Introduction to Compiler Construction

CIS400 (Special Topics) — Fall 2021

Kris Micinski

Course Logistics

- Welcome to the course, I'm really happy you're here!
- Today: building a compiler for a tiny language in Racket
- Next few weeks: boot-up on Racket and interpreters
- Soon: log in to autograde.org with provided credentials
  - **You'll get these at the end of the week!**
- Course website:
  - https://kmicinski.com/cis400-f21
- I will be making heavy use of Slack
  - **Please make sure you join the Slack right now!**

Course Delivery

- I will coordinate on Slack (check Slack frequently please!)
- I will run this as a lecture course, even if we switch to online
  - I will occasionally assign course videos, these are assigned viewing for lecture (no more than 80min/week)
- Active participation and attendance is expected
- But do not feel the need to come to class if you are sick
  - Feel free to DM me on Slack
  - I will make my **best effort** to hold class and record via Zoom when possible
- **Please keep me updated on your progress in the course!**

Grading

- **5 autograder projects (50%)**
  - On autograder.org
  - Worth 10% **each**
  - Deadline policy:
    - By 11:59PM deadline: 100%
    - Next 72 hours: 85% * grade
    - Until end of course: 70% * grade
  - 1 warmup project, 4 “assigned” projects (we have answer keys) build Scheme->LLVM compiler.
- **Two midterms (30%)**
  - Worth 15% each
  - Take-home (72hrs), several open-ended question/answers (writing, code, …)
  - Can collaborate with up to 3 students, but each student must be able to explain solns.
- **Final Project (20%)**
  - Wrap up your compiler, add language feature, (maybe) presentations.
- **Participation (5%)**
  - I will list various one-off opportunities to get +1% bonus participation credit, you may do a maximum of 5
  - “Meet your professor” may count for 1/5, Slack/ email me to set up a time to discuss your career / plans / etc..

Topics (**Very** tentative)

This is a rough list of topics, but we will spend **significant** time going over projects.

- Week 1 — Intro and Racket Intro
- Week 2 — Racket boot-up, interpreter intro
- Week 3 — Lambda Calculus, Church encoding, big-step interpreters
- Week 4 — Abstract Machines, set!, and store-passing style
- Week 5 — Desugaring letrec, promises, call/cc
- Week 6 — call/cc, CEK machine, stack-passing
- Week 7 — CEK livelivecoding, catch-up, midterm review
- Week 8 — SSA, ANF, assignment conversion
- Week 9 — ANF / CPS conversion
- Week 10 — Closure conversion
- Week 11 — Data and control-flow analysis
- Week 12 — Livecoding 0-CFA and LLVM intro
- Week 13 — Register allocation and HAMT
- Week 14 — Garbage collection, Boehm GC

At a high level, what are we doing?

- I’ll be teaching you to build a compiler from Scheme -> LLVM
- Scheme: high-level functional language with closures, primitives, and advanced higher-order control (exceptions, continuations, ..)
- Why is building a Scheme compiler interesting?
    - Compiling higher-order function invocation is the heart of many modern-day languages (Java’s objects, JavaScript methods, …)
        - Details (objects vs. closure vs. prototype) will differ, but principles stay largely the same.
    - Lots of features we can desugar to a very expressive core intermediate representation (IR)
        - Extremely high semantic density: write a full modern language in ~1.5-3kloc

At the end, what will your language do?

- Multi-argument and variadic higher-order functions (lambdas)
- Primitives (numeric ops, strings, lists, …)
- Conditional control flow (cond, if, …)
- Proper first-class continuations (call/cc, exceptions, dynamic-wind)
- Pattern matching
- Mutable variables (set!)
- Quote/quasiquotation
- Garbage collection (via Boehm GC)
- Basically: you'll be able to run quite-fully-featured 100s of line programs (e.g., sudoku solvers, etc…).

Why LLVM?

- LLVM is an industrial-strength compiler backend
- Basically: a good low-level IR for C-like languages
- We will be using its assembly format, though it also includes a C++ API (we won't be using this)
- Relatively portable assembly:
  - If you want to implement x86/ARM for your final project, you can…
- I am assuming you have seen **some** assembly before (but don't be scared, you can learn..)
  - However, figuring out the necessary parts is not hard and one of the funnest parts of the course!

```
define i32 @sum(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32* %a.addr, align 4
  %1 = load i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  ret i32 %add
}
```

Why Racket?

- Racket is a more-fully-featured Scheme, our metalanguage
- I'm not here to convince you Racket is a language you'll use in industry, or even day-to-day after this course
- Over the next few weeks we'll be introducing Racket per-se
- But Racket is **designed** to directly enable the succinct implementation of programming languages and compilers
- Allows ad-hoc structured data (S-expressions)
- Simple construction (quasiquoting)
- and destruction (pattern matching & quasipatterns)

Our first interpreter…

```
(define (interp 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))
```

Now, building our first language…

- Today, we will build a compiler from an arithmetic language to C
- I will be using Racket throughout
  - If you are feeling overwhelmed (or haven’t learned Racket) don’t worry—we’ll be doing a boot-up course in the next few weeks.
- This will be to give a broad introduction to the ideas..

```
;; 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))]
    [(? integer? n) #t]))

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

```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)