C source-to-source compiler enhancement from within

Jens Gustedt

C source-to-source compiler enhancement from within

Jens Gustedt

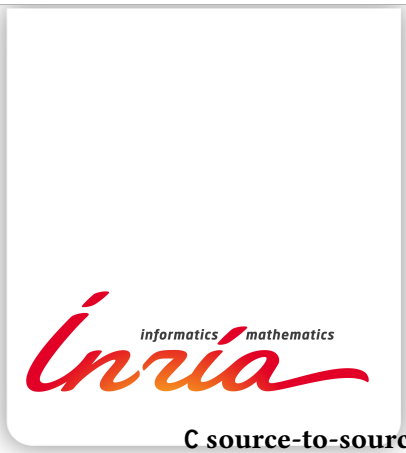
RESEARCH REPORT

N° 9375

November 2020

Project-Team Camus

ISSN 0249-6399 ISRN INRIA/RR--9375--FR+ENG



C source-to-source compiler enhancement from within

Jens Gustedt

Project-Team Camus

Research Report n° 9375 — November 2020 — 19 pages

Abstract: We show how locally replaceable code snippets can be used to easily specify and prototype compiler and language enhancements for the C language that work by local source-to-source transformation. A toolbox implements the feature and provides many directives that can be used for compile time configuration and tuning, code unrolling, compile time expression evaluation and program modularization. The tool is also easily extensible by simple filters that can be programmed with any suitable text processing framework.

Key-words: C programming language, source-to-source compilation, code unrolling, modularity of programs

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Amélioration source-à-source de compilateurs C

Résumé : Nous montrons comment un remplacement local source-à-source de fragments de code C peut être utilisé pour une spécification et un prototypage facile d'extensions de compilateurs ou de languages. Une boîte à outil implémente cet approche et fournit déjà beaucoup de directives qui peuvent être utilisées pour la configuration et la mise au point pendant la compilation, pour le déroulement de code, pour l'évaluation d'expressions et pour la modularisation de programmes. L'outil est aussi extensible par des filtres simples. Ils peuvent être codés avec n'importe quel mécanisme de transformation de texte.

Mots-clés : langage de programmation C, compilation source-à-source, déroulement de code, modularité de programmes

Contents

1 INTRODUCTION

For several generations of programmers now, the C programming is in the top tier of used programming languages [BV 2020]. It is under slow but constant development, the latest ISO standard is C17 [JTC1/SC22/WG14 2018] and a new version is under development by ISO JC1/SC22/WG14 [Jones and Gustedt 2020]. Configuration, tuning, extension and instantiation of programs written in C is not part of the language standard itself and besides feature test macros the language provides no tools that could easily be used or extended for these tasks. Therefore they are nowadays performed either manually or by a large variety of tools, and no clear dominant tool among these has ever emerged.

Shell scripts are used for configuration and compile time tuning (ATLAS [ATLAS 2018]), domain specific languages (DSL) are used for target code generation (*e.g* stencils [Henretty et al. 2013; Tang et al. 2011]), macro packages provide features that are merely language extensions (boost [Schäling 2020], P99 [Gustedt 2012]), configuration tools provide system information at compile time (POSIX' `getconf` [Open Group 2018], `cmake` [cmake 2020], GNU autotools [autotools 2020]), **`#pragma`** directives extend C to cope with parallelism and vectorization (*e.g.*

OpenMp [OpenMP 2018]) or polyhedral transformations (*e.g* OpenScop [Bastoul 2014]) as do language extensions such as Cilk [Leiserson and Plaat 1997].

We provide a tool (called `shnell`) to enhance `C` code by means of **`#pragma`** directives that has the potential to replace many of the above techniques and tools. It proceeds as a local text-replacement filter, namely the programmer marks parts or all of their code with a start pragma and specifies a transformation as arguments to that pragma. `shnell` proceeds as follows:

(1) It cuts out a marked code snippet.
(2) It pipes the snippet through a textual transformation filter (called a *directive*) that is named with the pragma.
(3) It splices the resulting snippet back into the same location.

Many small but convenient directives are already available and testify the simplicity of their use and creation:

- code unrolling (**`do`**, **`foreach`**, **`specialize`**, **`ranges`**),
- configuration
  - expansion of environment variables (**`env`**)
  - expansion of POSIX' **`getconf`** variables,
- compile time evaluation of expressions (**`compute`**, **`bc`**),
- structured identifier export (**`export`**)
- implicit import of library features (**`implicit`**).

The implementation is based on two major tools that are available on POSIX systems: namely shell programming (`sh`) and regular expression streaming (`sed`), but conceptually the approach is transferable to any modern system. In any case, the existing features can be used in daily programming without knowledge of these tools.

Filter programs that implement new directives can be written in any other programming language that suits the task, `Perl`, `python`, `java`, `C` itself, `Lisp` ..., or any other programming language that allows to write a text filter that receives the source snippet on **`stdin`** and writes an augmented version to **`stdout`**.

`shnell` features are easily applied either by explicit code transformation by the tool `shnell` itself or by using compiler a command line prefix to the compiler tool chain such as `shneller`, `trade`, or `posix`.

2 DESCRIPTION

2.1 Simple examples

`shnell` performs source-to-source transformations after identifying code ranges by means of **`#pragma`** directives. A declaration of an array `A` could for example be coded as follows:

```
double A[] = {
#pragma CMOD amend foreach ANIMAL:N = goose dog cat
   [${ANIMAL}] = 2*${N},
#pragma CMOD done
};
```

The goal is to generate repetitive code statically at compile time. The first **`#pragma`**, **amend**, indicates the start and the second, **done**, the end of a code snippet that is to be modified, here this is just one line of code that contains two *meta-variables*, `${ANIMAL}` and `${N}`:

```
   [${ANIMAL}] = 2*${N},
```

The arguments to the **amend** rule are

```
        foreach ANIMAL:N = goose dog cat
```

They specify that the inner line is copied three times, meta-variable `${ANIMAL}` iterates over the tokens `goose`, `dog` and `cat`, and meta-variable `${N}` holds the number of the current copy, starting at `0` for `goose`, `1` for `dog` and so on. So the resulting code replacement is

```
   [goose] = 2*0,
   [dog] = 2*1,
   [cat] = 2*2,
```

The overall code now is a completely expanded `C` source.

```
double A[] = {
   [goose] = 2*0,
   [dog] = 2*1,
   [cat] = 2*2,
};
```

Another provided feature is stringification. A code similar to the above can use the `"stringified"` parameters

```
char const* names[] = {
```

```
#pragma CMOD amend foreach ANIMAL = goose dog cat
   [${ANIMAL}] = #${ANIMAL},
#pragma CMOD done
};
```

Similar to C preprocessor expansion and single **#** before a meta-variable produces *stringified* expansion of the textual value of the variable. The replacement code looks as follows

```
char const* names[] = {
   [goose] = "goose",
   [dog] = "dog",
   [cat] = "cat",
};
```

2.2 The general approach

In the general case such a `shnell` pragma is identified with a *tag*, **CMOD** in the example. It is then followed by a *rule*, here **amend**, to indicate in this case that the code up to the next **done** is modified. The rule handles a *directive*, here **foreach**, that names the script that is to be run and with a *list of arguments* to the directive, here the five tokens `ANIMAL = goose dog cat`.

All three features of such a pragma (tag, rule and directive) can be specified or adapted to particular needs. A *tag* specifies the only lexical unity that `shnell` imposes. Once such a tag has been introduced (see below), `shnell` processes pragmas with these tags and expects them to match. A *rule* indicates which part (if any) of the following program text is to be treated by `shnell`. There are four predefined rules (**amend**, **done**, **insert**, **load**), but other rules can be introduced as shortcuts for pairs of rules and directives. A *directive* indicates the filter program that is to be run over the code snippet, if any. Such a filter program receives the list of arguments and is free to interpret them as a specification of the action that is to be taken.

3 COMMAND LINE TOOLS

Several command line tools are provide to ease the usage of the tool, depending for example if replacement code has to be conserved for a project or if it is just considered to be a short time auxiliary that should be removed after each compilation.

3.1 `shnell`

The central tool is `shnell`, which reads a C file, processes it and writes the result to **stdout**.

This should be used for projects that want to keep track of the intermediate code. It is easy to integrate into a compilation chain by inventing a new file extension for `shnell`-annotated C code code, for example `.cs` and then by providing a transformation rule for `make`.

3.2 Executable dialects

To avoid to keep track of the modified sources and to apply the same set of directives to each source file, *dialects* can be defined by means of `.shnl` files that group directives together. Dialects can be loaded with a **load** rule, or a *compiler prefix* can be used to apply such a dialect to a source and to run a compiler directly on the result.

There are some predefined dialects and prefixes, for example `shneller` as a generic prefix without specific dialect or trade as a prefix that provides a simple import/export feature. Such an executable prefix can easily be installed as soft-link to the `shneller` executable. For example if the link name is `trade` the script searches for a dialect `TRADE` (the link name in all caps) and loads it prior to processing the source.

As an example, the following applies the `TRADE` policy to a source file `toto.c` during compilation

```
trade gcc -Wall -c -O3 -march=native toto.c
```

That is we prefix the compiler command line by the command `trade`. The script then filters file names and the task to perform from the command line, performs the source-to-source rewriting and then compiles the result to an object file `toto.o`.

Similarly, without the `-c` command-line argument

```
trade gcc -Wall -O3 -march=native toto.c mind.o mund.o
```

it takes the first source (`toto.c`), performs the actions as described above and then links all the objects into an executable `toto` if possible.

If there are only `.o` files only the linker phase is performed.

```
    trade gcc -Wall -O3 -march=native toto.o mind.o mund.o
```

The `shneller` script and all prefixes that are derived from it understands the usual command line flags

- **-c** compile to object `.o` file
- **-E** perform all rewriting and preprocessing
- **-S** produce an assembler file
- **-M** produce nothing but the side effects of compilation such as a synthetic header file (see export)

-o *name* write the result to file *name*.

All other command-line arguments are forwarded to the compiler.

4 DIRECTIVES, C PROGRAMMER VIEW

Directives are found by one or several **scans** that `shnell` performs on a source. They are interpreted as to three different rules, that can induce no, one or several additional scans.

4.1 amend

The scope of an **amend** rule is up to the next (nesting) **done** or to the end of the source file if no such **done** is found. As mentioned above, the identified code is piped into the filter that corresponds to the directive and the result is inserted in place. The filter also receives the argument list of the directive over some side channel. As we have seen for the **foreach** directive above, typically an **amend** rule modifies the code that it receives.

4.2 Recursion

Nested occurrence of **amend** directives leads to finite recursion. The two nested **do** directives in the following are replaced by initializations for 6 elements.

```
double A[3][2] = {
#pragma CMOD amend do I = 3
 [${I}] = {
#pragma CMOD amend do J = 2
  [${J}] = 2*${I} + ${J},
#pragma CMOD done
 },
#pragma CMOD done
};
```

These are the processed with the following steps.

(1) Start collecting the code S_I immediately after the first **do**.
(2) When collecting S_I, the second **do** directive is encountered.
(3) Collection of the code S_J after that second **do** is started, until the first **done** is encountered. S_J now has:

```
[${J}] = 2*${I} + ${J},
```

(4) S_J is fed into the **do** directive for variable `J` and value `2`.
(5) The **do** directive replicates S_J twice and replaces all occurrences of `${J}` by `0` and `1`, respectively, to obtain a code T_J.

```
[0] = 2*${I} + 0,
[1] = 2*${I} + 1,
```

(6) T_J is inserted into S_I instead of the directive, resulting in a replaced code R_I.
(7) The scan for the first directive is continued until the second **done** is encountered, resulting in a code Q_I.

```
[${I}] = {
   [0] = 2*${I} + 0,
   [1] = 2*${I} + 1,
},
```

(8) Q_I is fed into the **do** directive for variable `I` and value `3`.
(9) The **do** directive replicates Q_I three times and replaces all occurrences of `${I}` by `0`, `1`, and `2`, respectively, to obtain a code T_I.

```
[0] = {
   [0] = 2*0 + 0,
   [1] = 2*0 + 1,
},
[1] = {
   [0] = 2*1 + 0,
   [1] = 2*1 + 1,
},
[2] = {
   [0] = 2*2 + 0,
   [1] = 2*2 + 1,
},
```

(10) T_I is then inserted in place of the whole **#pragma** construct.

So after completion of the **inner** directive, after step 5, the code as if we had written:

```
double A[3][2] = {
#pragma CMOD amend do I = 3
      [${I}] = {
        [0] = 2*${I} + 0,
        [1] = 2*${I} + 1,
      },
#pragma CMOD done
};
```

Only then the outer directive is applied and the over all result after step 10 is

```
double A[3][2] = {
  [0] = {
    [0] = 2*0 + 0,
    [1] = 2*0 + 1,
  },
  [1] = {
    [0] = 2*1 + 0,
    [1] = 2*1 + 1,
  },
  [2] = {
    [0] = 2*2 + 0,
    [1] = 2*2 + 1,
  },
};
```

4.3 insert

In contrast to that, an **insert** rule has no scope and the command that is specified by the directive does not receive input, but only the arguments. The output of the command is inserted in place. The scan of the source file then continues directly after the inserted code. So the inserted code has no further influence on `shnell` for the current scan.

Typically, an **insert** rule just puts some declarations or definitions in place. An example for the **insert** rule is the **enum** directive. It defines an enumeration type and some depending functions. For example

```
#pragma CMOD insert enum animal goose dog cat
```

defines an enumeration `animal` with three elements and some convenience functions such as a function `animal_names` which returns a string representation of an `animal` value.

4.4 load

A **load** rule is usually a bit more sophisticated than an **insert**. It inserts a set of directives that are found in an `.shnl` file and then the scanning continues from the **top** of the inserted lines.

An `.shnl` file may comprise any number of `shnell` directives and in particular several **amend** rules. Thereby it can result multiple scans of the parts or the whole source file.

An example for such a directive is **CONSTEXPR**. It allows to have several nested evaluations of meta-variables and thereby to use meta-variables also as the arguments to other `shnell` directives. For the example below, the effect is just that two scans are done one after another, namely first for the tag **VAR0** and then for the tag **CONSTEXPR**.

The example uses two environment variables `DIM` and `DUM` to configure a two-dimensional matrix, including an initializer.

```
#pragma CMOD load CONSTEXPR
// Use environment variables with default values
#pragma VAR0 env DUM=DIM:-4 DIM=DUM:-4

double A[${DIM}][${DUM}] = {
#pragma CONSTEXPR do I ${DIM}
 [${I}] = {
#pragma CONSTEXPR do J ${DUM}
           [${J}] = ${DUM}*${I}+${J},
#pragma CONSTEXPR done
 },
#pragma CONSTEXPR done
};
#pragma VAR0 done
```

In a first scan for **VAR0** tags the meta-variables `DIM` and `DUM` are replaced by the values that are received from the environment. Then, in a second scan all pragmas with tag **CONSTEXPR** then are processed. With environment variables `DIM` and `DUM` set to 2 and 3, respectively, the resulting code is as presented here:

```
double A[3][2] = {
    [0] = {
        [0] = 2*0+0,
        [1] = 2*0+1,
    },
    [1] = {
        [0] = 2*1+0,
        [1] = 2*1+1,
    },
    [2] = {
        [0] = 2*2+0,
        [1] = 2*2+1,
    },
};
```

4.5 Arguments to directives and meta-variables

As we have seen, several constructs define and use meta-variables of the form `${NAME}`. This is for example the case for **do**, **foreach**, **env**, **bind**, and **let**. The replacement by the text that these variables contain can be modified with **#** and **##** operators, similar to what happens in the C preprocessor.

- **#** is stringification as already described above,
- **##** merges a meta-variable to a token to the left or to the right.

For example the following two declarations result in different expansions.

```
#pragma CMOD amend bind L=int
   unsigned ${L} A;
   unsigned ${L} ## A;
#pragma CMOD done
```

$\Longrightarrow$

```
unsigned int A;
unsigned intA;
```

4.6 `amend.cfg`: a list of approved directives

For security reasons, `shnell` does not allow arbitrary code to be executed. Otherwise, a malign code that buries some directives deep down in some seemingly unimportant code or library header could be easily used as an attack vector to comprise any executable that is compiled with it. Therefore, directives and `.shnl` files have to be approved. This can be achieved by installing them into a specific directory and by adding an entry to the configuration file "`amend.cfg`".

5 A TOOLBOX OF DIRECTIVES

Our approach of selecting code snippets and transforming them by external programs has leads us to implement a wide variety of directives. Most of them are by themselves neither difficult to implement nor to understand, but together they form a quite powerful toolbox. Basically they comprise tasks that often are done more or less mechanically times.

- Configuring code according to external information.
- Copy-pasting code snippets and specializing them for some value.
- Query-replacing a set of identifiers by identifiers that comply to a naming convention.
- Manually evaluating expressions that are composed of compile time constants.
- Filtering code to collect a set of used identifiers.
- Inspecting an binary file to collect all external identifiers.

The following discussion introduces these directives with the goal to show how `shnell` provides the right abstraction to implement them.

One set of tools is then much more sophisticated, but hopefully still easy to use, namely im- and export of identifiers from translation units. This provides an example where `shnell` is used to program a more complex task.

As noted above, per default `shnell` always inserts source-line information in the generated code. In the following, this information is omitted to keep the snippets readable for the human eye.

5.1 Variable binding and configuration

A first important group of applications of `shnell` is compile time tuning and configuration, which as of today is either done by outside tools (such as `cmake`) or by cascades of **`#ifdef`** queries on feature test macros. Our tool unifies these approaches because it allows the programmer to perform simple compile time computations and to access external information such as environment variables or platform specific configuration functions. For the latter we currently provide access to `C`' **`getenv`** and `POSIX`' **`getconf`** functions, but adapting this approach to other platforms, operating systems or application specific utilities should not be a problem.

5.1.1 **`bind`**. This directive expects arguments in the form

```
bind NAME0=VALUE0 [TOK0_0 ...] NAME1=VALUE1 [TOK1_0 ...] ...
```

(without special characters) and replaces all ${NAME0} in the code snippet by the group of words VALUE0 TOK0_0 ...etc. All tokens in the argument that do not contain an = sign are collected into a list of value tokens for NAME0 etc. A token on the line that contains an = sign starts the list for the next meta-variable NAME1 etc.

5.1.2 **let**. This directive is a first primitive for compile time evaluation of expressions.

```
amend let NAME = EXPRESSION
```

the remaining substitution is then done as if we had

```
amend bind NAME=VALUE
```

where VALUE is the value is computed by /bin/sh from EXPRESSION.

The expression may use the usual suspects from the POSIX `test` utility such as `-lt` for *less than*. Other operators such as `-ls` (for *left shift*) and similar operators are provided as an extension to that scheme, as are Unicode characters (e.g.. ≤) for such operations.

5.1.3 **env**. The **env** directive is based on the C library function **getenv** and therefore independent of the platform.

```
amend env [NAME0=]VARIABLE0[:DEFAULT0] [NAME1=]VARIABLE1[:DEFAULT] ...
```

Replace meta-variables ${NAMEx} (${NAME0}, ${NAME1} ...) by the values of environment variables VARIABLEx. If NAMEx= is omitted, the name of the meta-variable is the same as the environment variable.

Depending on the status of VARIABLEx, DEFAULTx can be used to control the returned value. The syntax is borrowed from the POSIX shell for parameter expansion. Therefore DEFAULTx must start with one of the characters `-`, `+` or `?`.[1]

-VALUEx sets VALUEx as default value if the environment variable is unset or empty.

+VALUEx provides an alternate value VALUEx, but only if the environment variable is set and not empty. Otherwise the result is empty.

?VALUEx is similar to the previous, but if the environment variable is unset or empty, the compilation is aborted.

5.1.4 **getconf**. This directive is similar to **env** only that it queries the POSIX specific configuration variables that otherwise can be queried with the POSIX function **getconf**.

```
amend getconf [NAME0=]VARIABLE0 [NAME1=]VARIABLE1 ...
```

A leading underscore in such a POSIX configuration variable can be omitted. *E.g* the POSIX variable _REGEX_VERSION can be referred to as REGEX_VERSION.

Example:

```
#pragma CMOD amend getconf FALLBACK_PATH=PATH INT_MAX
char fallback[] = #${FALLBACK_PATH};
#define MAXVALUE ${INT_MAX}
```

This should be replaced by something similar to

```
char fallback[] = "/bin:/usr/bin";
#define MAXVALUE 2147483647L
```

5.1.5 **gitID**. Other than the previous, this is an **insert** directive. As the name indicates, for the version control system `git`, it provides the identifier of the last version for the current source file.

The result is a macro with the name that is passed as argument to the directive that expands to a sequence of the bytes (as integer values) of the git hash for the source file.

5.2 Code unrolling and specialization

Other than unrolling of **for**-loops or similar constructs, unrolling of arbitrary code is rarely foreseen in modern imperative programming languages. On the other hand, replication and specialization of code is a task with which many projects are faced, and the lack of a specific feature leads to much code replication and potential copy-and-paste errors.

The potential of code unrolling encloses and extends higher level programming features such as C++'s template, because it allows to intervene within the grammar in a very fine grained way. As we already have seen above, code unrolling can be used for the generation and use of lists of all sorts, *e.g.* enumeration constants or initializers. With `shnell`, we have the opportunity to make such code unrolling easily configurable and dependent of specific features of the platform or the application.

Code unrolling also extends to specializations that are named by the programmer. This allows to force the compiler to consider variants of code snippets in which certain values are fixed or are within a given range.

[1] In contrast to the POSIX shell the = character is not allowed.

5.2.1 **foreach**. As we already have seen above, this directive instantiates the depending code snippet several times according to a list of tokens.

```
foreach NAME[:INDEX] [=] TOKEN0 TOKEN1 ...
```

For each of these instantiations the meta-variable `${NAME}` is bound to the corresponding token `TOKENx`. If `:INDEX` is also present, the meta-variable `${INDEX}` is bound to the number of the current instance, instance counts starting at `0`.

```
typdef enum coucou {
    coucou_infinity = INT_MIN,
#pragma CMOD amend foreach COUCOU:K = ka ke ki kuku
    coucou_ ## ${COUCOU} = ${K},
#pragma CMOD done
    coucou_bound,
} coucou;
```

Will generate four copies of the depending source where the appearance of the string `${COUCOU}` will be replaced by the each of the words in the list and `${K}` with the corresponding position, that is

```
typedef enum coucou {
    coucou_infinity = INT_MIN,
    coucou_ka = 0,
    coucou_ke = 1,
    coucou_ki = 2,
    coucou_kuku = 3,
    coucou_bound,
} coucou;
```

5.2.2 **do**. This repeats the code snippet for a number of times, determined by a start value, an upper or lower bound, an arithmetic operation, and a stride.

```
amend do NAME[:INDEX] [=] [START] END [[OP]STRIDE]]
```

It binds the variable `${NAME}` to specific values and generates a copy of the code snippet for each of these values. `OP` can be any of `+`, `-`, `*` or `/`, if is omitted it defaults to `+`. `STRIDE` is an integer with no sign and defaults to `1`. If `START` is also omitted it is set to `0`.

The produced sequence of numbers is checked against `END` with `<` if the sequences increases (that is for `+` and `*`) and with `>` if it decreases (that is for `-` and `/`). As for **foreach**, an optional `INDEX` meta-variable may also be used to track the current iteration number.

E.g., an array with powers of two can be obtained as follows.

```
long const power[] = {
#pragma CMOD amend do I = 1 8 *2
          ${I},
#pragma CMOD done
};
```

$\Longrightarrow$

```
long const power[] = {
    1,
    2,
    4,
};
```

5.2.3 **specialize**. For an existing `C` variable NAME this directive can be used as

```
amend specialize NAME [value1 ... ]
```

Here values should be numbers (or resolve to numbers) and admissible for comparison to the `C` variable `NAME` by the `==` operator. The special token **else** can also be used to catch all other possible values for `NAME`. This construct is more powerful than a switch statement, because in addition to integers, floating point values can also be used as values, and because the list can contain names of variables.

A copy of the code snippet enclosed in its own block is created for each value, plus a fallback version if the value else is in the list. Within each of these copies the string `${NAME}` is replaced by the actual value within that copy. In the fallback version `${NAME}` is replaced by `NAME`, the `C` variable.

If **else** is not in the list, and thus there is no fallback version, a more specific fallback version can be added after the closing pragma:

```
else {
// code for the general case
}
```

As an example, the combination of this directive with stringification of meta-variables allows to generate output that does not depend on dynamic interpretation of format strings

```
#pragma CMOD amend          \
   specialize x false true
puts("x is " #${x})
#pragma CMOD done
else {
  puts("x not a boolean")
}
```

⟹

```
if (x == false) {
    puts("x is " "false")
} else if (x == true) {
    puts("x is " "true")
}
else {
  puts("x not a boolean")
}
```

5.2.4 **ranges**. This directive is similar to **specialize** but uses comparison with <.

```
#pragma CMOD amend ranges NAME [bound1 ... ]
```

Bounds should be numeric, of constant value and admissible for a < operator, or the special token **else**. The bounds have to be be sorted with respect to that ordering and **static_assert** code is inserted to ensure this property.

```
#pragma CMOD amend ranges argc 4 8 else
  int len = (${argc} < sizeof buf ? ${argc} : sizeof buf);
  printf("this\tis specific for 'argc < " #${argc} "': %.*s\n", len, buf);
#pragma CMOD done
```

```
if (argc < 4) {
    int len = (4 < sizeof buf ? 4 : sizeof buf);
    printf("this\tis specific for 'argc < " "4" "': %.*s\n", len, buf);
} else if (argc < 8) {
    static_assert(4 < 8, "4 and 8 in wrong order");
    int len = (8 < sizeof buf ? 8 : sizeof buf);
    printf("this\tis specific for 'argc < " "8" "': %.*s\n", len, buf);
} else {
    int len = (argc < sizeof buf ? argc : sizeof buf);
    printf("this\tis specific for 'argc < " "argc" "': %.*s\n", len, buf);
}
```

5.3 Compile-time expression evaluation

To specify and implement language extensions or to embed code from a secondary language, it can be convenient to be able to identify particular environments, for example expressions, in which the feature is sought and applied. The three directives **compute**, **factor** and **bc** implement such strategies. The first two generally process all expressions that are composed of integer literals, the latter needs special identifying brackets.

Externalizing computations at compile time could be slow if the external process that is needed for the replacement has significant startup times. The **bc** directive works around that problem by starting a `bc` *server* process and by sending it the expressions that are to be processed on separate lines. For this directive there is no hard evidence, yet, that this strategy is more efficient than calling `bc` separately for each expression, but for that aspect this directive serves as a proof of concept for such a server based approach.

5.3.1 **compute** *and* **factor**. These are just prove of concepts for directives that detect expressions or integers, respectively, and modify them:

- **compute** identifies some forms of expressions within the code snippet that are only composed of integer literals, computes their value and inserts them in place.
- **factor** identifies integer literals, feeds them into the POSIX utility `factor`, and inserts the result as a multiplication of these factors into the snippet.

5.3.2 **bc**. This directive spans the whole text that it amends for expressions that are enclosed in special brackets and performs computation on them, namely by sending them to the POSIX utility `bc`.

```
amend bc [LEFTPAREN RIGHTPAREN]
```

Where `LEFTPAREN` and `RIGHTPAREN` can be any strings that help you visually distinguish processed expressions from the rest of the programming text. *E.g*

```
  #pragma CMOD amend bc [[ ]]
```

processes all expressions such as `[[ 2 ^ 9 ]]` or `[[ 4*atan(1) ]]` through `bc`, and replaces them with their value, here these are `512` and `3.14159265358979323844`. For details on the capacity of `bc` to do computations we refer to the POSIX documentation. Beware that `bc` is C-like but not completely C-conforming. In particular, it has no bit operations (`^`, `|`, `&`, `<<` and `>>`) and the `^` character is used for the exponentiation operation. So in the above example the C expression corresponding to `[[ 2 ^ 9 ]]` would have been `1 << 9`.

The spacing around such replacements should remain intact, such that you may suffix such expressions with the appropriate C number suffix that you want to give the number. E.g `[[ 2 ^ 9 ]] ULL` would result in 512ULL, the value 512 with a type of unsigned long long.

POSIX' bc is a whole programming language by itself and allows much more than just evaluating simple expressions. If we detect `;`, `{` or `}` we switch to a "complicated" processing mode that ensures that changes to the state in one processed expression do not pollute the result of other expressions. The result of such an expression should then be assigned to the bc variable `r`. Embedded bc programs may start with an **`auto`** list of variables, but bc itself already uses the letters `a`, `c`, `e`, `j`, `l`, and `s` for functions. Also we internally use the letters `o`, `p`, `q` and `r`. The *variable declaration* part starting with the keyword **`auto`** must always be at the beginning of the expression, consist of a list of variables or arrays and be terminated by a `;` token.

```
[[ scale=200 ; r=4*atan(1) ]]
```

sets the precision of computation temporarily to `200` digits and inserts the value of π with that precision into the code snippet.

```
[[ auto i; for (i=0; i < 10; i++) { r += 2*i+1; } ]]
```

is a complicated way to express the value 100.

Meta-variables from **`do`**, **`let`**, **`foreach`** and similar directives can be used if they are expand in the right order:

```
[[ auto i; for (i=0; i < ${DIM}; i++) { r += 2*i+1; } ]]
```

is a complicated way to express the value `${DIM}*${DIM}`.

Within the limits of bc' capacities (one-letter names!), also user functions can be defined. Whenever an expression starts with the word **`define`** this is considered to contain a function definition. It is passed into bc as-is and is by itself supposed to not have a return value. The text is replaced by the expression itself, so you better put all of this inside a C comment.

As mentioned above, bc already uses some letters for predefined functions, and we restrict the choice even further. So unfortunately it would not be easy implement a general function library for bc. The following lines also compute π:

```
// Add a function to the bc state:
//  [[ define q(x) { return a(x); } ]]
double const π =  [[ 4*q(1) ]] ;
```

but the replacement only removes the brackets but conserves the text in the comment

```
// Add a function to the bc state:
// define q(x) { return a(x ) ; }
double pi = 3.14159265358979323844;
```

We use the bc tool with the `-l` option to ensure that the math library is effective. This enables limited support for mathematical functions, but only **`sqrt`** is a builtin, directly usable as we know it. We translate the names of the following mathematical functions to bc's supported one character function names: **`atan`**, **`sin`**, **`cos`**, **`log`**, **`exp`**, and `jn`.

5.4 Programming shnell

Two simple principles guide all `shnell` expansions:

- The recursion principle described above ensures that nested **`CMOD`** constructs work from the inside out.
- Other than for tokenization, see below, and line numbering, `shnell` is language agnostic. It particular, `shnell` itself simply ignores any pragma with a tag that is different from **`CMOD`**.

These principles make it easy to program a second level of tools that can help to program more complicated language enhancements, namely **`eval`** and **`dialect`**. By themselves both are quite simple filters: in a first scan they perform some simple replacement of a given pragma tag by the tag **`CMOD`**, and then recursively launch `shnell` on the resulting code.

5.4.1 **`eval`**. The **`eval`** directive allows to use constructs such as ‘ ${N}‘ also as arguments to subsequent directives.

```
amend eval TAG0 [TAG1 ...]
```

In it simplest form, something like

```
#pragma CMOD amend eval HERE
```

will first replace the tag **`HERE`** in occurrences such as

```
#pragma HERE amend ...
  ...
#pragma HERE done
```

with **CMOD** and then call the expansion procedure with input the modified program part, again. Several identifiers in the line for eval have the code fragment processed as often as there are such tags, last is inner most. Consider the following example:

```
#pragma CMOD amend eval OUTER INNER
#pragma CMOD amend do I = 1 3
#pragma OUTER amend do J = 0 3 +${I}
#pragma INNER amend let VAL = ${I}*${J}+${I}
  int A ## ${I} ## _ ## ${J} = ${VAL};
#pragma INNER done
#pragma OUTER done
#pragma CMOD done
#pragma CMOD done
```

(1) The snippet for the second directive with **CMOD** is collected.

```
#pragma OUTER amend do J = 0 3 +1
#pragma INNER amend let VAL = 1*${J}+1
  int A1_ ## ${J} = ${VAL};
#pragma INNER done
#pragma OUTER done
#pragma OUTER amend do J = 0 3 +2
#pragma INNER amend let VAL = 2*${J}+2
  int A2_ ## ${J} = ${VAL};
#pragma INNER done
#pragma OUTER done
```

(2) The **eval** directive does a first scan for tag **OUTER** and replaces it with **CMOD**.

```
#pragma CMOD amend do J = 0 3 +1
#pragma INNER amend let VAL = 1*${J}+1
  int A1_ ## ${J} = ${VAL};
#pragma INNER done
#pragma CMOD done
#pragma CMOD amend do J = 0 3 +2
#pragma INNER amend let VAL = 2*${J}+2
  int A2_ ## ${J} = ${VAL};
#pragma INNER done
#pragma CMOD done
```

(3) The snippet is scanned again, and all **CMOD** directives are expanded

```
#pragma INNER amend let VAL = 1*0+1
  int A1_0 = ${VAL};
#pragma INNER done
#pragma INNER amend let VAL = 1*1+1
  int A1_1 = ${VAL};
#pragma INNER done
#pragma INNER amend let VAL = 1*2+1
  int A1_2 = ${VAL};
#pragma INNER done
#pragma INNER amend let VAL = 2*0+2
  int A2_0 = ${VAL};
#pragma INNER done
#pragma INNER amend let VAL = 2*2+2
  int A2_2 = ${VAL};
#pragma INNER done
```

(4) Finally, the replacement and scan for **INNER** sets all five occurrences of ‘ ${VAL}‘ to their computed values.

```
  int A1_0 = 1;
  int A1_1 = 2;
  int A1_2 = 3;
  int A2_0 = 2;
  int A2_2 = 6;
```

5.4.2 **dialect**. This directive allows to promote directives to rules. It expects arguments in the form

```
amend dialect DIALECTNAME                          \
        AMEND0[=[VALUE0]] AMEND1[=[VALUE1]] ...    \
     [: INSERT0[=[VALEUR0]] INSERT1[=[VALEUR1]] ...]
```

That is, the directive receives two lists, a list for amendments and one for insertions. The ":" character separates the two lists. In its code snippet, this replaces all occurrences of the forms

```
#pragma DIALECTNAME AMENDx
#pragma DIALECTNAME INSERTx
```

by

```
#pragma CMOD amend VALUEx
#pragma CMOD insert VALEURx
```

If = and VALUEx (or VALEURx) are omitted, VALUEx defaults to AMENDx or INSERTx, respectively. If just the VALUEx or VALEURx part of a pair is omitted, but an = sign is still present, AMENDx or INSERTx is disabled for this dialect. In particular, **amend=** and **insert=** disable all directives for amendments or insertions, respectively, that are not contained in the respectively lists. As an example the **CONSTEXPR** and **VAR0** dialects that had been used previously are defined as follows.

```
#pragma CMOD amend dialect CONSTEXPR  amend= insert= \
  do foreach specialize ranges factor compute bc
#pragma CMOD amend dialect VAR0  amend= insert= \
  env getconf let bind
```

That is, code is first scanned for the **VAR0** dialect. This dialect elevates four types of amendments, namely **env**, **getconf**, **let**, and **bind**, to rules and switches off all other **amend** or **insert** directives. Thus it serves to set meta-variables to certain values, either deduced from the platform or set by the program.

In a second scan, once all **VAR0** meta-variables have been expanded, the dialect **CONSTEXPR** allows to run all code unrolling and expression evaluation directives by using the expansions of these meta-variables.

5.4.3 **oneline** *and* **logicalline**. These directives allow to write code that C needs to have in a single line more comfortably. The first just concatenates all lines in the code snippet into one physical line.

As its name indicates, the second transforms the lines of the code snippet into one logical line, namely it appends spaces and backslash characters to the end of each line, but for the last line. The number of spaces that are inserted is varied on each physical line such that the backslash characters all appear in the same position.

```
#pragma CMOD amend logicalline
#define MACRO
do {
  /* something */
  } while(false)
#pragma CMOD done
```

Is replaced by

```
#define MACRO       \
do {                \
    /* something */ \
} while(false)
```

5.5 Modularization of C

Many C projects have more or less explicit strategies that specify how the project is structured into translation units and how different translation units can refer to their respective features. By such naming strategies they also avoid name clashes between different TU, in particular with third party sources that are not controlled by the same project. For example a simple name prefix of the form projectname may introduce all external identifiers, followed by a structure name, toto say. A typical external identifier then would be composed by underscores to something like projectname_toto_myfunction.

Such naming strategies can be tedious to use if the names become long and it can also be difficult to ensure that no unprotected names are overlooked and exported by a given TU.

The dialect TRADE and the directives **export** and **implicit** that compose it provides a feature set that helps to deal with such strategies more easily. At the same time they are examples for more sophisticated filters that show the possible extend to which shnell can be used.

5.5.1 **export**. This directive rewrites identifiers in the code such that they are externally seen as composed according to a well-defined naming strategy.

```
amend export [ LPREFIX ] [ : [ PREFIX0 PREFIX1 ... ] ]
```

with the following definitions:

- LPREFIX is the local name prefix. All identifiers that are prefixed with LPREFIX:: in their declarations (or that are just this LPREFIX) are considered to have external linking, unless they are static objects or functions or they are explicitly made private, see below.
- PREFIXx are identifiers that are used to compose the external name prefix.

All of these have suitable defaults and this directive is best used indirectly together with the **implicit** directive, through the TRADE dialect or the trade compiler prefix.

None of the identifiers used above must be strictly reserved, that is none of LPREFIX PREFIX0 PREFIX1 ... should start with an underscore that is followed by a capital letter or a second underscore. Any of the two parts may be empty.

- If expanded in a scan prior to this one, the alias directive can be used to set LPREFIX.
- If there is no PREFIXx, the filename is split at - characters.
- If the filename is not known, LPREFIX is used.
- If LPREFIX is empty, the last component of PREFIX0 PREFIX1 ... is used.

If neither LPREFIX or PREFIXx are given (directly or via **alias**) the filename must be known and is used to determine all naming conventions.

Linkage of identifiers

Not concerned by this tool are objects, functions and types that are just used (including tag names) but not declared or defined. In particular types that are just forward declared (such as in **struct** toto;). For them we suppose that naming issues are taken care of by whichever TU defines them.

There are several categories of local identifiers, that is identifiers that are defined in the current TU:

(1) Some globally exposed identifiers without linkage are unchanged by this tool. Per default these are struct or union members, function parameters and struct, union or enum tags that are only declared but not defined. If you want to bless them with linkage, you'd have to use one of the methods above to force it.
(2) Identifiers with internal linkage. Per default these are all global static objects or functions. Identifiers can be added to that by using the private directive.
(3) Identifiers with external linkage. Per default these are all declared global objects and functions that are not static. To that are added all other identifiers for which short or long identifier variants (see below) are used.
(4) If not made external by one of the rules above, **typedef** names, macros and enumeration constants have internal linkage. The same holds for struct, union and enum tags for types that are defined within this TU.

If an identifier could have internal or external linkage by (2) or (3), internal linkage prevails. This is so that you may make an identifier private that otherwise would have external linkage. To simplify the use of such an internal identifier you are still able to use the short form, it is rewritten to the internal form.

All identifiers that have internal linkage by these rules are "private", those that have external linkage are "public". Those with no linkage in (1) are in a gray zone, but since we force privacy on all macros, there should be no bad interaction between foreign macros and such local identifiers.

Composite identifiers:

There are several types of identifiers that are dealt by this tool:

- Local identifiers are usual global C identifiers that are defined within this TU. They have no linkage, if none of the above provided some.

Without linkage can only be members or function parameters. Variables, functions, macros, types and enumeration constants always have linkage.

- Short identifiers are composed of two parts, LPREFIX and a local identifier ID, that are joined by the ::. Declaring a local identifier that is not static or explicitly private with such a form automatically elevates it to have external linkage in the whole TU. If you prefix, e.g, an enumeration constant in its definition with the local prefix, it becomes globally visible and usable by others without creating naming conflicts. For the whole TU, the local identifier ID can be used instead of the short identifier LPREFIX::ID, and is replaced by it accordingly. Then, for the output of this tool, all short identifiers are replaced by the corresponding long identifiers.
- Long identifiers come in several flavors, for external linkage, internal linkage, and may be different as used inside the code and as they are visible to the outside. Seen from inside the TU, for external linkage an ID is prefixed with the list PREFIX0, PREFIX1 ... and the parts are joined with ::. As for short identifiers this form in the declaration of the identifiers makes ID an identifier with external linkage, and the short and local forms can be used interchangeably. To make an external symbol from a long identifier, :: is replaced by SHNELL_SEPARATOR and, unless SHNELL_SEPARATOR is "::", this is how long identifiers are seen by the outside. E.g if SHNELL_SEPARATOR is _, test::string::length is presented to the outside as test_string_length.

If SHNELL_SEPARATOR is in itself "::", the name is mangled in a strategy similar to C++, see below.

There are also long identifiers to present internal linkage to the outside. The rules are similar as for those with external linkage, only that things are added to the list of prefixes, such that the long identifier becomes obfuscated. You may use these identifiers only with their short or local form.

A unique choice of prefixes for the TU within the same project guaranties that such an identifier can never clash with another one in the same project.

As mentioned, the separator that will be used for joining external name components is SHNELL_SEPARATOR. In the case of the special token :: the output names are also mangled. Mangling is done according to the mangle shnell-module. In particular, the prefix manglePrefix is used from there. There are special conventions:

:: C++ mangling

```
_ snail_case_identifiers
C camelCaseIdentifiers
P PascalCaseIdentifiers
```

Evidently you'd have to be careful that your identifiers fit with the convention you are using. For `C` and `P` only components that start with a lower case letter and do not contain an underscore are transformed. In particular, components that start with an underscore are left alone.

This defaults to "::", but can be set by the corresponding environment variable.

```
export SHNELL_SEPARATOR="${SHNELL_SEPARATOR:-::}"
```

Generated headers:

The **`export`** also generates a header file with all exported symbols in their mangled form. That header file can be saved externally and is also appended to the source in form of a text array. Thereby users of the translation unit (*e.g* via the **`implicit`** directive) may use a `.h`, if the find it, or extract a header from a compiled `.o` object file.

main is special

As for traditional C, the entry point main can be treated specially. If you name a function `stdc::main` (which you will probably do if you also use implicit) the following strategy is applied:

- Your function is still compiled as main (without prefix) such that the compiler may apply the special treatment that is reserved for that. E.g not returning a value from the function is not an error.
- The function symbol with the local name main is made "weak" such that it does not clash with similar functions from other TU that may be linked together.
- A public symbol with the long name for main is aliased to your function.
- The header that is produced for the TU has an entry for that long name.
  All of this allows you to have one entry point per TU without creating conflicts. This is particularly useful to implement a test program for your TU in place.
  Header files To determine the identifiers that are defined in the program, a header file is produced and stored in a directory named by the `SHNELL_INCLUDE` environment variable, if any:
- Objects or functions that are declared or defined static are suppressed, unless they are static inline functions.
- Function bodies of functions that are not inline are removed and replaced by a ; that terminates the declaration.
- Function bodies of functions that are inline are kept, so they appear in the header and in the output. In the output the inline keyword is removed such that when compiling the output, the function is instantiated.
- Initializers for objects are removed, such that only the declaration remains.
- All function and object declarations that are not static are prefixed with **`extern`**. Whereas this would be the default for functions anyhow, for objects this would otherwise create "tentative definitions", that could provoke linkage problems.
- All defined identifiers (functions, objects, macros, type tags, type-names, enumeration constants) are renamed to their long form in the whole header.
- If the code defines a `stdc::main` function, the declaration of it is removed and replaced by an equivalent one using the long name for main.
- The header is protected by an include guard.
- The header file is only replaced if it had changes.

Examples:

Without mangling and a simple universal prefix `SHNELL_SEPARATOR` equal to `_` the following introduces a translation unit (TU) that uses `string` as a prefix:

```
#pragma CMOD amend export : string
```

Here, the naming convention is then independent of the filename. All global, non-static, variable and function names are externally visible to have a `string_` prefix, if they don't have one, yet. If in addition, we have three identifiers (`EMPTY`, `INIT`, and `GET`) that are forced to be public (e.g by using `string::EMPTY`, `string::INIT`, and `string::GET` in their definition) they have external names (`string_EMPTY`, `string_INIT`, and `string_GET`) but within this TU the local names (`EMPTY`, `INIT`, and `GET`) or short names may be used just the same.

One additional identifier is special, `string` itself. With the setting as described here, it is left alone. This identifier should generally be reserved for the principal feature of this TU, such as the central data type or function that is defined in the TU. It is always external.

If the directive is instead

```
#pragma CMOD amend export string : strong type
```

Again, this defines a TU where the naming convention is independent of the filename. All global, non-static, variable and function names are augmented to internally have a `strong::type::` prefix and externally to have a `strong_type_` prefix, if they don't have one, yet. The three identifiers (`EMPTY`, `INIT`, and `GET`) as above, are forced to have external names, so `strong_type_EMPTY`, `strong_type_INIT`,

and `strong_type_GET`, but within this TU the long names (`strong::type::EMPTY`, `strong::type::INIT`, and `strong::type::GET`), short names (`string::EMPTY`, `string::INIT`, and `string::GET`) and local names (`EMPTY`, `INIT`, and `GET`) may be used just the same.

Again, string itself is special. Within the TU, it is left alone, but to the outside it is visible as `strong::type`, and that name can also be used internally just as string.

This naming scheme can also be made dependent on the source filename. If that would be `strong-type.c`, the strong type part above could be omitted.

If `SHNELL_SEPARATOR` is equal to "::" the internal names are again exactly the same as above. The outside visible forms are then mangled by using the `PREFIXx` components. For the strong type example this would result in something inhumane like `_ZN2_C6strong4type5EMPTYE`, `_ZN2_C6strong4type4INITE`, and `_ZN2_C6strong4type3GETE`. Within this TU the short, long and local names (`EMPTY`, `INIT`, and `GET`) may be used just the same as above.

Two companion directives also come with **`export`**, **`private`** and **`alias`**. The first can be used to hide identifiers such as function names that otherwise would be exported. The second allows to define aliases, that are prefixes that are replaced by longer composed identifiers.

5.5.2 **`implicit`**. This directive will seek for composed identifiers and load all necessary include files to resole these names.

```
amend implicit [LPREFIX] [ : [ PREFIX0 PREFIX1 ...] ]
```

With naming conventions similar to **`export`**, this directive will seek for all composed identifiers that are neither derived from the short nor the long prefix. These composed identifiers are used to extract the names of all TU that are implicitly used. An identifier $A_0::\ldots::A_N::A_{N+1}$ that is encountered by scanning the code could come from the TU $A_0::\ldots::A_N::A_{N+1}$ (if it represents the main feature of that TU) or from $A_0::\ldots::A_N$(if it is a secondary feature of that). Both are searched in the current directory and `SHNELL_INCLUDE` if such a file exists. Then it will look for `.h` or `.o` files and add corresponding includes in front of the code. The base name of such files is formed from the TU's prefix where the separators are replaced by dashes.

So for an identifier `good::old::function`, **`implicit`** would search for include files `good-old.h` and `good-old-function.h` and assume that one of them provides a declaration for a symbol `good_old_function`.

A list of legacy pseudo-TU can be kept for which it is assumed that they use top level identifiers for their symbols. Use this feature only if you must, notably for legacy interfaces that you can't change. The `stoke` prefix for the standard C library is always included in that list. Identifiers found in such a legacy realm are translated back to that top-level form by a set of macros that is put in front of the source code. You may add legacy realms to the list by using the legacy directive in a scan of the source that precedes the scan for implicit.

It is the responsibility of the code using this feature to ensure that the symbols that are used in this way are really declared by the header file.

Interfaces of the `C` library are special because they are already subsumed in an artificial translation unit named `stdc`. A typical example would be the use of the identifier `stdc::`**`printf`** which triggers the automatic integration of an include line

```
#include "stdc.h"
```

This include file is provided and gives access to all the C library features, **`printf`** among them.

This inclusion of `C` library features is also special for another reason. The linker symbol for **`printf`** usually is not the mangled version of `stdc::`**`printf`** as it would be expected for other implicitly included translation units, but the symbol **`printf`** itself. This is evidently taken care of for the special case of `C` library symbols, but there is also general directive **`legacy`** that can be used for other external libraries that are not compiled with **`export`**.

The directive **`implicit`** also has two companion directives, namely **`startup`** and **`atexit`**. These establish startup or **`atexit`** handlers that are intended to be run before any other code that is defined in the corresponding TU, or at the end of the execution, respectively.

6 DIRECTIVES, IMPLEMENTER VIEW

Directives themselves can be written in any programming language, script language or text processing tool; the `shnell` distribution has mainly examples that are written for `sh` or `sed`, but that is not an imposed restriction.

Any program that receives the code that it has to treat on **`stdin`** and sends the modified code to **`stdout`** can be used as a directive. Additionally to the code snippet, a directive can also receive arguments via the environment variable **`CMOD_AMEND_ARGUMENTS`**.

The surrounding tasks of cutting the code out of context and reinserting the result back in place is done by `shnell`.

6.1 Tokenization

To ease the processing of `C` snippets, directives receive a *tokenized* version of the snippet on **`stdin`**. That is that a specific phase of `shnell`, called tokenizer, splits the program into tokens as they are defined by the `C` language. Thereby, all intermediate tools see isolated tokens of the categories

- identifiers
- numbers
- punctuators

- strings
- comments
- and a lot of control characters and white space.

No further lexical analysis is required. For example, the expression `a+=b` is split into three tokens "`a`", "`+=`" and "`b`" with some unspecified white-space and control characters mixed around them. The latter, are used to reconstruct the original structure of the source such that users may more easily retrieve their modified source.

Strings and comments are handled specially by that procedure. They are replaced by tokens that are not normally part of a `C` program and therefore will never match any replacement requirement that a directive may have. So directives that perform text replacement on a word base will find these words if they are used as normal `C` keywords or identifiers, but if they are used within strings or comments they are not visible for replacement.

At the end of `shnell` processing, after all transformation have been applied the tokenization is reverted once and the original line and spacing structure reappears. This should help occasional readers of transformed code to keep or inspect intermediate steps of a transformation. Additionally, per default the code is annotated with `#line` directives that name the original source lines. Thereby, compiler errors can easily be traced back to the original source file.

This tokenization phase should be easily adaptable to other programming languages than `C` if need arises.

6.2 Shell modules

The whole `shnell` toolbox is by itself constructed from reusable pieces of shell (`/bin/sh`) functions. Programmers of directives that also use the shell language can choose among a long list of tools. Tools that are provided include

- argument processing for directives
- regular expression matching (`match`),
- temporary files with garbage collection,
- split and join of text
- hash tables

and many more.

These can be used by first sourcing an `import` module.

```
    SRC="$_" . "${0%%/${0##*/}}/import.sh"
```

Then other shell modules can be imported as this:

```
import arguments
import tmpd
import tokenize
import match
```

The `import` feature also includes a tool for automatic generation of the documentation of directives. terminate such an import and documentation section by a line

```
    endPreamble $*
```

Namely, if such a directive is run by itself as an executable script with an option `--html` will then extract documentation from its own source code.

7 CONCLUSION

We presented a general strategy and its implementation in form of the `shnell` toolbox that enables code rewriting within `C` programs on the level of embedded code snippets. Its capacity to handle recursion makes it a powerful abstraction that allowed us to provide tools for different tasks, such as configuration, code unrolling or im- and export of identifiers into and from translation units.

Our implementation is primarily designed for the `C` programming language, but easily extends to other languages with a similar lexical structure.

The `shnell` project is licensed under a standard MIT license. It is distributed at

https://gustedt.gitlabpages.inria.fr/shnell/

REFERENCES

ATLAS 2018. Automatically Tuned Linear Algebra Software. http://math-atlas.sourceforge.net/.

autotools 2020. GNU Build System Combined Reference Manual. http://buildsystem-manual.sourceforge.net/index.html.

Cédric Bastoul. 2014. OpenScop. http://icps.u-strasbg.fr/ bastoul/development/openscop/docs/openscop.pdf.

Tiobe Software BV. 2020. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html monthly since 2000.

cmake 2020. `cmake`. https://www.openhub.net/p/cmake.

Jens Gustedt. 2012. P99 – Preprocessor macros and functions for C99. https://gustedt.gitlabpages.inria.fr/p99/.

Tom Henretty, *Justin Holewinski*, *Richard Veras*, *Franz Franchetti*, *Louis-Noël Pouchet*, *J Ramanujam*, *Atanas Rountev*, and *P Sadayappan*. 2013. A Domain-Specific Language and Compiler for Stencil Computations on Short-Vector SIMD and GPU Architectures. In *Compilers for Parallel Computing Workshop*.

Larry Jones and *Jens Gustedt* (Eds.). 2020. *ISO/IEC 9899 working draft February 2020*. Number ISO/IEC 9899. ISO. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2478.pdf

JTC1/SC22/WG14 (Ed.). 2018. *Programming languages - C* (fourth ed.). Number ISO/IEC 9899. ISO. https://www.iso.org/standard/74528.html

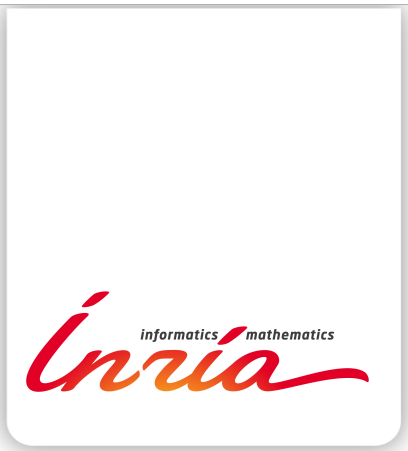
Charles Leiserson and *Aske Plaat*. 1997. Programming Parallel Applications in Cilk. *Siam news* (07 1997).

The Open Group (Ed.). 2018. *The Open Group Base Specifications*. Vol. Issue 7. IEEE and The Open Group, Chapter getconf – get configuration values.

OpenMP 2018. OpenMP Application Programming Interface. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. Version 5.0.

Boris Schäling. 2020. *The Boost C++ Libraries*. XML Press.

Yuan Tang, *Rezaul Chowdhury*, *Chi-Keung Luk*, and *Charles E Leiserson*. 2011. Coding stencil computations using the pochoir stencil-specification language. In *USENIX Workshop on Hot Topics in Parallelism*.



RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique

CS20101

54603 Villers-lès-Nancy Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399