Lecture 14

CIS 341: COMPILERS

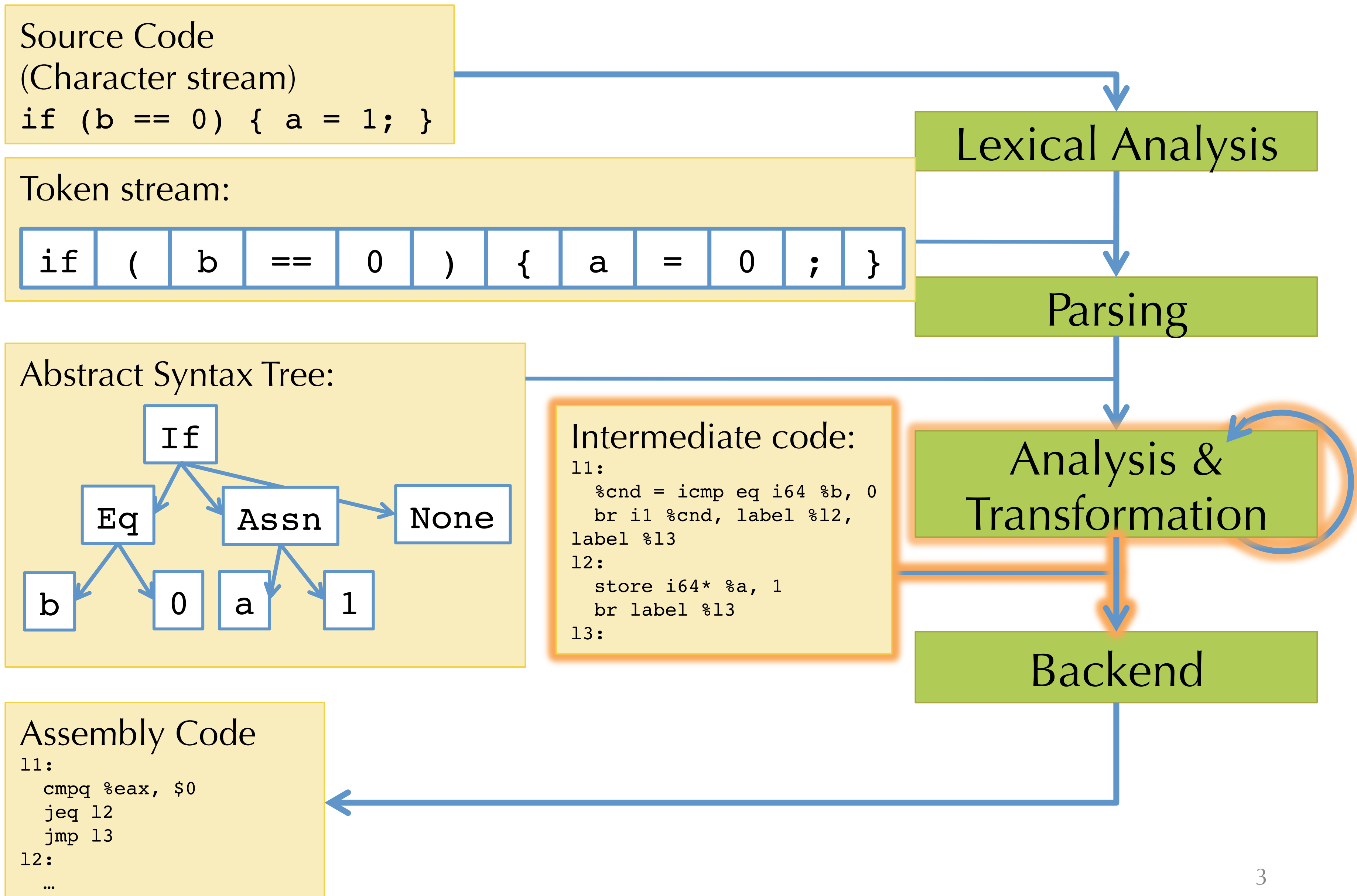
Announcements

- HW4: OAT v. 1.0
  – Parsing & basic code generation
  – **Due: March 28th**
  – **START EARLY!**

Note new Due Date!

- Midterm Exam
  – Grading in progress

Compilation in a Nutshell



See HW4

OAT V 1.0

OAT

- Simple C-like Imperative Language
  - supports 64-bit integers, arrays, strings
  - top-level, mutually recursive procedures
  - scoped local, imperative variables

- See examples in hw4 /atprograms directory

- How to design/specify such a language?
  - Grammatical constructs
  - Semantic constructs

Example Ambiguity in Real Languages

- Consider this grammar:
  S ↦ `if (E) S`
  S ↦ `if (E)` S `else` S
  S ↦ X = E
  E ↦ …

- Is this grammar OK?

- Consider how to parse:

  `if (`E_1`) if (`E_2`)` S_1
  `else` S_2

- This is known as the "dangling else" problem.
- What should the "right" answer be?

- How do we change the grammar?

How to Disambiguate if-then-else

- Want to rule out:

$$\texttt{if } (E_1) \left[\texttt{if } (E_2)\ S_1 \right] \texttt{ else } S_2$$

- Observation: An un-matched '`if`' should not appear as the '`then`' clause of a containing '`if`'.

$S \mapsto M \mid U$	// M = "matched", U = "unmatched"
$U \mapsto$ **`if`** `(`E`)` S	// Unmatched 'if'
$U \mapsto$ **`if`** `(`E`)` M **`else`** U	// Nested if is matched
$M \mapsto$ **`if`** `(`E`)` M **`else`** M	// Matched 'if'
$M \mapsto$ X **=** E	// Other statements

- See: else-resolved-parser.mly

OAT: Alternative: Use { }

- Ambiguity arises because the 'then' branch is not well bracketed:

  `if (E₁) { if (E₂) { S₁ } } else S₂` // unambiguous
  `if (E₁) { if (E₂) { S₁ } else S₂ }` // unambiguous

- So: could just require brackets
  - But requiring them for the else clause too leads to ugly code for chained if-statements:

```
if (c1) {
  …
} else {
  if (c2) {

  } else {
    if (c3) {

    } else {

    }
  }
}
```

So, compromise? Allow unbracketed else block only if the body is 'if':

```
if (c1) {

} else if (c2) {

} else if (c3) {

} else {

}
```

Benefits:
- Less ambiguous
- Easy to parse
- Enforces good style

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)

```
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 $\texttt{G;L} \vdash e : t$ as
  "the expression e is well typed and has type t"
- For any environment G, expression e, and statements s_1, s_2.

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

  holds if $\texttt{G;L} \vdash e : \text{bool}$ and $\texttt{G;L;rt} \vdash s_1$ and $\texttt{G;L;rt} \vdash s_2$
  all hold.
- More succinctly: we summarize these constraints as an *inference rule*:

Premises

$$\frac{\texttt{G;L} \vdash e : \text{bool} \qquad \texttt{G;L;rt} \vdash s_1 \qquad \texttt{G;L;rt} \vdash s_2}{\texttt{G;L;rt} \vdash \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2}$$

Conclusion

- This rule can be used for *any* substitution of the syntactic metavariables $\texttt{G}$, 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:

```
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

$$\mathcal{D}_1 \quad = \quad \dfrac{\dfrac{\dfrac{\dfrac{}{G_0;\cdot \vdash 0 : \mathtt{int}}\;[\textsc{Int}]}{G_0;\cdot \vdash 0 : \mathtt{int}}\;[\textsc{Const}]}{G_0;\cdot \vdash \mathbf{var}\ x_1 = 0 \Rightarrow \cdot, x_1 : \mathtt{int}}\;[\textsc{Decl}]}{G_0;\cdot\,;\mathtt{int} \vdash \mathbf{var}\ x_1 = 0; \Rightarrow \cdot, x_1 : \mathtt{int}}\;[\textsc{SDecl}]$$

$$\mathcal{D}_2 \quad = \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash + : (\mathtt{int},\mathtt{int}) \to \mathtt{int}}\;[\textsc{Add}] \qquad \dfrac{x_1 : \mathtt{int} \in \cdot, x_1 : \mathtt{int}}{G_0;\cdot, x_1 : \mathtt{int} \vdash x_1 : \mathtt{int}}\;[\textsc{Var}] \qquad \dfrac{x_1 : \mathtt{int} \in \cdot, x_1 : \mathtt{int}}{G_0;\cdot, x_1 : \mathtt{int} \vdash x_1 : \mathtt{int}}\;[\textsc{Var}]}{G_0;\cdot, x_1 : \mathtt{int} \vdash x_1 + x_1 : \mathtt{int}}\;[\textsc{Bop}]}{G_0;\cdot, x_1 : \mathtt{int};\mathtt{int} \vdash \mathbf{var}\ x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \mathtt{int}, x_2 : \mathtt{int}}\;[\textsc{Decl}]}{G_0;\cdot, x_1 : \mathtt{int};\mathtt{int} \vdash \mathbf{var}\ x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \mathtt{int}, x_2 : \mathtt{int}}\;[\textsc{SDecl}]$$

Example Derivation

$$\mathcal{D}_3 \quad \dfrac{\dfrac{x_1\!:\!\texttt{int} \in \cdot, x_1\!:\!\texttt{int}, x_2\!:\!\texttt{int}\ ;}{\dfrac{}{\vdash \texttt{-} : (\texttt{int},\texttt{int}) \to \texttt{int}}\,[\text{ADD}] \quad \dfrac{x_1\!:\!\texttt{int} \in \cdot, x_1\!:\!\texttt{int}, x_2\!:\!\texttt{int}}{G_0;\cdot,x_1\!:\!\texttt{int},x_2\!:\!\texttt{int} \vdash x_1 : \texttt{int}}\,[\text{VAR}] \quad \dfrac{x_2\!:\!\texttt{int} \in \cdot, x_1\!:\!\texttt{int}, x_2\!:\!\texttt{int}}{G_0;\cdot,x_1\!:\!\texttt{int},x_2\!:\!\texttt{int} \vdash x_2 : \texttt{int}}\,[\text{VAR}]}}{\dfrac{G_0;\cdot,x_1\!:\!\texttt{int},x_2\!:\!\texttt{int} \vdash x_1 \texttt{ - } x_2 : \texttt{int}}{G_0;\cdot,x_1\!:\!\texttt{int},x_2\!:\!\texttt{int};\texttt{int} \vdash x_1 \texttt{ = } x_1 \texttt{ - } x_2\texttt{;} \Rightarrow \cdot,x_1\!:\!\texttt{int},x_2\!:\!\texttt{int}}\,[\text{ASSN}]}\,[\text{BOP}]$$

$$\mathcal{D}_4 \quad = \quad \dfrac{\dfrac{x_1\!:\!\texttt{int} \in \cdot, x_1\!:\!\texttt{int}, x_2\!:\!\texttt{int}}{G_0;\cdot,x_1\!:\!\texttt{int},x_2\!:\!\texttt{int} \vdash x_1 : \texttt{int}}\,[\text{VAR}]}{G_0;\cdot,x_1\!:\!\texttt{int},x_2\!:\!\texttt{int};\texttt{int} \vdash \texttt{return}\ x_1\texttt{;} \Rightarrow \cdot,x_1\!:\!\texttt{int},x_2\!:\!\texttt{int}}\,[\text{RET}]$$

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?

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

- $[\![t]\!]$ is a target type
- $[\![e]\!]$ translates to a (potentially empty) sequence 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 $[\![C \vdash 341 + 5 : \text{int}]\!]$?

$[\![\vdash 341 : \text{int}]\!] =$ `(i64, Const 341, [])`

$[\![C \vdash 341 : \text{int}]\!] =$ `(i64, Const 341, [])`

$[\![\vdash 5 : \text{int}]\!] =$ `(i64, Const 5, [])`

$[\![C \vdash 5 : \text{int}]\!] =$ `(i64, Const 5, [])`

$[\![C \vdash 341 + 5 : \text{int}]\!] =$ `(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:

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

as expressions (which denote values)

where (`i64`, `%id_x`) = lookup ⟦L⟧ x

- What about statements?

$$\left[\!\!\left[\frac{x{:}t \in L \quad G;L \vdash exp : t}{G;L;rt \vdash x = exp;\ \Rightarrow L}\ \text{TYP\_ASSN} \right]\!\!\right] = $$ 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 ⊢ stmt ⇒ C′⟧ = ⟦C′⟧ , stream

- Declaration:
  ⟦G;L ⊢ t x = exp ⇒ G;L,x:t ⟧ = ⟦G;L,x:t⟧, stream

  INVARIANT: stream is of the form:

```
        stream' @
        [ %id_x = alloca ⟦t⟧;
          store ⟦t⟧ opn, ⟦t⟧* %id_x ]
```

  and ⟦G;L ⊢ exp : t ⟧ = (⟦t⟧, `opn`, stream′)

- Rest follow similarly

COMPILING CONTROL