Concepts of programming languages

Lecture 7

Wouter Swierstra

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Last time

- Relating evaluation and types
- How to handle variable binding in embedded languages?

DSLs: approaches

A stand-alone DSL typically has *limited expressivity* and requires writing a parser/interpreter from scratch.

An embedded DSL can re-use the host language's features, but is constrained by the host language's syntax and semantics.

Challenge

Suppose we want to have a variant of Markdown where we can have both computation and formatting.

```
@title{Fibonacci}

The first 5 fibonacci numbers are @fibs[5]
```

Stand-alone language

If we try to define our own Markdown flavour that allows you to mix computation in the layout…

Stand-alone language

If we try to define our own Markdown flavour that allows you to mix computation in the layout…

We end up needing to implement our own programming language.

Embedded approach

We can define a simple enough Haskell data type for representing Markdown:

```
data MDElt =
  Title Depth String
  | Bullets [MD]
  | Text String
  ...
type MD = [MDElt]
```

Embedded approach

But working with this language is awkward.

We need *some* computation, but we're mostly interested in writing strings and formatting them.

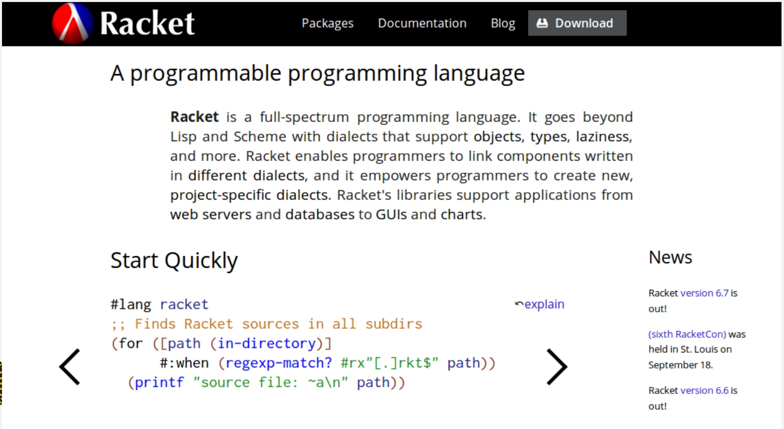
```
example =
  [Title 3 "Fibonacci numbers"
  , Text "The first five fibonacci numbers are"
  , Bullets (fibMD 5)]
  where
  fibMD :: Int -> MD
```

This really isn't user-friendly – we're mostly interested in **data** (strings), rather than **computations** (Haskell code).

Best of both worlds?

Today I want to present an alternative approach to embedding domain specific languages using macros and reflection.

But to do so, I want to introduce a new language **Racket**.



A programmable programming language

Racket is a full-spectrum programming language. It goes beyond Lisp and Scheme with dialects that support objects, types, laziness, and more. Racket enables programmers to link components written in different dialects, and it empowers programmers to create new, project-specific dialects. Racket's libraries support applications from web servers and databases to GUIs and charts.

Start Quickly

```
#lang racket
;; Finds Racket sources in all subdirs
(for ([path (in-directory)]
      #:when (regexp-match? #rx"[.]rkt$" path))
  (printf "source file: ~a\n" path))
```

News

Racket version 6.7 is out!

(sixth RacketCon) was held in St. Louis on September 18.

Racket version 6.6 is out!

Lisp

One of the oldest programming languages still used today is **Lisp**.

It dates back as far as 1958 and could be considered one of the earliest functional languages.

Originally Lisp was developed in the context of Artificial Intelligence research at MIT.

It pioneered a huge number of fundamental innovations to programming languages that we take for granted: tree data structures, automatic memory management, higher-order functions, and recursion...

Lisp dialects

- Common Lisp & Scheme – are the two most widely used dialects of Lisp. The two dialects forked as early as the 1970’s.
- Emacs Lisp – ever used the emacs text editor?
- Clojure – a Lisp implementation that runs the JVM.

Scheme & Racket

Scheme chose **lexical scoping** in favour of dynamic scoping and supported **tail call optimization**.

The philosophy has always been to define a *minimal* language, without too many bells and whistles.

As the language is quite small, there are numerous implementations – one of the most popular ones is *Racket*.

Programming in Racket

The syntax of Lisp-like families is designed to be simple.

Function calls are placed in parentheses; there are no infix operators, but only (prefix) function application:

```
> (+ 2 2)
4
> (= (+ 2 2) (* 2 2))
#t
> (and #t #t)
#t
> (and #t #f)
#f
> (0)
Error: application not a procedure...
```

Programming in Racket

The same syntactic style is used for defining constants, control flow, and functions:

```
> (define x 3)
> (+ x 1)
5
> (if (< x 4) 0 1)
0
> (define (double x) (* x 2))
> (double 3)
6
> (define triple (lambda (x) (* 3 x)))
> (triple 4)
12
```

Function application

Note that function application really requires explicit parentheses:

```
> (double 3)
6
> double 3
#<procedure:double>
3
```

An expression `(double 3)` calls the function `double` on the argument `3`.

Whereas `double 3` does not call the function.

Lists

```
(list 1 2 3)
> '(1 2 3)
(list)
> '()
```

What is the difference between `'(a b c)` and `(a b c)`?

- The first one is the list of three elements, `a`, `b`, and `c`.
- The second tries to apply the function `a` to the arguments `b` and `c`.

It is very important to make this distinction!

Programming with lists

Just as in any other language, there are plenty of functions to manipulate lists:

```
> (length (list 1 2 3))
3
> (reverse (list 1 2 3))
'(3 2 1)
> (map sqrt (list 1 4 9))
'(1 2 3)
> (first (list 1 2 3))
1
> (rest (list 1 2 3))
'(2 3)
```

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

‘Constructors’ and ‘pattern matching’

```
> empty
'()
> (cons 1 empty)
'(1)
> (empty? empty)
#t
> (empty? (cons 1 empty))
#f
```

Writing recursive functions

Question: What does foo compute?

```
(define (foo lst)
  (cond
   [(empty? lst) 0]
   [else (+ (first lst) (foo (rest lst)))]))
```

Square brackets `[x]` allow you to pass many different branches to `cond`; when you encounter a branch with a first expression evaluating to true, the result is the result of evaluating the remaining expressions.

Square brackets can also be used with let expressions to bind multiple values.

Pairs

Pairs are built up in a similar style to lists:

```
> (cons 1 2)
'(1 . 2)
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
> (cons 3 (list 1 2))
'(3 1 2)
> (cons (list 1 2) 3)
'((1 2) . 3)
```

Note: Racket is dynamically typed! You may get different results to what you might expect from other languages.

Syntax?

Why is Racket’s syntax so different from most other languages?

All code looks like some sort of list…

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Syntax?

Why is Racket's syntax so different from most other languages?

All code looks like some sort of list…

But this is on purpose!

Racket and other Lisp dialects keep the syntax simple, but make it easy to **manipulate** syntax.

Code is data

S-expressions

All programs in Lisp-like languages are built up from *S-expressions*:

- an atomic symbol, such as `3` is an S-expression.
- given two S-expressions `x` and `y` we can construct a binary node `(x . y)`.

There is some syntactic sugar, allowing you to write `(x y z)` rather than `(x . (y . (z . empty)))`.

Quotation

We can build a list using the `list` function:

```
> list (1 2 3)
'(1 2 3)
```

Alternatively, we can form the application `(1 2 3)` and turn this into an ‘abstract syntax tree’:

```
> (quote (1 2 3))
'(1 2 3)
```

Quotation

Quotation not only works for compound function applications, but we can also quote *symbols*:

```
> map
#<procedure:map>
> (quote map)
'map
> (symbol? map)
#f
> (symbol? (quote map))
#t
> (string->symbol "map")
'map
> (symbol->string (quote map))
"map"
```

More quotation

Quotation lets us **program** with the **syntax** of our language directly.

```
> (quote 12)
12
> (quote "Hello")
"Hello"
> (quote (road map))
'(road map)
> '(road map)
'(road map)
```

We can abbreviate `quote` with a prefix apostrophe `'`.

More quotation

What happens in the following example?

```
> (car (quote (road map)))
'road
```

More quotation

What happens in the following example?

```
> (car (quote (road map)))
'road
```

This shows how quoting a parenthesized sequence of identifiers automatically quotes the identifiers to create a list of symbols.

Eval: from data to code

The `quote` function lets us take an arbitrary piece of syntax and turn it into a list.

The dual function, `eval`, turns a list of values into the corresponding code.

```
> (+ 1 2)
3
> '(+ 1 2)
'(+ 1 2)
> (eval '(+ 1 2))
3
```

Metaprogramming

Question: What is the result of this expression?

```
(eval (rest '(1 + 2 3)))
```

Metaprogramming

Question: What is the result of this expression?

```
(eval (rest '(1 + 2 3)))
```

```
> 5
```

We can freely manipulate **code** as if it were **data**.

Metaprogramming

Question: What is the result of this expression?

```
(eval (rest '(1)))
```

Metaprogramming

Question: What is the result of this expression?

```
(eval (rest '(1)))
```

This throws an error when executed: we need to generate code from `'()`, which yields an illegal empty application.

Quasiquoting

So far we have two ways to move between code and data:

- the `quote` function produces data;
- the `eval` function evaluates data.

To mix the two, we'll introduce *quasiquoting*.

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Quasiquoting

Quasiquoting behaves just like `quote`…

```
> (quasiquote (0 1 2))
'(0 1 2)
```

Except if any of the terms being quoted contain `unquote`:

```
> (quasiquote (0 (unquote (+ 1 2)) 4))
'(0 3 4)
```

When the quasiquote encounters `(unquote expr)`, it evaluates `expr` and includes the result in the quoted term.

Quasiquoting – abbreviations

Quasiquote and unquote are abbreviated as ‘ (backtick) and , (comma) respectively.

```
> `(0 1 2)
'(0 1 2)
> `(1 ,(+ 1 2) 4)
'(1 3 4)
```

Back to variable binding

In the previous lectures, we saw how hard it was to handle variable binding in embedded languages.

We tried to reuse the host language's binding mechanism – with mixed results.

In a language like Racket – where we have quite some metaprogramming technology at our disposal – we can take a different approach.

Evaluation in a context

The `eval-formula` function evaluates its argument in an environment where `x` is 2 and `y` is 3:

```
> (define (eval-formula formula)
    (eval `(let ([x 2]
                 [y 3])
             ,formula)))

> (eval-formula '(+ x 3))
5
> (eval-formula '(* x y))
6
> (eval-formula '(* x z))
z: undefined. cannot reference an identifier
before its definition
```

Metaprogramming

What is `eval-formula` doing?

- it is passed a quoted expression;
- it builds up a new expression, wrapping its argument in a let binding;
- the result is then evaluated.

We're manipulating and constructing new syntax.

Doing so by hand is tedious – Racket's **macro** system makes this kind of thing much more practical.

Macros

A **macro** extends a language by specifying how to compile a new language feature into existing features.

Macros in C

```
#define foo "salad"

int main(){
  printf(foo);
```

The C preprocessor will substitute `"salad"` for `foo`.

Besides such substitutions, you can check for certain compile flags or the architecture that you're compiling on – and act accordingly.

Macros in C

```
#define foo "salad

int main(){
  printf(foo bar\n");
```

Unfortunately, these older versions of the C preprocessor do **not** preserve the lexical structure of C code.

Macros in C

```c
#define sqr(x) x*x

int main(){
  printf("%i",sqr(3));
}
```

Question: What is the result of running this program?

Macros in C

```
#define sqr(x) x*x

int main(){
  printf("%i",sqr(3));
}
```

Question: What is the result of running this program?

It prints 9 – after running the preprocessor, `sqr(3)` is expanded to `3 * 3`, which evaluates to 9.

Macros in C

```
#define sqr(x) x*x

int main(){
  printf("%i",sqr(3+2));
}
```

Question: What is the result of this program?

Macros in C

```
#define sqr(x) x*x

int main(){
  printf("%i",sqr(3+2));
}
```

Question: What is the result of this program?

`sqr(3+2)` expands to `3 + 2 * 3 + 2` or `3 + (2 * 3) + 2`, which reduces to `11`.

C Macros do not respect the expression level structure of our code.

This is an example of a not-so-very-clean macro system.

Macros in C

```
int x = 4;

#define incX x+1


int main(){
  int x = 5;
  printf("%i",incX);
}
```

Question: What is the result of running this program?

Macros in C

```
int x = 4;

#define incX x+1


int main(){
  int x = 5;
  printf("%i",incX);
}
```

Question: What is the result of running this program?

The macro `incX` expands to `x+1`, which evaluates to `6`.

The macro system does not respect the *lexical structure* of our programming language.

Macros in Racket

Macros in Racket are much more carefully designed.

They are careful to preserve the lexical structure and expression structure of Racket programs.

Let's explore a small case study.

Macro challenge

Define an operation `or` such that the call `(or e1 e2)` returns: * `e1` provided `e1` evaluates to a non-false value (and does not evaluate `e2`). * `e2` otherwise

We'll test our `or` operation using the following function:

```
(define (short-list? xs)
  (or (null? xs)
      (null? (rest xs))))
```

This function checks whether or not the list is empty or a singleton list.

Implementing `or`

As a first attempt, consider the following implementation of `or`:

```
(define (or x y)
  (if x x y))
```

Question: What is wrong with this definition?

Implementing `or`

As a first attempt, consider the following implementation of `or`:

```
(define (or x y)
  (if x x y))
```

Question: What is wrong with this definition?

The call `(short-list? empty)` fails dynamically.

Racket is strict: both arguments of `or` are evaluated before processing the body.

Hence `(or (null? empty) (null? (rest empty)))` fails because we are trying to take the tail of an empty list.

Implementing `or` using a macro

```
(define-syntax (or stx)
  (syntax-parse stx
    [(or x-exp y-exp)
    #' (if x-exp x-exp y-exp)]))
```

The `define-syntax` is used to define a macro.

In contrast to *functions* these macros are expanded at *compile time*.

Implementing `or` using a macro

```
(define-syntax (or stx)
  (syntax-parse stx
    [(or x-exp y-exp)
    #' (if x-exp x-exp y-exp)]))
```

Here we are defining a macro `or` that is a function.

It’s argument `stx` is a piece of syntax.

This is essentially the same as the quoted representation of the arguments to `or` – but contain additional information about source locations and variables that are in scope.

Implementing `or` using a macro

```
(define-syntax (or stx)
  (syntax-parse stx
    [(or x-exp y-exp)
    #' (if x-exp x-exp y-exp)]))
```

The `syntax-parse` function checks that we are applying the macro `or` to two arguments `x-exp` and `y-exp`.

If this check succeeds, we replace the macro with `if x-exp x-exp y-exp`.

Implementing `or` using a macro

```
(define-syntax (or stx)
  (syntax-parse stx
    [(or x-exp y-exp)
    #' (if x-exp x-exp y-exp)]))
```

Note that we need to quote `(if ...)` using `#'`.

The operator `#'` is similar to the `quote` operater `'` – the only difference is that it creates *syntax* (recording scope information) rather than 'just' S-expressions.

Expanding macros

At *compile time* any call to `or` is substituted:

```
(define (short-list? xs)
  (or (null? xs)
      (null? (rest xs))))
```

becomes:

```
(define (short-list? xs)
  (if (null? xs)
      (null? xs)
      (null? (rest xs))))
```

Expanding macros

```
(define (short-list? xs)
  (or (or (null? xs)
          (null? (rest xs)))
      (null? (rest (rest xs)))))
```

becomes:

```
(define (short-list? xs)
  (if (or (null? xs)
          (null? (rest xs)))
      (or (null? xs)
          (null? (rest xs)))
      (null? (rest (rest xs)))))
```

Expanding macros

```
(define (short-list? xs)
  (or (or (null? xs)
          (null? (rest xs)))
      (null? (rest (rest xs)))))
```

becomes:

```
(define (short-list? xs)
  (if (if (null? xs)
          (null? xs)
          (null? (rest xs)))
      (if (null? xs)
          (null? xs)
          (null? (rest xs)))
      (null? (rest (rest xs)))))
```

Duplicating code!

Our macro is **duplicating** code.

As we nest `or` expressions we are generating huge amounts of code.

To make matters worse, if the arguments of `or` have side-effects, such as mutating state, these can be run twice – which is not what we would expect.

Revisiting our macro

```
(define-syntax (or stx)
  (syntax-parse stx
    [(or x-exp y-exp)
    #' (let ([x x-exp]) (if x x y-exp))]))
```

We can avoid this duplication by introducing a local variable `x`.

Now we don’t need to worry about code duplication anymore.

Revisiting our macro

```
(define-syntax (or stx)
  (syntax-parse stx
    [(or x-exp y-exp)
    #' (let ([x x-exp]) (if x x y-exp))]))
```

We can avoid this duplication by introducing a local variable `x`.

Now we don't need to worry about code duplication anymore.

But we have a new problem to worry about…

Hygiene

Suppose we have the following variation of `short-list?`, that takes an argument `x`:

```
(define (short-list? x)
  (or (null? x)
      (null? (rest x))))
```

Naively expanding the macro yields:

```
(define (short-list? x)
  (let ([x (null? x)])
       (if x
           x
           (null? (rest x)))))
```

Question: What is the (naive) result of `(short-list? (list 1))`?

Hygiene

```
(define (short-list? x)
  (let ([x (null? x)])
       (if x x (null? (rest x)))))
```

The call `(short-list? (list 1))` expands to

```
(let ([x (null? (list 1))])
     (if x x (null? (rest x))))
```

```
(let ([x #f]) (if x x (null? (rest x))))
```

```
(let ([x #f]) (null? (rest x)))
```

which fails as we try to compute `rest #f`.

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

What went wrong?

The variable `x` was shadowed by our macro.

To avoid this, Racket uses so-called *hygienic macros* that guarantee that macros respect variable bindings upon expansion.

In our example, the `x` defined as the argument to `short-list?` is tagged to be at syntactic level 0; after expanding the macro, the let-bound `x` is at syntactic level 1.

The language is careful to distinguish the same identifier, living at different syntactic levels.

Hence our macro will work as expected.

Parsing Racket

The syntax of Racket is not defined directly in terms of strings, but is determined by two layers:

- a **reader layer** – turning a sequence of characters into lists, symbols, and other constants; and
- an **expander layer**, which processes the lists, symbols, and other constants to parse them as an expression.

The rules for printing and reading go together. For example, a list is printed with parentheses, and reading a pair of parentheses produces a list.

The macro system gives you quite a lot of power to customize this the expander layer; you can also customize the **reader**, which allows you to introduce your own surface syntax.

Beyond simple macros – Scribble

Racket lets you write entirely new programming languages based on this technology.

Each module can be written in its own ‘Racket-language’; many modules written in different languages can be tied together.

Scribble is Racket’s tool for writing documentation:

```
#lang scribble/base
@(require "fibs.rkt")

@title{On the Fibonacci numbers}

The @bold{Fibonacci} numbers are
amazing. The first few Fibonacci numbers are @fibs[10].
```

Beyond simple macros

There are many other dialects and languages implemented using Racket:

- Typed Racket adds static type safety to Racket;
- Lazy Racket introduces lazy evaluation;
- A DSL for writing slideshows;
- Java or Algol 60 as an embedded language;
- An object system.
- and many, many others…

Recap

- Racket is a strict, dynamically typed functional language of the Lisp/Sheme family.
- The syntax and language features are fairly minimalisticy.
- But this makes it very easy to manipulate *code* as *data*.
- *Macros* let us define new programming constructs in terms of existing ones.
- By customizing the parser even further, we can embed other languages in Racket or define our own Racket-dialects.

More Racket?

There are many, many interesting language features that I haven't talked about:

- dynamic correctness assertions using *contracts*;
- the philosophy of *How to design programs*;
- Bootstrap – Racket as a first programming language;
- Semantics engineering with PLT Redex.
- …

Further reading

These slides were loosely based on Robby Findler's talk 'Racket: A programming-language programming language' given at LambdaJam '15.

- *The Racket Manifesto* documents the vision behind Racket.
- *The Racket homepage* has a lot of links with more information about the language.