Every explanation of anything leaves something out. Once we break down understanding some programming language to various constituent implementation parts, the natural question is to then ask, “Well, how is that part implemented?” It may be that we never get to the end of the explanations, but without question, our explanation becomes more refined. This handout describes one of those parts: how do you type stuff in, and it magically ends up being a list structure?

For example, suppose you type in \texttt{(define (square x) (* x x))}. Our evaluator is given this code as a list of three things:

- the atom \texttt{define},
- the list \texttt{(square x)} (which is in turn a list of the atoms \texttt{square} and \texttt{x}), and
- the list \texttt{(* x x)}.

But all you did is type a string of characters—that is, \texttt{"(define (square x) (* x x))"}, where ( and ) are just characters, and do not mark anything about list structure. How does that string get turned into a list structure?

The first thing we need to do is define some standard operations on strings:

```
> (define (string-car s)
  (substring s 0 1))
> (string-car "string of stuff")
"s"

> (define (string-cdr s)
  (if (= (string-length s) 0)
   s
   (substring s 1 (string-length s))))
> (string-cdr "string of stuff")
"tring of stuff"

> (define (rub s) ;removes extra spaces
  (if (equal? s "")
   ""
   (if (equal? (string-car s) " ")
    (rub (string-cdr s))
    s)))
> (rub " string of stuff")
"string of stuff"

> (define (convert a)
  (if (number? (string->number a))
   (string->number a)
   (string->symbol a))
> "123"
"123"
> (convert "123")
123
```
The first component of the conversion from string to list structure is the *lexical analyzer* or *tokenizer*:

> (tokenize "(define (square x) (* x x))")
'("define" "(" "square" "x" ")" "(" "*" "x" "x" ")" ")")

What *tokenizer* does is take a string of characters, and break it into a list of *tokens*, where a token is either a left or right parenthesis, number, or atom. Procedure *tk* tries to assemble the first token in parameter *a* by reading characters from parameter *s*.

```
(define (glue a L)
  (cond ((equal? a "") L)
        (else (cons a L)))
)

define (tokenize s)
  (define (tk a s)
    ;; (write (list 't a)) (newline) (write (list 't s)) (newline) (newline)
    (cond ((and (equal? a "") (equal? s "")) '())
          ((equal? s "") (list a)) ; end of input? -- return a (as list)
          ((or (equal? (string-car s) ")")
             (equal? (string-car s) "")
             (glue a (glue (string-car s) (tk "" (rub (string-cdr s))))))
          ((equal? (string-car s) " ")
           (glue a (tk "" (rub s))))
          (else (tk (string-append a (string-car s))
                      (string-cdr s)))))
    (tk "" s))
```

> (define ts (tokenize "(define (square x) (* x x)) (square 10) (square 20)")
> ts
'("define" "(" "square" "x" ")" "(" "*" "x" "x" ")" ")")

With the “tracing” line above uncommented, we get:

> (tokenize "(xyz)")
(t "")
(t "(xyz)"
(t "")
(t "xyz")
(t "x"
(t "yz")
(t "xy")
(t "z")
(t "xyz")
(t ")")
(t "")
(t "")
> (tokenize "(define (sq x) (* x x))")
(t "")
(t "define (sq x) (* x x))")
(t ")")
(t "define (sq x) (* x x))")
(t "d")
(t "efine (sq x) (* x x))")
(t "de")
(t "fine (sq x) (* x x))")
(t "def")
(t "ine (sq x) (* x x))")
(t "defi")
(t "ne (sq x) (* x x))")
(t "defin")
(t "e (sq x) (* x x))")
(t "define")
(t " (sq x) (* x x))")
(t ")")
(t "(sq x) (* x x))")
(t "")
(t "sq x) (* x x))")
(t "s")
(t "q x) (* x x))")
(t "sq")
(t " x) (* x x))")
(t ")")
(t "x) (* x x))")
(t "x")
(t ") (* x x))")
(t ")")
(t ")(* x x))")
(t "")
(t "(* x x))")
(t "")
(t "* x x))")
(t "*")
(t " x x))")
Now for the parser—first, a procedure collect that does the following:

\[
\text{(define (collect s)}
\text{  (define (c a s)}
\text{      (if (equal? (car s) \"\")}
\text{          (cons a (cdr s))}
\text{          (c (cons (car s) a) (cdr s))))}
\text{  (c \('\) s))}
\text{)}
\]

\[
> \text{(collect }'(1 2 3 \\"this\" \"all\" \"stays\" \"the\" \"same\")\text{)}
\]

\[
'(3 2 1) \"this\" \"all\" \"stays\" \"the\" \"same\")
\]

Collect takes the first three elements (up to the "(" symbol) of the input list, and assembles them, reversed, as a single list element. The idea is this: that the above list of tokens should be thought of as reversed, namely

\[
'(\"same\" \"the\" \"stays\" \"all\" \"this\" \"(\" 3 2 1\")
\]

and we imagine that the list is followed by the token ")", which should match the left parenthesis "(". In turn, the list of four tokens "(" 3 2 1 is collected into the single list element (3 2 1).

\[
\text{(define (parse ts)}
\text{  (define (p a ts)}
\text{      (write a) (newline) (write ts) (newline) (newline)}
\text{      (cond ((null? ts) (reverse a)) ; token stream is empty, so return reversed list a}
\text{          ((equal? (car ts) \"\") ; starting a new list element, so push ( symbol}
\text{              (p (cons \"\" a) (cdr ts)))}
\text{          ((equal? (car ts) \")\") ; found matching ( symbol, so collect what is enclosed}
\text{              ; up to matching ) as a single list element in a}
\text{              (p (collect a) (cdr ts))))}
\text{          (else (p (cons (convert (car ts)) a) (cdr ts))))})}
\text{  (let ((ps (p \'() ts)))}
\text{    (if (null? (cdr ps))}
\text{      (car ps))}
\text{      ps))})}
\]
In what appears below, each call to procedure \( p \) in the above definition of \( \text{parse} \) prints out the lists (or really—stacks!) \( a \) (the accumulated list structure) and \( ts \) (the list of tokens to be parsed). Notice how the tokens pass from \( ts \) to \( a \), where the corresponding list structure is slowly accumulated.

\[
> \text{parse \ ts}
\]

\[
()\ (
"define" \ ""square" \ "x" \ "")\n\n("\ "*\ "x" \ "x" \ ")\n\n("\ "square" \ "10" \ ")\n\n("\ "square" \ "20" \ ")
\]

\[
("\ "define" \ "")
\]

\[
\n("\ "square" \ "x" \ ")\n\n("\ "*\ "x" \ "x" \ ")\n\n("\ "square" \ "10" \ ")\n\n("\ "square" \ "20" \ ")
\]

\[
("\ "define" \ "")
\]

\[
\n("\ "square" \ "x" \ ")\n\n("\ "*\ "x" \ "x" \ ")\n\n("\ "square" \ "10" \ ")\n\n("\ "square" \ "20" \ ")
\]

\[
\n("\ "define" \ "")
\]

\[
\n("\ "square" \ "x" \ ")\n\n("\ "*\ "x" \ "x" \ ")\n\n("\ "square" \ "10" \ ")\n\n("\ "square" \ "20" \ ")
\]

At this point, \( ts \) begins with a right parenthesis \("\)\), which matches the leftmost \("\ in \( a \). So procedure \( \text{collect} \) compresses the four tokens \("\ in \( a \) to the single list structure \("\ in \( a \).

\[
((\ "define" \ "")
\]

\[
\n("\ "define" \ "")
\]

\[
\n("\ "define" \ "")
\]

\[
\n("\ "define" \ "")
\]

\[
\n("\ "define" \ "")
\]

\[
\n("\ "define" \ "")
\]

\[((\ "define" \ "")
\]

\[
\n("\ "define" \ "")
\]

\[
\n("\ "define" \ "")
\]

\[
\n("\ "define" \ "")
\]

\[((\ "define" \ "")
\]

\[
\n("\ "define" \ "")
\]

\[((\ "define" \ "")
\]

\[\]
Here is another example (but you quickly realize, they are all the same!) of tokenizing and parsing the cube function:

> (tokenize "(define (cube x) (* x x x))")

"((define "cube" x) "(*" x" "x" ")")"

> (parse (tokenize "(define (cube x) (* x x x))")

"(" define "(" cube "x" ")" "(*" x" "x" "x" ")")")

"(" define "(" cube "x" ")" "(*" x" "x" "x" ")")")

"(" define "(" cube "x" ")" "(*" x" "x" "x" ")")")

"(" define "(" cube "x" ")" "(*" x" "x" "x" ")")")

"(" define "(" cube "x" ")" "(*" x" "x" "x" ")")")

"(" define "(" cube "x" ")" "(*" x" "x" "x" ")")")

"(" define "(" cube "x" ")" "(*" x" "x" "x" ")")")

"(" define "(" square "20" ")")"
(define (cube x) (* x x x))

What is the magic in the list of tokens that makes parsing possible? It is, of course, the parentheses "(" and ")". The parentheses tell the parser when to collect tokens on the stack/list a— into a single subexpression (sublist). What about languages where you can write things like

```
while n>0 do c <- b ; b <- a+b ; a <- b
```

How does the parser know how to break up the expression into its constituent parts? This is the big problem of parsing—largely solved, but we don’t have time to talk more about it.

Parsing plays a big part in linguistics also: it’s how we know what phrases such as “Fruit flies like a banana” get their meaning. Does all fruit have the aerodynamic properties exemplified by the banana, or do fruit flies—those tiny insects—like eating bananas?