CMSC 430 - 29 Oct 2026

Pattern matching compilation

- Knock - pattern matching

CMSC 430 - 29 Oct 2026

Announcements

- Assignment 7 due Tuesday
- Exam 2 next Thursday (11/5)
- Quiz released after class, due by start of Tuesday
- Assignments 8 and 9 out tonight

Knock

(cont.)

Compiling a single pattern

The heart of pattern matching compilation

```
;; Pat CEnv Symbol -> (list Asm CEnv)
(define (compile-pattern p cm next)
```

Two things:

- instructions that bind variables in p if it matches,
  or pops environment and jumps to next if it doesn't
- compile time environment that describes what p binds

Compiling a match expression

```
;; Expr [Listof Pat] [Listof Expr] CEnv Bool -> Asm
(define (compile-match e ps es c t?)
  (let ((done (gensym)))
    (seq (compile-e e c #f)
         (Push rax)     ;; save away to be restored by each clause
         (compile-match-clauses ps es (cons #f c) done t?)
         (Jmp 'err)
         (Label done)   ;; where to jump if you've finished a RHS
         (Add rsp 8)))) ;; pop the saved value being matched
```

Compiling a list of match clauses

```
;; [Listof Pat] [Listof Expr] CEnv Symbol Bool -> Asm
(define (compile-match-clauses ps es c done t?)
  (match* (ps es)
    [('() '()) (seq)]
    [((cons p ps) (cons e es))
     (seq (compile-match-clause p e c done t?)
          (compile-match-clauses ps es c done t?))]))
```

Just the sequence of compiling each clause

Compiling a match clause

```
;; Pat Expr CEnv Symbol Bool -> Asm
(define (compile-match-clause p e c done t?)
  (let ((next (gensym)))
    ;; Compiling a pattern needs to know two things:
    ;; - a compile-time environment that describes bindings made so far
    ;; - where to jump in case the pattern doesn't match
    (match (compile-pattern p '() next)
      ;; Compiling a pattern produces two things:
      ;; - instructions that will either:
      ;;   * match value currently in rax,
      ;;     binding things by pushing on the stack
      ;;   * not match, clear out the bindings, and jump to next
      ;; - a compile-time environment that describes the bindings
      ;;   of the pattern, should it succeed
      [(list i cm)
       (seq (Mov rax (Mem rsp)) ;; restore value being matched into rax
            i                   ;; do matching, binding, maybe jumping to next
            ;; If you made it here, the pattern matched and the bindings described
            ;; by cm have been pushed onto the stack
            (compile-e e (append cm c) t?) ;; Do the RHS of clause
            (Add rsp (* 8 (length cm)))    ;; Clear out bindings
            (Jmp done)                     ;; Jump to bottom of clauses code
            (Label next))])))              ;; Where to jump if p doesn't match
```

Compiling a wildcard or variable pattern

```
;; Pat CEnv Symbol -> (list Asm CEnv)
(define (compile-pattern p cm next)
  (match p
    ;; - Always matches
    ;; - No new bindings
    [(Var '_)
     (list (seq) cm)]
    ;; - Always matches
    ;; - Binds x to value in rax
    [(Var x)
     (list (seq (Push rax)) (cons x cm))]
    #;...))
```

Compiling a literal pattern

```
;; Pat CEnv Symbol -> (list Asm CEnv)
(define (compile-pattern p cm next)
  (match p
    [(Lit l)
     ;; - Matches if l is equal to value in rax
     ;; - No new bindings
     (let ((ok (gensym)))
       (list (seq (Mov r8 rax)
                  (compile-datum l)
                  (Cmp rax r8)
                  (Je ok)
                  (Add rsp (* 8 (length cm)))
                  (Jmp next)
                  (Label ok))
             cm))]
    #;...))
```

Compiling a box pattern

```
;; Pat CEnv Symbol -> (list Asm CEnv)
(define (compile-pattern p cm next)
  (match p
    [(Box p)
     ;; - Matches if rax is a box and value in box matches p
     ;; - Adds any bindings that p adds
     (match (compile-pattern p cm next)
       [(list i1 cm1)
        (let ((ok (gensym)))
          (list
           (seq (Mov r8 rax)
                (And r8 ptr-mask)
                (Cmp r8 type-box)
                (Je ok)
                ; haven't pushed anything yet
                (Add rsp (* 8 (length cm)))
                (Jmp next)
                (Label ok)
                (Mov rax (Mem rax (- type-box)))
                i1)
           cm1))])]
    #;...))
```

Compiling an and pattern

```
;; Pat CEnv Symbol -> (list Asm CEnv)
(define (compile-pattern p cm next)
  (match p
    [(Conj p1 p2)
     (match (compile-pattern p1 (cons #f cm) next)
       [(list i1 cm1)
        (match (compile-pattern p2 cm1 next)
          [(list i2 cm2)
           (list
            (seq (Push rax) ;; Save away value so it can be restored for i2
                 i1
                 ;; Reach into stack to grab stowed away value
                 (Mov rax (Mem rsp (* 8 (- (sub1 (length cm1)) (length cm)))))
                 i2)
            cm2)])])]
    #;...))
```

Compiling a cons pattern

```
;; Pat CEnv Symbol -> (list Asm CEnv)
(define (compile-pattern p cm next)
  (match p
    [(Cons p1 p2)
     (match (compile-pattern p1 (cons #f cm) next)
       [(list i1 cm1)
        (match (compile-pattern p2 cm1 next)
          [(list i2 cm2)
           (let ((ok (gensym)))
             (list
              (seq (Mov r8 rax)
                   (And r8 ptr-mask)
                   (Cmp r8 type-cons)
                   (Je ok)
                   (Add rsp (* 8 (length cm))) ; haven't pushed anything yet
                   (Jmp next)
                   (Label ok)
                   (Xor rax type-cons)
                   (Mov r8 (Mem rax 8))
                   (Push r8)                   ; push cdr
                   (Mov rax (Mem rax 0))       ; mov rax car
                   i1
                   (Mov rax (Mem rsp (* 8 (- (sub1 (length cm1)) (length cm)))))
                   i2)
              cm2))])])]
    #;...))
```