Lecture 17

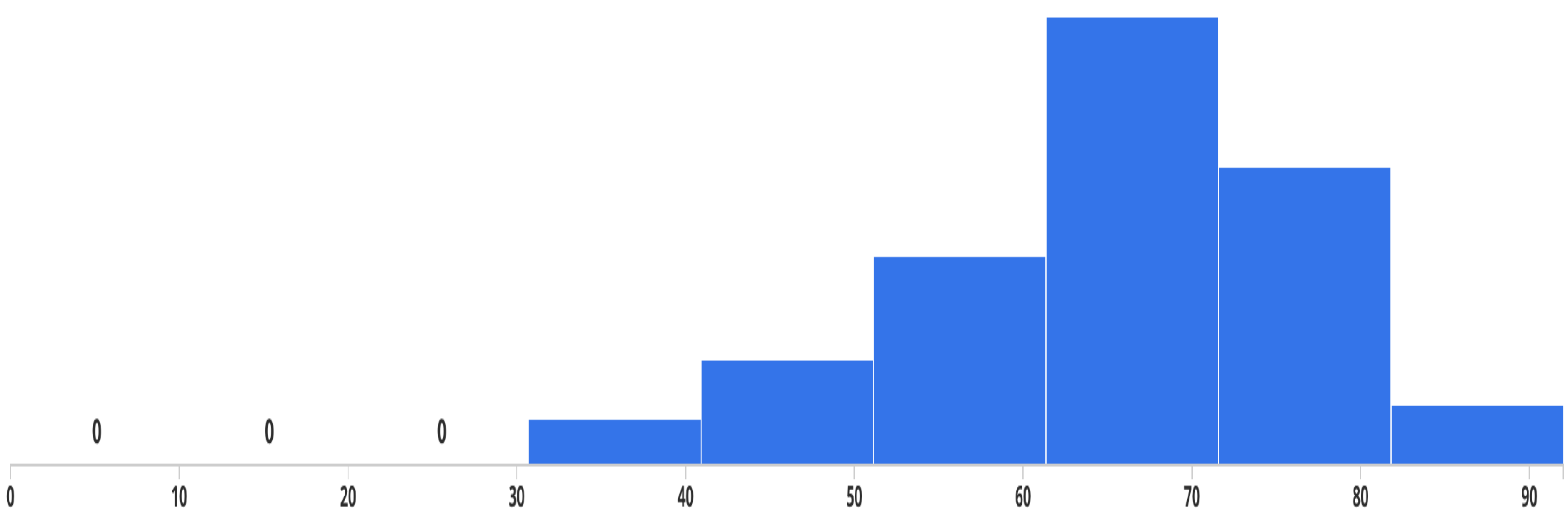
# EECS 483: COMPILER CONSTRUCTION

# Announcements

- HW4: OAT v.1.0
  - Parsing & translation to LLVM IR
  - Helps to start early!
  - **Due: Tuesday, March 26th**
- Professor New (Me): taking family leave soon, guest lectures starting in 1 or 2 weeks

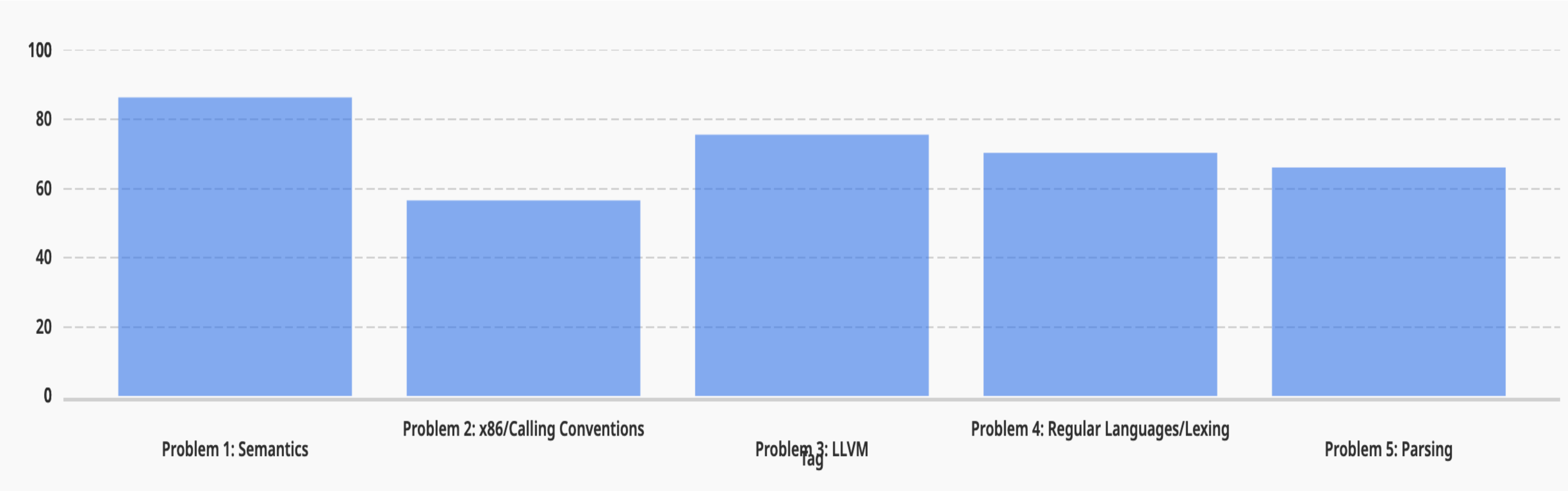
# Midterm

- Average: 65.26/92 = 70.9%
- Median: 66.25/92 = 72.0%
- Std. Dev.: = 13%
- Will curve median grade to B+



# Midterm

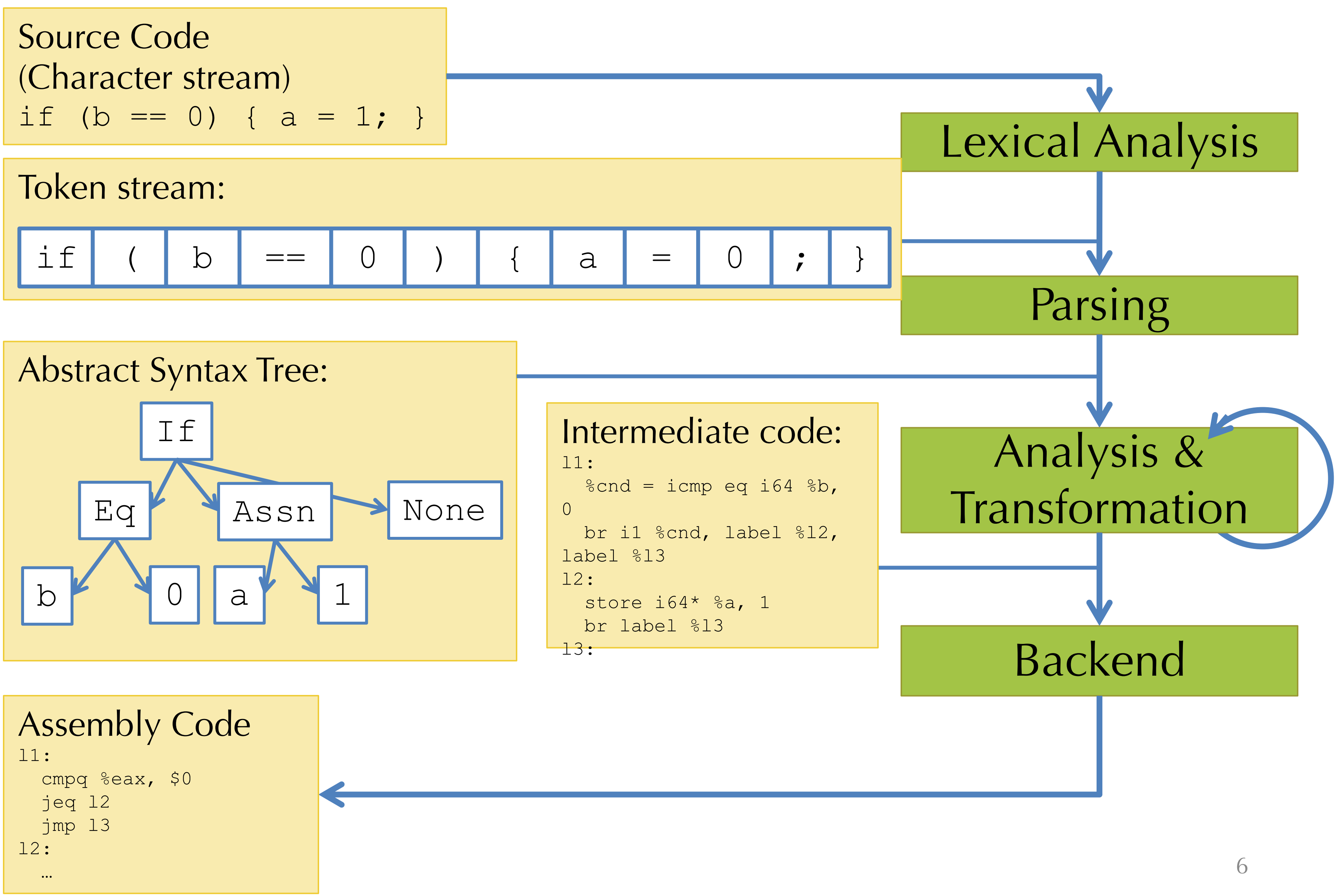
- Breakdown by question
  - All Qs received at least one perfect score answer
  - Lots of partial credit
  - Questions with lowest median:
    - Parsing T/F Questions
    - CBPV function call
    - LLVM DAGs vs Trees
    - Bounding number of states in NFA -> DFA conversion
    - LL(1) grammar



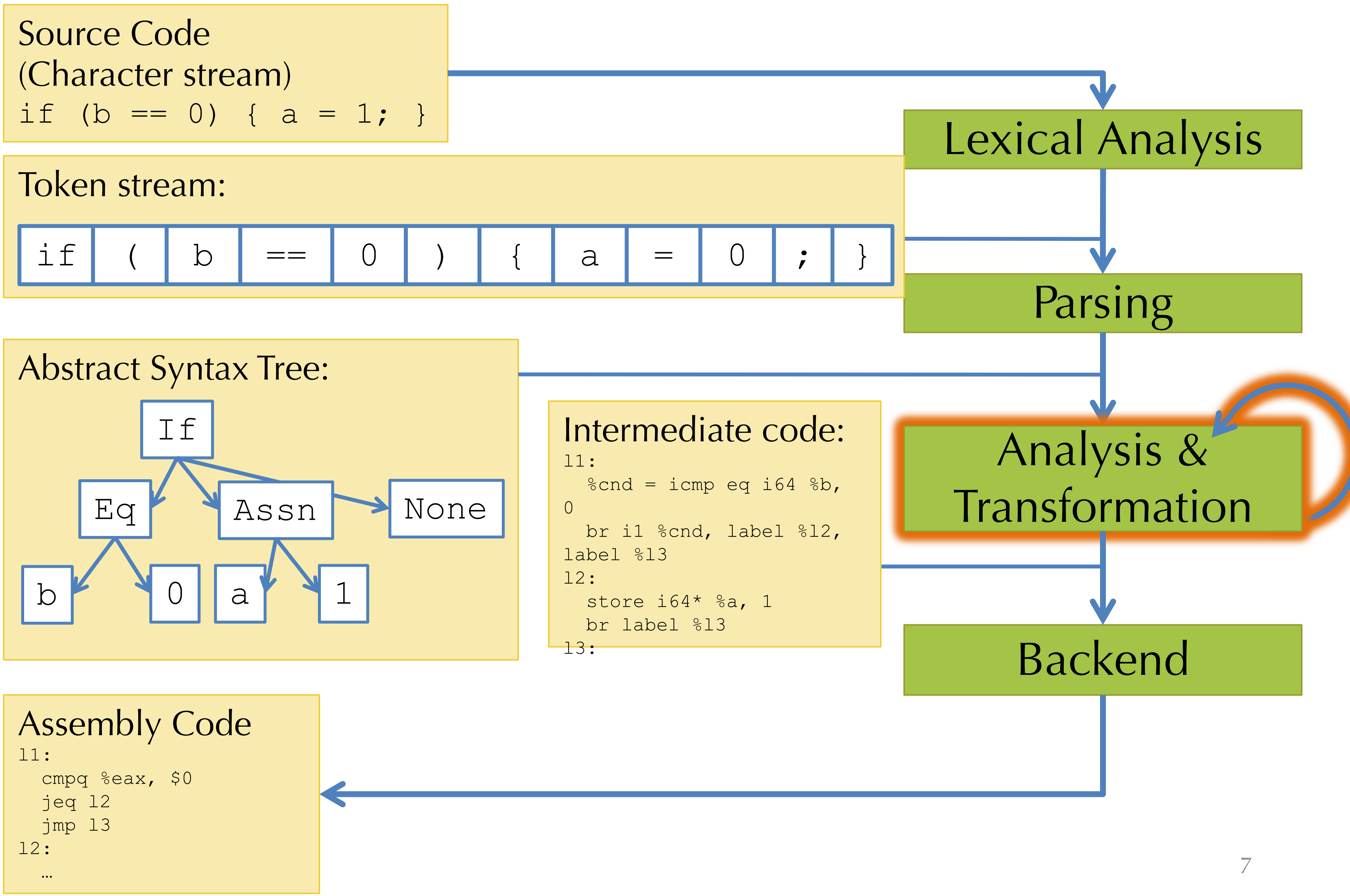
Scope, Types, and Context

# SEMANTIC ANALYSIS

# Compilation in a Nutshell



# Most of the Remainder of the Course



# Static Program Analysis

- **Static** means the program is analyzed at **compile-time**
- Used for two main purposes in the compiler:
  - Last stage of the frontend: "Type checking" or "Semantic Analysis"
    - Not every program that passes parsing is valid
      - `int main() { return x; }`
      - `int main() { return "hello world"; }`
    - If the type checker fails, the program is rejected, like a parse error
    - After the program passes the frontend, we consider it well-formed and will compile it.
  - During optimization: "static analysis"
    - We can do more optimizations if we know more about the program
    - Are these equivalent programs?
      - `int main() { int y = f(); return 0; }`
      - `int main() { return 0; }`
    - We can optimize the first to the second if we establish that f is side-effect free.
    - Since they take place after the frontend, the analysis never rejects the program
- Next few weeks: type checking, after that optimization and analyis

# 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)

```
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?

# Type Checking as Grammar

| | Specification | Implementation |
|---|---|---|
| Lexing | Regular Expressions | DFA |
| Parsing | CFG<br>LL(1) grammars<br>LR(1) grammars | Pushdown automata<br>Recursive descent<br>Shift/reduce parser |
| Type checking | Inference rules | Manual recursive descent<br>Constraint solving |

# Inference Rules

- We can read a judgment $G \vdash e$ as
  "the expression e is well scoped and has free variables in G"
- For any environment G, expression e, and statements $s_1$, $s_2$.

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

  holds if $G \vdash e$ and $G \vdash s_1$ and $G \vdash s_2$ all hold.
- More succinctly: we summarize these constraints as an ***inference rule***:

**Premises**

**Conclusion**

$$\frac{G \vdash e \qquad\qquad G \vdash s_1 \qquad\qquad G \vdash s_2}{G \vdash \text{if (e) } s_1 \text{ else } s_2}$$

- Such a rule can be used for *any* substitution of the syntactic metavariables G, e, $s_1$ and $s_2$.

# Judgments

- A ***judgment*** is a (meta-syntactic) notation that *names* a relation among one or more sets.
  - The sets are usually built from object-language syntax elements and other "math" sets (*e.g.*, integers, natural numbers, *etc.*)
  - We usually describe them using metavariables that range over the sets.
  - Often use domain-specific notation to ease reading.
  - The meaning of judgments, *i.e.*, which sets they represent, is defined by (collections of) inference rules

- Example: When we say "$G \vdash e$ is a judgment where G is a context of variables and e is a term, defined by these [...] inference rules" that is shorthand for this "math speak":

  - Let Var be the set of all (syntactic) variables
  - Let Exp be the set {e | e is a term of the untyped lambda calculus}
  - Let $\mathcal{P}$(Var) be the (finite) powerset of variables (set of all finite sets)
  - Define *well-scoped* ⊆ ($\mathcal{P}$(Var), Exp) to be a relation satisfying the properties defined by the associated inference rules [...]
  - Then "$G \vdash e$" is notation that means that (G, e) ∈ *well-scoped*

# Scope-Checking Lambda Calculus

- Consider how to identify "well-scoped" lambda calculus terms
  - Given: G, a *set* of variable identifiers, e, a term of the lambda calculus
  - *Judgment*: $G \vdash e$ "the free variables of e are included in G"

$$\frac{x \in G}{G \vdash x}$$

"the variable x is free, but in scope"

$$\frac{G \vdash e_1 \qquad G \vdash e_2}{G \vdash e_1\ e_2}$$

"G contains the free variables of $e_1$ and $e_2$"

$$\frac{G \cup \{x\} \vdash e}{G \vdash \text{fun } x \rightarrow e}$$

"x is available in the function body e"

# Scope-checking Code

- Compare the OCaml code to the inference rules:
  - structural recursion over syntax
  - the check either "succeeds" or "fails"

```
let rec scope_check (g:VarSet.t) (e:exp) : unit =
  begin match e with
  | Var x -> if VarSet.member x g then () else failwith (x ^ "not in scope")
  | App(e1, e2) -> ignore (scope_check g e1); scope_check g e2
  | Fun(x, e)   -> scope_check (VarSet.union g (VarSet.singleton x)) e
  end
```

$$\frac{x \in G}{G \vdash x}\ \boxed{\text{VAR}} \qquad \frac{G \vdash e_1 \qquad G \vdash e_2}{G \vdash e_1\ e_2}\ \boxed{\text{APP}} \qquad \frac{G \cup \{x\} \vdash e}{G \vdash \text{fun } x \to e}\ \boxed{\text{FUN}}$$

- The inference rules are a *specification* of the intended behavior of this scope checking code.
  - they don't specify the order in which the premises are checked

# Example Derivation Tree

VAR — x ∈ {x,y} / {x, y} ⊢ x

VAR — y ∈ {x,y} / {x, y} ⊢ y

APP — {x, y} ⊢ x y

FUN — {x} ⊢ fun y -> x y

VAR — z ∈ {z} / {z} ⊢ z

FUN — ⊢ (fun z -> z)

APP — {} ⊢ (fun x -> fun y -> x y)

{} ⊢ (fun x -> fun y -> x y) (fun z -> z)

- Note: the OCaml function **scope_check** verifies the existence of this tree. The structure of the recursive calls when running **scope_check** is the same shape as this tree!
- Note that x ∈ E is implemented by the function **VarSet.mem**

# Example Failed Derivation

$$
\textsf{APP}\ \dfrac{
\textsf{FUN}\ \dfrac{
\textsf{FUN}\ \dfrac{
\textsf{APP}\ \dfrac{
\cancel{\textsf{VAR}}\ \dfrac{z \in \{x,y\}}{\{x, y\} \vdash z}
\qquad
\textsf{VAR}\ \dfrac{y \in \{x,y\}}{\{x, y\} \vdash y}
}{\{x, y\} \vdash z\ y}
}{\{x\} \vdash \text{fun } y \text{ -> } z\ y}
}{\{\} \vdash (\text{fun } x \text{ -> fun } y \text{ -> } z\ y)}
\qquad
\textsf{FUN}\ \dfrac{\textsf{VAR}\ \dfrac{z \in \{z\}}{\{z\} \vdash z}}{\vdash (\text{fun } z \text{ -> } z)}
}{\{\} \vdash (\text{fun } x \text{ -> fun } y \text{ -> } z\ y)\ (\text{fun } z \text{ -> } z)}
$$

- This program is *not* well scoped
  - The variable z is not bound in the body of the left function.
  - The typing derivation fails because the VAR rule cannot succeed
  - (The other parts of the derivation are OK, though!)

# Uses of the inference rules

- We can do proofs by induction on the structure of the derivation.
- For example:

**Lemma:** If $G \vdash e$ then $fv(e) \subseteq G$.

Proof.

By induction on the derivation that $G \vdash e$.

$$\frac{x \in G}{G \vdash x}$$

- case: VAR then we have $e = x$ (for some variable x) and $x \in G$. But $fv(e) = fv(x) = \{x\}$, but then $\{x\} \subseteq G$.

$$\frac{G \vdash e_1 \quad G \vdash e_2}{G \vdash e_1\ e_2}$$

- case: APP then we have $e = e_1\ e_2$ (for some $e_1\ e_2$) and, by induction, we have $fv(e_1) \subseteq G$ and $fv(e_2) \subseteq G$, so $fv(e_1\ e_2) = fv(e_1) \cup fv(e_2) \subseteq G$

$$\frac{G \cup \{x\} \vdash e_1}{G \vdash \text{fun } x \rightarrow e_1}$$

- case: FUN then we have $e = (\text{fun } x \text{ -> } e_1)$ for some $x$, $e_1$ and, by induction, we have $fv(e_1) \subseteq G \cup \{x\}$, but then we also have $fv(\text{fun } x \text{ -> } e_1) = fv(e_1) \setminus \{x\} \subseteq ((G \cup \{x\}) \setminus \{x\}) \subseteq G$

| | | | |
|---|---|---|---|
| $fv(x)$ | = | $\{x\}$ | |
| $fv(\text{fun } x \rightarrow exp)$ | = | $fv(exp) \setminus \{x\}$ | *('x' is a bound in exp)* |
| $fv(exp_1\ exp_2)$ | = | $fv(exp_1) \cup fv(exp_2)$ | |

See tc.ml

# STATICALLY RULING OUT PARTIALITY: TYPE CHECKING

# Adding Integers to Lambda Calculus

exp ::=
  | …
  | n   *constant integers*
  | $exp_1$ + $exp_2$   *binary arithmetic operation*

val ::=
  | fun x -> exp   *functions are values*
  | n   *integers are values*

n{v/x} = n   *constants have no free vars.*
($e_1$ + $e_2$){v/x} = ($e_1${v/x} + $e_2${v/x})   *substitute everywhere*

$$\frac{exp_1 \Downarrow n_1 \quad exp2 \Downarrow n_2}{exp_1 + exp_2 \ \Downarrow (n1 \ [\![+]\!] \ n2)}$$

Object-level '+'   Meta-level '+'

**NOTE:** there are no rules for the case where exp1 or exp2 evaluate to functions! The semantics is *undefined* in those cases.

# Type Checking / Static Analysis

- Recall the interpreter from the Eval3 module:

```
let rec eval env e =
 match e with
  | …
  | Add (e1, e2) ->
      (match (eval env e1, eval env e2) with
        | (IntV i1, IntV i2) -> IntV (i1 + i2)
        | _ -> failwith "tried to add non-integers")
  | …
```

- The interpreter might fail at runtime.
  - Not all operations are defined for all values (*e.g.*, 3/0, 3 + true, …)
- A compiler can't generate sensible code for this case.
  - A naïve implementation might "add" an integer and a function pointer

# Type Judgments

- In the judgment: $E \vdash e : t$
  - E is a ***typing environment*** or a ***type context***
  - E maps variables to types. It is just a set of bindings of the form: $x_1 : t_1, x_2 : t_2, \ldots, x_n : t_n$

- For example: $x : \text{int}, b : \text{bool} \vdash \text{if } (b)\ 3 \text{ else } x : \text{int}$

- What do we need to know to decide whether "if (b) 3 else x" has type int in the environment x : int, b : bool?
  - b must be a bool i.e. $x : \text{int}, b : \text{bool} \vdash b : \text{bool}$
  - 3 must be an int i.e. $x : \text{int}, b : \text{bool} \vdash 3 : \text{int}$
  - x must be an int i.e. $x : \text{int}, b : \text{bool} \vdash x : \text{int}$

# Simply-typed Lambda Calculus

- For the language in "tc.ml" we have five inference rules:

INT

$$\frac{}{E \vdash i : int}$$

VAR

$$\frac{x : T \in E}{E \vdash x : T}$$

ADD

$$\frac{E \vdash e_1 : int \qquad E \vdash e_2 : int}{E \vdash e_1 + e_2 : int}$$

FUN

$$\frac{E, x : T \vdash e : S}{E \vdash \text{fun } (x{:}T) \text{ -> } e \ : T \text{ -> } S}$$

APP

$$\frac{E \vdash e_1 : T \text{ -> } S \qquad E \vdash e_2 : T}{E \vdash e_1 \ e_2 : S}$$

- Note how these rules correspond to the code.

# Type 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 typechecker: verify that such a tree exists.
- Example: Find a tree for the following program using the inference rules on the previous slide:

$$\vdash \texttt{(fun (x:int) -> x + 3) 5} : \texttt{int}$$

# Example Derivation Tree

$$
\boxed{\text{APP}}\;\dfrac{
\boxed{\text{FUN}}\;\dfrac{
\boxed{\text{ADD}}\;\dfrac{
\boxed{\text{VAR}}\;\dfrac{x : \text{int} \;\in\; x : \text{int}}{x : \text{int} \vdash x \;:\; \text{int}}
\qquad
\boxed{\text{INT}}\;\dfrac{}{x : \text{int} \vdash 3 \;:\; \text{int}}
}{x : \text{int} \vdash x + 3 : \text{int}}
}{\vdash (\text{fun } (x{:}\text{int}) \to x + 3) \;:\; \text{int} \longrightarrow \text{int}}
\qquad
\boxed{\text{INT}}\;\dfrac{}{\vdash 5 : \text{int}}
}{\vdash (\text{fun } (x{:}\text{int}) \to x + 3)\ 5 \;:\; \text{int}}
$$

- Note: the OCaml function **typecheck** verifies the existence of this tree. The structure of the recursive calls when running **typecheck** is the same shape as this tree!
- Note that $x : \text{int} \in E$ is implemented by the function **lookup**

# Ill-typed Programs

- Programs without derivations are ill-typed

Example: There is no type T such that

$\vdash$ (fun (x:int) → x 3) 5 : T

$x : \text{int} \to T \;\notin\; x : \text{int}$

VAR

$x : \text{int} \vdash x : \text{int} \to T$ $\quad\quad$ $x : \text{int} \vdash 3 : \text{int}$

APP

$x : \text{int} \vdash x\ 3 : T$

FUN

$\vdash$ (fun (x:int) → x 3) : $\text{int} \to T$ $\quad\quad$ $\vdash 5 : \text{int}$

APP

$\vdash$ (fun (x:int) → x 3) 5 : T

# Type Safety

***"Well typed programs do not go wrong."***

– Robin Milner, 1978

**Theorem:** (simply typed lambda calculus with integers)

If $\vdash e : t$ then there exists a value v such that $e \Downarrow v$ .

- Note: this is a *very* strong property.
  - Well-typed programs cannot "go wrong" by trying to execute undefined code (such as 3 + (fun x -> 2))
  - Simply-typed lambda calculus is guaranteed to terminate! (i.e. it isn't Turing complete)

# Notes about this Typechecker

- The interpreter evaluates the body of a function only when it's applied.
- The typechecker always checks the body of the function
  - even if it's never applied
  - We *assume* the input has some type (say $t_1$) and reflect this in the type of the function ($t_1$ -> $t_2$).

- Dually, at a call site ($e_1$ $e_2$), we don't know what *closure* we're going to get.
  - But we can calculate $e_1$'s type, check that $e_2$ is an argument of the right type, and determine what type $e_1$ will return.

- Question: Why is this an approximation?
- Question: What if `well_typed` always returns `false`?

oat.pdf

# TYPECHECKING OAT

# 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 oat.pdf:

```
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 – x2;
return(x1);
```

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

```
var x2 = x1 + x1;
return(x2);
```

# Example Derivation

```
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 – x2;
return(x1);
```

$$\dfrac{\dfrac{\mathcal{D}_1 \quad \mathcal{D}_2 \quad \mathcal{D}_3 \quad \mathcal{D}_4}{G_0\,;\,\cdot\,;\texttt{int} \vdash \texttt{var}\ x_1 = 0;\ \texttt{var}\ x_2 = x_1 + x_1\,;\ x_1 = x_1 - x_2\,;\ \texttt{return}\ x_1\,; \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}\ [\text{STMTS}]}{\vdash \texttt{var}\ x_1 = 0;\ \texttt{var}\ x_2 = x_1 + x_1\,;\ x_1 = x_1 - x_2\,;\ \texttt{return}\ x_1\,;}\ [\text{PROG}]$$

# Example Derivation

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

$$\mathcal{D}_2 \quad = \quad \dfrac{\dfrac{\dfrac{\overline{\vdash + : (\texttt{int},\texttt{int}) \to \texttt{int}}\;[\text{ADD}] \quad \dfrac{x_1 : \texttt{int} \in \cdot, x_1 : \texttt{int}}{G_0\,;\cdot, x_1 : \texttt{int} \vdash x_1 : \texttt{int}}\;[\text{VAR}] \quad \dfrac{x_1 : \texttt{int} \in \cdot, x_1 : \texttt{int}}{G_0\,;\cdot, x_1 : \texttt{int} \vdash x_1 : \texttt{int}}\;[\text{VAR}]}{G_0\,;\cdot, x_1 : \texttt{int} \vdash x_1 + x_1 : \texttt{int}}\;[\text{BOP}]}{G_0\,;\cdot, x_1 : \texttt{int}\,;\texttt{int} \vdash \texttt{var}\ x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}\;[\text{DECL}]}{G_0\,;\cdot, x_1 : \texttt{int}\,;\texttt{int} \vdash \texttt{var}\ x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}\;[\text{SDECL}]$$

# Example Derivation

$$\mathcal{D}_3 \quad = \quad \cfrac{\cfrac{}{\vdash \texttt{-} : (\texttt{int},\texttt{int}) \to \texttt{int}}[\textsc{add}] \quad \cfrac{x_1 : \texttt{int} \in \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_1 : \texttt{int}}[\textsc{var}] \quad \cfrac{x_2 : \texttt{int} \in \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_2 : \texttt{int}}[\textsc{var}]}{\cfrac{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_1 \texttt{ - } x_2 : \texttt{int}}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} ; \texttt{int} \vdash x_1 \texttt{ = } x_1 \texttt{ - } x_2 \texttt{;} \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}[\textsc{assn}]}[\textsc{bop}]$$

$$\mathcal{D}_4 \quad = \quad \cfrac{\cfrac{x_1 : \texttt{int} \in \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_1 : \texttt{int}}[\textsc{var}]}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} ; \texttt{int} \vdash \texttt{return } x_1 \texttt{;} \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}[\textsc{Ret}]$$

# Type Safety For General Languages

**Theorem: (Type Safety)**

If $\vdash P : t$ is a well-typed program, then either:
(a) the program terminates in a well-defined way, or
(b) the program continues computing forever

- Well-defined termination could include:
  - halting with a return value
  - raising an exception
- Type safety rules out undefined behaviors:
  - abusing "unsafe" casts: converting pointers to integers, etc.
  - treating non-code values as code (and vice-versa)
  - breaking the type abstractions of the language
- What is "defined" depends on the language semantics…

# 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 ( $E \vdash e : t$ ) 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 rules are very similar in structure to the typechecking rules
- Strong mathematical foundations
  - The "Curry-Howard-Lambek correspondence":
    - Programming Language : Logic : Category theory : Order Theory
    - Programs : Proof : Morphism : Inequality
    - Type : Proposition : Object : Element
    - See EECS 490, 590, my 598 if you're interested in type systems!

# COMPILING

# 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?

$$[\![C \vdash e : t]\!] = \quad ?$$

- $[\![C]\!]$ translates contexts
- $[\![t]\!]$ is a target type
- $[\![e]\!]$ translates to a (potentially empty) stream of instructions, that, when run, computes the result into some operand

- INVARIANT: if $[\![C \vdash e : t]\!]$ = ty, operand , stream
  then the type (at the target level) of the operand is ty=$[\![t]\!]$

# Example

- $C \vdash 341 + 5 : \text{int}$ what is $[\![ C \vdash 341 + 5 : \text{int} ]\!]$ ?

$$\dfrac{\dfrac{[\![ \vdash 341 : \text{int} ]\!] = \text{(i64, Const 341, [])}}{[\![ C \vdash 341 : \text{int} ]\!] = \text{(i64, Const 341, [])}} \qquad \dfrac{[\![ \vdash 5 : \text{int} ]\!] = \text{(i64, Const 5, [])}}{[\![ C \vdash 5 : \text{int} ]\!] = \text{(i64, Const 5, [])}}}{[\![ C \vdash 341 + 5 : \text{int} ]\!] = \text{(i64, \%tmp, [\%tmp = add i64 (Const 341) (Const 5)])}}$$

# 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?

$$\frac{x : t \in L}{G ; L \vdash x : t} \quad \textbf{TYP\_VAR}$$

as expressions
(which denote values)

$$\frac{x : t \in L \quad G ; L \vdash exp : t}{G ; L ; rt \vdash x = exp ; \Rightarrow L} \quad \textbf{TYP\_ASSN}$$

as addresses
(which can be assigned)

# Interpretation of Contexts

- $[\![C]\!]$ = a map from source identifiers to types and target identifiers

- INVARIANT:
  x:t ∈ C means that

  (1) 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:

$$\left[\!\left[ \frac{x : t \in L}{G ; L \vdash x : t}\ \textbf{TYP\_VAR} \right]\!\right] = \texttt{(\%tmp, [\%tmp = load i64* \%id\_x])}$$

as expressions
(which denote values)

where (i64, %id_x) = lookup ⟦L⟧ x

- What about statements?

$$\left[\!\left[ \frac{x : t \in L \quad G ; L \vdash exp : t}{G ; L ; rt \vdash x = exp ; \Rightarrow L}\ \textbf{TYP\_ASSN} \right]\!\right] = \text{stream @}$$

[store ⟦t⟧ opn, ⟦t⟧* %id_x]

as addresses
(which can be assigned)

where (t, %id_x) = lookup ⟦L⟧ x
and ⟦G;L ⊢ exp : t⟧ = (⟦t⟧, opn, stream)

# Other Judgments?

- Statement:
  $[\![C;\ rt \vdash stmt \Rightarrow C']\!] \quad = \qquad [\![C']\!]$ , stream

- Declaration:
  $[\![G;L \vdash t\ x = exp \Rightarrow G;L,x{:}t\ ]\!] = \quad [\![G;L,x{:}t]\!]$, stream

  INVARIANT: stream is of the form:
  stream’ @
  [ %id_x = alloca ⟦t⟧;
  store ⟦t⟧ opn, ⟦t⟧* %id_x ]

  and $\quad [\![G;L \vdash exp : t\ ]\!] = ([\![t]\!]$, opn, 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.

$[\![C;rt \vdash \text{while(e) s} \Rightarrow C']\!] \quad = \quad [\![C']\!],$

```
lpre:
        opn = ⟦C ⊢ e : bool⟧
        %test = icmp eq i1 opn, 0
        br %test, label %lpost, label %lbody
lbody:
    ⟦C;rt ⊢ s ⇒ C'⟧
    br %lpre
lpost:
```

- Note: writing opn = $[\![C \vdash e : \text{bool}]\!]$ is pun
  - translating $[\![C \vdash e : \text{bool}]\!]$ generates *code* that puts the result into 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 ⊢ e : bool⟧
        %test = icmp eq i1 opn, 0
        br %test, label %else, label %then
then:
    ⟦C;rt ⊢ s1 ⇒ C'⟧
    br %merge
else:
        ⟦C; rt s2 ⇒ 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