

CPP Denotational Semantics

Jean-Marie Favre

*Adele Team, Laboratoire LSR-IMAG
University of Grenoble, France
<http://www-adele.imag.fr/~jmfavre>*

Abstract

This paper shows that CPP, the preprocessor of the C language, can be seen as a programming language in which directives are statements, parametrized macros are functions, files are procedures, directories are modules, and command lines are programs. The semantics of CPP can therefore be described using traditional techniques. This paper describes the semantics of CPP in a denotational style. By contrast with previous work, the full semantics is taken into account including non trivial aspects such as recursive macros, stringification and concatenation.

1. Introduction

CPP is C PreProcessor. This tool is pervasive in Unix environments. For instance, all system libraries are based on an extensive use of this tool. C programs are processed by CPP before compilation takes place as shown in Figure 1.

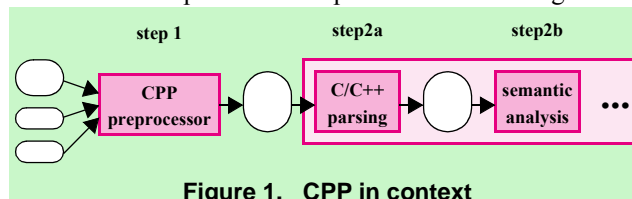


Figure 1. CPP in context

CPP is usually associated with the C and C++ language as shown in step 2, but this is not always the case. CPP can also be used independently. It has also been reported to be used with many different kinds of files: LaTeX documents, makefiles, X resources files, html files, java files, and even formal specification descriptions.

The success of CPP is indeed due to its apparent simplicity. Conditional compilation directives (`#if`) allow to include or remove some fragments of text depending on the value of some macros. These values can be set on the CPP invocation command line with `-D` options. Some pieces of text can be copied from a given file thanks to `#include` directives. The actual files to be selected are controlled at the level of the command line by `-I` options. These options set the “include path”, that is the sequence of directories in which included files have to be searched.

1.1. CPP analysis

Many authors discussed the practical significance of CPP to address portability and configuration management issues (e.g. [1][4][5][6][7][8][9]). CPP can be considered as a legacy tool for programming-in-the-large [16].

Though CPP is a very flexible tool, its extensive use leads to many issues. Understanding code with many CPP directives is sometimes a real challenge. Many tools have been proposed to analyse CPP files [10][11][12][13][14][18][20][21]. However these tools usually handle only a subset of CPP features. For instance [7] and [12] focus on conditional compilation and ignore complex macro definitions, [10] focuses on file inclusion, [13] concentrates on macro expansion, and [20] provides an approximate method to simplify conditional compilation.

As far as we know, none of these tools can ensure that they are fully compatible with the full semantics of CPP. There are two good reasons for that: (1) there is currently no description of the semantics of CPP, and (2) most of the time advanced features are simply ignored¹.

Our experience in different industrial settings shows that this is an important limitation in practice because analysing very large amount of code with a limited amount of confidence is not necessarily useful [1].

1.2. Rationale for our approach

The goal of this paper is to provide a framework on which reliable CPP analysis tools can be based. It is shown how the semantics of CPP can be described in a denotational style. Our objective is pragmatic and describing CPP semantics is not an academic exercise. For instance our objective is neither to determine if CPP is Turing complete, nor to define the exact CPP computational limits. CPP is definitively an ugly tool of absolutely no interest from a theoretical point of view. Its design is very poor and it is very unlikely that people who care about formal language descriptions and classifications take CPP into consideration.

1. Advanced features includes for instance stringification” (`#`), concatenation (`##`), recursive macros (i.e. `#define foo (4+foo)`), or the use of macros in `#include` directives.

On the contrary, *our goal is to provide a very precise and accurate specification of the CPP semantics, and this in order to get very practical results.* For instance a very important rationale behind our approach is to keep the specification of the semantics *executable*. Denotational semantics is very interesting with this respect because of its close relationship with functional implementations. Executability is of paramount importance to check the validity of the specification with actual CPP implementations. It is also very important in our context since *the ultimate goal is to derive executable CPP analysis tools by applying transformations on the semantic equations.* In fact, this approach was successfully applied in [1]. It was shown for instance how control flow analysis, data flow analysis, slicing, and program specialisation could be applied in the context of CPP by adapting techniques usually restricted to programming languages. All features of CPP were not supported however [1].

This paper goes one step further, by considering all CPP features. Due to limitation space this paper exclusively focuses on the foundation of this approach, that is the description of CPP semantics.

Finally before to enter to the core of the subject, emphasis should be put on the fact, that just like most other tools, the scope of this paper is limited to step 1 of the process described above. The results presented in this paper are therefore not restricted to C or C++.

The rest of this paper is structured as following. Section 2 explains the shortcomings of existing CPP informal descriptions. Section 3 introduces APP, an abstract language semantically equivalent to CPP. Section 4 defines the semantics of APP and therefore the semantics of CPP. A discussion is provided in Section 5. Finally, Section 6 concludes the paper.

2. Existing descriptions of CPP

Different descriptions of a given language can be given to suit the needs of different target audiences:

- *Programmers* need to understand the meaning of each constructs but also how to use these constructs.
- *Language implementers* need precise specifications of the language, as well as suitable test cases to check whether the implementation built is conform or not.
- *Analysis tool builders* need a description of the language that helps them reason about the language constructs and to derive new analysis techniques. Precise knowledge about the semantics is a prerequisite for safe transformations of source code.

As shown below there exist CPP descriptions both for programmers and language implementers but there is no suitable description of CPP for its manipulation.

2.1. CPP descriptions for programmers

Programmers can find descriptions of CPP in many books. However, most of these descriptions are inaccurate and incomplete. By contrast, in [15], Stallman gives a very good description of CCCP, the GNU implementation of the CPP. CPP is not as simple as most programmers believe: this document is 53 pages long. Complex features are explained through examples, because the behaviour of these features is quite difficult to grasp. This document also give hints on typical usage.

2.2. CPP descriptions for language implementers

Implementers of CPP require precise specifications with specific emphasis on standardization. In an attempt to give such precise description, the specification of the C language is given in [28] as a normative reference. This document includes a 17 page long specification of CPP. Interestingly, this specification is decomposed in various parts including a description of the syntax of CPP. For almost each CPP directives there is a section decomposed in 3 subsections: constraints, semantics and examples. While this structure seems quite good at the first sight, a closer look reveals serious issues. For instance, the description of the syntax is far from accurate. It does not reflect the actual complexity of CPP. Many tricks are later described in almost every section. The problem is in fact due to the very poor design of CPP which makes no separation between the lexical, syntactic and semantic facets of the language. As a result, the description of a single feature is often spread over many sections that handle “special cases”. Semantics is described through quite obscure concepts such as the concept of *placemaker* introduced in the left part of Figure 2 on the next page. The reader is encouraged to read carefully this description and observe how the concept of placemaker is described.

2.3. CPP descriptions for tool builders

Our work on reverse engineering configuration management artefacts [1][16] led us to search for a description of CPP that was precise enough to support the analysis and manipulation of CPP sets of files. Conversely to the approach presented in [12] which is restricted to a subset of CPP that makes it useless in practice, our goal was to deal with actual CPP code to handle very large industrial software products. We rapidly found that Stallman’s description of CCCP [15] was not precise enough. The ISO description didn’t revealed to be more helpful either. The interleaving between the number of special cases spread in the textual description makes this description too complex to reason about. It is impossible to give a confident

6.10.3.3 The ## operator

Constraints

A ## preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

Semantics

If, in the replacement list of a function-like macro, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence; however, if an argument consists of no preprocessing tokens, the parameter is replaced by a *placemaker* preprocessing token instead.

For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. Placemaker preprocessing tokens are handled specially: concatenation of two placemarkers results in a single placemaker preprocessing token, and concatenation of a placemaker with a non-placemaker preprocessing token results in the non-placemaker preprocessing token. If the result is not a valid preprocessing token, the behavior is undefined. The resulting token is available for further macro replacement. The order of evaluation of ## operators is unspecified.

EXAMPLE 3 To illustrate the rules for redefinition and reexamination, the sequence

```
#define x      3
#define f(a)   f(x * (a))
#undef x
#define x      2
#define g      f
#define z      z[0]
#define h      g(~
#define m(a)   a(w)
#define w      0,1
#define t(a)   a
#define p()    int
#define q(x)   x
#define r(x,y) x ## y
#define str(x) # x

f(y+1) + f(f(z)) % t(t(g)(0) + t)(1);
g(x+(3,4)-w) | h 5) & m
  (f)^m(m);
p() i[q()] = { q(1), r(2,3), r(4,), r(,5), r(,) };
char c[2][6] = { str(hello), str() };

results in

f(2 * (y+1)) + f(2 * (f(2 * (z[0]))) % f(2 * (0)) + t(1);
f(2 * (2+(3,4)-0,1)) | f(2 * (~ 5)) & f(2 * (0,1))^m(0,1);
int i[] = { 1, 23, 4, 5, };
char c[2][6] = { "hello", "" };
```

Figure 2. Excerpts of the the C language specification (see [28] page 140 and 142)

assessment of a property about CPP, unless if explicitly stated in the specification. Building a static analysis tool in such a context is almost impossible.

2.3.1. Finding an accurate description of CPP. We therefore decided to extract high level rules from the available textual descriptions and to come up with a set of mathematical equations explaining the CPP behaviour. During this process we didn't find precise and definitive answers to all our questions. Actually, we found that the more accurate specification of CPP was indeed its implementation itself. In other words, the best way to determine what CPP will produce for a given set of files, is to execute an implementation of it, which is after all, an executable specification. In some very specific situations, the outputs produced by different implementations slightly differ. In this paper CCCP [15] is taken as the reference implementation. The choice of the GNU implementation is due to the following facts: (1) as pointed out before the documentation is quite good, (2) the GNU implementation is widely used and is available on a wide range of platforms, and the last but not the least (3) the source code of CCCP is freely available. This last point was very important in our work because we often found necessary to have to look at the source code to fully understand the behaviour of strange constructs. This was really hard work because the implementation is highly-optimized C code. Moreover, the main file is more than 8000 lines long! The full understanding of CPP was achieved by reverse engineering manually some parts of the implementation. The rules that were missing to complete our specification were extracted from source code.

2.3.2. Finding a suitable body of knowledge. A set of mathematical rules is not necessarily easier to understand. It

nevertheless presents two benefits: (1) it is much more precise than textual descriptions, and (2) it is far more concise than source code. The main advantage of mathematical abstractions is that they can be transformed mechanically without a real understanding of their whole meaning. What is missing is a suitable body of knowledge on which to rely on. All CPP analysis tools mentioned in this paper describe this preprocessor as a very specific tool. The techniques applied are often totally disconnected from the huge amount of research work done over the last decades on static and dynamic analysis. Almost each year a new technique is published to handle a subset of CPP without any consistent framework to refer to. During our work on formalizing the CPP behaviour we found that consider CPP just like a programming language was a very powerful approach, because it makes it possible to reuse a very large amount of knowledge in program manipulation. The benefit of this approach was presented in [1] and [3], but the full semantics of CPP was not supported. In the rest of the paper this limitation is removed.

3. From CPP to APP

To cope with the difficulty in reasoning about CPP, this section describes APP, the *Abstract PreProcessor*. APP is an abstract language semantically equivalent to CPP. CPP is just one possible concrete syntax for APP¹. A very strong emphasis should be put on the fact that *APP is not yet another preprocessor. APP entities are just abstractions for CPP entities*. They are produced automatically through parsing.

1. In fact, APP is slightly more general than CPP because it is more regular. As discussed in Section 5 this is not an issue in practice.

CPP concept	APP concept	Z Constructor	Z Type Definition
Token	Factor		$tFactor$ (4)
Identifier	Variable	Φ (4)	$tFactor$ (4)
Non-identifier token	Constant	K (4)	$tFactor$ (4)
defined() operator	Operator	Ω (4)	$tFactor$ (4)
Regular macro	Expression	$Expr$ (18)	$tExpr$ (5)
Parametrized macro	Function	Fun (18)	$tFun$ (12)
Parametrized macro body	Function body expression		$tFunBodyExpr$ (11)
Macro formal parameter	Function formal parameter	Λ	$tFunBodyFactor$ (6)
Macro parameter	Value of parameter	Θ	$tFunBodyFactor$ (6)
Stringification (#)	Operator	$\$$	$tFunBodyFactor$ (6)
Concatenation (##)	Operator	Ξ	$tFunBodyFactor$ (6)
Directive	Statement		$tStmt$ (13)
#define #undef	Assignment	Set (13)	$tStmt$ (13)
#if #ifdef #ifndef	Conditional stmt	If (16)	$tStmt$ (13)
Piece of text	Output stmt	Out (14)	$tStmt$ (13)
#include	Procedure call	$Call$ (17)	$tStmt$ (13)
File	Procedure	$Proc$ (19)	$tProc$ (13)
#include argument	Procedure Expression		$tProcExpr$ (21)
Directory	Module	$UMod$ (27)	$tMod$ (29)
File or Directory	Unit		$tUnit$ (26)
Path name	Unit expression		$tUnitExpr$ (25)
Include path (-I)	Module imports		$tImports$ (26)
Command line	Program	$Prog$ (32)	$tProg$ (32)
Builtin Definitions	Builtin env.		$tBuiltinEnv$ (37)

Figure 3. Mapping CPP - APP

```

1 [ tChar ]
2 tString      == seq Char
3 tVarName    ::= Var « tString »
4 tFactor     ::= K « tString » |  $\Phi$  « tVarName » |  $\Omega$  « tVarName »
5 tExpr       == seq tFactor
6 tFunBodyFactor ::=  $\Gamma$  « tFactor »
7             |  $\Lambda$  «  $\mathbb{N}_1$  »
8             |  $\Theta$  «  $\mathbb{N}_1$  »
9             |  $\$$  « tFunBodyFactor »
10            |  $\Xi$  « tFunBodyFactor  $\times$  tFunBodyFactor »
11 tFunBodyExpr == seq tFunBodyFactor
12 tFun        == (iseq tVarName)  $\times$  tFunBodyExpr
13 tStmt      ::= Set « tVarName  $\times$  tValue »
14            | Out « tExpr »
15            | Seq « seq tStmt »
16            | If « tExpr  $\times$  tStmt  $\times$  tStmt »
17            | Call « tProcExpr »
18 tValue     ::= Nil | Expr « tExpr » | Fun « tFun »
19 tProc      ::= Proc « tStmt »
20 tProcKind  ::= Builtin | App
21 tProcExpr  ::= ProcLiteral « tProcKind  $\times$  tUnitExpr »
22            | ProcExpr « tExpr »
23 tUnitName   := UnitName « tString »
24 tUnitFactor ::= Root | Up | Down « tUnitName »
25 tUnitExpr  == seq tUnitFactor
26 tUnit      ::= UProc « tProc »
27            | UMod « tUnitName  $\rightarrow$  Unit »
28            | UAlias « tUnitExpr »
29 tMod       == ran UMod
30 tModExpr   == tUnitExpr
31 tImports   == seq tModExpr
32 tProg      ::= Prog « tImports  $\times$  tStmt »
33 [ tFile, tDir; tCmdLine ]
34 ParseFile : tFile  $\rightarrow$  tProc
35 ParseDir  : tDir  $\rightarrow$  tMod
36 ParseCmdLine : tCmdLine  $\rightarrow$  tProg
37 tBuiltinEnv ::= tImports  $\times$  tStmt

```

Figure 4. APP Abstract Syntax (syntactic domains)

```

#define x b+1
#undef u
#define u(x,y) x*2*a+#y
"foo" 3 x
#if A<B
#define R 2
#endif
#ifdef A
#endif
#include "../a/b"
#include <stdio.h>
#include A

```



```

Seq « Set «  $\bar{x}$ , Expr «  $\Phi \bar{b}$ , +, 1 ),
      Set «  $\bar{u}$ , Nil ),
      Set «  $\bar{u}$ , Fun « F «  $\bar{x}$ ,  $\bar{y}$ ,  $\langle \Theta I, \Gamma^*, \Gamma_2^*, \Gamma(\Phi a), \Gamma_+, \$(\Lambda_2) \rangle \rangle$  ),
      Out « "foo", 3,  $\Phi \bar{x}$  ),
      If «  $\langle \Phi \bar{a}, <, \Phi \bar{b} \rangle$ ,
          Set «  $\bar{R}$ , Expr « 2 ),
          Seq «  $\langle \rangle$  ),
      If «  $\langle \Omega \bar{a} \rangle$ ,
          Seq «  $\langle \rangle$ , Seq «  $\langle \rangle$  ),
      Call « ProcLiteral « App, « Up, Down  $\bar{a}$ , Down  $\bar{b}$  » ),
      Call « ProcLiteral « Builtin, « stdio.h » ),
      Call « ProcExpr «  $\bar{A}$  » )

```

To improve the reading of APP expressions different shortcuts are used without loss of precision:

- "A" denotes a $tString$
- \bar{A} is a shortcut for Var "A"
- + is a shortcut for K "+"
- stdio.h is a shortcut for $UnitName$ "+"

Figure 5. Example of mapping : Expressions, Functions and Statements

3.1. Conceptual mapping

Figure 3 shows the correspondence between CPP concepts and APP concepts. The third and fourth columns refer to Z expressions [25] defined in Figure 4. Numbers refer to line numbers¹. As shown in Figure 3 pieces of text are mapped to expressions; directives are mapped to statements, parametrized macros are mapped to functions, files are mapped to procedures, directories are mapped to modules and command-lines are mapped to programs. This strange mapping requires an explanation.

If one admits that a macro name corresponds to a variable, a `#define` directive becomes an assignment. The occurrence of a macro in a piece of text becomes an occurrence of the corresponding variable. Since this piece of text will be appended to the program output after the evaluation of all variables, it just behaves as if an implicit output statement were inserted in front of it. Parametrized macro definitions cannot produce side effects: they can neither contain assignments, nor output statements. They are therefore pure functions without side effects.

This contrasts with include files which contain arbitrary sequences of CPP directives. These directives are executed in order when the file is included in another one. The inclusion of CPP files can both changes the values associated to variables (macros) and append text to the output stream. CPP files are therefore procedures with side effects (but no parameters). The inclusion of a file is therefore equivalent to a procedure call.

Directories being namespaces for files, they become modules, that is namespaces for procedures or nested modules. Include paths defined via the `-I` option become imports: they specify the namespaces in which procedures are searched. Finally an invocation of CPP on a command line is just like the “main” piece of code: it imports some modules (`-I` option), defines some variables (`-D` options) and calls a given procedure (the file to be processed).

After this explanation, the analogy might still appear suspicious. Indeed, it would not be really useful if it were not based on a rigorous definition. In the following section the abstract syntax of APP is described. Due to a lack of space, the correspondence between CPP is sketched only through two small examples in Figure 5 and Figure 6. The reader is invited to refer to these figures and to Figure 3 to check for the correspondence between APP and CPP. The remainder of the paper uses mostly APP concepts. These concepts are just abstractions for CPP constructs.

1. Indeed, the Z language does not allow recursive type definitions. This theoretical issue is discussed in Section 5, but this has no practical consequence for the purpose of this paper..

3.2. APP language definition

APP is a small imperative language with no loops. It deals with strings and has special-purpose operators such as concatenation. The description below refers to APP abstract syntax given in Figure 4.

3.2.1. Strings. *Strings* (2), that is sequence of characters are the basic elements of APP. The actual set of *characters* (1) is of no interest here. It is modelled as a “given set” [25].

3.2.2. Expressions. APP, just like other programming languages is based on *expressions* (5). Expressions are made of factors. A *factor* (4) is either a constant (**K**), a variable occurrence (**Φ**) or an unary test operator (**Ω**). This last operator models the `defined()` operator in CPP.

3.2.3. Functions. Anonymous functions (i.e. lambda expressions) can be described using APP. A *function* (12) is described by the sequence of formal parameter names and a body expression. A *body expression* (11) is similar to regular expression as described above, excepted that in addition to regular factors (**Γ** (6)), the body of a function can also include:² (1) indexed references to formal operators (**Λ** (7) and **Θ** (8)), (2) stringification unary operators (**\$** (9), **#** in CPP), (3) token concatenation binary operators (**Ξ** (10), **##** in CPP).

3.2.4. Statements. CPP directives are represented as *statements* (13). There are 5 kinds of statements: assignments (*Set*), output statements (*Out*), sequence of statements (*Seq*), conditional statements (*If*), and procedure calls (*Call*). Sequences represent blocks of consecutive statements³.

3.2.5. Assignments. The purpose of an *assignment* (13) is to change the value of a given variable. They are three types of *values* (18): expressions (*Expr*), functions (*Fun*) and *Nil*. The *Nil* value means that the variable has no value. Note that no evaluation takes place during the assignment. Note also that anonymous functions are also treated as values, just as in functional programming languages.

3.2.6. Output statements. The purpose of an *output statement* (14) is to write the result of the evaluation of an expression at the end of the output stream.

3.2.7. Conditional statements. Depending on the evaluation of the condition, a conditional statement will execute either the first or second statement.

3.2.8. Procedures. *Procedures* (19) do not have parameters, just a body, which is a statement (and in particular a block (18) of statements).

2. In fact, this definition is more regular and general than what is allowed by CPP. For instance CPP put as additional constraint $dom \$ = ran \Lambda$ indicating that the argument of a stringifier operator must be a formal parameter. APP subsumes CPP. This is what is important in practice.

3. *Seq* should not be confused with *seq* which is the predefined type constructor for sequence in the Z notation.

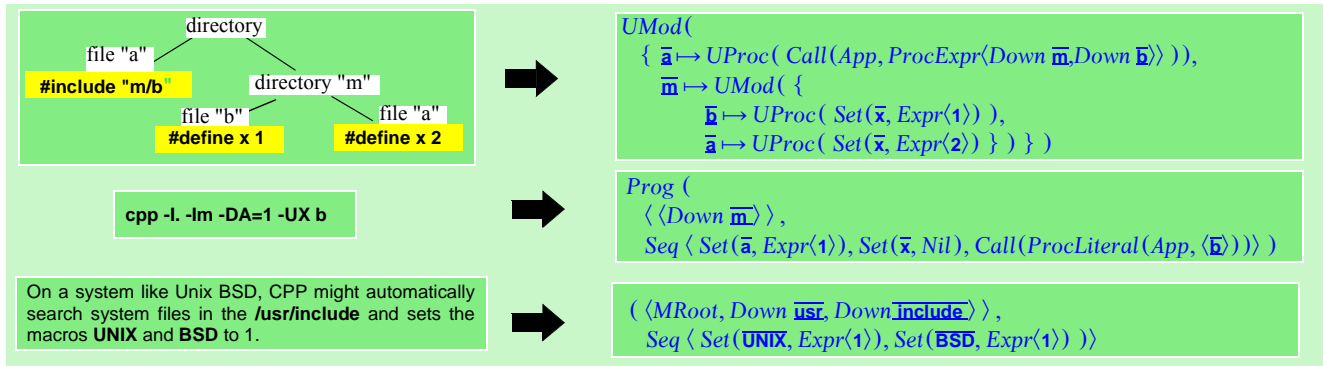


Figure 6. Example of mapping : Procedures, Modules, Programs and Builtin Environments

3.2.9. Procedure calls and procedure expressions. Procedure calls (17) do not directly refer to a given procedure: the procedure to be called is computed by evaluating a *procedure expression* (21). A procedure expression is either a literal (21) referring to a given module or procedure, or an expression (22)(5) which evaluation must return a procedure name. User-defined and builtin procedures are distinguished (20)(22) since they are handled in a very slightly different way. The same occurs in other languages as well: for instance in Java the java.lang package is automatically imported.

3.2.10. Modules. Just like in modular programming languages like Modula2, procedures are named and stored in *modules* (29). Modules can contain nested modules. In the remainder of this paper the term *unit* (26) is used as a generic term to name either procedures (*UProc* (26)), modules (*UMod* (27)) or aliases to another units (*UAliases* (28))¹. To navigate within such tree structures, the notion of qualified named found in many programming languages is here called a *unit expression* (25). A Unit expression is an absolute or relative path used to refer to a particular unit. By contrast with modular languages, module nesting does not offer encapsulation.

3.2.11. Imports. As in many other languages, *imports* (31) can be used to resolve unqualified names: relative procedure expressions are evaluated using imports, which are represented as an ordered sequence of *module expressions* (30). The ordering is used to resolve overriding.

3.2.12. Programs. By contrast with modular languages, where imports are "local" to each individual module, APP imports are associated with *programs* (32), that is, pieces of code that are used to start executions. The body of a program is simply a statement.

3.2.13. Builtin Environment. Just like some other languages, each implementation of APP on a particular

1. An alias is just like in Ada, a reference to another unit. This corresponds to an entity which is imported and re-exported with a new name. This feature models Unix symbolic links.

operating systems defines a *builtin environment* (37). Such an environment may include implicit imports and variable initialization.

3.3. Global view on APP abstract syntax

The relationships between syntactic domains is illustrated in Figure 7. In this dependency graph, nodes are types; edges indicate on which types a given type is based. As it can be seen statements, factors, body factors, and units are recursive domains. The graph also reveals the existence of 4 main clusters: statements, functions, values and procedures/modules. This graph provides useful insights into the CPP language, because it shows that the actual complexity of the language is not linked to the directives (Statements), but to the many others details. It also shows that all clusters are connected, meaning that tools that attempt to consider only one subset of the language can't provide accurate results. For instance there is a dependency between a procedure expression and a variable name. In other words file inclusion cannot be disconnected from macro evaluation.

3.4. From CPP to APP in practice

Giving a formal definition of CPP is not useful per se: practitioners usually do not like formal language like Z; theoreticians are not either interested by horrible tools like CPP. Moreover, nobody would like to write APP modules and programs. Anyway, this is not the goal of APP. APP artefacts are automatically produced by parsing existing CPP artefacts. Two prototypes were derived from the Z specification described in this paper: one in OCaml, a functional language, the other in Java.

3.4.1. CPP Parsing. At the level of specification, the parsing process is described by partial functions (\rightarrow) since some CPP artefacts may lead to parsing errors. Parsing directories (35) and command lines (36) is trivial in practice. Implementing the file parser (34) is much more complicated.

It must take into account all lexical details associated with the concrete syntax of CPP. This language contains many irregularities and building a parser for it is a very tedious task. Describing this parsing process is far beyond the scope of this paper. A parser was implemented using the JavaCC parser generator. JavaCC contextual tokens were used extensively to handle CPP tricks. This parser is 1521 lines long. A simplified version of it is described in [2].

3.4.2. AST representation in OCaml. The main benefit of the APP abstract syntax is that it is quite clean and quite regular. This enables to represent CPP artefacts by means of simple data structures directly derived from the Z specification given above. It was straightforward to derive the corresponding OCaml type definitions, since this functional language directly implements most of the notions found in Z. Similarly converting Z values into OCaml is also straightforward. For instance the program given in Figure 6, is represented by the following term in OCaml. This term is automatically generated by a parser.

```
Prog (
  [Down "m"],
  Seq[Set (VarName "a"); Expr[K "1"];
    Set (VarName "x");
    Call (ProcLiteral (App, [UnitName"x"])] ) )
```

3.4.3. AST representation in Java. A set of Java classes were also derived as an alternative implementation. This was also quite simple: each type and constructor was implemented as a class; the relationships between a type and its alternatives were represented as inheritance relationships. For instance `Statement` was transformed into an abstract class with 5 subclasses, namely `Set`, `Out`, `Seq`, `If` and `Call`. The result of this process was optimized to obtain a suitable data structure [2]. This resulted in about 30 Java classes.

4. CPP/APP Semantics

This section gives a precise definition of APP semantics (and thus of CPP semantics) using a denotational style. Figure 9 defines semantic domains. Figure 10 presents the type of semantic functions. Figure 12 The semantics of statements, procedures, modules and programs is given in Figure 12. These definitions are shortly commented in the next sections.

4.1. Semantic domains

The first step in denotational semantics is to define the semantic domains. These types are defined in Figure 9. The main artefact produced by APP is its *output* (1) which is a sequence of tokens. Assignment statements change the value stored in the *memory* (2), which is a mapping between variable names and values. Notice that the memory store

syntactical values (tValue). This contrasts with traditional languages in which memory store the result of the evaluation of expressions, for instance integers.

Different types are also introduced to model the evaluation of conditional statements (3)(4)(5)(6)(7). Module semantics also require specific domains. A *unit identifier* (8) identifies each unit in a non ambiguous way, since it represent its absolute path in a given hierarchy. This contrasts with unit expressions that allow to “navigate” in the tree structure. A module denotes a *unit environment* (10). In other words the meaning of a module is a mapping between a unit identifier and a unit. A *procedure environment* contains all information used to search a procedure from a procedure expression. It is therefore formed by a unit environment (where to search), a set of imports (in which places) and the a unit module identifier (where to start, that is the current module).

4.2. Semantic functions

Semantic functions are the core of denotational semantics. These functions map syntactic domains to semantic domains.

4.2.1. Signatures of semantic functions. The denotational semantics of CPP is based on the semantic functions described in Figure 10. There is one semantic function for each type, plus some additional functions used to define other functions. Due to the lack of space, Figure 12 give the semantics of four constructions only: statements, procedures, modules and programs.

4.2.2. Statement semantics. Given a procedure environment and a memory, a statement produces an output and a memory (12). In fact such information is very interesting since it makes explicit for instance that the procedure environment does not change during the execution; that the output produced has absolutely no impact on the execution. The semantics of each statements is given below. When a *assignment* `Set(x,v)` is executed (44), no output is produced (∅). The memory *r* is changed by replacing (⊕) the previous value of *x* by *v*. Note that no evaluation of *v* takes place at this level. This contrasts with traditional languages. Note also that the procedure environment *e* is not used. The execution of an *output statement* `Out e` (45) does not change the memory. It produces the output defined by the `SemOutExpr` semantic function which is not described due to a lack of space. When a *conditional statement* `If(e, s1, s2)` is executed and the condition *e* denotes an integer not equal to 0 (46), this execution is equivalent to the execution of *s₁* (47), else *s₂* is executed (49). The semantics of a *sequence of statements* is natural. Executing an empty sequence of statements `Seq ∅` neither change the output and nor the memory (50). Executing a statement *s₁* followed (⌢) by a sequence of

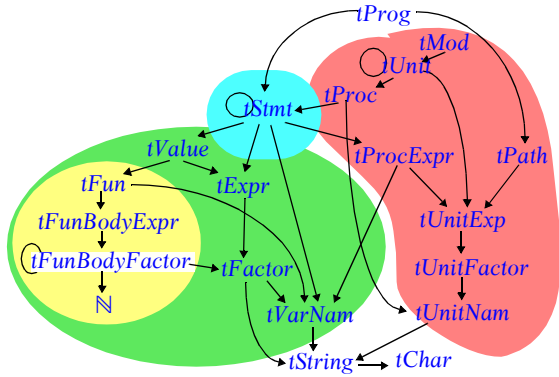


Figure 7. Dependency graph on syntactic domains

```

1  tOutput      == seq tString
2  tMem         == tVarName → tValue
3  tSemUndefVar == tVarName → tFactor
4  [ tCUnOP, tCBinOP ]
5  tCond       ::= CConst ⟨ ℤ ⟩
6               | CUnOp ⟨ tCUnOp × tCond ⟩
7               | CBinOp ⟨ tCBinOp × tCond × tCond ⟩
8  tUnitId     == seq tUnitName
9  tModId      == tUnitId
10 tUnitEnv    == tUnitId → tUnit
11 tProcEnv    == tUnitEnv × tImports × tModId

```

Figure 9. Semantic domains

```

12 SemStmt : tStmt → (tProcEnv × tMem → tOutput × tMem)
13 SemProc : tProc → (tProcEnv × tMem → tOutput × tMem)
14 SemProg : tProg → (tBuiltinEnv × tMod × tModId → tOutput)
15 SemMod : tMod → tUnitEnv
16 SemUnitExpr : tUnitExpr → (tUnitEnv × tModId → tUnitId)
17 SemUnitFactor : tUnitFactor → (tRscEnv × tUnitId → tUnitId)
18 SemImports :
19   tImports → (tUnitExpr → (tUnitEnv × tModId → tUnitId))
20 SemProcExpr : tProcExpr → (tProcEnv × tMem → tProcId)
21 SemOutExpr : tExpr → (tMem → tOutput)
22 SemCondExpr : tExpr → (tMem → ℤ)
23 SemExpr : tExpr → (tSemUndefVar × tMem → tExpr)
24 ParseCond : tExpr → tCond
25 SemCond : tCond → ℤ
26 ParseParamSeq : tExpr → seq tExpr
27 SemFunCall :
28   (tVarName seq tExpr) → (tSemUndefVar × tMem → tExpr)
29 SemFunBodyExpr :
30   tFunBodyExpr → (seq tExpr × seq tExpr → tExpr)
31 SemFunBodyFactor :
32   tFunBodyFactor → (seq tExpr × seq tExpr → tExpr)
33 SemStringifyExpr : tExpr → tFactor
34 SemStringifyFactor : tFactor → tFactor
35 SemConcatExpr : tExpr × tExpr → tExpr
36 SemConcatFactor : tFactor × tFactor → tFactor

```

Figure 10. Semantic functions

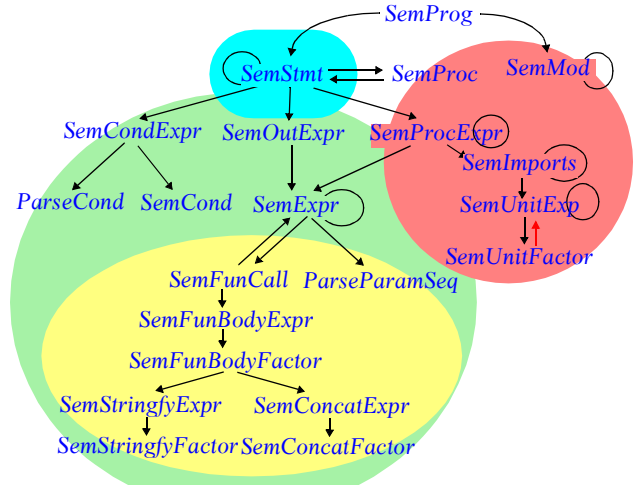


Figure 8. Dependency graph on semantic functions

```

37 w : tString ; e : tExpr ; es : seq tExpr ; f : tFactor
38 pe : tProcExpr ; x : tVarName ; xs : seq tVarName
39 b : tFunBodyExpr ; v : tValue ; s : tStmt ;
40 ss : seq tStmt ; r : tMem ; p : tProc ; p : tImports
41 g : tUnitEnv ; o : tOutput ; e : tProcEnv ; d : tModId
42 u : tSemUndefVar ; i : ℤ ; m : tMod ; un : tUnitName
43 unu : tUnitName → tUnit ; ui : tUnitId ; ue : tUnitExpr

```

Figure 11. Implicit variable declarations

```

44 SemStmt (Set(x,v)) (e,r) = (⟨ , r ⊕ { (x,v) } ⟩)
45 SemStmt (Out e) (e,r) = (SemOutExpr e r , r)
46 SemCondExpr e r ≠ 0 ⇒
47   SemStmt (If(e, s1, s2)) (e,r) = SemStmt s1 (e,r)
48 SemCondExpr e r = 0 ⇒
49   SemStmt (If(e, s1, s2)) (e,r) = SemStmt s2 (e,r)
50 SemStmt (Seq ⟨ ⟩) (e,r) = (⟨ ⟩ , r)
51 (o1, r1) = SemStmt s1 (e,r)
52 ∧ (o2, r2) = SemStmt (Seq ss) (e,r) ⇒
53   SemStmt (Seq(⟨s1⟩ ss)) (e,r) = (o1 ∧ o2, r2)
54 SemProcExpr pe ((g, p, d), r) = d' ∧ ⟨un⟩
55 ∧ g (d' ∧ ⟨un⟩) = UProc p
56   ⇒ SemStmt (Call pe) ((g, p, d), r) =
57     SemProc p ((g, p, d'), r)
58 SemProc (Proc s) (e,r) = SemStmt s (e,r)
59 SemMod m ⟨ ⟩ = m
60 un ∈ dom unu ⇒
61   SemMod (UMod unu) ⟨un⟩ ui = SemMod (unu un) ui
62 SemProg (Prog(p,s)) ((pb, sb), m, d) =
63   fst( SemStmt (Seq(sb, s))
64     (((SemMod m), (p ∧ pb), d), (λx:tVarName•Nil)))

```

Figure 12. Semantic equations for statements, procedures, modules and program semantics

statements ss (53) consists first in executing s_1 (51). This produces some piece of output o_1 and a memory r_2 . Then the statements ss are executed (52) using r_2 . The whole result (53) is the concatenation of the output ($o_1 \hat{\ } o_2$) and the memory after executing ss . Executing a **call statement** $Call\ pe$ (56) first requires to evaluate the procedure expression pe (54). If this expression refers to a procedure p (55) named un in a module d' , then this procedure will be executed but in a procedure environment in which the current module is d' (55). In other words, relative procedure expressions are relative to the module containing the expression, not to the caller module. This behaviour is very important to specify in the presence of procedure aliases.

4.2.3. Procedure semantics. The semantics of a procedure (13) is similar to the semantics of statements (12): executing a procedure means executing its body (58) since there is no need to handle parameters.

4.2.4. Module semantics. A module denotes a mapping between a unit name and a unit (59).

4.2.5. Program semantics. The execution of a program (14) depends on a builtin environment (p_b, s_b) (62), on a module m (this models the whole file system from the CPP point of view), and on the current module (i.e. the current directory in which the command line is executed). The statement s_b is executed before the body of the program (s) typically in order to define builtin variables; the builtin imports p_b are inserted after the user path (p). The execution starts in an empty memory, or better said in a memory where all variables are bound to the Nil value ($\lambda x:tVarName \bullet Nil$). Executing a program just returns the output. The final state of the memory is simply discarded (fst).

4.3. Global view on APP semantics

Figure 8 represents the graph dependency between all semantic functions: nodes are functions, edges are references between functions. Only the functions on the top have been presented in this paper. This picture clearly shows that the remaining functions greatly contribute to the complexity of CPP semantics. In fact the most difficult part is centred around the semantics of expressions. The semantics of an expression depends on its context of evaluation. For instance expressions playing the role of conditions in *If* statements are evaluated using their own language involving a specific parsing phase. Other discrepancy includes for instance the fact that the occurrence of an undefined variable is replaced by 0 if it appears in a conditional expression and by an empty token if it appears elsewhere. Some other issues are related to the recursive call of macro. The definition of semantics clearly states where this process stops, which is not always trivial to understand.

4.4. APP semantics in practice

Could some mathematical equations such as those above be useful from a practical point of view? The answer is yes. We built two CPP interpreters, one in OCaml, one in Java.

4.4.1. An CPP interpreter in OCaml. From the denotational semantics it is quite trivial to derive an interpreter by using a functional programming language. To validate the equations above, an interpreter was built in OCaml. The translation is quite direct thanks to OCaml pattern matching and to the ZML and ZLN packages [26]. These libraries provides implementations of most of the concepts found in the Z Mathematical Toolkit to ease the translation. In fact the OCaml prototype was build during the definition of the semantics to validate each equations.

It would be almost impossible to get confidence in an interpreter if the result could not be compared with a reference implementation. In our case we incrementally developed a set of test cases to check the conformance of our prototype (and therefore of the semantic definition) with CCCP. At this point it is also very important to stress that our prototype is far more simple than the CCCP implementation (about 200 lines in Ocaml vs. 10000 in C).

4.4.2. An CPP interpreter in Java. After a first stable version of the semantics, a prototype in Java was developed with two objectives in mind: (1) to cope with the performance issues of the OCaml prototype (it is implemented in a purely functional style and directly implements the equations), and (2) to ease the integration of this interpreter with exploration tools previously developed in Java. From a very concrete point of view, the interpreter was derived from the semantic equations by adding on each class defined in Section 3.4, a method *eval*. The signature of these methods directly depends on the signature of the corresponding semantic functions. Obviously, since Java is based on an imperative paradigm, parameters such as the memory are not duplicated but are modified “in-place”.

5. Discussion

The description of CPP presented in this paper may raise many objections both from practitioners and from theoreticians. We believe however that this contribution is valuable because of the balance between pragmatism and abstraction. This balance is needed in the case of CPP.

Theoreticians might argue that the definition provide above is not well founded. This is true. The first issuelem is related to the usage of the Z language which does not allow recursive definition of types. Each cycle in the graphs presented in Figure 7 is a potential problem. In fact, one of the major contribution of Scott in denotational semantics was to allow self-referential domains. This leads to the need

of using fix-point theory since in general semantic functions are defined as approximation within a lattice. In fact, there is a potential issue for each cycle in the graph presented in Figure 8. However, these issues are not important from our perspective. As pointed out in the introduction, our goal was to get an executable specification of CPP. The problem is on termination, but a careful study of all semantics functions shows that their execution will always stop. In particular macro definitions can be recursive, but their evaluation stops thanks to a rule embedded in CPP behaviour. In fact, the main problem we faced was to define the semantics of a tool that is specified by 10000 lines of C.

On the other side practitioners might argue that the description of CPP semantics is difficult to read. This is clearly an issue, especially since it is oriented towards tool builders. With this respect we should however mention our particular experience. The Java implementation of the reverse engineering tool was done during an internship by a professional with a huge experience on the maintenance of COBOL programs, but no knowledge about formal specifications. After a period of time, he got used to the Z notation and really found this technique useful to guide the implementation of the Java prototype. He found that the equations were so concise that it was far more convenient to refer to them instead of reading Java code. Some equations were translated by various pages of Java code which were much more difficult to understand.

6. Conclusion

This paper shows that CPP can be seen as a programming language. This analogy reveals very powerful, both from a conceptual and a practical points of view. From a conceptual point of view it brings a neat framework to study CPP using a well established body of knowledge. From a practical point of view, it provides a very concise definition of CPP. This definition can be used to adapt the many existing analysis techniques. As far as we know, all techniques proposed in the literature to handle CPP files are either ad-hoc solutions or take into account only a (small) subset of CPP features. Though useful, these approaches are limited in their scope. The approach presented in this paper is much broader, and should be seen as a reference foundation for further investigation.

7. References

- [1] J.M. Favre; “*An Approach to Support Software Maintenance and Reengineering-In-The-Large*”, (in french), Ph.D. dissertation, University of Grenoble, France, 1995
- [2] A. Morales, “*A Reverse Engineering Environment Supporting Program Families Analysis*”, (in french), mémoire CNAM, University of Grenoble, 1999
- [3] J.M. Favre; “*Preprocessors from an Abstract Point of View*”, Proc of ICSM and WCRE, 1996
- [4] A. Mahler; “*Variants: Keeping Things Together and Telling Them Apart*”, Chapter 3, in W.F. Tichy, Editor; “*Configuration Management*”, Trends in Software 2, ISBN 0-471-94245-6, John Wiley & Sons, 1994
- [5] W. M. Gentleman et al. ; “*Commercial Real-time Software Needs Different Configuration Management*”, Workshop on Software Configuration Management, 1989
- [6] D. Tilbrook, R. Crook; “*Large Scale Porting through Parametrization*” in USENIX, June 1992
- [7] A. Zeller; “*Configuration Management with Feature Logics*”, Technical report TR-94-01, Technische Universität Braunschweig, Germany, March 1994.
- [8] H. Spencer, G. Collyer; “*#ifdef Considered Harmful, or Portability Experience With C News*” USENIX, June, 1992.
- [9] A. Litman; “*An Implementation of Precompiled Headers*” in Software - Practice and Experience, 23:3, March 1993.
- [10] K.P. Vo, Y.F. Chen; “*Incl: A Tool to Analyze Include Files*” USENIX, June 1992
- [11] D. Spuler, A.S.M. Sajeew; “*Static Detection of Preprocessor Macro Errors in C*”, 1992
- [12] M. Krone, G. Snelting; “*On the Inference of Configuration Structures from Source Code*”, ICSE’1994.
- [13] P. E. Livadas, D.T. Small; “*Understanding Code Containing Preprocessor Construct*”, IWPC’1994.
- [14] M. Latendresse, “*Fast Symbolic Evaluation of C/C++ Preprocessing Using Conditional Values*”, CSMR 2003
- [15] R. Stallman; “*The C Preprocessor*”, GNU Project, Free software foundation, July 1992.
- [16] J.M. Favre; “*Understanding-in-the-large*”, IWPC’1997.
- [17] L. Ouarbya, S. Daninic, M. Daoudi, M. Harman, C. Fox; “*A Denotational Interprocedural Program Slicer*”, WCRE’2002
- [18] A. Cox, C. Clarke; “*Relocating XML Elements from Preprocessed to Unprocessed Code*”, IWPC’2002
- [19] B. Kullbach, V. Riediger; “*Folding: An Approach to Enable Program Understanding of Preprocessed Languages*”, WCRE’2001
- [20] I. Baxter, M. Mehlich; “*Preprocessor Conditional Removal by Simple Partial Evaluation*”, WCRE’2001
- [21] J.M. Gravley, A. Lakhota; “*Identifying Enumeration Types Modeled with Symbolic Constants*”, WCRE’1996
- [22] G. Badros, D. Notkin; “*A framework for preprocessor-aware C source code analyses*”, Technical Report UW-CSE-98-08-04, University of Washington, 1998
- [23] M. Ernst, G. Badros, and D. Notkin; “*An empirical analysis of C preprocessor use*”. Technical Report UW-CSE-97-04-06, University of Washington, 1997
- [24] M. Harsu; “*Translation of Conditional Compilation*”, Nordic Journal of Computing, 1997
- [25] Spivey; “*The Z Notation*”, Prentice Hall
- [26] J.M. Favre, “*ZML: Z Mathematical Library in OCaml*” and “*ZLN: Z-like Language Notation in OCaml*”, Information available at <http://www-adele.imag.fr/~jmfavre>
- [27] G.J. Badros, D. Notkin, “*A Framework for Preprocessor-aware C Source Code Analyses*”, Software-Practice And Experience, 30:907-924, 2000
- [28] ISO, “*Programming languages -- C*”, ISO WG14 N843, 550 pages, August 1998