Lecture 14

COMPILER DESIGN
Announcements

• **HW 3:** due soon

• **HW 4:** Building a frontend
  – Work with lexer and parser generators
  – Compile a C-like source language to LLVM
Debugging parser conflicts.
Disambiguating grammars.

MENHIR IN PRACTICE
(QUICK REVISIT OF EXAMPLES)
Operational Semantics

- Specified with 2 inference rules with judgments of the form \( \text{exp} \downarrow v \)
  - Read this notation as “program \( \text{exp} \) evaluates to value \( v \)”
  - We give a *call-by-value* semantics
    Function arguments are evaluated before substitution

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

\[
\begin{align*}
  \text{exp}_1 & \downarrow (\text{fun } x \rightarrow \text{exp}_3) & \text{exp}_2 & \downarrow v & \text{exp}_3\{v/x\} & \downarrow w \\
\end{align*}
\]

\[
\begin{align*}
  \text{exp}_1 \ \text{exp}_2 & \downarrow w \\
  \text{“To evaluate function application: Evaluate the function to a value, evaluate the argument to a value, and then substitute the argument for the function.”}
\end{align*}
\]
Adding Integers to Lambda Calculus

\[
\text{exp} ::= \\
\quad \text{...} \\
\quad n \\
\quad \text{exp}_1 + \text{exp}_2 \\
\]

\[
\text{val} ::= \\
\quad \text{fun } x \rightarrow \text{exp} \\
\quad n \\
\]

\[
n\{v/x\} = n \\
(e_1 + e_2\{v/x\} = (e_1\{v/x\} + e_2\{v/x\})
\]

\[
\text{exp}_1 \Downarrow n_1 \quad \text{exp}_2 \Downarrow n_2 \\
\]

\[
\text{exp}_1 + \text{exp}_2 \Downarrow (n_1 \llbracket + \rrbracket n_2) \\
\]

object-level ‘+’

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

STATIC ANALYSIS
Variable Scoping

• Problem: How to determine whether a declared variable is in scope?

• Issues
  – Q1: Which variables are available at a given program location?
  – Q2: Can the same identifier be reused (i.e., shadowing), or is it an error?

• Example: code below is syntactically correct, but not well-formed
  – y and q are used without being defined anywhere

```c
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 to describe this?
  – Compiler keeps a mapping from variables to information about them
  – Using a “symbol table”
Why Inference Rules?

- Allow a compact, precise way of specifying language properties
  - About 20 pages for full Java, versus
  - Hundreds of pages of prose Java Language Specification

- Correspond closely to recursive AST traversal for implementing them

- Type checking/inference tries to prove a different judgment $G; L \vdash e : t$
  - By searching backward through the rules

- Compiling is also a set of inference rules specifying $G \vdash \text{source} \Rightarrow \text{target}$
  - Compilation judgments are similar to the type checking judgments

- Strong mathematical foundations, e.g., “Curry-Howard correspondence”
  - Programming Language ~ Logic
  - Program ~ Proof
  - Type ~ Proposition
Inference Rules

• A judgment $G;L \vdash e : t$ is read "the expression $e$ is well typed and has type $t$"

• For any environment $G; L$, expression $e$, and statements $s_1, s_2$

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

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

• More succinctly, summarize these constraints as an inference rule

  \[ \text{Premises} \]

  $G;L \vdash e : \text{bool}$  $G;L;rt \vdash s_1$  $G;L;rt \vdash s_2$

  \[ \text{Conclusion} \]

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

• It can be used for any substitution of the metavariables $G$, $L$, $rt$, $e$, $s_1$, $s_2$
Checking Derivations

- **Derivation** or **proof tree**
  - Nodes: judgments
  - Edges: connect premises to a conclusion (according to inference rules)
- Leaves of the tree are **axioms** (i.e., rules with no premises)
  - Example: the INT rule we’ll see is an axiom
- **Goal of the type checker**: verify that such a tree exists

- Ex1: Check the code below using inference rules in the Oat spec. (HW4)

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

- Ex2: There is no tree for this ill-scoped program

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

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

\[
\begin{align*}
G_0; \cdot; \text{int} & \vdash \text{var } x_1 = 0; \text{var } x_2 = x_1 + x_1; x_1 = x_1 - x_2; \text{return } x_1; \Rightarrow \cdot, x_1: \text{int}, x_2: \text{int} \\
\vdash \text{var } x_1 = 0; \text{var } x_2 = x_1 + x_1; x_1 = x_1 - x_2; \text{return } x_1;
\end{align*}
\]

[STMTS]

[PROG]
Example Derivation

\[
D_1 = \frac{G_0; \cdot \vdash 0 : \text{int}}{\text{[INT]}} \quad \frac{G_0; \cdot \vdash 0 : \text{int}}{\text{[CONST]}} \quad \frac{G_0; \cdot \vdash \text{var } x_1 = 0 \Rightarrow \cdot, x_1 : \text{int}}{\text{[DECL]}} \\
\]

\[
D_2 = \frac{\vdash + : (\text{int, int}) \rightarrow \text{int}}{\text{[ADD]}} \quad \frac{x_1 : \text{int} \in \cdot, x_1 : \text{int}}{\text{[VAR]}} \quad \frac{x_1 : \text{int} \in \cdot, x_1 : \text{int}}{\text{[VAR]}} \quad \frac{\vdash \cdot, x_1 : \text{int} \vdash x_1 + x_1 : \text{int}}{\text{[BOP]}} \\
\]

\[
G_0; \cdot, x_1 : \text{int} \vdash x_1 + x_1 : \text{int} \\
G_0; \cdot, x_1 : \text{int} \vdash \text{var } x_2 = x_1 + x_1 \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int} \\
G_0; \cdot, x_1 : \text{int} \vdash \text{var } x_2 = x_1 + x_1 \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int} \\
\]

Zhendong Su  Compiler Design
Example Derivation

\[ D_3 \]
\[
\begin{align*}
\frac{x_1 : \text{int} \in \cdot, x_1 : \text{int}, x_2 : \text{int}}{\text{[ADD]}} & \quad \frac{x_1 : \text{int} \in \cdot, x_1 : \text{int}, x_2 : \text{int}}{\text{[VAR]}} & \quad \frac{x_2 : \text{int} \in \cdot, x_1 : \text{int}, x_2 : \text{int}}{\text{[VAR]}} \\
\frac{\vdash - : (\text{int, int}) \to \text{int}}{\text{[ADD]}} & \quad \frac{\vdash x_1 : \text{int}, x_2 : \text{int} \vdash x_1 : \text{int}}{\text{[VAR]}} & \quad \frac{\vdash x_1 : \text{int}, x_2 : \text{int} \vdash x_2 : \text{int}}{\text{[BOP]}} \\
G_0; \cdot, x_1 : \text{int}, x_2 : \text{int} \vdash x_1 - x_2 : \text{int} & \quad G_0; \cdot, x_1 : \text{int}, x_2 : \text{int} \vdash x_1 = x_1 - x_2; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int} & \quad \text{[ASSN]} \\
G_0; \cdot, x_1 : \text{int}, x_2 : \text{int}; \text{int} \vdash x_1 = x_1 - x_2; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int} & \quad \text{[ASSN]} \\
G_0; \cdot, x_1 : \text{int}, x_2 : \text{int}; \text{int} \vdash x_1 = x_1 - x_2; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int} & \quad \text{[ASSN]} \\
G_0; \cdot, x_1 : \text{int}, x_2 : \text{int}; \text{int} \vdash \text{return} x_1; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int} & \quad \text{[RET]}
\end{align*}
\]

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

• Allow a compact, precise way of specifying language properties
  – About 20 pages for full Java, versus
  – Hundreds of pages of prose Java Language Specification

• Correspond closely to recursive AST traversal for implementing them

• Type checking/inference tries to prove a different judgment $G; L \vdash e : t$
  – By searching backward through the rules

• Compiling is also a set of inference rules specifying $G \vdash \text{source} \Rightarrow \text{target}$
  – Compilation judgments are similar to the type checking judgments

• Strong mathematical foundations, e.g., “Curry-Howard correspondence”
  – Programming Language ~ Logic
  – Program ~ Proof
  – Type ~ Proposition
Why Inference Rules?

• Allow a compact, precise way of specifying language properties
  – About 20 pages for full Java, versus
  – Hundreds of pages of prose Java Language Specification

• Correspond closely to recursive AST traversal for implementing them

• Type checking/inference tries to prove a different judgment $C ⊢ e : t$
  – By searching backward through the rules

• Compiling is “interpreting” the type checking rules $[C ⊢ e : t]$
  – Compilation follows the type checking judgment

• Strong mathematical foundations, e.g., “Curry-Howard correspondence”
  – Programming Language ~ Logic
  – Program ~ Proof
  – Type ~ Proposition

Zhendong Su  Compiler Design
Compilation as Translating Judgments

• Consider the 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 (possibly empty) sequence of instructions
  – The instruction sequence computes \( e \)'s result into some operand

• Invariant

  If \( \llbracket C \vdash e : t \rrbracket = ty, \text{operand}, \text{stream} \)
  then the type (at the target level) of the operand is \( ty = \llbracket t \rrbracket \)
Example

\[ C \vdash 341 + 5 : \text{int} \]

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

\[
\llbracket C \vdash 341 : \text{int} \rrbracket = \langle i64, \text{Const} 341, [] \rangle
\]
\[
\llbracket 5 : \text{int} \rrbracket = \langle i64, \text{Const} 5, [] \rangle
\]

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

• What is \([C]\)?

• Source level \(C\) has bindings like: \(x: \text{int}, y: \text{bool}\)
  – 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?

\[
\frac{x : t \in L}{G ; L \vdash x : t} \quad \text{TYP\_VAR}
\]

\text{as expressions}
\text{(which denote values)}

\[
\frac{x : t \in L}{G ; L \vdash \text{exp} : t} \quad \text{TYP\_ASSN}
\]

\[
\frac{G ; L ; rt \vdash x = \text{exp} \Rightarrow L}{G ; L \vdash x : t}
\]

\text{as addresses}
\text{(which can be assigned)}

Zhendong Su  Compiler Design
Interpretation of Contexts

• $[C]$ = a map from source identifiers to types and target identifiers

• Invariant
  $x \cdot t \in C$ means that
  
  (1) $\text{lookup} \ C \ x = (t, \%\text{id}_x)$
  (2) the (target) type of $\%\text{id}_x$ is $\llbracket t \rrbracket^*$ (a pointer to $\llbracket t \rrbracket$)
Interpretation of Variables

- Establish invariant for expressions

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

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

- What about statements?

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

\[
\text{where } (t, \%id\_x) = \text{lookup } [L] \ x \\
\text{and } [G; L \vdash \text{exp} : t] = ([t], \text{opn}, \text{stream})
\]

Zhendong Su  Compiler Design
Other Judgments?

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

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

Invariant: stream is of the form
\[
\text{stream}' @ \llbracket \%\text{id}_x = \text{alloca } [\text{t}] ;
\text{store } [\text{t}] \text{ opn}, [\text{t}]^* \%\text{id}_x \rrbracket
\]

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

• Rest follow similarly
COMPILING CONTROL
Translating while

- Consider translating “while(e) s”
  - Test conditional e, if true jump to body s, else jump to label after body s

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

```plaintext
<table>
<thead>
<tr>
<th>lpre:</th>
<th>lbody:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>opn = [C \vdash e : bool]</code></td>
<td><code>[[C;rt \vdash s \Rightarrow C']]</code></td>
</tr>
<tr>
<td><code>%test = icmp eq i1 opn, 0</code></td>
<td><code>br %lpre</code></td>
</tr>
<tr>
<td><code>br %test, label %lpost, label %lbody</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>br %lpre</code></td>
</tr>
</tbody>
</table>

- Note: writing `opn = [C \vdash e : bool]` is pun
  - translating `[C \vdash e : 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

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

```
opn = [C ⊢ e : bool]
%test = icmp eq il opn, 0
br %test, label %else, label %then
then:
    [C;rt ⊢ s_1 ⇒ C']
    br %merge
else:
    [C;rt ⊢ s_2 ⇒ 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
• Consider compiling the following program fragment

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

```asm
%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 \( \lbrack e \rbrack \) to produce a value
  – \( \lbrack C \vdash e : t \rbrack = (\text{ty}, \text{operand}, \text{stream}) \)
  – e.g. \( \lbrack C \vdash e_1 + e_2 : \text{int} \rbrack = (\text{i64}, \%\text{tmp}, [\%\text{tmp} = \text{add} \ \lbrack e_1 \rbrack \ [e_2 \rbrack]) \)

• 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: $[C \vdash e : t] = (ty, operand, stream)$

• Conditional branch translation of Booleans, without materializing value
  $[C \vdash e : bool@] ltrue lfalse = stream$

[\[C, rt \vdash \text{if (e) then s1 else s2 } \Rightarrow C'] = [C'],
\]
\[
\begin{align*}
\text{insns}_3 \\
\text{then:} & \\
[s1] & \br \%\text{merge} \\
\text{else:} & \\
[s2] & \br \%\text{merge} \\
\text{merge:} &
\end{align*}
\]

Notes

• Two extra arguments
  – “true” branch label
  – “false” branch label

• Doesn’t “return a value”

• Aside: this is a form of continuation-passing translation

where
\[
[\[C, rt \vdash s_1 \Rightarrow C'] = [C'], \text{insns}_1 \\
[\[C, rt \vdash s_2 \Rightarrow C'] = [C'], \text{insns}_2 \\
[\[C \vdash e : bool@ ] \text{then else } = \text{insns}_3
\]
Short Circuit Compilation: Expressions

- $\dfrac{\left[ C \vdash e : \text{bool}@ \right]}{\text{ltrue lfalse} = \text{insns}}$

\[\left[ C \vdash \text{false} : \text{bool}@ \right] \text{ltrue lfalse} = \text{[br \ %lfalse]}\]

\[\left[ C \vdash \text{true} : \text{bool}@ \right] \text{ltrue lfalse} = \text{[br \ %ltrue]}\]

- $\dfrac{\left[ C \vdash e : \text{bool}@ \right]}{\text{lfalse ltrue} = \text{insns}}$

- $\dfrac{\left[ C \vdash !e : \text{bool}@ \right]}{\text{ltrue lfalse} = \text{insns}}$

FALSE

TRUE

NOT
Short Circuit Evaluation

Idea: build the logic into the translation

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

where \text{right} is a fresh label
Consider compiling the following program fragment:

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

```assembly
if (x & !y | !w)
    %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
```