

The SmPL Grammar (version 1.0.0-rc21)

Research group on Coccinelle

April 15, 2014

This document presents the grammar of the SmPL language used by the Coccinelle tool. For the most part, the grammar is written using standard notation. In some rules, however, the left-hand side is in all uppercase letters. These are macros, which take one or more grammar rule right-hand-sides as arguments. The grammar also uses some unspecified nonterminals, such as *id*, *const*, etc. These refer to the sets suggested by the name, *i.e.*, *id* refers to the set of possible C-language identifiers, while *const* refers to the set of possible C-language constants. A HTML version of this documentation is available online at http://coccinelle.lip6.fr/docs/main_grammar.html.

1 Program

```
program      ::= include_cocci* changeset+  
include_cocci ::= include string  
                | using string  
                | using pathToIsoFile  
                | virtual id (, id)*  
changeset    ::= metavariables transformation  
                | script_metavariables script_code
```

script_code is any code in the chosen scripting language. Parsing of the semantic patch does not check the validity of this code; any errors are first detected when the code is executed. Furthermore, @ should not be use in this code. Spatch scans the script code for the next @ and considers that to be the beginning of the next rule, even if @ occurs within e.g., a comment.

virtual keyword is used to declare virtual rules. Virtual rules may be subsequently used as a dependency for the rules in the SmPL file. Whether a virtual rule is defined or not is controlled by the `-D` option on the command line.

2 Metavariables for transformations

The *rulename* portion of the metavariable declaration can specify properties of a rule such as its name, the names of the rules that it depends on, the isomorphisms to be used in processing the rule, and whether quantification over paths should be universal or existential. The optional annotation *expression* indicates that the pattern is to be considered as matching an expression, and thus can be used to avoid some parsing problems.

The *metadefl* portion of the metavariable declaration defines various types of metavariables that will be used for matching in the transformation section.

```

metavariables      ::= @@ metadec* @@
                    | @ rulename @ metadec* @@
rulename           ::= id [extends id] [depends on dep] [iso] [disable-iso] [exists] [expression]
dep                ::= id
                    | !id
                    | ! (dep)
                    | ever id
                    | never id
                    | dep && dep
                    | dep || dep
                    | (dep)
iso                ::= using string (, string)*
disable-iso        ::= disable COMMA_LIST(id)
exists             ::= exists
                    | forall
COMMA_LIST(elem)   ::= elem (, elem)*

```

The keyword `disable` is normally used with the names of isomorphisms defined in `standard.iso` or whatever isomorphism file has been included. There are, however, some other isomorphisms that are built into the implementation of Coccinelle and that can be disabled as well. Their names are given below. In each case, the text describes the standard behavior. Using `disable-iso` with the given name disables this behavior.

- `optional_storage`: A SmPL function definition that does not specify any visibility (i.e., static or extern), or a SmPL variable declaration that does not specify any storage (i.e., auto, static, register, or extern), matches a function declaration or variable declaration with any visibility or storage, respectively.
- `optional_qualifier`: This is similar to `optional_storage`, except that here is it the qualifier (i.e., const or volatile) that does not have to be specified in the SmPL code, but may be present in the C code.
- `value_format`: Integers in various formats, e.g., 1 and 0x1, are considered to be equivalent in the matching process.
- `optional_declarer_semicolon`: Some declarers (top-level terms that look like function calls but serve to declare some variable) don't require a semicolon. This isomorphism allows a SmPL declarer with a semicolon to match such a C declarer, if no transformation is specified on the SmPL semicolon.
- `comm_assoc`: An expression of the form `exp bin_op . . .`, where `bin_op` is commutative and associative, is considered to match any top-level sequence of `bin_op` operators containing `exp` as the top-level argument.

The possible types of metavariable declarations are defined by the grammar rule below. Metavariables should occur at least once in the transformation immediately following their declaration. Fresh identifier metavariables must only be used in + code. These properties are not expressed in the grammar, but are checked by a subsequent analysis. The metavariables are designated according to the kind of terms they can match, such as a statement, an identifier, or an expression. An expression metavariable can be further constrained by its type. A declaration metavariable matches the declaration of one or more variables, all sharing the same type specification (e.g., `int a,b,c=3;`). A field metavariable does the same, but for structure fields.

```

metadecl ::= metavariable ids ;
| fresh identifier ids ;
| identifier COMMA_LIST(pmid_with_regexp) ;
| identifier COMMA_LIST(pmid_with_virt_or_not_eq) ;
| parameter [list] ids ;
| parameter list [ id ] ids ;
| parameter list [ const ] ids ;
| type ids ;
| statement [list] ids ;
| declaration ids ;
| field [list] ids ;
| typedef ids ;
| declarer name ids ;
| declarer COMMA_LIST(pmid_with_regexp) ;
| declarer COMMA_LIST(pmid_with_not_eq) ;
| iterator name ids ;
| iterator COMMA_LIST(pmid_with_regexp) ;
| iterator COMMA_LIST(pmid_with_not_eq) ;
| [local] idexpression [ctype] COMMA_LIST(pmid_with_not_eq) ;
| [local] idexpression [{ctypes} *] COMMA_LIST(pmid_with_not_eq) ;
| [local] idexpression *+ COMMA_LIST(pmid_with_not_eq) ;
| expression list ids ;
| expression *+ COMMA_LIST(pmid_with_not_eq) ;
| expression enum * COMMA_LIST(pmid_with_not_eq) ;
| expression struct * COMMA_LIST(pmid_with_not_eq) ;
| expression union * COMMA_LIST(pmid_with_not_eq) ;
| expression COMMA_LIST(pmid_with_not_ceq) ;
| expression list [ id ] ids ;
| expression list [ const ] ids ;
| ctype [ ] COMMA_LIST(pmid_with_not_eq) ;
| ctype COMMA_LIST(pmid_with_not_ceq) ;
| {ctypes} * COMMA_LIST(pmid_with_not_ceq) ;
| {ctypes} * [ ] COMMA_LIST(pmid_with_not_eq) ;
| constant [ctype] COMMA_LIST(pmid_with_not_eq) ;
| constant [{ctypes} *] COMMA_LIST(pmid_with_not_eq) ;
| position [any] COMMA_LIST(pmid_with_not_eq_mid) ;
| symbol ids;
| format ids;
| format list [ id ] ids ;
| format list [ const ] ids ;

```

A metavariable declaration local idexpression *v* means that *v* is restricted to be a local variable. If it should just be a variable, but not necessarily a local one, then drop local. A more complex description of a location, such as *a->b* is considered to be an expression, not an ideexpression.

Constant is for constants, such as 27. But it also considers an identifier that is all capital letters (possibly containing numbers) as a constant as well, because the names gives to macros in Linux usually have this form.

An identifier is the name of a structure field, a macro, a function, or a variable. Is is the name of something rather than an expression that has a value. But an identifier can be used in the position of an expression as well, where it represents a variable.

It is possible to specify that an expression list or a parameter list metavariable should match a specific number of expressions or parameters.

It is possible to specify some information about the definition of a fresh identifier. See the wiki.

A symbol declaration specifies that the provided identifiers should be considered C identifiers when encountered in the body of the rule. Identifiers in the body of the rule that are not declared explicitly are by default considered symbols, thus symbol declarations are optional.

A position metavariable is used by attaching it using @ to any token, including another metavariable. Its value is the position (file, line number, etc.) of the code matched by the token. It is also possible to attach expression, declaration, type, initialiser, and statement metavariables in this manner. In that case, the metavariable is bound to the closest enclosing expression, declaration, etc. If such a metavariable is itself followed by a position metavariable, the position metavariable applies to the metavariable that it follows, and not to the attached token. This makes it possible to get eg the starting and ending position of $f(\dots)$, by writing $f(\dots)@E@p$, for expression metavariable E and position metavariable p .

When used, a format or format list metavariable must be enclosed by a pair of @s. A format metavariable matches the format descriptor part, i.e., $2x$ in $\%2x$. A format list metavariable matches a sequence of format descriptors as well as the text between them. Any text around them is matched as well, if it is not matched by the surrounding text in the semantic patch. Such text is not partially matched. If the length of the format list is specified, that indicates the number of matched format descriptors. It is also possible to use \dots in a format string, to match a sequence of text fragments and format descriptors. This only takes effect if the format string contains format descriptors. Note that this makes it impossible to require \dots to match exactly in a string, if the semantic patch string contains format descriptors. If that is needed, some processing with a scripting language would be required. An example for the use of string format metavariables is found in `demos/format.cocci`.

Other kinds of metavariables can also be attached using @ to any token. In this case, the metavariable floats up to the enclosing appropriate expression. For example, $3 + @E 4$, where E is an expression metavariable binds E to $3 + 4$. A particular case is $Ps@Es$, where Ps is a parameter list and Es is an expression list. This pattern matches a parameter list, and then matches Es to the list of expressions, ie a possible argument list, represented by the names of the parameters.

```

ids          ::= COMMA_LIST(pmid)
pmid          ::= id
               | mid
mid          ::= rulename_id.id
pmid_with_regexp ::= pmid =~ regexp
               | pmid !~ regexp
pmid_with_not_eq ::= pmid [!= id_or_meta]
               | pmid [!= { COMMA_LIST(id_or_meta) }]
pmid_with_virt_or_not_eq ::= virtual.id
               | pmid_with_not_eq
pmid_with_not_ceq ::= pmid [!= id_or_cst]
               | pmid [!= { COMMA_LIST(id_or_cst) }]
id_or_cst     ::= id
               | integer
id_or_meta    ::= id
               | rulename_id.id
pmid_with_not_eq_mid ::= pmid [!= mid]
               | pmid [!= { COMMA_LIST(mid) }]

```

Subsequently, we refer to arbitrary metavariables as metaid^{ty} , where ty indicates the *metakind* used in the declaration of the variable. For example, $\text{metaid}^{\text{Type}}$ refers to a metavariable that was declared using `type` and stands for any type.

`metavariable` declares a metavariable for which the parser tried to figure out the metavariable type based on the usage context. Such a metavariable must be used consistently. These metavariables cannot be used in all contexts; specifically, they cannot be used in context that would make the parsing ambiguous. Some examples are the leftmost term of an expression, such as the left-hand side of an assignment, or the type in a variable declaration. These restrictions may seem somewhat arbitrary from the user's point of view. Thus, it is better to use metavariables with

metavariable types. If Coccinelle is given the argument `-parse_cocci`, it will print information about the type that is inferred for each metavariable.

The *c_{type}* and *c_{types}* nonterminals are used by both the grammar of metavariable declarations and the grammar of transformations, and are defined on page 9.

An identifier metavariable with `virtual` as its “rule name” is given a value on the command line. For example, if a semantic patch contains a rule that declares an identifier metavariable with the name `virtual.alloc`, then the command line could contain `-D alloc=kmalloc`. There should not be space around the `=`. An example is in `demos/vm.cocci` and `demos/vm.c`.

Warning: Each metavariable declaration causes the declared metavariables to be immediately usable, without any inheritance indication. Thus the following are correct:

```
@@
type r.T;
T x;
@@

[...] // some semantic patch code

@@
r.T x;
type r.T;
@@

[...] // some semantic patch code
```

But the following is not correct:

```
@@
type r.T;
r.T x;
@@

[...] // some semantic patch code
```

This applies to position variables, type metavariables, identifier metavariables that may be used in specifying a structure type, and metavariables used in the initialization of a fresh identifier. In the case of a structure type, any identifier metavariable indeed has to be declared as an identifier metavariable in advance. The syntax does not permit `r.n` as the name of a structure or union type in such a declaration.

3 Metavariables for scripts

Metavariables for scripts can only be inherited from transformation rules. In the spirit of scripting languages such as Python that use dynamic typing, metavariables for scripts do not include type declarations.

```
script_metavariables ::= @ script:language [rulename] [depends on dep] @ script_metadecl* @@
                        | @ initialize:language [depends on dep] @ script_virt_metadecl* @@
                        | @ finalize:language [depends on dep] @ script_virt_metadecl* @@
language              ::= python
                        | ocaml
script_metadecl       ::= id << rulename_id.id ;
                        | id ;
script_virt_metadecl  ::= id << virtual.id ;
```

Currently, the only scripting languages that are supported are Python and OCaml, indicated using `python` and `ocaml`, respectively. The set of available scripting languages may be extended at some point.

Script rules declared with `initialize` are run before the treatment of any file. Script rules declared with `finalize` are run when the treatment of all of the files has completed. There can be at most one of each per scripting language (thus currently at most one of each). Initialize and finalize script rules do not have access to SmPL metavariables. Nevertheless, a finalize script rule can access any variables initialized by the other script rules, allowing information to be transmitted from the matching process to the finalize rule.

Initialize and finalize rules do have access to virtual metavariables, using the usual syntax. As for other scripting language rules, the rule is not run (and essentially does not exist) if some of the required virtual metavariables are not bound. In `ocaml`, a warning is printed in this case. An example is found in `demos/initvirt.cocci`.

A script metavariable that does not specify an origin, using `<`, is newly declared by the script. This metavariable should be assigned to a string and can be inherited by subsequent rules as an identifier. In Python, the assignment of such a metavariable `x` should refer to the metavariable as `coccinelle.x`. Examples are in the files `demos/pythontococci.cocci` and `demos/camltococci.cocci`.

In an `ocaml` script, the following extended form of `script_metadecl` may be used:

```
script_metadecl ::= (id,id) << rulename_id.id ;
                  | id << rulename_id.id ;
                  | id ;
```

In a declaration of the form `(id,id) << rulename_id.id ;`, the left component of `(id,id)` receives a string representation of the value of the inherited metavariable while the right component receives its abstract syntax tree. The file `parsing_c/ast_c.ml` in the Coccinelle implementation gives some information about the structure of the abstract syntax tree. Either the left or right component may be replaced by `_`, indicating that the string representation or abstract syntax trees representation is not wanted, respectively.

The abstract syntax tree of a metavariable declared using `metavariable` is not available.

4 Transformation

The transformation specification essentially has the form of C code, except that lines to remove are annotated with `-` in the first column, and lines to add are annotated with `+`. A transformation specification can also use *dots*, “`...`”, describing an arbitrary sequence of function arguments or instructions within a control-flow path. Implicitly, “`...`” matches the shortest path between something that matches the pattern before the dots (or the beginning of the function, if there is nothing before the dots) and something that matches the pattern after the dots (or the end of the function, if there is nothing after the dots). Dots may be modified with a `when` clause, indicating a pattern that should not occur anywhere within the matched sequence. `when any` removes the aforementioned constraint that “`...`” matches the shortest path. Finally, a transformation can specify a disjunction of patterns, of the form `(pat1 | ... | patn)` where each `(, | or)` is in column 0 or preceded by `\`.

The grammar that we present for the transformation is not actually the grammar of the SmPL code that can be written by the programmer, but is instead the grammar of the slice of this consisting of the `-` annotated and the unannotated code (the context of the transformed lines), or the `+` annotated code and the unannotated code. For example, for parsing purposes, the following transformation is split into the two variants shown below and each is parsed separately.

```
1  proc_info_func(...) {
2      <...
3  -    hostno
4  +    hostptr->host_no
5      ...>
6  }
```

```

1  proc_info_func(...) {
2      <...
3  -    hostno
4      ...>
5  }

```

```

1  proc_info_func(...) {
2      <...
3  +    hostptr->host_no
4      ...>
5  }

```

Requiring that both slices parse correctly ensures that the rule matches syntactically valid C code and that it produces syntactically valid C code. The generated parse trees are then merged for use in the subsequent matching and transformation process.

The grammar for the minus or plus slice of a transformation is as follows:

```

transformation ::= include+
                | OPTDOTSEQ(top, when)
include         ::= #include include_string
top             ::= expr
                | decl_stmt+
                | fundecl
when           ::= when != when_code
                | when = rule_elem_stmt
                | when COMMA_LIST(any_strict)
                | when true != expr
                | when false != expr
when_code      ::= OPTDOTSEQ(decl_stmt+, when)
                | OPTDOTSEQ(expr, when)
rule_elem_stmt ::= one_decl
                | expr;
                | return [expr];
                | break;
                | continue;
                | \ (rule_elem_stmt (\ | rule_elem_stmt)+\)
any_strict     ::= any
                | strict
                | forall
                | exists

```

$OPTDOTSEQ(grammar_ds, when_ds) ::= [\dots (when_ds)^*] grammar_ds (\dots (when_ds)^* grammar_ds)^* [\dots (when_ds)^*]$

Lines may be annotated with an element of the set $\{-, +, *\}$ or the singleton $?$, or one of each set. $?$ represents at most one match of the given pattern, ie a match of the pattern is optional. $*$ is used for semantic match, *i.e.*, a pattern that highlights the fragments annotated with $*$, but does not perform any modification of the matched code. $*$ cannot be mixed with $-$ and $+$. There are some constraints on the use of these annotations:

- Dots, *i.e.* \dots , cannot occur on a line marked $+$.
- Nested dots, *i.e.*, dots enclosed in $<$ and $>$, cannot occur on a line with any marking.

Each element of a disjunction must be a proper term like an expression, a statement, an identifier or a declaration. Thus, the rule on the left below is not a syntactically correct SmPL rule. One may use the rule on the right instead.

```

1 @@
2 type T;
3 T b;
4 @@
5
6 (
7  writeb(...,
8  |
9  readb(...,
10 )
11 - (T)
12  b)

```

```

1 @@
2 type T;
3 T b;
4 @@
5
6 (
7  read
8  |
9  write
10 )
11  (... ,
12 -  (T)
13  b)

```

Some kinds of terms can only appear in + code. These include comments, ifdefs, and attributes (`__attribute__((...))`).

5 Types

<i>ctypes</i>	::= <i>COMMA_LIST</i> (<i>ctype</i>)
<i>ctype</i>	::= [<i>const_vol</i>] <i>generic_ctype</i> * [*] [<i>const_vol</i>] void * ⁺ (<i>ctype</i> (<i>ctype</i>)*)
<i>const_vol</i>	::= const volatile
<i>generic_ctype</i>	::= <i>ctype_qualif</i> [<i>ctype_qualif</i>] char [<i>ctype_qualif</i>] short [<i>ctype_qualif</i>] short int [<i>ctype_qualif</i>] int [<i>ctype_qualif</i>] long [<i>ctype_qualif</i>] long int [<i>ctype_qualif</i>] long long [<i>ctype_qualif</i>] long long int double long double float size_t ssize_t ptrdiff_t enum <i>id</i> { <i>PARAMSEQ</i> (<i>dot_expr</i> , <i>exp_whencode</i>) [,] } [struct union] <i>id</i> [{ <i>struct_decl_list</i> * }]
<i>ctype_qualif</i>	::= unsigned signed
<i>struct_decl_list</i>	::= <i>struct_decl_list_start</i>
<i>struct_decl_list_start</i>	::= <i>struct_decl</i> <i>struct_decl struct_decl_list_start</i> ... [when != <i>struct_decl</i>] [†] [<i>continue_struct_decl_list</i>]
<i>continue_struct_decl_list</i>	::= <i>struct_decl struct_decl_list_start</i> <i>struct_decl</i>
<i>struct_decl</i>	::= <i>ctype d_ident</i> ; <i>fn_ctype</i> (* <i>d_ident</i>) (<i>PARAMSEQ</i> (<i>name_opt_decl</i> , ε)) ;) [<i>const_vol</i>] <i>id d_ident</i> ;
<i>d_ident</i>	::= <i>id</i> [[<i>expr</i>]]*
<i>fn_ctype</i>	::= <i>generic_ctype</i> * [*] void * [*]
<i>name_opt_decl</i>	::= <i>decl</i> <i>ctype</i> <i>fn_ctype</i>

[†] The optional when construct ends at the end of the line.

6 Function declarations

```

fundecl ::= [fn_ctype] funinfo* funid ([PARAMSEQ(param,  $\varepsilon$ )] { [stmt_seq] }
funproto ::= [fn_ctype] funinfo* funid ([PARAMSEQ(param,  $\varepsilon$ )] );
funinfo ::= inline
           | storage
storage ::= static
           | auto
           | register
           | extern
funid ::= id
          | metaidld
          | OR(stmt)
param ::= type id
          | metaidParam
          | metaidParamList
decl ::= ctype id
          | fn_ctype ( * id ) (PARAMSEQ(name_opt_decl,  $\varepsilon$ ))
          | void
          | metaidParam

```

PARAMSEQ(*gram_p*, *when_p*) ::= *COMMA_LIST*(*gram_p* | ... [*when_p*])

To match a function it is not necessary to provide all of the annotations that appear before the function name. For example, the following semantic patch:

```

1 @@
2 @@
3
4 foo() { ... }

```

matches a function declared as follows:

```

1 static int foo() { return 12; }

```

This behavior can be turned off by disabling the `optional_storage` isomorphism. If one adds code before a function declaration, then the effect depends on the kind of code that is added. If the added code is a function definition or CPP code, then the new code is placed before all information associated with the function definition, including any comments preceding the function definition. On the other hand, if the new code is associated with the function, such as the addition of the keyword `static`, the new code is placed exactly where it appears with respect to the rest of the function definition in the semantic patch. For example,

```

1 @@
2 @@
3
4 + static
5 foo() { ... }

```

causes `static` to be placed just before the function name. The following causes it to be placed just before the type

```

1 @@
2 type T;
3 @@
4
5 + static
6 T foo() { ... }

```

It may be necessary to consider several cases to ensure that the added ode is placed in the right position. For example, one may need one pattern that considers that the function is declared `inline` and another that considers that it is not.

7 Declarations

```

decl_var      ::= common_decl
                | [storage] ctype COMMA_LIST(d_ident) ;
                | [storage] [const_vol] id COMMA_LIST(d_ident) ;
                | [storage] fn_ctype ( * d_ident ) ( PARAMSEQ(name_opt_decl, ε) ) = initialize ;
                | typedef ctype typedef_ident ;
one_decl      ::= common_decl
                | [storage] ctype id ;
                | [storage] [const_vol] id d_ident ;
common_decl   ::= ctype ;
                | funproto
                | [storage] ctype d_ident = initialize ;
                | [storage] [const_vol] id d_ident = initialize ;
                | [storage] fn_ctype ( * d_ident ) ( PARAMSEQ(name_opt_decl, ε) ) ;
                | decl_ident ( [COMMA_LIST(expr)] ) ;
initialize    ::= dot_expr
                | metaidInitialiser
                | { [COMMA_LIST(init_list_elem)] }
init_list_elem ::= dot_expr
                | designator = initialize
                | metaidInitialiser
                | metaidInitialiserList
                | id : dot_expr
designator      ::= . id
                | [ dot_expr ]
                | [ dot_expr ... dot_expr ]
decl_ident     ::= DeclarerId
                | metaidDeclarer

```

An initializer for a structure can be ordered or unordered. It is considered to be unordered if there is at least one key-value pair initializer, e.g., `.x = e`.

A declaration can have e.g. the form `register x;`. In this case, the variable implicitly has type `int`, and SmPL code that declares an `int` variable will match such a declaration. On the other hand, the implicit `int` type has no position. If the SmPL code tries to record the position of the type, the match will fail.

8 Statements

The first rule *statement* describes the various forms of a statement. The remaining rules implement the constraints that are sensitive to the context in which the statement occurs: *single_statement* for a context in which only one statement is allowed, and *decl_statement* for a context in which a declaration, statement, or sequence thereof is allowed.

```

stmt ::= directive
      | metaidStmt
      | expr;
      | if (dot_expr) single_stmt [else single_stmt]
      | for ([dot_expr]; [dot_expr]; [dot_expr]) single_stmt
      | while (dot_expr) single_stmt
      | do single_stmt while (dot_expr) ;
      | iter_ident (dot_expr*) single_stmt
      | switch ([dot_expr]) {case_line* }
      | return [dot_expr];
      | { [stmt_seq] }
      | NEST(decl_stmt+, when)
      | NEST(expr, when)
      | break;
      | continue;
      | id:
      | goto id;
      | {stmt_seq }

directive ::= include
      | #define id [top]
      | #define id (PARAMSEQ(id,  $\varepsilon$ )) [top]
      | #undef id
      | #pragma id id+
      | #pragma id (PARAMSEQ(expr,  $\varepsilon$ ))
      | #pragma id ...

single_stmt ::= stmt
      | OR(stmt)

decl_stmt ::= metaidStmtList
      | decl_var
      | stmt
      | OR(stmt_seq)

stmt_seq ::= decl_stmt* [DOTSEQ(decl_stmt+, when) decl_stmt*]
      | decl_stmt* [DOTSEQ(expr, when) decl_stmt*]

case_line ::= default : stmt_seq
      | case dot_expr : stmt_seq

iter_ident ::= iteratorId
      | metaidIterator

OR(gram_o) ::= ( gram_o (| gram_o)*)
DOTSEQ(gram_d, when_d) ::= ... [when_d] (gram_d ... [when_d])*
NEST(gram_n, when_n) ::= <... [when_n] gram_n (... [when_n] gram_n)* ...>
      | <+... [when_n] gram_n (... [when_n] gram_n)* ...+>

```

OR is a macro that generates a disjunction of patterns. The three tokens (, |, and) must appear in the leftmost column, to differentiate them from the parentheses and bit-or tokens that can appear within expressions (and cannot appear in the leftmost column). These token may also be preceded by \ when they are used in an other column. These tokens are furthermore different from (, |, and), which are part of the grammar metalanguage.

9 Expressions

A nest or a single ellipsis is allowed in some expression contexts, and causes ambiguity in others. For example, in a sequence ...*expr* ..., the nonterminal *expr* must be instantiated as an explicit C-language expression, while

in an array reference, $expr_1 [expr_2]$, the nonterminal $expr_2$, because it is delimited by brackets, can be also instantiated as \dots , representing an arbitrary expression. To distinguish between the various possibilities, we define three nonterminals for expressions: $expr$ does not allow either top-level nests or ellipses, $nest_expr$ allows a nest but not an ellipsis, and dot_expr allows both. The `EXPR` macro is used to express these variants in a concise way.

```

expr      ::= EXPR(expr)
nest_expr ::= EXPR(nest_expr)
           | NEST(nest_expr, exp_whencode)
dot_expr  ::= EXPR(dot_expr)
           | NEST(dot_expr, exp_whencode)
           | ... [exp_whencode]
EXPR(exp) ::= exp assign_op exp
           | exp++
           | exp--
           | unary_op exp
           | exp bin_op exp
           | exp ? dot_expr : exp
           | (type) exp
           | exp [dot_expr]
           | exp . id
           | exp -> id
           | exp ([PARAMSEQ(arg, exp_whencode)])
           | id
           | (type) { COMMA_LIST(init_list_elem) }
           | metaidExp
           | metaidConst
           | const
           | (dot_expr)
           | OR(exp)
arg        ::= nest_expr
           | metaidExpList
exp_whencode ::= when != expr
assign_op  ::= = | -= | += | *= | /= | %=
           | &= | |= | ^= | <<= | >>=
bin_op     ::= * | / | % | + | -
           | << | >> | ^ | & | |
           | < | > | <= | >= | == | != | && | ||
unary_op   ::= ++ | -- | & | * | + | - | !

```

10 Constants, Identifiers and Types for Transformations

```

const      ::= string
           | [0-9]+
           | ...
string     ::= "[^"]*"
id         ::= id | metaidId | OR(stmt)
typedef_ident ::= id | metaidType
type       ::= ctype | metaidType
pathToIsoFile ::= <.*>
regex     ::= "[^"]*"

```

11 Comments and preprocessor directives

A `//` or `/* */` comment that is annotated with `+` in the leftmost column is considered to be added code. A `//` or `/* */` comment without such an annotation is considered to be a comment about the SmPL code, and thus is not matched in the C code.

The following preprocessor directives can likewise be added. They cannot be matched against. The entire line is added, but it is not parsed.

- `if`
- `ifdef`
- `ifndef`
- `else`
- `elif`
- `endif`
- `error`
- `line`

12 Command-line semantic match

It is possible to specify a semantic match on the spatch command line, using the argument `-sp`. In such a semantic match, any token beginning with a capital letter is assumed to be a metavariable of type `metavariable`. In this case, the parser must be able to figure out what kind of metavariable it is. It is also possible to specify the type of a metavariable by enclosing the type in `:`'s, concatenated directly to the metavariable name.

Some examples of semantic matches that can be given as an argument to `-sp` are as follows:

- `f (e) :` This only matches the expression `f (e)`.
- `f (E) :` This matches a call to `f` with any argument.
- `F (E) :` This gives a parse error; the semantic patch parser cannot figure out what kind of metavariable `F` is.
- `F:identifier:(E) :` This matches any one argument function call.
- `f:identifier:(e:struct foo *):` This matches any one argument function call where the argument has type `struct foo *`. Since the types of the metavariables are specified, it is not necessary for the metavariable names to begin with a capital letter.
- `F:identifier:(F) :` This matches any one argument function call where the argument is the name of the function itself. This example shows that it is not necessary to repeat the metavariable type name.
- `F:identifier:(F:identifier):` This matches any one argument function call where the argument is the name of the function itself. This example shows that it is possible to repeat the metavariable type name.

When constraints, *e.g.* `when != e`, are allowed but the expression `e` must be represented as a single token. The generated semantic match behaves as though there were a `*` in front of every token.

13 Iteration

It is possible to iterate Coccinelle, giving the subsequent iterations a different set of virtual rules or virtual identifier bindings. An example is found in `demos/iteration.cocci`. The example shown there is as follows:

```
virtual after_start

@initialize:ocaml@

let tbl = Hashtbl.create(100)

let add_if_not_present from f file =
try let _ = Hashtbl.find tbl (f,file) in ()
with Not_found ->
  Hashtbl.add tbl (f,file) file;
  let it = new iteration() in
  (match file with
   Some fl -> it#set_files [fl]
  | None -> ());
  it#add_virtual_rule After_start;
  it#add_virtual_identifier Err_ptr_function f;
  it#register()
```

The virtual rule `after_start` is used to distinguish between the first iteration (in which it is not considered to have matched) and all others. This is done by not mentioning `after_start` in the command line, but adding it on each iteration.

The main code for performing the iteration is found in the function `add_if_not_present`, between the lines calling `new iteration` and `register`. `new iteration` creates a structure representing the new iteration. `set_files` sets the list of files to be considered on the new iteration. If this function is not called, the new iteration treats the same files as the current iteration. `add_virtual_rule A` has the same effect as putting `-D a` on the command line. Note that the first letter of the rule name is capitalized, although this is not done elsewhere. `add_virtual_identifier X v` has the same effect as putting `-D x=v` on the command line. Note again the case change. `extend_virtual_identifiers()` (not shown) preserves all virtual identifiers of the current iteration that are not overridden by calls to `add_virtual_identifier`. Finally, the call to `register` queues the collected information to trigger a new iteration at some time in the future.

Modification is not allowed when using iteration. Thus, it is required to use the `-no-show-diff`, unless the semantic patch contains `*s` (a semantic match rather than a semantic patch).

The remainder of the code above uses a hash table to ensure that the same information is not enqueued more than once. Coccinelle itself provides no support for this.

14 Examples

This section presents a range of examples. Each example is presented along with some C code to which it is applied. The description explains the rules and the matching process.

14.1 Function renaming

One of the primary goals of Coccinelle is to perform software evolution. For instance, Coccinelle could be used to perform function renaming. In the following example, every occurrence of a call to the function `foo` is replaced by a call to the function `bar`.

Before	Semantic patch	After
<pre>1 #DEFINE TEST "foo" 2 3 printf("foo"); 4 5 int main(int i) { 6 //Test 7 int k = foo(); 8 9 if(1) { 10 foo(); 11 } else { 12 foo(); 13 } 14 15 foo(); 16 }</pre>	<pre>1 @@ 2 3 @@ 4 5 6 - foo() 7 + bar()</pre>	<pre>1 #DEFINE TEST "foo" 2 3 printf("foo"); 4 5 int main(int i) { 6 //Test 7 int k = bar(); 8 9 if(1) { 10 bar(); 11 } else { 12 bar(); 13 } 14 15 bar(); 16 }</pre>

14.2 Removing a function argument

Another important kind of evolution is the introduction or deletion of a function argument. In the following example, the rule `rule1` looks for definitions of functions having return type `irqreturn_t` and two parameters. A second *anonymous* rule then looks for calls to the previously matched functions that have three arguments. The third argument is then removed to correspond to the new function prototype.

```
1 @ rule1 @
2 identifier fn;
3 identifier irq, dev_id;
4 typedef irqreturn_t;
5 @@
6
7 static irqreturn_t fn (int irq, void *dev_id)
8 {
9     ...
10 }
11
12 @@
13 identifier rule1.fn;
14 expression E1, E2, E3;
15 @@
16
17 fn(E1, E2
18 - ,E3
19 )
```

drivers/atm/firestream.c at line 1653 before transformation

```
1 static void fs_poll (unsigned long data)
2 {
3     struct fs_dev *dev = (struct fs_dev *) data;
4
5     fs_irq (0, dev, NULL);
6     dev->timer.expires = jiffies + FS_POLL_FREQ;
7     add_timer (&dev->timer);
8 }
```

drivers/atm/firestream.c at line 1653 after transformation

```
1 static void fs_poll (unsigned long data)
2 {
3     struct fs_dev *dev = (struct fs_dev *) data;
4
5     fs_irq (0, dev);
6     dev->timer.expires = jiffies + FS_POLL_FREQ;
7     add_timer (&dev->timer);
8 }
```

14.3 Introduction of a macro

To avoid code duplication or error prone code, the kernel provides macros such as `BUG_ON`, `DIV_ROUND_UP` and `FIELD_SIZE`. In these cases, the semantic patches look for the old code pattern and replace it by the new code.

A semantic patch to introduce uses of the `DIV_ROUND_UP` macro looks for the corresponding expression, *i.e.*, $(n + d - 1) / d$. When some code is matched, the metavariables `n` and `d` are bound to their corresponding expressions. Finally, Coccinelle rewrites the code with the `DIV_ROUND_UP` macro using the values bound to `n` and `d`, as illustrated in the patch that follows.

Semantic patch to introduce uses of the `DIV_ROUND_UP` macro

```
1 @ haskernel @
2 @@
3
4 #include <linux/kernel.h>
5
6 @ depends on haskernel @
7 expression n,d;
8 @@
9
10 (
11 - ((n) + (d)) - 1) / (d))
12 + DIV_ROUND_UP(n,d)
13 |
14 - ((n) + ((d) - 1)) / (d))
15 + DIV_ROUND_UP(n,d)
16 )
```

Example of a generated patch hunk

```
1 --- a/drivers/atm/horizon.c
2 +++ b/drivers/atm/horizon.c
3 @@ -698,7 +698,7 @@ got_it:
4         if (bits)
5             *bits = (div<<CLOCK_SELECT_SHIFT) | (pre-1);
6         if (actual) {
7 -             *actual = (br + (pre<<div) - 1) / (pre<<div);
8 +             *actual = DIV_ROUND_UP(br, pre<<div);
9             PRINTD (DBG_QOS, "actual_rate:_%u", *actual);
10        }
11        return 0;
```

The `BUG_ON` macro makes a assertion about the value of an expression. However, because some parts of the kernel define `BUG_ON` to be the empty statement when debugging is not wanted, care must be taken when the asserted expression may have some side-effects, as is the case of a function call. Thus, we create a rule introducing `BUG_ON` only in the case when the asserted expression does not perform a function call.

On particular piece of code that has the form of a function call is a use of `unlikely`, which informs the compiler that a particular expression is unlikely to be true. In this case, because `unlikely` does not perform any side effects, it is safe to use `BUG_ON`. The second rule takes care of this case. It furthermore disables the isomorphism that allows a call to `unlikely` be replaced with its argument, as then the second rule would be the same as the first one.

```

1 @@
2 expression E,f;
3 @@
4
5 (
6   if (<+... f(...) ...+>) { BUG(); }
7 |
8 - if (E) { BUG(); }
9 + BUG_ON(E);
10 )
11
12 @ disable unlikely @
13 expression E,f;
14 @@
15
16 (
17   if (<+... f(...) ...+>) { BUG(); }
18 |
19 - if (unlikely(E)) { BUG(); }
20 + BUG_ON(E);
21 )

```

For instance, using the semantic patch above, Coccinelle generates patches like the following one.

```

1 --- a/fs/ext3/balloc.c
2 +++ b/fs/ext3/balloc.c
3 @@ -232,8 +232,7 @@ restart:
4         prev = rsv;
5     }
6     printk("Window_map_complete.\n");
7 -     if (bad)
8 -         BUG();
9 +     BUG_ON(bad);
10 }
11 #define rsv_window_dump(root, verbose) \
12     __rsv_window_dump((root), (verbose), __FUNCTION__)

```

14.4 Look for NULL dereference

This SmPL match looks for NULL dereferences. Once an expression has been compared to NULL, a dereference to this expression is prohibited unless the pointer variable is reassigned.

Original

```
1 foo = kmalloc(1024);
2 if (!foo) {
3     printk ("Error_%s", foo->here);
4     return;
5 }
6 foo->ok = 1;
7 return;
```

Semantic match

```
1 @@
2 expression E, E1;
3 identifier f;
4 statement S1,S2,S3;
5 @@
6
7 * if (E == NULL)
8 {
9     ... when != if (E == NULL) S1 else S2
10    when != E = E1
11 * E->f
12    ... when any
13    return ...;
14 }
15 else S3
```

Matched lines

```
1 foo = kmalloc(1024);
2 if (!foo) {
3     printk ("Error %s", foo->here);
4     return;
5 }
6 foo->ok = 1;
7 return;
```

14.5 Reference counter: the of_xxx API

Coccinelle can embed Python code. Python code is used inside special SmPL rule annotated with `script:python`. Python rules inherit metavariables, such as identifier or token positions, from other SmPL rules. The inherited metavariables can then be manipulated by Python code.

The following semantic match looks for a call to the `of_find_node_by_name` function. This call increments a counter which must be decremented to release the resource. Then, when there is no call to `of_node_put`, no new assignment to the `device_node` variable `n` and a `return` statement is reached, a bug is detected and the position `p1` and `p2` are initialized. As the Python only depends on the positions `p1` and `p2`, it is evaluated. In the following case, some emacs Org mode data are produced. This example illustrates the various fields that can be accessed in the Python code from a position variable.

```
1 @ r exists @
2 local idexpression struct device_node *n;
3 position p1, p2;
4 statement S1,S2;
5 expression E,E1;
6 @@
7
8 (
9 if (!(n@p1 = of_find_node_by_name(...))) S1
10 |
11 n@p1 = of_find_node_by_name(...)
12 )
13 <... when != of_node_put(n)
14     when != if (...) { <+... of_node_put(n) ...+> }
15     when != true !n || ...
16     when != n = E
17     when != E = n
18 if (!n || ...) S2
19 ...>
20 (
21     return <+...n...+>;
22 |
23 return@p2 ...;
24 |
25 n = E1
26 |
27 E1 = n
28 )
29
30 @ script:python @
31 p1 << r.p1;
32 p2 << r.p2;
33 @@
34
35 print "* TODO [[view:%s::face=ovl-face1::linb=%s::colb=%s::cole=%s][inc.
    counter:%s::%s]]" % (p1[0].file,p1[0].line,p1[0].column,p1[0].column_end,
    p1[0].file,p1[0].line)
36 print "[[view:%s::face=ovl-face2::linb=%s::colb=%s::cole=%s][return]]" % (p2
    [0].file,p2[0].line,p2[0].column,p2[0].column_end)
```

Lines 13 to 17 list a variety of constructs that should not appear between a call to `of_find_node_by_name` and a buggy return site. Examples are a call to `of_node_put` (line 13) and a transition into the then branch of a conditional testing whether `n` is `NULL` (line 15). Any number of conditionals testing whether `n` is `NULL` are allowed as indicated by the use of a nest `<...>` to describe the path between the call to `of_find_node_by_name`, the return and the conditional in the pattern on line 18.

The previously semantic match has been used to generate the following lines. They may be edited using the emacs Org mode to navigate in the code from a site to another.

```

1 * TODO [[view:/linux-next/arch/powerpc/platforms/pseries/setup.c::face=ovl-
    face1::linb=236::colb=18::cole=20][inc. counter:/linux-next/arch/powerpc/
    platforms/pseries/setup.c::236]]
2 [[view:/linux-next/arch/powerpc/platforms/pseries/setup.c::face=ovl-face2::
    linb=250::colb=3::cole=9][return]]
3 * TODO [[view:/linux-next/arch/powerpc/platforms/pseries/setup.c::face=ovl-
    face1::linb=236::colb=18::cole=20][inc. counter:/linux-next/arch/powerpc/
    platforms/pseries/setup.c::236]]
4 [[view:/linux-next/arch/powerpc/platforms/pseries/setup.c::face=ovl-face2::
    linb=245::colb=3::cole=9][return]]

```

Note : Coccinelle provides some predefined Python functions, *i.e.*, `cocci.print_main`, `cocci.print_sec` and `cocci.print_secs`. One could alternatively write the following SmPL rule instead of the previously presented one.

```

1 @ script:python @
2 p1 << r.p1;
3 p2 << r.p2;
4 @@
5
6 cocci.print_main(p1)
7 cocci.print_sec(p2, "return")

```

The function `cocci.print_secs` is used when there is several positions which are matched by a single position variable and that every matched position should be printed.

Any metavariable could be inherited in the Python code. However, accessible fields are not currently equally supported among them.

14.6 Filtering identifiers, declarers or iterators with regular expression

If you consider the following SmPL file which uses the regexp functionality to filter the identifiers that contain, begin or end by foo,

```
1 @anyid@
2 type t;
3 identifier id;
4 @@
5 t id () {...}
6
7 @script:python@
8 x << anyid.id;
9 @@
10 print "Identifier: %s" % x
11
12 @contains@
13 type t;
14 identifier foo =~ ".*foo";
15 @@
16 t foo () {...}
17
18 @script:python@
19 x << contains.foo;
20 @@
21 print "Contains foo: %s" % x
```

```
23 @endsby@
24 type t;
25 identifier foo =~ ".*foo$";
26 @@
27
28 t foo () {...}
29
30 @script:python@
31 x << endsby.foo;
32 @@
33 print "Ends by foo: %s" % x
34
35 @beginsby@
36 type t;
37 identifier foo =~ "^foo";
38 @@
39 t foo () {...}
40
41 @script:python@
42 x << beginsby.foo;
43 @@
44 print "Begins by foo: %s" % x
```

and the following C program, on the left, which defines the functions foo, bar, foobar, barfoobar and barfoo, you will get the result on the right.

```
1 int foo () { return 0; }
2 int bar () { return 0; }
3 int foobar () { return 0; }
4 int barfoobar () { return 0; }
5 int barfoo () { return 0; }
```

```
1 Identifier: foo
2 Identifier: bar
3 Identifier: foobar
4 Identifier: barfoobar
5 Identifier: barfoo
6 Contains foo: foo
7 Contains foo: foobar
8 Contains foo: barfoobar
9 Contains foo: barfoo
10 Ends by foo: foo
11 Ends by foo: barfoo
12 Begins by foo: foo
13 Begins by foo: foobar
```

15 Tips and Tricks

15.1 How to remove useless parentheses?

If you want to rewrite any access to a pointer value by a function call, you may use the following semantic patch.

```
1 - a = *b
2 + a = readb(b)
```

However, if for some reason your code looks like `bar = *(foo)`, you will end up with `bar = readb((foo))` as the extra parentheses around `foo` are captured by the metavariable `b`.

In order to generate better output code, you can use the following semantic patch instead.

```
1 - a = *(b)
2 + a = readb(b)
```

And rely on your `standard.iso` isomorphism file which should contain:

```
1 Expression
2 @ paren @
3 expression E;
4 @@
5
6 (E) => E
```

Coccinelle will then consider `bar = *(foo)` as equivalent to `bar = *foo` (but not the other way around) and capture both. Finally, it will generate `bar = readb(foo)` as expected.