INTRODUCTION TO LISP FOR LANGUAGE ENGINEERS

alessio.stalla@strumenta.com

Strumenta Community Virtual Meetup, 2021-05-21

GOALS

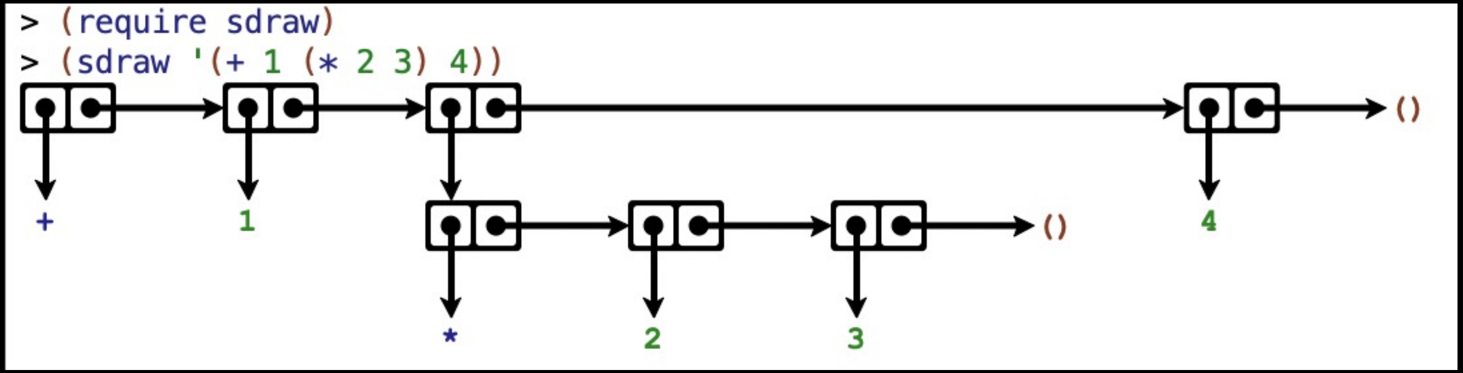
- Getting a glimpse of what metaprogramming techniques and DSL implementation techniques exist in the Lisp world
- Discussing parallels and differences with other technologies (most notably JetBrains MPS)

- Non-goals:
  - Teaching programming in Lisp
  - Giving a complete, in-depth picture
  - Saying everything (will leave something out)

LISP IS:

- A family of languages dating back to 1959
- A formal notation for computation (think lambda calculus or Turing machine), based on symbolic transformations
- Loosely, a paradigm that mixes compile-time metaprogramming, functional idioms and dynamic typing
    - Variants exist with optional static typing, imperative constructs, object systems, first-class continuations, etc.

LISP SOURCE CODE IS A TREE



```
> (require sdraw)
> (sdraw '(+ 1 (* 2 3) 4))
```

(image courtesy of DrRacket)

LISP SOURCE CODE IS A TREE

- Ofc it has a **canonical textual representation**, called **S-expressions**:
  - (+ 1 (* 2 3) 4) for the tree in the picture
- However, **S-expressions are NOT the source code and NOT the only possible form of the source code**
- S-expressions are **only a convenient representation** that has stuck basically unchanged from 1959, despite several attempts at other, «friendlier» syntaxes (M-expressions),
  - See also the Apple Dylan language (now OpenDylan)
- Kind of like XML is not the source code in MPS

…ACTUALLY, IT'S A GRAPH

- Internal nodes are **«cons cells»** or conses
    - mutable pairs of pointers, historically "car" and "cdr", or head/tail, first/rest, …
- Leaves are **atoms** (symbols, numbers, strings, NIL, …)
- Nodes can and do have pointers to other nodes to create a list, tree, or graph structure, as seen in the picture
    - Note that «atoms» can actually be composite objects (e.g. structs), but for the purpose of our definition, only the structure created with conses counts

EVAL: TREE → TREE (+ SIDE-EFFECTS)

- Most basic Lisp implementation: the eval function
- Eval assigns a meaning (with side-effects) to trees
- **Signature is tree → tree, NOT string → any!!** (As it is, e.g., in JS)
- Note: «tree» = atom or cons, i.e., any possible Lisp datum
  - But not all trees have a valid meaning, i.e. eval is a partial function that errors on malformed code

EVAL: TREE → TREE (+ SIDE-EFFECTS)

- eval(atom) => ?
- If atom is the symbol *S*: value of the variable named *S*
- Otherwise, self-evaluating:
    - eval(3) => 3
    - eval("hi") => "hi"
    - eval(NIL) => NIL
    - and so on

EVAL: TREE → TREE (+ SIDE-EFFECTS)

- eval(cons) => ?
- head is operator
- tail are arguments
- operator can be:
  - **function**, e.g. (+ 1 2 3) => 6
  - **special operator**, e.g. (lambda (x) (+ 1 x)) =>
    #<FUNCTION (LAMBDA (X)) {22661B4B}>
  - for functions, (f a b …) = (apply f (eval a) (eval b) …) **at runtime**
  - special operators receive unevaluated arguments **at compile time**

METACIRCULARITY

- The language defined by eval can be used (and HAS been used) to implement an eval function
- Therefore, eval is both the Lisp interpreter AND a Lisp function
- Of course, it can’t be «turtles all the way down» – at some point, someone will have written a lower-level eval function in C, ASM, Java, …
  - …or in another Lisp that compiles or cross-compiles to the target machine (common case)

SAME FOR COMPILE…

- compile: tree → tree is a Lisp function as well, and it's written in Lisp
- It doesn't evaluate its input, it only transforms it into a form which is closer to the machine – even down to machine code
- Example…
- compile is not special: we can write other functions that process trees

MACROS

- User-defined special operators (in Lisp)
- I.e., compile-time **tree-to-tree transformations**
- I.e., **language is extensible** and **extension lang same as target lang**
  - No text-based preprocessor (e.g., C)
  - No special metaprogramming language (e.g., C++ templates)
- I.e., **language can be brought closer to the domain** while keeping the same toolchain
- Macro are **composable:** if the tree produced by a macro contains calls to other macros, they're expanded recursively

QUOTATION

- How to construct the expression (f x y)?
- (list f x y) => error: unknown variable f
- list is a function, so (list f x y) = (apply list (eval f) (eval x) (eval y))
- How to refer to *the symbol F* rather than *the value of the variable F*?
- I.e., how to prevent evaluation?
- **Quote special operator**: (eval (quote x)) => x for any x
- Quote so important that it has its special syntax: ‘x = (quote x)

QUASI-QUOTATION

- (defmacro dummy () (list ‘f ’x ‘y))
- (defmacro too-dumb (f x y) (list ‘if (list ‘> (list f x 3) 0) (list + y 4) "uh?"))
- Constructing non-trivial code by means of list & co. is unreadable
- Enter quasiquote (template mini-language)
- (defmacro better (f x y) `(if (> (,f ,x 3) 0) (+ ,y 4) "oooh…!"))

NOW YOU KNOW EVERYTHING!

- Time for some questions before we go on

WHAT WE CAN DO WITH MACROS

- Resource management (e.g., with-open-file)
- Precompute expensive stuff at compile-time
- Declarative programming

WHAT WE CAN DO WITH MACROS

- CLOS (Common Lisp Object System) OOP on top of a functional-imperative language
- Prolog in Lisp (Norvig’s PAIP and Allegro Prolog)
- ACL2 modelling language and theorem prover
- Parenscript, Lisp-to-JS transpiler in ~4kloc
- Introducing language support for concurrency, continuations, FFI, …
- …i.e., what we can do on top of BaseLanguage in MPS, more or less

MACRO PITFALLS

(Unintentional) variable capture:

```
(defmacro foo (a b &body body)
  `(let ((intermediate-result ,(combine a b)))
    ,@body))
(let ((intermediate-result 42))
  (foo a b
    (print intermediate-result))) ;Does NOT print 42
```

MACRO PITFALLS

- (Unintentional) variable capture, solutions:
  - Manually ensure unique symbols with (gensym)
  - Hygienic macro systems in Scheme (disallow variable capture)
    - Particularly important because Scheme is a Lisp-1
- However, sometimes you *want* variable capture, in macros that introduce a local context, e.g.

```
(defmethod foo (…)
  (call-next-method)) ;this is like super.foo(…)
```

- Defmethod is a macro!

MACRO PITFALLS

- Lisp-1 vs Lisp-2

(let ((**list** …))
  (**list** …))

- Are the two «list» the same thing?
- I.e., is there a single namespace for functions and variables, or are those separate?
- Lisp-1 makes functional programming easier but metaprogramming harder
- Lisp-2 is more verbose – (apply #'f …) rather than (f …) when f is a variable
- Note that this is not specific to Lisp, however it interacts with macros

MACRO PITFALLS

- Phases of evaluation:

```
(defun foo (…) …)
(defmacro bar (…)
  (foo …))
```

- Foo is only available at runtime, while bar needs it at compile-time
- Solutions:
  - Put foo in another file that is loaded before the one where bar is defined
  - Use eval-when or similar to augment the compilation environment with foo

MACRO PITFALLS/ADVANCED

- Code walker: when a macro needs to analyze source code
  - E.g., finding free variables in its body
- The macro function must know what to expand in order to analyze
  - E.g. in (let ((foo 1)) (foo 2)), the first foo is a local variable declaration, the second one is an expression and foo may be a macro
  - In general, **some special operators are implementation-specific** and the macro doesn’t know about them
    - In (sys:%some-special-thing (foo bar)), is (foo bar) a macro invocation or not?
- As far as I know, no portable, universal code walker exists, at least in Common Lisp

FURTHER DISCUSSION

- Interactions with the type system
- Debugging
- Compiler macros (user-defined compiler optimization strategies)
- Symbol macros
    - (with-slots (name surname) (make-instance ‘person)
      (list name surname))
- …

THANKS!