Lecture 15

CIS 341: COMPILERS
Announcements

• COVID19 Logistics
  – Changed due dates for remaining projects
  – Final exam still TBD: I will follow the university guidelines
    • my preference would be to allow a “take home” complete in your own time “homework-style” exam.
  – The university allows you to choose to take the class Pass/Fail
    • See Piazza post for ongoing discussion

• Zoom recordings uploaded in the afternoon after the lecture.

• HW4: OAT v. 1.0
  – Parsing & basic code generation
  – Due: Monday, March 30th

• HW5: now (tentatively) due: Friday, April 17th
• HW6: now (tentatively) due: Wednesday, April 29th
Midterm Results

Midterm Exam
Grades Available on Gradescope
Please ask for regrade requests no later than Tuesday, March 31st

(Out of 80 points)
Median: 60
Mean: 58.2
Std.Dev: 11.3
Untyped lambda calculus
Substitution
Evaluation

FIRST-CLASS FUNCTIONS
“Functional” languages

• Languages like ML, Haskell, Scheme, Python, C#, Java 8, Swift
• Functions can be passed as arguments (e.g. map or fold)
• Functions can be returned as values (e.g. compose)
• Functions nest: inner function can refer to variables bound in the outer function

```
let add = fun x -> fun y -> x + y
let inc = add 1
let dec = add -1

let compose = fun f -> fun g -> fun x -> f (g x)
let id = compose inc dec
```

• How do we implement such functions?
  – in an interpreter? in a compiled language?
(Untyped) Lambda Calculus

- The lambda calculus is a minimal programming language.
  - Note: we’re writing \((\text{fun } x \rightarrow e)\) lambda-calculus notation: \(\lambda x. e\)
- It has variables, functions, and function application.
  - That’s it!
  - It’s Turing Complete.
  - It’s the foundation for a lot of research in programming languages.
  - Basis for “functional” languages like Scheme, ML, Haskell, etc.

Abstract syntax in OCaml:

```ocaml
type exp =
  | Var of var (* variables *)
  | Fun of var * exp (* functions: fun x \rightarrow e *)
  | App of exp * exp (* function application *)
```

Concrete syntax:

```
exp ::= 
  | x (* variables *)
  | fun x \rightarrow exp (* functions *)
  | exp1 exp2 (* function application *)
  | ( exp ) (* parentheses *)
```
Values and Substitution

- The only values of the lambda calculus are (closed) functions:
  \[
  \text{val ::= \mid \text{fun } x \rightarrow \text{exp}} \quad \text{functions are values}
  \]

- To \textit{substitute} a (closed) value \( v \) for some variable \( x \) in an expression \( e \):
  - Replace all \textit{free occurrences} of \( x \) in \( e \) by \( v \).
  - In OCaml: written \( \text{subst } v \ x \ e \)
  - In Math: written \( e\{v/x\} \)

- Function application is interpreted by \textit{substitution}:
  \[
  (\text{fun } x \rightarrow \text{fun } y \rightarrow x + y) \ 1 \\
  = \text{subst } 1 \ x \ (\text{fun } y \rightarrow x + y) \\
  = (\text{fun } y \rightarrow 1 + y)
  \]

Note: for the sake of examples we may add integers and arithmetic operations to the “pure” untyped lambda calculus.
Lambda Calculus Operational Semantics

• Substitution function (in Math):

\[
x{v/x} = v \\
y{v/x} = y \\
(fun x \rightarrow exp){v/x} = (fun x \rightarrow exp) \\
(fun y \rightarrow exp){v/x} = (fun y \rightarrow exp{v/x}) \\
(e_1 e_2){v/x} = (e_1{v/x} e_2{v/x})
\]

(replace the free \( x \) by \( v \))
(assuming \( y \neq x \))
\((x \text{ is bound in } \text{exp})\)
(assuming \( y \neq x \))
(substitute everywhere)

• Examples:

\[
(x y){(fun z \rightarrow z z)/y} = x \ (fun z \rightarrow z z)
\]

\[
(fun x \rightarrow x y){(fun z \rightarrow z z)/y} = fun x \rightarrow x \ (fun z \rightarrow z z)
\]

\[
(fun x \rightarrow x){(fun z \rightarrow z z)/x} = fun x \rightarrow x \quad \text{// x is not free!}
\]
Free Variables and Scoping

let add = fun x → fun y → x + y
let inc = add 1

• The result of `add 1` is a function
• After calling `add`, we can’t throw away its argument (or its local variables) because those are needed in the function returned by `add`.
• We say that the variable `x` is free in `fun y → x + y`
  – Free variables are defined in an outer scope
• We say that the variable `y` is bound by “`fun y`” and its scope is the body “`x + y`” in the expression `fun y → x + y`

• A term with no free variables is called closed.
• A term with one or more free variables is called open.
Free Variable Calculation

• An OCaml function to calculate the set of free variables in a lambda expression:

```ocaml
let rec free_vars (e:exp) : VarSet.t = begin match e with
  | Var x        -> VarSet.singleton x
  | Fun(x, body) -> VarSet.remove x (free_vars body)
  | App(e1, e2)  -> VarSet.union (free_vars e1) (free_vars e2)
end
```

• A lambda expression $e$ is closed if `free_vars e` returns `VarSet.empty`

• In mathematical notation:

\[
\begin{align*}
fv(x) &= \{x\} \\
fv(\text{fun } x \rightarrow \text{exp}) &= fv(\text{exp}) \setminus \{x\} \quad (\text{‘}x\text{’ is a bound in exp}) \\
fv(\text{exp}_1 \ \text{exp}_2) &= fv(\text{exp}_1) \cup fv(\text{exp}_2)
\end{align*}
\]
Variable Capture

- Note that if we try to naively "substitute" an open term, a bound variable might capture the free variables:

\[
(fun\ x \to\ (x\ y)\)(fun\ z \to\ x)/y = fun\ x \to\ (x\ (fun\ z \to x))
\]

- Usually not the desired behavior
  - This property is sometimes called "dynamic scoping" The meaning of "x" is determined by where it is bound dynamically, not where it is bound statically.
  - Some languages (e.g. emacs lisp) are implemented with this as a "feature"
  - But: it leads to hard-to-debug scoping issues
Alpha Equivalence

• Note that the names of bound variables don't matter to the semantics
  – i.e. it doesn't matter which variable names you use, as long as you use
    them consistently:
    \[ (\text{fun } x \to y x) \text{ is the "same" as } (\text{fun } z \to y z) \]
    the choice of "x" or "z" is arbitrary, so long as we consistently
    rename them

  Two terms that differ only by consistent renaming of
  bound variables are called alpha equivalent

• The names of free variables do matter:
  \[ (\text{fun } x \to y x) \text{ is not the "same" as } (\text{fun } x \to z x) \]

  Intuitively: y an z can refer to different things from some outer scope

Students who cheat by “renaming variables” are trying to exploit alpha equivalence...
• Consider the substitution operation:
  \{e_2/x\} e_1

• To avoid capture, we define substitution to pick an alpha equivalent version of $e_1$ such that the bound names of $e_1$ don't mention the free names of $e_2$.
  – Then do the "naïve" substitution.

For example:  
\[(\text{fun } x \to (x \ y))\{(\text{fun } z \to x)/y\} \]
\[= \text{(fun } x' \to (x' (\text{fun } z \to x))\]  
\text{rename } x \text{ to } x'
Operational Semantics

Specified using just two inference rules with judgments of the form $\text{exp} \downarrow \text{val}$

- Read this notation as “program exp evaluates to value val”
- This is call-by-value semantics: function arguments are evaluated before substitution

\[
\begin{align*}
\text{v} & \downarrow \text{v} \\
\text{“Values evaluate to themselves”}
\end{align*}
\]

\[
\begin{align*}
\text{exp}_1 & \downarrow (\text{fun } x \to \text{exp}_3) & \text{exp}_2 & \downarrow \text{v} & \text{exp}_3\{\text{v}/x\} & \downarrow \text{w} \\
\text{exp}_1 \text{ exp}_2 & \downarrow \text{w} \\
\text{“To evaluate function application: Evaluate the function to a value, evaluate the argument to a value, and then substitute the argument for the function.”}
\end{align*}
\]
IMPLEMENTING THE INTERPRETER

See fun.ml
Adding Integers to Lambda Calculus

\[
\begin{align*}
\text{exp} &::= \\
& \mid \ldots \\
& \mid n \\
& \mid \text{exp}_1 + \text{exp}_2 \\
\text{val} &::= \\
& \mid \text{fun } x \rightarrow \text{exp} \\
& \mid n \\
\end{align*}
\]

- \text{constant integers}
- \text{binary arithmetic operation}
- \text{functions are values}
- \text{integers are values}
- \text{constants have no free vars.}
- \text{substitute everywhere}

\[
\begin{align*}
\text{n}\{v/x\} & = n \\
(\text{e}_1 + \text{e}_2)\{v/x\} & = (\text{e}_1\{v/x\} + \text{e}_2\{v/x\})
\end{align*}
\]

\[
\begin{align*}
\text{exp}_1 \downarrow n_1 & \quad \text{exp}_2 \downarrow n_2 \\
\hline
\text{exp}_1 + \text{exp}_2 & \downarrow (n_1 \oplus n_2)
\end{align*}
\]

Object-level ‘+’

Meta-level ‘+’
Scope, Types, and Context

STATIC ANALYSIS
Variable Scoping

• Consider the problem of determining whether a programmer-declared variable is in scope.

• Issues:
  – Which variables are available at a given point in the program?
  – Shadowing – is it permissible to re-use the same identifier, or is it an error?

• Example: The following program is syntactically correct but not well-formed. (y and q are used without being defined anywhere)

```java
int fact(int x) {
    var acc = 1;
    while (x > 0) {
        acc = acc * y;
        x = q - 1;
    }
    return acc;
}
```

Q: Can we solve this problem by changing the parser to rule out such programs?
Contexts and Inference Rules

• Need to keep track of contextual information.
  – What variables are in scope?
  – What are their types?

• How do we describe this?
  – In the compiler there's a mapping from variables to information we know about them.
Why Inference Rules?

• They are a compact, precise way of specifying language properties.
  – E.g. ~20 pages for full Java vs. 100’s of pages of prose Java Language Spec.

• Inference rules correspond closely to the recursive AST traversal that implements them

• Type checking (and type inference) is nothing more than attempting to prove a different judgment ( \( G;L \vdash e : t \) ) by searching backwards through the rules.

• Compiling in a context is nothing more than a collection of inference rules specifying yet a different judgment ( \( G \vdash src \Rightarrow target \) )
  – Moreover, the compilation judgment is similar to the typechecking judgment

• Strong mathematical foundations
  – The “Curry-Howard correspondence”: Programming Language ~ Logic, Program ~ Proof, Type ~ Proposition
  – See CIS 500 next Fall if you’re interested in type systems!
Inference Rules

• We can read a judgment $G;L \vdash e : t$ as “the expression $e$ is well typed and has type $t$”
• For any environment $G$, expression $e$, and statements $s_1$, $s_2$.

$$G;L;rt \vdash \text{if} \ (e) \ s_1 \ \text{else} \ s_2$$

holds if $G;L \vdash e : \text{bool}$ and $G;L;rt \vdash s_1$ and $G;L;rt \vdash s_2$ all hold.

• More succinctly: we summarize these constraints as an inference rule:

<table>
<thead>
<tr>
<th>Premises</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G;L \vdash e : \text{bool}$</td>
<td>$G;L;rt \vdash \text{if} \ (e) \ s_1 \ \text{else} \ s_2$</td>
</tr>
<tr>
<td>$G;L;rt \vdash s_1$</td>
<td></td>
</tr>
<tr>
<td>$G;L;rt \vdash s_2$</td>
<td></td>
</tr>
</tbody>
</table>

• This rule can be used for any substitution of the syntactic metavariables $G$, $e$, $s_1$ and $s_2$. 
Checking Derivations

• A derivation or proof tree has (instances of) judgments as its nodes and edges that connect premises to a conclusion according to an inference rule.

• Leaves of the tree are axioms (i.e. rules with no premises)
  – Example: the INT rule is an axiom

• Goal of the type checker: verify that such a tree exists.

• Example1: Find a tree for the following program using the inference rules in oat0-defn.pdf:

```plaintext
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 - x2;
return(x1);
```

Example2: There is no tree for this ill-scoped program:

```plaintext
var x2 = x1 + x1;
return(x2);
```
Example Derivation

\[
\begin{align*}
\mathcal{D}_1 & \quad \mathcal{D}_2 \quad \mathcal{D}_3 \quad \mathcal{D}_4 \\
G_0; \quad \textit{int} \vdash \text{var } x_1 = 0; \text{ var } x_2 = x_1 + x_1; x_1 = x_1 - x_2; \text{return } x_1; & \Rightarrow \textit{.}, x_1: \text{int}, x_2: \text{int} \\
\vdash \text{var } x_1 = 0; \text{ var } x_2 = x_1 + x_1; x_1 = x_1 - x_2; \text{return } x_1; & \quad \text{[STMTS]} \\
& \quad \text{[PROG]}
\end{align*}
\]

\begin{verbatim}
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 - x2;
return(x1);
\end{verbatim}
Example Derivation

\[ \mathcal{D}_1 = \frac{G_0; \cdot \vdash 0 : \text{int} \quad [\text{INT}]}{G_0; \cdot \vdash 0 : \text{int}} \quad \frac{G_0; \cdot \vdash \text{var } x_1 = 0 \Rightarrow \cdot, x_1 : \text{int} \quad [\text{DECL}]}{G_0; \cdot, \cdot \vdash \text{var } x_1 = 0; \Rightarrow \cdot, x_1 : \text{int}} \quad [\text{SDECL}] \]

\[ \mathcal{D}_2 = \frac{\cdot \vdash + : (\text{int, int}) \rightarrow \text{int} \quad [\text{ADD}]}{\frac{\cdot \vdash x_1 : \text{int} \in \cdot, x_1 : \text{int} \quad [\text{VAR}]}{G_0; \cdot, x_1 : \text{int} \vdash x_1 : \text{int} \quad [\text{BOP}]}} \quad \frac{\cdot \vdash x_1 : \text{int} \in \cdot, x_1 : \text{int} \quad [\text{VAR}]}{G_0; \cdot, x_1 : \text{int} \vdash x_1 : \text{int}} \quad \frac{G_0; \cdot, x_1 : \text{int} \vdash \text{var } x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int} \quad [\text{DECL}]}{G_0; \cdot, x_1 : \text{int} \vdash \text{var } x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int}} \quad [\text{SDECL}] \]
Example Derivation

\[ D_3 \]
\[
x_1 : \text{int} \in \cdot, x_1 : \text{int}, x_2 : \text{int};
\]
\[
\frac{}{\vdash \cdot : (\text{int}, \text{int}) \rightarrow \text{int}} \quad \text{[ADD]}
\]
\[
\frac{x_1 : \text{int} \in \cdot, x_1 : \text{int}, x_2 : \text{int}}{G_0 ; \cdot, x_1 : \text{int}, x_2 : \text{int} \vdash x_1 : \text{int}} \quad \text{[VAR]}
\]
\[
\frac{x_2 : \text{int} \in \cdot, x_1 : \text{int}, x_2 : \text{int}}{G_0 ; \cdot, x_1 : \text{int}, x_2 : \text{int} \vdash x_2 : \text{int}} \quad \text{[VAR]}
\]
\[
\frac{G_0 ; \cdot, x_1 : \text{int}, x_2 : \text{int} \vdash x_1 - x_2 : \text{int}}{G_0 ; \cdot, x_1 : \text{int}, x_2 : \text{int} \vdash x_1 - x_2 : \text{int}} \quad \text{[ASSN]}
\]

\[ D_4 \]
\[
\frac{x_1 : \text{int} \in \cdot, x_1 : \text{int}, x_2 : \text{int}}{G_0 ; \cdot, x_1 : \text{int}, x_2 : \text{int} \vdash x_1 : \text{int}} \quad \text{[VAR]}
\]
\[
\frac{G_0 ; \cdot, x_1 : \text{int}, x_2 : \text{int} \vdash \text{return } x_1 ; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int}}{G_0 ; \cdot, x_1 : \text{int}, x_2 : \text{int} \vdash \text{return } x_1 ; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int}} \quad \text{[RET]}
\]
Why Inference Rules?

• They are a compact, precise way of specifying language properties.
  – E.g. ~20 pages for full Java vs. 100’s of pages of prose Java Language Spec.

• Inference rules correspond closely to the recursive AST traversal that implements them

• Compiling in a context is nothing more an “interpretation” of the inference rules that specify typechecking*:
  \[ C \vdash e : t \]
  – Compilation follows the typechecking judgment

• Strong mathematical foundations
  – The “Curry-Howard correspondence”: Programming Language ~ Logic, Program ~ Proof, Type ~ Proposition
  – See CIS 500 next Fall if you’re interested in type systems!

*Here (and later) we’ll write context C for G;L, the combination of the global and local contexts.
Compilation As Translating Judgments

• Consider the source typing judgment for source expressions:
  \[ C \vdash e : t \]

• How do we interpret this information in the target language?
  \[ \llbracket C \vdash e : t \rrbracket = ? \]

• \[ \llbracket t \rrbracket \] is a target type
• \[ \llbracket e \rrbracket \] translates to a (potentially empty) sequence of instructions, that, when run, computes the result into some operand

• INVARIANT: if \[ \llbracket C \vdash e : t \rrbracket = ty \], operand, stream
  then the type (at the target level) of the operand is \( ty = \llbracket t \rrbracket \)
Example

- \( C \vdash 341 + 5 : \text{int} \)  
what is \( \llbracket C \vdash 341 + 5 : \text{int} \rrbracket \)?

\[
\llbracket C \vdash 341 : \text{int} \rrbracket = (\text{i64}, \text{Const 341}, []) \quad \llbracket C \vdash 5 : \text{int} \rrbracket = (\text{i64}, \text{Const 5}, [])
\]

\[
\llbracket C \vdash 341 : \text{int} \rrbracket = (\text{i64}, \text{Const 341}, []) \quad \llbracket C \vdash 5 : \text{int} \rrbracket = (\text{i64}, \text{Const 5}, [])
\]

\[
\llbracket C \vdash 341 + 5 : \text{int} \rrbracket = (\text{i64}, \%\text{tmp}, [\%\text{tmp} = \text{add i64} (\text{Const 341}) (\text{Const 5})])
\]
What about the Context?

- What is $[C]$?
- Source level C has bindings like: $x$:int, $y$:bool
  - We think of it as a finite map from identifiers to types

- What is the interpretation of C at the target level?

- $[C]$ maps source identifiers, “x” to source types and $[x]$

- What is the interpretation of a variable $[x]$ at the target level?
  - How are the variables used in the type system?

\[
\begin{align*}
\frac{x: t \in L}{G; L \vdash x : t} & \quad \text{TYP\_VAR} \\
\text{as expressions} & \quad \text{(which denote values)}
\end{align*}
\]

\[
\begin{align*}
\frac{x: t \in L \quad G; L \vdash \text{exp} : t}{G; L; rt \vdash x = \text{exp}; \Rightarrow L} & \quad \text{TYP\_ASSN} \\
\text{as addresses} & \quad \text{(which can be assigned)}
\end{align*}
\]
Interpretation of Contexts

- \([C]\) = a map from source identifiers to types and target identifiers

- **INVARIANT:**
  \(x:t \in C\) means that
  
  1. \(\text{lookup} \ [C] \ x = (t, \%id\_x)\)
  2. the (target) type of \(\%id\_x\) is \([t]^*\) (a pointer to \([t]\))
Interpretation of Variables

- Establish invariant for expressions:
  \[
  \frac{x : t \in L}{G ; L \vdash x : t} \quad \text{TYP_VAR}
  \]
  as expressions
  (which denote values)

  \[
  \frac{x : t \in L}{G ; L \vdash \text{exp} : t} \quad \text{TYP_ASSN}
  \]
  as addresses
  (which can be assigned)

  \[
  (\%\text{tmp}, [\%\text{tmp} = \text{load i64*} \ %\text{id}_x])
  \]
  where \((\text{i64, } \%\text{id}_x) = \text{lookup} \ [L] \ x\)

- What about statements?

  \[
  \frac{x : t \in L}{G ; L ; \text{rt} \vdash x = \text{exp} ; \Rightarrow L} \quad \text{TYP_ASSN}
  \]
  \[
  \text{stream} @
  \]
  \[
  [\text{store} \ [t] \ \text{opn}, [t]^* \ %\text{id}_x]
  \]
  where \((t, \ %\text{id}_x) = \text{lookup} \ [L] \ x\)
  and \([G ; L \vdash \text{exp} : t] = ([t], \ \text{opn}, \ \text{stream})\)
Other Judgments?

• Statement:
  \[ [C; rt \vdash \text{stmt} \Rightarrow C'] = [C'] , \text{stream} \]

• Declaration:
  \[ [G;L \vdash t \times = \text{exp} \Rightarrow G;L,x:t ] = [G;L,x:t], \text{stream} \]

INVARIANT: stream is of the form:

\[
\text{stream'} @ \\
[ \%id_x = \text{alloca } \llbracket t \rrbracket; \\
\text{store } \llbracket t \rrbracket \text{ opn, } \llbracket t \rrbracket^* \%id_x ]
\]

and \[ [G;L \vdash \text{exp} : t ] = ([t], \text{opn}, \text{stream'}) \]

• Rest follow similarly
COMPILING CONTROL
Translating while

• Consider translating “while(e) s”:
  – Test the conditional, if true jump to the body, else jump to the label after the body.

\[ C; \text{lpre} \vdash \text{while}(e) \; s \Rightarrow C' \] = \[ C' \],

\begin{verbatim}
lpre:
  opn = \llbracket C \vdash e : bool \rrbracket
  %test = icmp eq i1 opn, 0
  br %test, label %lpost, label %lbody
lbody:
  \llbracket C; \text{rt} \vdash s \Rightarrow C' \rrbracket
  br %lpre
lpost:
\end{verbatim}

• Note: writing \( \text{opn} = \llbracket C \vdash e : \text{bool} \rrbracket \) is pun
  – translating \( \llbracket C \vdash e : \text{bool} \rrbracket \) generates code that puts the result into \( \text{opn} \)
  – In this notation there is implicit collection of the code
Translating if-then-else

- Similar to while except that code is slightly more complicated because if-then-else must reach a merge and the else branch is optional.

\[
[C; rt \vdash \text{if} (e_1) s_1 \ \text{else} \ s_2 \Rightarrow C'] = [C']
\]

```
opn = [C \vdash e : \text{bool}]
%test = icmp eq i1 opn, 0
br %test, label %else, label %then
then:
  [C; rt \vdash s_1 \Rightarrow C']
br %merge
else:
  [C; rt s_2 \Rightarrow C']
br %merge
merge:
```
Connecting this to Code

• Instruction streams:
  – Must include labels, terminators, and “hoisted” global constants

• Must post-process the stream into a control-flow-graph

• See frontend.ml from HW4
OPTIMIZING CONTROL
Consider compiling the following program fragment:

```plaintext
if (x & !y | !w)
    z = 3;
else
    z = 4;
return z;
```

The corresponding assembly code is:

```plaintext
%tmp1 = icmp Eq [%y], 0    ; !y
%tmp2 = and [%x], [%tmp1]
%tmp3 = icmp Eq [%w], 0
%tmp4 = or [%tmp2], [%tmp3]
%tmp5 = icmp Eq [%tmp4], 0
br [%tmp4], label %else, label %then
then:
    store [%z], 3
    br [%merge]
else:
    store [%z], 4
    br [%merge]
merge:
    %tmp5 = load [%z]
    ret [%tmp5]
```
Observation

• Usually, we want the translation $[[e]]$ to produce a value
  – $[[C \vdash e : t]] = (\text{ty}, \text{operand}, \text{stream})$
  – e.g. $[[C \vdash e_1 + e_2 : \text{int}]] = (\text{i64}, \%\text{tmp}, \%\text{tmp} = \text{add} \ [e_1] [e_2])$

• But when the expression we’re compiling appears in a test, the program jumps to one label or another after the comparison but otherwise never uses the value.

• In many cases, we can avoid “materializing” the value (i.e. storing it in a temporary) and thus produce better code.
  – This idea also lets us implement different functionality too: e.g. short-circuiting boolean expressions
Idea: Use a different translation for tests

Usual Expression translation:
\[
\llbracket C \vdash e : t \rrbracket = (ty, operand, stream)
\]

Conditional branch translation of booleans, without materializing the value:
\[
\llbracket C \vdash e : \text{bool@} \rrbracket \modtrue \modfalse = \text{stream}
\]

Notes:
• takes two extra arguments: a “true” branch label and a “false” branch label.
• Doesn’t “return a value”
• Aside: this is a form of continuation-passing translation…

where
\[
\begin{align*}
\llbracket C, r t \vdash s_1 \Rightarrow C' \rrbracket &= \llbracket C' \rrbracket, \text{insns}_1 \\
\llbracket C, r t \vdash s_2 \Rightarrow C'' \rrbracket &= \llbracket C'' \rrbracket, \text{insns}_2 \\
\llbracket C \vdash e : \text{bool@} \rrbracket \text{then else} &= \text{insns}_3
\end{align*}
\]
Short Circuit Compilation: Expressions

- \([C \vdash e : bool@] \ ltrue \ lfalse = \text{insns}\)

\[
\begin{align*}
[C \vdash \text{false} : bool@] \ ltrue \ lfalse &= \text{[br %lfalse]} \\
[C \vdash \text{true} : bool@] \ ltrue \ lfalse &= \text{[br %ltrue]} \\
[C \vdash e : bool@] \ lfalse \ ltrue &= \text{insns} \\
[C \vdash !e : bool@] \ ltrue \ lfalse &= \text{insns}
\end{align*}
\]
Idea: build the logic into the translation

\[
\begin{align*}
\llbracket C \vdash e_1 : \text{bool}\rrbracket & \quad \text{ltrue right} = \text{insns}_1 \\
\llbracket C \vdash e_2 : \text{bool}\rrbracket & \quad \text{ltrue lfalse} = \text{insns}_2 \\
\llbracket C \vdash e_1 \mid e_2 : \text{bool}\rrbracket & \quad \text{ltrue lfalse} = \\
\llbracket C \vdash e_1 \& e_2 : \text{bool}\rrbracket & \quad \text{ltrue lfalse} = \\
\end{align*}
\]

where right is a fresh label
Consider compiling the following program fragment:

```c
if (x & !y | !w)
  z = 3;
else
  z = 4;
return z;
```

```assembly
%tmp1 = icmp Eq [x], 0  
br %tmp1, label %right2, label %right1

right1:
  %tmp2 = icmp Eq [y], 0  
br %tmp2, label %then, label %right2

right2:
  %tmp3 = icmp Eq [w], 0  
br %tmp3, label %then, label %else

then:
  store [z], 3  
br %merge

else:
  store [z], 4  
br %merge

merge:
  %tmp5 = load [z]  
ret %tmp5
```