# Programming Languages

## Syntactic Sugar

Dr Russ Ross

Utah Tech University—Department of Computing

Fall 2024

# How SMoL Becomes Large

So far we have seen SImPl, the Standard Implementation Plan:

- Represent the program's syntax as abstract syntax using a recursive algebraic datatype
- Write a similar recursive program to process it
  - An interpreter that produces *values*
  - A compiler that produces *programs*
  - A type-checker that produces *judgements about type-correctness*
  - etc.

In practice, SImPl needs a case for each construct of the language, which is onerous in practice. Even two similar constructs can mean twice the implementation code, twice the *maintenance*, twice the bug fixes, etc.

# Redundancy in Languages

Consider a `for` loop in C:

```
for (x = 0; x < 10; x++) {
    sum += x;
}
```

This is *exactly* the same as

```
x = 0;
while (x < 10) {
    sum += x;
    x++;
}
```

# Redundancy in Languages

There is a general pattern:

```
for (INITIAL; CONDITIONAL; UPDATE) {
    sum += x;
}
```

is the same as

```
INITIAL;
while (CONDITIONAL) {
    sum += x;
    UPDATE;
}
```

# Redundancy in Languages

Imagine you are writing an interpreter for this. The `while` loop's implementation must

- make several recursive calls
- iterate, check, and perhaps perform some other bookkeeping
- maybe even manage temporary scope extensions

All of that work has to be *duplicated* for `for`! Wouldn't it be simpler to instead implement it just once and translate the `for` body into a `while` body?

Why have both at all?

- Each is convenience for different purposes
- Using only `while` would make it harder to pick out certain stylistic uses of `while` from `for`-style uses
- It adds to our vocabulary as programmers

We would like the convenience and richer vocabulary without the added pain of implementation.

# Desugaring

We make a distinction between

- The *core* language, whose constructs are all handled directly
- The *surface* language, which is translated into the core language

The extra constructs in the surface language make it "sweeter" to program and are called *syntactic sugar*.

The program that translates surface programs down to core is called a *desugarer* because it removes sugar.

Note: the desugarer is really a kind of compiler as it translates one language into another, but it is a special case since it translates into a sublanguage of the original. It is useful to have a special term to distinguish it from general compilation.

# Desugaring

Aside: In a real implementation, this compilation requires a little more care. Suppose you make an error using for, but the error was reported in terms of while: you'd be pretty confused, because you never did type the while. As a special case, you may be a student who doesn't even know what while is! Modern desugaring systems, such as that in Racket, have special support to take care of this in most of the common cases.

# Desugaring

The are many desugarings in real languages:

- `and` and `or` can desugar into nested `if`s
- In JavaScript, `o.x` desugars in `o["x"]`
- In many languages, `x += y` is sugar for `x = x + y`
- In Python, `+` desugars into the method `__add__`. Python has many of these desugarings; these methods are called "dunder" methods (short for double-underscore)
- Haskell, Python (and others) have list comprehensions, which desugar into function and method calls
- etc.

Desugaring is everywhere in programming. If you don't notice it, that's part of the point: it feels like you're working with larger surface syntax than the implementor has to manage.

# Desugaring

There are many ways desugaring can be implemented

- Parse the program normally, then rewrite the AST into a subset of the same AST
- In some languages (especially those with parenthetical syntax) there are two levels of parsing:
  - The coarser parenthetical level
  - The finer level of ASTs

  We can perform rewriting on the parenthetical terms and the internal AST never needs to know about the sugars, i.e., it only need cover the core language

These latter are sometimes called *macro* systems: systems in which program source (slightly abstracted) is rewritten into program source, before parsing takes place.

*Most* languages have syntactic sugar, but *very few* have macro systems. The distinction is that macro systems give program rewriting capabilities directly to the programmer, while regular desugaring is hidden inside the compiler.

# Macros By Example

Racket has a rich macro system. We have the full power of the Racket language available for macro processing and can write sophisticated systems. In essence, Racket macros compile an extended version of Racket down to regular Racket. The downside is that the macros do not port easily to other languages.

For these examples you should switch to using

```
#lang racket
```

# A New Conditional

Racket is a truthy/falsy language, where `if` takes any non-false value to be true. Suppose we want a strict `if` that only takes Booleans.

First try:

```
(define (strict-if C T E)
  (if (boolean? C)
      (if C T E)
      (error 'strict-if "expected a boolean")))
```

Try examples like:

```
(strict-if true 1 2)
(strict-if 0 1 2)
```

So far so good!

But this is not correct. Do you see the problem?

# A New Conditional

The problem is that we have an eager language, so `strict-if`s arguments are going to evaluated before the body begins to execute. The whole point of a conditional is to avoid evaluating one of the branches. Try

```
(strict-if true 1 (/ 1 0))
```

and compare with

```
(if true 1 (/ 1 0))
```

So we cannot use functions for this purpose. We need a mechanism that consumes the *syntax* and rewrites it, instead of letting it evaluate right away. These are macros.

# A New Conditional

Let's dive into how the macro is written. It's not so different from the function:

```
(define-syntax strict-if
  (syntax-rules ()
    [(strict-if C T E)
     (if (boolean? C)
         (if C T E)
         (error 'strict-if "expected a boolean"))]))
```

What are the pieces?

- `define-syntax` says we're defining a new piece of syntax (as opposed to a function)
- `syntax-rules` introduces a pattern-matcher (for now, ignore what the `()` means: but you do need to include it)
- Each rule, in brackets, is a pattern and output: if the input matches the pattern, then the desugarer (here called a *macro expander*) produces the corresponding output
- but with the *names* in the pattern (here `C`, `T`, and `E`) copied as program source into the output

# A New Conditional

The macro:

```
(define-syntax strict-if
  (syntax-rules ()
    [(strict-if C T E)
     (if (boolean? C)
         (if C T E)
         (error 'strict-if "expected a boolean"))]))
```

Given

```
(strict-if true 1 (/ 1 0))
```

the macro definition transforms it into

```
(if (boolean? true)
    (if true 1 (/ 1 0))
    (error 'strict-if "expected a boolean"))
```

which then evaluates exactly as we'd expect.

# A New Conditional

Racket has a Macro Stepper, which shows the program expanding step-by-step, which is useful both for understanding macros and debugging them. If necessary, change the "Macro hiding" option at the bottom-left to read "Standard".

Exercise: Try it out with the above macro definition and use. See what you get. Observe how, at each step, it highlights the macro use about to be expanded followed by the result of that expansion.

Note: The Macro Stepper is not an *evaluator*. It does not show the steps of evaluation, only the steps of expansion! Thus, if you write a program that will produce an error at run-time, the Macro Stepper does not show that error. It only shows *syntax* errors.

# Local Binding

Imagine we want to extend Racket with a `let1` construct. For example, we want

```
(let1 (x 3) (+ x x))
```

to evaluate to 6. Can `let1` be defined as a function? Why or why not?

# Local Binding

`let1` can't be a function. If it were, it would try to evaluate each of the sub-terms as arguments:

- `(x 3)`: this looks like an application, `x` isn't even bound, and there is no meaningful *value* it could produce
- `(+ x x)`: this cannot be evaluated until `x` has been bound, which is whole point of `let1`

`let1` is a new piece of syntax. Even if we could achieve the functionality we want using a function call, we would have to abandon the syntax we are trying to create, and the syntax is the whole point.

**Terminology:** We will often refer to these new pieces of syntax as constructs (as in, "a new language construct"). In the Lisp/Scheme/Racket community, these are sometimes also called special forms, because they are syntactic forms with their own special rules for binding and evaluation.

# Local Binding

We'll use the prefix `my-` on our macros so we do not clash with macros already built into Racket.

We can figure out the first part of the macro of `my-let1` based on what we've seen already:

```
(define-syntax my-let1
  (syntax-rules ()
    [(my-let1 (var val) body)
     …]))
```

But what would it expand into? We *could* just expand it into the existing `let` construct in Racket, but there's another interesting option.

# Local Binding

Think about what `my-let1` does:

- it *binds* a name to a value
- and then immediately *evaluates* its body
- in an environment extended by its name

Sound familiar? Functions:

- bind a formal parameter name to an argument value
- when applied to the argument, a function evaluates its body
- in an extended environment with the new binding

We can express `my-let1` in terms of an anonymous function that is applied immediately:

```
(define-syntax my-let1
  (syntax-rules ()
    [(my-let1 (var val) body)
     ((lambda (var) body) val)]))
```

# Local Binding

Sure enough:

```
(my-let1 (x 3) (+ x x))
```

will produce 6. Use the Macro Stepper to see how!

**Terminology:** This pattern, of an anonymous function that is used right away, is commonly called *left-left-lambda* (where "left" stands for left-parenthesis). For a long time this remained an obscure term in the Lisp/Scheme community. But JavaScript made this pattern popular again under the name *Immediately Invoked Function Expression* (IIFE), because of problems with the handling of scope in earlier versions of the language. If you think the parentheses look bad here, look up some examples of IIFE on the Web.

# Local Binding

Suppose we made a mistake in the macro and swapped two parts:

```
(define-syntax my-let1
  (syntax-rules ()
    [(my-let1 (var val) body)
     ((lambda (var) val) body)]))
```

What happens when we try to evaluate:

```
(my-let1 (x 3) (+ x x))
```

Use the Macro Stepper to see what happened.

# Binding More Locals

As we have noticed in Racket, however, the let can bind many names at once, not only one. It becomes clear how: the function takes formal arguments, and is applied to just as many actual arguments. There can be as many as we want! But how do we express this in macro syntax?

In mathematics, it's common to use ellipses (…) to denote a sequence of arbitrary length. Racket has the ability to write something like this:

```
(define-syntax my-let2
  (syntax-rules ()
    [(my-let2 ([var val] ...) body)
     ((lambda (var ...) body) val ...)]))
```

This says that `my-let2` is followed by any number of `var-val` pairs, followed by a body.

That turns into a `lambda` with all the `vars` as formal parameters, whose body is `body`, applied to all the same `vals` as the actual argument expressions.

We use it like this:

```
(my-let2 ([x 3] [y 4]) (+ x y))
```

Try the program above: run it, and also examine it in the Macro Stepper.

# Multi-Armed Conditionals

Here's another example that clarifies what ... means: it means "zero or more instances of the preceding pattern". Using it, we can define our own multi-armed conditional. Suppose we want to define a function called sign that produces a string based on the sign of a number:

```
(define (sign n)
  (my-cond
   [(< n 0) "negative"]
   [(= n 0) "zero"]
   [(> n 0) "positive"]))
```

Again it is clear that `my-cond` cannot be a function; we need to extend the language with a new construct using a macro.

How many arms should our multi-armed conditional have? As many as the programmer wants, of course. We'll further stipulate that if we have exhausted all the questions and none has yielded a true value, the "falling through" produces an error.

# Multi-Armed Conditionals

We want to

- peel off the first question-answer pair
- evaluate the question
- if it succeeds, evaluate the answer
- otherwise, recur on the remaining questions, essentially a smaller instance of `my-cond`
  - Yes, we are recurring on *syntax* now!

Since ... means "zero or more" we end up with a pattern where we repeat a pattern: the first copy peels off the first instance, while the second, followed by a ..., captures all the remaining instances:

```
(define-syntax my-cond
  (syntax-rules ()
    [(my-cond) (error 'my-cond "should not get here")]
    [(my-cond [q0 a0] [q1 a1] ...)
     (if q0
         a0
         (my-cond [q1 a1] ...))]))
```

**Exercise:** Examine this code in detail. Try out the example above. It's essential that you run this through the Macro Stepper: you'll learn a lot about macros from this example!

# A Definitional Convenience

Suppose we wanted to define a "one-armed `if`" (useful for check for errors), commonly called `unless`:

```
(define-syntax unless
  (syntax-rules ()
    [(_ cond body ...)
     (if (not cond)
         (begin
           body
           ...)
         (void))]))
```

We can use it like this:

```
(unless false
  (println 1)
  (println 2))
```

(Note the use of _ instead of repeating `unless`)

# Name Capture

But now, what if we use the above code in this kind of context:

```
(let ([not (lambda (v) v)])
  (unless false
    (println 1)
    (println 2)))
```

This seems problematic: it seems to expand into

```
(let ([not (lambda (v) v)])
  (if (not false)
      (begin
        (println 1)
        (println 2))
      (void)))
```

This looks kind of like dynamic scoping all over again: a detail of the macro definition coincidentally interfering with its use. Does it work?

# Name Capture

Running the macro shows that the name `not` is *not* being captured. Variables are more than just names, they record binding information, and Racket keeps them separate internally. We can see this in the macro stepper.

We started with:

```
(let ([not (lambda (v) v)])
  (unless false
    (println 1)
    (println 2)))
```

which expands into

```
(let ([not (lambda (v) v)])
  (if (not false)
      (begin
        (println 1)
        (println 2))
      (void)))
```

The `not` from the macro is effectively renamed internally so it does not conflict. This is called *hygiene*, and it is a critical feature for macros.

# A Truthy/Falsy Idiom

Unrelated to macros, here's a common idiom in truthy/falsy languages Consider a two-arm `or`, defined here as a macro:

```
(define-syntax or-2
  (syntax-rules ()
    [(_ e1 e2)
     (if e1
         true
         e2)]))
```

This works well enough for

```
(or-2 true false)
(or-2 false false)
(or-2 false true)
```

# A Truthy/Falsy Idiom

However, consider a function like `member`:

```
(member 'y '(x y z))
```

When it succeeds, it doesn't just return `true`, it returns the entire rest of the list (which is a truthy value). But if we combine this with `or-2`:

```
(or-2 (member 'y '(x y z)) "not found")
```

We do not get the desired result. To fix it we can redefine `or-2` as:

```
(define-syntax or-2
  (syntax-rules ()
    [(_ e1 e2)
     (if e1
         e1
         e2)]))
```

# A Macro Definition Peril

However, this macro contains a subtle peril. Consider:

```
(or-2 (print "hello") "not found")
```

That also returns a truthy value, but now we see the print twice. So we need

```
(define-syntax or-2
  (syntax-rules ()
    [(_ e1 e2)
     (let ([v e1])
       (if v v e2))]))
```

# Back to Hygiene

This now works fine for the printing example. But now we have to worry about

```
(let ([v 1])
  (or-2 false v))
```

Using fresh names, there are two things this could expand into:

```
(let ([v 1])
  (let ([v false])
    (if v
        v
        v)))

(let ([v0 1])
  (let ([v1 false])
    (if v1
        v1
        v0)))
```

The macro produces the latter version. In other words, hygiene works for local variables, not just built-in functions.

# Generalizing Macros

Finally, with prefix syntax we can generalize constructs to arbitrary *arity*:

```
(define-syntax orN
  (syntax-rules ()
    [(_ e1 e2 ...)
     (let ([v e1])
       (if v v (orN e2 ...)))]))
```

However, see what happens when we try:

```
(let ([v true])
  (orN false v))
```

# Generalizing Macros

Note the error message: `orN: bad syntax in: (orN)`

We need a base case. Our definition requires one or more sub-expressions: `e1` is the first, and `e2 ...` means *zero or more* from the second position onward. But nothing covers the case of no sub-terms. So we need:

```
(define-syntax orN
  (syntax-rules ()
    [(_) false]
    [(_ e1 e2 ...)
     (let ([v e1])
       (if v v (orN e2 ...)))]))
```

and this works fine.