On dynamic extensions of context-dependent parser — Extended Abstract —

Jūlija Pečerska[1], Artjoms Šinkarovs[2], Pavels Zaičenkovs[3]

[1] University of Latvia, Raiņa bulvāris 19, Rīga, Latvija, LV-1586
[2] Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS, United Kingdom
[3] Moscow Institute of Physics and Technology, 141700, 9, Institutskii per., Dolgoprudny, Moscow Region, Russia

Abstract. The idea of using a preprocessor before actual compilation is proven to be useful, however, automatic verification of a program that uses preprocessor macros is extremely hard. The main difficulty comes from conceptual separation of grammatical rules of the language and substitution mechanisms underneath a preprocessor. For example, C/C++ preprocessor is included in the language standards, but still the expressiveness of this tool is very limited in both computational power and syntax extension capabilities. Modern programming languages define a syntax that cannot be described precisely using context free grammars. This problem is well known, and as a solution there are parser generators like ANTLR, which generates LL(*) parsers rather than LALR/LR. Nevertheless, a lot of real-world parsers are implemented by hand.
In this paper we present a way of building a preprocessor that allows introducing dynamic changes to the grammar and that works on top of any recursive descent parser that meets certain requirements. The preprocessing step consists of 2 substeps. First is matching a list of tokens against a regular expression built on tokens and encoded production names of the given grammar. And the second is transforming the result using a functional language. We demonstrate that using this approach makes it possible to i) give static guarantees regarding the preprocessing rules; ii) safely express non-trivial syntactical constructions; and iii) perform a restricted partial evaluation.

1 Introduction

Very often expressiveness of a programming language introduces a number of ambiguities in its syntax. The language specification clearly states how to resolve the conflict, however it may not be possible to formulate the resolution in terms of context free grammars. In order to illustrate that we present the following examples.

1. The classical example from C language is a type-cast syntax. As a user can define an arbitrary type using `typedef` construct, evaluation of the expression `(x) + 5` is impossible, unless we know if `x` is a type or not.

2. Assume that we extend C syntax to allow array concatenation using infix binary `++` operator and constant-arrays to be written as `[1, 2, 3]`. We immediately run into the problem to disambiguate the following expression: `a ++ [1]`, as it could mean an application of postfix `++` indexed by `1` or it could be an array concatenation of `a` and `[1]`.
3. Assuming the language allows any unary function or operator to be applied as infix and postfix, we cannot disambiguate the following expression:

   ```
   log (x) - log (y)
   ```

   Potential interpretations are: `log (uminus (log (x))) (y)`, which is obviously an error, or `minus (log (x), log (y))`.

Sometimes the context may influence not only the parsing decisions but also lexing decisions. Consider the following examples:

1. C++ allows nested templates, which means that one could write an expression `template <typename foo, list <int>>`, assuming that the `>>` closes the two groups. In order to do that, the lexer must be aware of this context, as in a standard context character sequence `>>` means shift right operation.
2. Assuming that a programmer is allowed to define her own operators, the lexer rules must be changed, in case the name of a new operator extends the existing one. For example, assume one defines an operation `+-`. It means that from now on an expression `+-5` should be lexed as (`+-`, `5`), rather than (`+`, `-`, `5`).

In order to resolve the above ambiguities using table-based parser generator, we have to make sure that one can annotate the grammar with correct choices for each shift/reduce or reduce/reduce conflict. This puts a number of requirements on the syntax of a parser generator and on the finite-state machine execution engine. Firstly, one has to introduce contexts without interfering with the above conflict-resolution. Secondly, one has to have an interface to the lexer, in case lexing becomes context-dependent, and all the mechanisms should be aware of error-recovery facilities.

Having said that, we can see that using parser generators could be of the same challenge as writing a parser manually, where all the ambiguities could be carefully resolved according to the language specification. As it turns out, most of the real-world language front-ends use hand-written recursive descent parsers that specially treat cases that cause ambiguity. For example the following languages do: C/C++/ObjectiveC in GNU GCC [2], clang in LLVM [4], JavaScript in Google V8 [3].

The main goal of any preprocessor is to perform a substitution of one element sequence with another. The unit of the sequence may be different depending on the agreement, however the common case is to say that the unit is a sequence of characters of the same class. The number of classes is normally fixed, however character belonging to the class may be static as in C preprocessor, where, for example, notion of space cannot be changed, or dynamic as in TeX, where one could specify that a certain character is a delimiter. Then the substitution itself is a replacement of units in a sequence, which are treated as arguments, with the assigned arguments. The key problem here, as we are concerned in this paper, is a lack of separation between the rewriting itself and the transformation of a token-sequence. Consider an example of a C macro:

```
#define foo(x, y) x y
```

First of all, it is really hard to say anything about the result of this macro, as `foo (5,6)` expands to `5 6`, but both `foo (,5)` and `foo (5,)` expands to `5`. Secondly, as comma is a part of syntax definition of the macro then one cannot just pass a token sequence `5, 6` as a first argument of `foo`. In order to resolve this one may escape the comma by wrapping it in parentheses and calling `foo ((5,6), 7)` which will expand to `(5, 6) 7`. The only way to flattern the list is to perform an application of another macro. For example:

```
#define first(x, y) x
#define bar(x, y) first x y
```

So we have higher-order macro here, but it would work only if arguments have a right type and the application of `bar ((5,6), x)` would expand to `5, x`, however application of `bar (5, 6)` would not provide an error but would expand to `first 5 6`. And as a last example we can make original macro `foo` return 3 arguments by expanding `foo (5, foo (6, 7))` which will expand to `5 6 7`. We may clearly see that making some static conclusions by checking a system of macros is impossible, as it may all depend on the application; and making any dynamic decisions with respect to the correctness of substitution is also not possible, as there is no way to declare the criteria of correctness.

Despite all the correctness complications macro systems are not powerful enough to introduce new language constructs. For example, it would be natural to represent a number's absolute value as `|a|`, or to allow a number of user-defined literals to introduce units in a programming language like `5kg` or `8 mm`. Even if a macro-system can do it, resolving nested expressions treating one and the same symbol differently still might be confusing. For example, would it be possible for some macro system to transform an expression `| a|b |` into `abs (a|b)`?

The proper way of doing macro-substitutions is to allow an extension to the grammar. However, providing a handle for arbitrary changes of the grammar may lead to uncontrolled changes in the semantics of the language, which again would make the proof of program corectness hard to create.

As a solution to the given problem we introduce a preprocessor which works on the sequence of pseudo-tokens which is a combination of tokens recognized by parser and productions of the grammar used by parser. To make it even more powerful we allow a regular expression on pseudo-tokens, still being able to guarantee the correctness of the transformation.

The rest of the paper is organized as follows:

FIXME: *BLA-bla-bla*

2 Parser model

The parser which serves as a basis for building a preprocessor is based on recursive descent LL(k) or LL(*) algorithm. Recursive descent is a natural human approach to writing parsers, and in case if k is small, the efficiency of parsing is linear with respect to the number of tokens on the input stream.

As a running example in this paper we are going to use a grammar of a simple language with C-like syntax described in Fig. 1.

```
program       ::=  ( function ) * ;
function      ::=  type-id  '(' arg-list ')' stmt-block ;
arg-list      ::=  ( type-id  id ) * ;
stmt-block    ::=  '{' ( expr | return ';' ) * '}' ;
expr          ::=  fun-call | assign | cond-expr ;
fun-call      ::=  id '('expr (',' expr ) * ')' ;
assign        ::=  id '=' expr ;
cond-expr     ::=  bin-expr '?' cond-expr ':' expr ;
bin-expr      ::=  bin-expr binop primary-expr
primary-expr  ::=  number | prefix-op expr | '(' expr ')' ;
binop         ::=  '&&' | '||' | '==' | '!=' ... ;
prefix-op     ::=  '-' | '+' | '!' | '~' ;
```

Fig. 1. A grammar of a C-like language.

As the preprocessor is build as an extension to the parser, it expects a certain behaviour of the parser. Further down we list a set of properties we require to be present in the implementation of the parser.

Token stream The parser should conceptually represent a stream of tokens as a doubly linked list, which allows traversing in either direction, performing a substitution of a token group with another token group, and restarting a stream from an arbitrary position. The implementation details are left to the creators of the parser.

Pseudo-tokens The parser normally reduces the grammar rule by reading tokens from the input stream. We introduce a notion of pseudo-token, which conceptually is an atomic element of the input stream, but that represents a reduced grammar rule. The implementation details are left to the parser creator. The most straight forward and inefficient way would be to convert the pseudo-token back into the token stream and parse again.

Handle-functions First of all, we ask that every production is represented as a function[4] with a signature `Parser -> (AST|Error)`, i.e. function gets a parser-object as an input and returns either an Abstract Syntax Tree node or an error. We call those functions handle-functions. We require that

[4] Note that these functions have side-effects, so the order of calling is important.

handle-function structure mimics the formulation of the grammar, i.e. if a production A depends on a production B, we require function handle-A to call function handle-B.
Each handle-function implements error recovery (if needed) and takes care of disambiguating productions according to the language specification, resolving operation priorities, syntax ambiguities and so on. Each handle function has access to the parser, which keeps record of an internal state, which changes when a handle-function is applied.

Is-functions Each handle-function is paired with a predicate function which checks whether a sequence of tokens pointed at by a parser state matches a given rule. We will call this type of functions is-functions. Application of an is-function does not modify the state of the parser. Is-functions may require unbounded look-ahead in general case, however we leave the implementation decision to the parser creator. One can always reuse matched AST nodes to perform subsequent matches.

Match-function In the beginning of each handle-function, each production calls a function called `match` with a signature `(Parser, Production) -> Parser`. A match-function is an interface to the preprocessor that checks if a stream of tokens pointed at by the parser has a valid substitution in the given production; if it does, it performs the substitution and makes sure that the parser points to the beginning of the substitition in the token stream. In case if no matches were found, the preprocessor does not perform any substitutions and returns the parser in its original state.

Assuming that all the described requirements are met, the grammar $G = (N, T, P, S)$ provides complete information required to create support for user-defined matches.

3 Dynamic extension

The preprocessing rules are defined using the following syntax:

```
match [\prod1] v = regexp   ->   [\prod2] f (v)
```

This reads as following: if at the beginning of production `prod1` a stream of pseudo-tokens pointed by parser matches a regular expression `regexp`, which can be aliased with variable named `v` in the right hand side of the match, then the matched tokens will be replaced with a reduction of `prod2` production applied on a list of pseudo-tokens that is being returned by `f (v)`. Function `f` is a function which is defined in functional language T and which is used to perform a preprocessing transformation on the list of tokens.

The `regexp` regular expression is a box standard regular expression which is defined by the grammar at Fig. **??**. In this paper we are using a minimalistic syntax for regular expression to demonstrate some basic properties. Later on, this syntax may be easily extended.

Further down in this paper we are going to use an escaped syntax for pseudo-tokens which represent grammar production names, like `\expr`, `function`, etc. The

```
regexp          ::= concat-regexp '|' regexp
concat-regexp   ::= asterisk-regexp  concat-regexp
asterisk-regexp ::= unary-regexp '*' | unary-regexp
unary-regexp    ::= pseudo-token | '(' regexp ')'
```

Fig. 2. Grammar of the regular expressions on pseudo-tokens

4 Regular expressions

FIXME: *This is a weird draft by Petch*

4.1 Match as a regular expression

The left part of the match (without the resulting type) is actually a regular expression, but with tokens to be matched instead of single characters. That does not change the concept, because just the same as we can have a getter for the next symbol in the input stream, we have the get next token function in the parser. We operate not with an input stream of symbols, but with an input stream of pseudo-tokens.

Currently our regular expression syntax allows using or `|` and asterisk `*` notations. Asterisk is more binding than or, so if you want to have (a or b) zero to n times you will have to write `(a|b)*`. The supported syntax can easily be extended. For the time matching classes of tokens is unsupported, however the classes can be emulated by using or constructions. This is possible because the token count is fairly limited, unlike the character set that can be used in regular expressions.

The given regular expression is parsed to create a non-deterministic automaton.

4.2 NFA to DFA

Next step is to create a DFA out of the created NFA to minimize the effort needed to execute the given automata. Currently the subset construction algorithm is used for the determinisation process. It is described in detail in the "Dragon Book".

Then the determinate automata created is minimised. Minimisation algorithm is, again, described in detail in the "Dragon Book". The algorithm creates sets of states that cannot be distinguished by any input token sequence. Once the algorithm fails to break the sets into smaller ones it stops the process. These sets of states then become the new states of the minimal automaton.

It is essential to note that the existence of such minimal automaton is provable, despite the complexity of the regular expression it describes. This statement implies that we have a possibility to prove equality of two automata. As the naming of the states is unimportant, we will say that two automata are the same up to state names if one can be transformed to another be simply renaming the states. Therefore two regular expressions match the same input if and only if their automata are the same up to state names.

The created DFA is minimal and therefore the most effective for matching the given expression. The DFA is represented by a list of objects where each object contains a map of symbols and objects to which the current symbol transfers the automaton.

The construction of the DFA does not support back-referencing, so the bracket groups are intended only to change the priority of the operations in the regular expressions. We abandon the back-referencing feature in favor of maintaining a fully determinate automaton for each regular expression. Consequently, in the matches' output, all of the tokens are returned in a single unnested list.

FIXME: *is the paragraph below needed?*

We still consider an option to allow backtracking in case it will become necessary. If we perform determinisation only on the parts of the automaton that are enclosed in braces and then combine the created automata by consequently connecting the accepting and starting states together, we will create an automaton that allows grouping. However this approach disallows the combining of several matcher automata into a single one, as the procedure would ruin the grouping.

4.3 Several DFA to single DFA

As we operate with the source code, we want to check all of the matches that we have in one pass through the code. We will consider two possibilities to do this, which are described below. For both of them we evaluate DFA adding, matching and context inheritance algorithmic difficulty.

Context inheritance difficulty is important, as we can have several included contexts, where the matches introduced inside the included one have a higher priority. Imagining that we have a system of `n` automata we have two options of doing so. The first one is that on entering a context we add the `m` matches we come across to the main match set and remove the `m` matches upon exiting the context. The second one is that on entering a context we create a copy of the parent context, to which we add the `m` newly found matches and upon exiting the context throw out the entire new system of matches.

We do not explicitly select a method of operation with joining DFA, we only give the complexities of the proposed variants. This is because the optimal solution selection should be based on the practical uses of the system. It is impossible to state, without any actual examples, what will be more time-effective during execution. Even though the second option is time-consuming when adding and removing matches, the dramatic improvement in execution time might come from the fact that match count is relatively small, but the automaton execution will, at worst cases, be performed for every token in the source text.

List of DFA The simplest method of executing several DFA at one pass though the source code is to store them in a list. Let us assume that there are `n` matches to be checked. Then the automata list represents an NFA with `n` epsilon branches from the start state, each of which leads to one of the DFA we already created.

Match adding

DFA adding in this case is simple, as the only action necessary is to add the automaton to the list, so the complexity is $O(1)$.

Matching

In this case matching depends both on the count of automata `n` and on the length of the matched pattern `l`, as we have `n` branches of an NFA to traverse simultaneously. So in the worst case the algorithmic difficulty of the task will be $O(n * l)$.

Context inheritance

The first option of context inheritance can be executed easily, as the added `m` matches can be removed from the system in a single block. So it's complexity is $O(1)$. The creation and deletion of the whole match system also has time complexity of $O(1)$.

Combining several DFA into single DFA In this case we combine the automata created for each of the matches together into a single DFA. This is done in order to reduce matching time.

Match adding

The algorithm combines the states of two automata one by one, and is, in fact, an adaptation of the determinisation algorithm. Its complexity is $O(m * n)$, where `m` is the amount of states in one automata, and `n` is the amount of states in the second automata.
Imagine that we have two consequent matches, for both of which we have generated an automaton. At first we create a starting state for the merged automaton by combining the starting states of existing ones. Then we examine the paths leading by same tokens from the combined states in the two starting automata. We create states in the new automata from the end points of the paths in the existing automata and add paths to them by the examined symbol. By doing so for every new state we add all of the paths and create the combined automaton. By prioritising the regular expressions we can unambiguously identify the regular expression for which a specific state will be accepting.
The figures show an example of this algorithm execution. Figure 3 shows the automaton for `a*b*`. It has two states, each of which is an accepting state. Figure 4, on the other hand, shows the automaton for regular expression `ab`. It has 3 states, the last of which is an accepting state. Smaller circles in colors of the state id's show that the state is accepting for the corresponding automata.

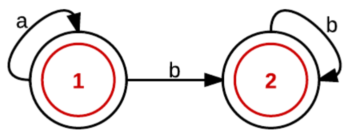

Fig. 3. Automaton for a*b*

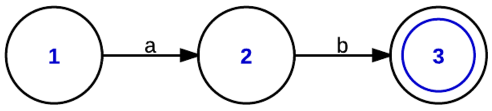

Fig. 4. Automaton for ab

The algorithm traverses the two automata step by step adding new states to the resulting one. Firstly, the state `(1, 1)` is created, which is a combination of the two automata starting states. By symbol `a` the first automata goes to state `2`, however the second one stays at state `1`. As a result state `(2, 1)` is added together with a path to it by token `a` from `(1, 1)`. Next paths are created for the newly added state `(2, 1)`. By token `b` we add state `(3, 2)` and the corresponding path and so on. The resulting automata is shown on figure 5.

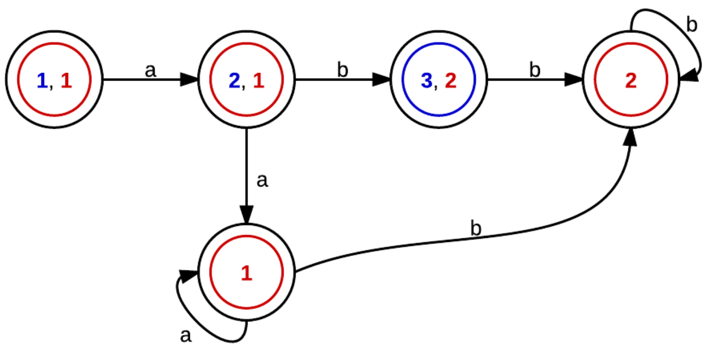

Fig. 5. Automaton for ab and a*b*

In figure 5 the states `(1, 1)`, `(2, 1)`, `(1)` and `(2)` are accepting for the regular expression `a*b*`, but the state `(3, 2)` is accepting for regular expression `ab`. Here we assume that the first regular expression to appear has lower priority than the ones that appear later. Otherwise state `(3, 2)` would also be accepting, but for the second automata.

Matching

Here matching depends only on the length of the matched token list, so the complexity is $O(l)$, as only a single automaton traverse option exists for a single input token.

Context inheritance

Context inheritance by adding and removing matches is not as efficient as in the case of automata list. Adding complexity was described above, however to remove an amtch from the system we will have to traverse all of the automata states and track those, that belong to the removed automata. So the complexity of a removing of an expression, if `m` is the amount of states of the system, will be $O(m)$.
If we adopt the second option, creation and deletion of a match system for a nested context, the complexity will remain the same as in adding DFA to the system, as the deletion will not add to it.

5 Application

5.1 Syntax extension

With the help of macro extension it is possible to extend language's syntax. Assume, we would like to have a mathematical notation for absolute value `|5|`. Our macros will look like:

```
\match [\expr] v = '|' \expr '|' -> [\num] abs-match (tail (init (v)))

abs-match :: \expr -> \expr
abs-match x = abs(x)
```

Here we define an additional production to parse `expr` rule. A matcher will return a list with matched tokens, such as `('|', ..., '|')` from which we need to remove the first and the last tokens. Then we substitute matched expression with a function call.

5.2 Templates

The macroprocessor can be used as templates in C++. Templates are used to evaluate expressions at compile-time. To illustrate this, let's consider factorial computation example.

```
\match [\number] v = \number '!' -> [\number] fact-match (v)
\match [\expr] v = \expr '!' -> [\expr] fact-match-expr (v)
```

```
match-fun fact-match :: \number -> \number
fact-match x = 1 if value (x)
      | value (x) * fact-match (\number[value (x) - 1]) otherwise

match-fun fact-match-expr :: \expr -> \expr
fact-match-expr x = factorial (x)
```

We extend here `number` and `expr` productions with mathematical notation for factorial using exclamation sign. In case of `expr` we simply call `factorial` function. In addition, we want to compute expression at compiler-time if we evaluate a constant expression. Similarly to templates, we can unroll macro-calls and evaluate constant expressions. Thus if we write `3!`, we will get `6` during compilation phase.

5.3 Preprocessing

The macroprocessor deals with language context. However, C preprocessor is working out of context. For example, we can write any expression inside `#if ... #endif` directives. In our macroprocessor we can allow to work out of parser's scope. However, in this case we have to deal with token conflicts between lexer and parser. In this way, if we define 'out of scope' macro `< ... >`, then binary operations `<` and `>` will be interpreted as parts of this directive.

6 Evaluation

Here is a bunch of links for the existing macro-preprocessors:

ML/I	http://www.ml1.org.uk/htmldoc/ml1sig.html
GEMA	http://gema.sourceforge.net/new/docs.shtml
GPP	http://files.nothingisreal.com/software/gpp/gpp.html В этой штуке советую заглянуть в ADVANCED EXAMPLES с лямбдой
TRAC	http://web.archive.org/web/20050205172849/http://tracfoundation.org/t2001tech.htm Это очень разумная идея правда совсем дохлая – там тоже функциональный язык внутри живет, но работает на строках кажись

Еще бывают: m4, cpp, lisp/scheme macros, tex?...

6.1 Macros in Lisp

Due to the Lisp's fully parenthesized Polish prefix syntax notation there is a powerful macro engine. We are going to cover its main principles and advantages.

1. There is no need to mark out a structure from a token sequence for the internal representation (IR). Every sequence of tokens in parenthesis shapes an expression called *form*. A program on Lisp represents a tree of nested forms. This property is called *homoiconicity*, as the source code of a program

can be proceeded by means of the same language. In this case it is possible to consider a macro-definition as a left tree substitution for one in the right part. For example,

```
(defmacro sum (x y z)
(list '+ (list '* x y) (list '* z z))
```

macros defines a list substitution for the nested list structure.

2. Any valid expression in Lisp represents a form. Even the whole program is a form. Therefore, it is possible to replace huge parts of a program and even the whole program.
3. Lisp's compiler does not match the whole expression (or a form) in order to find out either a corresponding macros is defined. Due to the prefix notation a 'keyword' is the first token in a form. If a macro definition is occured in the macro table under the same keyword, then we have to perform a macro substitution.
   However, this simplicity has a disadvantage too. Macros overloading is not allowed in Lisp.
4. A macro processor in Lisp not only substitutes expressions, but also evaluates them. There is an expression-value table in Lisp, where each expression is associated with it's value. For example, if we write `(setq a 3)`, then an expression `a` has a corresponding value `3`. Consequently, if we create a list `(list a (+ a 1))`, a list `(3 4)` will be returned, as this expression will be evaluated during compilation. This emphasizes an interpetive nature of the language. Although expression evaluation is not covered in the Common Lisp standart, many compilers, such as GNU CLisp, CMU Common List support this feature.

6.2 ML/1 macros

ML/1 is a stream-based macros processor[1]. It operates on a sequence of tokens. The processor reads tokens one by one and performs input stream transformation taking into account the rules defined.
In macro definition we define a correspondence between token sequence from the input and replace tokens. All atomic tokens are separated by delimiter tokens, such as 'space' or 'new line'. Suprisingly, there are no arguments placed in the macro rule to match. The arguments can appear between any atomic tokens. For example,

```
MCDEF foo bar baz AS ...
```

will match `foo xx bar yy baz`, because arguments are inserted between atomic tokens. In order to restrict such insertion, tokens have to be combined into another one, atomic token. Thus in most cases the information about exact number of arguments and their names is not accessible. Therefore, arguments are accessed by number in the order they are met in the input string. Basically,

this allows to support variable number of arguments. Here arises a problem of handling these arguments, because we do not know in advance how many arguments we have and we don't even know their types.
The following features of macro language allow to handle arguments properly:

1. Tokens placed between argument tokens are called delimiters. It is possible not only to access arguments, but also delimiter tokens, enumerating them.
2. It is possible to define 'if' condition statements. 'Jumps' or 'goto' statemens are also supported. Consequently, we can verify delimiter tokens number and its type, and perform substitutions accordingly.
3. Local variables can be used inside macros. This allows to describe loop statements for iterating over the arguments.
4. ML/1 supports nested macro calls. While searching for delimiters and arguments we can meet another macro call. In this case, we descend to a lower level and return it's delimiters and arguments. Finally, they are inserted in the 'top' list.

To sum up, ML/1 provides advanced features for macro processing. It is implemented as an imperative language operating on the stream of tokens. It supports conditions, loops, branchs and assignment, so the language is Turing-complete.

6.3 Templates

Another example of macro-processing are templates, which are widely used in C++. In contrast with macros, which performs a string substitution on preprocessing stage, templates are a language that is executed during compilation and integrated into the type system of C++.
Templates are used to operate with generic types. It allows to avoid code repetition for every specific type. We define a function using some abstract types and then abstract types are replaced with actual ones in function call. For every type a compiler generates a function, repeating the entire code for every type.
Templates can be used in terms of metaprogramming, as it allows to perform evaluations at compile-time, instead of computing at runtime. It provides sufficient features to perform all kind of evaluations. It can be proved that C++ templates are turing complete[5]. The matter is that templates support function specialisation and nested template calls. This allows to describe recursive calls which will be unrolled in compile-time. Code optimizations, such as constant folding or loop unrolling, are able to fold expressions produced by templates in order to reduce amount of runtime computations.

7 Future work

References

1. Bob Eager. The ML/I macro processor. http://www.ml1.org.uk/.
2. Free Software Foundation. gcc. http://gcc.gnu.org.

3. Google. V8 JavaScript Engine. htthttp://code.google.com/p/v8/.
4. LLVM Team. clang. http://clang.llvm.org.
5. Todd L. Veldhuizen. C++ Templates are Turing Complete. Technical report, 2003.