On dynamic extensions of context-dependent parser

Julija Pecherska Artoms Šinkarovs Pavels Zaičenkovs

March 17, 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 [1], 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

The depicted parser would be impractical without pattern matching. To illustrate this we would like to match expression `foo(a, b)`, which can be occured 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 fallaicious, 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: *Speak about rewriting systems*

FIXME: *Speak about functions*

FIXME: *Show that match-functions form a λ calculus*

4 Application

4.1 Preprocessing

4.2 Templates

4.3 Optimisation potential

5 Evaluation

6 Future work

References

[1] Free Software Foundation. GCC. http://gcc.gnu.org.