CMSC430: Introduction to Compilers

Michael Li

Contents

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`
- **Concrete Syntax**: Defines the "shape of the source program code"
- **Parsers**: Takes concrete syntax and produces an **Abstract Syntax Tree**
- **Grammar**: Defines what things you can express with concrete syntax
- **Note**: This does not necessarily have to be from higher-level program to a lower-level program (e.g. compilers exist that take C to javascript)
- 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

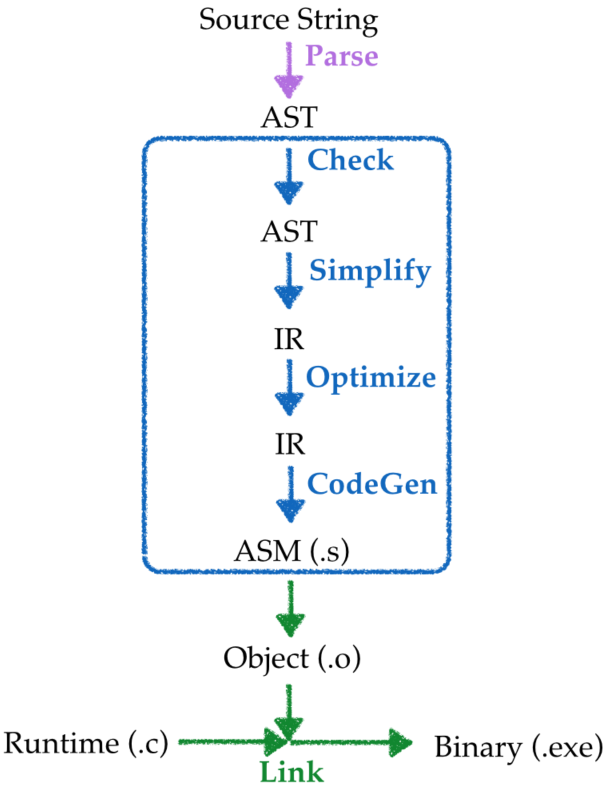

Figure 1: Compiler Pipeline

Few issues with the diagram above:

- Doesn't tell when compile errors occur
- Doesn't tell where warnings come from

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 shorthanded 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 it's 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 laef)
                    (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("%" PRId64 "\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 defintion 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
- assemblying 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
           (Jmp '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
           (Jmp '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 oft he 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 Interpreter vs Compiler Takeaways

Repeat of what is said above but worth reiterating as a reference point for the next few sections

Syntax: specifies form of programs

Operational Semantics: specifies meaning of programs

Interpreter: computes meaning of a program

Compiler maps input code in one language to output code in another language.

- Input parsed into **AST**
  - Convert input into an s-expression
  - Convert s-expression into an instance of AST
- Input syntax is checked
- Input is simplified into **Intermediate Representation**
- Program is optimized
- Program is linked at run-time

UPSHOT: compiler stages interpretation into 2 phases:

1. Translating original source language to another target language (**compile time**)
2. Running the program (**run-time**)

The general relationship between interpreter and compiler is

```
(source-interp e) = (target-interp (source-compile e))
```

5 Abscond: a Language of Numbers

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

5.1 Concrete Abscond Syntax

Grammar of expressions in Abscond is very simple:

$$e ::= integer$$

So a complete Abscond program looks like

```
#lang racket
42
```

5.2 Abstract Syntax for Abscond

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

- Each category has a 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")]))
```

5.3 Meaning of Abscond Programs

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]))
```

The following wrapper program can be used to interpret Abscond programs saved in files:

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

;; String -> Void
;; Parse and interpret contents of given filename,
;; print result on stdout
```

```racket
  (define (main fn)
    (let ([p (open-input-file fn)])
      (begin
        (read-line p) ; ignore #lang racket line
        (println (interp (parse (read p))))
        (close-input-port p))))
```

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**

$$\frac{}{A[(Int\ i), i]}$$

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 $(interp\ e) = i$

5.4 Compiler for Abscond

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

For simplicity, we write a run-time system in C that provides the main entry point for a program. The following is for the Abscond runtime

```c
#include <stdio.h>
#include <inttypes.h>
#include "print.h"

int64_t entry();

int main(int argc, char** argv) {
  int64_t result;

  result = entry();
  print_result(result);
  putchar('\n');
  return 0;
}

void print_result(int64_t x) {
  printf("%" PRId64, x);
}
```

This runtime must be linked against an object file that provides the definition of `entry`, which our compiler will provide. The `entry` function will compute the result of the Abscond code

AST representation of the `42` example is

```racket
(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

5.4.1 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

5.5 Correctness

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

6 Blackmail: Incrementing and Decrementing

Expressions in **Blackmail** include integer literals and increment/decrement operations

6.1 Concrete Syntax

Grammar of concrete expression is:

$$e ::= integer \mid (add1\ e) \mid (sub1\ e)$$

Example of a concrete program

```
#lang racket
(add1 (sub1 40))
```

6.2 Abstract Syntax

Grammar of abstract Blackmail expression is:

$$\begin{aligned} e &::= (Int\ i) \mid (\text{Prim1}\ p1\ e) \\ i &::= integer \\ p1 &::= \text{'add1} \mid \text{'sub1} \end{aligned}$$

So valid AST expressions include `(Int 0)` and `(Prim1 'add1 (Int 0))`

Datatype for representing expressions can be defined as:

```
#lang racket
(provide Int Prim1)

;; type Expr =
;; | (Int Integer)
;; | (Prim1 Op Expr)
;; type Op = 'add1 | 'sub1
(struct Int (i)     #:prefab)
(struct Prim1 (p e) #:prefab)
```

Parser for Blackmail is:

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

;; S-Expr -> Expr
(define (parse s)
  (match s
    [(? integer?) (Int s)]
    [(list (? op1? o) e) (Prim1 o (parse e))]
    [_ (error "Parse error")]))

;; Any -> Boolean
(define (op1? x)
  (memq x '(add 1 sub1)))
```

6.3 Meaning of Blackmail Programs

Meaning depends on the form of the expression

- Integer literal is the integer itself
- Increment expression is one more than the meaning of its subexpression
- Decrement expression is one less than the meaning of its subexpression

Operational semantics reflect on these dependencies by having 3 rules. One for each kind of expression

$$\frac{}{B[(Int\ i), i]} \qquad \frac{B[e_0, i_0] \quad i_1 = i_0 + 1}{B[(\text{Prim1 'add1 } e_0), i_1]} \qquad \frac{B[e_0, i_0] \quad i_1 = i_0 - 1}{B[(\text{Prim1 'sub1 } e_0), e_1]}$$

Second and third rules involve **premises** on the numerator. If the premises are true, then the **conclusion** below is true as well. Thus the rules can be understood as

- For all integeres i, $((Inti), i) \in B$
- For expressions e_0 and all integers i_0, i_1, if $(e_0, i_0) \in B$ and $i_1 = i_0 + 1$, then (Prim 1 'add1 $e_0), i_1) \in B$
- For expressions e_0 and all integers i_0, i_1, if $(e_0, i_0) \in B$ and $i_1 = i_0 - 1$, then (Prim 1 'sub1 $e_0), i_1) \in B$

These three rules make up the three cases for the interpreter

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

;; Expr -> Integer
(define (interp e)
  (match e
    [(Int i) i]
    [(Prim1 p e) (interp-prim1 p (interp e))]))
```

```
;; Op Integer -> Integer
(define (interp-prim1 op i)
  (match op
    ['add1 (add1 i)]
    ['sub1 (sub1 i)]))
```

Below are some examples

```
> (interp (Int 42))
42
> (interp (Prim1 'add1 (Int 42)))
43
```

NOTE: Given an expression e, the **interpreter** is computing the integer i such that $(e, i) \in B$ by using pattern matching to determine the form of the expression

- If e is an integer $(Int\ i)$, we are done and $(e, i) \in B$
- If e is an expression (Prim1 'add1 e_0), we recursively use the interpreter to compute i_0 such that $(e_0, i_0) \in B$. For now, we compute the RHS by adding 1 to i_0
- If e is an expression (Prim1 'sub1 e_0), we recursively use the interpreter to compute i_0 such that $(e_0, i_0) \in B$. For now, we compute the RHS by subtracting 1 from i_0

Interpreter Correctness: For all Blackmail expressions e and integers i, if $(e, i) \in B$, then `(interp e) = i`

6.4 Compiler for Blackmail

Suppose want to compile `(add1 (add1 40))`. We already know how to compile `40: (Mov 'rax 40)`. To increment (and decrement) we need to use the `add` (and `sub`) instructions. For example, to increment 40 twice,

```
    global entry
    default rel
    section .text
    global entry
entry:
    mov rax, 40
    add rax, 1
    add rax, 1
    ret
```

To accommodate for a86 instructions, we make use of `Add` and `Sub` instructions:

```
> (displayln
   (asm-string
    (list (Label 'entry)
          (Mov 'rax 40)
          (Add 'rax 1)
          (Add 'rax 1)
          (Ret))))
        global entry
        default rel
        section .text
entry:
    mov rax, 40
    add rax, 1
    add rax, 1
    ret
```

The compiler below consists of 2 functions

- Given a program, emit an entry point and return instructions
- Compile the expression

```
 #lang racket
  (provide (all-defined-out))
```

```
(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
    [(Prim1 p e) (compile-prim1 p e)]
    [(Int i)     (compile-integer i)]))

;; Op Expr -> Asm
(define (compile-prim1 p e)
  (seq (compile-e e)
       (match p
         ['add1 (Add 'rax 1)]
         ['sub1 (Sub 'rax 1)])))

;; Integer -> Asm
(define (compile-integer i)
  (seq (Mov 'rax i)))
```

6.5 Correctness

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

We can test this claim by comparing results run by the compiled and interpreted programs

```
(define (check-compiler e)
  (check-eqv? (interp e)
              (asm-interp (compile e))))
```

NOTE: TECHNICALLY a problem can arise since the integers in Blackmail are represented as 64-bit values. The problem arises when 64 bits isn't enough. Since the run-time system interprets 64-bit values as signed integers, we only have 63 bits to handle the magnitude of the integer. This is NOT an issue for the interpreter version. A few options exist to remedy this issue

- Change the spec (i.e. semantics and interpreter) to match the behavior of the compiler. This involves writing out definitions that match the behavior of the compiled code. However this isn't ideal since our program is be based on mathematics and independent of Racket
- Change the program (i.e. compiler). Doesn't work since we can't hope to represent all possible integers in just 64 bits

6.6 Looking Back, Looking Forward

This section goes over what we've discussed about for Abscond and Blackmail

Various phases of a compiler:

- **Parsing** input into an AST
  - Use `read` to parse input text into an s-expression
  - Use `parse` to convert s-expression into an AST
- **Check** to make sure code is well-formed
- **Simplified** into some convenient Intermediate Representation
  - We don't do any here; AST is the IR
- **Optimize** the program

    - We don't do any here

- **Generate** assembly x86 code
    - We use `compile` to generate assembly (in AST) form and use `asm-string` to obtain printable x86 code
- **Link** against run-time
    - Link against our run-time written in `main.c`

General steps to building a compiler

1. Build intuition through **examples**
2. Model problems with **data types**
3. Implement compiler via **type-transforming functions**
4. Validate compiler via **tests**

7 Con: Branching with Conditionals

7.1 Conditional Execution

For conditionals, we use the following concrete syntax: `(if (zero? e0) e1 e2)`, which leads to the following concrete grammar

$$e ::= \ldots \mid \text{(if (zero? e) e e)}$$

And abstract grammar

$$e ::= \ldots \mid \text{(IfZero e e e)}$$

Which can be modeled using the following definitions

```
#lang racket
(provide Int Prim1 IfZero)

;; type Expr =
;; | (Int Integer)
;; | (Prim1 Op Expr)
;; | (IfZero Expr Expr Expr)
;; type Op = 'add1 | 'sub1
(struct Int (i)           #:prefab)
(struct Prim1 (p e)       #;prefab)
(struct IfZero (e1 e2 e3) #:prefab)
```

The parser is similar to what was seen for the previous 2 languages

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

  ;; S-Expr -> Expr
  (define (parse s)
    (match s
      [(? integer?) (Int s)]
      [(list (? op1? o) e) (Prim1 o (parse e))]
      [(list 'if (list 'zero? e1) e2 e3)
       (IfZero (parse e1) (parse e2) (parse e3))]
      [_ (error "Parse error")]))

  ;; Any -> Boolean
  (define (op1? x)
    (memq x '(add1 sub1)))
```

7.2 Meaning of Con Programs

Meaning of Con programs depends on the form of the expression and if-expressions

- (IfZero e0 e1 e2) is the meaning of e1 if the meaning of e0 is 0. Otherwise the meaning is e2

Semantics is inductively defined as before. Con programs have 2 additional rules for handling if-expressions: one for when the test expression means 0 and one for when it doesn't

$$\frac{C[e_0, i_0] \quad i_0 = 0 \quad C[e_1, i_1]}{C[(\text{IfZero } e_0\, e_1\, e_2), i_1]} \qquad \frac{C[e_0, i_0] \quad i_0 \neq 0 \quad C[e_2, i_2]}{C[(\text{IfZero } e_0\, e_1\, e_2), i_2]}$$

The interpreter has an added case for if-expressions, which recursively evaluates the test expression and branches accordingly based on its value

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

  ;; Expr -> Integer
  (define (interp e)
    (match e
      [(Int i) i]
      [(Prim1 p e) (interp-prim1 p (interp e))]
      [(IfZero e1 e2 e3)
       (if (zero? (interp e1))
           (interp e2)
           (interp e3))]))

  ;; Op Integer -> Integer
  (define (interp-prim1 op i)
    (match op
      ['add1 (add1 i)]
      ['sub1 (sub1 i)]))
```

7.3 Compiler for Cons

In order to do if-expressions, we need a comparison instruction (Cmp rax 0) and a jump instruction to move to different parts of code. This is handled using the following approach

- If the comparison yields true, immediately after the comparison, jump to the code for the branch when zero
- Otherwise the next instructions will carry out the else branch and then jump over the **then** branch code

To accomplish this, we need two new labels: one for the **then** branch code and one for the end of the code branch. We can use the **gensym** function to generate symbols that have not appeared before

```
(let ((l0 (gensym 'if))
      (l1 (gensym 'if)))
  (append (compile-e e1)
          (list (Cmp 'rax 0)
                (Je l0))
          (compile-e e3)
          (list (Jmp l1)
                (Label l0))
          (compile-e e2)
          (list (Label l1))))
```

The rest of the compiler functions are identical to that of the previous languages

7.4 Correctness

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

We can test this claim by comparing results run by the compiled and interpreted programs

```
(define (check-compiler e)
  (check-eqv? (interp e)
              (asm-interp (compile e))))
```

8 Dupe: Duplicity of Types

Here we consider 2 types of data: integers and booleans

- From the syntax of Con, we make `'(zero? e)` its own expression form, allowing conditionals to be expressed as `(if e0 e1 e2)`

This leads to the following concrete grammar

$$e ::= \ldots \mid \text{boolean} \mid (\text{if } e\ e\ e) \mid (\text{zero? } e)$$

And abstract grammar

$$\begin{aligned} e &::= (\text{Int } i) \mid (\text{Bool } b) \mid (\text{Prim1 } p1\ e) \mid (\text{if } e\ e\ e) \\ p1 &::= \ldots \mid \text{'zero?} \\ v &::= i \mid b \\ b &::= \#t \mid \#f \end{aligned}$$

One thing to note is that the new nonterminal v ranges over **values** (integers and booleans)

The abstract syntax can be modelled using the following datatype definition:

```
#lang racket
(provide Int Bool Prim1 If)

;; type Expr =
;; | (Int Integer)
;; | (Bool Boolean)
;; | (Prim1 Op Expr)
;; | (If Expr Expr Expr)
;; type Op = 'add1 | 'sub1 | 'zero?
(struct Int (i)       #:prefab)
(struct Bool (b)      #:prefab)
(struct Prim1 (p e)   #:prefab)
(struct If (e1 e2 e3) #:prefab)
```

And the s-expression parser can be defined as

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

;; S-Expr -> Expr
(define (parse s)
  (match s
    [(? integer?) (Int s)]
    [(? boolean?) (Bool s)]
    [(list (? op1? o) e) (Prim1 o (parse e))]
    [(list 'if e1 e2 e3)
     (If (parse e1) (parse e2) (parse e3))]
    [_ (error "Parse error")]))

;; Any -> Boolean
(define (op1? x)
  (memq x '(add1 sub1 zero?)))
```

8.1 Meaning of Dupe Programs

The new if expression depends on whether the subexpression $e1$ evaluates to 0 or not

- `(if #f 1 2)` means 2 since $e1$ evaulated to `#f`
- `(if #t 1 2)` means `t` since $e1$ evaulated to `#t`

NOTE: here $e1$ is `(zero? e)` which means that 0 evaluates to `#t`

Another thing to consider is **type mismatches**: what should `(add1 #f)` mean? There are a few approaches

- Prohibit such programs statically with a type system (e.g. Ocaml, Java)
- Coerce values to different types (e.g. JavaScript)
- Signal a run-time error (e.g. Racket)
- Leave the behavior unspecified (e.g. C). **We use this approach for now**

Operational semantics are still binary relations between the expressions and their meaning, but now the type of relation is changed to reflect the values of Dupe (integers or booleans)

$$\frac{}{D[(\text{Int } i), i]} \qquad \frac{}{D[(\text{Bool } b), b]} \qquad \frac{D[e_0, v_0] \quad \text{D-prim}[p1, v0]}{D[(\text{Prim1 } p_1\, e_0),\, v_1)]}$$

$$\frac{D[e_0, v_0] \quad \text{is-true}[v_0] \quad D[e1, v1)]}{D[(\text{If } e_0\, e_1\, e_2),\, v_1)]} \qquad \frac{D[e_0, v_0] \quad \text{is-false}[v_0] \quad D[e2, v2)]}{D[(\text{If } e_0\, e_1\, e_2),\, v_2)]}$$

These relations rely on some helper judgements

$$\frac{}{\text{is-true}[\#t]} \qquad \frac{}{\text{is-true}[i]}$$

$$\frac{}{\text{is-false}[\#f]}$$

NOTE: Here we are following the semantics that anything NOT `#f` is considered `true` in a conditional

The semantics above also relies on a new metafunction `D-prim` ($\bot$ here means `undefined`):

$$\begin{aligned} \text{D-prim} : p1\, v &\to v \mid \bot \\ \text{D-prim}[\text{'add1}, i] &= i + 1 \\ \text{D-prim}[\text{'sub1}, i] &= i - 1 \\ \text{D-prim}[\text{'zero?}, 0] &= \#t \\ \text{D-prim}[\text{'zero?}, i] &= \#f \\ \text{D-prim}[\_, \_] &= \bot \end{aligned}$$

The above definition means that type mismatches (e.g. `(Prim1 'add1 (Bool #f))` is undefined

The interpreter follows the rules of the semantics is straightforward

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

;; type Value =
;; | Integer
;; | Boolean

;; Expr -> Value
(define (interp e)
  (match e
    [(Int i) i]
    [(Bool b) b]
    [(Prim1 p e)
     (interp-prim1 p (interp e))]
    [(If e1 e2 e3)
     (if (interp e1)
         (interp e2)
         (interp e3))]))

;; Op Value -> Value
(define (interp-prim1 op v)
  (match op
    ['add1  (add1 v)]
    ['sub1  (sub1 v)]
    ['zero? (zero? v)]))
```

Interpreter Correctness: For all Dupe expressions e and values v, if $(e, v) \in D$, then `(interp e) = v`

- **NOTE**: In the cases where the semantics is undefined, the interpreter can do whatever it wants since there is no specification

8.2 Out of One, Many

Since we are representing 2 types of data (integers and booleans), the representations should be disjoint sets of values. In x86, doing so is complicated since it doesn't have a literal notion of booleans, nor does it have a notion of disjoint datatypes; it only supports one datatype: bits

In Con, we used 64-bit integers to represent Con integers. To expand this representation to support both booleans and integers, we need to have the bits encode both the value and the type of value.

This is done by using a single (least significant) bit to represent whether a value is a boolean or an integer. The remaining 63 bits are used to represent that value (true, false, or an integer). For simplicity, we let

- `#b0` represent an integer. So the number `1` would be represented as `#b10`
- `#b1` represent a boolean. So the value `#t` is represented by `#b01` and `#f` is represented by `#b11`

Below are helper functions to convert bits to values and values to bits

```
#lang racket
(provide (all-defined-out))

;; type Value =
;; | Integer
;; | Boolean

;; type Bits = Integer

(define int-shift    1)
(define type-int   #b0)
(define type-bool  #b1)
(define val-true  #b01)
(define val-false #b11)

;; Bits -> Value
(define (bits->value b)
  (cond [(= type-int (bitwise-and b #b1))
         (arithmetic-shift b (- int-shift))]
        [(= b val-true)  #t]
        [(= b val-false) #f]
        [else (error "invalid bits")]))

;; Value -> Bits
(define (value->bits v)
  (match v
    [(? integer?) (arithmetic-shift v int-shift)]
    [#t val-true]
    [#f val-false]))
```

NOTE: the interpreter operates at the level of `Values` while the compiler operates at the level of `Bits`. We could also define an interpreter that works on bits, which will help us implement the compiler

- We want an function `interp-bits` that takes `Expr` as input and outputs `Bits`, and satisfies `(interp-bits e) = (value->bits (interp e))` for all `e`

We can design such a function by **derivation**: take a specification, perform algebraic transformations, and arrive at a correct implementation

-