Lint, a C Program Checker

Frans Kunst

Vrije Universiteit
Amsterdam

Afstudeer verslag
18 mei 1988

Lint, a C Program Checker

Frans Kunst

Vrije Universiteit
Amsterdam

This document describes an implementation of a program which does an extensive consistency and plausibility check on a set of C program files.
This may lead to warnings which help the programmer to debug the program, to remove useless code and to improve his style.
The program has been used to test itself and has found bugs in sources of some heavily used code.

Contents

1. Introduction

C [1][2] is a dangerous programming language. The programmer is allowed to do almost anything, as long as the syntax of the program is correct. This has a reason. In this way it is possible to make a fast compiler which produces fast code. The compiler will be fast because it doesn't do much checking at compile time. The code is fast because the compiler doesn't generate run time checks. The programmer should protect himself against producing error prone code. One way to do that is to obey the *Ten Commandments for C programmers* [appendix B]. This document describes an implementation of the *lint* program, as referred to in Commandment 1. It is a common error to run *lint* only after a few hours of debugging and some bug can't be found. *Lint* should be run when large pieces of new code are accepted by the compiler and as soon as bugs arise. Even for working programs it is useful to run *lint,* because it can find constructions that may lead to problems in the future.

2. Outline of the program

The program can be divided into three parts. A first pass, which parses C program files and outputs definitions, a second pass which processes the definitions and a driver, which feeds the set of files to the first pass and directs its output to the second pass. Both passes produce the warnings on standard error output, which are redirected to standard output by the driver.

The first pass is based on an existing C front end, called *cem* [3]. *Cem* is part of the Amsterdam Compiler Kit (ACK), as described in [4].

Most of the code of *cem* is left unchanged. This has several reasons. A lot of work, which is done by *cem* , must also be done by *lint.* E.g. the lexical analysis, the macro expansions, the parsing part and the semantical analysis. Only the code generation part is turned off. An advantage of this approach is, that a person who understands *cem* will not have to spend to much time in understanding *lint.*

All changes and extensions to *cem* can be turned off by not defining the compiler directive `LINT.` Compiling should then result in the original C compiler.

The second pass is a much less complex program. It reads simple definitions generated by the first pass and checks their consistency. This second pass gives warnings about wrong usage of function arguments, their results and about external variables, which are used and defined in more than one file.

The driver is a shell program, to be executed by the UNIX® shell *sh.* It executes the two passes and let them communicate through a filter (sort). Actually it is simplex communication: the first pass only talks to the second pass through the filter.

3. What lint checks

3.1. Set, used and unused variables

We make a distinction between two classes of variables: the class of automatic variables (including register variables) and the other variables. The other variables, global variables, static variables, formal parameters et cetera, are assumed to have a defined value. Global variables e.g., are initialized by the compiled code at zeros; formal parameters have a value which is equal to the value of the corresponding actual parameter. These variables can be used without explicitly initializing them. The initial value of automatic variables is undefined (if they are not initialized at declaration). These variables should be set before they are used. A variable is set by

1. an assignment (including an initialization)
2. taking the address

The first case is clear. The second case is plausible. It would take to much effort (if at all possible) to check if a variable is set through one of its aliases. Because *lint* should not warn about correct constructs, it does this conservative approach. Structures (and unions) can also be set by setting at least one member. Again a conservative approach. An array can be set by using its name (e.g. as actual parameter of a function call). *Lint* warns for usage as *rvalue* of automatic variables which are not set.

A variable is used if

1. it is used as a *rvalue*
2 its address is taken

   Arrays and structures (and unions) are also used if one entry or one member respectively is used.

When a variable is never used in the part of the program where it is visible, a warning is given. For variables declared at the beginning of a compound statement, a check is made at the end of this statement. For formal parameters a check is made at the end of the function definition. At the end of a file this is done for global static definitions. For external variables a warning can be given when all the files are parsed.

3.2. Flow of control

The way *lint* keeps track of the flow of control is best explained by means of an example. See the program of figure 1.

```
if (cond)
    /* a statement which is executed if cond is true,
     * the if-part
     */
else
    /* the else-part */
```

figure 1.

After evaluation of `cond`, two things can happen. The if-part is executed or the else-part is executed (but not both). Variables which are set in the if-part but not in the else-part, need not be set after the if statement, and vice versa. *Lint* detects this and assumes these variables after the if statement to be *maybe set*. (See figure 2.)

If both the if-part and the else-part are never left (i.e. they contain an endless loop or a return statement), *lint* knows that the if statement is never left too. Besides the if statement, *lint* knows the possible flows of control in while, do, for and switch statements. It also detects some endless loops like `while(1),do ... while (1),for (;;)`.

3.3. Functions

Most C compilers will not complain if a function is called with actual parameters of a different type than the function expects. Using a function in one file as a function of type *A* while defining it in another

```
int cond;

main()
{
        int i, j;

        if (cond) {
                i = 0;
                j = 0;
        }
        else
                use(i);  /* i may be used before set */
        use(j);          /* maybe j used before set  */
}
```

figure 2.

file as a function of type *B* is also allowed by most compilers. It needs no explanation that this can lead to serious trouble.

Lint checks if functions are called with the correct number of arguments, if the types of the actual parameters correspond with the types of the formal parameters and if function values are used in a way consistently with their declaration. When the result of a function is used, a check is made to see if the function returns a value. When a function returns a value, *lint* checks if the values of all calls of this function are used.

3.4. Undefined evaluation order

The semantics of C do not define evaluation orders for some constructs, which, at first sight, seem well defined. The evaluation order of the expression `a[i] = i++;` e.g., is undefined. It can be translated to something with the semantics of `a[i] = i; i++;` which is what probably was meant, or `a[i+1] = i; i++;`. An easier example to explain why, is `j = a[i] + i++;`. '+' Is a so called *commutative* operator (with respect to the evaluation order) , as is '='. This allows the compiler to choose which term to evaluate first. It is easy to see, that it makes a difference for the value of `j`, which order is chosen. The expression `i++` is said to have *side effects.* It affects the value of `i`. Because this value is used in the other term, this gives a conflict.

A function call with reference to a variable as argument can have side effects to. Therefor, the evaluation order of `i` in the expression `f(&i) + i` is undefined. When a function is called with an array as argument, this array can be affected by the function, because only the address of the array is passed to the function. (In Pascal a copy of the array is passed to the function if the formal parameter is not declared *var.*) So the evaluation order of `a` in the expression `f(a) + a[0]` is undefined. This one is not yet detected by *lint.*

Global variables can still cause trouble. If function `f` affects the global variable `i`, the value of the expression `f() + i` is undefined, because the evaluation order of `i` is undefined.

The evaluation order of the arguments of a function is not defined, so the expression `f(i, i++)` gives a warning `i evaluation order undefined.`

3.5. Pointer alignment problems

For pointers to objects of different types there are different alignment restrictions. On some machines pointers to type char can have both odd and even values, whereas pointers to type int should contain an even address. *Lint* could warn for all pointer conversions. This is not what *lint* does. *Lint* assumes that some pointers are more restricted than others, and that pointers of some types can safely be converted to a pointer of a less restrictive type. The order of restriction is as follows ('≤' means 'is not more restricted than') :

$$\text{char} \le \text{short} \le \text{int} \le \text{long}$$
$$\text{float} \le \text{double}$$

3.6. Libraries

C is a small language. As a matter of fact it has no i/o routines. To make it a useful language, C is supported by libraries. These libraries contain functions and variables that can be used by any C program. *Lint* knows some libraries too. At this moment it knows the '*-lc*', '*-lm*' and '*-lcurses*' libraries. The '*-lc*' library, containing definitions for functions from chapter two and three of the UNIX programmers manual, is default. *Lint* warns for definitions of functions or global variables with the same name as a function definition in a library.

4. How lint checks

4.1. The first pass first pass data structure

The data structure of *cem* is changed a little and some structures have been added.

4.1.1. The changes

4.1.1.1. Idf descriptor

A member `id_line` is added to the *idf* selector. This line number is used for some warnings.

4.1.1.2. Def descriptor

The *def* selector is extended with the members `df_set` df_line. The `df_used` member did exist already, but was only used for code generation. This usage is eliminated so it can be used by *lint.* The meaning of these members should be clear.

4.1.2. The additions

4.1.2.1. Lint_stack_entry descriptor

```
struct lint_stack_entry {
        struct lint_stack_entry *next;
        struct lint_stack_entry *previous;
        short ls_class;
        int ls_level;
        struct state *ls_current;
        union {
                struct state *S_if;
                struct state *S_end;
                struct switch_states switch_state;
        } ls_states;
};
```

Structure to simulate a stacking mechanism.

`next`	Pointer to the entry on top of this one.
`previous`	Pointer to the entry beneath this one.
`ls_class`	The class of statement this entry belongs to. Possible classes are `IF`, `WHILE`, `DO`, `FOR`, `SWITCH` and `CASE`.
`ls_level`	The level the corresponding statement is nested.
`ls_current`	A pointer to the state descriptor which describes the state of the function (the state of the automatic variables, if the next statement can be reached, et cetera) if control passes the flow of control to the part of the program currently parsed. The initialization of this state is as follows If `ls_class` in [`IF`, `SWITCH`] the state after parsing the conditional expression. If `ls_class` in [`WHILE`, `FOR`] the state after parsing the code between the brackets. If `ls_class` in [`DO`, `CASE`] the state at entrance of the statement after the `DO` or `CASE` token.
`ls_states`	Union of pointers to state descriptors containing different information for different values of `ls_class`. If `ls_class` is `IF` and in case of parsing an else part, `ls_states.S_if` points to the state that is reached after the if part.

If ls_class in [WHILE, FOR, DO] then ls_states.S_end contains a conservative description of the state of the program after ‘jumping’ to the end of the statement after the WHILE, DO or FOR token. I.e. the state at reaching a break (not inside a switch) or continue statement.

If ls_class is SWITCH, ls_states is used as a structure

```
struct switch_states {
        struct state S_case;
        struct state S_break;
};
```

containing two pointers to state descriptors. ls_states.switch_state.S_case contains a conservative description of the state of the program after case ... case parts are parsed. ls_states.switch_state.S_break the state after parsing all the case ... break parts. The reason for ls_states.switch_state.default_met should be self-explanatory.

In case ls_class is CASE, ls_states is not used.

4.1.2.2. State descriptor

```
struct state {
        struct state *next;
        struct auto_def *st_auto_list;
        int st_nrchd;
        int st_warned;
};
```

st_auto_list	Pointer to a list of definitions of the automatic variables whose scope contain the current position in the program.
st_nrchd	True if the next statement can’t be reached.
st_warned	True if a warning has already been given.

4.1.2.3. Auto_def descriptor

```
struct auto_def {
        struct auto_def *next;
        struct idf *ad_idf;
        struct def *ad_def;
        int ad_used;
        int ad_set;
        int ad_maybe_set;
};
```

next	Points to the next auto_definition of the list.
ad_idf	Pointer to the idf descriptor associated with this auto_definition.
ad_def	Ditto for def descriptor.
ad_used	Indicates the state of this automatic variable. Ditto for ad_set and ad_maybe_set. Only one of ad_set and ad_maybe_set may be true.

4.1.2.4. Expr_state descriptor

```
struct expr_state {
        struct expr_state *next;
        struct idf *es_idf;
        arith es_offset;
        int es_used;
        int es_set;
};
```

This structure is introduced to keep track of which variables, array entries and structure members (union members) are set and/or used in evaluating an expression.

`next`	Pointer to the next descriptor of this list.
`es_idf`	Pointer to the idf descriptor this descriptor belongs to.
`es_offset`	In case of an array, a structure or union, this member contains the offset the compiler would generate for locating the array entry or structure/union member.
`es_used`	True if the indicated memory location is used in evaluating the expression.
`es_set`	Ditto for set.

4.1.2.5. Outdef descriptor

```
struct outdef {
        int od_class;
        char *od_name;
        char *od_file;
        unsigned int od_line;
        int od_nrargs;
        struct tp_entry *od_entry;
        int od_returns;
        struct type *od_type;
};
```

As structures of this type are not allocated dynamically by a storage allocator, it contains no next member. An outdef can be given to to `output_def()` to be passed to the second pass. Basically this forms the interface with the second pass.

`od_class` Indicates what kind of definition it is. Possible classes are `EFDF`, `EVDF`, `SFDF`, `SVDF`, `LFDF`, `LVDF`, `EFDC`, `EVDC`, `IFDC`, `FC`, `VU`. ([`E`xternal, `S`tatic, `L`ibrary, `I`mplicit] [`F`unction, `V`ariable] [`DF`inition, `DC`laration, `C`all, `U`sage])

`od_name` The name of the function or variable.

`od_file` The file this definition comes from.

`od_nrargs` If `od_class` is one of `EFDF`, `SFDF` or `LFDF`, this member contains the number of arguments this function has. If the function was preceded by the pseudocomment `/* VARARGS */`, `od_nrargs` gets the value `-1-n`.

`od_entry` A pointer to a list of `od_nrargs` cells, each containing a pointer to the type descriptor of an argument. (`-1-od_nrargs` cells if `od_nrargs < 0`.) `Tp_entry` is defined as

```
struct tp_entry {
        struct tp_entry *next; /* pointer to next cell */
        struct type *te_type;  /* an argument type     */
};
```

`od_returns` For classes `EFDF`, `SFDF` and `LFDF` this member tells if the function returns an expression or not. In case `od_class` is `FC` it is true if the value of the function is used, false otherwise. For other classes this member is not used.

`od_type` A pointer to the type of the function or variable defined or declared. Not used for classes `FC` and `VU`.

4.2. The first pass checking mechanism

In the description of the implementation of the pass one warnings, it is assumed that the reader is familiar with the *LLgen* parser generator, as described in [6].

4.2.1. Used and/or set variables

To be able to give warnings like `%s used before set` and `%s set but not used in function %s` , there needs to be a way to keep track of the state of a variable. A first approach to do this was by adding two fields to the *def* selector: `df_set` and `df_used.` While parsing the program, each time an expression was met this expression was analyzed and the fields of each *def* selector were possibly set during this analysis. This analysis was done by passing each expression to a function `lint_expr` , which walks the expression tree in a way similar to the function `EVAL` in the file *eval.c* of the original *cem* compiler. This approach has one big disadvantage: it is impossible to keep track of the flow of control of the program. No warning will be given for the program fragment of figure 3.

```
func()
{
        int i;

        if (cond)
                i = 0;
        else
                use(i);  /* i may be used before set */
}
```

figure 3.

It is clear that it would be nice having *lint* warn for this construction.

This was done in the second approach. When there was a choice between two statements, each statement was parsed with its own copy of the state at entrance of the *choosing statement.* A state consisted of the state of the automatic variables (including register variables). In addition to the possibilities of being used and set, a variable could be *maybe set.* These states were passed between the statement parsing routines using the *LLgen* parameter mechanism. At the end of a choosing statement, the two states were merged into one state, which became the state after this statement. The construction of figure 4 was now detected, but switch statements still gave problems and continue and break statements were not understood. The main problem of a switch statement is, that the closing bracket (‘`)`’) has to be followed by a *statement.* The syntax shows no choice of statements, as is the case with if, while, do and for statements. Using the *LLgen* parameter mechanism, it is not a trivial task to parse the different case parts of a switch statement with the same initial state and to merge the results into one state. This observation led to the third and final approach, as described next.

Instead of passing the state of the program through the statements parsing routines using the *LLgen* parameters, a special stack is introduced, the *lint_stack.* When a choosing statement is parsed, an entry is pushed on the stack containing the information that is needed to keep track of the state of the program. Each entry contains a description of the *current* state of the program and a field that indicates what part of the program the parser is currently parsing. For all the possible choosing statements I describe the actions to be taken.

At entrance of an if statement, an entry is pushed on the stack with the current state being a copy of the current state of the stack element one below. The class of this entry is `IF`. At reaching the else part, the current state is moved to another place in this stack entry (to `S_IF`), and a new copy of the current state at entrance of this if statement is made. At the end of the else part, the two states are merged into one state, the new current state, and the `IF` entry is popped from the stack. If there is no else part, then the state that is reached after parsing the if part is merged with the current state at entrance of the if statement into the new current state.

At entrance of a while statement a WHILE entry is pushed on the stack containing a copy of the current state. If a continue or break statement is met in the while statement, the state at reaching this continue or break statement is merged with a special state in the WHILE entry, called S_END. (If S_END did not yet contain a state, the state is copied to S_END.) At the end of the while statement this S_END is merged with the current state, which result is merged with the state at entrance of the while statement into the new current state.

A for statement is treated similarly. A do statement is treated the same way too, except that S_END isn't merged with the state at entrance of the do statement, but becomes the new current state.

For switch statements a SWITCH entry is pushed on the stack. Apart from the current state, this entry contains two other states, S_BREAK and S_CASE. S_BREAK initially contains no state, S_CASE initially contains a copy of the current state at entrance of the switch statement. After parsing a case label, a CASE entry is pushed on the stack, containing a copy of the current state. If, after zero or more statements, we meet another case label, the state at reaching this case label is merged with S_CASE of the SWITCH entry below and a new copy of the state at entrance of the switch statement is put in the CASE entry. If we meet a break statement, we merge the current state with S_BREAK of the SWITCH entry below and pop the CASE entry. In addition to this, the occurrence of a default statement inside the switch statement is recorded in the SWITCH entry. At the end of the switch statement we check if we have met a default statement. If not, S_BREAK is merged with the current state at entrance of the switch statement. (Because it is possible that no case label will be chosen.) Next the S_CASE is 'special_merged' with S_BREAK into the new current state. For more details about these merge functions see the sources.

With the approach described above, *lint* is aware of the flow of control in the program. There still are some doubtful constructions *lint* will not detect and there are some constructions (although rare) for which *lint* gives an incorrect warning (see figure 4).

```
{
        int i;

        for (;;) {
                if (cond) {
                        i = 0;
                        break;
                }
        }
        use(i);
        /* lint warns: maybe i used before set
         * although  the  fragment  is correct
         */
}
```

figure 4.

A nice advantage of the method is, that the parser stays clear, i.e. it isn't extended with extra parameters which must pass the states. In this way the parser still is very readable and we have a nice interface with *lint* using function calls.

4.2.2. Undefined evaluation orders

In expressions the values of some variables are used and some variables are set. Of course, the same holds for subexpressions. The compiler is allowed to choose the order of evaluation of subexpressions involving a commutative and associative operator (*, +, &, |, ^), the comma in a parameter list or an assignment operator. In section 3.4 it is made clear that this will lead to statements with ambiguous semantics.

The way these constructs are detected is rather straight forward. The function which parses an expression (lint_expr) returns a linked list containing information telling which variables are set and which variables are used. A variable is indicated by its *idf* descriptor and an *offset.* This offset is needed for discriminating entries of the same array and members of the same structure or union, so it is possible to warn about the statement `a[b[0]] = b[0]++;`. When `lint_expr` meets a commutative operator (with respect to the evaluation order), it calls itself recursively with the operands of the operator as expression. The returned results are checked for undefined evaluation orders and are put together. This is done by the function `check_and_merge`.

4.2.3. Useless statements

Statements which compute a value that is not used, are said to have a *null effect*. Examples are `x = 2, 3;`, `f() + g();` and `*p++;`. (`*` and `++` have the same precedence and associate from right to left.)

A conditional expression computes a value too. If this value isn't used, it is better to use an if-else statement. So, if *lint* sees

```
b ? f() : g();
```

it warns `use if-else construction`.

4.2.4. Not-reachable statements

The algorithm to detect not-reachable statements (including not reachable initializations) is as follows. Statements after a label and a case statement and the compound statement of a function are always reachable. Other statements are not-reachable after:

- a goto statement
- a return statement
- a break statement
- a continue statement
- a switch statement
- an endless loop (a while, do or for loop with a conditional which always evaluates to true and without a break statement)
- an if-else statement of which both if part and else part end up in a not-reachable state
- a switch statement of which all `case ... break` parts (including a `default ... break` part) end up in a not-reachable state
- the pseudocomment `/* NOTREACHED */`

The algorithm is easily implemented using the `st_nrchd` selector in the *state* descriptor. The `st_warned` selector is used to prevent superfluous warnings. To detect an endless loop, after a while (<true>), for (..;<true>;..) and do part the current state of the stack entry beneath the top one is set to not reached. If, in the statement following, a break statement is met, this same state is set to reached. If the while (<cond>) part of the do statement is met, this state is set to reached if <cond> doesn't evaluates to true. The detection of not-reachable statements after a switch statement is done in a similar way. In addition it is checked if a default statement isn't met, in which case the statement after the switch statement can be reached. The warning `statement not reached` is not given for compound statements. If *lint* did, it would warn for the compound statement in a switch statement, which would be incorrect.

Not-reachable statements are still interpreted by *lint*. I.e. when *lint* warns that some statement can't be reached, it assumes this is not what the programmer really wants and it ignores this fact. In this way a lot of useless warnings are prevented in the case of a not-reachable statement. See figure 5.

```
{
        int i;

        for (;;) {
                /* A loop in which the programmer
                 * forgot to introduce a conditional
                 * break statement.
                 * Suppose i is not used in this part.
                 */
        }
        /* some more code in which i is used */
}
/* The warning "statement not reached" highlights the bug.
 * An additional warning "i unused in function %s" is
 * formally correct, but doesn't provide the programmer
 * with useful information.
 */
```

figure 5.

4.2.5. Functions returning expressions and just returning

Each time a return statement is met, *lint* checks if an expression is returned or not. If a function has a return with expression and a return without expression, *lint* warns `function %s has return(e); and return;`. If the flow of control can *fall through* the end of the compound statement of a function, this indicates an implicit return statement without an expression. If the end of the compound statement of the function can be reached, *lint* introduces this implicit return statement without expression.

Sometimes the programmer knows for sure that all case parts inside a switch statement include all possible cases, so he doesn't introduce a default statement. This can lead to an incorrect warning. Figure 6 shows how to prevent this warning.

```
                func()
                {
                        switch (cond) {
                        case 0: return(e0);
                        case 1: return(e1);
                        }
                        /* NOTREACHED */
                }
/* no warning: "function func has return(e); and return; */
```

figure 6.

The pseudocomment `/* NOTREACHED */` can also be used to tell *lint* that some function doesn't return. See figure 7.

```
 func()
 {
         switch (cond) {
         case 0: return(e0);
         case 1: return(e1);
         default: error();   /* calls exit or abort */
                  /* NOTREACHED */
         }
 }
/* no warning: "function func has return(e); and return;" */
```

figure 7.

4.2.6. Output definitions for the second pass

The first pass can only process one program file. To be able to process a program that spreads over more than one file, the first pass outputs definitions that are processed by a second pass. The format of such a definition is different for different classes:

For class in {EFDF, SFDF, LFDF}

<name>:<class>:<file>:<line>:<nr of args>:<type of args>:<returns value>:<type>

A negative *nr of args* indicates that the function can be called with a varying number of arguments.

For class = FC

<name>:<class>:<file>:<line>:<value is used>:<type>

The *value is used* part can have three meanings: the value of the function is ignored; the value of the function is used; the value of the function is cast to type *void*.

For other classes

<name>:<class>:<file>:<line>:<type>

Definitions of class VU (Variable Usage) are only output for *used* global variables.

Structure and union types that are output to the intermediate file are simplified. (The following occurrences of *structure* should be read as *structure or union* and *struct* as *struct or union*.) Structures that are identified by a *structure tag* are output to the intermediate file as `struct <tag>`. Structures without a structure tag are output as `struct {<mems>}` with `<mems>` a semicolon-separated list of types of the members of this structure. An alternative way would be to output the complete structure definition. However, this gives practical problems. It is allowed to define some object of a structure type with a structure tag, without this structure being defined at that place. The first approach leaves errors, such as in figure 8, undetected.

```
      "a.c"                                "b.c"

struct str {                         struct str {
        float f;                             int i;
} s;                                 };

main()                               func(s)
{                                            struct str s;
        func(s);                     {}
}
```

figure 8.

To be able to detect these errors, the first pass should also output definitions of structure tags. The example of figure 8 would then get a warning like `structure str defined inconsistently`

More information on these definitions in section 4.3 and 4.4.

4.2.7. Generating libraries

Lint knows the library ‘-lc’, ‘-lm’ and ‘-lcurses’. If a program uses some other library, it is possible to generate a corresponding *lint library*. To do this, precede all the C source files of this library by the pseudocomment `/* LINTLIBRARY */`. Then feed these files one by one to the first pass of *lint* collecting the standard output in a file and ignoring the warnings. The resulting file contains library definitions of the functions and external variables defined in the library sources, and not more than that. If this file is called ‘llib-l*name*.ln *lint* can be told to search the library by passing it as argument in the command line ‘-llib-l*name*.ln. The implementation of this feature is simple.

As soon as the pseudocomment `/* LINTLIBRARY */` is met, only function and variable definitions are output with class LFDF and LVDF respectively. Other definitions, which otherwise would have been output, are discarded.

Instead of generating a special lint library file, one can make a file containing the library definitions and starting with `/* LINTLIBRARY */`. This file can then be passed to *lint* just by its name. This method isn't as efficient as the first one.

4.2.8. Interpreting the pseudocomments

The interpretation of the pseudocomments is done by the lexical analyzer, because this part of the program already took care of the comments. At first sight this seems very easy: as soon as some pseudocomment is met, raise the corresponding flag. Unfortunately this doesn't work. The lexical analyzer is a *one token look ahead scanner*. This causes the above procedure to raise the flags one token too soon. A solution to get the right effect is to reserve two flags per pseudocomment. The first is set as soon as the corresponding pseudocomment is scanned. At the returning of each token this flag is moved to the second flag. The delay in this way achieved makes the pseudocomments have effect at the correct place.

4.3. The second pass data structure

4.3.1. Inp_def descriptor

```
struct inp_def {
        struct inp_def *next;
        int id_class;
        char id_name[NAMESIZE];
        char id_file[FNAMESIZE];
        unsigned int id_line;
        int id_nrargs;
        char argtps[ARGSTPSSIZE];
        int id_returns;
        char id_type[TYPESIZE];
        int id_called;
        int id_used;
        int id_ignored;
        int id_voided;
};
```

This description is almost similar to the *outdef* descriptor as described in 4.1.2.5. There are some differences too.

`next`	As structures of this type are allocated dynamically, this field is added so the same memory allocator as used in the first pass can be used.

`id_called`
`id_used`
`id_ignored`
`id_voided`
	Some additional fields only used for function definitions.Their meaning should be clear.

The other fields have the same meaning as the corresponding fields in the *outdef* descriptor. Some attention should be paid to `id_argtps` and `id_type`. These members have type `array of char`, in contrast to their counterparts in the *outdef* descriptor. The only operation performed on types is a check on equality. Types are output by the first pass as a string describing the type. The type of `i` in `int *i();` e.g. is output as `int *()`. Such a string is best put in an `array of char` to be compared easily.

4.4. The second pass checking mechanism

After all the definitions that are output by the first pass are sorted by name, the definitions belonging to one name are ordered as follows.

-
external definitions

- static definitions
- library definitions
- declarations
- function calls
- variable usages

The main program of the second pass is easily explained. For all different names, do the following. First read the definitions. If there is more than one definition, check for conflicts. Then read the declarations, function calls and variable usages and check them against the definitions. After having processed all the declarations, function calls and variable usages, check the definitions to see if they are used correctly. The next three paragraphs will explain the three most important functions of the program.

4.4.1. Read_defs()

This function reads all definitions belonging to the same name. Only one external definition is allowed, so if there are more, a warning is given. In different files it is allowed to define static functions or variables with the same name. So if a static function is read, `read_defs` checks if there isn't already an external definition, and if not it puts the static definition in the list of static definitions, to be used later. If no external or static definitions are met, a library definition is taken as definition. If a function or a variable is defined with the same name as a function or a variable in a library (which is allowed) *lint* gives a warning. Of course it is also possible that there is no definition at all. In that case `check` will warn.

4.4.2. Check()

`Check` verifies declarations, function calls and variable usages against the definitions. For each of these entries the corresponding definition is looked up. As there may be more than one static definition, first a static definition from the same file as the entry is searched. If not present, the external definition (which may be a library definition) is taken as definition. If no definition can be found and the current entry is an external declaration, *lint* warns. However in the case of an implicit function declaration *lint* will not warn, because we will get a warning `%s used but not defined` later on. Next a check is done if the declarations are consistent with their definitions. After the declarations, the function calls and variable usages are verified against their corresponding definitions. If no definition exists, *lint* warns. Else the field `id_called` is set to 1. (For variable definitions this should be interpreted as *used*.) For variable usages this will be all. If we are processing a function call we also check the number and types of the arguments and we warn for function values which are used from functions that don't return a value. For each function call we administrate if a function value is used, ignored or voided.

4.4.3. Check_usage()

Checks if the external definition and static definitions are used correctly. If a function or variable is defined but never used, *lint* warns, except for library definitions. Functions, which return a value but whose value is always or sometimes ignored, get a warning. (A function value which is voided (cast to void) is not ignored, but it isn't used either.)

5. How to make lint shut up

It can be very annoying having *lint* warn about questionable constructs of which the programmer already is aware. There should be a mechanism to give *lint* some extra information in the source code. This could be done by introducing some special keywords, which would have a special meaning to *lint.* This is a bad solution, because these keywords would cause existing C compilers not to work on these programs. A neater solution is to invent some comments having a special meaning to *lint.* We call these comments *pseudocomments.* The pseudocomments have no meaning to existing C compilers, so compilers will not have to be rewritten for C programs containing the previously proposed special keywords. The following pseudocomments are recognized by *lint.*

`/* VARARGS`*n*` */`

The next function can be called with a variable number of arguments. Only check the first *n* arguments. The *n* must follow the word `VARARGS` immediately. This pseudocomment is useful for functions like e.g. printf. (The definition of the function printf should be preceded by `/* VARARGS1 */`.)

`/* VARARGS */`

Means the same as `/* VARARGS0 */`.

`/* ARGSUSED */`

Don't complain about unused arguments in the next function. When we are developing a program we sometimes write functions of which we do not yet use the arguments. Because we do want to use *lint* on these programs, it is nice to have this pseudocomment.

`/* NOTREACHED */`

Lint makes no attempt to discover functions which never return, although it *is* possible to find functions that don't return. This would require a transitive closure with respect to the already known *not-returning* functions; an inacceptable time consuming process. To make *lint* aware of a function that doesn't return, a call of this function should be followed by the pseudocomment `/* NOTREACHED */`. This pseudocomment can also be used to indicate that some case part inside a switch (especially a default part) can't be reached. The above mentioned cases of use of this pseudocomment are examples. The comment can be used just to indicate that some part of the program can't be reached. It sometimes is necessary to introduce an extra compound statement to get the right effect. See figure 9.

```
        if (cond)
                /* if part */ ;
        else {
                error();  /* doesn't return */
                /* NOTREACHED */
        }
/* Without  the compound  else  part, lint  would  assume
 * the statement after the if statement to be NOTREACHED,
 * instead of the end of the else part.
 */
```

figure 9.

`/* LINTLIBRARY */`

All definitions following this comment are assumed to be library definitions. It shuts off complaints about unused functions and variables. See also section 4.2.7 for how to use this comment for generating lint libraries.

6. User options

Lint recognizes the following command line flags. Some of them are identical to the flags of *cem. Lint* warns for flags it doesn't know.

`-D<name>`
`-D<name>=<text>`
: Causes `<name>` to be defined as a macro. The first form is equivalent to '`-D<name>=1`'. The second form is equivalent to putting '`#define <name> <text>`' in front of all the source files.

`-U<name>`
: Acts as if the line '`#undef <name>`' is put in front of all the source files.

`-I<directory>`
: This puts `<directory>` in the include directory list.

`-R`
: Turn off the 'strict' option. Default *lint* checks the program according to the Reference Manual, because this gives a definition of the language with which there is a better chance of writing portable programs. With this flag on, some constructs, otherwise not allowed, are accepted.

`-l<name>`
`-llib-l<name>.ln`
`-l`
: '`-l<name>`' tells *lint* to search the lint library `llib-l<name>.ln` for missing definitions of functions and variables. The option '`-llib-l<name>.ln`' makes *lint* search the lint library file `llib-l<name>.ln` in the current directory for missing definitions. Default is '`-lc`'; this default can be suppressed by '`-l`'.

`-a`
: Warn for conversions from integer to long and vice versa.

`-b`
: Don't report not-reachable break statements. This flag is useful for running *lint* on a *lex*- or *yacc*-generated source file.

`-h`
: Check for useless statements and possible pointer alignment problems.

`-n`
: Don't complain about unused and undefined functions and variables.

`-v`
: Don't warn about unused arguments of functions.

`-x`
: Complain about unused external variables.

7. Ideas for further development

Although the program in its current state is a useful program, there are still a lot of features that should be implemented in following versions. I'll summarize them in this section.

- Actually the program consists of three passes. The filter *sort* is a complete pass, just as the first and the second pass. I think we speed up the program by removing the filter and making the second pass accept an unsorted file. The sorting process can be done in parallel to the first pass if both processes communicate through a pipe. In addition to this sorting, the second pass can generate already some warnings. (Warnings like `%s defined but never used` can only be generated after having processed all the input.) These warnings generated in parallel to the warnings of the first pass, should be sent to an intermediate file, otherwise the warnings would get messed up. Such an improvement will have best effect on a multi processing machine, but even on single processing machines this will give a better performance. (On a single processing machine the pipe should be replaced by an intermediate file.)
- Expressions could be classified so *lint* can warn for some classes of expressions in strange contexts. Suppose as class <boolean>. `b` Will be of class <boolean> if e.g. `b` is assigned to the expression `<ex1> || <ex2>`. The following expression should then give a warning

```
b + i;     /* weird expression */
```

- A mechanism to check printf like routines. This mechanism should verify the format string against the following arguments. There is a public domain program that can be used to do this job. It is called printfck and should be used as a filter between the source files and *lint*.
- Raise warnings for incomplete initializer lists like

```
int a[10] = {0, 1, 2};
/* initializer list not complete */
```

- Warnings for constructs like

```
for (i = 0; i < 10; i++) {
        . . . .
        i--;
        /* loop control variable affected */
        . . . .
}
```

and

```
while (var) {
        /* statements in which the value
         * of var is never changed
         */
}
/* loop control variable not updated */
```

- A warning `bad layout` for program fragments like

```
if (cond1)
        if (cond2)
                statement();
else   /* bad layout */
        statement();
```

- A warning `assignment in conditional context` in case of

```
if (a = b)
```

The programmer probably meant `if (a == b)`. No warning should be given for `if ((a = b) != c)`, nor for `if ((a = b))`.

- Warnings for empty statements in strange contexts, like

```
if (cond);  /* mistake */
        statement();
```

  (This mistake would also be detected by a warning `bad layout`.)

- A mechanism to prevent the warning `possible pointer alignment problem` for functions of which the programmer already knows that no problem will arise. E.g. for functions like malloc and family.

- The current version of *lint* warns for conversions from long to int (if -a flag is on). It even warns if the programmer used the proper cast, as e.g.

```
int i;
long l = 0L;

i = (int)l;
```

  In this case I think *lint* need not warn. The explicit cast indicates that the programmer knows what he is doing. This feature is not implemented because the expression tree doesn't show if the cast was implicit or explicit.

8. Testing the program

There is no test-suite for testing *lint.* I have written a lot of small files that each test one particular property of the program. At this moment there are about 220 test programs.

It would take a lot of time and effort to run these tests by hand. To ease this work I wrote a program that runs these tests automatically. The test program (the program that runs the tests) needs, associated with each .c file, a .w file, containing from each expected warning a substring. E.g. when the following warnings should be given by *lint:*

```
file t.c, line 3, i evaluation order undefined
file t.c, line 6, a set but not used in function main
```

it is sufficient to write a file `t.w` containing

```
a set but not used in function main
i evaluation order undefined
```

The test program is called with all the .c files to be tested as arguments.

Sometimes it is necessary to test *lint* on two files. The test program runs *lint* on two files when two consecutive arguments are of the form *name*a.c and *name*b.c. It then compares the output of *lint* with the file *name*.w.

Lint is also tested by running it on existing programs. *Lint* has been run on some UNIX utility programs in /usr/src/cmd, on Unipress Emacs (consisting of more than 30,000 lines of code) and the program itself. Bugs have been found in e.g. /usr/src/cmd/cat.c and /usr/src/cmd/ld.c. To test the robustness of the program, it was run on the password file /etc/passwd and on ‘mixed’ C program files. These mixed C program files are C program files that were broken in chunks and then put together in a different order.

9. References

[1] Dennis M. Ritchie, *C Reference Manual,* Bell Laboratories, Murray Hill, New Jersey, 1978.

[2] B.W. Kernighan and D.M. Ritchie, *The C Programming Language,* Prentice Hall, 1978.

[3] Eric H. Baalbergen, Dick Grune, Maarten Waage, *The CEM Compiler,* Manual IM-4, Vrije Universiteit, Amsterdam, 1985.

[4] Andrew S. Tanenbaum et al., *A practical tool kit for making portable compilers,* Comm. ACM, Sep. 1983.

[5] S. C. Johnson, *Lint, a C program verifier,* Bell Laboratories, Murray Hill, New Jersey, 1978.

[6] Dick Grune, Ceriel J. H. Jacobs, *A Programmer-friendly LL(1) Parser Generator,* IR 127, Vrije Universiteit, Amsterdam, 1987.

Appendix A

The warnings

Pass one warnings

```
%s may be used before set
maybe %s used before set
%s unused in function %s
%s set but not used in function %s
argument %s unused in function %s
static [variable, function] %s unused
%s declared extern but never used

long conversion may lose accuracy
comparison of unsigned with negative constant
unsigned comparison with 0?
degenerate unsigned comparison
nonportable character comparison
possible pointer alignment problem

%s evaluation order undefined

null effect
constant in conditional context
use if-else construction
while (0) ?
do ... while (0) ?
[case, default] statement in strange context

function %s has return(e); and return;
statement not reached
function %s declared %s but no value returned
```

Pass two warnings

```
%s variable # of args
%s arg %d used inconsistently
%s multiply defined
%s value declared inconsistently
%s used but not defined
%s defined (%s(%d)) but never used
%s declared but never defined
%s value is used but none is returned
%s returns value which is [sometimes, always] ignored
%s also defined in library
```

Appendix B

The Ten Commandments for C Programmers

Henry Spencer

1 Thou shalt run *lint* frequently and study its pronouncements with care, for verily its perception and judgement oft exceed thine.

2 Thou shalt not follow the NULL pointer, for chaos and madness await thee at its end.

3 Thou shalt cast all function arguments to the expected type if they are not of that type already, even when thou art convinced that this is unnecessary, lest they take cruel vengeance upon thee when thou least expect it.

4 If thy header files fail to declare the return types of thy library functions, thou shalt declare them thyself with the most meticulous care, lest grievous harm befall thy program.

5 Thou shalt check the array bounds of all strings (indeed, all arrays), for surely where thou typest "foo" someone someday shall type "supercalifragilisticexpialidocious".

6 If a function be advertised to return an error code in the event of difficulties, thou shalt check for that code, yea, even though the checks triple the size of thy code and produce aches in thy typing fingers, for if thou thinkest "it cannot happen to me", the gods shall surely punish thee for thy arrogance.

7 Thou shalt study thy libraries and strive not to re-invent them without cause, that thy code may be short and readable and thy days pleasant and productive.

8 Thou shalt make thy program's purpose and structure clear to thy fellow man by using the One True Brace Style, even if thou likest it not, for thy creativity is better used in solving problems than in creating beautiful new impediments to understanding.

9 Thy external identifiers shall be unique in the first six characters, though this harsh discipline be irksome and the years of its necessity stretch before thee seemingly without end, lest thou tear thy hair out and go mad on that fateful day when thou desirest to make thy program run on an old system.

10 Thou shalt foreswear, renounce, and abjure the vile heresy which claimeth that "All the world's a VAX", and have no commerce with the benighted heathens who cling to this barbarous belief, that the days of thy program may be long even though the days of thy current machine be short.