UNIVERSITEIT VAN AMSTERDAM INFORMATICA -

BACHELOR INFORMATICA



UVA **UNIVERSITEIT VAN AMSTERDAM** 

# Parsing macros without the preprocessor

J.E. Hendriks Student number 0513210

June 15, 2009

Supervisor: Dr. Jurgen Vinju (CWI) Signed: Dr. Inge Bethke (UvA)

#### Abstract

This paper investigates creating a parser capable of parsing non preprocessed source code. The method investigated adds #define statements to the SDF grammar as a language extension. The results are discussed and compared with other solutions.

# Contents

1	Introduction	5
2	Transforming #define statements         2.1 Definition         2.1.1 Preprocessor         2.1.2 Macros         2.1.2 Macros         2.2 #define statement         2.2.1 #define Usage         2.2.2 Semantics         2.2.3 C99         2.3 Problems when transforming #define statements	7 7 7 8 8 8 9 9
3	The solution: Adding macros as a language extension         3.1       Syntax Definition Formalism(SDF)         3.2       Requirements         3.3       Design         3.4       Claims	<b>11</b> 11 11 12 13
4	Proof Of Concept         4.1       Method	<b>15</b> 15 15 16 17
5	Related work and evaluation         5.1       Proteus         5.2       CScout         5.3       C-Transformers         5.4       ASTEC         5.5       POC	<ul> <li>21</li> <li>22</li> <li>22</li> <li>23</li> <li>23</li> </ul>
6	Discussion         6.1       Conclusion         6.2       Threats to validity         6.3       Future work	<b>25</b> 25 25 25
Bi	bliography	27
Α	Syntax Definition Formalism         A.1 What is SDF	<ol> <li>29</li> <li>29</li> <li>30</li> <li>30</li> <li>31</li> </ol>

Since computers are created there have been rules on how to execute programs on the computer. At first this was done with a switch board, later with punched cards and eventually with source code. The rules for source code where called a programming language. And as the computers evolved the programming languages evolved too.

This development is a good thing, however, old programming languages usually cannot be used on new computer systems. When using old programming languages special restrictions are needed, because the old languages lack certain safety locks. A virtual machine is even necessary when hardware cannot be addressed.

Rewriting the programs solves a great deal of the problems. However, the work involved doing it manually is huge. Automation might be a good solution. There are many solutions for rewriting source code. Transforming is the process of rewriting source code. Transforming is a source-tosource process. Refactoring is rewriting a code style within the same language. Reenginering source code is the transformation from one language to another language. Reenginering is the most important usage of transformations.

The transforming of macros within C source code is still a problem. Two compilers must be used, one for the macros (the preprocessor) and one for the source (gcc compiler). The last compiler step is no problem when transforming, however, the macros from the first step are mostly used for readability, usability and codability. After transforming these macros should be put back in the source to keep readability, usability and codability. This is what makes transforming macros non trivial and extremely difficult. The solutions for macro transformations are scarce. However, the difficulty of the problem is that it gives non expected or incomplete results. It is important to have a solution capable of giving complete valid results.

An unexplored solution was given by J. Vinju who has done much research on this subject. He proposed adding macros to the C language as an extension, bypassing the C preprocessor (CPP). There has not been that much research into this method.

The contribution of this thesis is a unique prototype based on J. Vinju his idea. This unique prototype has three qualities existing solutions do not have. These qualities are accuracy, locality and completeness. The prototype is tested in a small case study and compared to the existing solutions.

# Transforming #define statements

This paper is about a new method on transforming #define statements. To be able to transform #define statements its usage and definition must be known. Therefore the official definitions are needed. They are extracted from the ANSI C standard. The other part that will be explained in this chapter is why transforming macros and especially the #define statements is difficult.

## 2.1 Definition

The first thing to do is to take a look at the C standard for the definition and usage of macros. The standard used is the ANSI C89 standard because ANSI C89 is the most strict and most commonly used standard for the C language. This paper will only contain a summary of the standard definition. It is possible to look up the standard to get a complete definition [1].

#### 2.1.1 Preprocessor

The ANSI C89 standard contains information about a preprocessor. This preprocessor should always be executed before actually compiling the source code. The most important task of the preprocessor is to replace all macros with the according source code. Other tasks are: removing all comments and cleaning the code for the compiler itself [14].

The macro-replacing of the preprocessor makes transforming C code difficult. Transforming code processes only one language at a time. Transforming the macros and the C language at the same time is therefore impossible.

It is possible to execute the preprocessor to get the expanded source code. Transforming is then a straightforward process. After transformation the code is to be un-preprocessed to get the original macros back. There is no solution for un-preprocessing source code. This makes transforming source code with macros unusable when using the preprocessor.

Adding macros as a language extension combines the macro and C language. Transforming source code is a straightforward processed if the macros are part of the C language. This project will examine this method of extending the C language with macros. Only the #define statement is examined. Other macros will only be mentioned briefly.

#### 2.1.2 Macros

Macros always begin with a special macro character, the #. Then a command and its arguments follow. The macro statement is ended by a new line [8].

A possible macro for a preprocessor is the #include macro. It is usually a copy-paste of other sources into the source that will be compiled. #define statements from the included sources are added too.

Another macro is the #define statement. Using #define statements give the possibility to use other (shorter) names for certain functions or expressions. #define statements also make it possible to use the coding style from other languages. For example using BEGIN and END instead of { and }. The #ifdef and #ifndef statements can be used to check if a #define statement is defined in the past. Using #ifndef gives the possibility to create an "include once" operation (see Source example 1).

```
Source example 1 "Include once" using #ifdef
#ifndef H_EXAMPLE
#define H_EXAMPLE
/* Place code here */
#endif
```

The last type of macro is the #if/then/else macro. This macro is used to include or exclude certain pieces of code during compilation. For example if the code is written for UNIX, windows and a mac the includes are different for all these operating systems. Solving this problem uses the #if/then/else macros.

## 2.2 #define statement

The #define statement is defined in the ANSI C standard with these rules:

- #define identifier replacement-list new-line
- #define identifier ( identifier-list<opt>) replacement-list new-line
- #undef identifier new-line

The identifier is called macro-name in this paper and the second part of the statement is called macro-replacement.

#### 2.2.1 #define Usage

A #define statement may be used in any kind of language, because it is an extension to the given language. After processing it needs to generate valid source code. Declaring a #define statement requires undefining previous #define statements with the same macro-name. Otherwise the new #define statement does not work. This is a limitation of the preprocessor. It will replace all macro-names until the end of the file or until a corresponding #undef statement is reached.

When declaring a #define statement its scope lasts until the end of the file. When the source is included its scope extends into the parent source. The method to end the scope is to undefine the #define statement. When undefining the #define statement it is not needed to mention the arguments in the macro-name, e.g.:

#define macro-name(argument 1, argument 2) #undef macro-name

#### 2.2.2 Semantics

Apart from the usual restrictions and semantics on macro usage there are a few restrictions on function-macros. These are to make sure there is no confusion when parsing the macros. The most important restriction is that parentheses must be balanced within the function macro-names and arguments. If parentheses are not correctly balanced the preprocessor cannot guarantee the correct replacement of the macro-name in the source code. This restriction is not needed for normal macros, the preprocessor just replaces the entire macro-name with the macro-replacement.

#### 2.2.3 C99

As an addition to the C89 standard, C99 adds variadic macros. Variadic macros create the possibility to have macros with a variable number of parameters. Therefore it is possible to create wrappers for functions like "printf". The C89 standard uses variable parameter lists with a call to a special function ("vprintf") instead of the normal "printf" function.

# 2.3 Problems when transforming #define statements

When transforming source code with macros problems arise. Explaining the different methods of using a #define statement and pertaining problems will make it possible to verify the results of this project [6, 5].

The following list itemises all possible usages for a #define statement [8].

- 1. #define [macro-name]
- 2. #define [macro-name] [macro-replacement]
- 3. #define [macro-name(arguments)] [macro-replacement(arguments)]
- 4. #undef [macro-name]

#### 1. #define [macro-name]

The first list item is a blank macro. It is mostly used for "include once" constructions (See Source Example 1). It can also be used in combination with another preprocessor that is written for a certain project. For example a way of having debug information only at development time can be done by replacing all the macro-names with special debugging code.

Processing this #define statement in a parse tree is straightforward. The #define statement is ignored by the program. When this #define statement is used as external preprocessor code it could be parsed as layout. This #define statement can be used for #if/then/else constructions. However, this will not be investigated within this project.

#### 2. #define [macro-name] [macro-replacement]

The second list item is a normal macro-name to macro-replacement replacing. It is often used to overcome large one-lined sections of code that are identical. For example a "printf" statement that prints the current state of the system. But this statement is to small to give its own function.

The difficulty in this statement is not the normal usage. The difficult #define statements are those that have macro-replacements that are not valid C code. These macro-replacements need code from outside the #define statement to become valid C code. For example a partial "if" statement as in Source Example 2. The code in this example generates valid C code after preprocessing but when trying to parse only the #define statement problems arise. The nature of these problems is due to splitting the macro-replacement. Giving all parts their meaning according to the C standard is a difficult task when the macro-replacement is not valid C.

```
Source example 2 Partial if statement
#define IFEX if(example
int main(void) {
    IFEX == true) {
        printf("Example is true");
    }
}
```

#### 3. #define [macro-name(arguments)] [macro-replacement(arguments)]

The third list item is used to create small inline functions. The arguments given in the macroname are placed in the macro-replacement. Transferring arguments from the source code to the #define statement is the difficulty of this type of #define statement. Correct splitting of the arguments can be difficult, but the C standard requires arguments to have balanced parentheses and this makes splitting the #define statements more straightforward.

Another problem with the argumented #define statements is the question how to parse the macro-replacement. This is the same problem as the #define statements without arguments except for the fact that not just the macro-replacement needs parsing but also the arguments.

#### 4. #undef [macro-name]

Ending the scope of a #define statement requires a corresponding #undef statement. When parsing a given #define statement it is possible to stop when the corresponding #undef statement is reached. Thus using the #undef statement gives no parsing problems because parsing is done in order of appearance in the code.

# The solution: Adding macros as a language extension

The problem still requires a good solution since the solutions in the previous chapter all have their good and bad properties. The possible solution this project examines is the extending of the SDF grammar, c.q. adding macros as a language extension.

The first step explains what will be extended. The extending will be done in Syntax Definition Formalism (SDF) grammar. SDF is a meta syntax used to define context-free grammars. This chapter explains the SDF in a nutshell. An extensive explanation can be found in Appendix A.

# 3.1 Syntax Definition Formalism(SDF)

SDF is a way to write context-free grammars(CFG)[7]. These CFGs are used to interpret a source code file. This interpretation is done using a scannerless generalized left-right parser (SGLR). The SGLR creates a parse tree containing every line of code, every function from the source code file.

These parse trees can be used to transform source code and the parse trees should therefore contain all information about macro statements. This means that macros must be added before building the parse tree. Thus adding the macros must take place in the SDF grammar.

There are two possibilities to add macros to the SDF grammar. The first possibility is after an idea of J. Vinju: change the existing grammar to make it possible to process macros. The other possibility is to not change the SDF grammar, instead extend the SDF grammar with new rules.

## 3.2 Requirements

The next step is to define the requirements for the possible solutions. The functional requirement is to create a parser capable of parsing non preprocessed source code. The other requirements can be found in the following paragraphs. These requirements are used to define the quality of the solution investigated. Without these requirements an unwanted solution can be created. This solution only works on a special set of code, has a complete database of all possible macros, and every unknown statement is left untouched.

Accuracy Accuracy is the most important requirement. Accurate solutions have the same information when processing preprocessed code as they have when processing non-preprocessed code. E.g. the solution does not leave out certain details. Using parse trees the parse tree of the non-preprocessed source code must contain all parse tree productions of the preprocessed source code. The solutions usually get more information from the non-preprocessed source code. They therefore have a superset of information. A parse tree superset is defined as a parse tree with



Figure 3.1: Utopian SDF architecture

the same non lexical productions and identical right-hand side lexical to non-lexical production as the preprocessed parse tree.

Locality The locality requirement is to ensure all information is stored in the same way normal source code is stored (e.g. no extra database like systems). This requirement makes sure existing methods for transforming source code can be kept identical or only need minor changes.

**Completeness** The last requirement: the method should work for all instances of a certain macro type, e.g. if the method works for #define statements the method should work for all instances of the #define statement.

## 3.3 Design

An utopian solution (Figure 3.1) is to change the SDF grammar to accept all types of macros. This utopian solution seems impossible to create. However, creating a near utopian solution is possible.

This near utopian solution extends an existing SDF grammar with grammar to process only the #define statements of a given source. The solutions architecture (Figure 3.2) is explained in the following paragraphs.

**Extension Creator** The main part of this solution is the extension creator. The extension creator processes the original source code into a changed source and an extension for the SDF grammar.

The changes made to the source are simple. They replace the macro-names with new macronames. These new macro-names are used by the parser to link them with the macro-replacements. These macro-replacements are added in the SDF grammar extension. The next chapter explains these changes and the extension in more detail.



Figure 3.2: Extended SDF architecture

**Parse Tree Creator** The parse tree creator is an existing tool called SGLR (scannerless generalized Left-Right parser). This tool creates a parse tree with the changed source and the extended SDF grammar.

**Unparser** The unparser tool recreates source code out of a parse tree. This parse tree can be transformed or left intact. The unparser tool is already a part of the meta-environment.

**Unextender** Because the original source is changed by the extension creator it needs to be changed back. The unextender replaces all special macro-names with the original macro-names. When the parse tree is not transformed the result of the unextender is identical to the original source.

# 3.4 Claims

Our claim is this nearly utopian solution fulfils all requirements as stated in Section 3.2.

Accuracy The original SDF grammar does not change and the solution adds new rules to process #define statements.

Locality The only changes are made in the source and an extension of the SDF grammar. There is no external database or list.

**Completeness** It is not obvious if the solution is complete. Completeness is subject of the next chapter. We claim that this solution is valid for the #define statements.

# Proof Of Concept

The claims mentioned in previous section (See Section 3.4) are tested with a Proof Of Concept program (POC). This POC is capable of processing C source files. These processed sources can then be compared to code from the normal C preprocessor. The results of this comparison give an indication of the validity of the claims.

### 4.1 Method

To prove the claims with the POC the meta-environment is used. This system consists of a number of commandline programs and a graphical user interface for these programs[3]. Some bugs interfering with the special identifiers used by the POC made it impossible to use the graphical user interface. The POC process therefore only uses the commandline interface. However, if the bugs are removed from the graphical user interface, the POC can be used and viewed graphically too.

The project consists of two programs. The first program is the actual POC. This POC creates a parse tree with #define statements. The second program is a program to compare two parse trees. This program visualises the results of the POC.

#### 4.1.1 POC program

The first program, the POC, reads a given source and processes it into a modified source and an extension to the SDF grammar. The modified source only replaces the macro-names with a series of identifiers.

There is no SDF rule to parse #define statements as layout. Therefore the POC removes the #define statements to make parsing possible. This is similar to the preprocessor behaviour. These #define statements would otherwise give problems when creating a parse tree.

However this involves adding a SDF rule to parse #define statements as layout. Keeping the #define statements in the source is needed for a complete transformation. For our proof keeping the #define statements is not necessary. An example of the modified C source can be found in Source example 3 and 4.

```
Source example 3 Unmodified C source code
#include <stdio.h>
#define P printf("example");
int main(void){
    P
}
```

Source example 4 Modified C source code #include <stdio.h>

```
int main(void){
   $def0$$def1$$def2$$def3$$def4$
}
```

The other part of the POC is the SDF grammar. As can be seen in Source example 4 the original macro-name has been replaced by a new macro-name. This new macro-name makes it possible to add SDF grammar for all macro-replacement parts. The new macro-name consists of small unique parts. The uniqueness is obtained using the dollar signs as the dollar signs are not a part of the C language. When using another method of obtaining uniqueness (one without dollar signs) it is possible to keep the source code compilable. However this involves thorough investigation of the source code and it is not needed to prove the concept. Source example 4.1.1 contains the SDF grammar corresponding with Source example 4. The small unique parts of the macro-name can be seen in this SDF grammar.

The POC creates context-free syntax SDF grammar, but it can just as well create a lexical syntax instead. Using the lexical syntax creates some ambiguities and therefore the context-free syntax is preferred. Because of other ambiguities the "{prefer}"-string is added to some rules. Leaving out this "{prefer}"-string creates some ambiguities when a statement in the parse tree has identical children. For example if there is a normal "printf" statement in Source example 3 it generates an ambiguity between the "printf" statement and the SDF rule that says "printf" is an Identifier.

Source example 5 Extended SDF grammar

```
module poc
imports languages/ansi-c/syntax/Default-C-After-CPP
exports
context-free syntax
"$def2$" -> "\"example\""
"$def4$" -> ";"
"\"example\"" -> StringConstant {prefer}
"$def3$" -> ")"
"$def3$" -> ")"
"$def0$" -> "printf"
"$def1$" -> "("
"printf" -> Identifier {prefer}
context-free start-symbols
TranslationUnit
```

#### 4.1.2 Diff tool

The second program written for the project is a diff tool. It is capable of comparing two parse trees. Comparing is done by extracting all productions from the parse tree. These productions contain the complete production rule, e.g. when the rule looks like "left part -> right part {attributes}" the production is "prod(left part, right part, attributes)".

The productions also have an argument containing the corresponding part of the parse tree. The argument is omitted when comparing two parse trees. Omitting the argument is done because both parse trees are created using nearly the same SDF grammar. This assures there are no extra ambiguities added to the parse tree, making them comparable.

To compare both parse trees all the productions are listed. Next the diff tool removes all identical productions that are in the same order from both lists. The following step is to remove all partial identical productions. These are the productions that only have identical right parts and attributes. This is permitted because there are more ways to get the same production. E.g. printf may be in a #define statement or just in the source code. Both have the right production part of Identifier, but the left part differs.

Because of the way the POC works there are two additions to be made to the diff tool. The first is removing of the "{avoid}" attribute in the first list (the one with the POC parse tree). The "{avoid}" was added to solve some ambiguities that gave the exact same result. E.g. "printf" -> Identifier is the same as lex("printf") -> Identifier. The second addition is the possibility to remove all lexical productions before comparing.

Source example 6 Special SDF rule								
[L]?	[\"]	(	([\\]~[])	~[\\\"]	)*	[\"]	->	StringConstant

Removing the lexical productions is allowed because of the way SDF grammar works. Because the grammar is not in normal form the parser puts it in normal form. This gives certain SDF rules a lot of extra lexical steps in comparison to the POC created SDF rules. Take a look at Source example 6. It contains different space separated parts in the left part of the production rule. These different parts all get there own parse rule, whereas the POC generated SDF only has two rules (see Source example 7). In this example the "text" is split into the quotes and the text part when using the normal SDF rules. Next it is lexically combined into the StringConstant. The POC processes the "text" example in one step, without splitting the string.

```
Source example 7 Special POC SDF rules
$def_0$ -> "\"text\""
"\"text\"" -> StringConstant
```

Removing all lexical productions will overcome this difference. It is possible to keep the lexical productions and execute the diff tool with the verbose argument to see what productions are left.

## 4.2 Results

Comparing the parse tree generated by the POC and the parse tree generated by the C preprocessor gives a view how well it actually works. The preprocessor is known for giving correct code, but this code is without information about the #define statements. The code produced by the POC contains all information in the preprocessed code and has extra information about the #define statements.

To correctly compare a given source a few changes are made before starting to compare. These changes are necessary because of the way the C preprocessor works. The C preprocessor removes all comments and expands all macros. Therefore remove all comments from the original code and also remove the macros not addressed by the POC. These are the #include and #if/then/else statements. A complete scheme of the compare process is shown in Figure 4.1.

The comparison is done with a few small sources and a few large ones. The comparison also is done with and without lexical information to show the difference.

```
Source example 8 C source without #define statements
#include <stdio.h>
int main(void){
    printf("example without #define statements");
```

```
}
```



Figure 4.1: Scheme on how the POC is compared

Used source:	Source example 3	Source example 8	Quake2-3.21: linux/vid_menu.c	Gcc-4.4.0: libcpp/expr.c		
With lexical productions						
# Prods POC	80	110	5376	81716		
#  Prods CPP	84	110	5333	60833		
# Identicals removed	70	110	1001	54380		
# Semi identicals removed	3	0	4180	7354		
# Prods left in POC	9	0	221	21142		
# Prods left in CPP	13	0	178	259		
Without lexical pr						
# Prods in POC	52	43	2605	40940		
# Prods in CPP	43	43	2542	27511		
# Identicals removed	40	43	2542	23862		
# Semi identicals removed	3	0	0	4135		
# Prods left in POC	9	0	63	13520		
# Prods left in CPP	0	0	0	91		

Table 4.1: Comparing different sources

The small sources used are Source examples 3 and 8. Source example 8 is a source without any #define statements. As expected all productions are identical in both the POC and preprocessed code. They leave no productions in the POC and CPP list. This proves the source is not edited when no #define statements are available.

The large sources are taken from the quake 2 source and the gcc source. Some small changes where made to make them parse for the preprocessed source. In both large sources the code generated by the POC then also parsed without problems. The changes made where due to incompleteness of the C SDF grammar. All changes where exceptions on existing rules and therefore it is correct to change them for the comparison.

Taking a closer look at Table 4.1 it can be noticed that some productions are left, in the CPP production list for the gcc source. We expect zero instead of 91 productions.

Looking into the productions left in the list and taking a look at the #define statements it turns out these productions are the result of the POC not being able to parse expressions. All the productions that are left are from a few #define statements containing expressions. Ernst et al.[4] concluded simple preprocessor statements are most used. Expression statements are hardly used. Looking at the great number of productions left in the POC it is obvious that the method gives a lot of extra information. Therefore the POC result is surely a superset of the CPP parse tree.

# Related work and evaluation

To position the method proven with the proof of concept within the field of work a comparison to the existing solutions is needed. See Table 5.1 for a summary of the comparison. The results of this table will be discussed in the following sections.

	Accuracy	Locality	Completeness
Proteus	0	++	+
CScout	-		++
C-Transformers	++	++	
ASTEC	0	++	++
POC	++	++	++

Table 5.1: Comparing POC to other solutions

The solutions are compared using the requirements mentioned in Section 3.2. These requirements are

- Accuracy (How accurate is the method. It should have a accuracy of a 100% to be usable)
- Locality (All information is kept in the source)
- Completeness (How complete is the method)

#### 5.1 Proteus

Proteus is a system developed by Bell Labs, which is now a part of Alcatel-Lucent. The Proteus system has its own transformation language that can be used to transform different languages. The information about the languages is split into coding style and coding syntax: the code can be changed without changing the layout. After the transformation it is possible to recreate the code with its original layout and usage [15, 13].

The macros are processed by using the method of the C preprocessor and by adding extra comments to make sure the programs transformer has knowledge of the macros. This preprocessor method uses a set of predefined statements for the conditional macros. Proteus therefore needs to process a source file multiple times to transform all the code.

The special language (YATL, Yet Another Transformation Language) makes it possible to concentrate only on the coding and not on the layout. The Proteus system has been tested on millions of lines of code of Alcatel-Lucent. Accuracy (0) The way conditional statements are handled makes it impossible to keep track of all information. Proteus is accurate for all macro statements actually processed. This keeps Proteus from getting a negative rating.

Locality (++) Source code is converted to a special language. This converted code is processed by Proteus, just like other source code would be processed. Therefore the locality requirement is fulfilled.

Completeness (+) The tool is only tested on Alcatel-Lucent software. This is a bad thing, because there may be macro constructions not used in the Alcatel-Lucent software. However the Alcatel-Lucent software is large and contains many macro statements. Proteus therefore gets a + rating for the completeness requirement.

## 5.2 CScout

CScout does not handle macros to the full extent of their possibilities. CScout only implements macros by following the macros through the preprocessor. Because of the limited support for macros it has a few shortcomings. One of these shortcomings is it will only process the code executed. E.g. if certain macros are within conditional statements they will only be processed if the conditional statement is true. This gives the need to process the source several times with different conditional statement values.

Different #define statements cannot be kept apart when following these statements through the preprocessor. However, this only gives problems in rare cases.

Another problem that can occur is when a #define statement is never accessed. It will be ignored because CScout only processes code that is actually used (hence the similar problem with the conditional statements) [11, 12].

Accuracy (-) The method is not always accurate. Because it follows the #define statements through the preprocessor it cannot always keep them apart. This makes transformation inaccurate. The accuracy requirement is therefore violated.

Locality (- -) Because CScout follows macros through the preprocessor it keeps track of the macros using a database like system. Therefore the locality requirement is violated.

**Completeness** (++) #define statements are followed through the preprocessor. Therefore CScout knows the usage of the #define statements. The completeness requirement is fulfilled.

# 5.3 C-Transformers

C-Transformers uses some software of the Stratego/XT framework, which uses functions from the meta-environment. C-Transformers add their own preprocessor method to the framework, and thus also their own un-preprocessor method. The goal of C-Transformers is to edit human readable C++ code and transform the source code into efficient generic code. The code is used and tested in the Olena platform (image processing) and the Vaucanson project (finite state machine manipulation platform) [2, 9].

Accuracy (++) C-Transformers uses a POSIX like preprocessor to expand all macros. Because a preprocessor is used all macros are accurately expanded and transformed. The accuracy requirement is fulfilled.

Locality (++) As all code is expanded into the original source no information is kept outside the source. The locality requirement is therefore fulfilled.

**Completeness** (- -) C-Transformers does not implement layout or macro preservation. All code is kept expanded after transformation. This issue makes C-Transformers unusable. C-Transformers violates the Completeness requirement because of this issue.

# 5.4 ASTEC

There is a solution that looks similar to the intended solution. Macroscope processes the C source and converts it into the Astec language. The Astec language is a superset language of C. To convert C source code into Astec language Macroscope replaces all macros with equivalent Astec macros. After replacing it is possible to transform the Astec source code into safer C. The difference between our solution and the Astec solution occurs here. Astec intends to make C safer. Therefore Macroscope adds lots of extra information to the Astec macros. For example the statement "#define M(x) ((x)+2)" changes into "@macro int M(int x) = x+2;". Because of the extra information Astec is not usable for creating parse trees [10].

Accuracy (0) The macros are translated into a new language. The translation is accurate. However Macroscope adds extra information not in the source macro. This makes it less accurate. Astec does not fulfil or violate the accuracy requirement.

Locality (++) Source code is converted to a special language. This special language makes transforming possible without external files. Therefore the locality requirement is fulfilled.

Completeness (++) Translating the macros into the new language makes it possible for this method to be complete for all types of macros.

# 5.5 POC

To make the table complete the POC is added. These paragraphs cover the accuracy, locality and completeness of the POC.

Accuracy (++) The implemented parts are accurate and tested on source code of quake 2 and the gcc compiler. The proof of concept is not yet accurate to its full extent because a parser for the expressions in #define statements is missing. Adding such a parser would give a 100% accuracy.

Locality (++) POC keeps all changes in the source code and adds some rules to the SDF grammar. POC does not use a database like system. It fulfils the locality requirement.

Completeness (++) The completeness of the POC is set to ++. It is based on the current results and one could argue that because of my definition of completeness it may not be given a '++' or even a '+'. However it is given a '++' rating. The lack of an expression parser is an issue to be solved. Implementing the parser is straightforward because it uses the C SDF grammar. However implementing the expression parser requires a lot of time.

# Discussion

# 6.1 Conclusion

The outcome of the proof of concept implies that the claims stated are valid.

Accuracy The POC is tested on parts of the quake and gcc source. These sources are 100% accurate compared to the preprocessed source. The results of the POC are a superset of the preprocessed results. The original SDF grammar is not changed and the extension makes it a superset too.

Locality The only changes are made in the source and an extension of the SDF grammar. There is no external database or list.

**Completeness** The solution is capable of processing all types of #define statements. Adding a manual expression parser even makes it possible to parse expressions in the macro-replacements. Therefore the solution is complete (See Section 4.2, the completeness requirement).

## 6.2 Threats to validity

Accuracy Is the accuracy requirement as obvious as claimed? Extending the SDF grammar could override existing SDF rules. However, the diff tool compares the productions making sure the extended SDF grammar does not conflict with existing SDF rules.

**Locality** Like the accuracy requirement it can be questioned if the locality requirement is as obvious as claimed. It can be asked if adding an extension to the SDF grammar does not violate the locality requirement. The SDF extension is just an addition to link macro-names to macro-replacements. Therefore it is not a database like system, fulfilling the locality requirement.

**Completeness** The Completeness requirement demands the solution works for all instances of a #define statement. It can be argued the proof of concept is not tested on enough code and not on all instances of #define statements. However, the used sources contain virtually every type of #define statement. The sources are selected by taking large software projects and getting the sources with many diverse #define statements. For the proof of concept the chosen sources are a representative way of proving its completeness.

# 6.3 Future work

This thesis only proves the concept. It needs further research and implementation. Like the previous sections this section is also divided into the three requirements.

Accuracy The POC lacks a parser for expressions in #define statements. For the method is a "proof of concept" only a simple expression parser is implemented. Adding a complete expression parser is straightforward but it consumes a lot of time.

Locality The locality requirement is fulfilled completely. However, future work can involve examining the possibility to leave the original source code unchanged.

**Completeness** The #define statements have been examined and are proven to work with the proof of concept. To have practical use of this method extra research into other statements such as #if/then/else and #include is required.

Another tool that has to be created is the unextender. It is a straightforward tool to return changed macro-names back to the original macro-names.

# Bibliography

- A. ANS. X3. 159-1989-Programming Language C. American National Standards Institute Inc, 1989.
- [2] A. Borghi, V. David, and A. Demaille. C-Transformers: a framework to write C program transformations. *Crossroads*, 12(3):3–3, 2006.
- [3] M.G.J. van den Brand, A. van Deursen, J. Heering, HA de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, et al. The ASF+ SDF Meta-Environment: a component-based language development environment. *Electronic Notes in Theoretical Computer Science*, 44(2):3–8, 2001.
- [4] MD Ernst, GJ Badros, and D. Notkin. An empirical analysis of C preprocessor use. IEEE Transactions on Software Engineering, 28(12):1146–1170, 2002.
- [5] J.M. Favre. CPP denotational semantics. In Third IEEE International Workshop on Source Code Analysis and Manipulation, 2003. Proceedings, pages 22–31, 2003.
- [6] A. Garrido and R. Johnson. Challenges of refactoring C programs. In Proceedings of the international workshop on Principles of software evolution, pages 6–14. ACM New York, NY, USA, 2002.
- [7] J. Heering, PRH Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SD-Freference manual. ACM SIGPLAN Notices, 24(11):43–75, 1989.
- [8] Free Software Foundation Inc. Macros the c preprocessor. http://gcc.gnu.org/onlinedocs/cpp/Macros.html#Macros, April 2009.
- [9] M. de Jonge, E. Visser, and J. Visser. XT: A bundle of program transformation tools. In Workshop on Language Descriptions, Tools and Applications (LDTA01), volume 44. Citeseer.
- [10] B. McCloskey and E. Brewer. ASTEC: a new approach to refactoring C. ACM SIGSOFT Software Engineering Notes, 30(5):21–30, 2005.
- [11] D. Spinellis. CScout: A Refactoring Browser for C.
- [12] D. Spinellis. Cscout shortcomings. http://www.dmst.aueb.gr/dds/cscout/doc/short. html, August 2004.
- [13] D. Waddington and B. Yao. High-fidelity C/C++ code transformation. Science of Computer Programming, 68(2):64–78, 2007.
- [14] Wikipedia. C preprocessor Wikipedia, the free encyclopedia. http://en.wikipedia.org/ wiki/C\_preprocessor, April 2009. [Online; Accessed 11-May-2009].
- [15] B. Yao, W. Mielke, S. Kennedy, and R. Buskens. C Macro Handling in Automated Source Code Transformation Systems. In 22nd IEEE International Conference on Software Maintenance, 2006. ICSM'06, pages 68–69, 2006.

# APPENDIX A

# Syntax Definition Formalism

This appendix is meant for readers without basic knowledge of Syntax Definition Formalism (SDF). It explains SDF in more detail but leaves out unused parts of the SDF definition.

# A.1 What is SDF

SDF is a way of writing context-free languages. SDF combines lexical and context-free syntax in a single formalism [7].

There is not a great difference between lexical and context-free syntax. The difference is, both syntax's are kept in different name spaces and the lexical syntax is linked to the context-free syntax when creating the parser. Source example 9 shows how lexical syntax and context-free syntax are both in different name-spaces and how they are linked together.

```
Source example 9 Lexical to context-free syntax linking
```

```
lexical syntax
 "a" -> A
context-free syntax
 "b" -> A
This SDF code is linked in the following manner:
 "a" -> <A-LEX>
 <A-LEX> -> <A-CF>
 "b" -> <A-CF>
```

## A.2 Syntax

The SDF syntax is written in production rules. The productions rules consist of a terminals and non terminals combined. A production looks like  $\langle Symbol \rangle^* - \rangle \langle Symbol \rangle$ . The production must be read as a definition of a symbol. The right-hand side is defined by the left-hand side of the production.

A symbol is a terminal or non terminal. Non terminals are all defined on the right-hand side of the production before being used on the left-hand side. The terminals are character class symbols (All single characters, numbers and other symbols). They form the link between context-free syntax and the actual source code.

Start-symbols make it possible to parse a source. The start-symbols are the root of the parse tree and can be lexical or context-free. This project only uses a context-free start-symbol as the C grammar has a context-free start-symbol.

# A.3 Attributes

Adding attributes to the productions gives the possibility to solve ambiguities. Ambiguities are strings that can be parsed in multiple ways with the same grammar (See Source example 10). The horizontal ambiguities are solved using a prefer or avoid attribute. The vertical ambiguities are solved using associativity (left, right, assoc, non-assoc) or reject attributes. Not all ambiguities can be solved using these attributes. However the ambiguities introduced by the POC can be solved with the avoid and prefer attributes.

Source example 10 Simple context free grammar ambiguity

```
The context free grammar:

A \rightarrow A + A \mid a

Two ways of parsing the string a + a + a

A \rightarrow A + A A \rightarrow A + A

-> a + A -> A + A + A

-> a + A + A -> a + A + A

-> a + a + A -> a + a + A

-> a + a + a -> a + a + a
```

## A.4 Example

Bringing all the parts together can give something like Source example 11. The module and import are yet unexplained. The module is the name of the SDF module. All parts need a Module name as this makes it possible to import other SDF modules using the import operation.

This example can read all files that contain numbered textual lines. Its parse tree has a root called Source with immediate children called Line and Text. Eventually normal characters are in the leafs. The example imports the DefaultC syntax. However, it cannot be used because there is no production linked to the C grammar.

```
Source example 11 SDF code
```

```
module languages/example/syntax/Example
import languages/ansi-c/syntax/DefaultC
lexical syntax
"lexical sample" -> LexText {avoid}
"\"" -> Quote
[a-zA-Z] -> String {prefer}
context-free syntax
"Sample" -> Text
"Other" -> Text
[0-9]+ -> Number
LexText -> Text
Quote String Quote -> Text
Number Text+ -> Line
Line* -> Source
context-free start-symbols
Source
```



Figure A.1: Generating a Parse Tree with SGLR

# A.5 Parsing and transforming

For parsing (See Figure A.1) and transforming source code a parse table needs to be generated. This parse table is created by the sdf2table tool. Using the SGLR tool (scannerless generalized LR parser) to parse the source with the parse table it can build a parse tree of the source. This parse tree can then be changed and transformed into another parse tree. The transforming is done by searching parse tree snippets. These snippets can be replaced by other snippets. The new parse tree can be unparsed again into normal source code.