ACM LISP Pointers IV,2 (April-June 1991),3-15.

Pragmatic Parsing in Common Lisp

Henry G. Baker

Nimble Computer Corporation
16231 Meadow Ridge Way
Encino, CA 91436
USA
(818) 501-4956 (818) 986-1360 (FAX)

January, 1991

This work was supported in part by the U.S. Department of Energy Contract No. DE-AC03-88ER80663

We review META, a classic technique for building recursive descent parsers, that is both simple and efficient. While META does not handle all possible regular or context-free grammars, it handles a surprisingly large fraction of the grammars encountered by Lisp programmers. We show how META can be used to parse streams, strings and lists—including Common Lisp's hairy lambda expression parameter lists. Finally, we compare the execution time of this parsing method to the built-in methods of Common Lisp.

A. INTRODUCTION

Lisp has traditionally been a language that eschews complex syntax. According to John McCarthy, the inventor of Lisp:

> This internal representation of symbolic information gives up the familiar infix notations in favor of a notation that simplifies the task of programming the *substantive* computations, e.g., logical deduction or algebraic simplification, differentiation or integration. If customary notations are to be used externally, translation programs must be written. Thus LISP programs use a prefix notation for algebraic expressions, because they usually must determine the main connective before deciding what to do next. In this, LISP differs from almost every other symbolic computation system. ... This feature probably accounts for LISP's success in competition with these languages, especially when large programs have to be written. *The advantage is like that of binary computers over decimal—but larger.*
>
> ... Another reason for the initial acceptance of awkwardnesses in the internal form of LISP is that we still expected to switch to writing programs as M-expressions [infix format]. *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*, and a new generation of programmers appeared who preferred internal notation to any FORTRAN-like or ALGOL-like notation that could be devised.
>
> ... One can even conjecture that LISP owes its survival specifically to the fact that its programs are lists, which everyone, including me, has regarded as a disadvantage. *Proposed replacements for LISP ... abandoned this feature in favor of an Algol-like syntax, leaving no target language for higher level systems*. [McCarthy78], with emphasis added.

Accordingly, Lisp users and developers have usually had the luxury of dealing with more "substantive" computations, so that they are often at a loss when they face a *parsing* task. No matter how hard they try to write clean, efficient code, the Lisp language doesn't seem to provide them with the right linguistic constructs to make an elegant program.

The programs required to implement a Common Lisp system itself provide some good examples where parsing techniques are required. Parsing the rather hairy parameter lists of Common Lisp lambda-expressions [Steele90,5.2.2] has caused many good programmers to tear their hair out, and ditto for the syntax of many built-in Common Lisp macros (e.g., `defclass` and `defmethod`).[1]

[1]See, e.g., the definition of *Closette* [des Rivières90]. The inability of `defmacro` to easily parse the syntax of the more complex Common Lisp macros exhibits a significant weakness in Common Lisp. The technique reviewed here can handle complex macros such as `defmethod`.

The parsing of numbers and "potential numbers" in the Lisp reader [Steele90,22.1.2] requires extensive syntax-hacking. `format` control strings [Steele90,22.3.3] require parsing at run-time (or perhaps at compile time [Steele90,27.5], if `format` uses a compiler optimizer). Finally, many networking protocols require extensive (and often excessive) syntax analysis; slow protocol parsing is typically the chief cause of poor network performance.

Prolog programmers, however, find parsing tasks trivial,[2] and the ease of programming parsing tasks has been one of the selling points of Prolog to unsophisticated programmers.

Lisp has a few tricks up its sleeve, however. Lisp is a language-building language *par excellence*, and it can therefore easily emulate those Prolog capabilities that make parsing simple. Furthermore, since our emulation will include only those features we need, we will be able to parse much more quickly than a Prolog system. Finally, the technique does not require the splitting of the parsing task into separate lexical and syntactic analyses, futher simplifying the parsers.

B. REGULAR EXPRESSIONS

Every computer scientist knows about *regular expressions*, which describe *regular languages*, and the ability of deterministic and non-deterministic *finite state machines* to recognize these languages. Regular expressions over an *alphabet* consist of the letters of that alphabet ("symbols"), together with a number of operations: concatenation, union and Kleene star. *Concatenation* describes how the letters can follow one another to make strings, and *union* allows one to have different alternative expressions that are equivalent. Finally, Kleene *star* allows for "zero or more" concatenated occurrences of a particular expression to be another expression. Regular expressions can be a very compact and moderately readable description of a regular language, and they have become a standard as a result.

Finite state machines consist of an *alphabet*, a number of *states*, one of which is designated as an "initial" or "starting" state, and some of which are "final" or "accepting" states, and a *relation* which maps a combination of a state and an alphabet letter into a "next" state. If the relation is an algebraic *function*, then the finite state machine is *deterministic*, otherwise it is *non-deterministic*. Given any finite state machine, it can be algorithmically converted into a deterministic finite state machine by simulating sets of states starting from the singleton set consisting of the initial state, and tracing out all state combinations, which are necessarily finite. There may be an exponential blowup in the number of states, however.

Deterministic finite state machines make excellent parsers because they can be implemented very efficiently on serial computers using table-lookup, and their speed is therefore independent of the complexity of the next-state function. Unfortunately, the number of states—and hence the size of the next-state table—is usually quite large for relatively simple languages, and even if it is not, the programming of these tables is extremely tedious and error-prone. Thus, the computation of finite state machines is an excellent job for a compiler.

The mapping of regular expressions onto non-deterministic finite state machines is trivial; the harder part is the conversion to deterministic form, which can blow up exponentially.[3] Deterministic conversion has a number of drawbacks, however. The deterministic finite state machine may bear little resemblence to the original regular expression, and more importantly, the conversion to deterministic form will not generalize to context free languages, which we will tackle in the next section.

We would therefore like to investigate a scheme which keeps the original structure of the regular expression, and also has most of the efficiency of a deterministic finite state machine. This cannot be done in general, but it can be done for most regular expressions that are encountered in practise. It might be asked why one would consider a less powerful and potentially less efficient method, when computer science has already given us a universal and efficient method—deterministic finite

[2]Perhaps too trivial, as some Prolog programmers turn simple grammers into parsers with exponential behavior.

[3]The sheer size of these state tables may kill the advantage of instruction and/or data caches.

state machines. The reason is that we are usually interested in non-regular languages—particularly context free languages—where this particular sledgehammer fails.

1. META Parsing of Common Lisp Streams

The scheme we will describe has been used for at least 27 years, but is relatively unknown today, because computer science departments are rightly more interested in teaching theoretically pretty models like regular and context-free languages, rather than practical methods that may also be elegant and efficient.[4] The scheme involves building a tiny language on top of Lisp which compiles to extremely efficient code. The tiny language, called META [Schorre64], incorporates the basic operations of regular expressions, and the code is therefore quite perspicuous.

Let us see the META code for recognizing signed integers.

```
(deftype digit () '(member #\0 #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9))
```
[5]

```
(defun parse-integer (&aux d)
  (matchit [{#\+ #\- []} @(digit d) $@(digit d)]))
```

We first define a new Common Lisp type which is a subset of the characters that includes just the digits. We then define our parser for integers as a function with a temporary variable and a body which is a call to the macro `matchit`. `matchit` has an argument which is a META expression which it compiles into Common Lisp when it is expanded. The META expression includes two new kinds of "parentheses": brackets "[]" and braces "{}", as well as operators like "$" and "@". The brackets "[]" enclose *sequences*, while the braces "{}" enclose *alternatives*; the use of these additional "parentheses" eliminates the need for prefix or infix operations.[6]

The META operators "[]", "{}", and "$" provide the sequence, union and Kleene star operations of regular expressions, so most regular expressions can be converted into META expressions by inspection. Letters stand for themselves in normal Common Lisp syntax, e.g., `#\+`, and @ allows the matching of a number of character possibilities with a single operation. The use of `@(digit d)` in `parse-integer` could logically have been replaced by the expression

```
{#\0 #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9},
```

but it would not have been as clear or efficient.[7] The META expression for `parse-integer` says that integers should be preceded by a *sign* of plus or minus or nothing, followed by a *digit*, and then followed by any number (including none) of additional *digits*. We note that the regular expression alternative of having a possibly empty digit sequence *first*, followed by a single digit, will not work in META, however. The reason for this will become clear.

If all META did was recognize regular expressions, it would not be very useful. It is a programming language, however, and the operations [], {} and $ correspond to the Common Lisp control structures `AND`, `OR`, and `DO`.[8] Therefore, we can utilize META to not only *parse*, but also to *transform*. In this way, META is analogous to "attributed grammars" [Aho86], but it is an order of magnitude simpler and more efficient. Thus, with the addition of the "escape" operation "!", which allows us to incorporate arbitrary Lisp expressions into META, we can not only parse integers, but produce their integral value as a result.[9]

---

[4]Unfortunately, it is not clear where the *art* of programming will be taught.

[5]A potentially more efficient and general, but perhaps less perspicuous definition would define digit as `(deftype digit () '(satisfies digit-char-p))`.

[6]Modern text editors (e.g., EMACS) can be customized to match braces and brackets as easily as parentheses.

[7]We show later how to deal with inefficient Common Lisp `typep`'s.

[8]More precisely, `$x` is analogous to `(NOT (DO () ((NOT x))))`. META control structures thus bear more than a passing resemblance to the TECO text editor control structures (TECO still comes with DEC VMS software).

[9]If Kernighan and Ritchie had been aware of META parsing techniques, the C language [Kernighan78] would have had `for` (loop) *expressions*, rather than *statements*, and `lex/yacc` [Johnson78] might now be mere curiosities.

Below is a parser for signed integers which returns the integer.

```
(defun ctoi (d) (- (char-code d) #.(char-code #\0)))

(defun parse-int (&aux (s +1) d (n 0))
 (and
  (matchit
   [{#\+ [#\- !(setq s -1)] []}
    @(digit d) !(setq n (ctoi d))
    $[@(digit d) !(setq n (+ (* n 10) (ctoi d)))]])
  (* s n)))
```

Below is the code into which it compiles.

```
(defun parse-int (&aux (s +1) d (n 0))
 (and
  (and (or (match #\+)
           (and (match #\-) (setq s -1))
           (and))
       (match-type digit d) (setq n (ctoi d))
       (not (do () ((not (and (match-type digit d)
                              (setq n (+ (* n 10) (ctoi d)))))))))
  (* s n)))
```

Before we can delve further into `match` and `match-type`, we must make clear what it is we are trying to parse—whether it be a Common Lisp character stream, a character string, or a Lisp list. If the source of the text we are trying to parse is an internal source, like a character string or a Lisp list, then we are in a position to be able to back up. If the source is a standard Common Lisp character stream, however, then we can only look ("peek") one character ahead.

First, consider a standard Common Lisp character stream source:

```
(defmacro match (x) `(when (eql[10] (peek-char) ',x) (read-char)))

(defmacro match-type (x v)
  `(when (typep (peek-char) ',x) (setq ,v (read-char))))
```

These macros allow us to match a given character or character type against the input stream, and if the match succeeds, the stream is advanced, while if the match fails, then the stream is left where it was. Unfortunately, once a match succeeds against such a source, we are now committed to that path, because we can no longer back up. This means that the original regular expression must be "deterministic", in the sense that any sequence is determined by its first element.

While this determinism requirement would seem to substantially limit our technique, one can usually get around the restriction by "factoring out" of an alternative the leftmost character of the alternate sequences. For example, the expression `{[#\: #\@] [#\:]}` can be factored to get `[#\: {#\@ []}]`. (Of course, one should also put off performing any transformation side-effects until one has arrived at the correct alternative branch.) Using this technique, we can scan an optional sign, digits, and an optional decimal point from the front of a number, even before we know whether the number will be an integer, a ratio, or a floating-point number [Steele90,22.1.2]. A ratio will be signalled by a "`/`", a floating-point number will be signalled by additional digits or an exponent, and an integer will have none of these.[11]

[10]The use of `eql` keeps `match` and `match-type` consistent; `eql` is required, in general, for comparing characters. See [Baker93] for a lengthy discussion on object equality.

[11]The case of Lisp's isolated dot "." is most easily and efficiently handled by initially parsing it as the "integer" zero!

Below is a parser/transformer for Common Lisp real numbers [Steele90,22.1.2].

```
(deftype sign () '(member #\+ #\-))

(deftype expmarker () '(member #\e #\s #\f #\d #\l #\E #\S #\F #\D #\L))

(defun parse-number (&aux x (is #\+) id (i 0) dd (d 0)
                          fd (f 0) (nf 0) (es #\+) ed (e 0) (m #\e))
  ;;; Parse CL real number according to [Steele90,22.1.2]
  ;;; Return 2 values: the number and a reversed list of lookahead characters.
  (matchit
   [{[@(sign is) !(push is x)] []} ; scan sign.
    $[@(digit id) !(setq x nil i (+ (* i 10) (ctoi id)))]    ; integer digits.
    {[!id #\/ !(push #\/ x)                                  ; "/" => ratio.
      $[@(digit dd) !(setq x nil d (+ (* d 10) (ctoi dd)))]] ; denom. digits.
     [{[#\. {!id !(push #\. x)}                              ; decimal point.
        $[@(digit fd)
          !(setq x nil nf (1+ nf) f (+ (* f 10) (ctoi fd)))]] ; fract. digits.
       []}
      {[{!id !fd} @(expmarker m) !(push m x)                 ; exp. marker.
        {[@(sign es) !(push es x)] []}                       ; exponent sign.
        $[@(digit ed) !(setq x nil e (+ (* e 10) (ctoi ed)))]] ; exp. digits.
       []}]}])
  (let ((sign (if (eql is #\-) -1 1))
        (ex (if (eql es #\-) (- e) e)))
    (values (cond ((or fd ed) (make-float m sign i f nf ex)) ; see [Clinger90]
                  (dd (/ (* sign i) d))
                  (id (* sign i))
                  (t nil))
            x)))
```

We first note that this half-page function is only slightly longer than the grammar for numbers given in [Steele90,22.1.2], and is only slightly less readable. Second, we note that we have utilized both null tests on local variables in addition to the standard `match` predicates in this program. For example, the test `{!id !fd}`, which checks for the existence of either integer or fraction digits, must succeed before the exponent marker can be scanned. Third, preserving the characters that were looked at but not used requires additional work, because Common Lisp streams support only 1 character lookahead, yet many Common Lisp parsing tasks require up to 3 lookahead characters.

Below is the actual META compiler.

```
(defun compileit (x)
  (typecase x
    (meta
     (ecase (meta-char x)
       (#\! (meta-form x))
       (#\[ `(and ,@(mapcar #'compileit (meta-form x))))
       (#\{ `(or ,@(mapcar #'compileit (meta-form x))))
       (#\$ `(not (do ()((not ,(compileit (meta-form x)))))))
       (#\@ (let ((f (meta-form x))) `(match-type ,(car f) ,(cadr f))))))
    (t `(match ,x))))

(defmacro matchit (x) (compileit x))
```


We will also need a few macro character definitions for the additional syntax.

```
(defstruct (meta
             (:print-function
              (lambda (m s d &aux (char (meta-char m)) (form (meta-form m)))
                (ecase char
                  ((#\@ #\! #\$) (format s "~A~A" char form))
                  (#\[ (format s "[~{~A~^ ~}]" form))
                  (#\{ (format s "{~{~A~^ ~}}" form))))))
  char
  form)

(defun meta-reader (s c) (make-meta :char c :form (read s)))

(mapc #'(lambda (c) (set-macro-character c #'meta-reader)) '(#\@ #\$ #\!))

(set-macro-character #\[
 #'(lambda (s c) (make-meta :char c :form (read-delimited-list #\] s t))))

(set-macro-character #\{
 #'(lambda (s c) (make-meta :char c :form (read-delimited-list #\} s t))))

(mapc #'(lambda (c) (set-macro-character c (get-macro-character #\) nil)))
      '(#\] #\}))
```

2. META Parsing of Common Lisp Strings

In many cases, we want to parse strings rather than streams. While we could use the Common Lisp function `make-string-input-stream` along with our previous code, we will find that matching and especially backing-up will be faster on strings. Since we no longer have to worry about backing up, we will be able to search further forward without having to factor the grammar as described in the previous section. We need not capture the actual characters as a substring is scanned, but need only save their beginning and ending locations, because we still have access to the original string; this feature speeds the parsing of format control strings, in which major portions of the given string are just constant characters.

Our `matchit` macro will now generate code that implicitly refers to the lexical variables `string`, `index` (the starting index), and `end` (the ending index), which should be defined in the lexically surrounding environment. By utilizing lexical instead of special (dynamic) variables, we can gain substantially in execution speed, and the same techniques will work in other languages—e.g., Scheme—assuming that they have powerful enough macro facilities.

```
(defmacro match (x)
  `(when (and (< index end) (eql (char string index) ',x))
     (incf index))

(defmacro match-type (x v)
  `(when (and (< index end) (typep (char string index) ',x))
     (setq ,v (char string index)) (incf index))

(defun parse-int (string &optional (index 0) (end (length string))
                         &aux (s +1) d (n 0))
 ;;; Lexical 'string', 'index', and 'end', as required by matchit.
 (and
  (matchit
   [{#\+ [#\- !(setq s -1)] []}
    @(digit d) !(setq n (ctoi d))
    $[@(digit d) !(setq n (+ (* n 10) (ctoi d)))]])
  (* s n)))
```

Due to our ability to back up when parsing strings, we can now enhance our META language with a construct to match a constant string—e.g., `"abc"`—instead of having to match the individual letters `[#\a #\b #\c]`. We do this by saving the current string pointer before beginning the match, and backing up to the saved index if the match fails—even after the first character.

```
(defmacro match (x)
  (etypecase x
   (character
    `(when (and (< index end) (eql (char string index) ',x))
        (incf index)))
   (string
    `(let ((old-index index)) ; 'old-index' is a lexical variable.
        (or (and ,@(map 'list #'(lambda (c) `(match ,c)) x))
            (progn (setq index old-index) nil))))))
```

3. META Parsing of Lisp Lists

So far, we have only parsed character strings. Common Lisp lambda parameter lists and macros require the ability to parse Lisp lists, however. Below is a parser for lambda parameter lists with only required and optional parameters. We give a parser for full lambda-lists as an appendix.[12]

```
(deftype vname () `(and symbol (not (member ,@lambda-list-keywords))))

(defun lambda-list (ll &aux (index `(,ll)) var initform svar)
 (matchit
  ($@(vname var)
   {[&OPTIONAL                ; we use upper case here only for readability.
     ${@(vname var)
       (@(vname var)
        {[@(t initform) {@(vname svar) []}]
         []})}]
    []})))
```

Interestingly enough, we can utilize the same parsing techniques on lists that we used on streams and strings. We need only rewrite `match` and `match-type`. We first show a version that matches only atoms.

```
(defmacro match (x)
  `(when (and (consp index) (eql (car index) ',x))
     (pop index) t))

(defmacro match-type (x v)
  `(when (and (consp index) (typep (car index) ',x))
     (setq ,v (car index)) (pop index) t))
```

We now extend `match` so that it can recursively match on sublists.

```
(defun compilelst (l)
  (if (atom l) `(eql index ',l)
      `(and ,(compileit (car l)) ,(compilelst (cdr l)))))

(defmacro match (x)                ; sublist uses new lexical index
 `(when (and (consp index)
             ,(if (atom x) `(eql (car index) ',x)
                  `(let ((index (car index))) ,(compilelst x))))
    (pop index) t))
```

C. CONTEXT-FREE GRAMMARS

So far, we have only considered grammars which were expressed as a single regular expression. Since we have iteration, but not recursion, we cannot leave the domain of finite state languages.

[12] An efficient META-based lambda-list parser allows for a code-*runner* (as opposed to a code-walker).

By giving portions of our grammars names, however, and then utilizing those names recursively, we immediately gain the possibility of parsing (some) context-free languages.[13]

As might be expected, each named expression becomes a "production", which is implemented as a Lisp function. These Lisp functions must return a truth value, because our `AND`/`OR` nests use truth values for navigation. This means that a grammar which performs a transformation cannot communicate its results through a normal Lisp return, but must communicate by means of side-effects. In order for these side-effects to be communicated through lexical instead of special/dynamic variables, we group the "production" functions into a `LABELS` nest which is lexically included within a larger function which provides the lexical variables used for communication. We have already seen the use of lexical variables local to each production for communication within that production; we use lexical variables outside the `LABELS` nest for communication among the productions and for communication of the result outside of the `LABELS` nest. As a mnemonic, we often use a nasty Common Lisp pun, and make the name of the result variable for a function be the same as the name of the function.

The code below illustrates the use of a `LABELS` nest of productions, although we have already shown how to parse numbers using a single grammar. Note that each production saves the value of the `index` location, so that if the production fails, the program can back up to where it was when the production started. Note also that we utilize the capability of executing arbitrary Lisp code within the "!" escape expression to actually call the production as a function; this capability could be used to parameterize a production by passing arguments.[14]

```
(defun parse-number (&aux (index 0) integer ratio floating-point-number)
 (labels
  ((integer (&aux (old-index index) <locals for integer>)
     (or (matchit <grammar for integer>)
         (progn (setq index old-index) nil)))
   (ratio (&aux (old-index index) <locals for ratio>)
     (or (matchit <grammar for ratio>)
         (progn (setq index old-index) nil)))
   (floating-point-number (&aux (old-index index) <locals for f-p-n>)
     (or (matchit <grammar for floating-point-number>)
         (progn (setq index old-index) nil))))
  (matchit {!(integer) !(ratio) !(floating-point-number)})
  (return (or integer ratio floating-point-number))))
```

Our compiler for Common Lisp format strings utilizes this technique.

D. EFFICIENCY, OPTIMIZATION AND A CHALLENGE TO SCHEME

We have claimed that META can be efficient, so we must show how META's compilation can produce nearly optimal machine code. The basic tools we will use are declarations, caching, inlining and short-circuiting.

1. Declarations

Through the use of declarations, we aid the compiler and make sure that it knows which variables are characters, which are fixnums and which are more general variables, so that the most efficient code is generated. Of particular importance are the fixnum declarations for string index variables which are incremented/decremented and the cons declarations for variables which must be popped; in both these cases only one or two machine instructions need be generated.

If parsing a string, we should find the underlying "simple" string and parse that instead, after suitably translating the starting and ending indices. This is because it is cheaper to perform the

---

[13]An *arbitrary* context-free language can be parsed by techniques such as the LINGOL parser [Pratt73], which can parse in time $O(n^3)$; we have also converted this parser to Common Lisp.

[14]Woods' *Augmented Transition Networks* (ATN's) [Woods70] were a rediscovery of the META technique for CF grammars.

translation once, rather than performing it on every access to the string. On a byte-addressed machine, access to a "simple" string consists of an index check plus an indexed load; since we are already checking the index in the parser, we can dispense with the redundant index check during the string access.

If parsing a list, then the parser will already be checking for the end of the list, so that unchecked `CAR` and `CDR` instructions may be safely used in this instance.

2. Caching

In several cases, we "peek" at the same character several times before it finally matches. This problem can easily be corrected by caching the next character in a lexical variable, so that the compiler may possibly keep this lexical variable in a register. The need to cache such a character is not so acute in the C language, where stream accessing functions `getc` and `ungetc` are most likely defined as macros which already manipulate a cached value, but Common Lisp's `peek` and `read-char` are quite heavyweight due to the flexibility of streams and the large variety of options.

The one potential problem with caching is what value to store in the cache on an end-of-file. For maximum efficiency, one should be able to treat it as just another character, which doesn't match any real character or character type. One does not want to restrict the kinds of characters that can appear in a grammar, however. One is therefore led to the C solution of utilizing a non-character as an EOF value.

3. Inlining

Potentially the most expensive single operation in a META parser is the call to `typep` which is produced by "`@(`<type> <variable>`)`". We have used the Common Lisp type system for character class hacking, because it is very flexible and it was already there. In most Common Lisp implementations, however, a `typep` call is likely to be quite slow. If one is lucky, then simply proclaiming `typep` to be `inline` should speed parsing up substantially. If this is still too slow, we have two choices: we can either compile more complex code, or we can fix `typep` to be more efficient. We choose the course of making `typep` more efficient.

We change our compiler to compile into a macro `my-typep`,[15] which recognizes the important special cases, such as a `member` list, which we will compile into a `case` macro. The `case` macro already has enough restrictions and context to compile into a table-driven dispatch, and if a particular implementation does not do this, then we can expand `my-typep` into the macro `my-case`, which will.

4. Short-circuiting

META uses a plethora of `AND`'s and `OR`'s, which can be quite inefficient if not compiled properly. If your compiler compiles these expressions efficiently, then you may skip this section.[16]

The proper compiling technique for compiling `AND`/`OR` expressions has been known since the earliest days of Lisp, but is not *well*-known, and students continually rediscover it.[17] Short-circuited boolean expressions can be efficiently compiled in a "single" recursive pass which produces nearly optimal code—even in the presence of weak jump instructions which cannot reach all of memory [Baker76a] [Baker76b].

The trick is to pass two "continuations" (really jump addresses) as arguments to the boolean expression compiler which are the "success" and "failure" continuations of the expression being compiled, and to emit the code *backwards* in the style of dynamic programming; the compiler returns as a result the address of the compiled expression. Since one already knows the location to

---

[15]We could alternatively install a compiler optimizer, in which case *every* call to `typep` would be speeded up.

[16]Apple Coral Common Lisp for the Macintosh appears to handle short-circuits reasonably well.

[17]The technique has been described for Lisp and ML [Harper86] [Harper88] pattern matchers, as well as for generic short-circuit evaluation; see references in [Aho86,p.512].

which one must jump at the time a branch is emitted, as well as the location of the current instruction, one can easily choose the correct short/long jump sequence.[18] Our parser also requires the efficient compilation of loops, which require a little more work because the jump-to location is not yet known for backward jumps (which occur at the end of a loop). The simplest solution is to always emit a backwards unconditional long jump, which we will then patch after the first instruction of the loop has been emitted. Since we know at that time the length of the jump, we can patch in a short jump/no-op sequence if short jumps are faster; the no-op's are never executed because the loop never "falls through".

Even though our technique wastes a little space after loops in the code, the code is otherwise nearly optimal, since the majority of jumps (and all conditional jumps) are forward jumps. Another source of non-optimality comes from early exits from the body of a loop to the end of the loop. One can conceive of these branches also being optimized to jump to the beginning of the loop, but this situation is rare enough not to cause any significant loss of speed. In any case, the advantage of a compiler optimization must be traded against the cost of programming and maintaining it; the optimizations we suggest are extremely simple and quite often advantageous.

Below is a simple compiler for short-circuited boolean expressions with loops.

```
(defvar *program* nil "The reversed list of program steps.")

(defun emit (instruction next &aux (label (gentemp)))
  (unless (eql (car *program*) next) (push `(go ,next) *program*))
  (push instruction *program*) (push label *program*)
  label)

(defun emit-test (test succ fail &aux (label (gentemp)))
  (push (cond ((eql (car *program*) succ) `(unless ,test (go ,fail)))
              ((eql (car *program*) fail) `(when ,test (go ,succ)))
              (t `(if ,test (go ,succ) (go ,fail))))
        *program*)
  (push label *program*)
  label)

(defun compile-seq (x s f)
  (if (null x) s (compile (car x) (compile-seq (cdr x) s f[19]) f)))

(defun compile-alt (x s f)
  (if (null x) f (compile (car x) s (compile-alt (cdr x) s f))))

(defun compile (x succ fail)
  (typecase x
   (sequence (compile-seq x) succ fail)
   (alternative (compile-alt x) succ fail)
   (loop (let* ((go-back (append '(go nil) nil))
                (loop (compile (loop-body x) (emit go-back succ) succ)))
           (setf (cadr go-back) loop)))
   (execute (emit-test (execute-body x) succ fail))
   (t (emit-test `(match ,x) succ fail))))
```

5. Results and a Challenge to Scheme

Through the use of these techniques, we are able to achieve nearly the same machine code that we would have generated ourselves if we were asked to program the parser in assembly language in the first place. We have programmed and optimized a simple integer parser similar to the our first example grammar which operates on simple strings—much like the standard built-in Common Lisp

[18]The backwards emission technique can also easily handle the "pipelined" jumps found in RISC architectures.

[19]This parameter should be an error label once the sequence has committed and can no longer back up; see the literature on post-Prolog "committed choice" languages [Maher87] [Shapiro89].

function `parse-integer`. We ran this function repeatedly on a 80,000-character simple string which consisted of 10,000 copies of the sequence "`+123456 `". Our META `parse-integer` took about 25.4µsec/char., the built-in `parse-integer` took about 222µsec/char., and `read-from-string` took about 700µsec/char.[20] (Timings were performed on an Apple Macintosh Plus with a Radius 68020 accelerator card, 4Mbytes of memory, and Coral Common Lisp v1.2 set to the highest optimization levels.) Our optimized META `parse-integer` did not call any other functions, not even subprimitives. It therefore appears that no further speed can be gained until Coral utilizes the full 68020 instruction set and does a better job of compiling fixnum arithmetic; e.g., it refuses to keep unboxed fixnums in registers very long.

The META parsing technique is an interesting challenge to Scheme compiler writers, because a Scheme compiler must itself perform all of the optimizations we have discussed. Of particular interest is the ability of a Scheme compiler to transform the nest of mutually recursive tail-calling functions which are Scheme's equivalent of assembly language "goto" labels into assembly language "goto" labels. This transformation should be possible, because none of these "functions" have arguments.

E. CONCLUSIONS

We have shown a simple technique called META for building very fast parsers/translators in Common Lisp, which is more general than some other techniques—e.g., the *CGOL* [Pratt76] operator precedence system used in Macsyma. This technique already produces readable code, as we have shown by a number of examples, but some might wish for even more readable code. In such a case, one can easily utilize the META technique to produce a "reader macro" which translates the *exact* [Steele90]-style syntax equations into an efficient parser. Our stomach, still queasy from too much syntax, has so far vetoed these efforts.[21] Should a grammar require it, the META technique can also be easily extended to handle minimum/maximum numbers of loop iterations in the sequence "$" construct.

There are several advantages to embedding a special-purpose parsing language into Common Lisp. First, it provides a higher level of abstraction, which allows one to concentrate on recognizing the correct syntax. Second, this abstraction is more compact, allowing the programmer to focus his attention on a smaller body of code.[22] Third, this higher level of abstraction allows for a number of different implementations, of which we have exhibited three. Finally, we can get the parser running relatively quickly, and if later additional speed is required, we can change the underlying implementation without changing the code for the grammar.

We have found one problem in the use of the META system presented here—the inadvertent returning of `NIL` by an escape expression "!", which causes a failure out of a sequence and a possible nasty bug. We considered the possibility of including *two* execution escape characters—one for *predicates*, and one for *statements* like `setq`. This change would avoid the necessity of wrapping `(progn ... t)` around statements. We have not done this, however, because our shyness about using up macro characters has exceeded our fear of bugs.

We are uncomfortable about the large volume of side-effects present in META-style parsers. Given the nature of a parsing task, however, which is emulating a state machine (with or without a push-down stack), it is unlikely that a parser without side effects could be very efficient without an extremely clever (i.e., very expensive) compiler. The META technique does not seem to easily extend to parsing tasks which must be able to "rub out"—e.g., directly parsing user type-in—because that task seems to require the ability for arbitrary back-up, including backing up over side-

---

[20]Coral Common Lisp v1.2 `read-from-string` appears to erroneously ignore its `:start` argument.

[21]The original META paper [Schorre64] gives such a compiler for "BNF"-style syntax equations, which we previously implemented (1966) in IBM 360 machine code.

[22]The code complexity of the resulting code is extremely high when measured by software engineering complexity metrics; "productivity", as measured by these metrics, thus goes off-scale.

effects. The *LINGOL* parsing system [Pratt73], based on "mostly functional" techniques, handles the "rub out" problem quite elegantly.

Because META parsers address the backing-up issue directly as they are programmed, META needs no complex run-time system like ATN's [Woods70] or Prolog. We conjecture, therefore, that Prolog would need an extremely clever compiler to deduce the same degree of backing-up optimization that the programmer manually performs in META.

While we have argued for the use of an embedded special purpose parsing language, we have *NOT* advocated adding this language to the Common Lisp standard, which is weighted down far too much as it is. A language like META that can be programmed in less than 1 page of code is hardly worth standardizing. Standardization would also inhibit the possibilities of *ad hoc* optimizations for a particular purpose, in which case the programmer would not use META at all. For these reasons, we have presented META as a technique, rather than as a rigidly-defined language.

We have with great trepidation reviewed this technique for making parsing tasks easier to program, because we feel that parsing complex syntax is an inherently low-value activity. Unfortunately, any tool that makes these tasks easier is likely to get used, but not necessarily for the good; e.g., it is said that all C programmers ever do is `yacc`, `yacc`, `yacc`!

F. ACKNOWLEDGEMENTS

We appreciate the suggestions of André van Meulebrouck for improving this paper.

G. REFERENCES

Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.

Baker, Henry. "COMFY—A Comfortable Set of Control Primitives for Machine Language Programming". Unpublished manuscript, 1976.

Baker, Henry. "COMFY-65—A Medium-Level Machine Language for 6502 Programming". Unpublished manuscript, 1976.

Baker, Henry. "Equal Rights for Functional Objects". ACM *OOPS Messenger 4*,4 (Oct. 1993), 2-27.

Clinger, William D. "How to Read Floating Point Numbers Accurately". *ACM PLDI'90, Sigplan Not. 25*,6 (June 1990), 92-101.

Curtis, Pavel. "(algorithms)" column on Scheme macros. Lisp Pointers 1,6 (April-June 1988),LPI-6.19-LPI-6.30.

des Rivières, Jim, and Kiczales, Gregor. *The Art of the Metaobject Protocol, Part I.* Unpublished manuscript, Xerox PARC, Oct. 1990.

Harper, R., MacQueen, D., and Milner, R. "Standard ML". ECS-LFCS-86-2, Comp. Sci. Dept., U. of Edinburgh, March 1986,70p.

Harper, R., Milner, R., Tofte, Mads. "The Definition of Standard ML, Version 2". ECS-LFCS-88-62, Comp. Sci. Dept., U. of Edinburgh, Aug. 1988,97p.

Haynes, Christopher T. "Logic Continuations". *J. Logic Progr. 4* (1987),157-176.

Johnson, S.C. and Lesk, M.E. "UNIX Time-Sharing System: Language Development Tools". *Bell Sys. Tech. J. 57*,6 (July-Aug. 1978),2155-2175.

Kernighan, B. W., and Ritchie, D. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1978.

Maher, M.J. "Logic semantics for a class of committed-choice programs". *Proc. 4th Int'l Conf. of Logic Progr.*, MIT Press, 1987,858-876.

McCarthy, John. "History of LISP". *ACM Sigplan Not. 13*,8 (Aug. 1978),217-223.

Pratt, V.R. "A Linguistics Oriented Programming Language". *Proc. IJCAI 3* (Aug. 1973),372-381.

Pratt, V.R. "CGOL—an Alternative External Representation for LISP users". AI Working Paper 121, MIT AI Lab., March 1976,13p.

Rulifson, J.F., Derksen, J.A., and Waldinger, R.J. "QA4: A Procedural Calculus for Intuitive Reasoning". SRI AI Ctr. Tech. Note 73, Nov. 1972,363p.

Schorre, D.V. "META II: A Syntax-Oriented Compiler Writing Language". *Proc. 19'th Nat'l. Conf. of the ACM* (Aug. 1964),D1.3-1-D1.3-11.

Schneider, F., Johnson, G.D. "META-3: A Syntax-Directed Compiler Writing Compiler to Generate Efficient Code". *Proc. 19'th Nat'l. Conf. of the ACM* (1964),D1.5-1-D1.5-8.

Shapiro, E. "The Family of Concurrent Logic Programming Languages". *ACM Comput. Surv. 21*,3 (Sept. 1989),412-510.

Woods, W.A. "Transition Network Grammars for Natural Language Analysis". *CACM 13*,10 (Oct. 1970),591-606.

APPENDIX — COMMON LISP LAMBDA PARAMETER LISTS

```
(defun parse-lambda-exp
       (x &optional (env nil) &aux
        body rqds opts rst kwds? kwds okys? auxs fenv senv lenv
        v i sv k type form d pdecls specials)
  ;;; Parse a lambda-expression x and return 11 values:
  ;;; 1-3. body, reqd vars, opts (v i sv)
  ;;; 4-5. rest variable (or nil), keys? (can be t even when #6 is nil)
  ;;; 6-8  kwds ((k v) i sv), other keys?, auxs (v i)
  ;;; 9-11. lcl fn env, special env (except local specials), lcl var env.
  ;;; This lambda parser is presented for illustration only, and may not
  ;;; correctly implement ANSI Common Lisp syntax or semantics.
  (matchit x
   (LAMBDA                      ; we use upper case here only for readability.
     ($[@(vname v) !(push v rqds) !(push (make-vbind v) lenv)]
      {[&OPTIONAL
        $[{@(vname v) (@(vname v) {[@(t i) {@(vname sv) []}] []})}
          !(progn (push `(,v ,i ,sv) opts) (push (make-vbind v) lenv)
                  (when sv (push (make-vbind sv '(member nil t)) lenv))
                  (setq i nil sv nil) t)]] []}
      {[&REST @(vname rst) !(push (make-vbind rst 'list) lenv)] []}
      {[&KEY !(setq kwds? t)
        $[{@(vname v)
           ({@(vname v) (@(symbol k) @(vname v))}
            {[@(t i) {@(vname sv) []}] []})}
          !(progn (unless k (setq k (intern (symbol-name v) 'keyword)))
                  (push `((,k ,v) ,i ,sv) kwds) (push (make-vbind v) lenv)
                  (when sv (push (make-vbind sv '(member nil t)) lenv))
                  (setq k nil i nil sv nil) t)]
        {[&ALLOW-OTHER-KEYS !(setq okys? t)] []}] []}
      {[&AUX $[{@(vname v) (@(vname v) {@(t i) []})}
               !(progn (push `(,v ,i) auxs) (push (make-vbind v) lenv)
                       (setq i nil) t)]] []})
     ;;; Now process declarations.
     $[(DECLARE
        ${(SPECIAL $[@(vname v) !(push v specials)])
          ({[TYPE @(t type)] @(typename type)}
           $[@(vname v) !(setf (dtype (lookup v lenv))
                               `(and ,type ,(dtype (lookup v lenv))))])
          (IGNORE $[@(vname v) !(progn (setf (dtype (lookup v lenv)) nil) t)])
          [@(t d) !(progn
                    (when (eq (car d) 'function)
                     (setq d `(ftype (function ,@(cddr d)) ,(cadr d))))
                    (push d pdecls)
                   t)]})]
     $[@(t form) !(push form body)])) ; Error if body ends in non-list.
  ;;; Fix up local environment to correctly handle special variables.
  (dolist (binding lenv)
    (let* ((v (vbind-name binding)))
      (when (or (declared-special-p v env) (member v specials))
        (setf (vbind-special-p binding) t))))
  ;;; Create special environment
  (let* ((nenv (append lenv env)))
    (dolist (v specials)
      (unless (declared-special-p v nenv)
        (setq senv (lx-bind-special v senv)))))
  (values (reverse body) (reverse rqds) (reverse opts) rst
          kwds? (reverse kwds) okys? (reverse auxs) fenv senv lenv))
```

We have shown one scheme for parsing Common Lisp lambda parameter lists ("lambda-lists") which easily generalizes to handle the full complexity of lambda-lists. It has one source of inefficiency which is not easily eliminated, however. Whenever a variable name is to be matched, we first check that the tentative variable name is a symbol, and we then check to see that it is *not* one of the lambda-list keywords (`&optional`, `&rest`, &cetera). While this check can be open-coded quite efficiently, we find that if a lambda-list keyword does occur, then it will be compared against the list twice—once to determine that it is not a legal variable name, and once to determine which of the lambda-list keywords it is. This inefficiency is due to the definition of variable names as the *complement* of the set of lambda-list keywords *relative* to the set of symbols. While this inefficiency could conceivably be eliminated by the use of escape forms like block/return-from, we feel that any additional speedup will be overwhelmed by the additional costs of regularizing the lambda-list for a less-sophisticated consumer.

