On dynamic extensions of context-dependent parser

Julija Pecherska Artoms Šinkarovs Pavels Zaičenkovs

March 8, 2012

Abstract

1 Introduction

Most of the modern programming language syntax cannot be formulated using a context free grammar. The problem is that rich and comfortable syntax brings a lot 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 that `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 on the parsing decisions, but on the lexing decisions either. 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 ambiguities mentioned witin the classical grammar formulation assuming its execution on a hift-reduce machine, we have to make sure that we can annotate the grammar with a correct solution 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 for that which would not interfear 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 a tool that generates an execution engine from the grammar specification 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 we are looking for a way to allow user change a certain parser behaviour on the fly, we would like to formulate our approach not in terms of standard grammars, but in terms of a parser, which implements a certain language and meet a certain requirements.

We are going to formulate those requirements starting with an example of context-free grammar of an imaginary language with a C-like syntax.

```
program       ::=  ( function ) * ;
function      ::=  type-id  '(' arg-list ')' stmt-block ;
arg-list      ::=  ( type-id  id ) * ;
stmt-block    ::=  '{' ( expr ';' ) * '}' ;
expr          ::=  fun-call | assign | return | 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 poduction B, we require funtion 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.

2 Parser model

3 Dynamic extension

4 Application

4.1 Preprocessing

4.2 Templates

4.3 Optimisation potential

5 Evaluation

6 Future work