What defines a language?

- syntax
- semantics
- libraries
- idioms

What defines a language?

- syntax
- semantics
- libraries
- idioms

Side-note:
E.W. DIJKSTRA
"Goto Statement Considered Harmful."
This paper tries to convince us that the well-known goto statement should be eliminated from our programming languages or, at least (since I don't think that it will ever be eliminated), that programmers should not use it. It is not clear what should replace it. The paper doesn't explain to us what would be the use of the "if" statement without a "goto" to redirect the flow of execution: Should all our postconditions consist of a single statement, or should we only use the arithmetic "if," which doesn't contain the offensive "goto"?
And how will one deal with the case in which, having reached the end of an alternative, the program needs to continue the execution somewhere else?
The author is a proponent of the so-called "structured programming" style, in which, if I get it right, gotos are replaced by indentation. Structured programming is a nice academic exercise, which works well for small examples, but I doubt that any real-world program will ever be written in such a style. More than 10 years of industrial experience with Fortran have proved conclusively to everybody concerned that, in the real world, the goto is useful and necessary: its presence might cause some inconveniences in debugging, but it is a de facto standard and we must live with it. It will take more than the academic elucubrations of a purist to remove it from our languages.
Publishing this would waste valuable paper: Should it be published, I am as sure it will go uncited and unnoticed as I am confident that, 30 years from now, the goto will still be alive and well and used as widely as it is today.
Confidential comments to the editor: The author should withdraw the paper and submit it someplace where it will not be peer reviewed. A letter to the editor would be a perfect choice: Nobody will notice it there!

How important is each of these?

- libraries give you the run-time support, not an important part of the language itself.
- idioms originate from both language design and **culture**. Which of the two JavaScript samples is idiomatic?

1

```
if (isExplorer)
  document.onmousemove = function () { ... };
else
  document.onmousemove = function () { ... };
```

```
function foo(n) {
  return function(m) { return m+n; };
}
```

How important is each of these?

- libraries give you the run-time support, not an important part of the language itself.
- idioms originate from both language design and **culture**. Which of the two JavaScript samples is idiomatic?

```
if (isExplorer)
  document.onmousemove = function () { ... };
else
  document.onmousemove = function () { ... };

function foo(n) {
  return function(m) { return m+n; };
}
```

1. the line between "a library" and "part of the language" is less obvious than it seems.

Syntax versus semantics

- Syntax is mostly in the cosmetics department.
- Semantics is the real thing.
- Suppose **a** is an array with 3 elements.

2

```
a[25]+5                          (Java: exception)
(+ (vector-ref a 25) 5)          (Racket: exception)
a[25]+5                          (JavaScript:
                                   NaN or undefined)
a[25]+5                          (Python:exception)
$a[25]+5                         (Perl: 5)
a[25]+5                          (C: Undefined
                                     Behaviour)
```

More Syntax vs. Semantics

- Consider the difference between "murder" the word and the action it names.
- The evaluation function that Racket uses is actually a function that takes a piece of syntax and returns (or executes) its semantics.

How should we talk about semantics?

- A few well-known formalisms for semantics.
- We will use programs to explain semantics: the best explanation **is** a program.
- Ignore possible philosophical issues with circularity (but be aware of them).

1. Actually, they are solved: Scheme has a formal explanation that can be taken as a translation from Scheme to logic, which means that things that we write can be translated to logic.

Syntax and semantics in this course

3

```
#lang ragg

ae: NUMBER
  | ae "+" ae
  | ae "-" ae
; <<driver>>
```

4

```
(define (eval expr)
  (type-case AE expr
    [(Num n)   n]
    [(Add l r) (+ (eval l) (eval r))]
    [(Sub l r) (- (eval l) (eval r))]))

(eval (Add (Num 1) (Num 2)))
```

Special forms

- There are certain things that are part of Racket's syntax – for example 'if' and 'define' are special forms, they do not have a value! More about this shortly.
- Bottom line: much more things do have a value, compared with other languages.

Cond versus If

Don't use deeply nested if's

```
5  (define (digit-num n)
     (if (<= n 9)   1
     (if (<= n 99)   2
     (if (<= n 999)   3
     (if (<= n 9999)   4
      (error 'digit-num "much digits"))))))
```

- Instead, we can use Racket's 'cond' statement, like this:

6

```
(define (digit-num n)
  (cond [(<= n 9)    1]
        [(<= n 99)   2]
        [(<= n 999)  3]
        [(<= n 9999) 4]
        [else (error 'digit-num "much
           digits")]))
```

- Instead, we can use Racket's 'cond' statement, like this:

```
(define (digit-num n)
  (cond [(<= n 9)    1]
        [(<= n 99)   2]
        [(<= n 999)  3]
        [(<= n 9999) 4]
        [else (error 'digit-num "much
           digits")]))
```

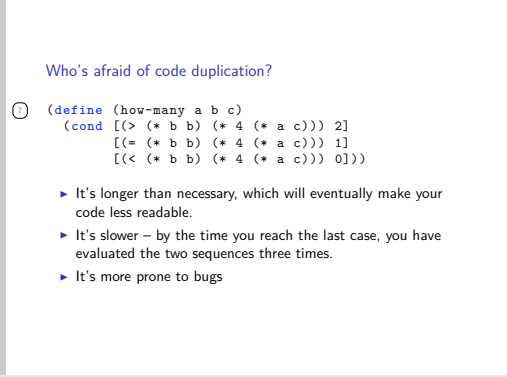
1. Note that 'else' is a keyword that is used by the 'cond' form – you should always use an 'else' clause (for similar reasons as an 'if', to avoid an extra expression evaluation there, and we will need it to pacify the typechecker.
2. Also note that square brackets are read by DrRacket like round parens, it will only make sure that the paren pairs match. We use this to make code more readable – specifically, there is a major difference between the above use of "[]" from the conventional use of "()". Can you see what it is?

Who's afraid of code duplication?

7

```
(define (how-many a b c)
  (cond [(> (* b b) (* 4 (* a c))) 2]
        [(= (* b b) (* 4 (* a c))) 1]
        [(< (* b b) (* 4 (* a c))) 0]))
```

- It's longer than necessary, which will eventually make your code less readable.
- It's slower – by the time you reach the last case, you have evaluated the two sequences three times.
- It's more prone to bugs



1. the above code is short enough, but what if it was longer so you don't see the three occurrences on the same page? Will you remember to fix all places when you debug the code months after it was written?

- In general, the ability to use names is probably the most fundamental concept in computer science – defining "high level languages"
- We already have a facility to name values: function arguments. We could split the above function into two like this:


```
; note the identifier names
(define (how-many-helper b^2 4ac)
  (cond [(> b^2 4ac) 2]
        [(= b^2 4ac) 1]
        [else        0]))

(define (how-many a b c)
  (how-many-helper (* b b) (* 4 (* a c))))
```

Local names

- Recall the syntax for a ‘let’ special form is

9
```
(let ([id expr] ...) expr)
```

For example,

10
```
(let ([x 1] [y 2]) (+ x y))
```

- But note that the bindings are done ”in parallel”, for example, try this:

11
```
(let ([x 1] [y 2])
  (let ([x y] [y x])
    (list x y)))
```

Local names

- Recall the syntax for a 'let' special form is

```
(let ([id expr] ...) expr)
```

For example,

```
(let ([x 1] [y 2]) (+ x y))
```

- But note that the bindings are done "in parallel", for example, try this:

```
(let ([x 1] [y 2])
  (let ([x y] [y x])
    (list x y)))
```

1. Instead of the awkward solution of coming up with a new function just for its names, we have a facility to bind local names – 'let'.

Using let for the above problem:

12

```
(define (how-many a b c)
  (let ([b^2 (* b b)]
        [4ac (* 4 (* a c))])
    (cond [(> b^2 4ac) 2]
          [(= b^2 4ac) 1]
          [else        0])))
```

Some notes on writing code

- See also the course web page(s).
- Code quality will be graded to in this course!
- Use abstractions whenever possible, as said above. This is bad:

13

```
(define (what-kind a b c)
  (cond
    ((= a 0) 'degenerate)
    ((> (* b b) (* 4 (* a c))) 'two)
    ((= (* b b) (* 4 (* a c))) 'one)
    ((< (* b b) (* 4 (* a c))) 'none)))
```

- Don't over abstract: (define one 1) (define two "two")
- Always do test cases (show coverage tool),
- Do not under-document, but also don't over-document.
- INDENTATION! (Let DrRacket/emacs decide for you, and get used to its rules)
- As a general rule, 'if' should be either all on one line, or the condition on the first and each consequent on a separate line.


```
(if impending-doom
    (panic)
    (chill))

(if impending-doom (panic) (chill))
```

- Don't over abstract: (define one 1) (define two "two")
- Always do test cases (show coverage tool),
- Do not under-document, but also don't over-document.
- INDENTATION! (Let DrRacket/emacs decide for you, and get used to its rules)
- As a general rule, 'if' should be either all on one line, or the condition on the first and each consequent on a separate line.

14

```
(if impending-doom
    (panic)
    (chill))

(if impending-doom (panic) (chill))
```

1. The indentation is part of the culture that was mentioned before, but it's done this way for good reason: decades of programming experience have shown this to be the most readable format.

- Similarly for ‘define’ – either all on one line or a newline after the object that is being define (either an identifier or a an identifier with arguments).


```
(define impending-doom #t)

(define impending-doom
  (if (and (time-is-late)
           (not (assignment-done)))))
```

- Another general rule: you should never have white space after an open-paren, or before a close paren (white space includes newlines).

16

```
( ( ( don't do this
)
)
)
```

How many style issues can you find?

17

```
(define (how-many a b c)
  (cond ((> (* b b) (* (* 4 a) c))
         2)
        ((< (* b b) (* (* 4 a) c))
         0)
        (else
         1)))

(define (what-kind a b c)
  (if (equal? a 0) 'degenerate
      (if (equal? (how-many a b c) 0) 'zero
          (if (equal? (how-many a b c) 1) 'one
              'two)
          )
      )
  )
```

18

```
(define (interest deposit)
  (cond
  [(< deposit 0) (error 'interest "invalid
     deposit")]
  [(and (>= deposit 0) (<= deposit 1000)) (*
     deposit 1.04) ]
  [(and (> deposit 1000) (<= deposit 5000)) (*
     deposit 1.045)]
  [(> deposit 5000) (* deposit 1.05)]))
```

Recursion and tail recursion

Run this under the debugger in DrRacket. How deep does the stack get?

19

```
(define (fact n)
  (if (zero? n)
    1
    (* n (fact (- n 1)))))

(fact 10)
```

Compare to this version

20

```
(define (fact n)
  ; invariant: m! * acc = n!
  (local [(define (helper m acc)
            (if (zero? m)
                acc
                (helper (- m 1) (* acc m))))]
    (helper n 1)))

(fact 10)
```

Compare to this version

```
(define (fact n)
  ; invariant: m! * acc = n!
  (local [(define (helper m acc)
            (if (zero? m)
                acc
                (helper (- m 1) (* acc m))))]
    (helper n 1)))

(fact 10)
```

1. Try the following tools in DrRacket: syntax-checker, stepper, submission tool (installing, registering and submitting)

Lists and Recursion

21

```
(define (list-length-helper list len)
    (if (empty? list) len
        (list-length-helper (rest list)
                            (+ len 1))))

(define (list-length list)
  (list-length-helper list 0))
```

Reversing a list

22

```
(define (reverse lst)
  (if (empty? lst)
      null
      (append (reverse (rest lst))
              (list (first lst)))))
```

Is this tail recursive?

Making reverse tail recursive

23

```
(define (reverse in-list)
  ; What is the invariant?
  (letrec ([rev (λ (lst acc)
                  (cond
                    [(empty? lst) acc]
                    [else (rev (rest lst)
                               (cons (first lst)
                                     acc))]))])
    (rev in-list empty)))
```

Tests

24

```
(define (reverse lst)
  (letrec ([rev (λ (lst acc)
                  (cond
                    [(empty? lst) acc]
                    [else (rev (rest lst)
                               (cons (first lst)
                                     acc))]))])
    (rev lst empty)))

(module+ test
  (test (reverse empty) empty)
  (test (reverse (list 1 2 3)) (list 3 2 1)))
```