Compiling Arithmetic Expressions to C

CIS531 — Fall 2025, Syracuse

Kristopher Micinski

Syracuse University
College of Engineering
& Computer Science

Lecture Roadmap

Let's write an end-to-end compiler, minimal possible overhead!

- First, define the language
  - Write a parser / lexer / etc.
- Second, define its semantics
  - Write an interpreter
- Third, compile it to x86_64
  - Keep it as simple as possible!

By the end of the lecture, we'll have a complete compiler

The R_0 language

- Consists of *arithmetic expressions* (which may be recursive) wrapped in a single, top-level ‘(program …) block
- As a grammar (more on this soon), we would write (Fig 1.1 in the book):

  exp ::= *int* | (read) | (- *exp*) (+ *exp exp*)
  R_0 ::= (program *exp*)

- Only (binary) addition and (unary) negation, can accept user input (integers) from stdin via (read).

```
(define (exp? sexp)
  (match sexp
    [(? fixnum?) #t]
    [‘(read) #t]
    [‘(- ,e) (exp? e)]
    [‘(+ ,e1 ,e2)
      (and (exp? e1) (exp? e2))]
    [else #f]))

(define (R0? sexp)
  (match sexp
    [‘(program ,e) (exp? e)]
    [else #f]))

(R0? ‘(program (+ (read) (- 8)))) ;; #t
(R0? ‘(program (- (read) (+ 8)))) ;; #f
```

An interpreter for R_0

- Consists of *arithmetic expressions* (which may be recursive) wrapped in a single, top-level ‘(program …) block
- As a grammar (more on this soon), we would write (Fig 1.1 in the book):

  exp ::= *int* | (read) | (- *exp*) (+ *exp exp*)
  R_0 ::= (program *exp*)

- Only (binary) addition and (unary) negation, can accept user input (integers) from stdin via (read).

Building the interpreter

- We make things simple: no parsing, input program is an S-expression (read)
- Now, we define an interpreter using recursion and pattern matching
- The output of our interpreter is a number
  - All programs in this language produce a single number—this makes it easy!
- So now define interp : Expression -> Integer
- Let’s code it up in Racket!

Building the interpreter

```
(define (interp-R0 e)
  (define (interp e)
    (match e
      [(? fixnum? n) n]
      ['(read) (read)]
      [`(- ,e+) (- (interp e+))]
      [`(+ ,e0 ,e1) (+ (interp e0) (interp e1))]))
  (match e
    [`(program ,e+) (interp e+)]))
```

Compiling R_0 to C

- Our main goal is to write an end-to-end compiler, we'll make it easy by compiling to C
- To compile to assembly, we'll need to first have a crash course on x86_64 assembly
- C is a great compilation target—we intentionally compile to x86_64 to learn how
- Key: just translate the nested expressions into nested C expressions
- Print the result to the screen
- Input (from stdin) happens via a utility function we will write

Here's the ***whole compiler***

The language is ***very*** simple, using C makes all the hard parts easy

```
(define (r0->c r0)
  (define (translate-expr e)
    (match e
      [(? fixnum? i) (number->string i)]
      ['(read) "read_int64()"]
      [`(- ,e) (format "(- ~a)" (translate-expr e))]
      [`(+ ,e0 ,e1) (format "(~a + ~a)"
                            (translate-expr e0)
                            (translate-expr e1))]))
  (match r0
    [`(program ,e)
     (format (string-append "#include \"runtime.h\"\n\n"
                            "int main(int argc, char **argv) {\n"
                            "    print_int64(~a);\n"
                            "}\n")
             (translate-expr e))]))
```

To make generated code as easy as possible, I use this runtime.h

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <inttypes.h>

int64_t read_int64(void) {
  int64_t value;
  if (scanf("%" SCNd64, &value) != 1) {
    /* handle input error as needed */
    printf("Error: expected an integer. Exiting.");
    exit(1);
  }
  return value;
}

void print_int64(int64_t n) {
    printf("%" PRId64 "\n", n);
}
```

Example compilation...

```
(program (+ (+ 5 3) (- (+ 2 (read)))))
```

r0->c

```
#include "runtime.h"

int main(int argc, char **argv) {
    print_int64(((5 + 3) + (- (2 + read_int64()))));
}
```

Compile and run this code…

```
example/ # gcc compiled.c -o output
example/ # ./output
… # awaits user input…
```

Debrief

That was easy! We wrote a ***whole*** compiler!?

- We skipped all of the hard parts:
  - C supports features like nested expressions,
  - Variables,
  - User input,
  - Etc…
- Parsing was simple: (read)
- From now on, *we'll use x86_64 assembly*