Language-Oriented Programming

Principles of Programming Languages

Colorado School of Mines

`https://lambda.mines.edu`

Learning Group Activity

Share with your group the macro you made for class today. Explain how it works, and when you might use it.

Why couldn’t you use a function instead of a macro for the scenario you selected?

What is LOP?

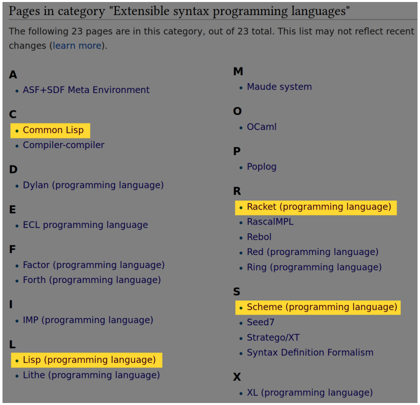
Language-Oriented Programming is a programming paradigm where you either:

- Extend an existing programming language to create the syntax needed to solve your problem elegantly (**extensible programming language**)
- Or, create a new **domain specific language** for solving your problem

Programming languages with *homoiconic syntax* and *macros* have historically been very good at LOP.

Extensible Syntax

- The domain of extensible languages is well dominated by Lisp, with the others sharing in common homoiconic syntax and macros.
- The exception to the above is concatenative languages, like Forth.



Pages in category "Extensible syntax programming languages"

The following 23 pages are in this category, out of 23 total. This list may not reflect recent changes (learn more).

A
- ASF+SDF Meta Environment

C
- Common Lisp
- Compiler-compiler

D
- Dylan (programming language)

E
- ECL programming language

F
- Factor (programming language)
- Forth (programming language)

I
- IMP (programming language)

L
- Lisp (programming language)
- Lithe (programming language)

M
- Maude system

O
- OCaml

P
- Poplog

R
- Racket (programming language)
- RascalMPL
- Rebol
- Red (programming language)
- Ring (programming language)

S
- Scheme (programming language)
- Seed7
- Stratego/XT
- Syntax Definition Formalism

X
- XL (programming language)

Domain Specific Languages

Domain Specific Languages are languages *tailored to solve a specific kind of problem*. For example, the **object property definition language** is created for defining properties about types of data:

```
(type house
  :bases (building living-space)
  :nouns ("house" "home"))
```

- More examples: HTML, CSS, Glade (GUI), Sieve (mail filtering), Regular Expressions ...
- DSLs can be either intended to stand-alone in their own files, or to be used inline in other languages. In the latter case, we often call them **domain specific mini-languages**.

A Domain Specific Mini-Language you already know

`match` is a DSL built into Racket… in fact, it just translates into a bunch of `cond`s and `let`s:

```
> (syntax->datum (expand-once '(match a [(list-rest b c) b] [_ a])))
'(let ((a1 a))
   (let ((fail2
          (λ ()
            (match:error a1 (syntax-srclocs (quote-syntax srcloc)) 'match)
     (let* ((f3
             (lambda ()
               (syntax-parameterize
                ((fail (make-rename-transformer (quote-syntax fail2))))
                (let () a)))))
       (cond
        ((pair? a1)
         (let ((unsafe-car6 (unsafe-car a1)) (unsafe-cdr7 (unsafe-cdr a1))
           (syntax-parameterize
            ((fail (make-rename-transformer (quote-syntax f3))))
            (let ((c unsafe-cdr7)) (let ((b unsafe-car6)) (let () b))))))
```

Why DSLs?

- Domain-specific syntax can eliminate repetitive code
- Domain-specific syntax can make it easier to express certain concepts
- Can be restricted, which allows us to prove certain things while compiling:
  - **Example:** regular expressions are a DSL which can be translated to finite state machines, which we can prove certain properties about

DSLs in Racket

There's two things that go into a `#lang` in Racket:

- A **reader module**, which parses a custom syntax to (Racket) s-expression syntax
- A **expander module**, which provides the macros and functions in the language.

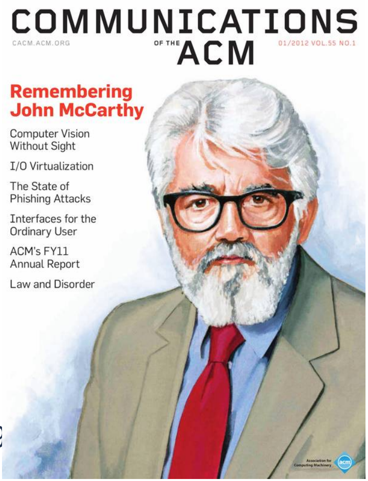
Custom Reader Optional

Many DSLs just use s-expression syntax, as it's easy and usually expressive enough for most applications. Racket comes with the `s-exp` reader which provides you with exactly this functionality.

Let's make a DSL!

- For the second part of lecture, I will be covering an example implementation of a DSL in detail.
- The DSL shown in class today is derived from one published in **Volume 55, Issue 1** of the **CACM** by **Matthew Flatt**:

`doi: 10.1145/2063176.206319`


COMMUNICATIONS OF THE ACM
CACM.ACM.ORG
01/2012 VOL.55 NO.1
Remembering John McCarthy
Computer Vision Without Sight
I/O Virtualization
The State of Phishing Attacks
Interfaces for the Ordinary User
ACM's FY11 Annual Report
Law and Disorder
Association for Computing Machinery

Motivation

Text-based adventures are some of the earliest computer games. They gained quite a lot of popularity in the 1970s and 1980s:

```
You're standing in a meadow.
There is a house to the north.
> north
You are standing in front of a house.
There is a door here.
> open door
The door is locked.
>
```

Anyone who has written a text-based adventure in a general purpose language can tell you they often result in a load of spaghetti code. Let’s clean that up.

Conceptual Model

In order to define a DSL for text-based adventures, we must define a model which text-based adventures follow. This is a critical part of designing any DSL:

Items:	*Items have a state and the user can store them in their inventory.*
Verbs:	*Verbs conduct an action on an item or the place. Verbs can have multiple names (e.g.,* `north` *and* `n`*)*
Places:	*Places have a description, items, and verbs which can move to other places.*

Storing our Model in Racket

Using structs makes for an easy way to store objects in our model:

```
(struct verb (aliases              ; list of names
              desc                 ; string
              thing?))             ; does it take an item?
(struct item (name                 ; symbol
              [state #:mutable]    ; state of item
              actions))            ; list of verb -> function co
(struct place (desc                ; string
               [items #:mutable] ; list of items
               actions))           ; list of verb -> function co
```

Making Syntax Easier

We can't expect our users of our DSL to be using our structs directly, let's make easy syntaxes to define them:

- `define-verbs`: Define a list of verb aliases to their corresponding verb structs, additionally providing a name to refer to a list of all of the verbs
- `define-item`: Define an item, specifying the verbs associated and what they do.
- `define-place`: Define a place, specifying the verbs associated and what they do.

define-verbs Example

```
(define-verbs all-verbs
  [(north n) "go north"]
  [(south s) "go south"]
  [(east e) "go east"]
  [(west w) "go west"]
  [(up) "go up"]
  [(down) "go down"]
  [(in enter) "enter"]
  [(out leave) "leave"]
  [(get grab take) thing "take"]
  [(put drop) thing "drop"]
  [(open unlock) thing "open"]
  [(close lock) thing "close"]
  [(knock) thing "knock"])
```

Implementing define-verbs

```
(define-syntax-rule (define-verbs all-id
                     [(id aliases ...) spec ...] ...)
  (begin
    (define-one-verb (id aliases ...) spec ...) ...
    (define all-id (list id ...))))

(define-syntax define-one-verb
  (syntax-rules (thing)
    [(_ (id ...) desc)
     (begin
       (define id (verb (list 'id ...) desc #f))
       ...)]
    [(_ (id ...) thing desc)
     (begin
       (define id (verb (list 'id ...) desc #t))
       ...)]))
```

define-item Example

```
(define-item door 'closed
  [open (if (have-item? key)
            (begin
              (set-item-state! door 'open)
              "You use the key to unlock and open the door.")
            "The door is locked.")]
  [close (set-item-state! door 'closed)
         "The door is now closed."]
  [knock "No one is home."])
```

define-item Implementation

```
(define-syntax-rule (define-item id
                       start-state
                       [vrb expr exprs ...] ...)
  (define id
    (item
      'id
      start-state
      (list (cons vrb (λ () expr exprs ...)) ...))))
```

define-place Example

```
(define-place house-front
  "You are standing in front of a house."
  (door)
  ([in (if (eq? (item-state door) 'open)
         room
         "The door is not open.")]
   [south meadow]))
```

Implementation is very similar to `define-item`.

Game Logic & Demo

- Game logic omitted from slides, as not super relevant to the DSL (available on course site)
- Demo game!

Announcements

- Today is last lecture
- Thursday (11/15) is optional lab day held in ALAMODE
- No class or office hours Tuesday (11/20) due to Thanksgiving Break
- Tuesday (11/27) is optional lab/work day (ALAMODE)
- Presentations 11/29, 12/4, 12/6