Homoiconic languages, self-parsing

- A language is homoiconic if code written in it also forms valid data under the language
- This means you can effectively “see” the internal representation just by looking at the source code
- Lisp is a good example, where you can see your source code as a lisp list, and your lisp code can read, manipulate, and generate lisp code
- Other homoiconic languages include scheme, racket, closure, mathematica, wolfram, julia, prolog, snobal, tcl, ...

Parsing lisp in lisp

- We'll build up a simple translator, that takes a list of lisp statements and builds a list of strings describing them
- We'll have a recursive function, interpretter, go through the list and translate one statement at a time (using function intepret1) and add the resulting string to a list
- In the beginning we'll just handle a few kinds of statements, but we could incrementally add support for more and more types

Parsing lisp in lisp

- Our top level instruction to go through the list of statements and build up a list of descriptions

```
(defun interpretter (statements)
   (cond
      ((not (lisp statements)) nil)
      ((null statements) nil)
      (t (cons (interpret1 (car statements)
                (interpret (cdr statements))))))
```

Interpretting a statement

- Our interpret1 function takes a single statement and generates the description string for it
- The function begins by looking at the data type for statement (is it a function, is it a number, is it a list, etc)
- If the statement is actually a list then we'll recursively analyze that
- As a first pass we'll simply return a string for the type of the statement (e.g. for a statement like (f x) it will just return "function call" as the description)
- Later we can replace the strings with function calls that build more accurate descriptions

interpret1

```
(defun intepret1 (statement)
   (typecase statement
      (function “function_call”)
      (number “numeric_value”)
      (string “text_string”)
      ; for lists, refer back to interpret to analyze contents
      (list (list “list_of “ (interpret statement)))
      ; add more cases to cope with more of language
      (t “something_else”)))
```

Trial run

- If we try interpret on `(25 “foo” t (interpret 10))` we get

```
(numeric_value text_string something_else
   (list_of (function_call numeric_value))))
```

- This is on the right track, but for a function call like `(interpret 10)` we might want it to say something like

```
   (function_call function_name numeric_val)
```

instead of

```
   (list_of (function_call numeric_value)
```

Tweak for functions

```
(defun intepret1 (statement)
   (typecase statement
      (number “numeric_value”)
      (string “text_string”)
      ; introduce special intepret function for lists
      (list (interpretList statement))
      ; add more cases to cope with more of language
      (t “something_else”)))
```

interpretList

- Check if it is a list or a function call

```
(defun interpretList (L)
   (cond
      ((not (listp L)) nil)
      ((null L) “empty list”)
      ; special handling of function calls
      ((typep (car L) ‘function)
          (list “func_call (car L) (interpret (cdr L)))
      ; regular handling of a data list
      (t (list “list_of (interpret L)))))
```

Trial run 2

- Try interpret on `(25 “foo” t (interpret 10))` again:

```
(numeric_value text_string something_else
   (func_call INTERPRET (numeric_value)))
```

- This is pretty close, though we might want to get rid of the brackets around INTERPRET’s parameter list, e.g. using

```
(append (list “func_call (car L)) (interpret (cdr L)))
```

- Instead of

```
(list “func_call (car L) (interpret (cdr L)))
```

Continuing on ...

- We can add parsing for more language features by expanding our typecase in interpret1, so that it calls a custom function for each different possible item type
- We could expand the intepretList to recognize key lisp keywords such as let, cond, if, etc where the function name appears, and call custom interpret routines for each
- We could add file handlers, to read the data from .cl files, and error handling etc
- Note the built in (read) function must be doing something like this already....