# The Termprocessor *Kimwitu*

Peter van Eijk, Axel Belinfante, Henk Eertink, Henk Alblas

## Overview of this talk:

- *Kimwitu*: what is it?
- example (demo!)
- conclusions
- plans for the future

## *Kimwitu*: what is it?

- A meta-tool, to support construction of
- programs that operate on trees or terms.
- It supports open
- multi-paradigm programming.
- Terms can be implemented in a tool
- as well as exchanged between tools
  which eases tool integration.

multi-paradigm programming: in that it allows to express each part of an implementation in the most appropriate language, e.g. AG, unparsing, rewriting, functional programming style, ...

*Kimwitu*: demo!

build small toolset for language consisting of

- (non-negative) numbers
- plus (+) and times ($*$) operators

with toolfunctions to

- read (*parse*) expressions
- prettyprint (*unparse*) expressions
- simplify (*rewrite*) expressions
- find common sub-expressions

demo: overview

- abstract syntax
- creating terms (and creating *dot* input)
- unparsing
- parsing
- tool environment “architecture”
- rewriting
- finding common sub-expressions
  - *uniq*ue storage option
  - attributes

## abstract syntax:

```
expr:    Const( int )    /* int is predefined */
|    Plus( expr expr )
|    Times( expr expr)
;
```

## term creation (C function def in *kimwitu* input):

```
int main() {                /* build (1+1) * (1+1) */
  expr e = Times(
             Plus( Const( 1 ), Const( 1 )),
             Plus( Const( 1 ), Const( 1 )) );
  fprintdot_expr( stdout, e, "(1+1) * (1+1)",
                  0, 0, True, True, True );
}
```

## usage in shell:

```
$ build_expr | dot -Tps | ghostview -
```

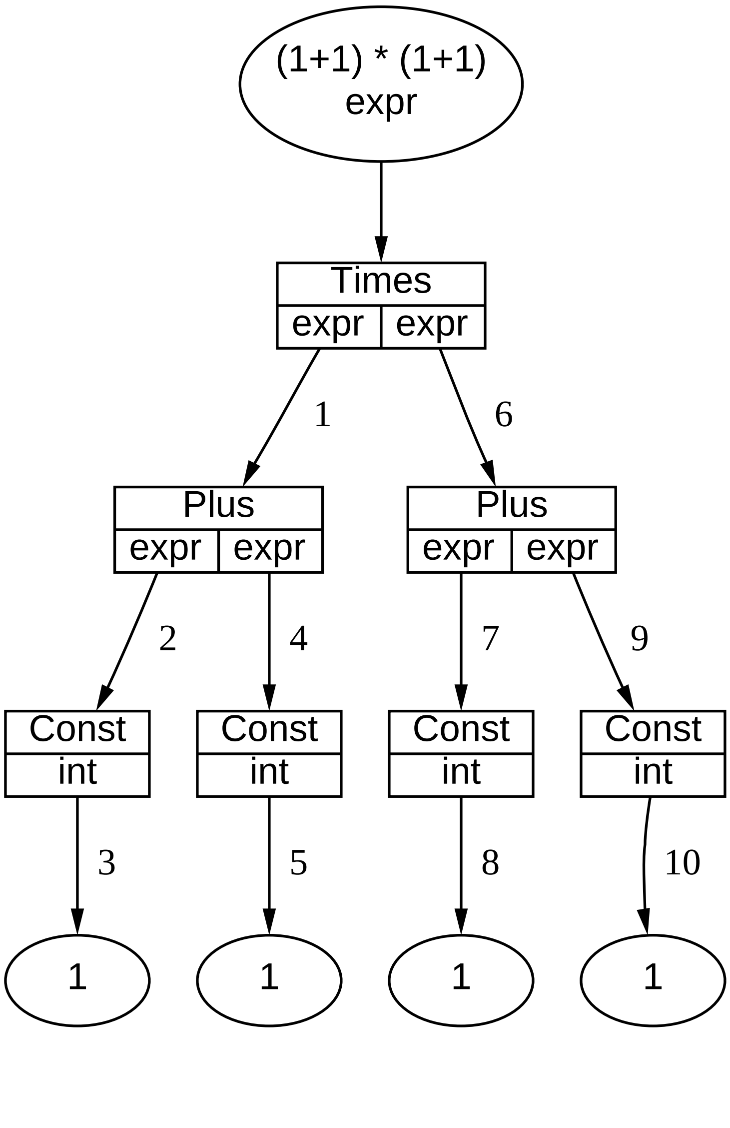
usage in shell:

---

$ build_expr | dot −Tps | ghostview −

---

output:



## unparse rules:

```
Plus( e1, e2 )              -> [:      e1   " + "  e2      ];
Times( e1, e2 )             -> [:      e1   " * "  e2      ];

Times( e1 = Plus(*, *),
       e2               ) -> [: "(" e1 ") * "  e2      ];
Times( e1,
       e2 = Plus(*, *)) -> [:      e1 "  * (" e2 ")" ];
Times( e1 = Plus(*, *),
       e2 = Plus(*, *)) -> [: "(" e1 ") * (" e2 ")" ];
```

## unparser:

```
%{
#include "unpk.h"          /* unparse (view) defs */
#include "printer.h"            /* printer fn def */
%}
int main() {               /* build (1+1+1) * (1) */
  expr e = Times(
     Plus( Plus( Const( 1 ), Const( 1 )), Const( 1 )),
     Const( 1 ) );
  unparse_expr( e, printer, base_uview );
  fprintf( stdout, "\n" );
}
```

unparser: printer function:

```
%{
#include "unpk.h"            /* unparse (view) defs */
%}
void printer( char*s, uview v )
{ fprintf( stdout, "%s", s ); }
```

## parser: yacc input

```
%{
#include "k.h" /* data type defs (incl. YYSTYPE) */
extern expr the_expr;  /* root of the parse tree */
%}

%token <yt_int>   T_INT
%type  <yt_expr>  Y_expr Y_term Y_factor

%%

Y_root:   Y_expr              { the_expr = $1; } ;

Y_expr:   Y_term              { $$ = $1; }
|   Y_expr '+' Y_term         { $$ = Plus( $1, $3 ); }
;

Y_term:   Y_factor            { $$ = $1; }
|   Y_term '*' Y_factor       { $$ = Times( $1, $3 ); }
;

Y_factor:   T_INT             { $$ = Const( $1 ); }
|   '(' Y_expr ')'            { $$ = $2; }
;
```

## parser: lex input

```
%{
#include "k.h" /* data type defs (incl. YYSTYPE) */
#include "y.tab.h"    /* yacc (token) definitions */
%}

%%

[\t\n ]          { ; }          /* skip white space */
[0-9]+            { yylval.yt_int = atoi(yytext);
                    return T_INT; }
.                { return yytext[0]; }
```

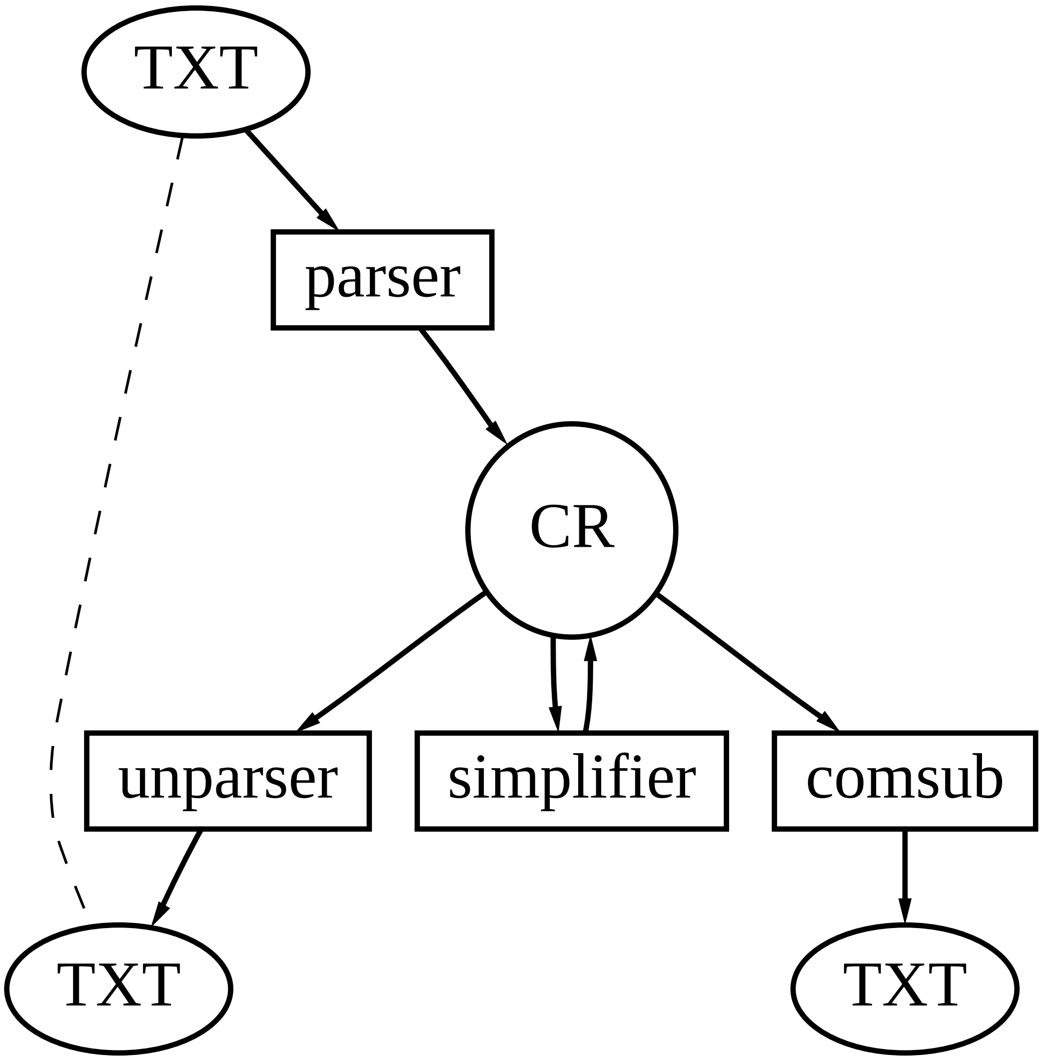
## parser: main program

```
%{
#include "y.tab.h"    /* yacc (token) definitions */
expr the_expr;        /* the root of our parse tree */
%}

int main() {
  yyparse();                          /* invoke parser */
  fprintdot_expr( stdout, the_expr, "the_expr",
                  0, 0, True, True, True );
}

void yyerror(char *s)
{ fprintf(stderr, "%s\n", s); exit(1); }
```

## parser: usage from shell

```
$ echo '(1+1)*(1+1)' | parser | dot -Tps | ghostview -
```

the toolset, structure:



the toolset, usage:

```
$ echo '(1+1)*(1+1)' | parser | unparser
```

toolset parser main routine:

```
int main() {
  char *io;              /* error message, or == 0 */
  yyparse();                          /* invoke parser */
  io = CSGIOwrite_expr(stdout, the_expr);
  if (io != 0) fatal("Write error: %s\n", io);
  exit(0);                                /* success! */
}
```

toolset unparser main routine:

```
int main() {
  char *io;              /* error message, or == 0 */
  expr the_expr;         /* root of the parse tree */
  io = CSGIOread_expr(stdin, &the_expr);
  if (io != 0) fatal("Read error: %s\n", io);
  unparse_expr(the_expr, printer, base_uview);
  exit(0);                                /* success! */
}
```

simplify, usage:

```
$ echo '(1+1)*(1+1)' | parser | simplify | unparser
```

simplify, rewrite rules:

```
Plus( Const( i ), Const( j ))  -> Const( cplus( i, j ));
Times( Const( i ), Const( j )) -> Const( ctimes( i, j ));
```

simplify, main routine:

```
int main() {
  char *io;              /* error message, or == 0 */
  expr the_expr;         /* root of the parse tree */
  io = CSGIOread_expr(stdin, &the_expr);
  if (io != 0) fatal("Read error: %s\n", io);
  the_expr = rewrite_expr(the_expr, base_rview);
  io = CSGIOwrite_expr(stdout, the_expr);
  if (io != 0) fatal("Write error: %s\n", io);
  exit(0);                               /* success! */
}
```

simplify, rewrite rules:

```
Plus( Const( i ), Const( j ))  -> Const( cplus( i, j ));
Times( Const( i ), Const( j )) -> Const( ctimes( i, j ));
```

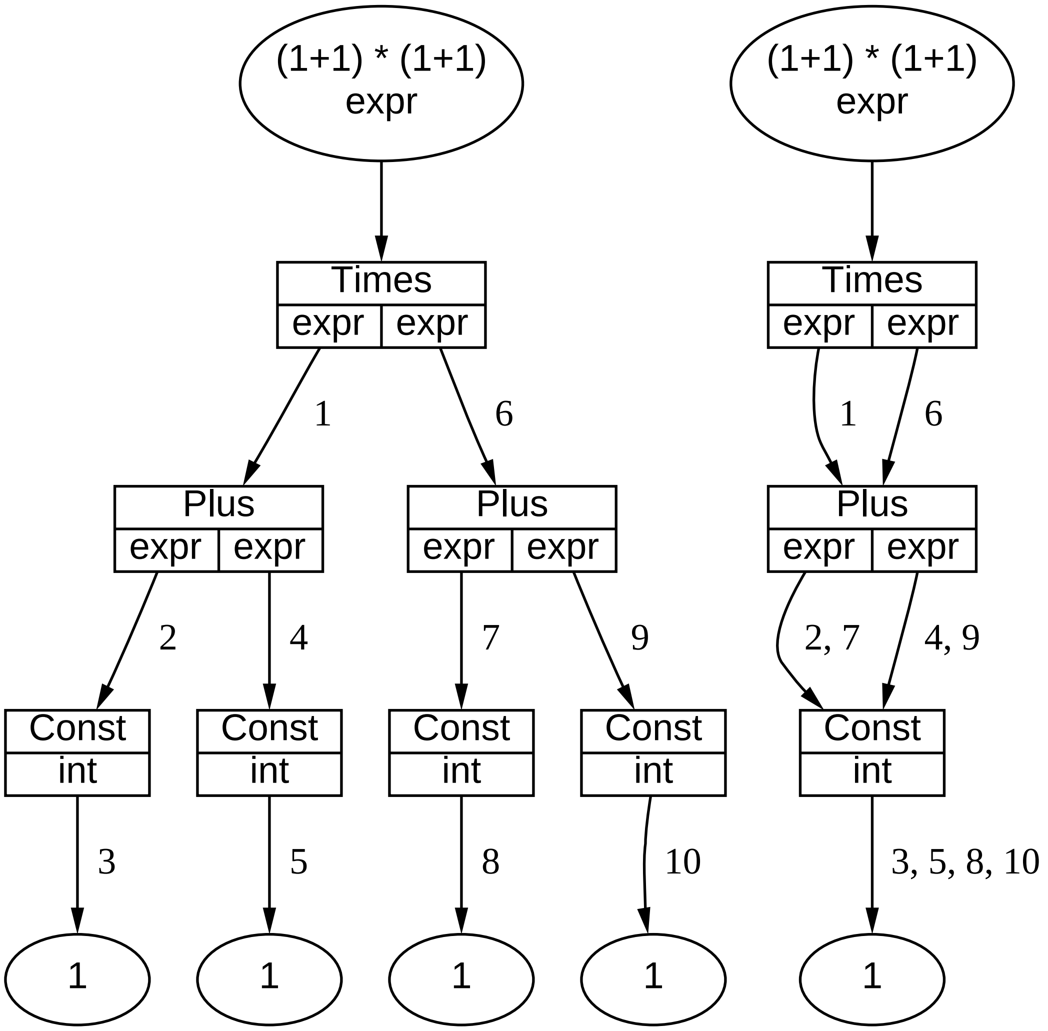
simplify, cplus and ctime routines:

```
%{ KC_REWRITE  /* export fn defs to rewrite code */
#include "compute.h"
%}

int cplus( int i, int j ) {
  return i + j;
}
int ctimes( int i, int j ) {
  return i * j;
}
```

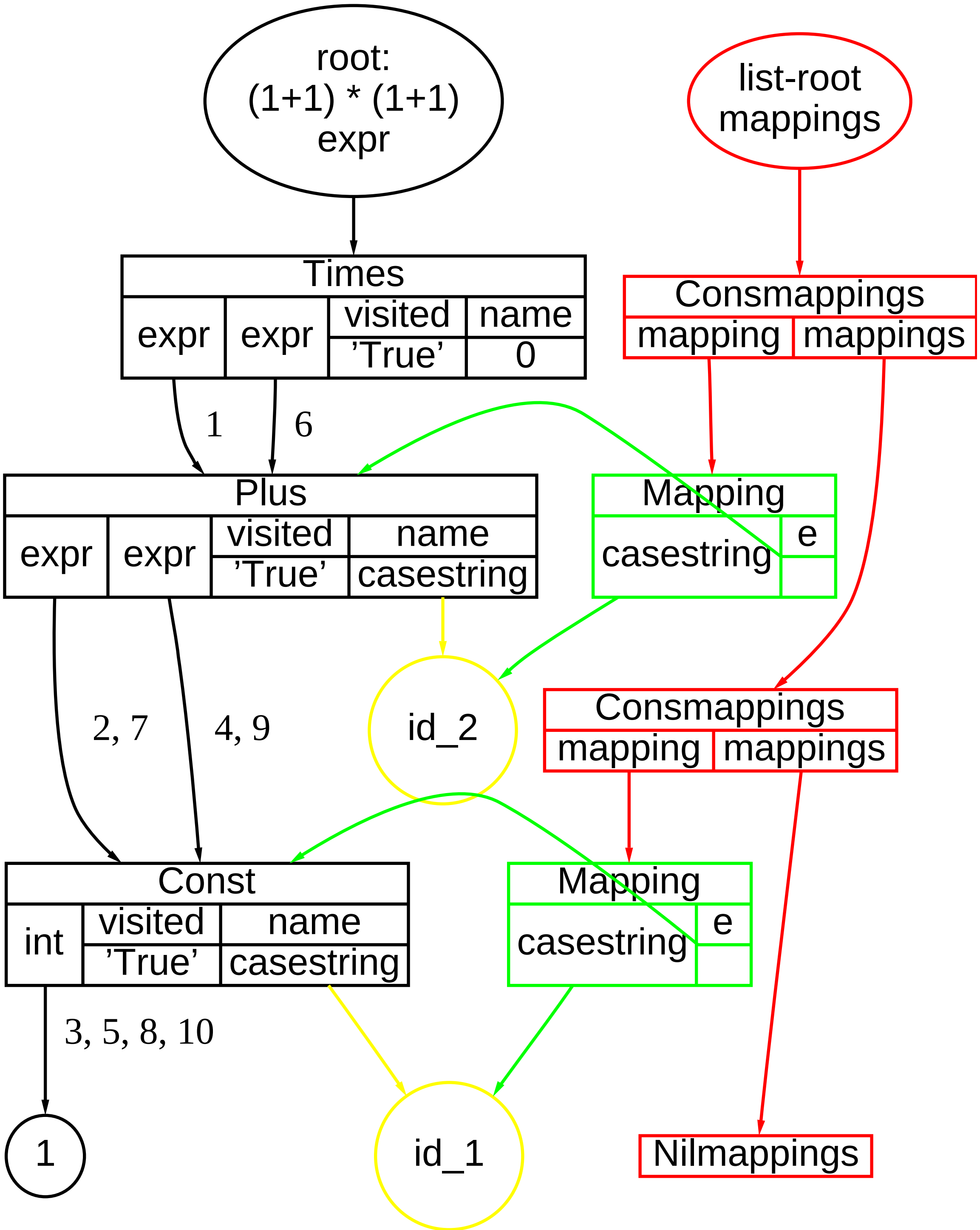
## intermission: the *uniq* storage option

expr {**uniq**} :;

## result (left non-uniq, right uniq):



# common sub-expressions finding using *uniq*



## common sub-expressions: attribute definitions

```
expr {uniq} :
{ boolean visited = False;   /* initial value */
  casestring name = 0;   /* predef case sensitive string */
};

/* mapping ('assoc array'): casestring -> expr */
mapping {uniq} : Mapping( casestring )
{ expr e = 0; };

mappings: list mapping;
```

## common sub-expressions: tree-walk

```
Plus( e1, e2 ),
Times( e1, e2 )        -> [ view_visit:
  { visit($0); }      /* $0 refers to current node */
  e1
  e2
];

Const( i )             -> [ view_visit:
  { visit($0); }      /* $0 refers to current node */
];
```

# common sub-expressions: utility functions

```
%{
#include "toolset_comsub.h"
%}
void visit (expr n)
{ if ( ! n->visited ) {   /* node not yet visited */
    n->visited = True;          /* mark as visited */
  } else {              /* visited this node before */
    if ( n->name == 0 ) {       /* not yet a name */
      mapping m;
      n->name = gen_name(); /* generate new name */
      m = Mapping( n->name ); /* and new mapping */
      m->e = n;           /* map from name to node */
      /* insert mapping in list of all mappings: */
      all_mappings = Consmappings( m, all_mappings );
} } }

casestring gen_name()
{ static int i = 1;
  char tmp[50];
  sprintf(tmp, "id_%d", i++);
  return mkcasestring( tmp );
}
```

## common sub-expressions: main routine

```
void main() {
  char *io;              /* error message, or == 0 */
  expr the_expr;         /* root of the parse tree */
  all_mappings = Nilmappings();
  io = CSGIOread_expr(stdin, &the_expr);
  if (io != 0) fatal("Read error: %s\n", io);

  unparse_expr(the_expr, printer, view_visit);

  unparse_expr(the_expr, printer, test);
  unparse_mappings(all_mappings, printer, base_uview);
  printer("\n", base_uview);

  exit(0);                             /* success! */
}
```

## common sub-expressions: unparsing

```
Consmappings( h, t )     -> [: ";\n"  h t ];
Mapping( s )             -> [: s  " = "  $0->e ];

Plus( *, * ),  Times( *, * ),
Const( * )               -> [ test:
  { if ($0->name != 0) }  $0->name:base_uview
  { else }                $0:base_uview
];

Plus( e1, e2 )           -> [: e1:test   " + "  e2:test  ];
Times( e1, e2 )          -> [: e1:test   " * "  e2:test  ];

Times( e1 = Plus(*, *),
       e2              ) -> [: e1:test2  " * "  e2:test  ];
Times( e1,
       e2 = Plus(*, *)) -> [: e1:test   " * "  e2:test2 ];
Times( e1 = Plus(*, *),
       e2 = Plus(*, *)) -> [: e1:test2  " * "  e2:test2 ];

Plus( *, * )             -> [ test2:
  { if ($0->name != 0) }  $0->name:base_uview
  { else }                ${ "(" $0:base_uview ")" $}
];
```

demo: we have seen:

- data structure definitions
- unparse rules
- cooperation with yacc and lex
- function definitions
- rewrite rules
- use of structure-file io
- some use of attributes
- some use of *uniq* storage option

demo: we have *not* seen:

- pattern matching in function defs
- structure-file io interface with SG editors
- use of *uniq* for memo functions
- use of *uniq* for symbol tables
- open-ness of generated code
- use of phylum initialization code

conclusions: *kimwitu* offers

- construction of programs that operate on trees or terms
- open
  (cooperation with yacc and lex, use of C functions in rewrite- and unparse rules)
- multi-paradigm programming
  (attribute grammars, unparse-, rewrite rules, functional programming)
- tool integration
- short development time
- generation of production quality programs

*Kimwitu*'s future

- cooperate with CORBA (IDL)
- generate C++ (or Java?)
- AG evaluator