```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  PARCIL - A Parser for C syntax In Lisp
;;;  version 0.1a
;;;
;;;  copyright (c) 1992 by Erann Gat, all rights reserved
;;;
;;;  This program is free software; you can redistribute it and/or modify
;;;  it under the terms of the GNU General Public License as published by
;;;  the Free Software Foundation.
;;;
;;;  This program is distributed in the hope that it will be useful,
;;;  but WITHOUT ANY WARRANTY; without even the implied warranty of
;;;  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
;;;  GNU General Public License for more details.
;;;
;;;  You should have received a copy of the GNU General Public License
;;;  along with this program; if not, write to the Free Software
;;;  Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
;;;
;;;
;;;  This is a very preliminary release and almost certainly contains bugs.
;;;  Please send bug reports and comments to:
;;;  Erann Gat
;;;  JPL MS 525-3660
;;;  4800 Oak Grove Drive
;;;  Pasadena, CA 91109
;;;  (818) 306-6176
;;;  gat@robotics.jpl.nasa.gov or gat@aig.jpl.nasa.gov
;;;
;;;  Revision history:
;;;  v0.1a - Initial release
;;;

;;;  PARCIL is a parser for a subset of the syntax for the C programming
;;;  language.  PARCIL is written in Common Lisp, making it potentially
;;;  a useful building block for user interfaces for people who do not
;;;  like prefix syntax.
;;;
#|
PARCIL is a recursive descent parser optimized to parse C.  This makes it
fairly brittle and difficult to modify.  However, it does make it fairly fast,
and it also allows the parser to deal with lots of C idiosyncrasies which are
difficult to implement in general-purpose parsers, e.g. operator precedence,
prefix and postfix operators, etc.
  NOTE:  While PARCIL is designed to be a component in user interfaces for
people who are not regular LISP users, it is probably not usable for that
purpose as-is.  There are two major problems with it.  First, it is
incomplete. It currently includes no support for any high-level C construct
(i.e. it implements the syntax described in the original Kernighan and Richie
book, section 18.1).  The second problem is that PARCIL is so faithful to C
syntax that it can easily fool the unwary into believing that they are writing
```

```
C code when in fact they are writing LISP code, only with a different syntax.
You need a fairly deep understanding of the distinction between syntax and
semantics in order to use PARCIL.  The main stumbling block to its use by
beginners is that PARCIL does very little error checking.  Thus, many errors
which should be detected by PARCIL are passed on and caught by LISP.  The
resulting error messages can be very cryptic if you don't know what's going
on.
  PHILOSOPHICAL RANT:  Infix notation is a blight on the intellectual
landscape.  It is confusing to read, difficult to parse, and to avoid
ambiguity must rely on precedence rules that are hopelessly obscure.  People
who prefer infix notation do so only because they have been indoctrinated to
it since childhood and do not have the intellectual strength to break free.
It is far better to convince people to use prefix notation, with its easy to
read and easy to parse, unambiguous syntax, than to provide them with crutches
such as PARCIL which perpetuate such evils as infix, prefix and postfix unary
operators.  (In C, "x++++++++y" is a legal expression, and the first *
doesn't mean the same thing as all the other *'s.)  Nevertheless, I
acknowledge the reality that infix and C are here to stay, and that is why I
have written PARCIL.  But that doesn't mean I have to like it.
|#
;;;
;;;  USER'S GUIDE:
;;;
;;;  The top-level function is called PARCIL.  Pass a string consisting of a C
;;;  expression (not a command!) to PARCIL and it will return a parsed ;;;
;;;  version. For example:
;;;
;;;  (parcil "x=y*sin(pi/2.7)") ==> (SETF X (* Y (SIN (/ PI 2.7))))
;;;
;;;  PARCIL supports all syntax defined in section 18.1 of the original
;;;  Kernighan and Ritchie book, plus all C numerical syntax including floats
;;;  and radix syntax (i.e. 0xnnn, 0bnnn, and 0onnn).  In addition, PARCIL
;;;  supports multiple array subscripts.  There is also a preliminary version
;;;  of {} blocks, but it doesn't quite do the right thing.  Parcil also
;;;  allows strings to be delimited using single quotes as well as double
;;;  quotes (but you must use the same type to close the string as you did to
;;;  open it).

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  Program starts here
;;;

;;;  Misc. utilities
;;;
(defmacro iterate (name args &rest body)
  `(labels ((,name ,(mapcar #'car args) ,@body))
     (,name ,@(mapcar #'cadr args))))

(defmacro while (condition &body body)
  `(iterate loop () (if ,condition (progn ,@body (loop)))))

;;;  Crufty pseudo-text-file interface.  Don't let impressionable young minds
;;;  see this code.
```

```
;;;
(defvar *the-string* "")
(defvar *the-pointer* 0)

(defun parse-init (s)
  (setq *the-string* s)
  (setq *the-pointer* 0))

(defun eof? (&optional (offset 0))
  (>= (+ *the-pointer* offset) (length *the-string*)))

(defun peek (&optional (offset 0))
  (if (eof? offset)
    nil
    (char *the-string* (+ *the-pointer* offset))))

(defun readc ()
  (prog1 (peek) (incf *the-pointer*)))

;;;  The PARCIL tokenizer.  (FSA?  What's an FSA?)
;;;
(defun letter? (c)
  (and (characterp c) (or (char<= #\a c #\z) (char<= #\A c #\Z))))

(defun digit? (c)
  (and (characterp c) (char<= #\0 c #\9)))

(defun ident? (thing)
  (and thing
       (symbolp thing)
       (letter? (char (symbol-name thing) 0))))

(defvar *binary-ops*
  '((\. ->) (* / %) (+ -) (<< >>) (< > <= >=) (== !=) (&) (^) (\|) (&&) (\|\|)
    (= += -= *= /= %= &= ^= \|= >>= <<=)))

;;; Any binary operator in this alist will be renamed in the parsed version.
(defvar *binop-alist*
  '((\. . struct-ref) (= . setf) (% . mod) (<< . ashl) (>> . ashr)
    (& . logand) (^ . logxor) (\| . logior) (&& . and) (\|\| . or)))

(defun binop? (s) (member s *binary-ops* :test #'member))
(defun assignop? (s) (member s (car (last *binary-ops*))))
(defun priority (s)
  (let ( (p (position s *binary-ops* :test #'member)) )
    (and p (- 40 p))))
(defun translate-binop (op) (or (cdr (assoc op *binop-alist*)) op))

(defun eat-spaces ()
  (do () ( (not (eql (peek) #\Space)) )
    (readc)))

(defun syntax-error ()
  (error "Syntax error near ~S"
```

```
          (subseq *the-string* (max 0 (1- *the-pointer*)))))

(defun parse-fixnum (&optional (base 10))
  (multiple-value-bind
    (n cnt) (parse-integer *the-string* :start *the-pointer* :radix base
                           :junk-allowed t)
    (setq *the-pointer* cnt)
    (if (not (numberp n)) (syntax-error))
    n))

(defun parse-atom ()
  (eat-spaces)
  (if (eof?)
    nil
    (let ( (c (peek)) )
      (cond ( (letter? c) (parse-symbol) )
            ( (eql c #\0) (if (letter? (peek 1))
                              (parse-radix-integer)
                              (parse-number)) )
            ( (digit? c) (parse-number) )
            ( (or (eql c #\") (eql c #\')) (parse-string c) )
            (t (parse-operator))))))

(defun parse-symbol ()
  (intern
   (string-upcase
    (with-output-to-string (s)
      (while (let ( (c (peek)) )
                  (and c (or (letter? c) (digit? c) (eql c #\_))))
        (princ (readc) s))
      s))))

(defun parse-radix-integer ()
  (readc)
  (parse-fixnum (ecase (readc) (#\x 16) (#\o 8) (#\b 2))))

(defun parse-number ()
  (let* ( (n1 (parse-fixnum))
          (c (peek)) )
    (prog ( (d 0.1) )
      (if (eql c #\.) (go decimal))
      (if (or (eql c #\e) (eql c #\E)) (go expt))
      (return n1)
    decimal
      (readc)
      (let ( (c (peek)) )
        (when (digit? c)
          (incf n1 (* d (- (char-code c) (char-code #\0))))
          (setf d (/ d 10))
          (go decimal))
        (if (or (eql c #\e) (eql c #\E)) (go expt))
        (return n1))
    expt
      (readc)
```

```
      (let ( (e (parse-fixnum)) )
        (return (* n1 (expt 10 e)))))))

(defun parse-string (terminator)
  (readc)
  (with-output-to-string (s)
    (iterate loop ()
      (let ( (c (readc)) )
        (when (eql c terminator) (return-from loop s))
        (princ c s)
        (loop)))))

(defun parse-operator ()
  (let* ( (c (intern (string (readc))))
          (s (intern (format nil "~A~A" c (peek)))) )
    (cond ( (member s '(<< >>))
            (readc)
            (if (eql (peek) #\=)
              (intern (format nil "~A~A" s (readc)))
              s) )
          ( (member s '(++ -- << >> -> <= >= != == &&
                        += -= *= /= %= &= ^= \|= \|\|))
            (readc)
            s )
          (t c))))

;;;  Crufty interface to the tokenizer.
;;;
(defvar *next*)

(defun scan ()
  (setf *next* (parse-atom)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  The recursive-descent parser.  Look Ma, no tables!
;;;
(defun parse-expression (&optional (priority -1))
  (iterate loop ( (result (parse-term)) )
    (let ( (op (translate-binop *next*))
           (new-priority (priority *next*)) )
      (cond
       ( (assignop? *next*) (scan) (list op result (loop (parse-term))) )
       ( (and (binop? *next*) (> new-priority priority))
         (scan) (loop (list op result (parse-expression new-priority))) )
       (t result)))))

(defun parse-arglist (&optional (terminator '\)) (separator '\,))
  (iterate loop ()
    (cond ( (null *next*) (error "Missing ~S" terminator) )
          ( (eq *next* terminator) (scan) nil )
          (t (let ( (arg1 (parse-expression)) )
               (unless (or (eq *next* separator) (eq *next* terminator))
                 (syntax-error))
```

```
                (if (eq *next* separator) (scan))
                (cons arg1 (loop)))))))

;;;  Any prefix unary operator included in this table will be renamed in the
;;;  parsed version. (Postfix ++ and -- are handled specially, in PARSE-TERM.)
(defvar *unary-op-alist*
  '((* . deref) (& . address-of)
    (- . -) (! . not) (~ . lognot)
    (++ . incf) (-- . decf)))

;;;  This function parses what K&R call primary expressions.  These include
;;;  numbers, variables, structure references, array references, and all unary
;;;  operators.  Parsing of curly brackets is also stuck in here, though it
;;;  probably shouldn't be.  The weird precedence rules make this a fairly
;;;  hairy and brittle piece of code.
;;;
(defun parse-term ()
  (iterate loop ( (term (prog1 *next* (scan))) )
    (cond
     ( (numberp term) term )
     ( (assoc term *unary-op-alist*)
       (list (cdr (assoc term *unary-op-alist*)) (parse-term)) )
     ( (eq term '\( )
       (cons 'progn (parse-arglist)) )
     ( (eq term '{)
       (list* 'let '() (parse-arglist '} '\;)) )
     ( (eq *next* '\( )
       (scan)
       (loop (cons term (parse-arglist))) )
     ( (eq *next* '\[ )
       (scan)
       (loop `(aref ,term ,@(parse-arglist ']))) )
     ( (eq *next* '\.)
       (loop `(struct-ref ,term ,(prog1 (scan) (scan)))) )
     ( (eq *next* '->)
       (loop `(-> ,term ,(prog1 (scan) (scan)))) )
     ( (eq *next* '++)
       (scan)
       (loop `(prog1 ,term (incf ,term))) )
     ( (eq *next* '--)
       (scan)
       (loop `(prog1 ,term (decf ,term))) )
     (t
      (if (and (atom term) (not (ident? term)))
        (syntax-error))
      term))))

;;;;;;;;;;;;;;;;
;;;
;;;  The top level
;;;
(defun parcil (s)
  (parse-init s)
  (scan)
```

```
  (prog1
    (parse-expression)
    (if *next* (syntax-error))))
                ; If there's stuff left over something went wrong.

;;; Reader hook (optional)
;;; Evaluating the following forms will allow you to type C-like expressions
;;; directly at the lisp reader and have them evaluated, e.g.:
;;;
;;;  ? #{ ( x=1,y=2,print(x+y),sin(pi/2) ) }
;;;  3
;;;  1.0
;;;  ?
(defun |#{-reader| (stream char arg)
  (declare (ignore char arg))
  (parcil
   (with-output-to-string (s)
     (loop
       (let ( (c (read-char stream)) )
         (if (eql c #\})
           (return s)
           (princ c s)))))))

(set-dispatch-macro-character #\# #\{ #'|#{-reader|)
```