Introducing IfArith

Compiler Construction (CIS400) — Fall 2021

Syracuse University
Kris Micinski

Today's class

- **Project will be released this afternoon**
- Quoting, quasiquoting, and pattern matching
- Defining languages via patterns
- A recursive interpreter for a tiny language
- Expanding our tiny language to include let
- Compiling our small language to C
- Next time:
  - Deep dive into binary layout and organization
  - Full IfArith language
  - Hand-translation of IfArith programs to LLVM, hints on p1

S-exprs *(symbolic expressions)*

- The **S-expression** is our parenthesized notation for a list
  - Can use lists to group data common to some structure
- We can ***tag*** expressions with a symbol to note its "type"
  - `‘(point 2 3)`
  - `‘(square (point 0 1) 5)`
- Can define "constructor" functions

```
(define (mk-point x y)
  (list ‘point x y))

(define (mk-square pt0 len)
  (list ‘square pt0 len))
```

quasi-quotes

- Racket offers **quasi-quotes** to build S-expressions fast
- `` `(,x y 3) `` is equivalent to `(list x `y `3)`
  - I.e., Racket splices in values that are unquoted via `,`
  - (quasiquote …) will substitute any expression `,e` with the return value of `e` within the quoted S-expression
- Works multiple levels deep:
  - `` `(square (point ,x0 ,y0) (point ,x1 ,y1)) ``
- Can unquote entire expressions:
  - `` `(point ,(+ 1 x0) ,(- 1 y0)) ``

Exercise

Define mk-point and mk-square using Quasi-quotation:

```
(define (mk-point x y)
  (list 'point x y))

(define (mk-square pt0 pt1)
  (list 'square pt0 pt1))
```

Exercise

Define mk-point and mk-square using Quasi-quotation:

```
(define (mk-point x y)
  (list 'point x y))

(define (mk-square pt0 pt1)
  (list 'square pt0 pt1))
```

Answer

```
(define (mk-point x y)
  `(point ,x ,y))

(define (mk-square pt0 pt1)
  `(square ,pt0 ,pt1))
```

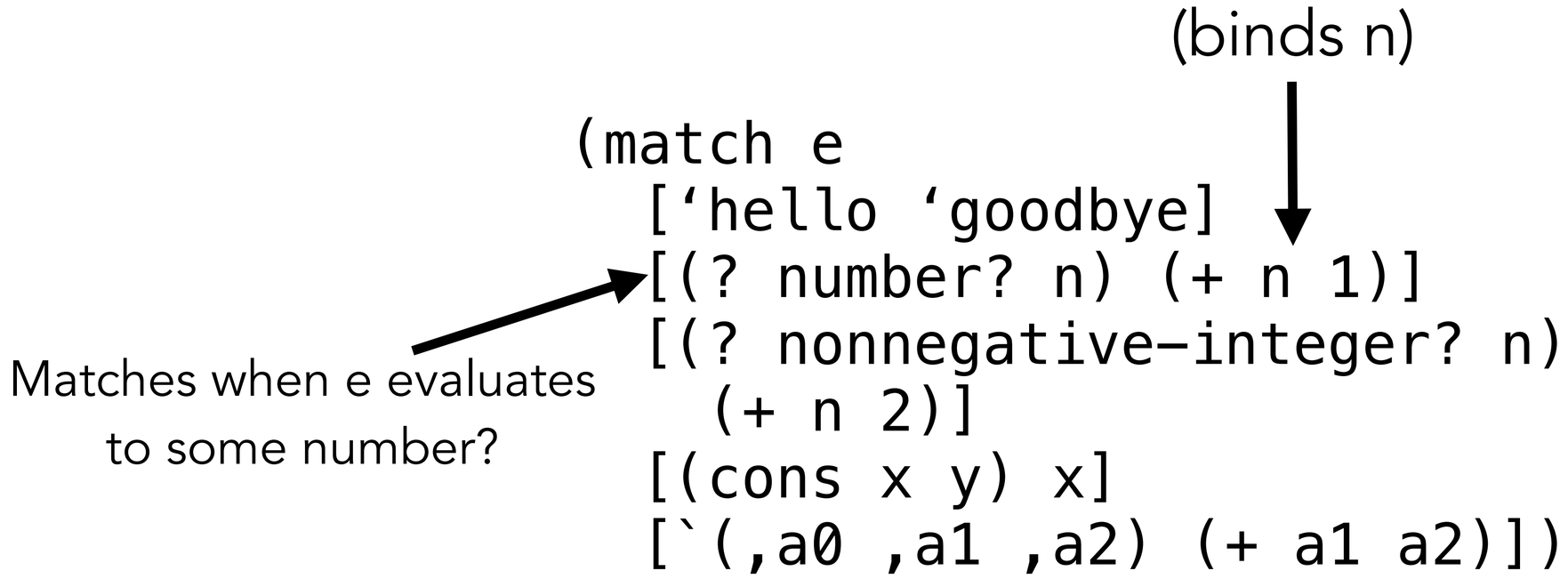
Pattern Matching

- Racket also has **pattern matching**
  - `(match e [pat₀ body₀] [pat₁ body₁]…)`
- Evaluates e and then checks each **pattern**, in order
- Pattern can bind variables, body can use pattern variables
- Many patterns (check docs to learn various useful forms)
- Patterns checked in order, first matching body is executed
  - Later bodies won't be executed, even if they also match!
- E.g.,

```
(match '(1 2 3)
       [`(,a ,b) b]
       [`(,a . ,b) b])  ; returns '(2 3)
```

Matching a literal

```
(match e
[‘hello ‘goodbye]
[(? number? n) (+ n 1)]
[(? nonnegative-integer? n)
  (+ n 2)]
[(cons x y) x]
[`(,a0 ,a1 ,a2) (+ a1 a2)])
```

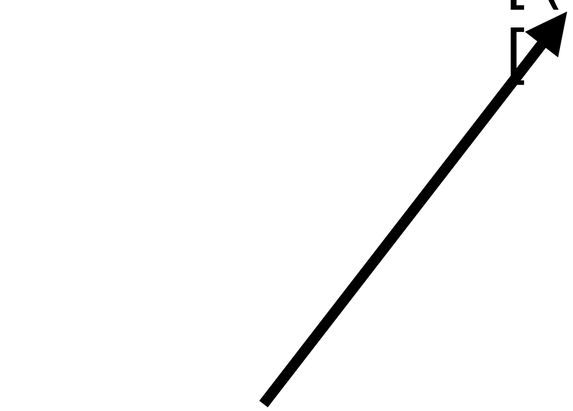

(binds n)
(match e
['hello 'goodbye]
[(? number? n) (+ n 1)]
[(? nonnegative-integer? n)
(+ n 2)]
[(cons x y) x]
[`(,a0 ,a1 ,a2) (+ a1 a2)])
Matches when e evaluates
to some number?

```
(match e
  [‘hello ‘goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
   (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)])
```

Never matches!

Subsumed by previous case!

```
(match e
  [‘hello ‘goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
    (+ n 2)]
  [(cons x y) x]
  [(,a0 ,a1 ,a2) (+ a1 a2)])
```



Matches a cons cell, binds x and y

```
(match e
  [‘hello ‘goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
    (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)])
```

Matches a list of length three

Binds first element as `a0`, second as `a1`, etc…

Called a “quasi-pattern”

Can also test predicates on bound vars:

```
`(,(? nonnegative-integer? x) ,(? positive? y))
```

```
(match e
  [‘hello ‘goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
    (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)]
  [_ 23])
```

Can also have a **default case**

Exercise

Define a function `foo` that returns:

- twice its argument, if its argument is a number?
- the first two elements of a list, if its argument is a list of length three, as a list
- the string “error” if it is anything else

```
(define (foo x)
  (match x
    [(? …) …]
    …))
```

Exercise

Define a function `foo` that returns:

- twice its argument, if its argument is a number?
- the first two elements of a list, if its argument is a list of length three, as a list
- the string “error” if it is anything else

Answer (one of many)

```
(define (foo x)
  (match x
    [(? number? n) (* n 2)]
    [`(,a ,b ,_) `(,a ,b)]
    [_ "error"]))
```

Exercise

Define a function `foo` that returns:

- twice its argument, if its argument is a number?
- the first two elements of a list, if its argument is a list of length three, as a list
- the string “error” if it is anything else

Answer (one of many)

```
(define (foo x)
  (match x
    [(? number? n) (* n 2)]
    [`(,a ,b ,_) `(,a ,b)]
    [_ "error"]))
```

Observe how quasipatterns and quasiquotes interact

Important Racket / Scheme form: **let**

- Allows you to perform local variable binding
- Establishes a new **binding site**
- Has the form:
  - (let ([x0 e0] …) e-body …)
- Note: evaluation of e0, … happen **independently** of other variable bindings
  - The following is **invalid**:
    - ```
      (let ([x 0] [y x]) y) ;; x out of scope
      ```
      ```

let* is *sequenced* let

- (let* ([x0 e0] …) body …)
  - x0 is in the scope of e1, x0/x1 in the scope of e2, …
- Regular let is like “parallel” let, whereas let* is more like “sequenced” let

A tiny language via pattern matching…

```
;; atomic operations over integers
(define (tinyif? e)
  (match e
    [`(+ ,(? integer? x) ,(? integer? y)) #t]
    [`(/ ,(? integer? x) ,(? integer? y)) #t]
    [else #f]))
```

I’m using Racket to define the ***language*** via the tinyif? predicate

Using this convention, we can avoid parsing **completely**!

(We may briefly cover LL(k) / recursive-descent parsing later)

Expressions in our language

```
(tinyif? 2) ;; #f, can't write bare number!
(tinyif? `(+ 1 2)) ;; #t, this is a tinyif?
(tinyif? `(/ 1 0)) ;; #t, this is a tinyif?
```

Our first interpreter…

```
(define (interp-tintyif e)
  (match e
    [`(+ ,x ,y)
     ;; (displayln (format "The user wants to add ~a and ~a" x y))
     (+ x y)]
    [`(/ ,x ,y)
     ;; (displayln (format "The user wants to divide ~a by ~a" x y))
     (/ x y)]))

(interp '(+ 1 2))
```

Key idea: our interpreter interp-tinyif returns a **Racket** value, thus our interpreter is embedded in Racket (Racket is the “host” language of the interpreter)

Let’s add a let, variables, and constants…

```
;; atomic operations over integers
(define (largerif? e)
  (match e
    [`(+ ,(? integer? x) ,(? integer? y)) #t]
    [`(/ ,(? integer? x) ,(? integer? y)) #t]
    [`(let [,(? symbol? x) ,(largerif? e)] ,(? largerif? body)) #t]
    [`(? symbol? x) #t]  ;; variables
    [`(? integer? x) #t] ;; constants
    [else #f]))
```

Note how let is recursive…

```
(define e0 ‘(let [x 0] (let [y 1]) (+ x y)))
```

Racket **hashes** are key/value stores

- Key/value maps (similar to hash tables)
  - ***Immutable* w/ O(1) runtime for lookup/insert**
    - Based on Hash Array-Mapped Tries (HAMT)
- `(hash 'a 0 1 2 "hello" 'c)` — creates hashes, note keys can be heterogeneous type
- `(hash-ref x 'a)` — Looks up value for key `'a`
- `(hash-set x 'a 2)` — Returns a **new** hash with updated key for 'a
- `(hash-keys x)` and `(hash-values x)`— Return list of keys / values (useful for iterating)

Now, let's define an **interpreter** for largerif?

```
;; env is a hash-map
(define (interp-largerif e env)
  ;; we will do this in class!
```

Key idea: add an **environment** to assign / lookup variables…

IfArith has **four** forms

```
(define value? number?)
(define var? symbol?)

(define (ifarith-exp? e)
  (match e
    [`(let ([,(? var?) ,(? value?)]) ,(? ifarith-exp?)) #t]
    [`(let ([,(? var?) (,(? ifarith-math-op?) ,(? var?) ,(? var?))])
            ,(? ifarith-exp?)) #t]
    [`(if ,(? var?) ,(? ifarith-exp?) ,(? ifarith-exp?)) #t]
    [`(println ,(or (? var?) (? value?))) #t]
    [_ #f]))
```

Values are **numbers**

```
(define value? number?)
(define var? symbol?)

(define (ifarith-exp? e)
  (match e
    [`(let ([,(? var?) ,(? value?)]) ,(? ifarith-exp?)) #t]
    [`(let ([,(? var?) (,(? ifarith-math-op?) ,(? var?) ,(? var?))])
            ,(? ifarith-exp?)) #t]
    [`(if ,(? var?) ,(? ifarith-exp?) ,(? ifarith-exp?)) #t]
    [`(println ,(or (? var?) (? value?))) #t]
    [_ #f]))
```

```
(define value? number?)
(define var? symbol?)
```

Variables are **symbols**

```
(define (ifarith-exp? e)
  (match e
    [`(let ([,(? var?) ,(? value?)]) ,(? ifarith-exp?)) #t]
    [`(let ([,(? var?) (,(? ifarith-math-op?) ,(? var?) ,(? var?))])
            ,(? ifarith-exp?)) #t]
    [`(if ,(? var?) ,(? ifarith-exp?) ,(? ifarith-exp?)) #t]
    [`(println ,(or (? var?) (? value?))) #t]
    [_ #f]))
```

There is **print** which accepts a variable or value

```
(define value? number?)
(define var? symbol?)

(define (ifarith-exp? e)
  (match e
    [`(let ([,(? var?) ,(? value?)]) ,(? ifarith-exp?)) #t]
    [`(let ([,(? var?) (,(? ifarith-math-op?) ,(? var?) ,(? var?))])
            ,(? ifarith-exp?)) #t]
    [`(if ,(? var?) ,(? ifarith-exp?) ,(? ifarith-exp?)) #t]
    [`(println ,(or (? var?) (? value?))) #t]
    [_ #f]))
```

There is **if**, which takes a variable, a value, and a true / false expr

```
(define value? number?)
(define var? symbol?)

(define (ifarith-exp? e)
  (match e
    [`(let ([,(? var?) ,(? value?)]) ,(? ifarith-exp?)) #t]
    [`(let ([,(? var?) (,(? ifarith-math-op?) ,(? var?) ,(? var?))])
            ,(? ifarith-exp?)) #t]
    [`(if ,(? var?) ,(? ifarith-exp?) ,(? ifarith-exp?)) #t]
    [`(println ,(or (? var?) (? value?))) #t]
    [_ #f]))
```

Let’s define an interpreter for IfArith

(Livecoding in class)

Now, let's define a **compiler** for largerif?

```
;; atomic operations over integers
(define (largerif? e)
  (match e
    [`(+ ,(? integer? x) ,(? integer? y)) #t]
    [`(/ ,(? integer? x) ,(? integer? y)) #t]
    [`(let [,(? symbol? x) ,(largerif? e)] ,(? largerif? body)) #t]
    [`(? symbol? x) #t]  ;; variables
    [`(? integer? x) #t] ;; constants
    [else #f]))
```

```c
#include <stdio.h>

int main(int argc, char **argv) {
  // code here…
}
```

```
(define template "#include <stdio.h>\n\nint main(int argc, char **argv) {\n~a}\n")

(define (binop? bop) (member bop '(+ - * /)))
```

```
;; generates a list `(,c-lines ,result-var) consisting of...
;; - list of lines of C to generate the expression
;; - resulting variable name (output expression stored here)
(define (compile-expr e)
  (define variable-name (gensym 'x))
  (match e
    ;; base case: handles constants
    [(? integer? i)
       ;; note the double parens here give us *singleton* list..
     `((,(format "    int ~a = ~a;\n" variable-name i)) ,variable-name)]
```

```
;; generates a list `(,c-lines ,result-var) consisting of...
;; - list of lines of C to generate the expression
;; - resulting variable name (output expression stored here)
(define (compile-expr e)
  (define variable-name (gensym 'x))
  (match e
    ;; base case: handles constants
    [(? integer? i)
       ;; note the double parens here give us *singleton* list..
     `((,(format "    int ~a = ~a;\n" variable-name i)) ,variable-name)]
    [`(,(? binop? bop) ,e0 ,e1)
     ;; compile e0 to a list of e0-lines (compiled C++ code to compute e0)
     ;; and variable name (assume generated by C++ code in e0-lines)
     (match-define `(,e0-lines ,e0-var) (compile-expr e0))
     ;; ^^ same but for e1
     (match-define `(,e1-lines ,e1-var) (compile-expr e1))
     ;; make this line
     (define new-line
       (format "    int ~a = ~a ~a ~a;\n" variable-name e0-var bop e1-var))
     ;; for our answer: append them all together, plus resulting variable
     `(,(append e0-lines e1-lines (list new-line)) ,variable-name)]
```

```
;; last, handle print...
[`(print ,e)
 (match-define `(,e-lines ,e-var) (compile-expr e))
 `(,(append e-lines (list (format "    printf(\"%d\\n\", ~a);\n" e-var))) ,e-var)]))
```

In sum...

```
;; generates a list `(,c-lines ,result-var) consisting of...
;; - list of lines of C to generate the expression
;; - resulting variable name (output expression stored here)
(define (compile-expr e)
  (define variable-name (gensym 'x))
  (match e
   [(? integer? i)
    `((,(format "    int ~a = ~a;\n" variable-name i)) ,variable-name)]
    [`(,(? binop? bop) ,e0 ,e1)
    (match-define `(,e0-lines ,e0-var) (compile-expr e0))
    (match-define `(,e1-lines ,e1-var) (compile-expr e1))
    (define new-line
       (format "    int ~a = ~a ~a ~a;\n" variable-name e0-var bop e1-var))
    `(,(append e0-lines e1-lines (list new-line)) ,variable-name)]
   [`(print ,e)
     (match-define `(,e-lines ,e-var) (compile-expr e))
     `(,(append e-lines (list (format "    printf(\"%d\\n\", ~a);\n" e-var))) ,e-var)]))
```

Last, a driver function that calls compile-expr…

```
(define (compile e)
  (display (format template (string-join (first (compile-expr e))))))
```

Calls compile-expr to produce the lines for the input expr e, then extracts the list of lines (don’t need to know answer var) and uses string-join to stick them together to put into template, then prints that to terminal.

So we've got a compiler from this tiny lang to C in ~50 lines of code.

```
#lang racket

;; my small language of arithmetic,
;; constants, and a single "print"
;; statement (can be nested)
(define (lang? e)
  (match e
    [`(+ ,(? lang? e0) ,(? lang? e1)) #t]
    [`(- ,(? lang? e0) ,(? lang? e1)) #t]
    [`(* ,(? lang? e0) ,(? lang? e1)) #t]
    [`(/ ,(? lang? e0) ,(? lang? e1)) #t]
    [`(print ,(? lang? e)) #t]
    [(? integer? n) #t]))

;; args evaluated left-to-right so this prints (to the console)
;; 1
;; 2
;; 3
(define example0 '(print (+ (print 1) (print 2))))

(define template "#include <stdio.h>\n\nint main(int argc, char **argv) {\n~a}\n")

(define (binop? bop) (member bop '(+ - * /)))

;; generates a list `(,c-lines ,result-var) consisting of...
;; - list of lines of C to generate the expression
;; - resulting variable name (output expression stored here)
(define (compile-expr e)
  (define variable-name (gensym 'x))
  (match e
    ;; base case: handles constants
    [(? integer? i)
       ;; note the double parens here give us *singleton* list..
     `((,(format "    int ~a = ~a;\n" variable-name i)) ,variable-name)]
    [`(,(? binop? bop) ,e0 ,e1)
     ;; compile e0 to a list of e0-lines (compiled C++ code to compute e0)
     ;; and variable name (assume generated by C++ code in e0-lines)
     (match-define `(,e0-lines ,e0-var) (compile-expr e0))
     ;; ^^ same but for e1
     (match-define `(,e1-lines ,e1-var) (compile-expr e1))
     ;; make this line
     (define new-line
       (format "    int ~a = ~a ~a ~a;\n" variable-name e0-var bop e1-var))
     ;; for our answer: append them all together, plus resulting variable
     `(,(append e0-lines e1-lines (list new-line)) ,variable-name)]
    ;; last, handle print...
    [`(print ,e)
     (match-define `(,e-lines ,e-var) (compile-expr e))
     `(,(append e-lines (list (format "    printf(\"%d\\n\", ~a);\n" e-var))) ,e-var)]))

(define (compile e)
  (display (format template (string-join (first (compile-expr e))))))
```

Next time in class…

- Will be doing Racket boot-up / review
- I will be assigning several videos from 352
  - You may skip them if you've seen them, I'll be working independent exercises in class
- All students: read over Racket documentation and work through refresher exercise I put up on Slack to discuss in class
  - This exercise will be worth 1 participation point
    - Due Friday night(ish, I am flexible but not too long)