Towards Dynamically Extensible Syntax

Tijs van der Storm

Centrum Wiskunde & Informatica
Science Park 123
1098 XG Amsterdam

Abstract. Domain specific language embedding requires either a very flexible host language to approximate the desired level of abstraction, – or elaborate tool support for "compiling away" embedded notation. The former confines the language designer to reinterpreting a given syntax. The latter prohibits runtime analysis, interpretation and transformation of the embedded language. I tentatively present CherryLisp: a Lisp dialect with dynamically user-definable syntax that suffers from neither of these drawbacks.

Jan Heering often speaks fondly of Lisp and much of his research has been dedicated to programming environments, syntax, semantics, and domain specific languages. On the occasion of his retirement I would like to honour him and his work with this extended abstract which touches upon some of these subjects.

1 Introduction

The project of defining M-expressions precisely and compiling them or at least translating them into S-expressions was neither finalized nor explicitly abandoned. It just receded into the indefinite future, ...
John McCarthy [14].

Domain specific languages (DSLs) are a powerful tool to increase the level of abstraction in programming [15]. Using a notation closer to the problem domain increases productivity, improves communication with stakeholders, reduces code size and allows for domain specific analysis and optimization. Nevertheless, developing a DSL requires a considerable investment up-front.

A more approach to DSL engineering is embedding domain specific notation in a (general purpose) host language. There are basically two approaches: "bending" the syntax of the host language to reach the desired level of abstraction, or compile-time preprocessing to assimilate the domain specific notation into the host language by transformation. A drawback of the former is that a DSL designer is confined to the syntactic (and semantic) freedom provided by the host language. The latter suffers from the fact that domain specific notation is "compiled away": the embedded notation is inaccessible at runtime, which prohibits dynamic interpretation, analysis, debugging, pretty printing, optimization and disambiguation of embedded DSLs. So we find ourselves in a quandary: restricted syntax with runtime access, or rich syntax without.

Can we combine the best of both embedding worlds? This extended abstract presents some evidence that the answer is affirmative. I present the prototype implementation of a monolingual programming environment [9,11], called CherryLisp, which features arbitrary dynamically user-defined syntax. Embedded syntax is accessible at runtime, so that language related tooling – interpreters, typecheckers, debuggers, formatters, optimizers etc. [12] – can be developed from within. Key features of CherryLisp include:

1. **Dynamic syntax extension** the syntax of CherryLisp can be extended at runtime. The abstract syntax trees (ASTs) that are the result of parsing embedded syntax are available at runtime for analysis, assimilation, translation, interpretation etc.
2. **Immediate extension** syntax extensions can be immediately used in the same file that contains the very expressions to extend the language. CherryLisp's grammar can be modified on the fly during a single parse.
3. **Arbitrary context-free grammars** CherryLisp's parser is based on generalized parsing techniques to allow the use of arbitrary context-free grammars for embedded notation. Because of 1, should ambiguity arise, it can be dealt with from within CherryLisp.

Related work There is ample related work on syntax extension. Almost all related work employs some form of static assimilation of embedded notation. Below I briefly discuss some relevant references.

The Lithe language features completely user defined syntax [18]. Inspired by Smalltalk-72 it combines a object-oriented class model with syntax directed translation (SDT) to attach actions to grammar rules ("methods"). A (parsed) string thus leads to a sequence of rule applications. It is unclear which grammar class is allowed, but ambiguities are explicitly disallowed. ASTs are not first-class and non-existent at runtime.

Cardelli et al. [6] presented incremental syntax extensions which respect the scoping rules and type system of the host language. The syntax extensions are compiled away to the host language. Their implementation is based on an LL(1) parser so there are restrictions as to what kind of extensions are possible. An approach that does not suffer from such restrictions is described in [5]. This work is based on the syntax definition formalism SDF [10,22] which allows arbitrary context-free grammars. Embedded languages are assimilated into the host language in a compile-time preprocessing phase. Since SDF supports arbitrary context-free grammars, there is a risk of ambiguity; ambiguities must be resolved before assimilation.

The MS^2 programmable syntax macro system [25] provides the macro writer with full C to define their semantics. Macro expansion has to be completed before running the program: "none of it exists at runtime". Metamorphic syntax macros are another powerful tool to extend a host language syntax [4]. However, "a syntax macro must ultimately produce host syntax and thus cannot return user defined ASTs" hence the ASTs are not available at runtime. Finally, OMeta [24] is fully dynamic language extension tool, allowing lexically scoped syntax extensions. It is based on Packrat parsing. This has the advantage that no ambiguities are possible, but this also means it does not support arbitrary context-free grammars. For instance, no left-recursive rules can be used (see however, [23]). Additionally, there is no access to parse-trees, since it is recognition based (in essence similar to Lithe and inspired by Meta II [19]).

In fact Common Lisp's [20] reader macros provide almost the functionality we are looking for: by hooking into the reader programmers can seize the opportunity of parsing any embedded syntax (see [17] for an example). CherryLisp differs from reader macros in that it uses context-free grammars to define syntax and a dedicated parsing algorithm producing uniform ASTs. In other words CherryLisp provides *declarative* reader macros. The result can be interpreted, translated, analyzed etc. using powerful macro facilities just like in Common Lisp [21].

2 Dynamic Syntax Extension in CherryLisp

CherryLisp is a Scheme dialect featuring runtime, grammar-based syntax extension. Below I illustrate a single special form used for syntax extension. Cursory knowledge of Lisp is assumed.

To allow arbitrary user-defined syntax, CherryLisp features the following special form:

(extend-syntax $[$(l s ($x_1 ... x_n$))$]^*$)

This special form is used for dynamically extending the current grammar with context-free productions $s ::= x_1 ... x_n$. The syntax extension will be immediately available. Resulting AST nodes will be labeled l. To illustrate how this special form works, let's extend the syntax with the notation for absolute values $|x|$ by entering the following `extend-syntax` invocation in the read-eval-print-loop (REPL[1]):

```
> (extend-syntax (abs Form ("|" Form "|")))
()
```

Since $|x|$ should be a normal expression, we extend the non-terminal for CherryLisp expressions (*Form*). After evaluation, the notation is immediately available:

```
> |1|
Unbound variable: abs
```

[1] The `>` indicates the prompt, the last line is the evaluation result.

A snippet of syntax like this will be converted to internal S-expressions by the parser. After parsing, the REPL attempts to evaluate this expression. Since the AST label for absolute value notation is "abs" the evaluator attempts to call this function which results in an unbound variable error. To see what the parser returns, we have to quote the snippet:

```
> '|x|
(abs 1)
```

To give semantics to this construct, ordinary functions or macros can be used:

```
> (define (abs x) (if (>= x 0) x (- 0 x)))
abs
> |1|
1
> |-1|
1
```

In this a case a simple function suffices, but the real power lies in arbitrary combinations of functions and macros. For instance, to give a more involved example, let's try and self-apply syntax extension by adding syntax for context-free productions so that grammars can be written in more natural form:

```
> (extend-syntax
    (sort Element (Symbol))
    (lit  Element (String))
    (star Element (Symbol "*"))
    (plus Element (Symbol "+"))
    (prod Form (Symbol ":" Symbol "::=" (star Element) ";")))
()
```

Note the use of ordinary S-expressions for complex grammar symbols in last production (`(star Element)`). Evaluating this form at the command-line will add context-free productions defining the syntax of context-free productions itself to the current grammar of CherryLisp. We now can write productions as follows:

```
> 'abs:Form ::= "|" Form "|";
(prod "abs" "Form" ((sort "Form")))
```

This new syntax makes it a lot more easy to define extension grammars. It is not readily possible to give semantics to these productions using just functions because of eager argument evaluation. For example, we could use a macro to interpret `prod` AST forms, as follows:

```
> (define prod (macro (label sym elts)
    `'(,(str2sym label) ,(str2sym sym) ,elts)))
prod
> abs:Form ::= "|" Form "|";
(abs Form ((sort "Form")))
```

Macros in CherryLisp are similar to ordinary `lambda`s but with unevaluated arguments. The quasi-quotation (`` ` ``) returns its argument unevaluated except where anti-quotes (`,`) are encountered [1]. So this macro returns a list with the label and symbol strings converted to symbols. Note how the result of entering a syntax production is evaluated on the fly via the `prod` macro, returning an S-expression conforming to the rule format accepted by `extend-syntax` (with exception of the , as of yet, unevaluated `(sort "Form")` construct). In the following I will assume there is a macro `syntax` that completely interprets production ASTs and converts them `extend-syntax` invocations. So the following expression `(syntax tuple:Form ::= "|" Form "|";)` will effectively add absolute value notation to the language.

3 Embedding a Programming Language: Pico

3.1 Syntax

Pico [2] is a small, While-like language, featuring assignment, if-then-else, while-do and skip statements. Supported expressions are literals (strings, naturals), variables, addition, subtraction and string concatenation. Using the `syntax` macro from Section 2 the syntax of CherryLisp is extended with the syntax of Pico (an excerpt of which is listed in Figure 1).

```
program:Program ::= "begin" Declare  Series "end";
declare:Declare ::= "declare" Decls ";";
decl:Decls      ::= Decl;
decls:Decls     ::= Decl ";" Decls;
idtype:Decl     ::= Symbol ":" Type;
assign:Stat     ::= Symbol ":=" Exp;
skip:Stat       ::= "skip";
if:Stat         ::= "if" Exp "then" Series "else" Series "fi";
while:Stat      ::= "while" Exp "do" Series "od";
add:Exp         ::= Exp "+" Term;
sub:Exp         ::= Exp "-" Term;
term:Exp        ::= Term;
cat:Term        ::= Term "||" Factor;
fact:Term       ::= Factor;
bracket:Factor  ::= "(" Exp ")";
```

Fig. 1. Excerpt of a (context-free) grammar for Pico

To be able to embed Pico programs in CherryLisp programs, we have to inject Pico sorts into the primary sort of CherryLisp. This is where the `form` macro comes in. For instance, evaluating `(form Stat)` adds the production *Form* ::= "[" *Stat* "]" to the syntax. Recall that *Form* is the non-terminal capturing

CherryLisp expressions. This means that Pico statements enclosed in square brackets are valid CherryLisp expressions:

```
> '[x := 3]
(form (stat (assign "x" (term (factor (int "3"))))))
```

The use of the `form` macro only allows ground Pico terms to be embedded in CherryLisp, however, to allow for patterns with holes, another syntax extension is required. One could, for instance, define a macro (`var` *str* *S*) for declaring pattern variables, which, when evaluated, extends the grammar with the following productions[2]:

```
var:S     ::= S-Var;
var:S-Var ::= ":" str;
var:S-Var ::= ":" str Number;
```

These productions allow Pico syntax patterns to contain variables. For example, after having evaluated `(var "exp" Exp)`, we can evaluate:

```
> '[x := :exp]
(form (stat (assign "x" (var ":exp"))))
```

Of course, such pattern variables are of little use if these patterns cannot be used to match Pico source code. Since embedded syntax is converted to ASTs by the parser, matching can be programmed in plain CherryLisp. Such a match function is illustrated as follows:

```
> (match '[x := x + 1] '[x := :exp])
((":exp" add (term (factor (id "x"))) (factor (int "1"))))
```

The `match` function traverses the structure of both its arguments, and compares each pair of subtrees it visits. If a variable is encountered in the pattern, the corresponding subtree of the first argument is bound to it in an environment structure. The result of matching a term against a pattern is the set of bindings created during matching, in this case the binding "*:exp*" to the AST for "$x+1$". Note that since patterns and pattern variables are defined using ordinary macros, nothing keeps the user from using different, embedded-language specific quotes (i.e. other than "[...]" and ":").

3.2 Semantics

Using patterns to embed Pico source code and pattern variables in matching Pico terms against such patterns allows us to define an interpreter for Pico in a style similar to the style employed in [2, 3]. A function to evaluate Pico statements is shown below:

[2] The latter two are actually lexical rules that do not contribute to the AST; for the sake of exposition, however, I choose to gloss over this detail.

```
(define (ev-stat stat env)
  (switch stat
    ([skip] env)
    ([:sym := :exp] (bind sym (ev-exp exp env) env))
    ([if :exp then :series1 else :series2 fi]
     (if (ev-exp exp env)
        (ev-series series1 env)
        (ev-series series2 env)))
    ([while :exp do :series od]
     (if (ev-exp exp env)
       (ev-stat stat (ev-series series env))
       env))))
```

The function `ev-stat` evaluates a statement; it assumes two auxiliary functions: `ev-series` for evaluating sequences of statements and `ev-exp` for evaluating expressions. The top-level expression of the function is a `switch` construct. This macro attempts to match sequences of patterns to its first argument. If a match is found, the captured variables are made into normal CherryLisp bindings so that code right of the pattern can refer to them. For instance, the pattern for assignment contains a variable *:sym* which matches a Pico variable. In the right-hand side of this case the captured variable is available as `sym`.

4 Implementation: GLR & IPG

The current version of CherryLisp is implemented in Ruby[3]. There are three important parts: the parser, de parser generator (the "grammar") and the interpreter. The parser algorithm is the Generalized LR (GLR) algorithm as described in [16]. The LR(0) parsetable generation algorithm is both incremental and lazy (IPG) [13, 16]. I have instantiated both the parsing and parser generation algorithms in scannerless style; this obviates the need for a separate lexical analysis phase[4].

The basic configuration of the CherryLisp interpreter proceeds as follows:

```
g = IPGGrammar.new("boot.grammar")
p = GLRParser.new(g)
e = Evaluator.new(g)
p.add_reduction_observer(e)
```

First the IPG parser generator is initialized with the initial grammar containing the syntax of the host language, – in this case a small syntax for S-expressions suffices. Then both the parser and evaluator are initialized with the parser generator. The GLR algorithm will use g to parse text and produce AST. The interpreter has to know about the same g in order to add productions to it.

[3] http://www.ruby-lang.org

[4] ... but might require some provision to enforce longest match for certain lexical sorts, such as identifiers; again, I choose to gloss over this technical detail.

Finally, following the Observer design pattern, the interpreter is added as an observer of the GLR algorithm. In this case it means that the interpreter will be notified of reductions performed by the GLR algorithm.

Figure 2 illustrates some of the internals of the interpreter. Evaluators are initialized (through `initialize`) with a grammar and a fresh environment[5]. The next method, `notify_reduction` is called during runs of the GLR parsing algorithm. The argument is the AST created in a reduce action. If this AST is of the `extend-syntax` kind, the grammar is immediately modified.

```
def initialize(grammar)
  @grammar = grammar
  @env = Environment.new
end

def notify_reduction(form)
  if node.first == "extend-syntax"
    @grammar.extend(node.rest)
  end
end
```

```
def eval(form, env)
  ...
  case form.first
   when "extend-syntax"
     return nil
    ...
  end
end
```

Fig. 2. Simplified method skeleton of the CherryLisp interpreter.

Finally, the main content of the interpreter is in the `eval` method. It consists of a large **case** statement that dispatches on the incoming expression (*form*). For the sake of brevity, only the cases relevant to syntax extension are shown. If the interpeter encounters an `extend-syntax` form it just returns **nil**, since the arguments to it have already been evaluated *during* parsing (in `notify_reduction`), and from the fact that *form* is a parameter to `eval` it naturally follows that parsing (with respect to *form*) has already finished.

5 Future Work

Understanding First of all, further research should concentrate at deeper understanding of the consequences of modifying LR parsetables *during* parsing. The experiments with the current prototype of CherryLisp are encouraging (however naive the implementation may be). Nevertheless, it is unclear whether there are circumstances where the current scheme would not work. For instance, currently, there is control over what would happen if, in the case of ambiguity, the parsetable is modified "concurrently" in different, possibly interacting, ways (!). Similarly, there is no rollback for undoing parsetable modifications in the case that a certain path exercised by the GLR algorithm does *not* eventually lead to a valid parse. In other words, currently, parsetable modifications do not obey the stack discipline of the GLR algorithm and ignore the its quasi concurrency.

[5] In Ruby, instance variables start with an "@".

A purely function implementation of GLR using a persistent datastructure for the parsetable could be a viable approach to solve these problems.

Bootstrapping Lisps have a very simple syntax: CherryLisp therefore begs to be bootstrapped. In fact, I think that the following five (context-free) productions are should be enough to bootstrap CherryLisp to having a full-fledged Lisp syntax:

```
nil:Form    ::= "(" ")"                  num:Form ::= Number
pair:Form   ::= "( Prefix "." Form ")"   sym:Form ::= Symbol
form:Prefix ::= Form
```

These rules suffice for Lisp expressions consisting of dotted pairs (e.g. `(1 . 2)`), number literals and symbols. Using the syntax extension mechanism of CherryLisp any other syntax can be added: list syntax, string literals, quotation syntax, etc.

Syntax restrictions Currently, CherryLisp only allows the current syntax to be extended, but not restricted. It can be useful however, to sometimes undo a syntax extension: this will also help to minimize possible ambiguities arising from the interaction among embedded languages and the host language. The incremental parser generation algorithm IPG already caters for the removal of productions from a grammar.

Lexically scoped syntax extensions This area of future work is closely related to the previous point. Similar to the approach of [6] one would like to restrict the grammar for a certain region of source code without having to introduce extra non-terminals. Lexically scoped syntax extensions can be implemented by performing extension and restriction using a stack discipline. However, there are techniques to (temporarily) restrict a grammar without adding and/or removing productions ([16], Chapter 3).

I have performed some experiments based on adding and subsequently removing productions from a grammar the results of which are somewhat encouraging. Scoped grammar modification would cater for idioms like:

```
(with-syntax (abs:Form ::= "|" Form "|";) |x - y|)
```

This would effectively set up a kind of *syntax jail*: the absolute value notation is only available within the confines of the `with-syntax` construct. Additionally, one could imagine syntax jails where a completely different grammar is active and not just a certain *extended* grammar (as in the example). Further research is required however to understand the possible effects of destructively (i.e. non-conservatively) modifying the parse table during parsing.

First class grammars To make the previous construct even more powerful the current grammar (or really any kind of grammar) should be a first-class value in the runtime system. Grammars could then be assigned to variables just like any other value. In combination with lexically scoped syntax regions, this would allow idoms like: `(with-syntax abs-syntax |-1|)`, where the variable `abs-syntax` holds the productions defining the absolute value notation.

Additionally, first class grammars would enable grammars to be processed within CherryLisp itself. In combination with Lisp's powerful macro facilities, one could generate default source code formatters [8] and AST manipulation programming interfaces [7].

First class parse trees The current focus of CherryLisp is syntax extension for DSL embedding. In this case, abstract syntax is often sufficient for analysis, interpretation and transformation purposes. However, for reverse- and reengineering purposes concrete syntax trees are needed. Such trees could readily be made available in CherryLisp since the GLR parsing algorithm, in fact, already produces them[6]. Concrete syntax trees preserve information on layout, literals, and comments, – essential information, for instance, when implementing automatic refactorings or source level debugging facilities.

6 Conclusion

I have described a simple, yet very powerful approach to dynamic, incremental and immediate syntax extension. Embedded syntax is transformed by the parser into S-expressions which are readily available for analysis, evaluation, expansion or any kind of processing one could conceive of in a Lisp-like language. The current prototype is admittedly naive, and further work should focus on deeper understanding of the dynamics of syntax extension.

References

1. A. Bawden. Quasiquotation in lisp. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 4–12, 1999.
2. J. A. Bergstra, J. Heering, and P. Klint. Algebraic definition of a simple programming language. CS R8504, Centrum voor Wiskunde en Informatica (CWI), February 1985.
3. J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press/Addison-Wesley, 1989.
4. C. Brabrand and M. I. Schwartzbach. Growing languages with metamorphic syntax macros. In *PEPM '02: Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 31–40, New York, NY, USA, 2002. ACM.
5. M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2004. ACM.
6. L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. SRC RR-121, DEC, 1994.

[6] To be completely correct, it produces shared packed parse forests; see [16].

7. H. A. de Jong and P. A. Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 59(1–2):35–61, April-May 2004.
8. M. de Jonge. Pretty-printing for software reengineering. In *ICSM '02: Proceedings of the International Conference on Software Maintenance*, page 550, Washington, DC, USA, 2002. IEEE Computer Society.
9. J. Heering. Eentalige programmeeromgevingen. CS N8503, Centrum voor Wiskunde en Informatica (CWI), April 1985. In Dutch.
10. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF: reference manual. *SIGPLAN Not.*, 24(11):43–75, 1989.
11. J. Heering and P. Klint. Towards monolingual programming environments. *ACM Trans. Program. Lang. Syst.*, 7(2):183–213, 1985.
12. J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM Sigplan Notices*, 35(3):39–48, Mar. 2000.
13. J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1350, 1990.
14. J. McCarthy. History of LISP. In *History of programming languages I*, pages 173–185. ACM, New York, NY, USA, 1981.
15. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.
16. J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.
17. A. Repenning and A. Ioannidou. X-expressions in XMLisp: S-expressions and extensible markup language unite. In *Proceedings of the ACM SIGPLAN International Lisp Conference (ILC 2007)*. ACM Press, 2007.
18. D. Sandberg. Lithe: a language combining a flexible syntax and classes. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 142–145, New York, NY, USA, 1982. ACM.
19. D. V. Schorre. META II a syntax-oriented compiler writing language. In *Proceedings of the 1964 19th ACM national conference*, pages 41.301–41.3011, New York, NY, USA, 1964. ACM.
20. G. L. Steele. *Common Lisp the Language*. Digital Press, 2nd edition, 1990.
21. G. L. Steele. RE: macros vs. blocks. Mailing list: `ll1-discuss@ai.mit.edu`, November 2002. Online: `http://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg02088.html`.
22. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.
23. A. Warth, J. R. Douglass, and T. Millstein. Packrat parsers can support left recursion. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 103–110, New York, NY, USA, 2008. ACM.
24. A. Warth and I. Piumarta. OMeta: an object-oriented language for pattern matching. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 11–19, New York, NY, USA, 2007. ACM.
25. D. Weise and R. Crew. Programmable syntax macros. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 156–165, New York, NY, USA, 1993. ACM.