

CMSC430: Introduction to Compilers

Michael Li

1 Intro to Compilers

Compiler: maps an *input* string to an *output* string

- Typically *input* and *output* strings are *programs*. For example `gcc: C -> Binary` OR `pdftex: LaTeX -> PDF`
- Output program must
 1. Have the same *meaning* as the input program
 2. Be *executable* in the relevant *context* (e.g. VM, web browser)

Compiler **Pipeline** converts input program to an executable binary through many stages

- **Parsed** into a data structure called **Abstract Syntax Tree**
- **Checked** to make sure code is well-formed
- **Simplified** into some convenient **Intermediate Representation**
- **Optimized** into an equivalent but faster program
- **Linked** against a run-time

2 Racket

2.1 Basic Operations

Uses prefix notation where parentheses play an important role

```
> (1 + 1 ( * 2 2))  
5  
> ( * (+ 1 2) 2)  
6
```

2.2 Functions

Anonymous functions are written as a series of lambda functions

```
> (lambda (x) (lambda (y) (+ x y)))  
#<procedure>
```

To apply the function, add the necessary arguments (written in parentheses). Both ways below are valid

```
> (((lambda (x) (lambda (y) (+ x y))) 3) 4)  
7  
  
> ((lambda (x) (lambda (y) (+ x y))) 3 4)  
7
```

NOTE: wrong number of arguments will cause an error (Racket doesn't allow partial functions)

2.3 Variable Definition

Variables are defined using *define* form

```
> (define x 3)
> (define y 4)
> (+ x y)
7
```

Functions are defined in a similar form, using `lambda` (brackets below are only stylistic). Function definitions can also be shorthand by removing the `lambda`

```
> (define fact
  (lambda (n)
    (match n
      [0 1]
      [n (* n (fact (- n 1)))])))
> (fact 5)
120

> (define (fact n)
  (match n
    [0 1]
    [n (* n (fact (- n 1)))]))
```

2.4 Lists

Empty lists is written with `'()` and elements are added using `cons`. Shorthand is done by using `list`

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

> (list "1" 3)
'("a" 3)
```

NOTE: Racket lists are heterogenous so they can store different data types

NOTE: `cons` is both a pair constructor and a list.

- Tuple pairs are created by writing `(cons "a" 3)`. Lists themselves are considered as pairs (an element and another list)
- Non-empty lists are a subset of pairs, ones whose second component is the rest of the list. For non-empty lists, `first` and `rest` produce the first element and the tail of the list, respectively. However, `first` and `rest` CANNOT be used for tuple pairs
- Instead, `car` and `cdr` are used to access the left and right components of a pair. When given pairs that are also lists, they behave like `first` and `rest`

```
> (first (cons 3 (cons 4 '())))
3
> (rest (cons 3 (cons 4 '())))
'(4)

> (car (cons 3 4))
3
> (cdr (cons 3 4))
4

> (car (cons 3 (cons 4 '())))
3
> (cdr (cons 3 (cons 4 '())))
'(4)
```

2.5 Pattern Matching

Rackets use *structures* that are a single variant of a data type in OCaml

```
> (struct leaf ())  
> (struct node (i left right))
```

These create constructors that we can call (`leaf` requires no arguments, `node` requires 3 arguments)

```
> (leaf)  
(leaf)  
> (node 5 (leaf) (leaf))  
(node 5 (leaf) (leaf))  
> (node 3 (node 2 (leaf) (leaf)) (leaf))  
(node 3 (node 2 (leaf) (leaf)) (leaf))
```

Pattern Matching is used to *discriminate* and *deconstruct* arguments in a function

```
> (define (bt-empty? bt)  
  (match bt  
    [(leaf) #t]  
    [(node _ _ _) #f]))  
> (bt-empty? (leaf))  
#t  
> (bt-empty? (node 5 (leaf) (leaf)))  
#f  
  
> (define (bt-height bt)  
  (match bt  
    [(leaf) 0]  
    [(node _ left right)  
      (+ 1 (max (bt-height left)  
                 (bt-height right)))]))  
> (bt-height (leaf))  
0  
> (bt-height (node 4 (ndoe 2 (leaf) (leaf)) (leaf)))  
2
```

2.6 Symbols

A **symbol** is an atomic piece of data, written using **quote** notation (`'symbol-name`). Symbols can

- Be directly compared
- Converted to and from strings
- Generated uniquely using **gensym**
- Used to define *enum* like datatypes

```
> (equal? 'fred 'fred)
#t
> (equal? 'fred 'wilma)
#f

> (symbol->string 'fred)
"fred"
> (string->symbol "fred")
'fred

> (gensym)
'g2905
> (gensym)
'g2906

> (define (flintstone? x)
  (match x
    ['fred #t]
    ['wilma #t]
    ['pebbles #t]
    [_ #f]))
> (flintstone? 'fred)
#t
> (flintstone? 'barney)
#f
```

2.7 Quote, Quasiquote, Unquote

Quotes can be used to make a list of symbols. For example, `'(x y z)` is equivalent to `(list 'x 'y 'z)`

They can also nest lists within quoted list. For example, `((x) y (q r))` is equivalent to `(list (list 'x) 'y (list 'q 'r))`

Strings, booleans and numbers can also be put inside a quote. When called upon, they act like a string, boolean, or number so `'5` and `'"Fred"` are just 5 and "Fred"

Quotes can also be used to generate pairs, separated by a dot `.` for the left and right components. For example, `'(1 . 2)` is equivalent to `(cons 1 2)`

The different ways to construct with quote form are called **symbolic-expressions**

- Inside a quote, you can write down a *data representation* of an expression. For example, `(+ 1 2)` is an expression whereas `'(+ 1 2)` constructs a piece of data with 3 elements

Quasiquotes ``d` and **unquote** `,e` are used in the following way.

Quasiquotes are the same thing as **quotes** except that if an **unquote** is seen anywhere inside `d`, then the *expression* `e` is evaluated and its value will be used in place of `,e`

- This gives the ability to escape out of a quoted piece of data and go back to expression mode

``(+ 1 ,(+ 1 1))` equivalent to `(list '+ 1 (+ 1 1))` equivalent to `(list '+ 1 2)`

Unquote-splicing ,@e is similar to unquote except that the expression must produce a list (or pair) and the elements of that list (or pair) are spliced into the outer data

```
> `( + 1 ,@(map add1 '(2 3)))  
  
> (cons '+ (cons 1 (map add1 (list 2 3))))  
  
> (list '+ 1 3 4)  
  
> '(+ 1 3 4)
```

2.8 Poetry of S-Expressions

Use symbols instead of constructors and lists in place of fields. The following code snippets represent a binary trees

```
> 'leaf  
'leaf  
> '(node 3 leaf leaf)  
'(node 3 leaf leaf)  
> '(node 3  
      (node 7 leaf leaf)  
      (node 9 leaf leaf))  
'(node 3 (node 7 leaf leaf) (node 9 leaf leaf))
```

Now rewrite functions to match new datatype definition

```
> (define (bt-empty? bt)  
  (match bt  
    ['leaf #t]  
    [(cons 'node _) #f]))  
> (bt-empty? 'leaf)  
#t  
> (bt-empty? '(node 3  
                  (node 7 leaf leaf)  
                  (node 9 leaf leaf)))  
#f  
> (define (bt-height bt)  
  (match bt  
    ['leaf 0]  
    [(list 'node _ left right)  
     (+ 1 (max (bt-height left)  
                (bt-height right)))]))  
> (bt-height 'leaf)  
0  
> (bt-height '(node 3  
                  (node 7 leaf leaf)  
                  (node 9 leaf leaf)))  
2
```

Can make definitions more concise using quasiquote notation

```
> (define (bt-empty? bt)
  (match bt
    [`leaf #t]
    [`(node . ,_) #f]))
> (bt-empty? 'leaf)
#t
> (bt-empty? '(node 3
                    (node 7 leaf laef)
                    (node 9 leaf leaf)))
#f
> (define (bt-height bt)
  (match bt
    [`leaf 0]
    [`(node ,_ ,left ,right)
     (+ 1 (max (bt-height left)
                (bt-height right)))]))
> (bt-height 'leaf)
0
> (bt-height '(node 3
                    (node 7 leaf leaf)
                    (node 9 leaf leaf)))
2
```

2.9 Testing, Modules, Submodules

Unit tests will be made using `rackunit` library. Must **require** it to use it

`check-equal?` function takes 2 arguments and checks if the first argument produces something **equal?** to the second argument

```
> (require rackunit)
> (check-equal? (add1 4) 5)
> (check-equal? (* 2 3) 7)
```

```
-----
FAILURE
name:      check-equal?
location:  eval:76:0
actual:    6
expected:  7
-----
```

Module: basic unit of code organization. Every file is a module. Below code snippet defines the module `bt` that has a single value named `bt-height`

```
> (module bt racket
  (provide bt-height)
  (define (bt-height bt)
    (match bt
      [`leaf 0]
      [`(node ,_ ,left ,right)
       (+ 1 (max (bt-height left)
                  (bt-height right)))])))
```

Then we can import the module and use `bt-height`

```
> (require 'bt)
> (bt-height 'eaf)
0
```

3 x86/a86

Instruction Set Architecture: programming language whose interpreter is implemented in hardware

- Since x86 is complicated, for this class we use a86, an abstracted version of x86

Below is a example code snippet that computes the 36th triangular number and stores it in the `rax` register

- `nth` triangular number is the sum of integers from 0 to n
- `entry` is a global label that is the main entry point for the program
- **NOTE:** for MacOS all label names need to be prefixed with an underscore (e.g. `_entry`)

```
global entry
default rel
section .text
global entry
entry:
    mov rbx, 36          ; the "input"
;;; tri: a recursive function for computing nth triangular number,
;;; where n is given in rbx
tri:
    cmp rbx, 0           ; if rbx = 0, done
    je done
    push                ; save rbx
    sub rbx, 1
    call tri             ; compute tri(rbx - 1) in rax
    pop rbx              ; restore rbx
    add rax, rbx          ; result is rbx + tri(rbx -1)
    ret
done:                    ; jump here for base case
    mov rax, 0           ; return 0
    ret
```

Above code can be compiled using `nasm`: `nasm -f elf64 -o tri.o tri.s`

- **NOTE:** use `macho64` for Mac and `elf64` for Linux

To run the object file, we need to link a small C program to call the `entry` label and print the result

```
# include <stdio.h>
# include <inttypes.h>
int64_t entry();
int main(int argc, char** argv) {
    int64_t result = entry();
    printf("%" PRIu64 "\n", result);
    return 0;
}
```

In the code above

- When the program calls `entry`, it places a return pointer on the stack and jumps to `entry`
- `rax` register by convention is used for a return value. Convention is part of the **Application Binary Interface**
- When the assembly code executes its final `ret` instruction, it jumps back to C with the 36th triangular number in `rax`

3.1 a86: Representing x86 Code as Data

Goal is to ignore the complexities of x86 and write programs as data, designing a data type definition representing x86 programs and *compute* programs

a86 program is a list of a86 instructions. Each instruction is represented as a structure. Below is the triangular number program written in Racket

- **NOTE:** this a86 program is just a value in Racket, meaning we can use Racket as a **Meta-Language** to write programs that compute *with* x86 programs

```

> (require a86)
> (define tri n
  (list (Global 'entry)
        (Label 'entry)
        (Mov 'rbx n)
        (Label 'tri)
        (Cmp 'rbx 0)
        (Je 'done)
        (Push 'rbx)
        (Sub 'rbx 1)
        (Call 'tri)
        (Pop 'rbx)
        (Add 'rax 'rbx)
        (Ret)
        (Label 'done)
        (Mov 'rax 0)
        (Ret)))
> (define tri-36 (tri 36))

```

To convert our data representation to its interpretation as an x86 program, first convert the data to a string (`asm-string` notation)

```

> (display (asm-string (tri 36)))
global entry
default rel
section .text
global entry
entry:
  mov rbx, 36
tri:
  cmp rbx, 0
  je done
  push rbx
  sub rbx, 1
  call tri
  pop rbx
  add rax, rbx
  ret
done:
  mov rax, 0
  ret

```

Next we assemble, link, and execute. We can do this using `asm-interp` function, which consumes an a86 program as input and produces the integer output. Here, `asm-interp` is an **Interpreter** for a86. Behind the scenes it is

- converting input into nasm
- assembling the code
- compiling a thin C wrapper
- executing the program
- reading the results

3.2 Stacks: Pushing, Popping, Calling, Returning

a86 can manipulate the stack register pointer `'rsp` using `Push`, `Pop`, `Call`, and `Ret`

- Useful to save values that may be needed later
- **NOTE:** stack memory is allocated in low address space and grows downward. So pushing an element onto the stack *decrements* `'rsp`

For example, to add 2 values `'f` and `'g`, we use the stack to save the return value of `'f` while the call to `'g` proceeds


```
(seq (Call 'f)
      (Push 'rax)
      (Call 'g)
      (Pop 'rbx)
      (Add 'rax 'rbx))
```

Code above pushes the value in 'rax onto the stack and then pops it off into 'rbx after 'g returns

- Here Push and Pop give the illusion of a stack, but in reality we are just working in memory and incrementing/decrementing the 'rsp register value

In a similar fashion, Call and Ret provide useful illusions; a notion of a procedure and procedure call mechanism

- (Call 'f) control jumps to the instruction following (Label 'f)
- Once (Ret) is called, the CPU jumps back to the instruction following (Call 'f)
- Here (Call 'f) pushes the address of the subsequent instruction onto the stack and then jumps to the label 'f. Ret pops the return address off the stack and jumps to it
- An important instruction is Lea which loads the address of something (e.g. the address of a label): (Lea 'rax 'f)

```
> (eg (seq (Lea 'rax 'fret) ; load address of 'fret label into 'rax
          (Push 'rax)      ; push the return pointer on to stack
          (Jump 'f)        ; jump to 'f
          (Label 'fret)    ; <- return point for "call" to 'f
          (Push 'rax)      ; save result (like before)
          (Lea 'rax 'gret) ; load address of 'gret label into 'rax
          (Push 'rax)      ; push the return pointer on to stack
          (Jump 'g)        ; jump to 'g
          (Label 'gret)    ; <- return point for "call" to 'g
          (Pop 'rbx)       ; pop saved result from calling 'f
          (Add 'rax 'rbx)))
```

42

3.3 Instruction Set

a86 has 16 registers: 'rax, 'rbx', 'rdx, 'rbp, 'rsp, 'rsi, 'rdi, 'r8, 'r9, 'r10, 'r11, 'r12, 'r13, 'r14, and 'r15

- 'eax accesses the lower 32-bits of 'rax

```
(register? x) → boolean?
x : any/c
```

Tests if it is a register

```
(label? x) → boolean?
x : any/c
```

Test if it is a label name (symbols which are not register names)

```
(instruction? x) → boolean?
x : any/c
```

Test if it is an instruction

```
(offset? x) → boolean?
x : any/c
```

Test if it is an offset

```
(seq x ...) → (listof instruction?)  
x : (or/c instruction? (listof instruction?))
```

Function for splicing together instructions and list of instructions

Example:

```
> (seq (list (Label 'foo)  
            (Mov 'rax 0))  
      (Mov 'rdx 'rax)  
      (list (Call 'bar)  
            (Ret)))  
  
(list  
  (Label 'foo)  
  (Mov 'rax 0)  
  (Mov 'rdx 'rax)  
  (Call 'bar)  
  (Ret))
```

```
(prog x ...) → (listof instruction?)  
x : (or/c instruction? (listof instruction?))
```

Similar to `seq` but also checks that the instructions are well-formed

- Programs have at least one label; the first label is used as an entry point
- All label declarations are unique
- All label targets are declared

```
(struct % (s))  
s : string?  
  
(struct %% (s))  
s : string?  
  
(struct %%% (s))  
s : string?
```

Creates a comment in assembly code

- `%` adds a comment towards the right side of the current line
- `%%` creates a comment 1 tab over
- `%%%` creates a comment on its own line

```
(struct Offset (r i))  
r : register?  
i : exact-integer?
```

Creates a memory offset from a register. Offsets are used as arguments to instructions to indicate memory locations

```
(struct Label (x))  
x : label?
```

Creates a label from a given symbol. Each label must be unique and register names cannot be used as label names

```
(struct Extern (x))  
x : label?
```

Declares an external label

```
(struct Call (x))
  x : (or/c label? register?)
```

Calls an instruction

```
(struct Ret ())
```

A return instruction

```
(struct Mov (dst src))
  dst : (or/c register? offset?)
  src : (or/c register? offset? exact-integer?)
```

Moves `src` to `dst`. **NOTE:** Either `dst` or `src` may be offsets, but NOT both

```
(struct Add (dst src))
  dst : register?
  src : (or/c register? offset? exact-integer?)
```

Adds `src` to `dst` and writes the result to `dst`

```
(struct Sub (dst src))
  dst : register?
  src : (or/c register? offset? exact-integer?)
```

Subtracts `src` from `dst` and writes the result to `dst`

```
(struct Cmp (a1 a2))
  a1 : (or/c register? offset?)
  a2 : (or/c register? offset? exact-integer?)
```

Compares `a1` to `a2`, setting the status flags for conditional instructions like `Je` and `Jl`

```
(struct Jmp (x))
  x : (or/c label? register?)
```

Jump to label `x`

```
(struct Je (x))
  x : (or/c label? register?)
```

Jump to label `x` if the conditional flag is set to “equal”

```
(struct Jne (x))
  x : (or/c label? register?)
```

Jump to label `x` if the conditional flag is set to “not equal”

```
(struct Jl (x))
  x : (or/c label? register?)
```

Jump to label `x` if the conditional flag is set to “less than”

```
(struct Jg (x))
  x : (or/c label? register?)
```

Jump to label `x` if the conditional flag is set to “greater than”

```
(struct And (dst src))
  dst : (or/c register? offset?)
  src : (or/c register? offset? exact-integer?)
```

Compute logical “and” of `dst` and `src` and put the result in `dst`

```
(struct Or (dst src))
  dst : (or/c register? offset?)
  src : (or/c register? offset? exact-integer?)
```

Compute logical “or” of `dst` and `src` and put the result in `dst`

```
(struct Xor (dst src))
  dst : (or/c register? offset?)
  src : (or/c register? offset? exact-integer?)
```

Compute logical “xor” of `dst` and `src` and put the result in `dst`

```
(struct Sal (dst i))
  dst : register?
  i : (integer-in 0 63)
```

Shift `dst` to the left `i` bits and put the result in `dst`. Leftmost bits are discarded

```
(struct Sar (dst i))
  dst : register?
  i : (integer-in 0 63)
```

Shift `dst` to the right `i` bits and put the result in `dst`. Rightmost bits are discarded

```
(struct Push (a1))
  a1 : (or/c exact-integer? register?)
```

Decrements stack pointer and stores the source operand on top of the stack

```
(struct Pop (a1))
  a1 : register?
```

Loads the value of the top of the stack to the destination operand and then increments the stack pointer

```
(struct Lea (dst x))
  dst : (or/c register? offset?)
  x : label?
```

Loads the address of the given label into `dst`

3.4 From a86 to x86

```
(asm-string is) -> string?
is : (listof instruction?)
```

Converts an a86 program to a string in nasm syntax

3.5 Interpreter for a86

```
(asm-interp is) → integer?
is : (listof instruction?)
```

Assemble, link, and execute an a86 program

- **NOTE:** programs don't have to start with `'entry`. The interpreter will jump to the first label in the program

```
(current-objs) → (listof path-string?)
(current-objs objs) → void?
  objs : (listof path-string?)
= '()
```

Links a list of paths to object files to be interpreted that will be linked during `asm-interp`

4 Abscond: a Language of Numbers

Specification of a programming language consists of

- **Syntax:** specifies the **form** of programs
- **Semantics:** specifies the **meaning** of programs

In **Abscond**, the only expressions are integer literals (so running a program just produces an integer)

4.1 Abscond Syntax

The **parser** constructs an **Abstract Syntax Tree** from the textual representation of a program. Consists of two phases:

- Converts a stream of textual input into an s-expression (relies on **read** function to convert strings into s-expressions)
- Converts an s-expression into an instance of a datatype representing expressions (**AST**)

Grammar of expressions in Abscond is very simple:

$$e ::= integer$$

So a complete Abscond program looks like

```
#lang racket
42
```

4.2 Abstract Syntax for Abscond

We use an AST datatype for representing expressions and another for syntactic categories

- Each category has an appropriate constructor. For Abscond, since all expressions are integers, we have a single constructor: `Int`

$$e ::= (Int\ i)$$

$$i ::= integer$$

We can define a datatype for representing expressions like so

```
#lang racket
(provide Int)

;; type Expr = (Int Integer)
(struct Int (i) #:prefab)
```

The parser for Abscond checks that a given s-expression is an integer and construct an instance of the AST datatype. Otherwise it signals an error

```
#lang racket
(provide parse)
(require "ast.rkt")

;; S-Expr -> Expr
(define (parse s)
  (match s
    [(? integer?) (Int s)]
    [_ (error "Parse error")]))
```

4.3 Meaning of Abscond Programs

Operational Semantics defines the meaning of Abscond programs

Interpreter computes the meaning of the program

For abscond, the program itself is just a number. We can write an interpreter that consumes an expression and produces its meaning

```
#lang racket
(provide interp)
(require "ast.rkt")

;; Expr -> Integer
;; Interpret given expression
(define (interp e)
  (match e
    [(Int i) i]))
```

We can provide a formal definition of Abscond using **operational semantics**, a mathematical way of characterizing the meaning of programs

- We define semantics of Abscond as a *binary relation* between programs and their meanings. In Abscond this will be a set of pairs of expressions and integers
- The relation will be defined inductively using *inference rules*

$$A[(Int\ i), i]$$

Here we define a binary relation, called A , that says every integer literal expression is paired with the integer itself. So $((Int\ 2), 2) \in A$ and $((Int\ 5), 5) \in A$

- Here the inference rule defines the *evidence* for being in the relation
- The rule uses *meta-variables* drawn from the non-terminals of the language grammar
- A pair is in the relation if you can construct an instance of the rule

Interpreter Correctness: for all expressions e and integers i , if $(e, i) \in A$, then $(interp\ e) = i$

4.4 Compiler for Abscond

Compiler is an implementation of a programming language. However, it stages the work of interpreting a program into 2 phases

- Translating the source language into a target language (**compile-time**)
- Running the program in the target language (**run-time**)

The general relationship between interpreter and compiler is

$(\text{source-interp } e) = (\text{target-interp } (\text{source-compile } e))$

We will now write a compiler for Abscond. The target language will be x86-64 assembly.

AST representation of the 42 example is

```
(list (Label 'entry)
      (Mov 'rax 42)
      (Ret))
```

Compile function is

```
#lang racket
(provide compile)
(require "ast.rkt" a86/ast)

;; Expr -> Asm
(define (compile e)
  (prog (Global 'entry)
        (Label 'entry)
        (compile-e e)
        (Ret)))

;; Expr -> Asm
(define (compile-e e)
  (match e
    [(Int i) (seq (Mov 'rax i))]))
```

The result of `compile` can be passed into `asm-string` to convert it into concrete NASM syntax to run

4.4.1 Comparison of Interpreter and Compiler Approaches

The interpreter approach requires all of Racket in the run-time system. When a program is running, we have to

- Parse the Abscond program
- Check the syntax of the program
- Run the interpreter
- Print the result

The compiler approach has parsing and syntax checking occur at *compile-time*. When the program is running, this work will have already been done. Now we don't need Racket available during run-time, we just need the small object file compiled from C

4.5 Correctness

Compiler Correctness: For all expressions e and integers i , if $(e, i) \in A$ then $(\text{asm-interp } (\text{compile } e)) = i$

Basically, we want the compiler to capture the operational semantics of our language

- We can test the compiler against the interpreter (which we assume to be correct with respect to semantics)
- If the compiler and interpreter agree on all possible inputs, then the compiler is correct with respect to semantics
- $(\text{interp } e) = (\text{asm-interp } (\text{compile } e))$