Contents

1 Introduction

XD is a procedural programming language with rule abstraction, a generalized iteration mechanism and lexical closures. It is a proper superset of its host language, and the most natural way to implement it is to write a preprocessor on top of a C or C++ compiler. XD is designed to be used for rule-based programming, when more sophisticated alternatives are not available or when they are not efficient enough. In the following we shall, for simplicity, assume the host language to be C.

2 XD Constructions

XD inherits all the lexical conventions of its parent language, and syntactically it is a proper superset of C, in the sense that every correct C program is also an XD program. The main criterion in the design of the syntax of XD was the simplicity of its implementation as a preprocessor.

2.1 Iterator

Iterators are a generalization of the usual iteration constructs of conventional procedural languages. In Pascal or Ada one can iterate over a range of integers or over the members of an enumeration type. XD generalizes these constructs in the same way as CLU [5] and XE [3].

An iterator is a special kind of procedure, which is executed as a coroutine yielding a sequence of values. It is called within a **foreach** statement. The definition of an iterator is identical to the definition of a function except for the reserved word **iterator**, and **yield** statements within the body:

> **iterator** *type name* (*parameters*) *declarations body*

The type of an iterator is the type of the values it yields. A **yield** statement consists of the reserved word **yield** and an expression, just like the **return** statement.

After yielding a value, the iterator passes control to the **foreach** statement executing it. When the body of the **foreach** statement has finished, the iterator execution is resumed just after the **yield** statement. The local variables of the iterator remain unchanged between yield and resume (of course, indirect assignments via pointers may change them). The execution of an iterator terminates when a **return** statement without a return value is executed, or when the end of the iterator body is reached. An iterator body may not contain a **return** statement returning a value. The following is an example of an iterator yielding a range of integers.

```
iterator int from_to(from,to)
int from,to; {
  while(from <= to) {
    yield(from);
    from ++;
  }
}
```

2.2 Foreach

Foreach is a simple iteration statement for iterator execution. It invokes an iterator, and for each value yielded by the iterator, the body of the **foreach** statement is executed. While executing the body, the variable following the word **foreach** is bound to the last value yielded by the iterator. The syntax of the **foreach** statement is

foreach *id* **in** *primary* (*expressions*) *statement*

Here *id* is a variable defined before the **foreach** statement and it is of the same type as the iterator called; *primary* references the iterator. **Foreach** statements may be nested.

An example is a **foreach** statement, which calls the iterator of the previous example.

```
foreach i in from_to(1,10) {
    printf("Iterator yielded %d",i);
}
```

2.3 Rule

A rule in XD is a special kind of an iterator implementing a condition–action abstraction. Rules are reducible to ordinary iterators, but they are more restricted and syntactically more comprehensible than corresponding ordinary iterators. The syntax and semantics of rules are as far as possible analogous to rules in XE [1, 2, 3].

The condition part of a rule consists of **foreach** statements producing a sequence of values and a clause of universal and existential quantifications over a set of values produced by an iterator, connectives **cand** and **cor**, the negation operator **not** and truth-valued expressions. When the clause is true for a value, the action statements are executed. The syntax of a rule is:

rule *type name* (*parameters*) *declarations rule-body*

rule-body $\equiv$ { *declarations* **when** *condition* : *statements* }

condition $\equiv$ **foreach** *id* **in** *primary* (*expressions*) *condition*
condition $\equiv$ *clause*

clause $\equiv$ *clause* **cand** *clause*
clause $\equiv$ *clause* **cor** *clause*
clause $\equiv$ **not** *clause*
clause $\equiv$ (*clause*)
clause $\equiv$ **some** *declaration* **in** *primary* (*expressions*) *clause*
clause $\equiv$ **all** *declaration* **in** *primary* (*expressions*) *clause*
clause $\equiv$ *expression*

The variables of the **foreach** statements and the local variables of a rule must be defined in the declarations part of the rule body, and they are also visible in the action statements.

The *declaration* in a **some** or an **all** quantification is a variable declaration local to the quantification. The variable is bound to the values the iterator yields, and it is visible in the clause part of the quantification.

Some invokes the iterator, and its value is true, if for one or more of the values the iterator yielded, the quantified clause was true. Otherwise the value is false. The value of an **all** clause is false, if for one or more of the values the iterator yielded, the quantified clause was false. Otherwise the value is true. The operators **cand**, **cor** and **not** are evaluated as the C operators &&, || and !, respectively.

The following example is from the XD version of the well-known monkey and banana program.

```
rule m_inst holds_nil(m, goals, satisfied)
monkey *m; goal_list *goals, *satisfied; {
  goal *g;
  when
  foreach g in goal_elements(goals, holds)
    g->obj == NULL cand m->holds != NULL
:
  yield(procedure void() binds monkey *m, goal *g; {
    move(goals,satisfied,g);
    printf("Drop %s",m->holds->name);
    drop(m);
  });
}
```

2.4 Itertype

Itertype is a type specifier, which specifies iterators in the following way:

> **itertype** *typespec* (*typespec-list*)

Here *typespec* is the type of the values the iterator yields, and *typespec-list* is a list of comma-separated parameter type specifiers. The following example defines a variable *ft*, which is a pointer to an iterator, which has two integer parameters and yields character pointers.

> **itertype** char *(int,int) ft;

2.5 Procedure

XD extends C with lexical closures. Closure expressions are defined by using the reserved word **procedure**. Closure types are declared by using the reserved word **closure** (see Section 2.7). A lexical closure is a functional object with local variables. When creating a closure, the closure variables are bound by the **binds** clause to values of identically named variables. The result of a procedure expression is a pointer to a lexical closure. Syntactically procedure expressions are identical to unnamed function definitions, except for the added **binds** clause:

> **procedure** *typespec* (*parameters*) **binds** *closure-variable-declarations*;
> *declarations compound-statement*

The non-terminal *closure-variable-declarations* represents a comma-separated, possibly empty, list of closure variable declarations, e.g.

> int a, int b, char *c

The declarations must be identical to the declarations of the identically named variables outside the procedure.

Note, that a procedure expression allocates memory dynamically via *malloc*, and it is the user's responsibility to free this storage when it is not needed anymore. The example procedure expression below creates a closure, which returns ascending numbers on consecutive calls.

```
closure int() numbers;
int begin_from = 7;
numbers = procedure int() binds int begin_from; {
    return begin_from++;
}
```

2.6 Execute

The only way to invoke lexical closures is the **execute** expression. The result of an **execute** expression is the result of the lexical closure called. The syntax of the expression is

> **execute** *primary* (*expressions*)

The *primary* references a closure. An example is a pair of calls of the closure created in the previous example. When executed, the first statement outputs the number 7, and the second statement outputs the number 8.

```
printf("The next number is %d",execute numbers());
printf("The next number is %d",execute numbers());
```

2.7 Closure

Pointers referencing a closure may be declared using the type specifier **closure**. The syntax is

> **closure** *typespec* (*typespec-list*)

The *typespec* is the result type of the closure, and the *typespec-list* is a list of parameter type specifiers. Our example specifies a type name *returns_char* of closures returning one character.

```
typedef closure char() returns_char;
```

2.8 Module

XD provides a way to preprocess a portion of a XD source file to a separate target file. This may be used to restrict the visibility of global names. The construct is of form:

> **module** *name* { *body* }

Here *name* is an XD identifier used as the output file name, and *body* is any proper sequence of XD definitions and declarations. Modules may be nested. An example:

```
module Sub_Module {
  int f(n) int n; {
    return n+1;
  }
  ...
}
```

3 Syntax of XD++

XD++ is the C++ version of XD. Most of the XD constructs remain identical in XD++, but some differences remain due to the dissimilar function definitions of C++ and C. These differences affect the syntax of iterators, rules and procedure expressions.

Detailed descriptions of iterators, rules and procedure expressions are in Sections 2.1, 2.3 and 2.5, respectively. For iterators, rules and procedures, the declaration of the parameters conforms to that of usual function definitions of C++, i.e., they appear inside the parentheses. Thus, in XD++ the syntax of iterators, rules and procedure expressions are respectively

> **iterator** *type name* (*parameter-declarations*) *body*
>
> **rule** *type name* (*parameter-declarations*) *rule-body*

and

> **procedure** *typespec* (*parameter-declarations*)
> **binds** *closure-variable-declarations* ; *compound-statement*.

The non-terminal *parameter-declarations* is a list of parameter type declarations and *closure-variable-declarations* is a list of closure variable declarations. Both lists are separated by commas, e.g.

> monkey *m, goal_list *goals, goal_list *satisfied

In XD++, rule and iterator definitions may be overloaded. Iterators and rules can be members and friends of classes. For example, a class implementing linked integer lists could be defined, with an iterator *members*, which iterates over the members of a list:

```
struct int_node {
  int element;
  int_node *next;
}

class linked_list {
  int_node *list;
public:
  itertype int members();
  ...
}
```

```
iterator int linked_list :: members() {
   int_node *n = list;
   while(n != NULL) {
      yield n->element;
      n = n->next;
   }
}
...
```

4 Invoking the XD preprocessor

The XD preprocessor is invoked by writing **xd** *source target* or **xd** < *source* > *target*, where the *source* is the name of the source file, and the *target* is the desired name of the target file. The XD preprocessor first invokes the standard C preprocessor, and then it parses the whole input file and translates the XD constructs to ordinary C. The name of the XD++ preprocessor is **XD**, otherwise it works in the same way.

5 Implementation of XD

XD is implemented as a preprocessor reading XD source and producing C source. XD extensions can be reduced to ordinary C, except for iterators and rules, which need three small assembler routines for coroutining. These routines are used to jump into and out of an active iterator without creating or removing the stack frame. Currently, we have these routines for Microsoft C 5.1 with Intel 8086, and Gnu-C with Motorola 68020 (Sun–3).

Currently, there are some restrictions due to the implementation as a preprocessor. None of these restrictions is severe.

1. The *primary* in an **execute** statement may not be a **procedure** expression.
2. A **procedure** expression may not appear before a corresponding **closure** declaration.

5.1 The Parser

Some of the XD extensions are rather complex, and therefore the XD source has to be parsed. For this purpose, we use the yacc parser of the Gnu-C [6] extended with productions for XD constructs. The action parts of grammar productions usually construct a tree node with the results of the left hand side non-terminal and terminal symbols as subtrees. We cannot directly output a token to the output file because of the extensive reordering of source code fragments required. Some context dependent information is also recorded in the nodes of the tree. The action parts of the grammar productions for XD extensions create a temporary character buffer, where they output the C source, called replacement below, corresponding to an XD construct. The temporary buffers are returned as the result of the grammar productions, and they usually become nodes in the tree.

Some XD constructs produce in addition to the C language replacement some top level type and function definitions. These definitions are directly appended to the target file, while other definitions are attached to the tree described above. The action parts of the grammar productions of the top level definitions and declarations output and free their constituent trees.

5.2 Compilation of XD Constructs

In this section we describe the C replacements of the XD constructs, and the generation of these replacements in the action parts of the grammar productions for XD constructs.

The replacements for iterators and **foreach** statements contain calls to three small assembler routines for coroutining. The routines communicate through a small activation table, which is allocated locally from the stack. All other constructs reduce to ordinary expressions and statements.

5.2.1 Iterator

An iterator is compiled to a usual C function with some modifications. The values the iterator yields, are assigned to the iteration variable of the calling **foreach** statement via a pointer, which was passed to the iterator as a parameter. A pointer to the activation table used is also passed as a parameter. Hence, the C function has two more parameters. The type of the C function is *int*, because an integer flag, 0 or 1, is returned to the **foreach** statement when yielding or terminating. The value is 0 when just yielding, and 1 when the iterator has no more values to yield. The assembler routine *_XD_create* initializes the activation table and the routine *_XD_jumpout* jumps to the calling **foreach** statement. A more detailed description of these routines is in section 5.3.

> **iterator** *type id* (*parameters*) *declarations* {
>   *statements*1
>   **yield**(*expression*);
>   *statements*2
>   **return**;
>   *statements*3
> }

compiles to

```
int id(parameters, _XD_result, _XD_actx)
declarations type *_XD_result; _XD_activation _XD_actx;
{
  _XD_create(_XD_actx);
  {
  statements1
  { *_XD_result = expression;
  _XD_jumpout(_XD_actx); }
  statements2
  return(1);
  statements3
  }
  return(1);
}
```

5.2.2 Foreach

A **foreach** statement is replaced with a simple compound statement. The activation table is allocated in the declarations part of the statement. A status variable for the iterator is defined and initialized. When initializing the status variable, the actual call and initialization of the iterator also takes place. The **if** statement checks, whether the iterator has terminated, and if not, the body of the **foreach** statement is executed, and the iterator execution is resumed by calling the assembler routine *_XD_jumpin*.

foreach *id* **in** *primary*(*expressions*) *body*

compiles to

```
{ _XD_activation _XD_act;
  int _XD_terminated = primary (expressions, &id, _XD_act);
  if(!_XD_terminated) {
    body
    _XD_jumpin(_XD_act);
  }
}
```

5.2.3 Rule

The compilation of a rule is identical to the compilation of an iterator, except that the body of a rule is a condition–action pair. Here we describe the compilation of the body, and regarding the parameters and return values of a rule we refer the Section 5.2.1, which describes the compilation of iterators.

The syntax of the rule body is rather complex, and we shall describe the grammar rules of Section 2.3 one at a time.

1. **rule** *type name* (*parameters*) *declarations rule-body*

   is compiled to a function:

   > int *id*(*parameters*, _XD_result, _XD_act)
   > *declarations type* *_XD_result; _XD_activation _XD_act;
   > {
   >   *compiled-rule-body*
   >   **return**(1);
   > }

2. *rule-body* $\equiv$ { *declarations* **when** *condition* : *statements* }

   compiles to a compound statement, which declares an auxiliary variable for evaluation of the clause part of the condition.

   > {
   >   int _CLAUSE_result;
   >   *declarations*
   >   *compiled_condition_and_statements*
   > }

3. *condition* $\equiv$ **foreach** *id* **in** *primary* (*expressions*) *condition*

   is compiled in the same way as **foreach** statements.

   > {
   >   _XD_activation _XD_act;
   >   int _XD_terminated = *primary* (*expressions*, &*id*, _XD_act);
   >   **if**(!_XD_terminated) {
   >   *compiled_condition*
   >   _XD_jumpin(_XD_act);
   >   }
   > }

4. *condition* $\equiv$ *clause*

The innermost condition, i.e. the topmost clause, is compiled to a compound statement, which first computes the result of the clause and saves it to the variable *_CLAUSE_result*, and then executes the statements, if the clause was true.

{
  compiled_clause
  if(_CLAUSE_result) *compiled-statements*
}

5. *clause* $\equiv$ *clause*1 **cand** *clause*2

If the first clause of a conjunction is evaluated to false, the result is false, otherwise the result is the value of the second clause. The n of the label is a previously unused label number.

compiled_clause1
if(!_CLAUSE_result) **goto** label$n$;
compiled_clause2
label$n$:

6. *clause* $\equiv$ *clause*1 **cor** *clause*2

If the first clause of a disjunction evaluates to true, the result is true, otherwise the result is the value of the second clause.

compiled_clause1
if(_CLAUSE_result) **goto** label$n$;
compiled_clause2
label$n$:

7. *clause* $\equiv$ **not** *clause*

For a negation, the clause is evaluated, and the resulting value is negated.

compiled_clause
_CLAUSE_result = !_CLAUSE_result;

8. *clause* $\equiv$ (*clause*)

The parantheses are needed just for parsing, and the clause is evaluated in the usual way.

compiled_clause

9. *clause* $\equiv$ **some** *declaration* **in** *primary* (*expressions*) *clause*

A **some** quantification is compiled to a compound statement, which defines local variables for iterator execution and a flag. The flag is initially set to false, and when the clause of the quantification is true for a value the iterator yielded, the flag is set to true. The value of a quantification is the value of the flag when the iterator terminates. The terminal *declaration-id* is the identifier of the declaration in the quantification.

```
{
  _XD_activation _XD_act;
  int _QUANTIFICATION_flag = 0;
  declaration;
  int _XD_terminated = primary ( expressions, &declaration-id,
  _XD_act);
  if(!_XD_terminated) {
    compiled_clause
    if(_CLAUSE_result) _QUANTIFICATION_flag = 1;
    _XD_jumpin(_XD_act);
  }
  _CLAUSE_result = _QUANTIFICATION_flag;
}
```

10. *clause* $\equiv$ **all** *declaration* **in** *primary* (*expressions*) *clause*

An **all** quantification is compiled to a compound statement, which defines local variables for iterator execution and a flag. The flag is initially set to true, and when the clause of the quantification is false for a value the iterator yielded, the flag is set to false. The value of a quantification is the value of the flag when the iterator terminates. The terminal *declaration-id* is the identifier of the declaration in the quantification.

```
{
   _XD_activation _XD_act;
   int _QUANTIFICATION_flag = 1;
   declaration;
   int _XD_terminated = primary ( expressions, &declaration-id,
   _XD_act);
   if(!_XD_terminated) {
      compiled_clause
      if(!_CLAUSE_result) _QUANTIFICATION_flag = 0;
      _XD_jumpin(_XD_act);
   }
   _CLAUSE_result = _QUANTIFICATION_flag;
}
```

11. *clause* ≡ *expression*

is compiled to an assignment statement, which assigns the value of the expression to the auxiliary variable *_CLAUSE_result*.

```
_CLAUSE_result = expression;
```

5.2.4 Itertype

itertype *type*(*parameter-types*)

generates a global type definition of form

typedef int (*_XD_IT*n*)(*parameter-types*, *type**, _XD_activation);

and the original **itertype** expression is replaced with _XD_IT*n*, where *n* is the number of the type definition.

5.2.5 Procedure

The compilation of a **procedure** is perhaps the most complicated part of XD, because so many definitions and declarations are involved. A type definition for the closure created for a **procedure** expression is a structure with fields for closure variables and the function part of the closure. For the initialization of the fields, a global auxiliary variable is required. A function definition is also produced. The closure structure itself is passed as one parameter, so that the function can access the closure variables.

```
procedure type (parameters) binds closure-variable-declarations ;
declarations {
  ...
  id /* Closure variable */
  ...
}
```

generates a global type definition for the closure structure, a global auxiliary variable for closure initialization, and the actual closure function:

```
typedef struct {
  type (*_XD_f)(parameters, closure_argtypes *);
  closure-variable-declarations
} _XD_proctype_n;

_XD_proctype_n *_XD_procauxvar_n;

static
type _XD_proc_n(parameters, _XD_closure) declarations
closure_argtypes _XD_closure;
{
  ...
  ((_XD_proctype_n *)_XD_closure)-> id) /* closure variable */
  ...
}
```

The replacement for the **procedure** expression is an expression list, which allocates a closure structure, and initializes the fields for the closure function and the closure variables, and finally returns a pointer to the closure.

```
(_XD_procauxvar_n (_XD_proctype_n *) =
  malloc(sizeof(_XD_proctype_n)),
_XD_procauxvar_n->_XD_f = _XD_proc_n,
_XD_procauxvar_n-> id = id,
_XD_procauxvar_n)
```

5.2.6 Execute

The *primary* of an **execute** statement references a closure structure, which has as one field a pointer to the function of the closure.

The XD expression

execute *primary*(*expressions*);

is compiled to

(*primary*->_XD_f)(*expressions*, *primary*);

5.2.7 Closure

A closure type specifier is compiled to a structure type definition, and the type specifier is replaced with a type specifier referencing the structure.

closure *typespec* (*typespec-list*)

is compiled to

#ifndef DEF*closure_argtypes*
#define DEF*closure_argtypes*
typedef struct _*closure_argtypes* {
  typespec (*_XD_f)(*typespec-list*, **struct** _*closure_argtypes* *);
  char *_XD_fields;
} * *closure_argtypes*;
#endif

The *closure_argtype* is an identifier, which is formed from the type names of the *typespec-list*. The purpose of the *closure_argtype* is to support type checking of closure pointers.

5.3 The Assembler Routines

An efficient implementation of iterator and rule execution is impossible without small assembler routines violating the normal function call convention of C programs. The stack frames of the iterator and the caller of the iterator must be preserved during the execution of the **foreach** statement, i.e., when switching from the caller to the callee and back, the frame pointer should be correctly adjusted.

In the current implementation for the Intel 8086 the iterator execution proceeds in the following way. In the beginning of the **foreach** statement the caller uses the usual C function call to initialize the stack frame of the iterator. The first thing done in the iterator, is to call the assembler routine *_XD_create*, which saves the frame pointer (BP) values of the caller and callee to the table of type *_XD_activation*. Then, the iterator body is executed, until the expression of the **yield** statement has been computed and assigned to the variable of the **foreach** statement. To jump to the caller, the iterator calls the routine *_XD_jumpout*, which restores the frame pointer, saves the address of the instruction just after the **yield** statement to the activation table, loads a zero return value to the AX register, and finally jumps to the return address found in the stack frame. When the body of the **foreach** statement has been executed and a new value is required, the routine *_XD_jumpin* is called. It restores the frame pointer and jumps to the address found in the activation table. When terminating, the iterator uses the usual return with a non-zero value, i.e. loads a non-zero value to the AX register.

5.4 The Cost of Executing XD Constructs

A procedure expression allocates the closure structure using the library function *malloc*, and initializes the fields. The closure structure is usually small, from ten to twenty bytes, but the time spend inside *malloc* is a possible source of overhead.

The execution of a **foreach** statement involves one assembler routine call when initializing the iterator, and two calls for each time the iterator yields. These calls could be eliminated, if the implementation was based on a compiler. The execution of a **yield** statement takes 33 μs, excluding the time spent on computing the value of the expression. This is quite acceptable when comparing to a function call, 22 μs.

The compilation result of the condition part of a rule is not a very efficient piece of C. The timings of usual **foreach** statements (described above) apply to **foreach** statements and **some** and **all** clauses of rules. In the condition part, **some** and **all** are used as expressions. Due to their statement character and the lack of possibility for statements to return values, the condition part cannot be compiled to a conventional expression, but to a sequence of more inefficient conditional statements.

The cost of other XD constructs is insignificant.

6 An Example Program

The example below is a fragment extracted from a small file name expert system, which was originally written in XE, and later easily translated to XD. The first definitions define the type of a rule and the type of the rule set. Then the functions initializing and accessing the rule set are defined. The three rules of this example are typical rules from the original rule set. The *main* function initializes the rule set, and contains a simple inference engine.

```
/*
* File system expert rules (A fragment of a complete XD program)
*/

# include <stdio.h>
# define FALSE 0
# define TRUE 1

typedef itertype char *(char *)ruletype;

typedef struct {
  int size;
  ruletype *r;
} ruleset;

iterator ruletype iterate_rules(set) ruleset *set; {
  int i;
  for(i=0;i<set->size;i++) yield(set->r[i]);
}

rule char *rule_0001(fname) char *fname; {
  when strcmp(basename(fname),"a.out") == 0:
  yield("Unnamed executable file – removable");
}

rule char *rule_0002(fname) char *fname; {
  when strcmp(s_part(fname,0,5),"/tmp/") == 0 cor
    strcmp(s_part(fname,0,9),"/usr/tmp/") == 0:
  yield("File in temporary storage – removable");
}

/* ... */

rule char *rule_0011(fname) char *fname; {
  when strcmp(s_part(fname,0,5),"/tmp/") != 0 cand
    strcmp(s_part(fname,0,9),"/usr/tmp/") != 0 cand
      some char *component in filename_components(fname)
        (strcmp(component,"tmp") == 0 cor
        strcmp(component,"temp") == 0):
      yield("Temporary storage – removable");
}

/* ... */

ruletype our_ruleset[] = {
```

```
  rule_0001,
  rule_0002,
  rule_0003,
  rule_0004,
  rule_0005,
  rule_0006,
  rule_0007,
  rule_0008,
  rule_0009,
  rule_0010,
  rule_0011,
  rule_0012,
  rule_0013,
  rule_0014,
  rule_0015,
  rule_0016,
  rule_0017,
  rule_0018,
  rule_0019,
  rule_0020,
  rule_0021
};

ruleset *ruleset_create() {
  ruleset *new_ruleset = (ruleset *) malloc(sizeof(ruleset));
  new_ruleset->size = 22;
  new_ruleset->r = our_ruleset;
  return new_ruleset;
}

main() {
  FILE *po = stdout;
  FILE *pi = stdin;
  int quitp = FALSE;
  char *fname = (char *) malloc(60);

  fprintf(po,"File system expert - analyses file names");
  while(!quitp) {
    fprintf(po,"File name: ");
    fname = fgets(fname,60,pi);
    if(strcmp(fname,"") == 0) quitp = TRUE;
```

```
      else {
        ruleset *rules = ruleset_create();
        int count = 0;
        ruletype r;
        char *s;
        foreach r in iterate_rules(rules) { /* inference engine */
          fprintf(po,"Trying rule");
          foreach s in r(fname) {
            count ++;
            fprintf(po,"Conclusion %d:%s",
              count, s);
          }
        }
      }
    }
  }
```

References

[1] J. Arkko, V. Hirvisalo, J. Kuusela, E. Nuutila and M. Tamminen, *Some Experiences with Rules in Procedural Programming Languages*, Proc. International Computer Science Conference '88, Hong Kong, December 19-21 1988, pp. 712-718.

[2] J. Arkko, V. Hirvisalo, J. Kuusela, E. Nuutila and M. Tamminen, *Programming in XE*, Helsinki University of Technology, Report HTKK-TKO-C33, 1989.

[3] J. Arkko, V. Hirvisalo, J. Kuusela, E. Nuutila and M. Tamminen, *XE Reference Manual*, Helsinki University of Technology, Report HTKK-TKO-A26, 1989.

[4] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[5] B. Liskov et al., *CLU Reference Manual*, Springer-Verlag, 1981.

[6] R. M. Stallman et al., *GNU CC Documentation*, 1986-1988.

[7] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.