EECS 483
Conditionals

Announcements

1. No in-person class Monday, back on Wed
2. Next homework: Adder is out
3. Grading HW0: take a while, but autograder is most of the score

Conditionals

1 Growing the language: adding conditionals

Reminder: Every time we enhance our source language, we need to consider several things:

1. Its impact on the *concrete syntax* of the language
2. Examples using the new enhancements, so we build intuition of them
3. Its impact on the *abstract syntax* and *semantics* of the language
4. Any new or changed *transformations* needed to process the new forms
5. Executable *tests* to confirm the enhancement works as intended

Concrete Syntax

```
⟨expr⟩: ...
      | if ⟨expr⟩ : ⟨expr⟩ else: ⟨expr⟩
```

Examples

Concrete Syntax	Answer
`if 5: 6 else: 7`	6
`if 0: 6 else: 7`	7
`if sub1(1): 6 else: 7`	7

Abstract Syntax

```
enum Exp {
  ...
  If { cond: Box<Exp>, thn: Box<Exp>, els: Box<Exp> }
}
```

Semantics

EXERCISE:

extend the interpreter
to support if

Compilation

Need to be able to choose dynamically *which* instructions to execute…

Registers

RAX: return register
RSP: stack pointer

RIP: instruction pointer
FLAGS: status ~~pointer~~ register

Registers

RIP: instruction pointer

RIP

```
mov RAX, 1
mov RBX, 2
ret
```

Jumps

jmp target

Sets RIP to target
target is usually a **label**

Jumps

```
    mov RAX, 1
    mov RBX, 2
    jmp landing
skipped:
    mov RAX, 3
landing:
    ret
```

RIP →

RAX: 1

RBX: 2

Conditional Jumps & FLAGS

cmp arg1, arg2

Compares args, set FLAGS
Leaves args unchanged

Conditional Jumps & FLAGS

Instruction	Jump if ...
`je` LABEL	... the two compared values are equal
`jne` LABEL	... the two compared values are not equal
`jl` LABEL	... the first value is less than the second
`jle` LABEL	... the first value is less than or equal to the second
`jg` LABEL	... the first value is greater than the second
`jge` LABEL	... the first value is greater than or equal to the second
`jb` LABEL	... the first value is less than the second, when treated as unsigned
`jbe` LABEL	... the first value is less than or equal to the second, when treated as unsigned

Let's examine the last example above: `if sub1(1): 6 else: 7`. Which of the following could be valid translations of this expression?

```
  mov RAX, 1
  sub1 RAX
  cmp RAX, 0
  je if_false
if_true:
  mov RAX, 6
  jmp done
if_false:
  mov RAX, 7
done:
```

```
  mov RAX, 1
  sub1 RAX
  cmp RAX, 0
  je if_false
if_true:
  mov RAX, 6

if_false:
  mov RAX, 7
done:
```

```
  mov RAX, 1
  sub1 RAX
  cmp RAX, 0
  jne if_true
if_true:
  mov RAX, 6
  jmp done
if_false:
  mov RAX, 7
done:
```

```
  mov RAX, 1
  sub1 RAX
  cmp RAX, 0
  jne if_true
if_false:
  mov RAX, 7
  jmp done
if_true:
  mov RAX, 6
done:
```

```
let x = if 10:
          2
        else:
          0
in
if x:
  55
else:
  999
```

```
  mov RAX, 10
  cmp RAX, 0
  je if_false
if_true:
  mov RAX, 2
  jmp done
if_false:
  mov RAX, 0
done:
  mov [RSP-8], RAX
  mov RAX, [RSP-8]
  cmp RAX, 0
  je if_false
if_true:
  mov RAX, 55
  jmp done
if_false:
  mov RAX, 999
done:
```

```
$ nasm -f elf64 -o output/test1.o output/test1.s
output/test1.s:20: error: symbol `if_true' redefined
output/test1.s:23: error: symbol `if_false' redefined
output/test1.s:25: error: symbol `done' redefined
```

Unique Name Generation

1. Use a counter during codegen
2. Tag nodes with unique ids before codegen

```
enum Exp<Ann> {
    Num(i64, Ann),
    Prim1(Prim1, Box<Exp<Ann>>, Ann),
    Var(String, Ann),
    Let { bindings: Vec<(String, Exp<Ann>)>,
          body: Box<Exp<Ann>>,
          ann: Ann
        }
    If { cond: Box<Exp<Ann>>, thn: Box<Exp<Ann>>, els: Box<Exp<Ann>>, ann: Ann }
}
```

```
type Tag = u64;

fn tag<Ann>(e: &Exp<Ann>) -> Exp<Tag> {
   tag_help(e, &mut 0)
}
fn tag_help<Ann>(e: &Exp<Ann>, counter: &mut Tag) -> Exp<Tag> {
    let cur_tag = *counter;
    *counter += 1;
    match e {
        Exp::Prim1(op, e, _) => Exp::Prim1(*op, Box::new(tag_help(e, counter)), cur_tag),
        ...
    }
}
```

```
fn compile_with_env<'exp>(e: &'exp Expr<Span>, mut env: Vec<(&'exp str, i32)>) -> Result<Vec<I
    match e {
        Exp::If { cond, thn, els, ann } => {
            let else_lab = format!("if_false#{}", ann);
            let done_lab = format!("done#{}", ann);

            let mut is = compile_with_env(cond, env.clone())?;
            is.push(Instr::Cmp(BinArgs::ToReg(Reg::Rax, Arg32::Imm(0))));
            is.push(Instr::Je(else_lab.clone()));
            is.extend(compile_with_env(thn, env.clone())?);
            is.push(Instr::Jmp(done_lab.clone()));
            is.push(Instr::Label(else_lab.clone()));
            is.extend(compile_with_env(els, env)?);
            is.push(Instr::Label(done_lab));
            Ok(is)
        }
    ...
    }
}
pub fn compile_to_string(e: &Exp<Span>) -> Result<String, CompileErr> {
    let tagged = tag(e);
    let is = compile_with_env(&tagged, Vec::new())?;
... // insert the section .text etc
}
```

Exercise

Add a new `Prim1` operator to the language, that you can recognize and *deliberately compile* into invalid assembly that crashes the compiled program. Use this side-effect to confirm that the compilation of if-expressions only ever executes one branch of the expression. *Hint*: using the `sys_exit(int)` syscall is probably helpful.