Gen\_ast  
IR → C++/Ocaml interface

In this part, we more precisely describe the tool “gen\_ast”.

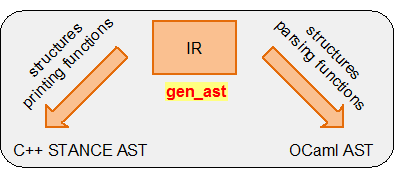


Figure 1: gen\_ast description

The objective of “gen\_ast” is to generate the following files from the C++ intermediate representation written in the file “intermediate\_format.ast”.

* intermediate\_format.h, intermediate\_format.c to be used by “FramaCIRGen”
* intermediate\_format.mli, intermediate\_format\_parser.mli, intermediate\_format\_parser.ml to be used by the “fclang” Frama-C plugin

“gen\_ast” is written in a single OCaml file gen\_ast.ml. It reads the file “intermediate\_format.ast”, produces an internal ast and generates the files. The following sections are a raw documentation of the code of get\_ast.ml.

Format of the internal AST

The format of the internal ast is composed of serveral nodes. Each nodes can be defined as a Union or as a Nuple. A Nuple is just a n-tuple of fields (name that is typed by typ). A Union is a list of named n-tuple of fields.

As a simple example, the description

**type** qualification =

| QNamespace { name: string; }

| QStructOrClass { name: string; }

| QTemplateInstance { name: string; parameters: template\_parameter list; }

;;

**type** qualified\_name =

{ prequalification: qualification list; decl\_name: string; }

;;

has the following translation in the internal AST:



Generation of the AST

This section concerns the  part of Figure 1 that is automatically generated from the file “intermediate\_format.ast”.

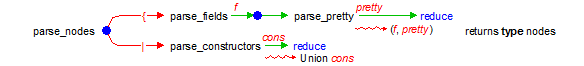
The generation algorithm is composed of a simple lexer lexer built with Genlex[[1]](#footnote-1).make\_lexer and of a simple parser built upon the standard camlp4 (top down, recursive descent parsing) parser. The main parsing function is parse\_ast. parse\_ast takes a stream as input and produces the previous internal ast.

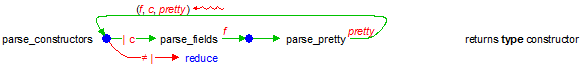
parse\_ast calls the following functions, depending on the token that appears in the input stream: parse\_nodes, parse\_constructors, parse\_fields, parse\_type, parse\_arg\_name, parse\_field. The rules follow the schemas below with the following conventions:

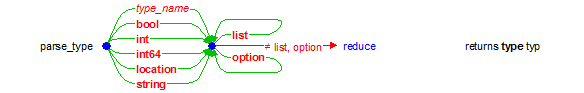
*  means that the token “token” has been recognized and that the parsing goes on the next point after having consumed the token.
*  means that the parsing emits the result “result”.
* defines a shift parsing rule: if the token “token” is recognized, “token” is consumed and the parser calls the rule “parse\_...” and when the rule is reduced, the parsing goes on the next point (in blue).
* defines another kind of shift parsing rule: if the token “token” is recognized, “token” is not consumed and the parser calls the rule “parse\_...” with “token” as first token and when the rule is reduced, the parsing goes on the next point.
* ,  are reduction rules. The latter one means that the parsing isfinished.

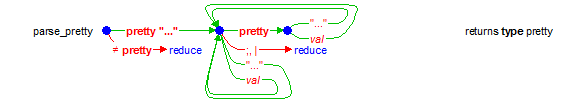
Hence the rules issued from the function parse\_ast are defined below:



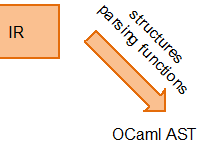








Generation of the OCaml source files for the “fclang” Frama-C plugin

This section concerns the  part of Figure 1. The function generate\_ocaml\_type generates the interface file “intermediate\_format.mli” from the **type** ast that is issued from the parsing of the file “intermediate\_format.ast” by parse\_ast. This function generates a header in the file and then it calls print\_ocaml\_typedef multiple times to generate the structure of the Intermediate Representation. For the following **type** ast,



it generates in “intermediate\_format.mli”

**type** …

…

**and** qualification =

| QNamespace **of** **string**

| QStructOrClass **of** **string**

| QTemplateInstance **of** **string** \* (template\_parameter) **list**

**and** qualified\_name =

{ prequalification: (qualification) **list**;

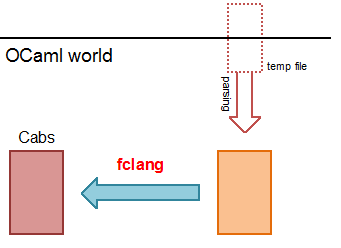
decl\_name: **string**

}

…

The next function generate\_ocaml\_parser\_sig generates the declarations of the OCaml parsing functions of the Intermediate Representation. These declarations are written in the file “intermediate\_format\_parser.mli” that generate\_ocaml\_parser\_sig generates from the **type** ast that is issued from the parsing of the file “intermediate\_format.ast” by parse\_ast. This function generates a header in the file “intermediate\_format\_parser.mli” and then it calls generate\_parse\_fun\_sig multiple times to generate the functions parse\_....

These parsing functions will parse the generated “temp file” (see the extracted part in figure below) to produce the structure of the Intermediate Representation. This structure is available for the “fclang” Frama-C plugin for the need of the conversion into CAbs.



For the following **type** ast,



generate\_parse\_fun\_sig generates in “intermediate\_format\_parser.mli”

…

val parse\_qualification: tree -> qualification

val parse\_qualified\_name: tree -> qualified\_name

…

The implementation of these parsing functions (temp file → OCaml IR) is performed by the function generate\_ocaml\_parser\_impl. It is the third and last main printing function. generate\_ocaml\_parser\_impl visits the **type** ast that is issued from the parsing of the file “intermediate\_format.ast” by parse\_ast and it write the implementation of the functions parse\_... are into the file “intermediate\_format\_parser.ml”.

This function first generates a consequent header – **type** tree representing the C++ AST and the function print\_tree – in the file “intermediate\_format\_parser.ml”. Then it calls generate\_type\_parser multiple times to generate the implementations of the functions parse\_....

The function generate\_type\_parser follows the structure of **type** ast. On **type** nodes that are multiple unions of **type** constructor, it calls the function generate\_union\_parser. On **type** nodes that are a **type** args tuple, generate\_type\_parser calls the function generate\_field\_child to first generate the parsing of the sub-structures of the IR and then it calls the function generate\_field\_node to generate the resulting node of the OCaml IR. On each union of type generate\_union\_parser does the same job but it calls generate\_constructor\_child instead of generate\_field\_child and generate\_constructor\_node instead of generate\_field\_node.

Both functions generate\_constructor\_child and generate\_field\_child call the function generate\_child for the parsing of the sub-structures. The key function of generate\_child is the function parse\_typ that generates the parsing of the different types of **type** typ within sub-fields of the **type** args.

For the following **type** ast,



generate\_ocaml\_parser\_impl generates in “intermediate\_format\_parser.mli”

…

**and** parse\_qualification \_\_node= *(\* <= generate\_type\_parser \*)*

**let** children = \_\_node.children **in**

**match** \_\_node.value **with**

| VName "QNamespace" -> *(\* <= generate\_union\_parser \*)*

**let** \_\_node = *(\* <= generate\_constructor\_child, generate\_child \*)*

**try** List.hd children

**with** Failure \_ -> parse\_error "No node for child name of qualification" \_\_node

**in** **let** name =

**match** \_\_node.value **with** *(\* <= parse\_typ \*)*

| VString s -> s

| \_ -> parse\_error "Expecting child name of qualification" \_\_node

**in** QNamespace (name) *(\* <= generate\_constructor\_node \*)*

| VName "QStructOrClass" -> *(\* <= generate\_union\_parser \*)*

**let** \_\_node = *(\* <= generate\_constructor\_child, generate\_child \*)*

**try** List.hd children

**with** Failure \_ -> parse\_error "No node for child name of qualification" \_\_node

**in** **let** name =

**match** \_\_node.value **with** *(\* <= parse\_typ \*)*

| VString s -> s

| \_ -> parse\_error "Expecting child name of qualification" \_\_node

**in** QStructOrClass (name) *(\* <= generate\_constructor\_node \*)*

| VName "QTemplateInstance" -> *(\* <= generate\_union\_parser \*)*

**let** \_\_node = *(\* <= generate\_constructor\_child, generate\_child \*)*

**try** List.hd children

**with** Failure \_ -> parse\_error "No node for child name of qualification" \_\_node

**in** **let** name =

**match** \_\_node.value **with** *(\* <= parse\_typ \*)*

| VString s -> s

| \_ -> parse\_error "Expecting child name of qualification" \_\_node

**in** **let** children = List.tl children in *(\* separator defined in generate\_union\_parser \*)*

**let** \_\_node = *(\* <= generate\_constructor\_child, generate\_child \*)*

**try** List.hd children

**with** Failure \_ -> parse\_error "No node for child parameters of qualification" \_\_node

**in** **let** parameters =

**match** \_\_node.value **with**

| VNil -> []

| VName "Cons" ->

List.map (fun \_\_node -> parse\_template\_parameter *(\* <= parse\_typ \*)* \_\_node)

\_\_node.children

| \_ -> parse\_error "Expecting a list" \_\_node

**in** QTemplateInstance (name, parameters) *(\* <= generate\_constructor\_node \*)*

| VName \_ -> parse\_error "Unknown constructor for qualification" \_\_node

| \_ -> parse\_error "Expecting constructor of qualification" \_\_node

**and** parse\_qualified\_name \_\_node= *(\* <= generate\_type\_parser \*)*

**let** children = \_\_node.children **in** *(\* <= generate\_field\_child \*)*

**let** \_\_node = *(\* <= generate\_child \*)*

**try** List.hd children

**with** Failure \_

-> parse\_error "No node for child prequalification of qualified\_name" \_\_node

**in** let prequalification =

**match** \_\_node.value **with** *(\* <= parse\_typ \*)*

| VNil -> []

| VName "Cons" ->

List.map (fun \_\_node -> parse\_qualification *(\* <= parse\_typ \*)*\_\_node)\_\_node.children

| \_ -> parse\_error "Expecting a list" \_\_node

**in** **let** children = List.tl children *(\* <= generate\_field\_child \*)*

**in** **let** \_\_node = *(\* <= generate\_child \*)*

**try** List.hd children

**with** Failure \_ -> parse\_error "No node for child decl\_name of qualified\_name" \_\_node

**in** **let** decl\_name =

**match** \_\_node.value **with** *(\* <= parse\_typ \*)*

| VString s -> s

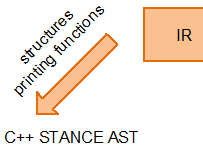
| \_ -> parse\_error "Expecting child decl\_name of qualified\_name" \_\_node

**in** {prequalification *(\* <= generate\_field\_node \*)*; decl\_name *(\* <= generate\_field\_node \*)*}

…

As a conclusion, gen\_ast generates from the IR intermediate\_format.ast the files intermediate\_format.mli, intermediate\_format\_parser.mli, intermediate\_format\_parser.ml to be used by the compilation of the “fclang” Frama-C plugin.

Generation of the C source files for “FramaCIRGen”

This section concerns the  part of Figure 1. The function generate\_c\_file generates both the header file “intermediate\_format.h” and the implementation file “intermediate\_format.c” from the **type** ast that is issued from the parsing of the file “intermediate\_format.ast” by parse\_ast.

For the header file “intermediate\_format.h”, generate\_c\_file first produces the structures to handle the locations, the options and the lists used in the IR. Then it calls the function generate\_c\_ast and generate\_c\_output\_proto to generate the structure printing functions available for “FramaCIRGen”. In fact the function generate\_c\_ast does all the preliminary work before the “FramaCIRGen” can call the persistent functions output\_... generated by generate\_c\_output\_proto. For the now well known **type** ast,



generate\_c\_ast generates:

* all the **enum** declarations (see the function generate\_enum\_decl) that will be used as flags to discriminate the sub-fields of the C++ STANCE AST that will be persisted (the Union of constructor in **type** nodes is translated into a discriminated **union** in C). On our example, this generates

**enum** tag\_qualification { QNAMESPACE, QSTRUCTORCLASS, QTEMPLATEINSTANCE };

* all the pointer declarations (see the function generate\_typedef) that point onto the sub-structures of the C++ STANCE AST. On our example, this generates

**typedef** **struct** \_qualification \*qualification ;

**typedef** **struct** \_qualified\_name \*qualified\_name ;

* all the structure declarations (see the function generate\_discriminated\_union) that define the sub-structures of the C++ STANCE AST. On our example, this generates

**struct** \_qualification {

**enum** tag\_qualification tag\_qualification;

**union** {

**struct** {

**const** **char** \* name;

} QNamespace;

**struct** {

**const** **char** \* name;

} QStructOrClass;

**struct** {

**const** **char** \* name;

list parameters;

} QTemplateInstance;

} cons\_qualification;

};

**struct** \_qualified\_name {

**list** prequalification;

**const** **char** \* decl\_name;

};

* all the construction functions (see the function generate\_constructor\_proto) that enable to build the complete structure of the C++ STANCE AST. In fact, these construction functions are called everywhere in “FramaCIRGen” and only these functions are really called. When the C++ STANCE AST is complete, the top writing function output\_file automatically generates the temp file to transfer the source code information in the OCaml world. On our example, generate\_constructor\_proto generates

qualification qualification\_QNamespace( **const** **char** \* name );

qualification qualification\_QStructOrClass( const **char** \* name );

qualification qualification\_QTemplateInstance( **const** **char** \* name,

list parameters );

qualified\_name qualified\_name\_cons( list prequalification,

**const** **char** \* decl\_name );

* all the destruction functions (see the function generate\_destructor\_proto) that enable to locally destroy sub-structures no more registered the C++ STANCE AST. It is sometimes convenient to manipulate the C++ STANCE AST before the top writing function output\_file automatically generates the temp file to transfer the source code information in the OCaml world. Among such manipulations, we can list
  + the transformation of a function definition into a function declaration followed later by a function definition required by the template instantiation mechanism – a template instance may be visited in FramaCIRGen before the actual template arguments.
  + more anecdotally, the **sizeof** construct in our IR does not take the type qualification into account: **sizeof**(type **const**) is always equal to **sizeof**(type). So the destruction function helps to remove the qualification.
  + the parsing of the annotation language performs a type inference to automatically introduce cast operations where they are needed. If no cast is needed, the inferred type has no utility and so it should be destroyed.

On our example, generate\_constructor\_proto generates

**void** free\_qualification(qualification);

**void** free\_qualified\_name(qualified\_name);

* all the duplication functions (see the function generate\_dup\_proto) that define inner copies of the sub-structures of the C++ STANCE AST. It is useful to duplicate a type when it is used for the declaration of multiple field or variables like in the declaration of “**int** x, y, z;” On our example, this generates

qualification qualification\_dup(**const** qualification);

qualified\_name qualified\_name\_dup(**const** qualified\_name);

* all the equality functions (see the function generate\_equal\_proto) that define inner comparison of the sub-structures of the C++ STANCE AST. It is only useful to parse the annotation language. For example, there is no need to generate a cast in case of binary operation if both arguments have the same type. On our example, this generates

\_Bool qualification\_equal(**const** qualification, **const** qualification);

\_Bool qualified\_name\_equal(**const** qualified\_name, **const** qualified\_name);

and generate\_c\_output\_proto generates:

* all the declarations of the printing function (see the function generate\_output\_proto). In fact, only the top one output\_file is called by “FramaCIRGen”. This top function calls all the other printing functions on the sub-structures of the C++ STANCE AST. On our example, generate\_output\_proto generates

**void** output\_qualification(FILE\*,qualification);

**void** output\_qualified\_name(FILE\*,qualified\_name);

For the implementation file “intermediate\_format.c”, generate\_c\_file produces the implementation of the function that it has declared in the header file. For the now well known **type** ast,



generate\_c\_file generates:

* the implementation of all the construction functions (see the function generate\_c\_constructor). On our example, generate\_constructor\_proto generates

qualification qualification\_QNamespace( **const** **char** \* name ) {

qualification obj = **NULL**;

obj = malloc(**sizeof**(\*obj));

**if**(obj){

obj->tag\_qualification=QNAMESPACE;obj->cons\_qualification.QNamespace.name = name;

}

**return** obj;

}

qualification qualification\_QStructOrClass( **const** **char** \* name ) {

qualification obj = **NULL**;

obj = malloc(**sizeof**(\*obj));

**if**(obj){

obj->tag\_qualification=QSTRUCTORCLASS;

obj->cons\_qualification.QStructOrClass.name = name;

}

**return** obj;

}

qualification qualification\_QTemplateInstance( const char \* name, list parameters ) {

qualification obj = **NULL**;

obj = malloc(**sizeof**(\*obj));

**if**(obj){

obj->tag\_qualification=QTEMPLATEINSTANCE;

obj->cons\_qualification.QTemplateInstance.name = name;

obj->cons\_qualification.QTemplateInstance.parameters = parameters;

}

**return** obj;

}

qualified\_name qualified\_name\_cons( list prequalification, **const** **char** \* decl\_name ) {

qualified\_name obj = malloc(**sizeof**(\*obj));

**if** (obj) {

obj->prequalification = prequalification;

obj->decl\_name = decl\_name;

}

**return** obj;

}

* the implementation of all the destruction functions (see the function generate\_c\_destructor). On our example, it generates

**void** free\_qualification(qualification obj) {

**switch** (obj -> tag\_qualification) {

**case** QNAMESPACE: **break**;

**case** QSTRUCTORCLASS: **break**;

**case** QTEMPLATEINSTANCE:

{ list elt = obj->cons\_qualification.QTemplateInstance.parameters;

**while**(elt) {

list tmp=elt->next;

template\_parameter content = (template\_parameter)elt->element.container;

free\_template\_parameter(content);

free(elt);

elt=tmp;

}

} **break**;

}

free(obj);

}

**void** free\_qualified\_name(qualified\_name obj){

{ list elt = (\*obj).prequalification;

**while**(elt) {

list tmp=elt->next;

qualification content = (qualification)elt->element.container;

free\_qualification(content);

free(elt);

elt=tmp;

}

}

free(obj);

}

* the implementation of all the duplication functions (see the function generate\_c\_dup). On our example, this generates

qualification qualification\_dup(**const** qualification src) {

**if** (src == **NULL**) **return** **NULL**;

qualification dst = malloc(sizeof(\*src));

**if**(dst == **NULL**) memory\_exhausted();

dst->tag\_qualification = src->tag\_qualification;

**switch** (src->tag\_qualification) {

case QNAMESPACE: {

dst->cons\_qualification.QNamespace.name

=strdup(src->cons\_qualification.QNamespace.name);

**if**(dst->cons\_qualification.QNamespace.name==**NULL**) memory\_exhausted();

**break**;

}

**case** QSTRUCTORCLASS: {

dst->cons\_qualification.QStructOrClass.name

=strdup(src->cons\_qualification.QStructOrClass.name);

**if**(dst->cons\_qualification.QStructOrClass.name==**NULL**) memory\_exhausted();

**break**;

}

case QTEMPLATEINSTANCE: {

dst->cons\_qualification.QTemplateInstance.name

=strdup(src->cons\_qualification.QTemplateInstance.name);

**if**(dst->cons\_qualification.QTemplateInstance.name==**NULL**) memory\_exhausted();

{ list elt\_src = src->cons\_qualification.QTemplateInstance.parameters;

dst->cons\_qualification.QTemplateInstance.parameters = **NULL**;

list elt\_dst = **NULL**;

**while**(elt\_src) {

list tmp = malloc(**sizeof**(struct \_list));

**if** (!tmp) { memory\_exhausted(); };

tmp->element.container

= template\_parameter\_dup((template\_parameter)elt\_src->element.container);

tmp->next=**NULL**;

**if**(elt\_dst) {

elt\_dst->next = tmp;

} **else** {

dst->cons\_qualification.QTemplateInstance.parameters=tmp;

}

elt\_dst=tmp;

elt\_src=elt\_src->next;

}

}

**break**;

}

}

**return** dst;

}

qualified\_name qualified\_name\_dup(**const** qualified\_name src) {

**if** (src == **NULL**) return **NULL**;

qualified\_name dst = malloc(**sizeof**(\*src));

**if** (dst == **NULL**) memory\_exhausted();

{ list elt\_src = (\*src).prequalification;

(\*dst).prequalification = **NULL**;

**list** elt\_dst = **NULL**;

**while**(elt\_src) {

list tmp = malloc(**sizeof**(**struct** \_list));

**if** (!tmp) { memory\_exhausted(); };

tmp->element.container

= qualification\_dup((qualification)elt\_src->element.container);

tmp->next=**NULL**;

**if**(elt\_dst) {

elt\_dst->next = tmp;

} **else** {

(\*dst).prequalification=tmp;

}

elt\_dst=tmp;

elt\_src=elt\_src->next;

}

}

(\*dst).decl\_name=strdup((\*src).decl\_name);

**if**((\*dst).decl\_name==**NULL**) memory\_exhausted();

**return** dst;

}

* the implementation of all the equality functions (see the function generate\_c\_equal). On our example, this generates

\_Bool qualification\_equal(**const** qualification v1, **const** qualification v2) {

**if** (v1->tag\_qualification != v2->tag\_qualification) **return** **false**;

**switch** (v1->tag\_qualification) {

**case** QNAMESPACE: {

**if** (strcmp(v1->cons\_qualification.QNamespace.name,

v2->cons\_qualification.QNamespace.name) != 0) **return** **false**;

return true;

}

**case** QSTRUCTORCLASS: {

if (strcmp(v1->cons\_qualification.QStructOrClass.name,

v2->cons\_qualification.QStructOrClass.name) != 0) **return** **false**;

**return** **true**;

}

**case** QTEMPLATEINSTANCE: {

**if** (strcmp(v1->cons\_qualification.QTemplateInstance.name,

v2->cons\_qualification.QTemplateInstance.name) != 0) **return** **false**;

{ list l1 = v1->cons\_qualification.QTemplateInstance.parameters,

l2 = v2->cons\_qualification.QTemplateInstance.parameters;

**while** (**true**) {

**if** (l1 == **NULL** && l2 == **NULL**) **break**;

**if** (l1 == **NULL** || l2 == **NULL**) **return** **false**;

**if** (!template\_parameter\_equal((template\_parameter)l1->element.container,

(template\_parameter)l2->element.container)) **return** **false**;

l1 = l1 -> next;

l2 = l2 -> next;

}

}

**return** **true**;

}

}

**return** **false**;

}

\_Bool qualified\_name\_equal(const qualified\_name v1, const qualified\_name v2) {

{ list l1 = (\*v1).prequalification, l2 = (\*v2).prequalification;

**while** (**true**) {

**if** (l1 == **NULL** && l2 == **NULL**) **break**;

**if** (l1 == **NULL** || l2 == **NULL**) **return** **false**;

**if** (!qualification\_equal((qualification)l1->element.container,

(qualification)l2->element.container)) **return** **false**;

l1 = l1 -> next;

l2 = l2 -> next;

}

}

**if** (strcmp((\*v1).decl\_name,(\*v2).decl\_name) != 0) **return** **false**;

**return** **true**;

}

* the implementation of all the declarations of the printing function (see the function generate\_c\_output\_func). On our example, generate\_output\_proto generates

**void** output\_qualification(FILE\* out,qualification obj) {

**switch** (obj -> tag\_qualification) {

**case** QNAMESPACE:

fprintf(out,"%\*s%s\n",indent,"","QNamespace");

indent+=2;

fprintf(out,"%\*s\"%s\"\n",indent,"",obj -> cons\_qualification.QNamespace.name);

indent-=2;

fflush(out);

break;

**case** QSTRUCTORCLASS:

fprintf(out,"%\*s%s\n",indent,"","QStructOrClass");

indent+=2;

fprintf(out,"%\*s\"%s\"\n",indent,"",obj -> cons\_qualification.QStructOrClass.name);

indent-=2;

fflush(out);

**break**;

**case** QTEMPLATEINSTANCE:

fprintf(out,"%\*s%s\n",indent,"","QTemplateInstance");

indent+=2;

fprintf(out,"%\*s\"%s\"\n",indent,"",obj->cons\_qualification.QTemplateInstance.name);

{ list elt = obj -> cons\_qualification.QTemplateInstance.parameters;

fprintf(out, elt?"%\*sCons\n":"%\*snil\n",indent,"");

indent+=2;

**while**(elt) {

template\_parameter content = (template\_parameter)elt->element.container;

output\_template\_parameter(out,content);

elt=elt->next;

}

indent-=2;

}

indent-=2;

fflush(out);

**break**;

}

}

**void** output\_qualified\_name(FILE\* out, qualified\_name obj) {

fprintf(out,"%\*stuple\n",indent,"");

indent+=2;

{ list elt = obj->prequalification;

fprintf(out, elt?"%\*sCons\n":"%\*snil\n",indent,"");

indent+=2;

**while**(elt) {

qualification content = (qualification)elt->element.container;

output\_qualification(out,content);

elt=elt->next;

}

indent-=2;

}

fprintf(out,"%\*s\"%s\"\n",indent,"",obj->decl\_name);

indent-=2;

fflush(out);

}

As a conclusion, gen\_ast generates from the IR intermediate\_format.ast the files intermediate\_format.h, intermediate\_format.c to be used by the compilation of “FramaCIRGen”.

1. [http://caml.inria.fr/pub/docs/manual-OCaml/libref/Genlex.html](http://caml.inria.fr/pub/docs/manual-ocaml/libref/Genlex.html) [↑](#footnote-ref-1)