 + 3 + b –> 4 + b
Plus(SimpleTerm(Number(a)),Plus(SimpleTerm(Number(b)),rem))–>
  <simplify:Plus(SimpleTerm(Number(plus(a,b))),rem)>;
// 1 + 3 – b –> 4 – b
Plus(SimpleTerm(Number(a)),Minus(SimpleTerm(Number(b)),rem))–>
  <simplify:Minus(SimpleTerm(Number(plus(a,b))),rem)>;
// 6 – 2 –> 4
Minus(SimpleTerm(Number(a)),SimpleTerm(Number(b)))–>
  <simplify:SimpleTerm(Number(minus(a,b)))>;
// 6 – 4 – b –> 2 – b
Minus(SimpleTerm(Number(a)),Minus(SimpleTerm(Number(b)),rem))–>
  <simplify:Minus(SimpleTerm(Number(minus(a,b))),rem)>;
// 6 – 4 + b –> 2 + b
Minus(SimpleTerm(Number(a)),Plus(SimpleTerm(Number(b)),rem))–>
  <simplify:Plus(SimpleTerm(Number(minus(a,b))),rem)>;
// 3 * 2 * b –> 6 * b
Mul(SimpleTerm(Number(a)),Mul(SimpleTerm(Number(b)),rem))–>
  <simplify:Mul(SimpleTerm(Number(mul(a,b))),rem)>;
// 3 * 2 / b –> 6 / b
Mul(SimpleTerm(Number(a)),Div(SimpleTerm(Number(b)),rem))–>
  <simplify:Div(SimpleTerm(Number(mul(a,b))),rem)>;
// 3 * 2 –> 6
Mul(SimpleTerm(Number(a)),SimpleTerm(Number(b)))–>
  <simplify:SimpleTerm(Number(mul(a,b)))>;
```

```
// 6 / 2 -> 3
Div(SimpleTerm(Number(a)),SimpleTerm(Number(b)))->
  <simplify:SimpleTerm(Number(div(a,b)))>;
// 6 / 2 / b -> 3 / b
Div(SimpleTerm(Number(a)),Div(SimpleTerm(Number(b)),rem))->
  <simplify:Div(SimpleTerm(Number(div(a,b))),rem)>;
// 6 / 2 * b -> 3 * b
Div(SimpleTerm(Number(a)),Mul(SimpleTerm(Number(b)),rem))->
  <simplify:Mul(SimpleTerm(Number(div(a,b))),rem)>;

// a + a -> 2 * a
Plus(b=SimpleTerm(Ident(a)),SimpleTerm(Ident(a)))->
  <simplify:Mul(SimpleTerm(Number(mkinteger(2))),b)>;
// a - a -> 0
Minus(SimpleTerm(Ident(a)),SimpleTerm(Ident(a)))->
  <simplify: SimpleTerm(Number(mkinteger(0)))>;
// a / a -> 1
Div(SimpleTerm(Ident(a)),SimpleTerm(Ident(a)))->
  <simplify: SimpleTerm(Number(mkinteger(1)))>;
// 6 * a + a -> 7 * a
Plus(Mul(SimpleTerm(Number(a)),SimpleTerm(Ident(b))),c=SimpleTerm(Ident(b)))->
  <simplify: Mul(SimpleTerm(Number(plus(a,mkinteger(1)))),c)>;
// 6 * a -a -> 5 * a
Minus(Mul(SimpleTerm(Number(a)),SimpleTerm(Ident(b))),c=SimpleTerm(Ident(b)))->
  <simplify: Mul(SimpleTerm(Number(minus(a,mkinteger(1)))),c)>;
// 6 * a + 3 * a -> 9 * a
Plus(Mul(SimpleTerm(Number(a)),SimpleTerm(Ident(b))),
     Mul(SimpleTerm(Number(d)),c=SimpleTerm(Ident(b))))->
  <simplify: Mul(SimpleTerm(Number(plus(a,d))),c)>;
// 6 * a - 2 * a -> 4 * a
Minus(Mul(SimpleTerm(Number(a)),SimpleTerm(Ident(b))),
      Mul(SimpleTerm(Number(d)),c=SimpleTerm(Ident(b))))->
  <simplify: Mul(SimpleTerm(Number(minus(a,d))),c)>;
// a + (a + 2 * b) -> 2 * a + 2 * b
Plus(b=SimpleTerm(Ident(a)),Plus(SimpleTerm(Ident(a)),rem))->
  <simplify: Plus(Mul(SimpleTerm(Number(mkinteger(2))),b),rem)>;
    // a - (a +- 2 * b) -> 2 * b
Minus(SimpleTerm(Ident(a)),Plus(SimpleTerm(Ident(a)),rem)),
Minus(SimpleTerm(Ident(a)),Minus(SimpleTerm(Ident(a)),rem))->
  <simplify: rem>;
// a + (a - 2 * b) -> 2 * a - 2 * b
Plus(b=SimpleTerm(Ident(a)),Minus(SimpleTerm(Ident(a)),rem))->
  <simplify: Minus(Mul(SimpleTerm(Number(mkinteger(2))),b),rem)>;

// (a + b) + c -> a + (b + c)
```

```
Plus( Plus(a, b ),  c)
        −> < canonify: Plus(a, Plus(b, c ))>;
// (a − b) + c −> c + (a − b)
Plus( Minus(a, b), c)
        −> < canonify: Plus(c, Minus(a, b))>;
// (a + b) − c −> a + (b − c)
Minus( Plus(a, b), c)
        −> < canonify: Plus(a, Minus(b, c))>;
// (a ∗ b) ∗ c −> a ∗ (b ∗ c)
Mul( Mul(a, b), c)
        −> < canonify: Mul(a, Mul(b, c))>;
// a + 5 −> 5 + a
Plus( a=SimpleTerm(Ident(∗)), b=SimpleTerm(Number(∗)) )
        −> < canonify: Plus(b, a)>;
// a ∗ 5 −> 5 ∗ a
Mul( a=SimpleTerm(Ident(∗)), b=SimpleTerm(Number(∗)) )
        −> < canonify: Mul(b, a)>;
// a + (6 + b) −> 6 + (a + b)
Plus( a=SimpleTerm(Ident(∗)), Plus(b=SimpleTerm(Number(∗)), rest) )
        −> < canonify: Plus(b, Plus(a,rest))>;
// a ∗ (6 ∗ b) −> 6 ∗ (a ∗ b)
Mul( a=SimpleTerm(Ident(∗)), Mul(b=SimpleTerm(Number(∗)), rest) )
        −> < canonify: Mul(b, Mul(a,rest))>;

%{ KC_REWRITE
inline integer plus(integer a, integer b){
  return mkinteger(a−>value+b−>value);
}
inline integer minus(integer a, integer b){
  return mkinteger(a−>value−b−>value);
}
inline integer mul(integer a, integer b){
  return mkinteger(a−>value∗b−>value);
}
inline integer div(integer a, integer b){
  return mkinteger(b−>value==0 ? 0 : a−>value / b−>value);
}
%}

%uview infix,postfix;

Plus(a,b)−>[infix: "("  a "+" b ")" ];
Minus(a,b)−>[infix: "(" a "−" b ")" ];
Mul(a,b)−>[infix: "("  a "∗"  b ")" ];
Div(a,b)−>[infix: "("  a "/"  b ")" ];
```

A.4 lexic.l

```
%{
#include <iostream>
#include "k.h"
#include "yystype.h"
#include "syntax.h"
#include "rpn.h"
%}

%option noyywrap

%%

−?[0−9]+    { yylval.yt_integer = mkinteger(atoi(yytext)); return NUMBER;}
[a−z]+      { yylval.yt_casestring = mkcasestring(yytext); return IDENT; }
[+∗−/]      { return yytext[0]; }
[\t ]+      { /∗empty∗/ }
\n          { return EOF; }
.           { std::cerr << "Unkown␣character:␣" <<yytext[0] <<std::endl; }

%%

extern void yyerror(const char ∗s) {
  std::cerr << "Syntax␣error:␣" <<s << std::endl;
}
```

A.5 syntax.y

```
%{
#include "k.h"
#include "yystype.h"

extern void yyerror(const char∗);
extern int yylex();
%}

%token <yt_integer> NUMBER
%token <yt_casestring> IDENT
%token NEWLINE

%type <yt_aritherm> aritherm
```

```
%type <yt_simpleterm> simpleterm

%%


aritherm:
        simpleterm
        { root_term = $$ = SimpleTerm($1); }
        | aritherm aritherm '+'
        { root_term = $$ = Plus($1,$2); }
        | aritherm aritherm '*'
        { root_term = $$ = Mul($1,$2); }
        | aritherm aritherm '–'
        { root_term = $$ = Minus($1,$2); }
        | aritherm aritherm '/'
        { root_term = $$ = Div($1,$2); }
;

simpleterm:
        NUMBER
        { $$ = Number($1); }
        | IDENT
        { $$ = Ident($1); }
;
```

A.6 Makefile

```
# Reverse Polish Notation, Makefile
# c 2001, Michael Piefel <piefel@informatik.hu–berlin.de>

.PRECIOUS: lexic.c y.output

# Tools
SHELL  = /bin/sh
YACC   = bison
LEX    = flex
CC     = ${CXX}
KC     = kc++

#Flages
YFLAGS = –d –y
LFLAGS = –t
CXXFLAGS = –g –Wall –Wno–unused –DYYDEBUG –DYYERROR_VERBOSE
```

```
CFLAGS = ${CXXFLAGS}

# Sources
KFILES = rpn.k main.k abstr.k
YFILES = syntax.y
LFILES = lexic.l


# Goals
PARSER = rpn−parser

# Help files

KC_TIME = .kc_time_stamp

KC_OGEN = k.o csgiok.o unpk.o rk.o
KC_OSRC = $(KFILES:.k=.o)
OBJS   = $(KC_OGEN) $(KC_OSRC) $(YFILES:.y=.o) $(LFILES:.l=.o) $(CFILES:.cc=.o)


# default rule
$(PARSER)::

# include or make autodependencies
ifeq  (. depend,$(wildcard .depend))
include .depend
else
$(PARSER):: depend
endif

# Rules
$(KC_TIME): $(KFILES)
        $(KC) $(KFILES)
        touch $(KC_TIME)

$(PARSER):: $(KC_TIME) $(OBJS)
        $(CXX) $(CFLAGS) $(OBJS) ${LIBS} −o $@


syntax.h : y.tab.h
        −cmp −s syntax.h y.tab.h || cp y.tab.h syntax.h

y.tab.h  : syntax.c

depend dep:
        @echo Make dependencies first
        $(MAKE) $(KC_TIME)
        $(MAKE) $(YFILES:.y=.c)
        $(MAKE) $(LFILES:.l=.c)
        $(MAKE) syntax.h
        $(CC) −M ∗.cc > .depend

clean:
        −rm −f $(OBJS) core ∗~ ∗.bak $(KFILES:.k=.cc) $(KC_OGEN:.o=.cc) $(KC_TIME) .kc∗ \
        $(YFILES:.y=.c) $(LFILES:.l=.c) $(KC_OGEN:.o=.h) $(KFILES:.k=.h) out.∗ \
        y.tab.h syntax.h syntax.output syntax.tab.c yystype.h
```

B GNU Free Documentation License

Version 1.1, March 2000

B.0 Preamble

The purpose of this License is to make a manual, textbook, or other written document 'free' in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of 'copyleft', which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

B.1 Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The 'Document', below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as 'you'.

A 'Modified Version' of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

B.2 Verbatim Copying

copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

B.3 Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

B.4 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

13. Delete any section entitled 'Endorsements'. Such a section may not be included in the Modified Version.

14. Do not retitle any existing section as 'Endorsements' or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled 'Endorsements', provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

B.5 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled 'History' in the various original documents, forming one section entitled 'History'; likewise combine any sections entitled 'Acknowledgements', and any sections entitled 'Dedications'. You must delete all sections entitled 'Endorsements.'

Index