On dynamic extensions of context-dependent parser

Julija Pecherska Artoms Šinkarovs Pavels Zaičenkovs

March 23, 2012

Abstract

1 Introduction

Most of the modern programming language syntax cannot be formulated using a context free grammar only. The problem is that rich syntax very often comes with a number of ambiguities. Consider 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, the type casting expression `(x) + 5` is undecidable, unless we know if `x` is a type or not.

2. Assume that we extend C syntax to allow an 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 to be applied as infix, postfix and standard notation, we cannot disambiguate an expression `log (x) - log (y)`, if we allow unary application of postfix minus. Potential interpretations are: `log (- log (x)) (y)` which is obviously an error, or `minus (log (x), log (y))`.

Sometimes it may be the case that context influences not only parsing decisions but also the lexing decisions. Consider the following examples:

1. C++ allows nested templates, which means that one could write an expression `template <type foo, list <int>>`, assuming that the last `>>` is two closing groups. In order to do that, the lexer must be aware of this context, as in a standard context character sequence `<<` means shift left.

2. Assuming that a programmer is allowed to define her own operators, the lexer rules must be changed, in case the name of the operator extends the existing one. For example, assume one defined 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 LALR parser generator engine, we have to make sure that one can annotate the grammar with a correct choices for each shift/reduce or reduce/reduce conflict, which puts a number of restrictions on the execution engine. Secondly, we have to implement the context support, which means that we need to have a mechanism which would not interfere with conflict-resolution. Finally, one has to have an interface to a lexer in case lexing becomes context-dependent, and it may be integrated with an error-recovery mechanism.

Having said that, we may see that using parser generators could be of the same challenge as writing a parser by hands, where all the ambiguities could be carefully resolved according to the language specification. As it turns out most of the complicated languages front-ends use hand-written recursive descendent parsers, specially treating ambiguous cases. For example the following languages do: C/C++/ObjectiveC in GNU GCC [2], clang in LLVM [], javascript in google V8 [].

2 Parser model

Our work is concerned with a dynamic grammar modification on the fly, and as a base of our approach we are going to consider an LL(k) recursive descent parser with a certain properties.

As a running example in this paper we are going to use an imaginary language with a C-like syntax. Consider a grammar of the language.

```
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 ) * ')' ;
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      ::=  '-' | '+' | '!' | '~' ;
```

First of all we ask, that every production is represented as a function with a signature `Parser -> (AST|Error)`, i.e. function gets a parser-object on input and returns either an AST node or an error. We would call those functions handle-functions. We require that handle-functions structure mimic a 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 about disambiguating productions according to the language specification, resolving operation priorities, syntax ambiguities and so on. Each handle function has an access to the parser, which keeps has an internal state, which changes when a handle-function is applied. In a some sense an application of a handle-function is a reduce step of a shift-reducer.

Each handle-function is paired with a predicate function which checks whether a sequence of tokens pointed by a parser-state matches a given rule. This type of functions we will call is-functions. Application of an is-function does not modify the state of the parser. Is-functions may require unbounded look-ahead from the parser, which also happens to be a requirement. We assume that in order to resolve complicated ambiguities unbounded look-ahead is needed anyways, as language expressions normally allow unbounded nesting.

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

3 Dynamic extension

We introduce a generic syntax extension which can be applied to any language recognized by a parser which meet all the requirements from section 2. The syntax extension is capable to perform standard preprocessing tasks providing also a functionality to do partial evaluation and non-trivial generic code transformations.

On the user level we introduce a single macro definition which is called `match` and which substitutes a sequence of tokens matched with a certain pattern with another sequence of tokens. Consider the following example:

```
match [\expr] foo ( a , b )   ->  [\expr] a + b
```

where we substitute a sequence of tokens `foo (a,b)`, which would be normally matched by an 'expr' rule of our grammar, with token-sequence `a + b` and applying 'expr' production on them. The above definition has a number of differences from the classical C preprocessor macro-definition `#define foo(a, b) a + b`:

- The above macro definition is not a function and `a b` are not arguments. The macro will match expressions where identifiers `a` and `b` are passed. In terms of tokens, only the sequence of tokens `'foo', '(', 'a', 'b', ')'` will be matched. Hence, the match would not replace expressions `foo (2, 3)` or `foo (b, a)`.
- The match is bounded to one particular production in the grammar, which is 'expr' in this example. It means that it would not perform a substitution in case one wrote `foo (a, b)` as a member of a statement block or a function header.
- The result of the substitution is always a single value, which avoids the classical situation with missing parentheses in the macro definition, i.e. if a macro-definition `#define foo(a, b) a + b` is applied to `foo (2,3) * 5`, expansion would make it `2 + 3 * 5`, where a conceptual expansion of the above match would look like `(2+3) * 5`.

It should be pointed out that in order to associate macro with some production it's necessary to provide grammar rules for a programmer. This will allow to take context into account and to interact with grammar parser dynamically.

3.1 Language patterns

FIXME: *This subsection has to be rewritten!* The depicted parser would be impractical without pattern matching. To illustrate this we would like to match expression `foo(a, b)`, which can be occurred in place of 'expr' production and `a`, `b` are allowed to be any relevant arguments.

```
match [\expr] foo ( \expr , \expr )
   -> [\expr] \expr[1] + \expr[2]
```

Let's compare this example with the previous one. We state here that we expect two token sequences in the brackets that would be interpreted as 'expr' productions. The type of production is important as this allows to perform an effective type checking. Specifically, this macro will recognize `foo (return 0, 1)` as a fallacious, unlike the C macro which will not point out any error.
It is noteworthy to mention about pitfalls of this approach. The macro extension associates user-defined rules with the grammar of the language. Therefore, these rules might conflict with existing ones and an ambiguous grammar can be produced. We state here that user has to control such situations himself, otherwise, an error of the parser will be raise.
Furthermore, we provide an interface to a lexer. For instance, it's possible to use some specific tokens in user-defined productions.

```
match [\expr] | \expr | -> [\expr] absolute_value (\expr[1])
```

Here we introduce a new `|` token which could be used for getting an absolute number value. A remarkable point is that expressions such as `|-5|` or even `||-5||`, as this macro takes lexical scope into account. Notice that `|1|2|` will produce an error as expected.
New tokens defined in the left part of the matcher are appended to a valid token table. We can use them equally well as 'native grammar' tokens. It allows to build legacy rules using new tokens:

```
match [\expr] < \expr > -> [\expr] | \expr[1] |
```

As a matter of fact only defined tokens can be used in the right part of the macro.

FIXME: *Rewritten until this very moment*
Matches work as a standard Term Rewrite System (S, R), where S is a set of terms; $S = L_t \cup P_t$, where L_t is a set of tokens recognized by lexer, and P_t is a set of pseudo-tokens which are escaped production-names of the parser. R is a set of rewrite rules, where $\forall r_i \in R \Rightarrow r_i :: \{S \cup R_t\}^n \to S^m$. R_t is a set of regular expression symbols which is allowed to formulate a rule.
Each rule has a general form of:

$$s_1 \text{ if } p(s_1) \to s_2.$$

Where s_1 is a regular expression over tokens and pseudo-tokens, $p(s_1)$ is a predicate which must evaluate to true, in order to enable match expand; s_2 is a sequence of tokens and pseudo-tokens which is used as a substitution.

The left hand side of each match allows a user to build a new production of the grammar restricted by a power of regular expressions. So, for instance it would not be possible to build a rule $(a^k b^m, k = m)$, as the new rule is recognized by DFA without any memory.

Now as a left-hand side of the match has a free form and the rewriting system is recursive by its nature, we face face a number of problems in case we want to prove correctness of the system. We have a standard word problem and stopping problem of the rewrite system. It is important to understand that in our case we are not dealing with a single TRS, but we have a mechanism to construct an arbitrary rewrite system.

Another important problem is to guarantee that the right-hand side of the match, is a valid rule in a given production of a given grammar. For example:

```
...
match [\expr] bar ( \id, \expr )      -> [\expr] \id{1} ( \expr{1} )
match [\expr] foo ( \expr , \expr )  -> [\expr] bar ( baz , \expr{1} )
match [\expr] \id ( \expr )          -> [\expr] \id{1} ( \expr{1}, \expr{1} )
...
```

Here we can see, that there is no chance to check statically whether the last rule is going to be expanded to the one of the matches or not, as it depends only on the value of `id`. Now, in case of `foo` we cannot check, if a user meant to pass a higher-order function `baz` or just a token `baz`.

It would be possible to resolve the situation at runtime, and one still would not be able to get a program that does not belong to the language generated by the grammar. However, the meaning of the program is unprovable, which practically means we have a powerful tool to obfuscate the code.

In order to resolve the situation, we want to introduce more static knowledge to the rewriting rules by introducing types. Now, we have to understand, that there is a clear distinction between the match that performs a substitution, and the helper-matches which make a transformation of the tokens matched by the left part of the match. The main reason here is that when we express a transformation of the matched token-sequence, the intermediate results we pass through helper-matches don't have to be a valid parser expressions. However we still want to type-check them. Consider the following example: `(expr + expr ...)`:

```
match [\expr] foo (\expr \(, \expr \) *)
   -> [\expr] bar replace (tail (res), \, , +)
```

This match replaces function `foo (1,2,3)` call with `bar (1+2+3)`. In order to do that we want to have a generic function that operates on the list of tokens and pseudo-tokens, which replaces every occurrence of ',' with '+'. Evaluation of 'replace' happens outside of any productions and in general case, the return type of such a type of a function could be not a valid input for any parser production.

It means that here we would like to make a clear cut between the match and the match-function. We should keep in mind, that it is always possible to express any match-function using the rewriting system of the matches, however first of all we would like to introduce the semantics and type system description for both matches and match-functions.

3.2 Type system

Obviously matches and match-functions have to share the notion of types they are operating with. What kind of types the matched left-hand side of the match can produce? If we consider that each pseudo-token represents a type, we may note that the regular expression automatically generates an algebraic data type. This is fairly easy to prove constructively:

1. Each pseudo-token generates a type.
2. Each concatenation generates a tuple.
3. Each choice generates an alternative. For instance `\id|\expr` can be represented with a type `(id|expr)`.
4. Each asterisk generates a list of types generated by a sub-asterisk expression. For example: `\( a | b \) *` can be represented as `[(a | b)]`.

Finally, in order to express a bounded recursion, one has to operate with integer and boolean types. Supporting algebraic data types comes with several built-in function in order to traverse the lists and to find the type of the variable at runtime, for being able to branch depending on the type of the expression. So we introduce the following built-in functions:

head Returns the first element of the list or `nil`

tail Returns the list without the first element.

concat Construct a list from two lists.

type Return a type of a given expression.

Finally we have to introduce the syntax of the match-functions and describe the way one can check that the type returned by an application of the match-functions is correct with respect to the grammar rule on the right hand side of the match.

3.3 Match-function syntax

The syntax of the match function can be derived from a functional language like ML, making sure that the types are properly recognized.

FIXME: *BLA-BLA-BLA*

Now let's consider an example which replaces `foo (expr, expr ...)` with `bar (expr + expr + ...)`.

```
match-fun replace :: ([expr|\op[',']], \op[')']) \op \op
                        -> ([expr|\op['+']], \op[')'])
replace lst r w = lst if len (lst) == 1
                  |
                  concat (head (lst), replace (tail (lst), r, w))
                  if (type (head (lst)) == \expr)
                  |
                  tail (lst) otherwise
```

```
match [\expr] foo ( \expr \(, \expr\) * )
   -> [\expr] bar ( replace (tail (tail (res)), \,, + )
```

4 Regexps

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 tokens, generated by the parser.

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, but it is unnecessary for demonstration purposes. Matching classes of tokens is unneeded, because the classes can be emulated by using or constructions. The token count is fairly limited (how many token types do we have?), unlike the character set that can be used in regular expressions, so symbol classes like (not a) are not essential.

The given regular expression is parsed to create an executable automaton.

4.2 Regular expression to NFA

During parsing each element of the regular expression is depicted with an object of a specific class. There are three classes with a similar interface, every class is a matcher object for a construction from the parsed expression. Token matching is a simple class, whereas asterisk and or classes are containers for other token matching sequences. The created objects are then linked in a list. These classes form an undetermined automata, where a token matching object depicts a transfer between automata states by a specific token, and all other connections are actually epsilon transfers (e.g. next element for an object or the path from asterisk object to it's contents).

4.3 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. (Should I describe the algorithm? It's pretty standard, although I couldn't find the name of it). Then the determinate automata created is minimized. (Should I describe the algorithm, again?)

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.

Currently we decided to give up on matching result grouping support in the regular expressions in favour 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. (An option to allow grouping - determinate only parts of the automaton that are enclosed in braces, then combine the created automata by consequently glueing the accepting and starting states together. This approach disallows the combining of several matcher automata into a single one, as the procedure would ruin the grouping.)

4.4 Several DFA to single DFA

Although the prototype doesn't support this feature yet, we intend on creating a single automaton from the several automata we created.

For the time being, however, the prototype executes the matches as follows. We have, for example, m matches, and the simplest of the decisions on how to match (or not) all of them at one pass through the source text is to depict them as an NFA with m epsilon branches from the start state, each of which leads to a DFA we already created. This is a simple solution and the complexity of it is $O(m)$.

FIXME: *Add text and stuff*

In due course we will combine the given m automata into 1 bigger automata. One by one as the matches are being added to the match list, the previous and the new automata are combined into a single DFA, so that it's execution complexity is $O(1)$, a single traverse option for a single input token.

Some tests should be created to understand whether the combination and determinisation is time-effective in a general case. It might take more time than actual execution of the improvised NFA.

4.5 Optimisation

FIXME: *Write*

1. Context matches can be stored for later use, although it is unlikely that an exact same match will appear later in the code.
2. Maybe it is reasonable to create a huge automata for the global matches and independent ones for each of the contexts. They can be thrown away once function is parsed.
3. For the big automata - store only a single regexp id in the accepting states, no id lists and stuff.
4. Cache automata while matches come in an uninterrupted sequence, once the sequence is disrupted, combine the cached automata into a single one and match.

5 Application

5.1 Preprocessing

5.2 Templates

5.3 Optimisation potential

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 shape 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) (+ (* x y) (* 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.

7 ML/I macros

ML/I macros processor[1] is going to be covered in this section. Basically our dynamic parser described here and ML/I share the idea of supporting the regular expression functionality. This could be a very powerful instrument to support almost any expression substitution. However, there is a bunch of problems. Let's have a close look on ML/I macros processor and see how the problems are handled there.
ML/I can be regarded as a processor which operates with strings. In many cases it behaves like a macro-processor in C. If we define macro `#define foo bar` in C, it will substitute `xfoo foo yfooz foox` string for `xfoo bar yfooz foox`. It follows that delimeter characters as whitespaces and 'new line' are taken into account. In ML/I the same macro will look like

```
MCDEF foo AS bar
```

This leads to the fact that even in the simplest case characters can't be interpreted in the same way.
ML/I would be a poor macro-processor if it didn't support any argument passing. The argument passing is tightly connected with delimiter characters in ML/I. Arguments can be occured in any place between non-delimiter characters. An equivalent expression for

```
#define unstack x \
{   \
  x = stack[ptr]; \
  ptr = ptr - 1;  \
}
```

will be[3]

```
MCDEF UNSTACK
AS <%A1. = STACK[PTR];
PTR = PTR - 1;>
```

Access to arguments is done by writing `\%An.` where n is the number of the argument passed. We access to the arguments in the dynamic parser in the same way, by enumerating arguments. Problems arise when we would like to support variable number of arguments and more complex match expressions. ...Some text here...

8 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] P.J. Brown R.D. Eager. ML/I. http://www.ml1.org.uk/htmldoc/ml1sig.html.