CMSC 430, Feb 6th 2020

Abscond and Blackmail

First things first

First things first

- I messed up!

First things first

First things first

```
(define (get-elems bt)
   (match bt
     [(leaf) '()]
     [(node i left right)
        (cons i (append (get-elems left)
                        (get-elems right)))]))
```

First things first

```
(define (get-elems bt)
   (match bt
     [(leaf) '()]
     [(node i left right)
        (cons i (append (get-elems left)
                        (get-elems right)))]))
```

- Was correct!

First things first

First things first

- The problem was in how the functio was *called*

First things first

- The problem was in how the functio was *called*

```
sorry> (require "trees.rkt")
      (get-elems (node 1
                       (leaf)
                       (leaf)))
```

Second things second

Second things second

- One last things about quasiquoting

Second things second

- One last things about quasiquoting
- If the thing we want to **`unquote`** is a list, we can use **`unquote splicing`** to put the elements of the list directly in our structure

Second things second

- One last things about quasiquoting
- If the thing we want to **`unquote`** is a list, we can use **`unquote splicing`** to put the elements of the list directly in our structure

```
uqs> (define xs '(1 2 3))
     `(huh ,@xs)
```

Lastly, before we begin

Lastly, before we begin

- Read the lecture notes!

Lastly, before we begin

- Read the lecture notes!
  - It will be increasingly important as we progress through the course

If you see what I mean

If you see what I mean

- There are several ways of defining a language

If you see what I mean

- There are several ways of defining a language
  - By example

If you see what I mean

- There are several ways of defining a language
  - By example
  - By informal description

If you see what I mean

- There are several ways of defining a language
  - By example
  - By informal description
  - Via reference implementation

If you see what I mean

- There are several ways of defining a language
  - By example
  - By informal description
  - Via reference implementation
  - With a formal (mathematical) semantics

How it’s made

How it’s made

- C
  - Informal Description

How it's made

- OCaml
  - Defined by its implementation

How it’s made

- Standard ML
  - Fully formalized

How it's made

- Python
  - Informal Description
  - Examples
  - Mostly defined by CPython?

How it's made

- Haskell
  - Informal Description
  - Appeal to some formalism

Abscond

Abscond

- For our first language

Abscond

- For our first language
  - Formal Definition

Abscond

- For our first language
  - Formal Definition
  - Via reference implementation

Abscond

- For our first language
  - Formal Definition
  - Via reference implementation
- If everything is done right, the two should match*

Abscond's AST

Abscond's AST

- We've got expressions

Abscond’s AST

- We’ve got expressions
  - **`e ::= i`**

Abscond's AST

- We've got expressions
  - **`e ::= i`**
- We've got **`` ` i``**'s

Abscond’s AST

- We’ve got expressions
  - **`e ::= i`**
- We’ve got ` **`i`**’s
  - **`i ::=`** $\mathbb{Z}$

Abscond's AST

- We've got expressions
  - **`e ::= i`**
- We've got ` **`i`**'s
  - **`i ::=`** $\mathbb{Z}$
- That's it

Let's argue semantics

Let’s argue semantics

- Abscond has an *operational* semantics:

Let’s argue semantics

- Abscond has an *operational* semantics:
  - We relate a program to its meaning via a *relation* **`A[_,_]`**

Let’s argue semantics

- Abscond has an *operational* semantics:
  - We relate a program to its meaning via a *relation* **`A[_,_]`**
- For Abscon we have only a single instance of this relation because we only have a single kind of expression

Let's argue semantics

- Abscond has an *operational* semantics:
  - We relate a program to its meaning via a *relation* **`A[_,_]`**
- For Abscon we have only a single instance of this relation because we only have a single kind of expression
  - **`A[`** *i* **`,`** *i* **`]`**

Let’s write an interpreter!

```
abs> (define (interp e)
```

What about compilers?

What about compilers?

- Having an interpreter is useful for a few reasons (non-exhaustive):

What about compilers?

- Having an interpreter is useful for a few reasons (non-exhaustive):
    - (tend to be) easier to reason about than compilers

What about compilers?

- Having an interpreter is useful for a few reasons (non-exhaustive):
  - (tend to be) easier to reason about than compilers
  - Easier to experiment with language features

What about compilers?

- Having an interpreter is useful for a few reasons (non-exhaustive):
  - (tend to be) easier to reason about than compilers
  - Easier to experiment with language features
  - They let us ’borrow’ more from the host language

What about compilers?

- Having an interpreter is useful for a few reasons (non-exhaustive):
  - (tend to be) easier to reason about than compilers
  - Easier to experiment with language features
  - They let us ’borrow’ more from the host language
  - We can test our compiler against them! (believe me, this is helpful!)

What about compilers?

What about compilers?

- Testing against a reference interpreter:

What about compilers?

- Testing against a reference interpreter:

```
(check-eqv? (source-interp e)
            (target-interp (source-compile e)))
```

Running on our target (x86)

Running on our target (x86)

- Assume we had a compiler that could produce **x86** code

Running on our target (x86)

- Assume we had a compiler that could produce **x86** code
- Executables have to know where to start execution

Running on our target (x86)

- Assume we had a compiler that could produce **`x86`** code
- Executables have to know where to start execution
  - This is different from **`main()`**!

Running on our target (x86)

- Assume we had a compiler that could produce **`x86`** code
- Executables have to know where to start execution
  - This is different from **`main()`**!
- We need a *runtime system*

A simple runtime system

A simple runtime system

```
#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;
}
```

The object we desire

The object we desire

- Let’s run the following to get a linkable RTS
  - `gcc -m64 -c -o main.o main.c`

What do we want?

What do we want?

- Let's look at an example assembly file.

Making an AST

Making an AST

- In OCaml we'd make a few types:

Making an AST

- In OCaml we'd make a few types:
  - `type Reg = RAX`

Making an AST

- In OCaml we'd make a few types:
  - `type Reg = RAX`
  - `type Arg = Int | Reg`

Making an AST

- In OCaml we'd make a few types:
  - **`type Reg = RAX`**
  - **`type Arg = Int | Reg`**
  - **`type Lab = Symbol`**

Making an AST

- In OCaml we'd make a few types:
  - `type Reg = RAX`
  - `type Arg = Int | Reg`
  - `type Lab = Symbol`
  - `type Inst = Lab | RET | MOV Arg Arg`

Making an AST

- In OCaml we'd make a few types:
  - **`type Reg = RAX`**
  - **`type Arg = Int | Reg`**
  - **`type Lab = Symbol`**
  - **`type Inst = Lab | RET | MOV Arg Arg`**
  - **`type Asm = Inst list`**

Making an AST

- In OCaml we'd make a few types:
  - `type Reg = RAX`
  - `type Arg = Int | Reg`
  - `type Lab = Symbol`
  - `type Inst = Lab | RET | MOV Arg Arg`
  - `type Asm = Inst list`
- In Racket we will do none of that

Making an AST

- In OCaml we'd make a few types:
  - **`type Reg = RAX`**
  - **`type Arg = Int | Reg`**
  - **`type Lab = Symbol`**
  - **`type Inst = Lab | RET | MOV Arg Arg`**
  - **`type Asm = Inst list`**
- In Racket we will do none of that
  - Dynamic types!

Our first compiler

Our first compiler

```
abs> (define (compile e)
```

Our first compiler

```
abs> (define (compile e)
```

- lol

pretty-print

pretty-print

- Good: now we have the structure we want

pretty-print

- Good: now we have the structure we want
- Bad: Assemblers take flat strings, not racket structures

pretty-print

- Good: now we have the structure we want
- Bad: Assemblers take flat strings, not racket structures
- Solution: Write a pretty-printer

Settling an argument

Settling an argument

```
(define (arg->string a)
  (match a
    [`rax "rax"]
    [n (number->string n)]))
```

Settling an argument

Settling an argument

```
(define (instr->string i)
  (match i
    [`(mov ,a1 ,a2)
     (string-append "\tmov "
                    (arg->string a1) ", "
                    (arg->string a2) "\n")]
    [`ret "\tret\n"]
    [l (string-append (label->string l) ":\n")]))
```

Settling an argument

```
(define (instr->string i)
  (match i
    [`(mov ,a1 ,a2)
     (string-append "\tmov "
                    (arg->string a1) ", "
                    (arg->string a2) "\n")]
    [`ret "\tret\n"]
    [l (string-append (label->string l) ":\n")]))
```

- the rest are in the lecture notes online!

Take it for a spin

Our Second Compiler

Our Second Compiler

- Let’s add a feature to our compiler: incrementing and decrementing.

Our Second Compiler

- Let's add a feature to our compiler: incrementing and decrementing.
- We'll call it blackmail

Blackmail's AST

Blackmail’s AST

- We’ve got expressions

Blackmail's AST

- We've got expressions
  - `e ::= i | add1 e | sub1 e`

Blackmail's AST

- We've got expressions
  - `e ::= i | add1 e | sub1 e`
- We've got ` **`i`**'s

Blackmail's AST

- We've got expressions
  - `e ::= i | add1 e | sub1 e`
- We've got ` `i`'s
  - `i ::= ℤ`

Blackmail's AST

- We've got expressions
  - **`e ::= i | add1 e | sub1 e`**
- We've got ` **`i`**'s
  - **`i ::=`** $\mathbb{Z}$
- And we've got two functions:

Blackmail's AST

- We've got expressions
  - **`e ::= i | add1 e | sub1 e`**
- We've got ` **`i`**'s
  - **`i ::=`** $\mathbb{Z}$
- And we've got two functions:
  - **`add1 :`** $\mathbb{Z} \rightarrow \mathbb{Z}$

Blackmail's AST

- We've got expressions
  - **`e ::= i | add1 e | sub1 e`**
- We've got ` **`i`**'s
  - **`i ::=`** $\mathbb{Z}$
- And we've got two functions:
  - **`add1 :`** $\mathbb{Z} \rightarrow \mathbb{Z}$
  - **`sub1 :`** $\mathbb{Z} \rightarrow \mathbb{Z}$

Blackmail's AST

- We've got expressions
  - **`e ::= i | add1 e | sub1 e`**
- We've got ` **`i`**'s
  - **`i ::=`** $\mathbb{Z}$
- And we've got two functions:
  - **`add1 :`** $\mathbb{Z} \rightarrow \mathbb{Z}$
  - **`sub1 :`** $\mathbb{Z} \rightarrow \mathbb{Z}$
- That's it

It’s dangerous to go alone

It's dangerous to go alone

- In Abscond, it was only integers, parsing was trivial.

It’s dangerous to go alone

- In Abscond, it was only integers, parsing was trivial.
  - Now we have to make sure what we have is actually an expression.

It’s dangerous to go alone

- In Abscond, it was only integers, parsing was trivial.
    - Now we have to make sure what we have is actually an expression.

```
(define (expr? x)
  (match x
    [(? integer? i) #t]
    [`(add1 ,x) (expr? x)]
    [`(sub1 ,x) (expr? x)]
    [_ #f]))
```

It’s dangerous to go alone

- In Abscond, it was only integers, parsing was trivial.
  - Now we have to make sure what we have is actually an expression.

```
(define (expr? x)
  (match x
    [(? integer? i) #t]
    [`(add1 ,x) (expr? x)]
    [`(sub1 ,x) (expr? x)]
    [_ #f]))
```

- As mentioned on Tuesday, since we don’t have static types, we can use validation like the above to make sure our values are well formed

Blackmail is all about interpretation

Blackmail is all about interpretation

- In Abscond, interpreter was ’trivial’

Blackmail is all about interpretation

- In Abscond, interpreter was ’trivial’
    - For blackmail we have to think a bit more

Blackmail is all about interpretation

- In Abscond, interpreter was ’trivial’
  - For blackmail we have to think a bit more

```
(define (interp e)
  (match e
    [(? integer? i) i]
    [`(add1 ,e0)
     (match (interp e0)
       [i0 (+ i0 1)])]
    [`(sub1 ,e0)
     (match (interp e0)
       [i0 (- i0 1)])]))
```

Seeing how blackmail feels

What’s different about compilation?

What’s different about compilation?

- Runtime system?

What’s different about compilation?

- Runtime system?
- What about entry?

What’s different about compilation?

- Runtime system?
- What about entry?
- What about return?

What’s different about compilation?

- Runtime system?
- What about entry?
- What about return?

```
(define (compile e)
  (append '(entry)
          (compile-e e)
          '(ret)))
```

compile-e coyote

compile-e coyote

- Take a deep breath

compile-e coyote

- Take a deep breath

```
(define (compile-e e)
  (match e
    [(? integer? i) `((mov rax ,i))]
    [`(add1 ,e0)
     (let ((c0 (compile-e e0)))
       `(,@c0
         (add rax 1)))]
    [`(sub1 ,e0)
     (let ((c0 (compile-e e0)))
       `(,@c0
         (sub rax 1)))]))
```

Seeing how compiled blackmail feels

Assignment 2

- Details on the website