CMSC 430, Feb 11th 2020

Con

First things first

First things first

- Reflection on what a compiler *is*

Recap

Recap

- Compilers translate a *source language* to some *target language*

Recap

Recap

- In this class we will have *many* source lanagues

Recap

- In this class we will have *many* source lanagues
- We will only have one target language

Our languages so far:

Our languages so far:

- The two languages so far are quite limited but still interesting

Our languages so far:

- The two languages so far are quite limited but still interesting
  - We could extend the runtime system to allow some sort of integer-IO

Our languages so far:

- The two languages so far are quite limited but still interesting
  - We could extend the runtime system to allow some sort of integer-IO
  - We could imagine `finishing up' a calculator-like language

Our languages so far:

- The two languages so far are quite limited but still interesting
  - We could extend the runtime system to allow some sort of integer-IO
  - We could imagine `finishing up' a calculator-like language
- However there are a few things that, without them, we'd be hamstrung in developing more sophisticated languages

Our languages so far:

- The two languages so far are quite limited but still interesting
  - We could extend the runtime system to allow some sort of integer-IO
  - We could imagine `finishing up' a calculator-like language
- However there are a few things that, without them, we'd be hamstrung in developing more sophisticated languages
  - We'd like to be able to name things: variables

Our languages so far:

- The two languages so far are quite limited but still interesting
  - We could extend the runtime system to allow some sort of integer-IO
  - We could imagine `finishing up' a calculator-like language
- However there are a few things that, without them, we'd be hamstrung in developing more sophisticated languages
  - We'd like to be able to name things: variables
  - We'd like to be able to *make decisions*, i.e. perform branching

Language du jour

Language du jour

- We will look at naming things next week

Language du jour

- We will look at naming things next week
- Today, we will look at branching via conditionals

Language du jour

- We will look at naming things next week
- Today, we will look at branching via conditionals
  - Because we want to focus on the branching aspect, we will not introduce booleans (yet!)

Language du jour

- We will look at naming things next week
- Today, we will look at branching via conditionals
  - Because we want to focus on the branching aspect, we will not introduce booleans (yet!)
  - Instead we will allow only a single predicate, that we define up-front

Con

Con

- Our language **`Con`** is going to extend **`blackmail`** with only one new syntactic feature

Con’s AST

Con's AST

- We've got expressions

Con's AST

- We've got expressions
  - **`e ::= i | add1 e | sub1 e | if (zero? e) e e`**

Con’s AST

- We’ve got expressions
  - **`e ::= i | add1 e | sub1 e | if (zero? e) e e`**
- Everything works, as before...

Con's AST

- We've got expressions
  - **`e ::= i | add1 e | sub1 e | if (zero? e) e e`**
- Everything works, as before...
  - but now we can decide between two programs depending on whether some expression results in **`0`**

Con's AST

- We've got expressions
  - **`e ::= i | add1 e | sub1 e | if (zero? e) e e`**
- Everything works, as before...
  - but now we can decide between two programs depending on whether some expression results in **`0`**
- Important Point:

Con's AST

- We've got expressions
  - **`e ::= i | add1 e | sub1 e | if (zero? e) e e`**
- Everything works, as before...
  - but now we can decide between two programs depending on whether some expression results in **`0`**
- Important Point:
  - This does not mean we have booleans!

Part-n Parse-L

Part-n Parse-L

- Extending our parser/validator is not too difficult

Part-n Parse-L

- Extending our parser/validator is not too difficult

```
(define (expr? x)
  (match x
    [(? integer? i) #t]
    [`(add1 ,x) (expr? x)]
    [`(sub1 ,x) (expr? x)]
    [`(if (zero? ,x) ,y ,z)
     (and (expr? x)
          (expr? y)
          (expr? z))]
    [_ #f]))
```

What does it mean?

What does it mean?

- This is a job for semantics

Some Antics

Some Antics

- The meaning of integers is unchanged since **`abscond`**

Some Antics

- The meaning of integers is unchanged since **`abscond`**

$$\frac{}{\boldsymbol{C}[\![i,\, i]\!]}$$

Some Antics

Some Antics

- The meaning of **`add1/sub1`** is unchanged since **`blackmail`**

Some Antics

- The meaning of **`add1/sub1`** is unchanged since **`blackmail`**

$$\frac{\boldsymbol{C}[\![e_0,\, i_0]\!] \quad i_1 = i_0 + 1}{\boldsymbol{C}[\![(\mathsf{add1}\ e_0),\, i_1]\!]}$$

Some Antics

- The meaning of **`add1/sub1`** is unchanged since **`blackmail`**

$$\frac{\boldsymbol{C}[\![e_0, i_0]\!] \quad i_1 = i_0 + 1}{\boldsymbol{C}[\![(\text{add1 } e_0), i_1]\!]}$$

$$\frac{\boldsymbol{C}[\![e_0, i_0]\!] \quad i_1 = i_0 - 1}{\boldsymbol{C}[\![(\text{sub1 } e_0), i_1]\!]}$$

Some Antics

Some Antics

- The new stuff in **con**

Some Antics

- The new stuff in **con**

$$\frac{\boldsymbol{C}[\![e_0,\, i_0]\!] \quad i_0 = 0 \quad \boldsymbol{C}[\![e_1,\, i_1]\!]}{\boldsymbol{C}[\![(\text{if } (\text{zero? } e_0)\ e_1\ e_2),\, i_1]\!]}$$

Some Antics

- The new stuff in **con**

$$\frac{\boldsymbol{C}[\![e_0,\ i_0]\!] \quad i_0 = 0 \quad \boldsymbol{C}[\![e_1,\ i_1]\!]}{\boldsymbol{C}[\![(\text{if } (\text{zero? } e_0)\ e_1\ e_2),\ i_1]\!]}$$

$$\frac{\boldsymbol{C}[\![e_0,\ i_0]\!] \quad i_0 \neq 0 \quad \boldsymbol{C}[\![e_2,\ i_2]\!]}{\boldsymbol{C}[\![(\text{if } (\text{zero? } e_0)\ e_1\ e_2),\ i_2]\!]}$$

Semantics -> Interpreter

- The interpreter can still fit on a single slide

Semantics -> Interpreter

- The interpreter can still fit on a single slide

```
(define (interp e)
  (match e
    [(? integer? i) i]
    [`(add1 ,e0)
     (+ (interp e0) 1)]
    [`(sub1 ,e0)
     (- (interp e0) 1)]
    [`(if (zero? ,e0) ,e1 ,e2)
     (if (zero? (interp e0))
         (interp e1)
         (interp e2))]))
```

Semantics -> Interpreter

- But let's just focus on the new bit:

```
(define (interp e)
  (match e
    (...)
    [`(if (zero? ,e0) ,e1 ,e2)
     (if (zero? (interp e0))
         (interp e1)
         (interp e2))]))
```

Semantics -> Interpreter

- But let's just focus on the new bit:

```
(define (interp e)
  (match e
    (...)
    [`(if (zero? ,e0) ,e1 ,e2)
     (if (zero? (interp e0))
         (interp e1)
         (interp e2))]))
```

- the **`zero?`** functions are not the same!

Semantics -> Interpreter

- But let's just focus on the new bit:

```
(define (interp e)
  (match e
    (...)
    [`(if (zero? ,e0) ,e1 ,e2)
     (if (zero? (interp e0))
         (interp e1)
         (interp e2))]))
```

- the **`zero?`** functions are not the same!
  - **`con`** has no notion of booleans (yet!)

Let’s think through two examples

- Example 1

Let's think through two examples

- Example 1

```
(if (zero? 8) 2 3)
```

Let’s think through two examples

- Example 2

Let's think through two examples

- Example 2

```
(if (zero? (add1 -1)) (sub1 2) 3)
```

Follow these instructions

- Here is a quick overview of some useful instructions:
- **`CMP`**

Follow these instructions

- Here is a quick overview of some useful instructions:
- **CMP**

```
CMP RAX, imm32
```

Follow these instructions

- Here is a quick overview of some useful instructions:
- **`CMP`**

`CMP RAX, imm32`

- imm32 sign-extended to 64-bits with RAX.
  - limit of 32 bit immediate not an issue for us (always 0)

Follow these instructions

- Here is a quick overview of some useful instructions:
- **`JMP`**

Follow these instructions

- Here is a quick overview of some useful instructions:
- **JMP**

JMP <label>

Follow these instructions

- Here is a quick overview of some useful instructions:
- **JMP**

JMP <label>

- Jump to an absolute address
  - we are going to let the assembler deal with whether it's direct of indirect

Follow these instructions

- Here is a quick overview of some useful instructions:
- **`JNE`**

Follow these instructions

- Here is a quick overview of some useful instructions:
- **JNE**

JNE <label>

Follow these instructions

- Here is a quick overview of some useful instructions:
- **JNE**

JNE <label>

- IFF **ZF!=0** jump to absolute address
  - we are going to let the assembler deal with whether it's direct of indirect

Follow these instructions

- Here is a quick overview of some useful instructions:
- **`JE`**

Follow these instructions

- Here is a quick overview of some useful instructions:
- **JE**

JE <label>

Follow these instructions

- Here is a quick overview of some useful instructions:
- **JE**

JE <label>

- IFF **ZF==0** jump to absolute address
  - we are going to let the assembler deal with whether it's direct of indirect

Let’s write it!