Draft_0.03

Browser & Refactor

Abidos C++

Programmer Manual

O(n)

Fructu

Abidos C++ Programmer Manual

COLLABORATORS			
	TITLE : Abidos C++ Programmer Manual		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	fructu	November 2012	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.03	November 2012	starting to write this documentation:	Fructu

Contents

1 Introduction **1**

1.1 Milestones . . . 1

1.2 How read this book . . . 1

2 Making **2**

2.1 Configuration . . . 2

2.1.1 processor/CMakeLists.txt . . . 2

2.1.2 processor/src/CMakeLists.txt . . . 3

2.1.3 processor/includes/CMakeLists.txt . . . 3

2.2 Files generated automatically . . . 3

2.3 Compilation & linking stages . . . 5

2.4 Other make rules . . . 5

3 Architecture of abidos **6**

3.1 Abidos Components . . . 6

4 Loader **8**

4.1 One C++ file . . . 8

4.2 .abidos_cpp/files_input . . . 9

4.3 Loader classes . . . 10

5 Lexer **11**

6 Mangling **12**

6.1 Composite . . . 12

6.2 Mangling classes . . . 14

6.3 Mangling functions . . . 15

7 Recursive descent parser **17**

7.1 Classes . . . 17

7.2 Debugging Abidos . . . 20

7.3 Meaningful parts of parser . . . 24

7.4 How do token_get() token_next() do their work ? . . . 28

8 Trace system **31**

9 Context **35**

9.1 How do context do it? . . . 35

9.2 Parts of context . . . 38

10 Symbols table **41**

10.1 Most important classes of symbols table . . . 41

10.2 Saving context . . . 44

11 Testing Abidos C++ **45**

11.1 Executing the tests . . . 45

11.2 Hey wait a moment, testing files are C++ files! . . . 46

11.3 Test files and what they testing . . . 49

12 Advanced trace **51**

12.1 Inner classes . . . 51

12.2 Templates . . . 65

12.3 Templates instantiation . . . 70

13 Index **76**

List of Figures

2.1 make process . . . 4

3.1 Abidos components . . . 6

4.1 Loader one source file . . . 8
4.2 Loader filers from .abidos_cpp/files_input . . . 8
4.3 c_loader UML diagram . . . 10

5.1 stack changes with #includes . . . 11

6.1 Object diagram when book_03.cpp has been parsed (this is an approximation) . . . 13

7.1 c_parser UML diagram 1 . . . 18
7.2 c_parser UML diagram 2 . . . 19
7.3 variable trace tree . . . 27
7.4 function trace tree . . . 27
7.5 token_next() calls and buffer changes . . . 28

8.1 trace tree example . . . 32
8.2 trace tree pruned example . . . 34

9.1 trace_book_02.cpp_urls_pruned . . . 36
9.2 state machine . . . 38

10.1 c_symbols_table UML diagram 1 . . . 42
10.2 c_symbols_table c_declarator UML diagram 2 . . . 43

11.1 make test_run . . . 45
11.2 out_t001.cpp.dot . . . 46
11.3 trace_t001.cpp_pruned.gv . . . 48

12.1 classes rules until A . . . 53
12.2 inner classes rules until B . . . 55
12.3 inner classes rules declaration of f_b . . . 56

12.4 inner classes rules definition of f_b 62
12.5 state machine 65
12.6 book_template template_parameter_list subtree 67
12.7 book_template function_definition subtree 69
12.8 classes rules to parse B: 71
12.9 classes rules to parse B::a1: 72

List of Tables

11.1 test_out files 46
11.2 test files rules used part 1 49
11.3 test files rules used part 2 50

Chapter 1

Introduction

In this manual i am going to explain how Abidos works, is important that the reader of this manual have read the user manual and knows how to use Abidos to parse C++ projects.

Parsing C++ is a hard task it has a complex grammar and Abidos is a experimental solution using a descent parser handmade instead of an ascent parser (yacc/bison).

1.1 Milestones

When i began this project i think in a set of phases, Milestones or goals that i want achieve with the time:

1. Parsing C++ and generate UML browsable diagrams; so far this phase is working but not full implemented yet, Abidos can parse a big portion of C++ grammar above all the part about declaration of classes, methods, functions, variables, parameters… but all the inside body of method and functions need to be development.
2. Refactor parts of code.
3. Generating a set of backends customized to future projects, one of this backends is the UML generator.

For now there is a lot of work to do in the parser core of Abidos.

1.2 How read this book

In order to have a good knowledge about Abidos C++ parts and learn to hack it there are topics to study and these topics are very interconnected therefore is a tough task write a clear manual due to explain topics needs to explain another topics and these topics needs the first topics maybe the best way is a whole first read to keep in mind what parts Abidos have and which are his relations and later you can read some parts again when you need an explanation about some topic or Abidos part.

The goal for the reader is understand the examples in advance trace chapter where the most important cases are explained, but to understand these cases you need pass through the valley of the previous topics.

Chapter 2

Making

Abidos making process is not than simple that i would want, there are a set of scripts doing parts of the making process, trying to follow DRY.

> The DRY principle is stated as "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." When the DRY principle is applied successfully, a modification of any single element of a system does not require a change in other logically unrelated elements.
>
> — Andy Hunt and Dave Thomas

Abidos is an experiment about a set of technologies that i am trying to put together in this project, working and learning a lot of interesting things. therefore lets go to see how the making process of Abidos is.

2.1 Configuration

Abidos uses cmake http://www.cmake.org/ for this purpose.

> CMake is a family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files. CMake generates native makefiles and workspaces that can be used in the compiler environment of your choice.
>
> — cmake

To configure Abidos C++ write in a terminal:

```
cd processor
cmake .
```

Note
You can see more about configure in the user manual.

2.1.1 processor/CMakeLists.txt

Abidos sets here:

- his version
- the directories that make process should pass to the scripts.

Then the others CMakeLists.txt are processed in order to generate the Makefile.

2.1.2 processor/src/CMakeLists.txt

Here cmake process:

- check if all necessary includes are installed in the system.
- check if Flex and Bison are installed in the system.

Note
Abidos do not use the code generated from Bison to do an ascent parser only use some declarations needed by Flex.

- sets the build options.

generates rules for Makefile:

- calls flex_execute.pl.
- calls bison_execute.pl.
- to copy ts.cpp ts.h from preprocessor "a module used to scan files and generate graphs with the includes net and can be used to obtain a list with files and with includes"
- calls extract_symbols.pl.
- calls check_directories.pl and generate_begin_graph.pl.
- to compile and link Abidos.
- indent_run that calls check_indent.sh and indent_run.sh.
- test_run that calls tests_run.pl
- install Abidos.
- uninstall Abidos.

Note
These scripts are explained in the next section.

2.1.3 processor/includes/CMakeLists.txt

Generates part of the make process where the version are wrote inside of config.h using the script put_ifndef_date.pl

2.2 Files generated automatically

I am going to describe you how the builder scripts generate files needed by Abidos. These scripts are located in processor/scripts/ directory.

When you have configured Abidos C++ a Makefile was generated, you can invoke:

```
make all
```

You can see in this diagram the scripts needed by make with his input files and his output files.

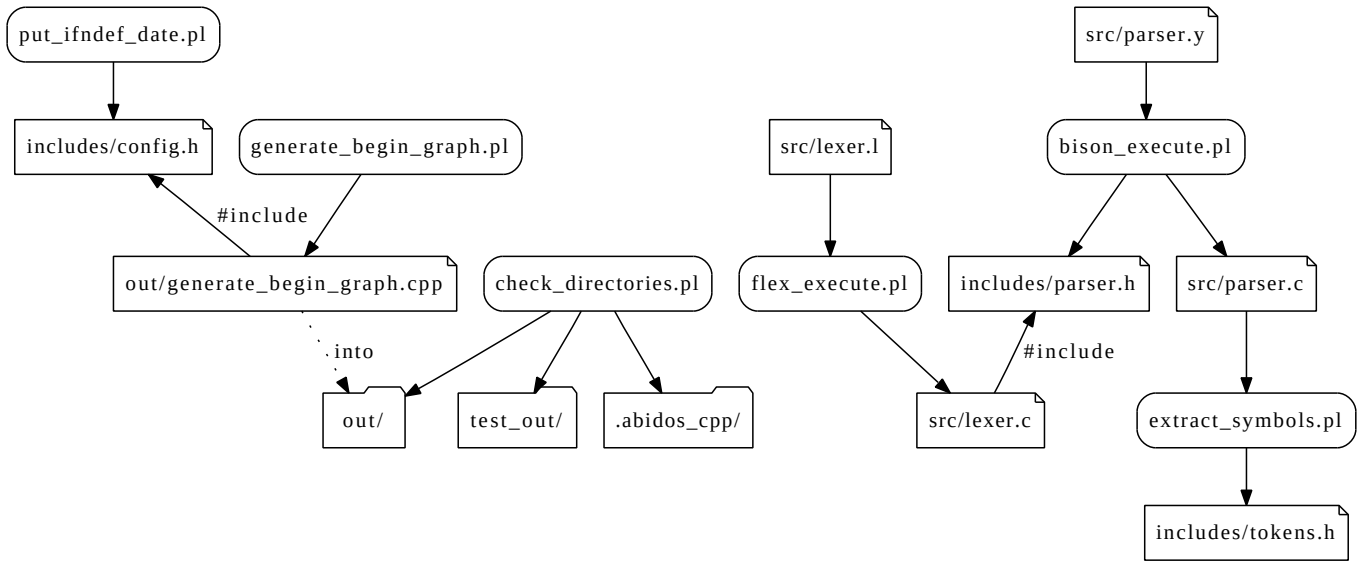

Figure 2.1: make process

These scripts are invoked in this order:

1. scripts/put_ifndef_date.pl Generates includes/config.h with compilation time and version extracted from includes/config.h.in
2. scripts/check_directories.pl if any of this directories (out, test_out, .abidos) does not exist this script creates them.
   - out here is where some files created automatically are put. So far only generate_begin_graph.cpp is put here but in the future i want to put the output files of flex, bison ... but for the moment i have other priorities.
   - test_out the output of the **make test_run** the files generates in this directory are explained forward in the testing chapter.
   - .abidos_cpp this directory has the files input and output of Abidos C++ :
     – files_input
     – files_output.dot
3. /scripts/generate_begin_graph.pl generates generate_begin_graph.cpp and this file generates the header of files_output.dot with version of Abidos and fonts sizes of the graphs.
4. scripts/bison_execute.pl from src/parser.y generates:
   - src/parser.c this file is no used to compile Abidos directly but from this file some things are extracted.
   - includes/parser.h
5. scripts/extract_symbols.pl Parses src/parser.c and generates includes/tokens.h with 2 important tables:
   - yytokens: static const char *const yytokens[] In this table we have the description of the tokens for debug purposes, for example if lexer passes token number 258 to parser with this table we know 258 is "IDENTIFIER". Is used by descent parser to print the tokens processed and to generates the nodes of trace tree, trace tree is an output of the trace system of Abidos this system can generate kind of trees to show how the syntactic rules are used and there are nodes with **token is** and **token is not** entries where you can see what tokens are processed that will be explained deeply in the trace chapter.
   - yytokens_short: static const char *const yytokens_short[] It is similar than the previous table but when the token is an ASCII symbol, we can read 36Th ASCII symbol like this "36→$".
6. scripts/flex_execute.pl Parses src/lexer.l and generates src/lexer.c (this file uses parser.h generates by Bison).

2.3 Compilation & linking stages

So far make process has generated all necessary files, with this files and the other files in src/ and include/ compilation and linking process generates src/abidos_cpp executable.

2.4 Other make rules

- test_run that calls tests_run.pl Explained in the testing chapter
- indent_run that calls check_indent.sh and indent_run.sh, this scripts put all the Abidos code in a style of codification for now that style is Kernighan and Ritchie's http://en.wikipedia.org/wiki/Indent_style and the beautifier used is **astyle**

Chapter 3

Architecture of abidos

The Goal of this chapter is to show you what components have Abidos and which is the purpose of this components. All this components have a brief explanation and will be explain deeply in a chapter for each of them. The idea is you reach a overview knowledge of the Abidos parts and know his most important relations between them.

3.1 Abidos Components

In this diagram you can see the main parts of Abidos:

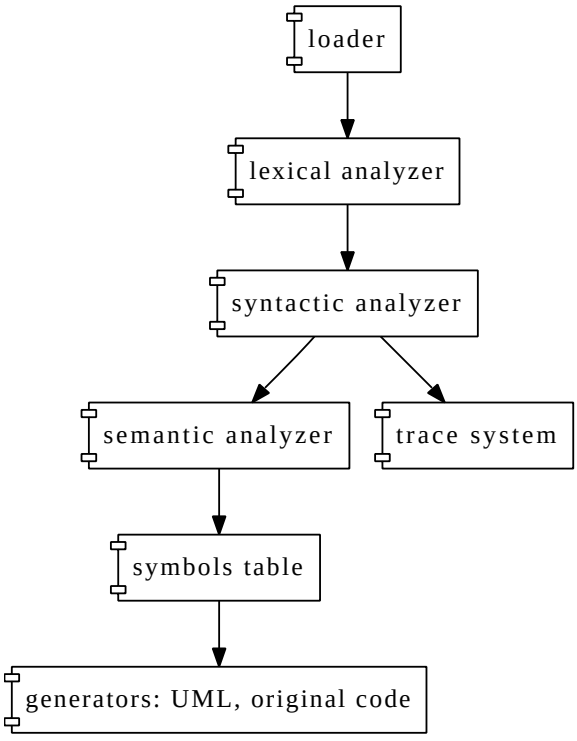

Figure 3.1: Abidos components

Abidos have these main modules:

- loader

- lexical analyzer: The lexer of Abidos generated by Flex.
- syntactic analyzer: The Parser of Abidos, hand made, is a descent parser.
- semantic analyzer: The deep knowledge about C++ is here this component put it in symbols table.
- symbols table: Stores information about Classes, Structures . . . this information is used by parser and generators.
- generators: Traverses the information of symbols table and generates an output:
  - UML diagrams: The output are dot files **.abidos_cpp/files_output.dot**
  - original code: This generator tries to generate the same input cpp file for example **test/t001.cpp** → **test_out/out_t001.cpp**
- trace system: When the parser is working send to trace system information about what rules has been used, when the parser finishes his work, trace system generates trace output with **annotated parse trees**.
- dot viewer: used to see the dot output of Abidos examples:

```
xdot_run.py .abidos_cpp/files_output.dot ❶
xdot_run.py test_out/trace_t031_01.cpp_urls_pruned.gv ❷
```

❶ This is the standard output of Abidos to see the UML diagram generated.

❷ Parser annotated tree of the test file tests/t031_01.cpp.

Chapter 4

Loader

Yo have seen in the user manual that abidos needs to be loaded with 2 options:

- A C++ file

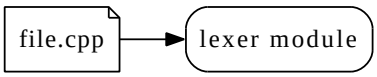

Figure 4.1: Loader one source file

- With .abidos_cpp/files_input has a list of C++ files and directories.

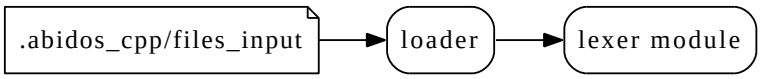

Figure 4.2: Loader filers from .abidos_cpp/files_input

In order to parse all cpp files from a C++ project Abidos uses this module/class you can read and introduction about it in the **user manual** in the section **The hacker way Abidos toolchain explanation**, the next step is understand how works internally and in this section is the explanation.

4.1 One C++ file

When Abidos is called with a C++ file, Abidos will start to process that file the loader does not have work to do.

one file to parse

```
.../abidos_cpp/processor/src/abidos \
    --includes .../abidos_cpp/processor/test_includes/ \
    --out_dir  .../abidos_cpp/processor/test_out/ \
    --test_all_tokens_consumed_flag \
    --test_original \
    --ts_show \
    --verbose \
    .../abidos_cpp/processor/test/book_01.cpp > \ ❶
      .../abidos_cpp/processor/test_out/out_book_01.cpp.txt
```

❶ here is the file to will being parsed with Abidos.

Abidos in this mode calls **parser.yyparse(file_name)** is used for example in **make test_run**.

Note
yyparser call is in src/main.cpp file

If that file has **#include** directives them will be processed by the preprocessor rules of **src/parser_descent_preprocessor.cpp**, there are not work to loader here; are all business from lexical module and syntactic module.

4.2 .abidos_cpp/files_input

The most easy way to use this mode is call the script **abidos_make_process.pl**, you can see it in the user manual.

This script invokes Abidos like this:

loader file

```
/opt/abidos_cpp/abidos_cpp \
    --test_all_tokens_consumed_flag \
    --no_std \
    --out_dir .abidos_cpp \
    --loader .abidos_cpp/files_input ❶
```

❶ **--loader** is the option to use the **loader** module.

Note
loader is called from src/main.cpp **parser.yyparse_loader**(file_name);

4.3 Loader classes

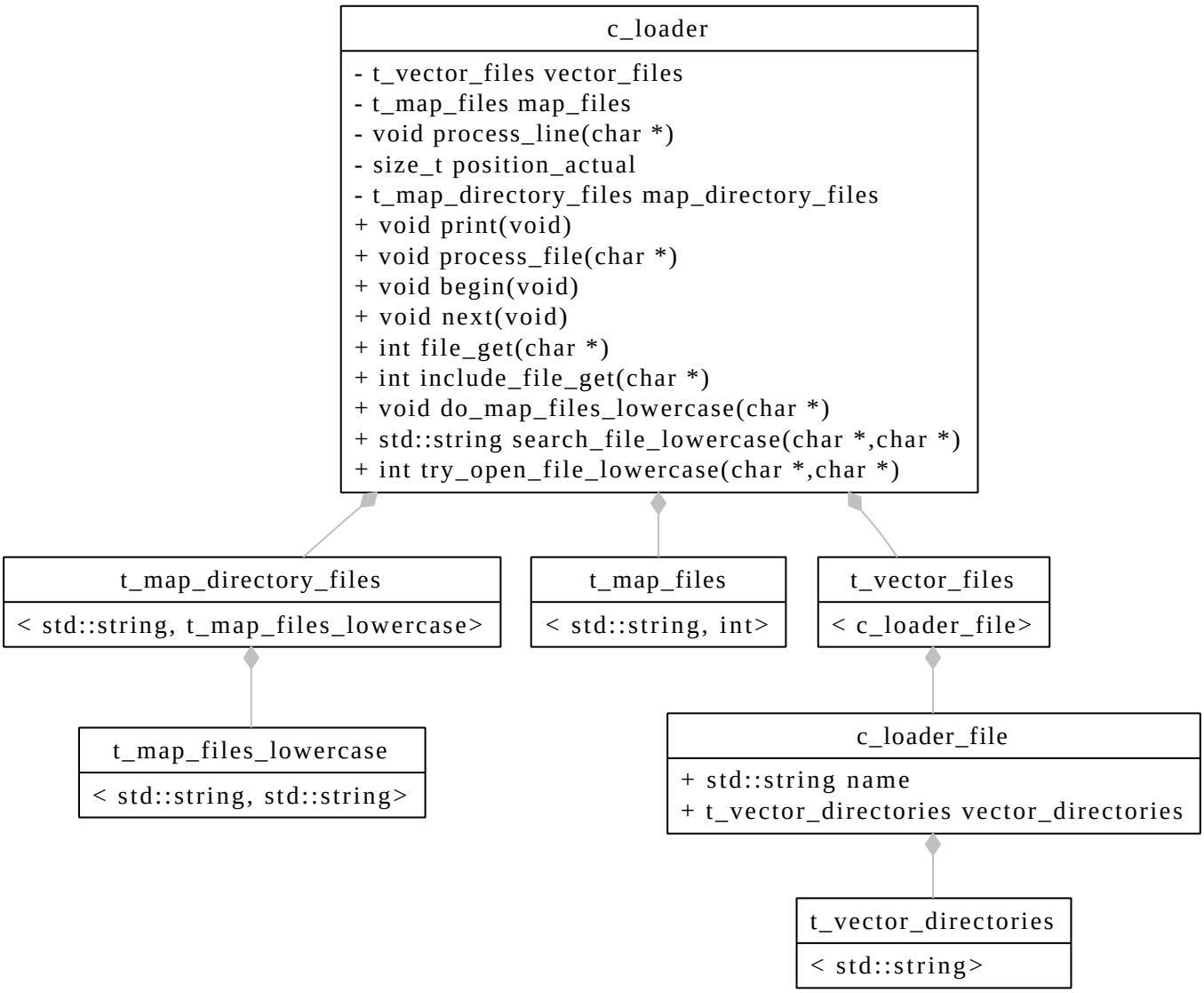

Figure 4.3: c_loader UML diagram

The loader module stores the files that the parser will need, when the parser asks for the next file; loader module tries to search this file in the directories extracted from the .abidos_cpp/files_input. If not found that file in the fist try loader will try do it again in a case insensitive mode (due are the _lowercase classes)

Chapter 5

Lexer

Abidos has a lexer generated by flex, lexer can work alone or with the loader component see chapter Loader.

Lexer is used by parser **src/parser_descent.cpp** the steeps are:

1. lex_file_init(file_name), lexer open the file_name and initializes his structures to scan it for example the stack of files.
2. yylex(), iterating with this function parser obtain the tokens of the source file.
3. lex_file_push, if the actual file has #include "<file>" then src/parser_descent_preprocessor.cpp push it and the scanner:
    a. stop to scan the actual file.
    b. saves the environment of his structures.
    c. begin to parse the included file.
    d. when the included file has bee finished, resume the parsing of the previous file.
4. lex_file_end(), parser call this to close the file.
5. yylex_destroy(), free memory used by the lexer to do his work.

Lets look the next example:

We have **f1.cpp** file that includes **f2.h** and this includes **f3.h**, lexer starts parsing **f1.cpp**, when parser found **#include "f2.h"** uses **lex_file_push** to store the context about **f1.cpp**, and lex starts to scan **f2.h** until parser founds **#include "f3.h"**, then another call to **lex_file_push** and lexer starts to scan **f3.h**, when the eof is reached, lexer pop out the file from stack and restores the context of **f2.h** when the eof of this file is reached another pop out restores the context of the first file **f1.cpp**.

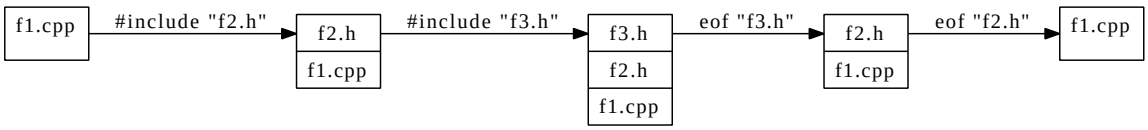

Figure 5.1: stack changes with #includes

Chapter 6

Mangling

In order to store the symbols Abidos C++ uses 2 different ways to set the relations between them.

6.1 Composite

Is used inside of classes lets see it with an example:

test/book_03.cpp

```
class A {
  int a1;
  int a2;
};
```

The symbols table at the end of his execution has:

test_out/out_book_03.cpp.txt

```
c_symbols_table::print
{
  stack level[0]
  {
    first[A]\
      /.../abidos_cpp/processor/test/book_03.cpp:4 \
      id[258]->[IDENTIFIER] text[A] type[265]->[CLASS_NAME] \
      class_key[300]->[CLASS]
    {
      map_base_class
      {
        empty
      }
      map_friend_class
      {
        empty
      }
      vector_class_member [2] ❶
      {
       [int] [a1]
       [int] [a2]
      }
      map_class_member [2] ❷
      {
       [PRIVATE]: [int] first[a1]->[a1]
       [PRIVATE]: [int] first[a2]->[a2]
```

```
      }
      free_declarator
      {
      }
    }
  }
}
```

1. c_class_members has vector_class_member to show the members of a class in the same order than has been founded in the parsing of their class.
2. c_class_members has map_class_member too, to do search of this symbols faster than in the previous vector.

We can see a diagram of how is that table of symbols when the example has been parsed.

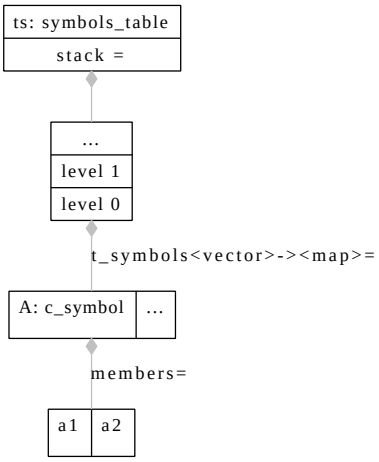

Figure 6.1: Object diagram when book_03.cpp has been parsed (this is an approximation)

Symbols table has a stack to deal the scope of the symbols for example:

```
int i; ❶

void f1(void)
{
  int a1; ❷
};

void f2(void)
{
  int b1; ❸

  ++i;
};
```

1. this is the lowest level of the stack.
2. in this level we can see **int i** and **int a1**.
3. this level is the same level than the previous level but when f1 has been parsed the level can be deleted and there is not a1 token anymore, but from this level we can reach **i**.

A stack is the easy way used by compilers to deal with scopes.

But when we are deal with declarations of classes

test/book_04.cpp

```
int i;

class A
{
  int a1;
};

class B
{
  int b1;
  void bf(void){
    A a; ❶
  }
};
```

❶ **i**,**A** and **B** are in the same level of scope, inside of A we are in a inner scope saved in the members vector of the **A** symbol we can not lose this information when the declaration has been finished because can happens like this and we need check what is A and if A contains a1.

Note
we can dispose the scope inside of a function implementation when its scope finished because we do not need this information more but we can not dispose the information of the scope inside of a class therefore deleting scopes only is when we are parsing functions.

6.2 Mangling classes

When abidos C++ has to deal with inner classes or classes inside of name spaces would can use a composite too. But do not use it.

```
class A {
  int a1;
  class B{
    int b1;
  }
};
```

The symbols table at the end of his execution has:

test_out/out_book_04.cpp.txt

```
c_symbols_table::print
{
  stack level[0]
  {
    first[A]/.../abidos_cpp/processor/test/book_04.cpp:4 \
      id[258]->[IDENTIFIER] text[A] type[265]->[CLASS_NAME] \
      class_key[300]->[CLASS]
    {
      ...
      vector_class_member [1]
      {
       [int] [a1]
```

```
      }
      map_class_member [1]
      {
       [PRIVATE]: [int] first[a1]->[a1]
      }
      ...
    }
    first[A::B]/.../abidos_cpp/processor/test/book_04.cpp:6 \
      id[258]->[IDENTIFIER] text[A::B] type[265]->[CLASS_NAME] \ ❶
      class_key[300]->[CLASS]
    {
      ...
      vector_class_member [1]
      {
       [int] [b1]
      }
      map_class_member [1]
      {
       [PRIVATE]: [int] first[b1]->[b1]
      }
      ...
    }
  }
```

❶ Here we can see the trick **B** is saved like **A::B**.

With this mangling technique Abidos C++ does not need a composite system to store inner classes and the processes of store and retrieve that information from the symbols table is more straight.

6.3 Mangling functions

In C++ you can have 2 function like this:

test/book_05.cpp

```
int f1(void);
int f1(int a);
```

This overloaded functions need something more than **f1** to identified each one, lets go to see how Abidos do it.

```
c_symbols_table::print
{
  stack level[0]
  {
    first[f1(int)]/.../abidos_cpp/processor/test/book_05.cpp:5 \ ❶
    id[258]->[IDENTIFIER] text[f1] type[0]->[0 UNDEFINED] \
    class_key[0]->[0 UNDEFINED]
    {
      ...
      free_declarator
      {
       [int] [f1]( [int] [a])
      }
    }
    first[f1(void)]/.../abidos_cpp/processor/test/book_05.cpp:4 \ ❷
    id[258]->[IDENTIFIER] text[f1] type[0]->[0 UNDEFINED] \
    class_key[0]->[0 UNDEFINED]
    {
      ...
```

```
      free_declarator
      {
       [int] [f1]( [void] [void])
      }
    }
  }
}
```

1. the name mangled of the first function is **f1(int)**
2. the name mangled of the second function is **f1(void)**

I think this is a easy form to understand what is each symbol, you see the () and you know that symbol is a function.

Chapter 7

Recursive descent parser

When i was studying compilers in university, i did not like yacc to implement parsers, i used it to development a Java compiler and i think a descent parser would be more suitable for that purpose. Yacc uses LALR http://en.wikipedia.org/wiki/LALR_parser, the parser must decide what rule use with only 1 token.

This code:

```
int a;
int f(void);
```

Have 2 lines starting with the same token, in yacc that would be a problem, therefore Abidos is an experiment about do a descent parser of a complex language (C++), using backtracking http://en.wikipedia.org/wiki/Backtracking to explore different rules that begin with the same token

7.1 Classes

These are the most important classes of this component.

Figure 7.1: c_parser UML diagram 1

c_context class is a **key part** of Abidos, for each node in the **annotated parse tree**

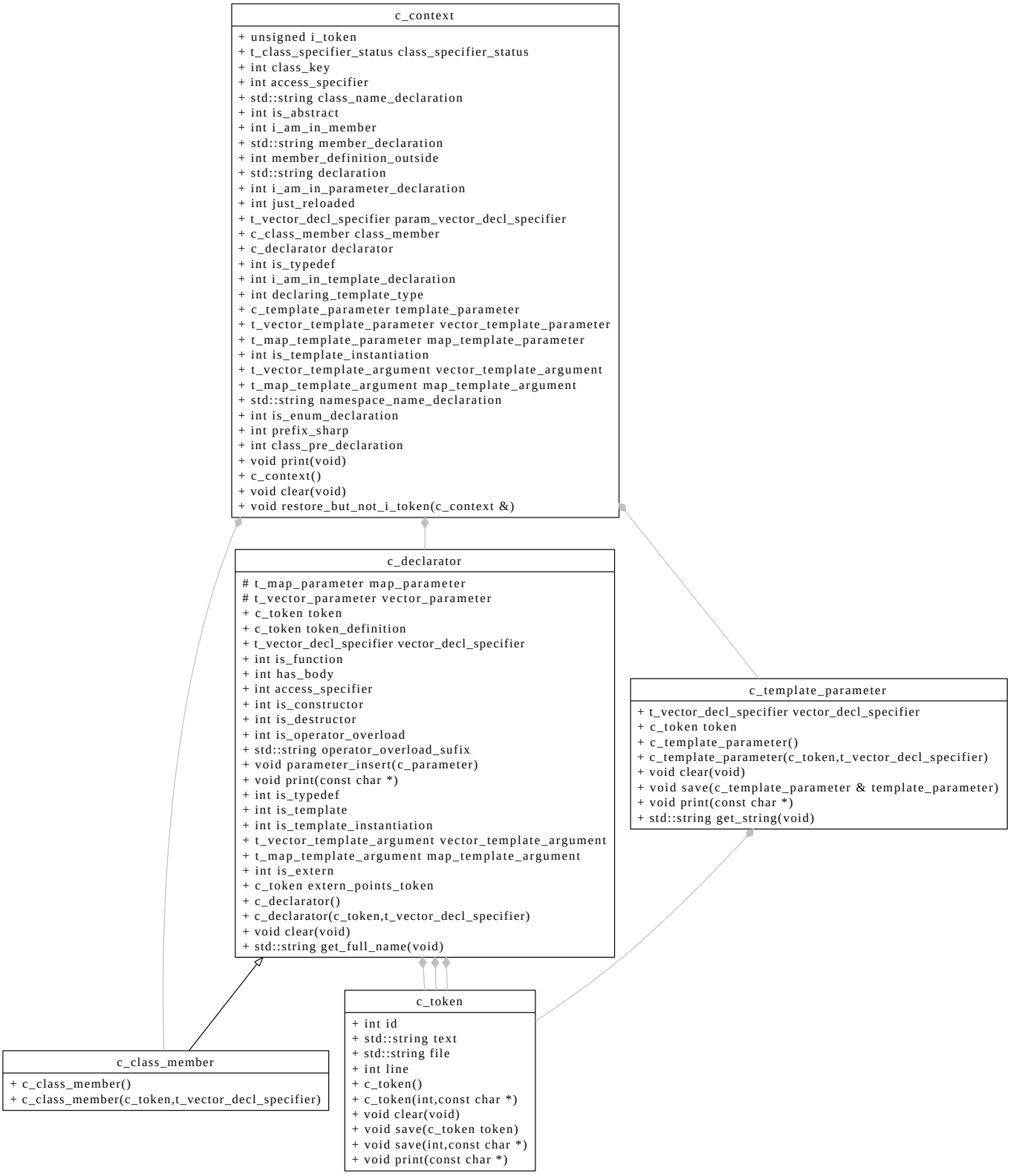

Figure 7.2: c_parser UML diagram 2

Parser uses this classes all the time to get knowledge about tokens and too restore states in the backtracking process.

7.2 Debugging Abidos

In order to see how Abidos works we will use gdb to trace an Abidos execution, in this session i introduce you some topics that will be explained more deeply in next chapters: context, translation rules from bison to descent form . . .

Abidos have an X file, it is used for debugging purposes with gdb:

```
dir .
set print address off
b main
run --includes test_includes/ \
    --out_dir test_out/ \
    --test_all_tokens_consumed_flag \
    --test_original \
    --ts_show \
    --verbose --verbose test/book_01.cpp > test_out/out_book_01.cpp.txt
```

We will use the same previous example:

Example

```
int a;
```

Remember the tree graph with more than 40 nodes generated, and will see how abidos parses this and why the tree is so big.

Execute Abidos with gdb like this:

```
abidos_cpp/processor$ gdb src/abidos_cpp -x X
```

Note
I like use X file, in this way i have all the begin commands in the file and is more easy execute gdb the next day.

First the loader starts to work looking what files Abidos will parse.

The parser starts to work in this line of main.cpp:

```
c_parser_descent parser;

parser.yyparse(file_name);
```

yyparser() function do:

Note
We will see here the most important rules from the parser tree, in the figure of trace tree there are more rules where Abidos can not match tokens.

- lex_file_init().
- a set of the ts (symbols table).
- And calls **translation_unit()** the fist rule of C++ Grammar

```
    c_trace_node trace_node; ❶
    trace_node.set("translation_unit"); ❷

    tokens_vector_clear(); ❸

    if (1 == declaration_seq_opt(trace_node)) { ❹
        return 1;
    }
    ...
```

❶ Here starts the trace tree.

❷ With this trace system, Abidos knows which node is using in each step.

❸ This is the main rule of the grammar we start with a fresh tokens_vector.

❹ in the grammar appears "translation_unit: declaration_seq_opt;" and here is how write this rule in descent form is quite easy.

- declaration_seq_opt rule.
- declaration_seq.

```
declaration_seq: declaration | declaration_seq declaration ;
```

I translated this to the descent form:

```
int c_parser_descent::declaration_seq(c_trace_node trace_node)
{
    trace_graph.add(trace_node, "declaration_seq");

    int result = 0;

    while (1 == declaration(trace_node)) {
        tokens_vector_clear();
        result = 1;
    }

    return result;
}
```

In a descent parser we can not have left recursion (in yacc we can and we want it)

Note
I clear the vector of tokens here, i think there is safe and not more backtracking will bee work with these tokens.

- Abidos goes to declaration rule

Here we can see in the trace tree how Abidos enters in the rules: is_eof, extern_c, preprocessor, template_declaration and can not match the actual token **int**

the original rule is

```
declaration: block_declaration | function_definition |
  template_declaration | explicit_instantiation | explicit_specialization
  | linkage_specification | namespace_definition ;
```

i write it with changing his order of son rules because block_declaration is more cost to check it and i had to add some rules of my own:

1) is_eof, check if we are in the end of the file (we can cut the descent here).

2) preprocessor, in the original grammar do not have this rule, C++ have a external preprocessor and works before compile do, but Abidos do this sort of things in compile phase.

- block_declaration rule.
- simple_declaration, this is a kind of rule not than easy than previous

```
  if ( CLASS_SPECIFIER_STATUS_MEMBER_SPECIFIER
        != context.class_specifier_status ) {
    semantic.clear_decl_specifier();
  }
```

This lines are to know if Abidos is parsing a declaration inside a class scope or not, lets see this with a little example:

```
class A{
  int a;
  int f(int i);
}
```

When Abidos parses **int a;** this will be a part of A class like an attribute, but when Abidos are parsing **int i** this will be a part of **f** declaration not a part of A directly, this 2 variables **a** and **i** are parsed in the same rules and with the context Abidos know what they are.

Note
context allow Abidos to know where he is, and what is the semantic value of a token.

- decl_specifier_seq_opt
- decl_specifier_seq, a little hack in this rule to put in descent way, the original rule in left recursion fashion is:

```
decl_specifier_seq: decl_specifier_seq_opt decl_specifier;
```

There is a indirect recursion by left calling **decl_specifier_seq_opt** and this call decl_specifier_seq again. The goal of this recursion is iterating **decl_specifier** for example "long int . . . ", if i would write this rule exact like his original form i will have a stack overflow like this:

```
//
// i drop the indirect recursion for establish a clear example
// the execution is quite similar
//
int decl_specifier_seq(void)
{
  decl_specifier_seq(); ❶
  decl_specifier(); ❷
}
```

❶ program call this one time, and again and again . . . until stack overflow.

❷ program never reach this line

The easy solution taken in Abidos is:

```
    while (1 == decl_specifier(trace_node)) { ❶
        result = 1;
    }
```

❶ iteration over **decl_specifier** reached with a while.

- decl_specifier, here we can see another trick

```
    const int vector_id[]={';' , ')', COLONCOLON,IDENTIFIER, '~', '#', -1};
    if (preanalisys_has_one(  vector_id,trace_node) ) {
        return 0;
    }
```

With this lines of code we can **prune** sub-trees and Abidos save resources in this rule we know that a **decl** can not be a **#** or a **;** therefore if some of this symbols are present we prune and does not get more deep.

- type_specifier
- simple_type_specifier, here is a lot of things but where **int** is matched is in this lines

```
    const int vector_id[]={
        CHAR, WCHAR_T, BOOL, SHORT, INT, LONG
        , SIGNED, UNSIGNED, FLOAT, DOUBLE, VOID, -1
    };

    if (token_is_one(vector_id,trace_node) != 0) { ❶
        result = 1;
    }
```

❶ **int** is matched here and this method, this method calls **is_one** and this calls trace_graph.token_is_add here.

Then there are a lot of code about scopes of classes and templates parsing that will see forward for this example is not used.

Abidos put the decl **int** in the semantic class.

```
  semantic.push_back_vector_decl_specifier(decl);
```

And this rule returns 1 indicating success, and In this case int should be consumed therefore **context = context_tokens.restore();** are not used to restore the context and forcing to process **int** again with other rules.

- decl_specifier_seq, Abidos try to iterates again with decl_specifier but now **i** token is an IDENTIFIER.
- init_declarator_list_opt
- init_declarator_list, in this rule we can see another trick to development rules in descent form:

```
    c_context_tokens context_tokens(context); ❶
    c_context_tokens context_good_way(context); ❷
```

❶ here Abidos save context to restore it if the rule don not match the token.

❷ here is another context saver in a declaration.

Lets see this last point C++ allows to put 1 or more, IDENTIFIERS separated by **,** for example

```
  int a, b, c;
```

I use **;** like a terminator for this rule but i should restore put the token **;** on the context queue again to be parsed in **simple_declaration** rule, for that is this if:

```
        if ( token_is(';', trace_node) ) {
            // yes i restore here to consume ';' more up in the tree
            context = context_good_way.restore();
            return 1;
        }
```

And Abidos prune this rule with that.

Note
I would use **preanalisys_has_one** to do this prune but is a little more inefficiency because it saves and restores the context in each check.

- init_declarator
- declarator
- direct_declarator, there are a lot of code here but in this case it calls next rule.
- declarator_id, this rule is uses to declare constructors too, but now it calls next rule.
- id_expression
- unqualified_id, this rule is used to declare destructors, but now it calls next rule.
- identifier, Abidos put the identifier in the semantic

```
  semantic.identifier(context, c_token_get());
```

7.3 Meaningful parts of parser

Now we can go to see what parts have Abidos parser to do his work, all these are the core of the project and understand them is a **must** to begin to hack Abidos.

Parser function token_next() uses yylex() function to get tokens from lexer, but parser needs a buffer to store these tokens and tokens_vector is that buffer. To explain this more deeply lets see an example.

Abidos have a big grammar because C++ is a complex language, the first step is hack that grammar and do a little grammar in order to have a clear explanation, the mechanics of the whole set of Abidos rules are the same. we will work with this little grammar:

```
S -> VARIABLE | FUNCTION
VARIABLE -> int <IDENTIFIER> ";"
FUNCTION -> int <IDENTIFIER> "(" ")" ";"
```

this grammar can match expressions like these:

```
int i;
int f();
```

We can translate these rules to Abidos fashion and the result is these method rules of parser:

- S rule

Has the initialization of the trace system and the calls to the other rules.

```
int c_parser_descent::S(void)
{
    c_trace_node trace_node; ❶

    trace_node.set("S");

    tokens_vector_clear(); ❷

    if (1 == VARIABLE(trace_node)) { ❸
        return 1;
    }

    if (1 == FUNCTION(trace_node)) {
        return 1;
    }

    return 0;
}
```

❶ here is where trace tree begins, with this tree we will see the rules used by Abidos to parse an example.

❷ parser uses this vector like a buffer between lexer and himself, it will full explained in a moment.

❸ this is the codification of a call rule.

- VARIABLE rule

This rule tries to match expressions like **"int i;"**.

```
int c_parser_descent::VARIABLE(c_trace_node trace_node)
{
    trace_graph.add(trace_node, "VARIABLE"); ❶
    c_context_tokens context_tokens(context); ❷

    token_next(trace_node.get_tab()); ❸
    if ( token_is_not(INT, trace_node) ) { ❹
        context = context_tokens.restore(); ❺
        return 0;
    }

    token_next(trace_node.get_tab());
    if ( token_is_not(IDENTIFIER, trace_node) ) {
        context = context_tokens.restore();
        return 0;
    }

    token_next(trace_node.get_tab());
    if ( token_is_not(';', trace_node) ) {
        context = context_tokens.restore();
        return 0;
    }

    return 1;
}
```

❶ this is how trace system knows what rule is working.

❷ here we save the context in the point 5 we will see why is important.

❸ in this function parser read a token from the buffer or from the lexer.

❹ this is the way to check if the actual token is an **INT**.

❺ suppose the token read is **VOID** we need restore the context in order to can read **VOID** in other rule if we do not restore the context we would lose the actual token.

- FUNCTION rule

This rule tries to match expressions like **int f();**.

```
int c_parser_descent::FUNCTION(c_trace_node trace_node)
{
    trace_graph.add(trace_node, "FUNCTION");
    c_context_tokens context_tokens(context);

    token_next(trace_node.get_tab());
    if ( token_is_not(INT, trace_node) ) {
        context = context_tokens.restore();
        return 0;
    }

    token_next(trace_node.get_tab());
    if ( token_is_not(IDENTIFIER, trace_node) ) {
        context = context_tokens.restore();
        return 0;
    }

    token_next(trace_node.get_tab());
    if ( token_is_not('(', trace_node) ) {
        context = context_tokens.restore();
        return 0;
    }

    token_next(trace_node.get_tab());
    if ( token_is_not(')', trace_node) ) {
        context = context_tokens.restore();
        return 0;
    }

    token_next(trace_node.get_tab());
    if ( token_is_not(';', trace_node) ) {
        context = context_tokens.restore();
        return 0;
    }

    return 1;
}
```

This rule is pretty similar to the previous one except in this rule we need match "**(**" and "**)**" before "**;**".

Lets see an example of a variable declaration:

```
int i;
```

We can see how all the tokens are matched in the VARIABLE rule.

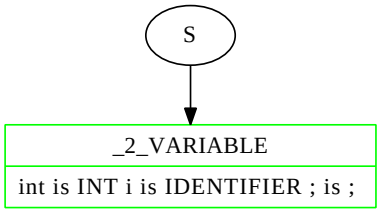

Figure 7.3: variable trace tree

Now lets see an example of a function declaration:

```
int f();
```

now the VARIABLE rule does not match all the tokens and Abidos executes the another rule FUNCTION.

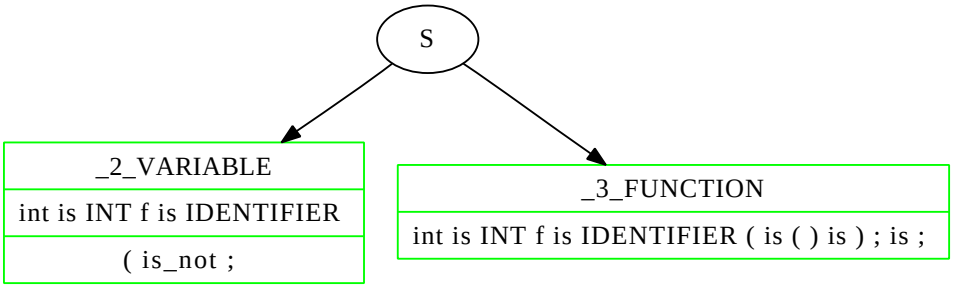

Figure 7.4: function trace tree

When Abidos executes VARIABLE rule here get from lexer the tokens **int f "("** and stores these in **tokens_vector** a buffer queue each time **token_next()** is called, when VARIABLE rule tries to match **"("** like a **";"** fails and restore context, when FUNCTION rule starts to work token_next() use **tokens_vector** to get the tokens starting in **context.i_token** therefore **int f "("** are processed another time, when **"("** is read the next time token_next() is called it calls **yylex()** to obtain **")"** from lexer because **context.i_token** reaches the end of **tokens_vector**.

In the next diagram we can see how S calls VARIABLE rule and inside of the rule execute the **token_next()** in order to obtain the tokens from lexer and putting them in the queue **tokens_vector**.

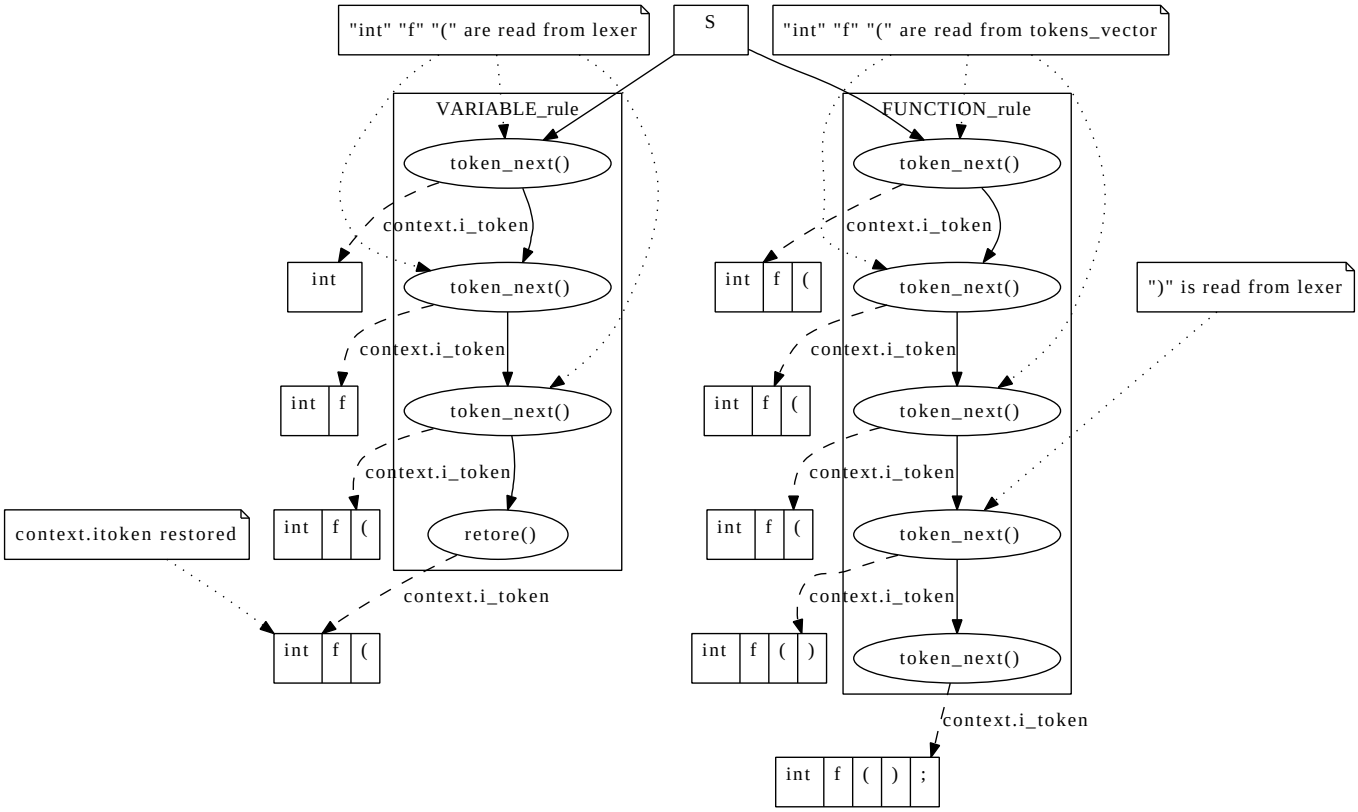

Figure 7.5: token_next() calls and buffer changes

context have **i_token** which is an index to the actual token, token_next() use **i_token** to get tokens from buffer tokens_vector while i_token is < tokens_vector.size() and when restore() is called **context.i_token** points to **int** again and then FUNCTION rule can parse the same tokens and ")" ";" too.

In some points of the grammar **tokens_vector_clear()** can be called to clear the buffer but you must be sure that you won need those tokens anymore.

7.4 How do token_get() token_next() do their work ?

Now we already know that this is a very important method of Parser lets go to see what others things do.

We studied in mangling chapter, we can have in the symbols table "A::B" to store a class B declared in this form:

```
class A
{
  class B
  {
    public:
      void f(void);
  };
};

A::B b; ❶
```

❶ in this case the lexer would pass to parser the tokens "A" "::" "B" parser joins these tokens to have an class identifier "A::B" that if search in symbols table it would returns the related class.

- token_get()

First we need see the pseudo-code of token_get():

```
int token_get()
{
  return tokens_vector[context.i_token].id;
}
```

We can see that the **context.i_token** points to the actual token in this state or node in the syntax tree.

- token_next()

The pseudo-code of token_next is:

```
token_next()
{
  if( context.i_token < tokens_vector.size() ) {
    ++context.i_token;
    return;
  }

  t = yylex(); ❶

  while(1) {
    check_abidos_command(t); ❷
    switch drop_head_namespace(tab, token) {
      0: break;
      1: continue; // in using namespace
      2: return;   // is in using namespace but not is a N::
    }
  }

  colon_colon_chain_process(token); ❸

  if (IDENTIFIER == t) {
    check_identifier(tab, token);
  }

  tokens_vector.push_back(token);

  context.i_token = (tokens_vector.size() - 1);

  return;
}
```

❶ reached this point the tokens is read from lexer.

❷ maybe is an internal command of Abidos.

❸ put in the string attribute **colon_colon_chain** consecutive tokens like **A::B** uses in check_identifier().

- drop_head_namespace()

when we have something like:

```
using namespace N;

class B{
  C n1;
  N::C n2;
};
```

Abidos cuts **N::** because is reconstructing more later in **void c_parser_descent::check_identifier()** because N is in the vector **semantic.vector_using_namespace**

Note
This will change in the future because is a work-around.

- check_identifier()

Lexer not distinguish between IDENTIFIER and CLASS_NAME because lexer does not check that with the symbols table therefore parser do it in this method.

```
check_identifier()
{
    if ( 1 == identifier_search_with_type(tab, yytext, token) ) {
        return;
    }

    //from this point i cut the code because is too much long
    //and you can search it in the code src/parser_descent.cpp

    check destructors A::B::~B() using chain_is_tail()

    check if is template type // template <class T> --> T

    check declarations of members functions outside of his class
      but inside of the namespace

    check the using namespace

    check context.namespace_name_declaration + "::" + yytext;

    check class defined inside the current class

    check preprocessor hack "#" + yytext;
}
```

Chapter 8

Trace system

Abidos when finishes the parsing process can generates a diagram tree with the rules used, you should use it only with small examples. In c_parser_descent you can see how all rules pass trace_node and add it to trace_graph

```
int c_parser_descent::extern_c(c_trace_node trace_node)
{
    trace_graph.add(trace_node, "extern_c"); ❶
...
}
```

❶ that is how trace system knows what rule has been called.

c_trace_graph trace_graph

Here is the output tree drawn by Abidos trace system for the previous example:

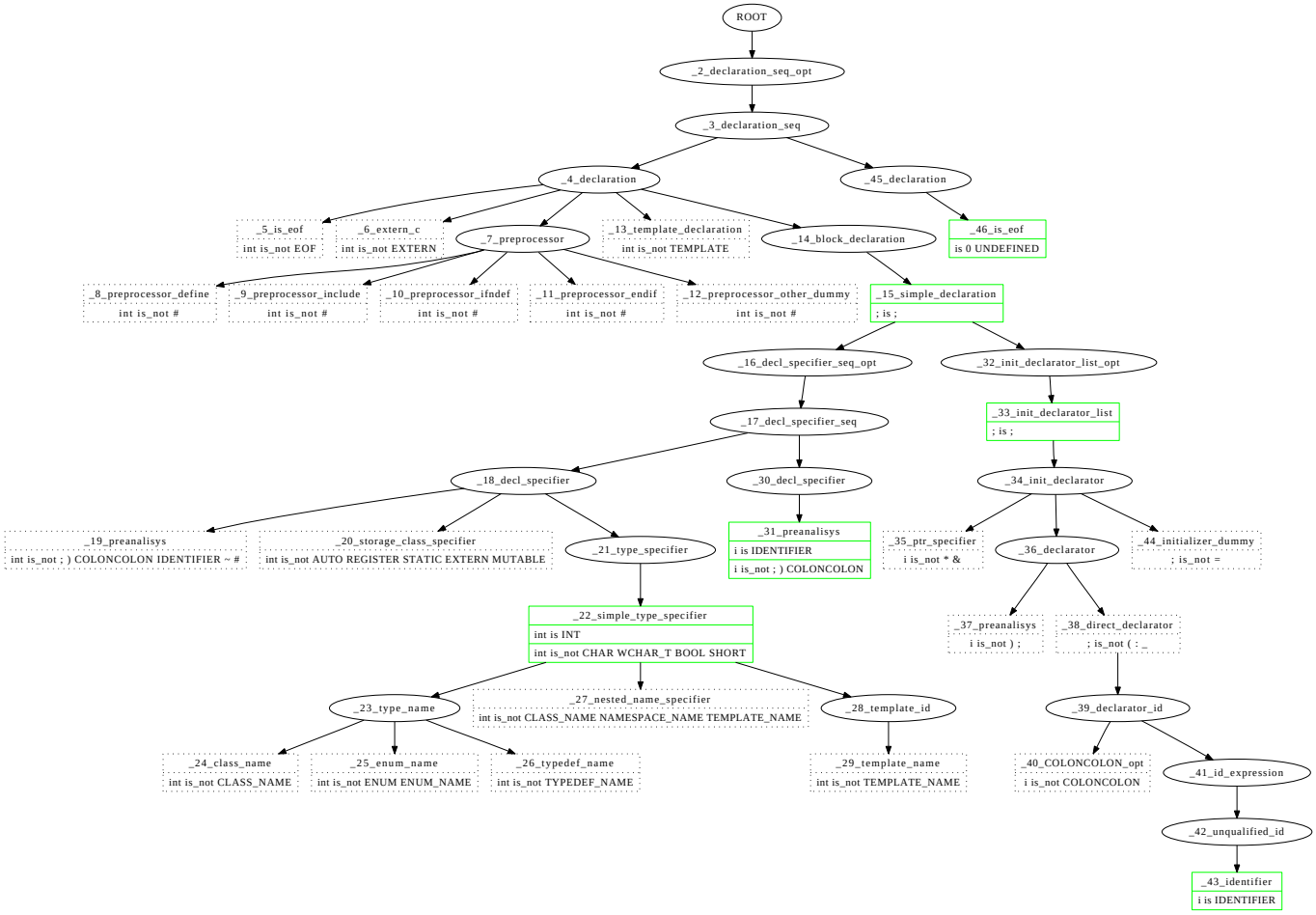

Figure 8.1: trace tree example

Is a super-set of an **annotated parse tree** we can see the successfully branches and the unsuccessfully.

Nodes begin with *<number>*: is the order in which abidos have processed it.

Shape of nodes have these meanings:

- ellipsis, are non terminal rules, like this

```
declaration_seq:
        declaration
        | declaration_seq declaration
        ;
```

- box, are rules when some terminal tokens are processed, it can have no terminal rules too, perimeter line of nodes have these meanings:

1) dotted in this nodes the terminals not matched, for example in:

```
      _13_template_declaration
      int is not TEMPLATE ❶
```

❶ Abidos was trying to match "int" with the reserved word "TEMPLATE".

2) green line: in this nodes some terminals matched, for example in:

```
_15_simple_declaration
; is ; (1)
```

(1) Abidos was trying to match ";" with ";" and it matched.

You can see parsing a little example like that generate a tree with more than 40 nodes, descent parsing is more inefficient than yacc parsing, but i will try to improve this with pruning methods. And remember C++ is a complex grammar to do in yacc because is a tough task write his grammar using LALR.

If you wan see the **annotated parse tree** without only the successfully branches or at least branches where some tokens has been matched because in some example one successfully branch can be the son of a node pruned by backtracking and Abidos shows it. You can open the **_pruned** version like this:

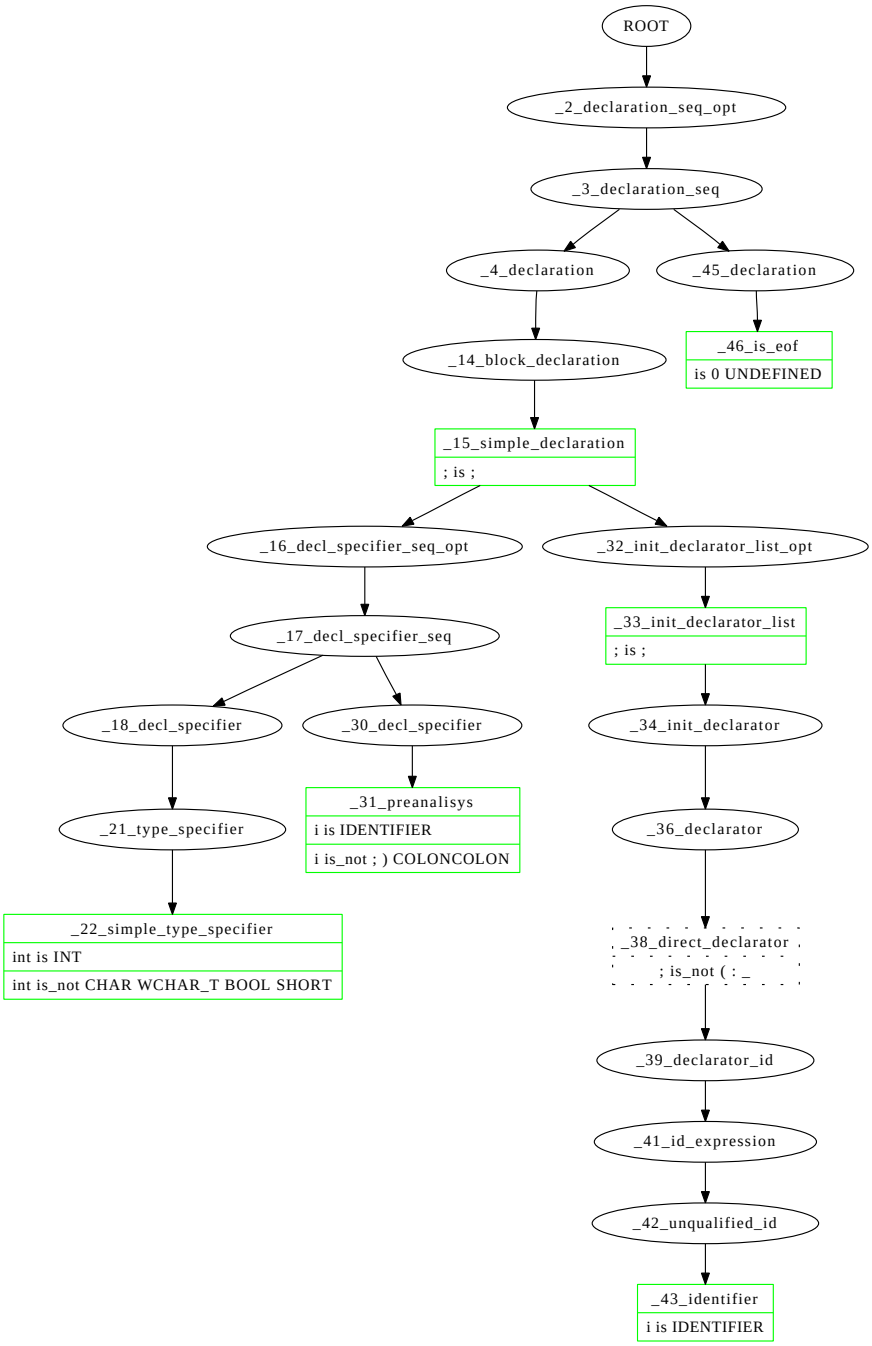

Figure 8.2: trace tree pruned example

Chapter 9

Context

Now you had a quick look about how parser works and you know about trace system, we can understand **Context** that is a very important part of Abidos, in order to know the need of Context lets see some things Abidos needs to know when he is parsing a C++ project.

9.1 How do context do it?

We start with this example:

test/book_02.cpp

```
int a1; ❶

class A
{
        int a2; ❷
};
```

❶ here a1 is a free variable declaration.

❷ a2 is a member variable of A class.

When Abidos enters in a Class Scope he has to know that he is inside of a Class in all rules executed in that class, in the previous example How do Abidos know **a2** is a member of **A class** ?

To answers this question first take a look of the **annotated parse tree** for this example.

We can see the **annotated parse tree** writing in a console:

```
make test_run
xdot_run.py test_out/trace_book_02.cpp_urls_pruned.gv
```

You will see a tree like this:

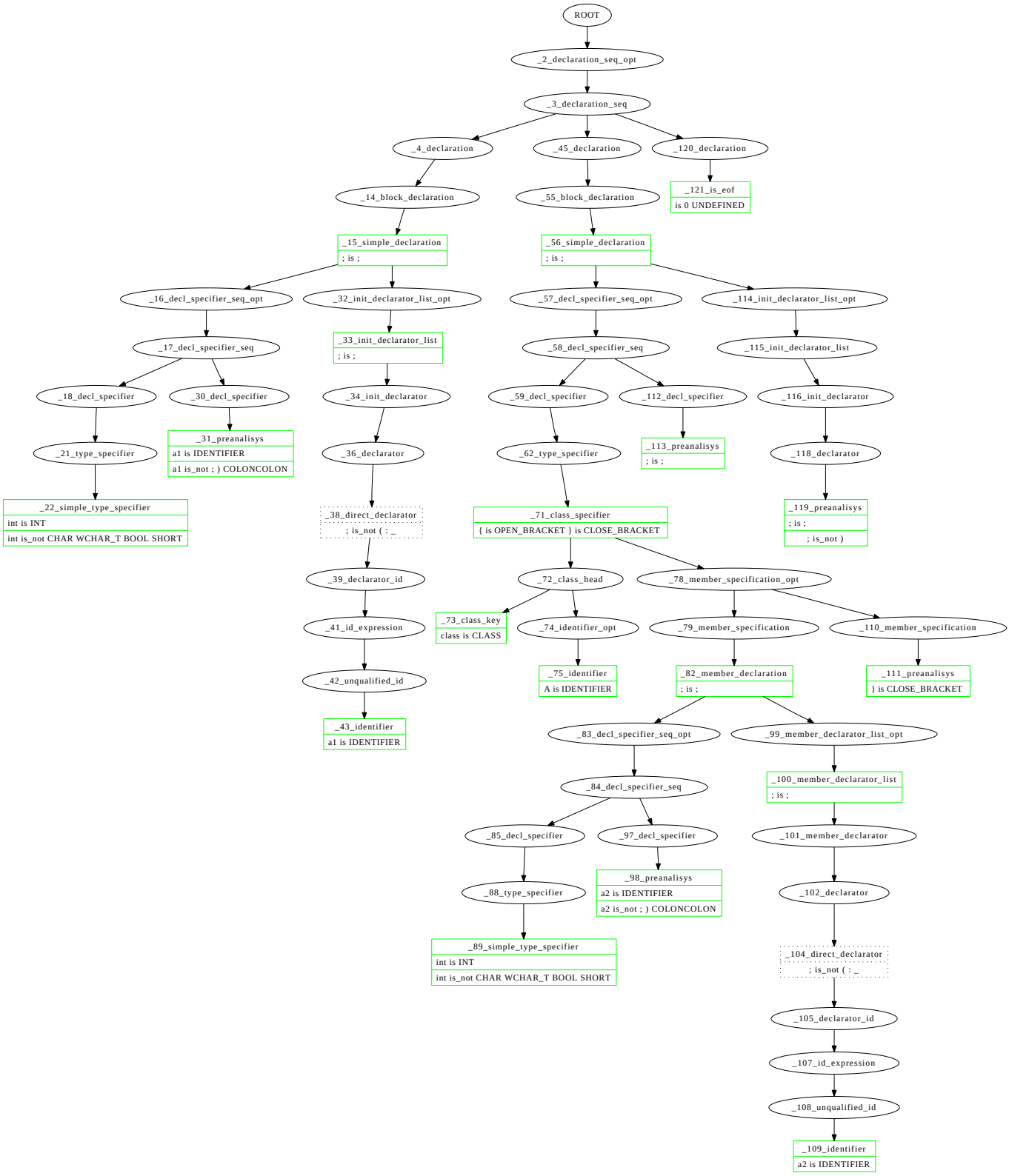

Figure 9.1: trace_book_02.cpp_urls_pruned

- How Abidos parses **A**

When Abidos enters in **_71_class_specifier** the context set the next value:

```
int c_parser_descent::class_specifier(c_trace_node trace_node)
{
  ...
  context.class_specifier_status = CLASS_SPECIFIER_STATUS_IDENTIFIER; ❶
  ...
}
```

❶ with this line Abidos knows he are inside of a Class scope.

Later in the rule **_75_identifier** Abidos parses **A** identifier and calls

```
int c_parser_descent::identifier(c_trace_node trace_node)
{
  ...
  semantic.identifier(context, c_token_get());
  ...
}
```

Semantic process it:

```
void c_semantic::identifier(c_context & context, c_token token)
{
    ...
    if (CLASS_SPECIFIER_STATUS_IDENTIFIER ==
            context.class_specifier_status) {
        class_specifier_identifier(context, token); ❶
    }
    ...
}
```

❶ by the **context** knows **A** is the name of a Class.

Note
context is a glue to join the knowledge of parser and semantic.

- When Abidos parses **a2**

first in _101_member_declaration context set:

```
int c_parser_descent::member_declaration(c_trace_node trace_node)
{
    ...
    context.class_specifier_status =
        CLASS_SPECIFIER_STATUS_MEMBER_SPECIFIER;
    ...
    decl_specifier_seq_opt(trace_node); ❶
    ...
    context.class_specifier_status =
        CLASS_SPECIFIER_STATUS_MEMBER_DECLARATOR; ❷
    ...
}
```

❶ this rule is to process the **int** token.

❷ this state will be use by semantic.

and when Abidos uses _109_identifier and this rule calls **semantic::identifier**:

```
...
if (CLASS_SPECIFIER_STATUS_MEMBER_DECLARATOR ==
        context.class_specifier_status) {
...
        class_member_declarator(context, token); ❶
...
```

❶ semantic process **a2** like a member.

A and **a2** has been processed by the same rule **c_parser_descent::identifier** but the context is the key for semantic can do his work, and put **A** like a class in the symbols table and **a2** like a a member of **A**.

In this state machine we can see the states and the syntactic rules that change the states:

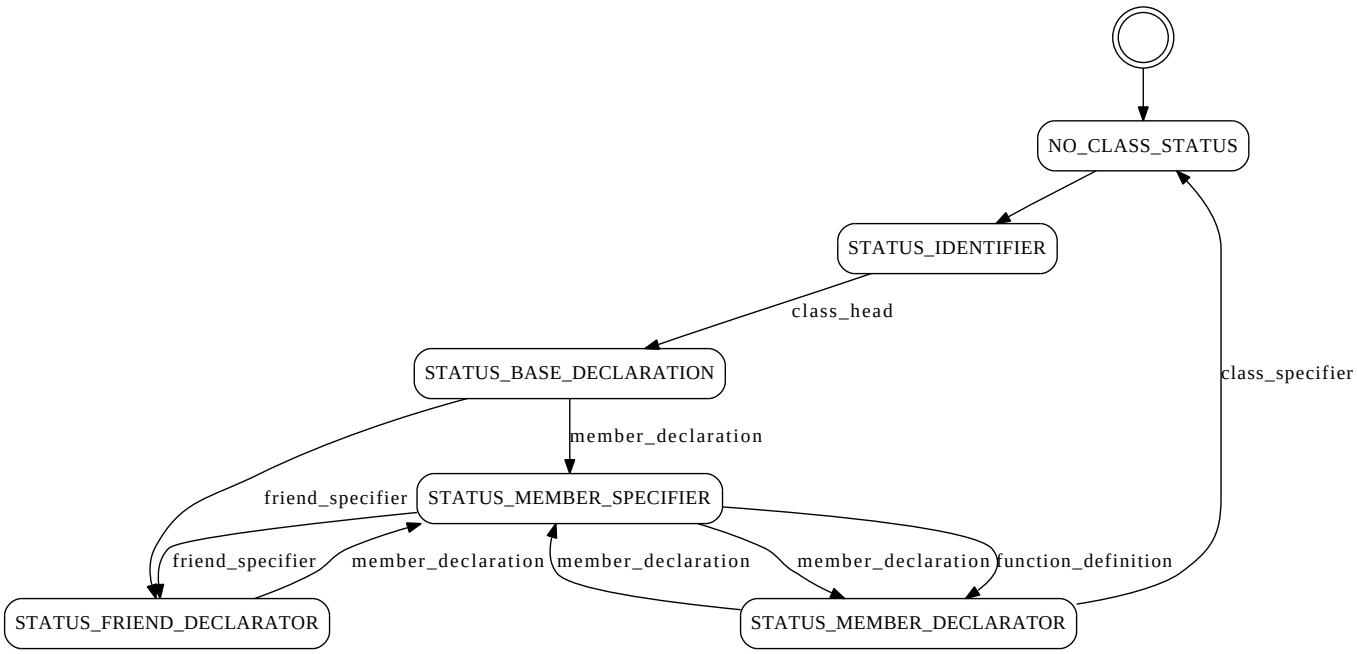

Figure 9.2: state machine

9.2 Parts of context

- unsigned i_token: is a pointer to **tokens_vector**
- enum t_class_specifier_status class_specifier_status: with this semantic know where the syntactic are:
  - NO_CLASS_STATUS : parser is out of a class.
  - CLASS_SPECIFIER_STATUS_IDENTIFIER : parser passed for class_specifier rule.
  - CLASS_SPECIFIER_STATUS_BASE_DECLARATION : parser passed for class_head rule and now is inside of base_clause_opt rule.
  - CLASS_SPECIFIER_STATUS_MEMBER_SPECIFIER : parser passed for member_declaration rule and now is inside of decl_specifier_seq_opt rule.

  - CLASS_SPECIFIER_STATUS_MEMBER_DECLARATOR : parser passed for member_declaration rule and now is inside of member_declarator_list_opt rule.
  - CLASS_SPECIFIER_STATUS_FRIEND_DECLARATOR : parser is inside of friend_specifier rule.
- int class_key : in class_key rule this member specified: CLASS, STRUCT, UNION.
- int access_specifier: in class_key rule this member contains one of: PUBLIC, PRIVATE, PROTECTED.
- string class_name_declaration : contains the name of the class that parser are inside.
- int is_abstract : if a class have a function member with "=0;" in his finish then the class is abstract.
- int i_am_in_member : set inside of member_declaration rule.
- string member_declaration : set by semantic::identifier() when i_am_in_member is 1.
- int member_definition_outside : set by c_semantic::check_coloncolon_member_function() inside of this function where can are in the scope of a class for example inside of the member function **A::f(){}** we are in the scope of A then we need put that in the context and restore the previous context when we reach the **"}"**.
- string declaration : set by c_semantic::identifier() when there is **NO_CLASS_STATUS == context.class_specifier_status** condition.
- int i_am_in_parameter_declaration : set in parameter_declaration_clause rule.
- int just_reloaded : when the parser calls token_vector.clear() this member is set to 1, this is to avoid token_next() go over the first element of vector_tokens.
- t_vector_decl_specifier param_vector_decl_specifier : set in simple_type_specifier, parameter_declaration_list, ELLIPSIS_opt rules when context.i_am_in_parameter_declaration is 1.
- c_class_member class_member : used in the next rules
  - unqualified_id : class_member.is_destructor.
  - operator_function_id : class_member.operator_overload_suffix, class_member.is_operator_overload.
  - _operator : class_member.operator_overload_suffix, class_member.is_operator_overload.
  - direct_declarator : class_member.is_function.
  - consume_array_brackets : so far this is a dummy version, use class_member.is_function.
- c_declarator declarator : used in the next rules :
  - compound_statement : context.declarator.has_body = 1.
  - simple_declaration : context.declarator.has_body = 0.
  - direct_declarator : when the function is free → context.declarator.is_function = 1, if the function is a member class_member is used.
- int is_typedef : used in typedef_specifier rule and in class_specifier some people use typedef class . . .
- int template_status : used in simple_type_specifier, template_declaration.
- int declaring_template_type : used in type_parameter.
- c_template_parameter template_parameter : used in type_parameter.
- t_vector_template_parameter : used in template_id, template_argument rules.
- t_map_template_parameter map_template_parameter : used in template_id rule.
- int is_template_instantiation : used in simple_type_specifier, ptr_specifier, template_id, template_argument, init_declarator rules.
- t_vector_template_argument vector_template_argument : used in simple_type_specifier, ptr_specifier, template_argument

- t_map_template_argument map_template_argument : used in c_semantic: class_member_declarator(), identifier_typedef(), declarator_insert().
- string namespace_name_declaration : used in original_namespace_definition rule.
- int is_enum_declaration : used in enum_specifier rule.
- int prefix_sharp : used in preprocessor_ifndef rule this is to search preprocessor symbols in symbols table with "#" prefix.
- int class_pre_declaration : used in class_specifier rule and semantic::class_pre_declaration_to_declaration()

```
class A;
class A {}; <-- we are in A here
```

Chapter 10

Symbols table

When Abidos are parsing a source C++ file; lexical analyzer give to syntactic analyzer all the tokens founded in the source file, some of this tokens are saved in the table symbols, this is more easy to understand with an easy example:

Example

```
int a; ❶
```

❶ there are 3 tokens here: "int", "a" and ";"

Abidos when process this 3 tokens knows "a" is a variable and his type is "int", this identifier "a" is stored in the symbols table.

10.1 Most important classes of symbols table

The symbols table is a stack of vectors of symbols, in the stack for each scope of the code a vector of his symbols is saved.

In computer programming, a scope is the context within a program in which a variable name or other identifier is valid and can be used, or within which a declaration has effect.

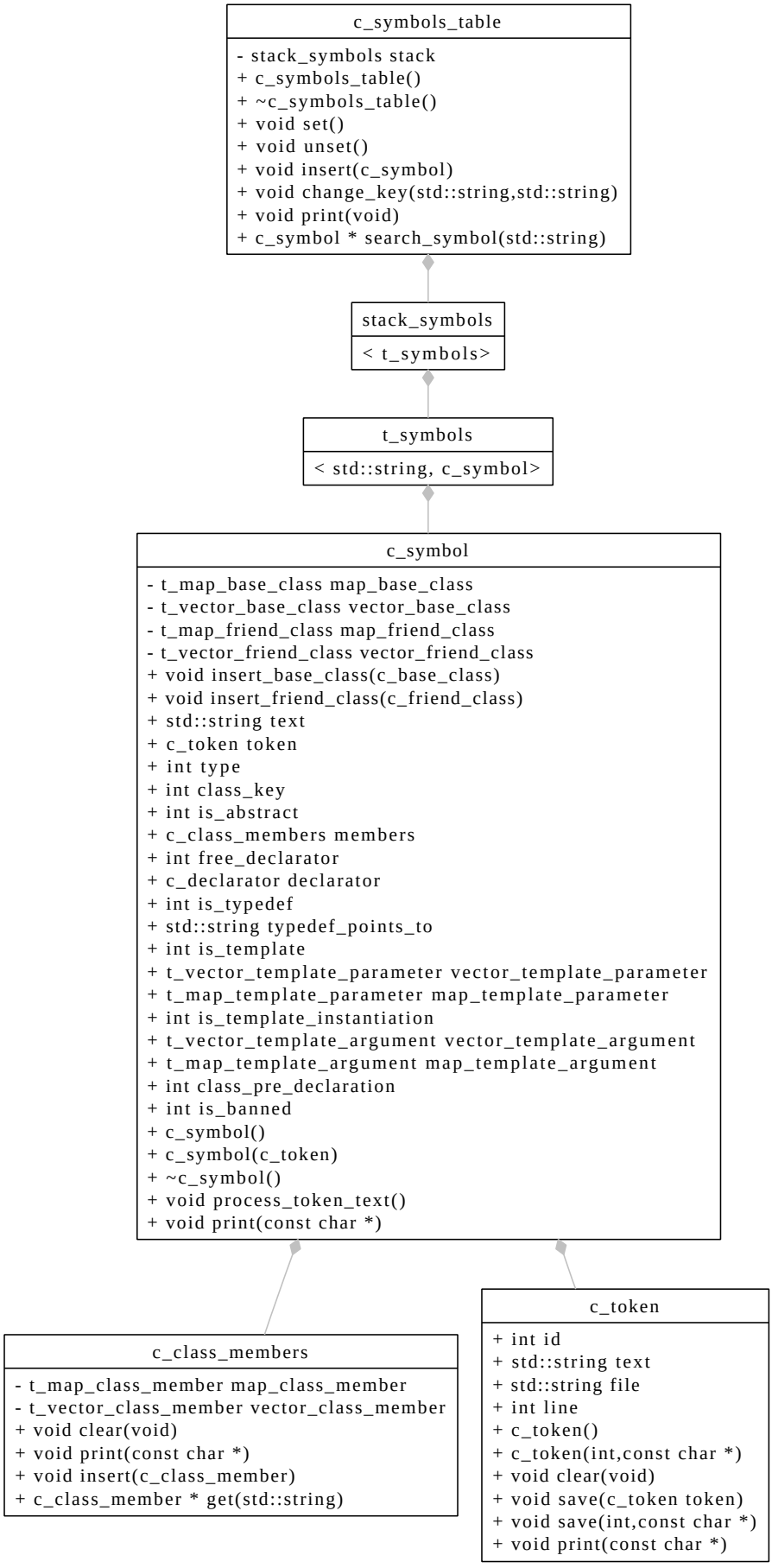

Figure 10.1: c_symbols_table UML diagram 1

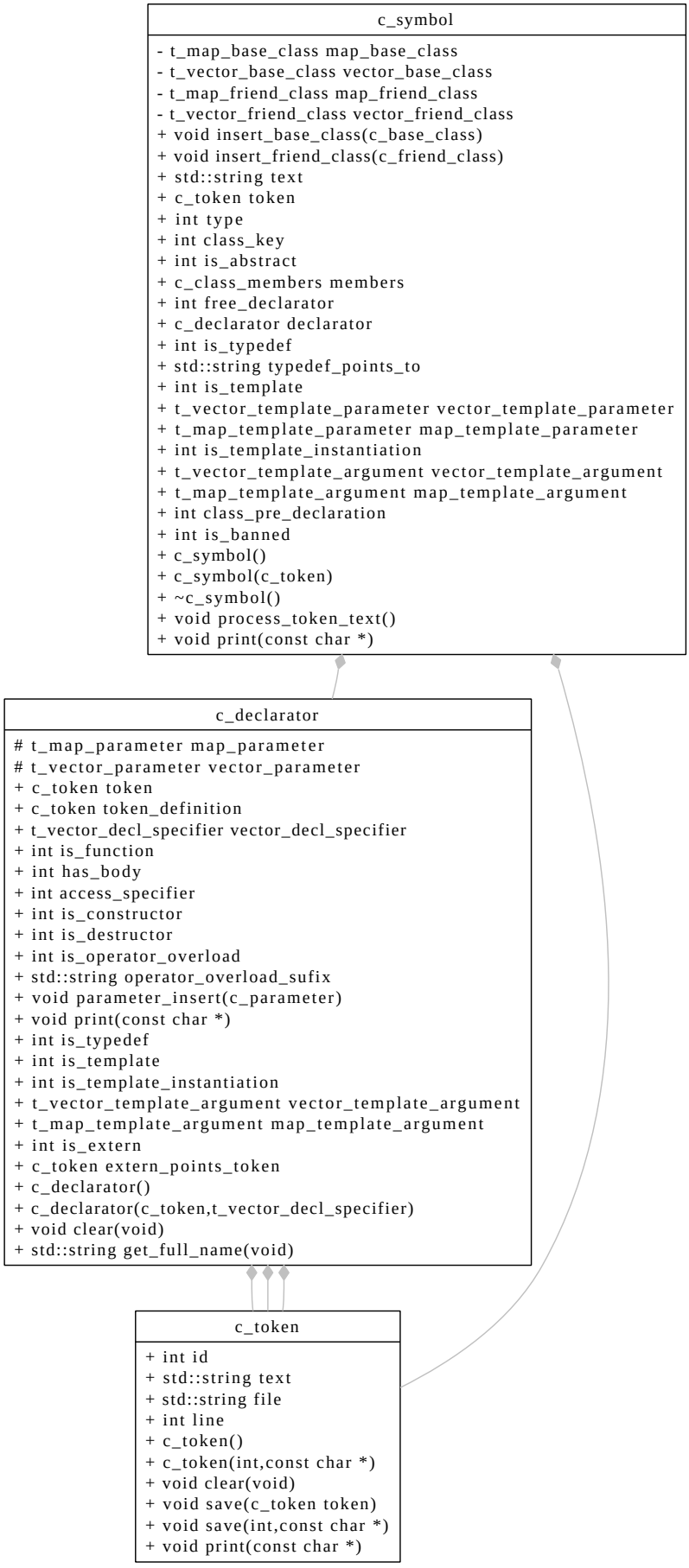

Figure 10.2: c_symbols_table c_declarator UML diagram 2

10.2 Saving context

Abidos uses c_context_tokens to store in which token has being parsing in the begin of a syntactic rule.

c_context_tokens context_tokens(context);

You can read more about context in Chapter context.

Chapter 11

Testing Abidos C++

In the making chapter you can see how Abidos C++ is built, is very important that when we are development a new rule, or we are changing some of this rules or maybe doing other hacking of the project, we should test that the new patches has not introduced new faults.

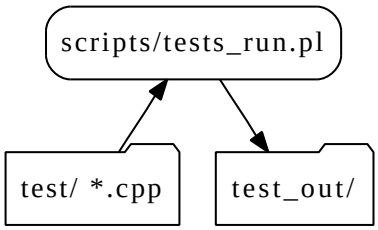

Figure 11.1: make test_run

11.1 Executing the tests

Before you can run the tests you need **abidos_make_process.pl** has been executed.

To run the testing process write this in a console:

```
cd processor
make test_run
```

This command internally invokes the script **scripts/tests_run.pl**; you would call this script in a console but is a clumsy way of do it, because you must pass it a lot of directories that make do for you.

example of test_run output

```
abidos running suit tests [0.0.06]
{
 executable_with_dir [.../processor/src/abidos_cpp]
 test_dir            [.../processor/test/]
 test_includes_dir   [.../processor/test_includes/]
 test_out dir        [.../processor/test_out/]
 abidos_working_dir  [.../processor/.abidos_cpp/]
 [t001.cpp]-> [OK] [t002.cpp]-> [OK] ❶
 ....
  ----------------------------------
  tests OK 17/17
  ----------------------------------
}
```

1 this is the files test executed and her result.

This script generates these files in test_out/ directory:

Table 11.1: test_out files

File	Description
out_t<number>.cpp.dot	UML diagram
out_t<number>.cpp	in the future these file will be like the input file
out_t<number>.cpp.txt	Abidos verbose output with Symbols table printed
trace_t<number>.cpp.gv	tree with the rules used by Abidos C++
trace_t<number>.cpp_urls_pruned.gv	tree with rules and **click & go**
prune.log	shows how many nodes have been cut in the prune process
test_out.log	shows all Abidos calls with his parameters

11.2 Hey wait a moment, testing files are C++ files!

Very clever that is, when Abidos parses a file it use a set of grammar rules to do it, when i write a new rule i write a set of cpp testing files to check that this rules are working correctly, this means:

- UML diagram, it is a hand made test; check the UML output of the test that the testing process generated, for example to check the diagram of the previous file:

```
xdot_run.py test_out/out_t001.cpp.dot
```

And you will see this diagram:

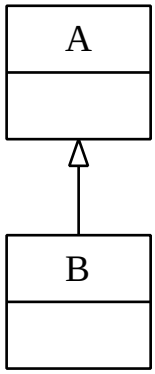

Figure 11.2: out_t001.cpp.dot

- The Symbols table must has all the information that Abidos has parsed, this part until now is hand made you need see the output of the test and studying his parts.

In the future this will be check automatically using the generating files like test_out/out_t001.cpp, this file is generating from the symbols table, and if this file have the same information than the original file the whole process is all right. But so far this file has not all the original tokens... yes exactly i have other priorities.

For now we can see the information gathered in the symbols table, for example in this file **test/t001.cpp**:

```
class A
{
};

class B: public A
{
};
```

You can see his output in **test_out/out_t001.cpp.txt**:

```
 less test_out/out_t001.cpp.txt
 ....
 first[B]/.../t001.cpp:5 \
  id[258]->[IDENTIFIER] text[B] \
  type[265]->[CLASS_NAME] \ ❶
  class_key[300]->[CLASS]
    {
      map_base_class
      {
        first[A]->[PUBLIC][A] ❷
      }
      map_friend_class
      {
        empty
      }
      vector_class_member [0]
      {
      }
      map_class_member [0]
      {
        empty
      }
      free_declarator
      {
      }
    }
 ....
```

❶ B token has been processed with these information, the token is scanned like an IDENTIFIER then the parser knows with the help of some rules that this token should be a symbol of type=CLASS.

❷ Abidos knows than B is a class therefore Abidos checked if B has fathers classes and there is A.

- Abidos generates another useful file for each cpp test file trace tree, in this file we can see the rules executes by Abidos.

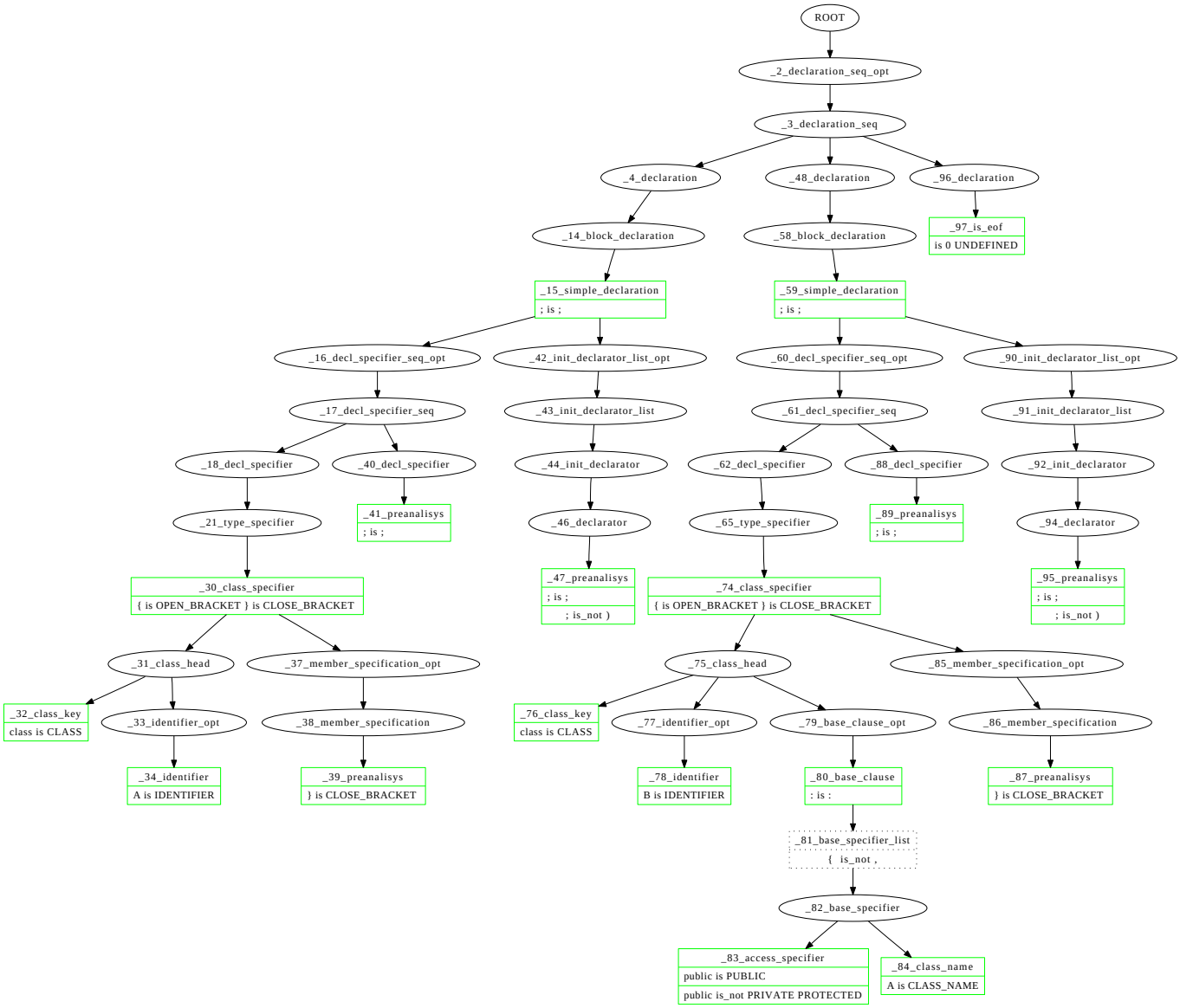

Figure 11.3: trace_t001.cpp_pruned.gv

This tree is generates by Abidos C++ making process in **test_out/** when the make test_run is executed.

One example is **test_out/trace_t001.cpp_pruned.gv**, you can read more about this in the chapter trace, trace component is the part of Abidos that generates annotated parse trees.

Note
You can see the version without pruning if you need more detailed history of the syntactic rules used in the cpp file, just open **test_out/trace_t001.cpp.gv**.

- Abidos must consume all the tokens scanned in the test file, when Abidos is called from the test_run script **scripts/tests_run.pl** the call is like this:

```
/.../processor/src/abidos \
```

```
--includes /.../processor/test_includes/ \
--out_dir  /.../processor/test_out/ \
--test_all_tokens_consumed_flag \ ❶
--test_original \
--ts_show \
--verbose /.../processor/test/book_01.cpp \
  > /.../processor/test_out/out_book_01.cpp.txt
```

❶ If all tokens has been processed and matched generates in his output **##ALL_TOKENS_CONSUMED** then the script knows the cpp file tested is OK. For this all the tokens in the tested file must bee processed successfully in some grammar rules.

This is the only automated test you can see it in the begin of this chapter figure test run output

11.3 Test files and what they testing

When Abidos C++ grows some new rules are written then some new test files are written in order to test this new functionality and to preserve that this new functionality is not broken in the future due to the introduction of more rules or changes did in other rules, will see a useful catalog about functionality and what test files use it.

Note
test files are in abidos_cpp/processor/test directory

Table 11.2: test files rules used part 1

functionality	file test	some important rules tested
abstract class	t044.cpp	direct_declarator
array	t046.cpp	consume_array_brackets
ban symbols	t046.cpp	ban_symbols_on, ban_symbols_off
class access specifier	t010.cpp, t032.cpp	access_specifier (PRIVATE, PROTECTED, PUBLIC)
class declaration	t001.cpp, t002.cpp and almost all test files	class_specifier
class simple inheritance	t001.cpp, t002.cpp	base_clause_opt, base_clause
class multiple inheritance	t002.cpp, t003.cpp	base_clause_opt, base_clause
constructor	t014.cpp, t030.cpp	function_definition, decl_specifier, type_specifier, nested_name_specifier, qualified_id
compositions & aggregations	t023.cpp	
cv_qualifier	t026.cpp	decl_specifier, cv_qualifier (const, VOLATILE)
decl specifier FRIEND	t022.cpp, t024.cpp, t027.cpp	decl_specifier (FRIEND, TYPEDEF)
default parameters values	t045.cpp	parameter_declaration
destructor	t015.cpp, t030.cpp	function_definition, direct_declarator, unqualified_id
enum	t039.cpp	enum_specifier, enum_name
extern	t046.cpp	extern
extern_c	t042.cpp	extern_c
free declarator	t008.cpp	declarator, member_declaration
free declarator multiples decl	t008.cpp	decl_specifier_seq
free declarator multiples declarator	t008.cpp	decl_specifier_seq

Table 11.2: (continued)

functionality	**file test**	**some important rules tested**
free function body	t011.cpp	function_definition, function_body
function specifier	t021.cpp	function_specifier (INLINE, VIRTUAL, EXPLICIT)
namespace	t037.cpp, t038.cpp	original_namespace_definition, named_namespace_definition
namespace using	t046.cpp	using_directive
default parameters values	t045.cpp	parameter_declaration
mangling class names	t028.cpp, t031.cpp	class_name, nested_name_specifier
member variable declarator	t004.cpp, t006.cpp	member_declaration
member function declarator	t005.cpp	member_declaration
member function definition inside	t012.cpp	member_declaration, function_definition, function_body
member function definition inside with dummy body	t013.cpp	member_declaration, function_definition, function_body
member function definition outside	t007.cpp, t016.cpp, t017.cpp, t018.cpp, t029.cpp	member_declaration, function_definition, function_body

Table 11.3: test files rules used part 2

functionality	**file test**	**some important rules tested**
parameter declaration	t006.cpp	parameter_declaration
parameter declaration multiples decl	t009.cpp	parameter_declaration, decl_specifier
parameter ellipsis	t007.cpp	ELLIPSIS_opt (. . .)
pointer operator	t019.cpp	ptr_operator (*,&)
pre-declaration	t044.cpp	
preprocessor include	t034.cpp	preprocessor_include
preprocessor	t040.cpp	#ifndef, #define, #endif
overloading functions	t009.cpp, t012.cpp, t017.cpp, t018.cpp, t019.cpp	
overloading operators	t045.cpp	operator_function_id, _operator
static outside initialization	t043.cpp	storage_class_specifier, init_declarator
storage class specifier	t020.cpp	storage_class_specifier (AUTO, REGISTER, STATIC, EXTERN, MUTABLE)
struct alignment	t046.cpp	direct_declarator
using class like a type	t025.cpp	decl_specifier, type_name
template declaration	t035.cpp, t038.cpp	template_declaration, template_parameter
template instantiation	t036.cpp, t038.cpp	template_argument_list

Chapter 12

Advanced trace

Now you have an idea about how Abidos works, we can start the explanation of the advanced issues. In this chapter we will see examples of C++ code and how Abidos deal with them using all the previous explained topics (trace trees, mangling, symbols table, . . .)

12.1 Inner classes

When we have a class inside another class Abidos uses a mangling method to put the classes in the symbol table, lets go to see the whole process with this example:

abidos_cpp/processor$ vi test/book_inner_class_01.cpp

```
class A {
  class B {
    void f_b(void);
  };
};

void A::B::f_b(void)
{
}
```

The interesting point in this example is how Abidos do the mangling of **B** and the mangling of **f_b**, lets start with the first:

abidos_cpp/processor$ vi test_out/out_book_inner_class_01.cpp.txt

```
    ...
    first[A::B]/.../abidos_cpp/processor/test/book_inner_class_01.cpp:2 \
      id[258]->[IDENTIFIER] \
      text[A::B] type[265]->[CLASS_NAME] \
      class_key[300]->[CLASS]
    ...
```

We are going to trace the code with gdb using this X file:

X file for gdb

```
# gdb src/abidos_cpp -x X
dir .
set print address off
b c_parser_descent::class_key
run --includes test_includes/ \
    --out_dir test_out/ \
    --test_all_tokens_consumed_flag \
```

```
    --test_original \
    --ts_show \
    --verbose test/book_inner_class_01.cpp \
      > test_out/out_book_inner_class_01.cpp.txt
```

Run gdb like this:

gdb execution

```
abidos_cpp/processor$ gdb src/abidos_cpp -x X
```

abidos uses the next rules to reach **A**:

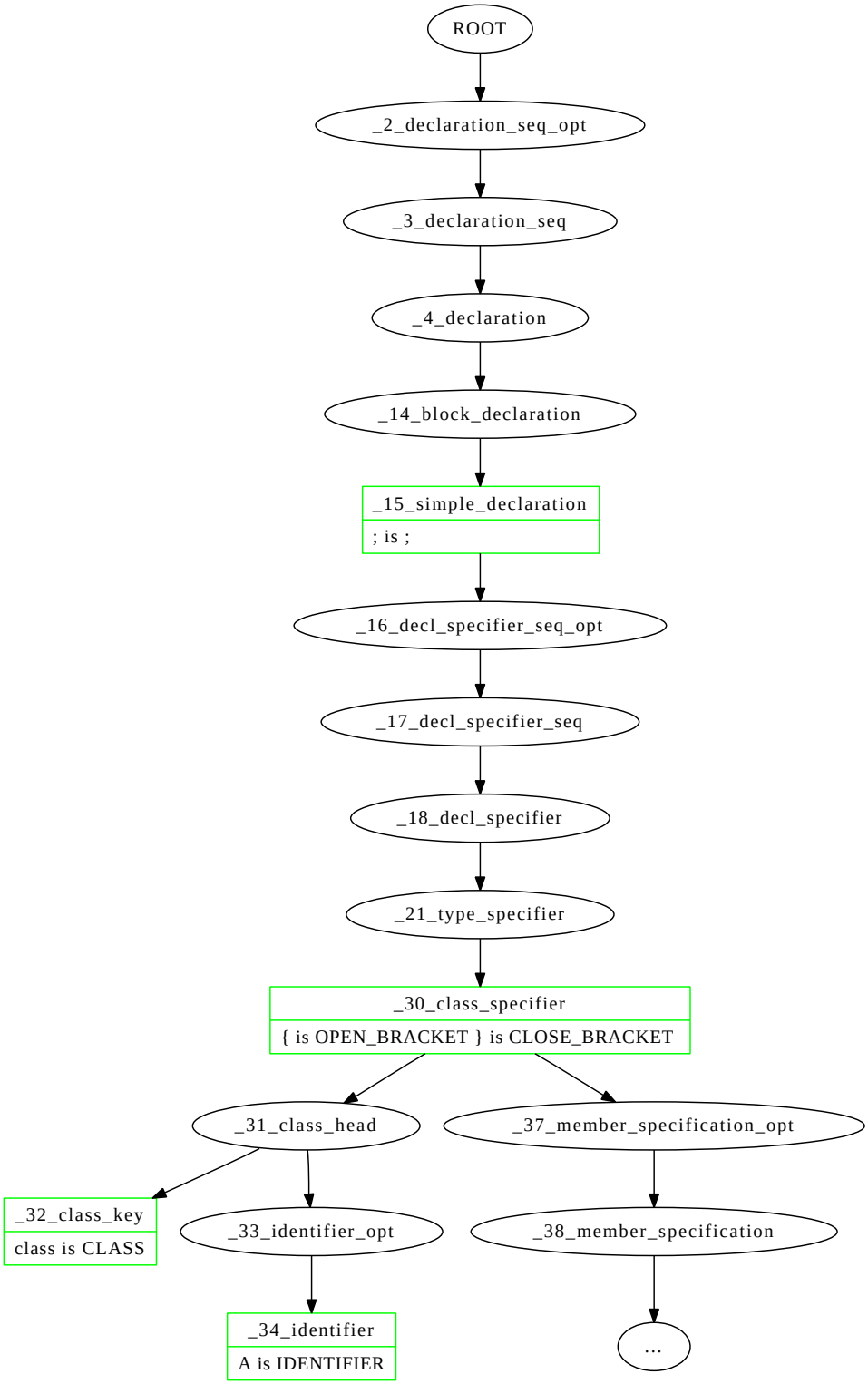

Figure 12.1: classes rules until A

- in **class_key** abidos set context:

```
        context.class_key = CLASS;
```

- in **class_head** abidos set context:

```
    context.class_specifier_status = CLASS_SPECIFIER_STATUS_IDENTIFIER;

    identifier_opt(trace_node);

    context.class_specifier_status =
        CLASS_SPECIFIER_STATUS_BASE_DECLARATION;
```

- when abidos are in identifier rule semantic is called **c_semantic::identifier()** and this code is executed:

```
void c_semantic::identifier(c_context & context, c_token token)
{
    ...
    if (CLASS_SPECIFIER_STATUS_IDENTIFIER ==
            context.class_specifier_status) {
        class_specifier_identifier(context, token);
    }
    ...
}
```

- semantic invokes his method **class_specifier_identifier()** where the next code is executed:

```
void
c_semantic::class_specifier_identifier(c_context & context, c_token token)
{
    if (CLASS == context.class_key) {
        symbol.type = CLASS_NAME;
        symbol.class_key = context.class_key;
    }

    ...

    symbol.is_abstract = context.is_abstract;

    context.class_name_declaration = symbol.token.text;

    ts.insert(symbol);
}
```

The important thing at this moment is this method receives **A** identifier and stores **A** symbol and **context.class_name_declaration** stores **A** value too with that abidos_cpp knows he is in the scope of **A** class.

- now abidos begin to use **member_specification_opt** rule to parse inside of class **A** while expand this subtree (i cut some rules in order to get an easy graph):

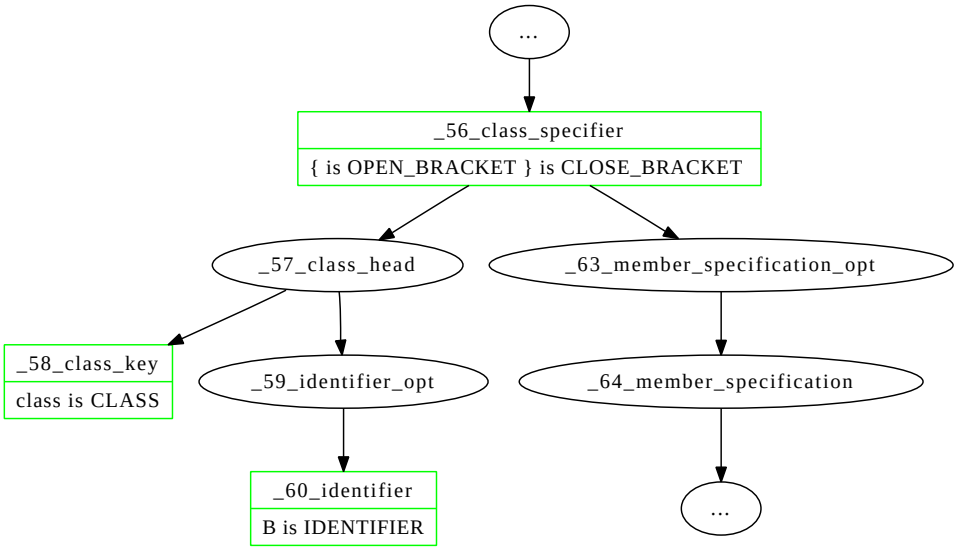

Figure 12.2: inner classes rules until B

- abidos uses again **c_semantic::class_specifier_identifier** this time with **B** token, now context.class_name_declaration has "A" value therefore this code is executed:

```
void
c_semantic::class_specifier_identifier(c_context & context, c_token token)
{
    if ( 0 != context.class_name_declaration.size() ) {
        string s = symbol.token.text;
        symbol.token.text = context.class_name_declaration + "::" + s;
        symbol.text = symbol.token.text;
    } else {
      ...
    }

    symbol.is_abstract = context.is_abstract;

    context.class_name_declaration = symbol.token.text;

    ts.insert(symbol);
}
```

symbol.text now has "A::B" value and is saved in symbols_table, and it is saved in **context.class_name_declaration** too then abidos knows that he is inside of **A::B** scope now.

- here we continue with the last part parsing **f_b** this is his subtree:

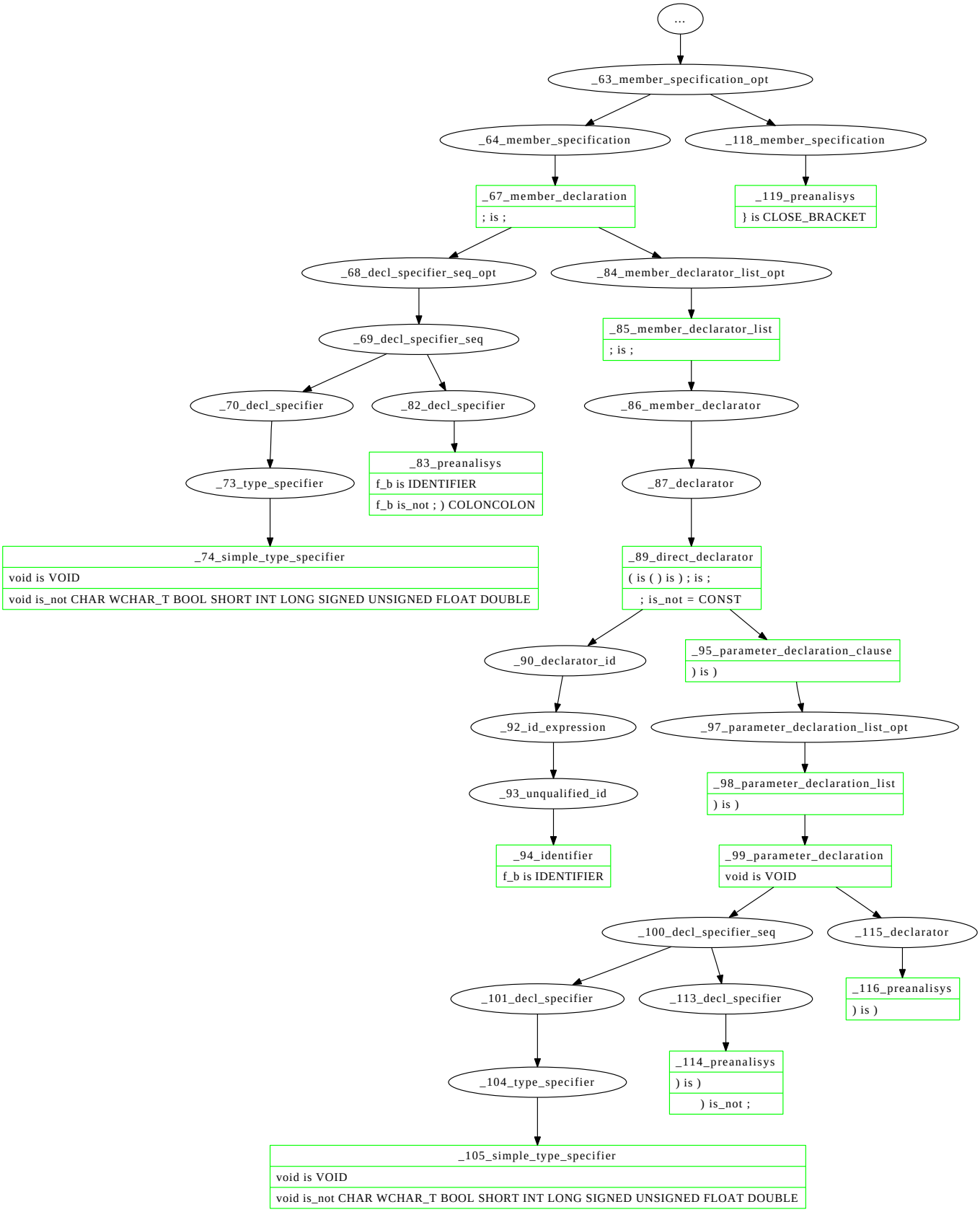

Figure 12.3: inner classes rules declaration of f_b

- first in this subtree we can see in **_74_simple_type_specifier** how **void** is processed:

```
int c_parser_descent::simple_type_specifier(c_trace_node trace_node)
{
    ...
    const int vector_id[]={
        CHAR, WCHAR_T, BOOL, SHORT, INT, LONG
        , SIGNED, UNSIGNED, FLOAT, DOUBLE, VOID, -1
    };

    if (token_is_one(vector_id,trace_node) != 0) {
        result = 1;
    }

    ...

    if (1 == result) {
        c_decl_specifier decl(c_token_get());
        decl.type_specifier = 1;
        decl.has_colon_colon_after = has_colon_colon_after;

        if (1 == context.i_am_in_parameter_declaration) {
          ...
        } else if (1 == context.is_template_instantiation) {
          ...
        } else {
            semantic.push_back_vector_decl_specifier(decl); ❶
        }

        return 1;
    }

    context = context_tokens.restore();

    return 0;
}
```

❶ semantic stores the decls in this case **void**.

If abidos needs to store decls of parameters → then abidos stores them in context. context decls.

- abidos runs member_declaration() → … direct_declarator() → … > identifier(): reads **"f_b"** and calls **c_semantic::identifier()** in this method, due to the **context** abidos enters in this **if**:

```
void c_semantic::identifier(c_context & context, c_token token)
{
    ...
    if (CLASS_SPECIFIER_STATUS_MEMBER_DECLARATOR ==
            context.class_specifier_status) {
        ....

        if (1 == context.i_am_in_member) {
            class_member_declarator(context, token);
            context.member_declaration = token.text;
        }

        return;
    }
    ...
}
```

- in **c_semantic::class_member_declarator()** semantic do:
  - check context status is CLASS_SPECIFIER_STATUS_MEMBER_DECLARATOR.
  - check **context.class_name_declaration** is in the symbols table and get a pointer to the **"A::B"** symbol.
  - check if that symbol is a class.
  - put the type of access (PUBLIC, PRIVATE, PROTECTED) looking if the symbol is an struct or a class or a namespace.
  - stores that information in **context.class_member**:

```
    class_member.access_specifier = context.access_specifier;
    context.class_member = class_member;
```

and calls semantic.identifier():

```
void c_semantic::identifier(c_context & context, c_token token)
{
    ...

    if (CLASS_SPECIFIER_STATUS_MEMBER_DECLARATOR ==
            context.class_specifier_status) {

        ...

        if (1 == context.i_am_in_member) {
            class_member_declarator(context, token);
            context.member_declaration = token.text;
        }

        return;
    }

    ...
}
```

Note
Until now abidos knows it is in "A::B" scope and identifier of method is "f_b" but this identifier is not enough because there can be more methods inside of the same scope with the same identifier therefore abidos needs mangling the name with the types of parameters of the method.

in semantic::class_member_declarator():

```
void
c_semantic::class_member_declarator(c_context & context, c_token token)
{
  ...

    if ( 0 == context.access_specifier) {
        context.class_key = p_symbol->class_key;
        switch (p_symbol->class_key) {
        case CLASS:
            context.access_specifier = PRIVATE;
            break;
        ...
    }

    ...
```

```
    class_member.access_specifier = context.access_specifier;
    context.class_member = class_member;

    return;

}
```

- c_parser_descent::parameter_declaration_list
- c_parser_descent::parameter_declaration(): for now abidos has in context.class_member.token "f_b" but abidos needs to know the parameters to mangling the full name of this method

first calls the rules decl_specifier_seq → decl_specifier → type_specifier → simple_type_specifier and here in context is actualized with the types of "f_b(TYPES)":

```
int c_parser_descent::simple_type_specifier(c_trace_node trace_node)
{
   ...
       if (1 == context.i_am_in_parameter_declaration) {
            context.param_vector_decl_specifier.push_back(decl);
       }
   ...
}
```

abidos stores the parameters decls in context but the decls return of a method or function are stores in semantic semantic decls.

when **parameter_declaration** continues running semantic.identifier is called again :

```
int c_parser_descent::parameter_declaration(c_trace_node trace_node)
{
  ...

    if ( token_is(VOID, trace_node) ) {
        c_token token(IDENTIFIER, (char *) "void");
        semantic.identifier(context, token);
        return 1;
    }

  ...
}
```

in semantic.identifier the next code runs:

```
void c_semantic::identifier(c_context & context, c_token token)
{
  ...
    if (CLASS_SPECIFIER_STATUS_MEMBER_DECLARATOR ==
            context.class_specifier_status) {
        if (1 == context.i_am_in_parameter_declaration) {
            member_param_declarator(context, token);
            return;
        }

        ...
    }
  ...
}
```

in c_semantic::member_param_declarator, by the context (context.class_specifier_status) abidos knows is in CLASS_SPECIFIER_STATUS_MEMBER_DECLARATOR state and executes the next code

```
void
c_semantic::member_param_declarator(c_context & context, c_token token)
{
        ...

        c_symbol *p_symbol =
            ts.search_symbol(context.class_name_declaration);
        if (p_symbol) {

            ...

            c_parameter parameter(token,
                                  context.param_vector_decl_specifier); ❶
            context.class_member.is_function = 1;
            context.class_member.parameter_insert(parameter);

            return;
        }
}
```

❶ set in parameter_declaration() parameter_declaration.

- abidos continues the execution finished this subtree in member_declaration where the whole mangling name of the method can be established:

```
int c_parser_descent::member_declaration(c_trace_node trace_node)
{
    ...

    if ( 1 == function_definition(trace_node) ) {
        //SEMICOLON_opt(trace_node);
        semantic.declarator_insert(trace_node.get_tab(), context);
    ...
}
```

- c_semantic::declarator_insert(): here abidos has all components to set the mangled name of the method → "f_b(void)":

```
void c_semantic::declarator_insert(string tab, c_context & context)
{
  ...
    member_insert(tab, context);
  ...
}
```

- c_semantic::member_insert(): in this method semantic stores the function member **f_b** mangled like **f_b(void)**:

```
void c_semantic::member_insert(string & tab, c_context & context)
    ...

    c_symbol *p_symbol = ts.search_symbol(context.class_name_declaration);
    if (p_symbol) {
        if (0 == p_symbol->class_key) {
          ...
        }
```

```
        ...

        p_symbol->members.insert(context.class_member);
    }

    ...
```

inside of this method is where the full name is compose by **get_full_name()**:

```
void c_class_members::insert(c_class_member member)
{
    ...

    map_class_member[member.get_full_name()] = member;
    vector_class_member.push_back(&map_class_member
                                  [member.get_full_name()]);
}
```

Note
Abidos can not stores the function when **f_b** is reached because due to the polymorphic nature of C++ abidos must parses inside **"(" ")"** in order to compose a name to store **f_b** and have not problems if other **f_b** with other parameters appears in the same scope in this case in the same class.

symbols table has this information:

```
c_symbols_table::print
{
  stack level[0]
  {
    first[A]/.../book_inner_class_01.cpp:1 id[258]->[IDENTIFIER] \
      text[A] type[265]->[CLASS_NAME] class_key[300]->[CLASS]
    {
      ...
    }
    first[A::B]/.../book_inner_class_01.cpp:2 id[258]->[IDENTIFIER] \
      text[A::B] type[265]->[CLASS_NAME] class_key[300]->[CLASS]
    {
      ...

      vector_class_member [1]
      {
       [void] [f_b]( [void] [void])
      }
      map_class_member [1]
      {
       [PRIVATE]: [void] first[f_b(void)]->[f_b]
      }

      ...
    }
  }
```

Note
You can see how **A** and **A::B** are stored in the same real level there is not a real composition here but there is convection composition because **A::B** is part of **A**, when abidos will need search **f_b** he will need search **A::B** and then inside of that class search **f_b(void)**

- before of reach **function_definition** abidos executes **block_declaration** that is because in **declaration** block_declaration is called before, as abidos has backtracking the environment is restored before parser descent through function_definition.

Now we will see how the definition of **A::B::f_b** is parsed:

```
void A::B::f_b(void)
{
}
```

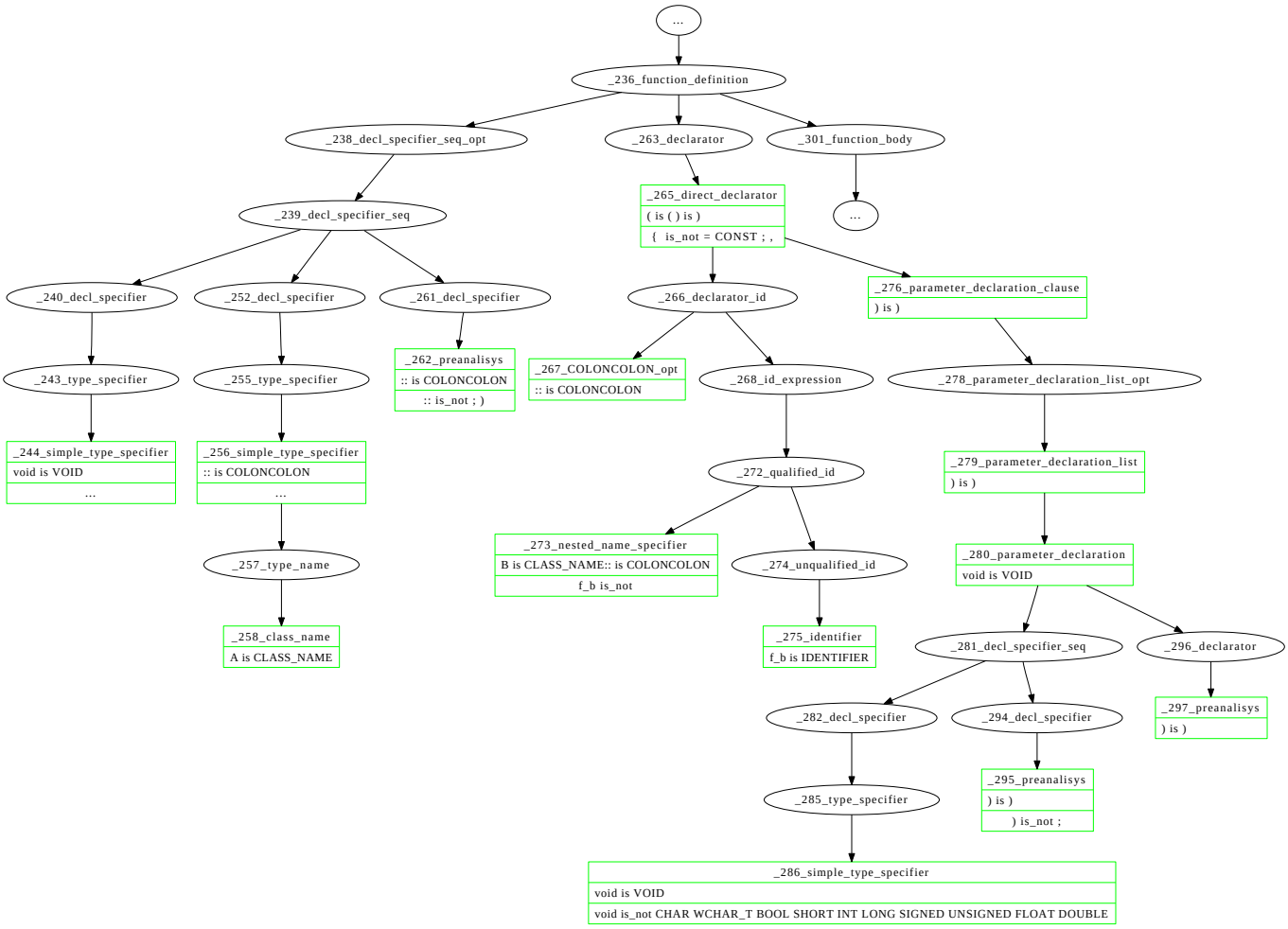

Figure 12.4: inner classes rules definition of f_b

- when abidos enters in the **function_definition** sub tree uses **nested_name_specifier** to concatenate **"A::B"** and stores this in **context.class_name_declaration** in parallel abidos stores the same with **colon_colon_chain_process()**.

Warning
MAYBE THE CODE IN colon_colon_chain_process() IS REDUNDANT AND context.class_name_declaration CAN BE USED INSTEAD OF IT.

- from **c_parser_descent::identifier parser** calls **c_semantic::check_coloncolon_member_function** in this function abidos enters in this **if**:

```
void c_semantic::check_coloncolon_member_function(c_context & context, c_token token)
{
    ...

    if ( 1 == vector_decl_specifier[last].has_colon_colon_after ) {

        context.i_am_in_member = 1;
        context.member_definition_outside = 1; ❶
        context.class_specifier_status = CLASS_SPECIFIER_STATUS_MEMBER_DECLARATOR;

        ...

        return;
    }

    ...
}
```

❶ abidos knows that this is outside of the class declaration block because abidos entered in this if.

Warning
maybe would be better create context.outside and put to 0 when class is matched and put to 1 when the context is restored.

- then **c_semantic::class_member_declarator** is called from **c_parser_descent::identifier**, the next code is executed:

```
void
c_semantic::class_member_declarator(c_context & context, c_token token)
{
  ...
    c_symbol *p_symbol =
        ts.search_symbol(context.class_name_declaration);
  ...
    c_class_member class_member(token,
                                vector_decl_specifier); ❶
  ...
    class_member.access_specifier = context.access_specifier;
    context.class_member = class_member; ❷
  ...
}
```

❶ remember semantic stores decls like the return type **void** of this method semantic_decls

❷ Now abidos have in context the begin of a class_member and can add more symbols inside it and have established the scope in **context.class_name_declaration** with "**A::B**" while abidos is inside of **f_b**.

Note
So far abidos can not know he is inside of **"A::B::f_b(void)"** because he does not parse **"(void)"** yet to can mangling the full name but knows is inside of **a** method with incomplete mangling name beginning with **"f_b"** and in **"A::B"** scope.

- abidos needs to mangling the name of the method for now abidos only knows that it is in the scope of "**A::B**" and abidos is in the definition of a method with a mangling name starting with **f_b** therefore **c_parser_descent::parameter_declaration** calls:

```
void c_semantic::declarator_insert(string tab, c_context & context)
{
      ...
            if (CLASS_SPECIFIER_STATUS_MEMBER_DECLARATOR ==
                if (1 == context.i_am_in_parameter_declaration) {
                    member_param_declarator(context, token);
      ...
}
```

- in **member_param_declarator** the next line is executed:

```
void
c_semantic::member_param_declarator(c_context & context, c_token token)
{
    ...
            context.class_member.parameter_insert(parameter); ❶
    ...
}
```

❶ this is how context get knowledge about the name and the parameters of the actual method.

- in **c_parser_descent::declaration** is executed the next code:

```
int c_parser_descent::declaration(c_trace_node trace_node)
{
    string class_name_bk = context.class_name_declaration;

    if (1 == function_definition(trace_node)) {
        semantic.declarator_insert(trace_node.get_tab(), context);
        context.class_name_declaration = class_name_bk;
        return 1;
    }
}
```

- semantic.declarator_insert

- member_insert here the next code is executed:

```
void c_semantic::member_insert(string & tab, c_context & context)
    ...
    c_symbol *p_symbol = ts.search_symbol(context.class_name_declaration); ❶
    if (p_symbol) {
        ...
        if ( 1 == context.member_definition_outside ) {
            c_class_member * p_member = 0;

            p_member = p_symbol->members.get(
              context.class_member.get_full_name()); ❷

            if ( 0 == p_member ) {
                return;
            }
```

```
            p_member->token_definition = context.class_member.token; ❸
            return;
        }
        ...
    }
    ...
```

❶ abidos get a pointer to the class **"A::B"**.

❷ abidos get a pointer to the method with the name mangled **"f_b(void)"** the mangling is accomplished by string c_declarator::get_full_name(void).

❸ the new information like the line where the method is defined is store in symbols table.

In declaration **get_full_name** is called inside of **c_class_members::insert** get_full_name

12.2 Templates

When abidos_cpp parses templates there are a set of states each of them are stored in **context.template_status**.

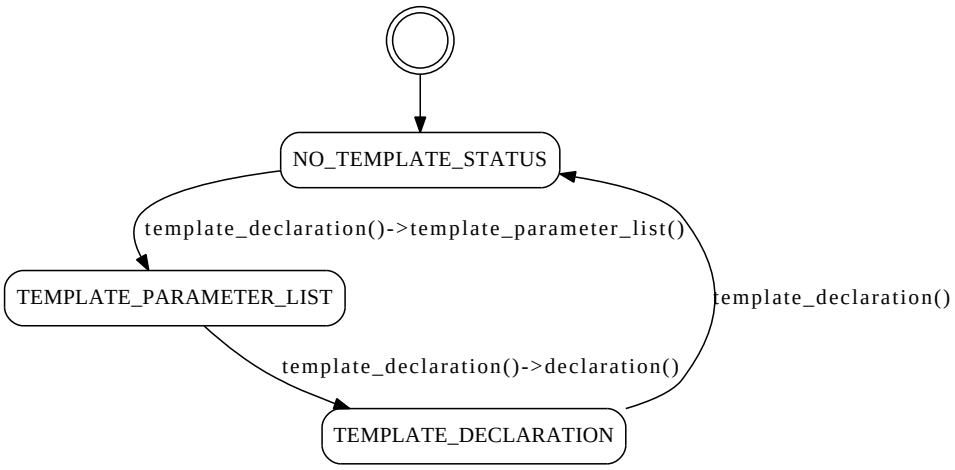

Figure 12.5: state machine

In the state diagram we can see the states and how the calls inside the rule **template_declaration** change this state, lets see this with this example:

```
#                       ❶
template < class T1>    ❷
void f(T1 t1)           ❸
{
}
```

❶ here abidos_cpp has the state **NO_TEMPLATE_STATUS**.

❷ while abidos_cpp are parsing inside < and > has **TEMPLATE_PARAMETER_LIST** state.

❸ while abidos_cpp are parsing the body of the template has **TEMPLATE_DECLARATION** state.

Lets see how abidos parses the previous example executing it with gdb

X file for gdb

```
# gdb src/abidos_cpp -x X
dir .
set print address off
b c_parser_descent::colon_colon_chain_process
run --includes test_includes/ \
    --out_dir test_out/ \
    --test_all_tokens_consumed_flag \
    --test_original \
    --ts_show \
    --verbose test/book_template_01.cpp \
      > test_out/out_book_template_01.cpp.txt
```

Run gdb like this:

gdb execution

```
abidos_cpp/processor$ gdb src/abidos_cpp -x X
```

This is the subtree to parse the first part:

```
template < class T1>
```

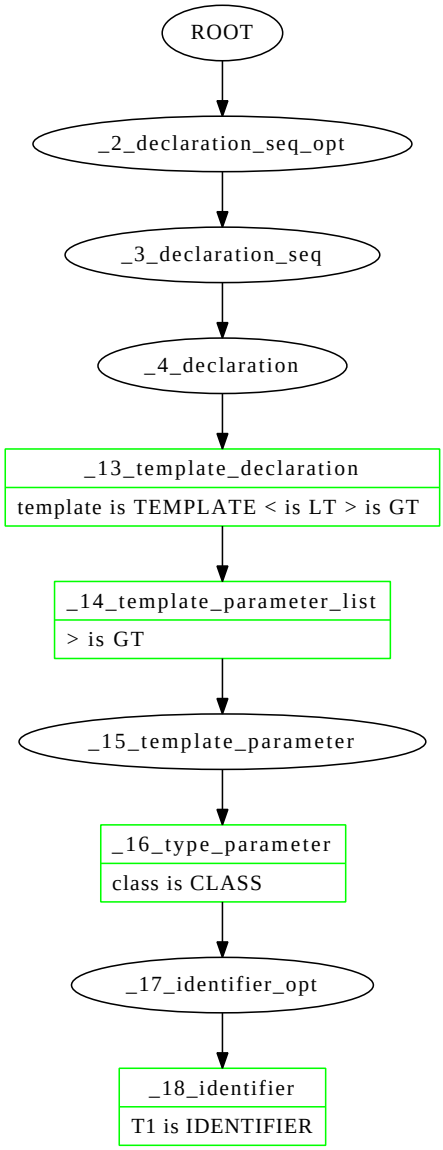

Figure 12.6: book_template template_parameter_list subtree

- template_declaration(): here the token **template** is matched and the **TEMPLATE_PARAMETER_LIST** state is set in **context.template_status** before the next rule is called.
- template_parameter_list(): here parameters are matched until > is reached this token is restores because is matched in **template_declaration()** too.
- template_parameter()
- type_parameter(): **context.template_parameter.vector_decl_specifier** is set with the actual meta-type.

```
int c_parser_descent::type_parameter(c_trace_node trace_node)
{
    trace_graph.add(trace_node, "type_parameter");
    c_context_tokens context_tokens(context);

    //## to do rest
    token_next(trace_node.get_tab());
    const int vector_id[]={CLASS, TYPENAME, INT, -1}; ❶
```

```
    if (token_is_one(  vector_id,trace_node) != 0) {
        c_decl_specifier decl(c_token_get());
        context.template_parameter.vector_decl_specifier.push_back(decl);

        context.declaring_template_type = 1;

        identifier_opt(trace_node);
        context.declaring_template_type = 0;
        return 1;
    }

    context = context_tokens.restore();
    return 0;
}
```

❶ In order to declare the parameters CLASS and TYPENAME has the same meaning.

- identifier_opt()
- identifier(): calls **semantic.identifier()**
- semantic.identifier():

```
void c_semantic::identifier(c_context & context, c_token token)
{
    ...
    if ( TEMPLATE_PARAMETER_LIST == context.template_status ) {
        if ( 1 == context.declaring_template_type ) {
            context.template_parameter.token = token; ❶

            context.vector_template_parameter.push_back(context.template_parameter); ❷
            context.map_template_parameter[token.text] = context.template_parameter;
            //parameter has been processed
            context.template_parameter.clear();

            return;
        }
    }
    ...
}
```

❶ here is processed **T1**.

❷ decls are stored in this case **class** is the type parameter.

Now abidos returns to **template_declaration()** and set the status:

```
  ...
  context.template_status = TEMPLATE_DECLARATION;
  ...
```

and **declaration()** is run:

_29_block_declaration is restored with backtracking we continue in _90_function_definition:

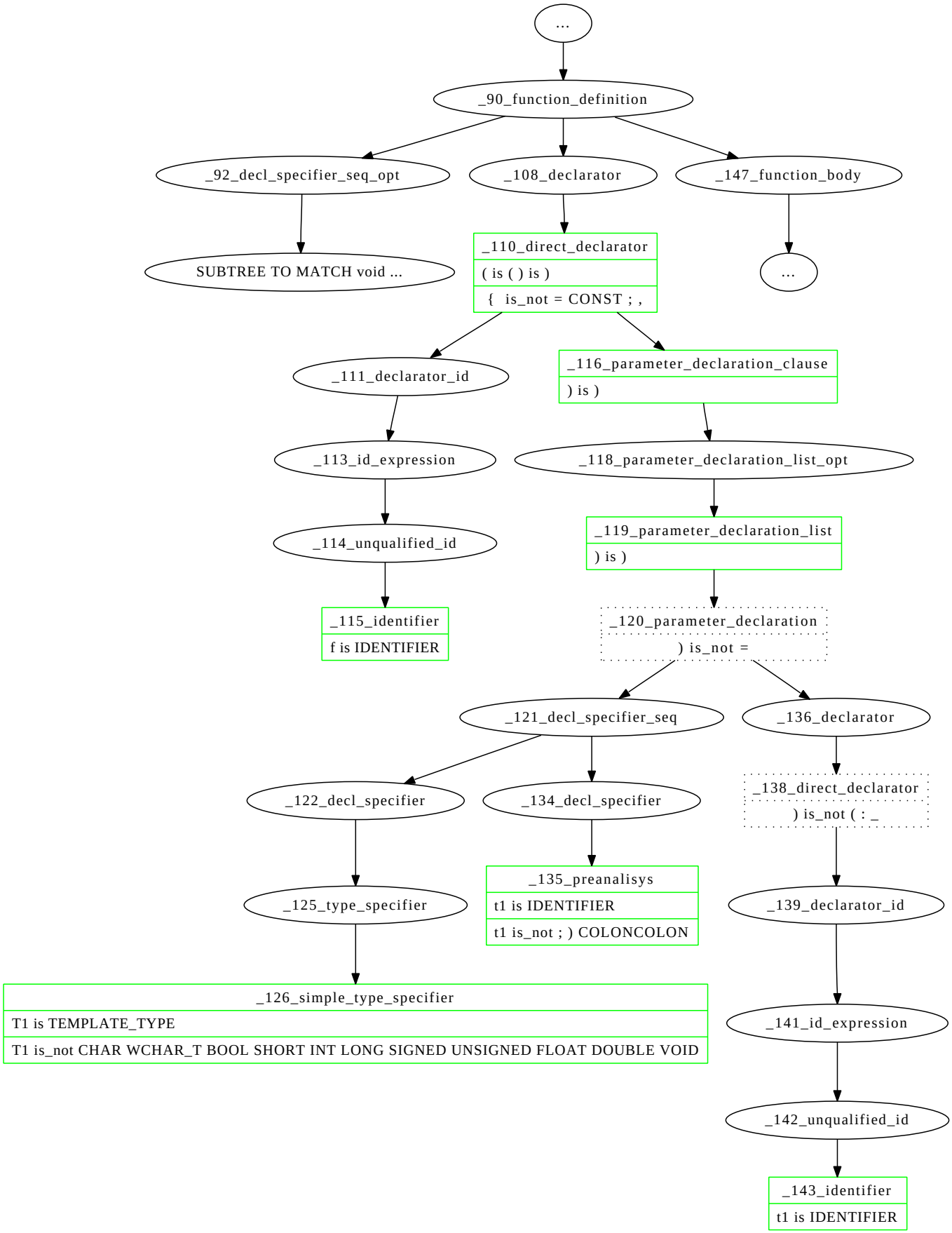

Figure 12.7: book_template function_definition subtree

we will see the rule **_126_simple_type_specifier**, the others rules of this subtree are pretty similar to other examples of this book.

- _126_simple_type_specifier(): here the next code runs:

```
int c_parser_descent::simple_type_specifier(c_trace_node trace_node)
{
    ...
    if ( TEMPLATE_DECLARATION == context.template_status ) {
        if ( token_is(TEMPLATE_TYPE, trace_node) ) {
            result = 1;
        }
    }
    ...
    if (1 == result) {
        c_decl_specifier decl(c_token_get());
        decl.type_specifier = 1;
        decl.has_colon_colon_after = has_colon_colon_after;

        if (1 == context.i_am_in_parameter_declaration) {
            context.param_vector_decl_specifier.push_back(decl); ❶
        } else if (1 == context.is_template_instantiation) {
            ...
        }
        ...
        return 1;
    }
}
```

❶ here abidos stores in context the decl **T1**

12.3 Templates instantiation

Now lets see another example of template, this time we will see a class template instead of a function template and how it is instantiated in another class.

```
template <class T1>
class A {
};

class B{
  A<int> a1; ❶
};
```

❶ here is the instantiation of template class **A**.

This is the subtree of **B** parsing:

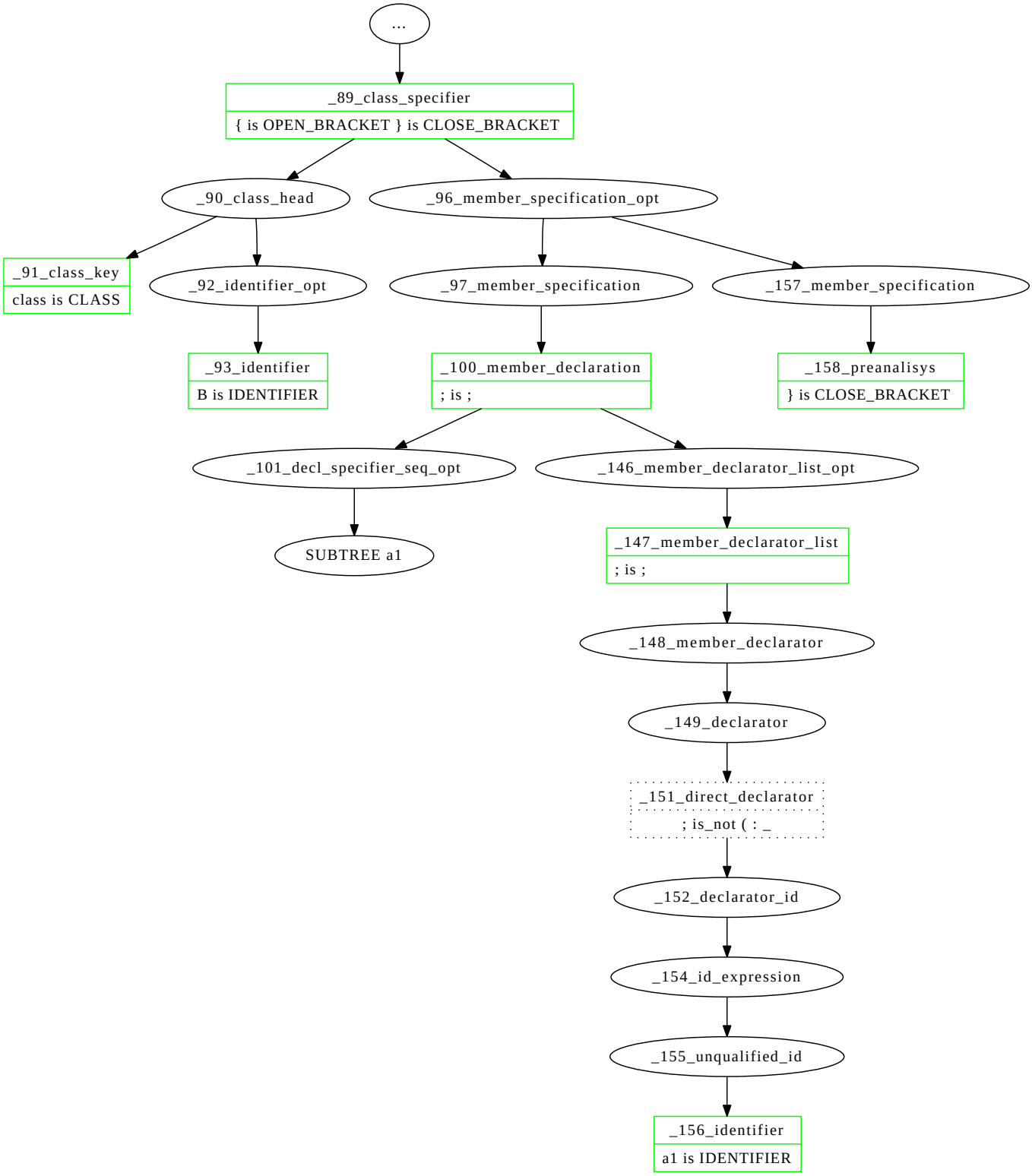

Figure 12.8: classes rules to parse B:

In this sub tree we will see the part when **a1** is parsed:

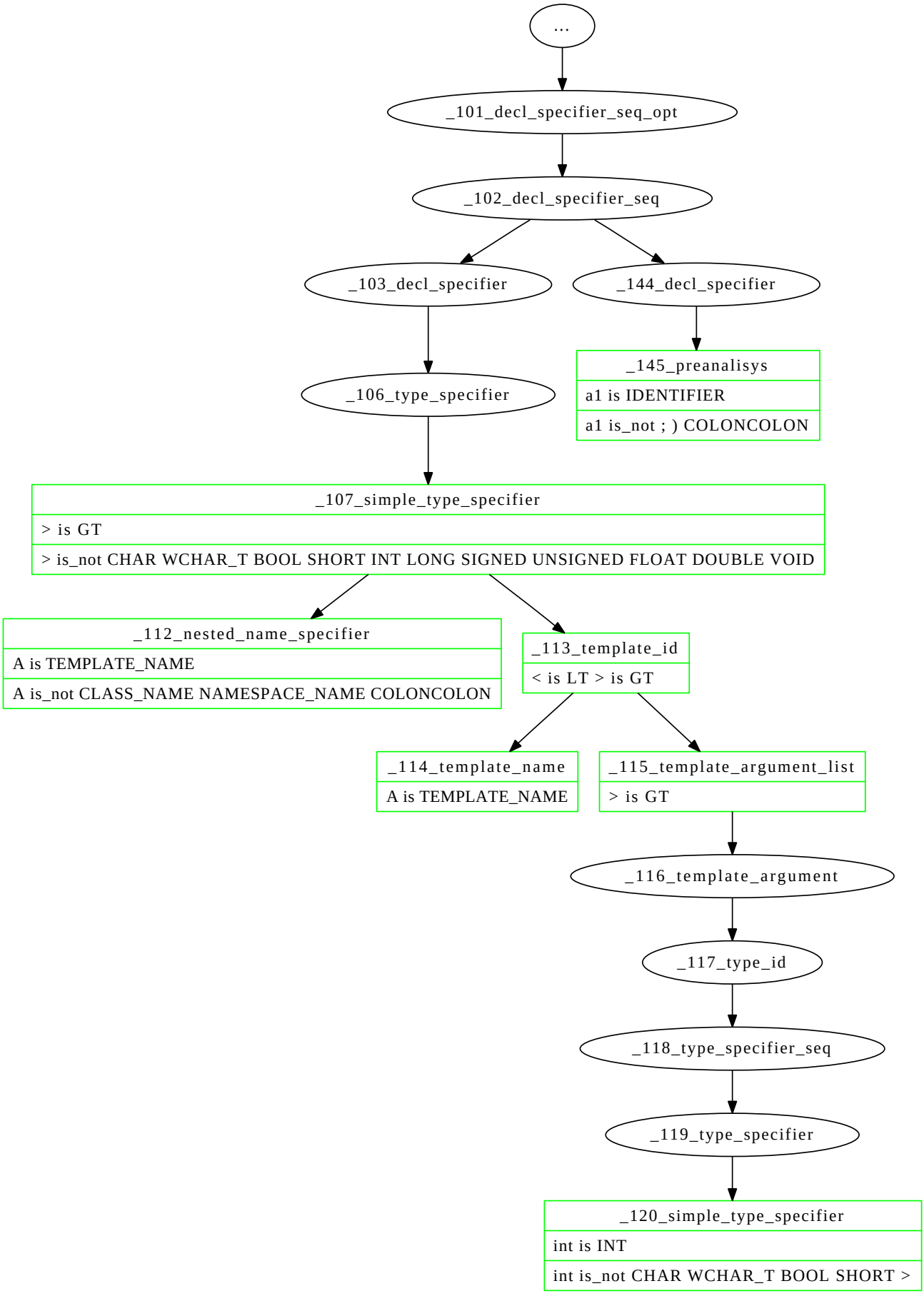

Figure 12.9: classes rules to parse B::a1:

Lets see the most meaningful rules of this subtree:

- simple_type_specifier():

```
int c_parser_descent::simple_type_specifier(c_trace_node trace_node)
{
    ...
    const int vector_id[]={
        CHAR, WCHAR_T, BOOL, SHORT, INT, LONG
        , SIGNED, UNSIGNED, FLOAT, DOUBLE, VOID, -1
    };

    if (token_is_one(vector_id,trace_node) != 0) {
        result = 1;
    }
    ...
    if (1 == result) {
        c_decl_specifier decl(c_token_get());
        decl.type_specifier = 1;
        decl.has_colon_colon_after = has_colon_colon_after;

        if (1 == context.i_am_in_parameter_declaration) {
            ...
        } else if (1 == context.is_template_instantiation) {
            //### to do this if is weird
            if ( token_is_not('>', trace_node) ) {
                semantic.template_instantiation_argument(context, decl);
            }
        } else {
            ...
        }

        return 1;
    }
}
```

- semantic.simple_type_specifier()

- template_instantiation_argument():

```
void
c_semantic::template_instantiation_argument(c_context & context, c_decl_specifier & decl) ❶
{
    ...
    c_template_argument argument;

    // yes we need the next parameter to fill vector_argument
    size_t i = context.vector_template_argument.size();

    if ( 0 < i ) {
        argument = context.vector_template_argument[i - 1]; ❷
        context.vector_template_argument.pop_back();

        argument.vector_decl_specifier.push_back(decl);
        context.vector_template_argument.push_back(argument); ❸
    }
}
```

❶ decl.token.text = **"int"**.

❷ argument.token.text = **"T1"**.

❸ now abidos knows **T1** is **"int"** type for this instantiation of the template.

- c_parser_descent::member_declaration() abidos returns to this method before **decl_specifier_seq_opt(trace_node);** line and calls **_146_member_declarator_list_opt()**

We are going to jump some rules and go to the most important function.

- 156_identifier(): token.text has the value **"a1"**
- c_semantic::class_member_declarator():

```
void
c_semantic::class_member_declarator(c_context & context, c_token token)
{
    ...
    c_symbol *p_symbol =
        ts.search_symbol(context.class_name_declaration);  // "B"
    ...
    int is_constructor = 0;
    int is_destructor = 0;
    int is_function = 0;

    ...

    c_class_member class_member(token,
                                vector_decl_specifier);

    class_member.is_constructor = is_constructor;
    class_member.is_destructor = is_destructor;
    class_member.is_function = is_function;

    ...

    if ( 0 == context.access_specifier) {
        context.class_key = p_symbol->class_key;
                switch (p_symbol->class_key) {
            ...
            context.access_specifier = PRIVATE;
            break;
            ...
          }
    ...


    if ( 1 == context.is_template_instantiation ) { ❶
        class_member.is_template_instantiation = context.is_template_instantiation;
        class_member.vector_template_argument = context.vector_template_argument;
        class_member.map_template_argument    = context.map_template_argument;
    }

    class_member.access_specifier = context.access_specifier;
    context.class_member = class_member;

    return;
}
```

❶ here is where abidos saves the information about this instantiation of the template.

- 151_direct_declarator(): after run [152-156] rules, direct_declarator calls semantic.declarator_insert();
- semantic.declarator_insert():

```
void c_semantic::declarator_insert(string tab, c_context & context)
{
    ...
    c_symbol *p_symbol = ts.search_symbol(context.class_name_declaration);
    if (p_symbol) {
        ...
        p_symbol->members.insert(context.class_member);
    }
}
```

Chapter 13

Index

_
_operator, 50

A
abidos_make_process.pl, 45
abstract class, 49
access_specifier
    PRIVATE
        PROTECTED, 49
aggregations, 49
alignment, 50
annotated parse tree, 32, 33, 35, 48
annotated parse trees, 7
architecture, 6
array, 49
astyle, 5
AUTO
    REGISTER, 50

B
Backtracking, 17
ban symbols
    baning symbols, 49
baning symbols, 49
base_clause, 49
Bison, 3
bison, 4
bison_execute.pl, 3, 4

C
C++, 1
c_context, 19
c_parser_descent::declaration(), 64
c_parser_descent::member_insert(), 64
c_parser_descent::simple_type_specifier(), 57, 59
c_semantic::check_coloncolon_member_function(), 62
c_semantic::class_member_declarator(), 58
c_semantic::declarator_insert(), 60, 64
c_semantic::identifier(), 58, 59
c_semantic::member_insert(), 60
c_semantic::member_param_declarator(), 59, 64
c_trace_graph, 31
c_trace_node, 31, 32
check_directories.pl, 3, 4
check_identifier(), 30
check_indent.sh, 3, 5
class_specifier, 49
CLASS_SPECIFIER_STATUS_IDENTIFIER, 54
CLASS_SPECIFIER_STATUS_MEMBER_DECLARATOR, 57
cmake, 2
code refactor, 1
compilation, 2
compile, 3
compositions, 49
constructor, 24
    nested_name_specifier, 49
context, 22, 35, 44
context decl, 59
context.i_token
    i_token, 27
context.restore()
    restore(), 27
context_good_way, 24
cv_qualifier, 49

D
decl multiples, 49
declarator, 49
declarator multiples, 49
default parameters values, 49, 50
define, 50
descent parser, 17
design, 6
destructor, 24
    unqualified_id, 49
dot viewer, 7
dotted, 32
drop_head_namespace(), 29
DRY, 2

E
ELLIPSIS, 50
enum
    enum_specifier
        enum_name, 49

enum_name, 49
enum_specifier
    enum_name, 49
extern, 49
extern_c, 49
extract_symbols.pl, 3

F
Flex, 3
flex, 11
flex_execute.pl, 3
free function
    function free, 50
FRIEND, 49
function free, 50
function_body, 50
function_definition, 49
function_specifier
    INLINE
        VIRTUAL, 50

G
gdb, 20
generate_begin_graph.pl, 3, 4
generators, 7
get_full_name(), 61, 64
green, 33

I
i_token, 27
ifndef
    define, 50
include, 50
indent_run.sh, 3, 5
inheritance multiple, 49
inheritance simple, 49
INLINE
    VIRTUAL, 50
Inner classes, 51

J
java, 17

L
lexer, 11
lexical, 7
link, 3
loader, 6, 8, 10

M
make, 2
Makefile, 2
mangling, 12, 50
member_declaration, 49
member_declaration(), 57
multiple inheritance
    inheritance multiple, 49

N
namespace, 50
    using_directive, 50
nested_name_specifier, 49, 50

O
operator_function_id
    _operator, 50
overloaded functions, 15
overloading functions, 50
overloading operators
    operator_function_id
        _operator, 50

P
parameter_declaration, 50
Parser, 1
pointer, 50
pre-declaration, 50
preanalisys_has_one, 23
preprocessor
    ifndef
        define, 50
    include, 50
PRIVATE
    PROTECTED, 49
PROTECTED, 49
prune, 23, 24

Q
qualified_id, 49

R
Refactor
    code refactor, 1
REGISTER, 50
restore(), 27

S
semantic, 7
semantic decl, 57
simple inheritance
    inheritance simple, 49
storage_class_specifier
    AUTO
        REGISTER, 50
symbol.class_key, 54
symbol.type, 54
symbols, 7
syntactic, 7

T
template, 50, 65
template declaration, 65
template instantiation, 70
template parameter list, 66
template_declaration(), 68
template_instantiation_argument(), 73

test, 45
tests_run.pl, 3, 5, 45
token_get(), 29
token_next(), 29
tokens_vector, 24, 27
trace_graph, 31, 47
type_specifier, 49
TYPEDEF
    typedef, 49
typedef, 49
typename, 50, 68

U
unqualified_id, 49
using_directive, 50

V
vector_template_argument, 73
VIRTUAL, 50

X
X file, 20

Y
yacc, 17
yylex, 24
yytokens, 4
yytokens_short, 4