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]`.

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      ::=  primary_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 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.

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

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 extensions

In this section we are going to describe a syntax of the preprocessing rules and demonstrate the way to prove a correctness of the transformation.

3.1 Match Syntax

The preprocessing rules are defined using the following syntax:

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

This reads as follows: if at the beginning of production `prod1` a stream of pseudo-tokens pointed at by the 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. 2. In this paper we are using a minimalistic

```
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

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 operators of regular expression, namely (`|`, `*`, `(`, `)`) will be escaped as well like: (`\|`, `\*`, `\(`, `\)`). For simplicity reasons we assume that we do not introduce any conflicts by defining escape symbols. In case a conflict appears, we may change the escape symbol, or even the whole escaping mechanism, but it does not influence the matter.

Now we can demonstrate a simple substitution example on the language defined in Fig. 1. Assume that function `replace` is defined in T with three arguments and it replaces in any list of pseudo-tokens occurrence each of the second argument with the third, and what we need is to call a function called `bar` with an argument being a summed-up arguments of function called `foo`. In that case the following match would perform such a substitution.

```
match [\fun_call] v = foo ( \expr \( , \expr \) \* )
   -> [\fun_call] cons bar (replace (tail v) (,) (+))
```

3.2 Definition of matches

Match rules can be defined in the arbitrary places of a program and the rule activates immediately after the definition was parsed. We do however differentiate between the global matches and context matches. In our case the context is created by a `stmt_block` production. In that case, all the matches declared within the `stmt_block` production are valid only within this particular production. When the production is finished, the matches declared within the production would be removed. Declaration of the context is up to the parser, it may define it in any which way by calling two interface functions. The context definition can be omitted in which case all the matches would be global.

Both global and context matches depend on the order of definition and in both cases the first match has a stronger priority than the subsequent in case the token stream can be matched with more than one regular expression. The priority helps to cover the following case:

```
match [\expr] v = f ( \num )             ...
mathc [\expr] v = f ( \primary_expr )    ...
match [\expr] v = f ( \expr )            ...
```

Here we can see that the last regular expression includes the previous as `\primary_expr` is also an `\expr` and so on. But introducing priorities, one may still have a different behaviour in each case.

Nested context matches overload the outer matches, in a same way as local variables have a first match, in case a variable of the same name exists in the outer context.

3.3 *T* language and correctness

In order to perform a transformation of the matched list we define a minimalistic functional language called T in order to demonstrate the approach. The core definition of T is given by Fig. 3.

```
program         ::= ( function ) *
function        ::= id '::' fun_type id id * '=' expr
fun_type        ::= (type | '(' fun_type ')') '->' fun_type
expr            ::= id | expr expr | let_expr | if_expr | builtin
let_rec         ::= 'let' id '=' expr (',' id '=' expr) * 'in' expr
if_expr         ::= 'if' cond_expr 'then' expr 'else' expr
cond_expr       ::= 'type' expr '==' type | expr
builtin         ::= 'cons' expr  (expr | 'nil')
                    | 'head' expr | 'tail' expr | 'value' expr
                    | pseudo_token '[' expr ']' | number
                    | + | - | ...
type            ::= pseudotoken_regexp | int | regexp_t
```

Fig. 3. Grammar to define language T

The main use-case of the language is to traverse over the matched list of pseudo-tokens applying recursion, head, tail and cons constructs. In order to stop the recursion we also introduce arithmetic operations on integers. In order to perform partial evaluation, we need to have an interface to the value of the pseudo-token. For that reason we introduce function `value` which is applicable to the pseudo-tokens which have a constant integer value (in our example it is a `\number` pseudo-token). In order to construct an object from integer, we are using `\number[42]` syntax. The `value` function operates on integers only for the simplicity of the model only, the basic types can be extended in future.

Type system In order to prove the correctness of the match, we are going to use a specially designed type-system which is based on the regular expressions being treated as types. We are going to use a type inference to check if the function application in the right hand side of the match is allowed within a given production. In the current paper we are not going to give a definition of all the type deduction rules, however, we are going to do that in the full paper. Further in this section we are going to share the basic ideas and principles.

The main idea of the type system for T is to treat a regular expression as a type. We start with a fact that if the right-hand side of the system was called and the match succeeded then the result of the match is a flat list of pseudo-tokens. However from the regular expression we have additional information about the structure of this list. In order to perform a type inference, we observe that regular languages, hence regular expression bring a number of set operations, which is the key driving force of the inference. First of all, it is easy to define a subset relationship on two regular expressions $r_1 \sqsubseteq r_2$. As we know, we can always build a DFA for a regular expression and minimize it, which gives us a minimal possible automaton for the language recognized by a given regular expression. It means that $r_1 \sqsubseteq r_2 \Rightarrow min(det(r_1)) \sqsubseteq min(det(r_2))$. For two minimized automata A_1 and A_2, $A_1 \sqsubseteq A_2$ means that there is a mapping Ψ of A_1 states to A_2 states such that:

$$Start(A_1) \to Start(A_2) \in \Psi$$

$$\forall s \in States(A_1) \forall e \in Edges(s), \Psi(Transition(s, e)) = Transition(\Psi(s), e)$$

$States(x)$ denotes a set of all the states of automaton x, $Edges(s)$ is a set of pseudo-tokens which mark the outcoming edges of state s. Finally, $Transition(s, t)$ denotes a state which is reachable from s, using edge marked with t.

The subset relationship is going to be used to create a sub-typing hierarchy, and we also have a notion of a super-type of the hierarchy, which is represented by a regular expression `.*`, we are going to denote this type $\top$. It is obvious to see that $\forall t_i \in R, t_i \sqsubseteq \top$.

It is possible to construct a type for head and tail application by following the edges from the starting state of DFA in case of head, and creating a set of sub-automata in case of tail. Furthermore we can infer a type for recursive head/tail traversal over the list in either direction. In order to do that, one has to construct a set of all the sub-automata of a given one in case of forward traversal and the set of all sub-automata of the reversed automaton in case of backward.

In order to propagate type information in the branches obtained from the application of `type` construct, we have to know how to subtract two regular expressions, which is, again, simple. If T and S are respective DFA for subtracted types, all we have to do is to construct $C = T \times S$ and make the final states of C be the pairs where T states are final and S states are not.

The type inference procedure itself borrows the idea from SaC type system [] when we start with an abstract type `regexp_t` which is $\top$ and recursively precise the type until we get to a fixed point.

4 Regular expressions

We have devised a method of implementing match syntax that will be time effective during pre-processing. We have also created a prototype that ilustrates some of the features described in this article. This section will, in short, describe the approach we selected, for the full description, however, turn to the full paper.

As it was said, match syntax is a simple regular expression. This expression is parsed and transformed into a non-deterministic finite automaton (NFA). Next step is to create a deterministic finite automaton (DFA) out of the created NFA to minimize the effort needed to execute the match. This task is performed using the subset construction algorithm.

In order to minimise execution time even further, we minimise the created DFA. The minimisation algorithm is described in detail in the "Dragon Book"[1]. 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. These sets of states then become 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 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.

4.1 DFA grouping

As we operate with source code, we want to check all of the matches that we have in one pass through the code. We consider two possibilities to merge the created automata into a match system in order to improve effieciency. For both of them we evaluate DFA adding, matching and context inheritance algorithmic difficulty in the main article.

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.

First of the options we consider is joining the automata 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. This case is fairly simple and will not be described here extensively.

The second option is more complex. 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, the automata megring algorithm is described in the full article. The second option is more interesting, as it allows matching

of `n` independent patterns in $O(l)$ steps, where `l` is the amount of tokens to be matched. It is important to note as well that our method of joining the automata preserves the priority hierarchy of the matches.

We do not explicitly select a method of operation with joining DFA in the article, 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. 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. The actual selection remains to be made by the readers.

5 Evaluation and Conclusions

In this section we are going to consider the typical cases for an application of the designed approach and evaluate the result.

5.1 Syntax extension

One of the typical examples of using the preprocessor is to extend the syntax of the language. Assume, we would like to have a mathematical notation for absolute value, such as $|5| \equiv abs(5)$. It can be achieved using the following match:

```
match [\expr] v = | \expr |
   -> [\expr] cons (\id[abs]) (cons (head (head v)) nil)
```

Keep in mind, that we could also define the same match on a numbers and in that case we may use a partial evaluation capabilities of the system. Consider the following example:

```
match [\expr] v = | \num |  -> [\num] mabs (head (head v))
mabs :: \num -> \num
mabs x =  if value (x) > 0 then
             \num[(value x)]
          else
             \num[0 - (value x)]
```

The one thing that is left outside the scope of the paper and which is directly connected with this example – do we allow to parser `| a|b |`. Currently it is going to be an error, as there is no way to express the fact that one should use an original pseudo-token without a applying matches. It depends on the agreement of course, but currently one can overload a special case for `| \expr \| \expr \* |` to allow the latter. However, in the current model the following would work just fine: `||A| + |B||` Even the following: `|(a|b)|`.

5.2 C++ Templates

The preprocessor may be used in a similar way as template meta-progrmming technique. The only difference is that templates may be instantiated when a certain optimization inferred that something is a constant. In our setting all the information we have is the one that we can get from parser. However, we are going to consider the way to express a factorial function in compile time.

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

fact-match :: \number -> \number
fact-match x = if (value x) == 0 then
                    \number[1]
               else
                    \number[(value x)
                             * (value (fact-match
                                       \number[(value x) - 1]))]

fact-match-expr :: \expr -> \expr
fact-match-expr x = cons \id[factorial] cons \( cons x cons \) nil
```

This example demonstrates a nice advantage over the C++ templates, which is the shared syntactical form of compile-time instance and run-time. It means that in C++ if a factorial is defined as a template, like:

```
template <int n> struct fact
{
  static const int value = n * fact<n - 1>::value;
};

template <>
struct fact<0>
{
  static const int value = 1;
};
```

There is no way to call it with a non-static variable. The only way would be to create a wrapper function/macro, but in order to find out if something is a constant at compile time one has to have mechanisms similar to `__builtin_constant_p` introduced in GCC v4.7.

5.3 Preprocessing

We provide the preprocessor which operates in terms of a single production only. C preprocessor on the other hand works out of scope. For example, we can use `#if ... #endif` directives wherever in the source code, for example starting

in the middle of one production and finishing in the middle of another, which brings a lot of headache to verify. Using the preprocessor defined in this paper it is possible to create a match that would emulate C preprocessor macro-if. It may look as follows:

```
match [\p1] v = # if expr  \. \* # endif  -> [\p2] ...
```

The dot syntax is currently not allowed, but it is very well possible to emulate it using a disjunction over all the pseudo-tokens. Now, depending on the fact what is written in the right hand side of the macro, it may be accepted by the type system or not. In case the type-system can prove that the result of expression on the right-hand side of the match could be recognized by `\p2` it is allowed. However, it is easy to understand, that there is always a way to make type-system happy, i.e. generating a constant expression. It means that the macro-if could be emulated. Either it is good or bad we would leave for a discussion. In order to avoid this happening one may declare more strict type-checking rules.

References

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools, Second Edition.* Addison-Wesley, 2007.
2. Free Software Foundation. gcc. http://gcc.gnu.org.
3. Google. V8 JavaScript Engine. http://code.google.com/p/v8/.
4. LLVM Team. clang. http://clang.llvm.org.