CIS 341: COMPILERS
Announcements

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

• Midterm Exam
  – Grading in progress

Note new Due Date!
Compilation in a Nutshell

Source Code
(Character stream)
if (b == 0) { a = 1; }

Token stream:
if ( b == 0 ) { a = 0 ; }

Abstract Syntax Tree:

If
   Eq     Assn     None
      b    0      a     1

Intermediate code:
11:
   %cnd = icmp eq i64 %b, 0
   br i1 %cnd, label %l2, label %l3
12:
   store i64* %a, 1
   br label %l3
13:

Assembly Code
11:
   cmpq %eax, $0
   jeq 12
   jmp 13
12:
   ...

Lexical Analysis

Parsing

Analysis & Transformation

Backend
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 (E₁) if (E₂) S₁ else S₂

• 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:

\[
\text{if } (E_1) \begin{cases} \text{if } (E_2) S_1 \end{cases} \text{ else } S_2
\]

• Observation: An un-matched ‘if’ should not appear as the ‘then’ clause of a containing ‘if’.

\[
\begin{align*}
S & \rightarrow M \mid U & \quad & \text{M = “matched”, U = “unmatched”} \\
U & \rightarrow \text{if } (E) S & & \text{Unmatched ‘if’} \\
U & \rightarrow \text{if } (E) M \text{ else } U & & \text{Nested if is matched} \\
M & \rightarrow \text{if } (E) M \text{ else } M & & \text{Matched ‘if’} \\
M & \rightarrow X = E & & \text{Other statements}
\end{align*}
\]

• 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
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)

```plaintext
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?
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 ⊢ 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 ⊢ src ⇒ 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!
• 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>$G;L;rt \vdash s_2$</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.
- Example 1: Find a tree for the following program using the inference rules in oat0-defn.pdf:

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

Example 2: There is no tree for this ill-scoped program:

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

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

\[
\begin{array}{cccc}
D_1 & D_2 & D_3 & D_4 \\
G_0; \cdot ; \text{int} \vdash \text{var } x_1 = 0; \text{var } x_2 = x_1 + x_1; x_1 = x_1 - x_2; \text{return } x_1; \Rightarrow \cdot, x_1: \text{int}, x_2: \text{int} \\
\vdash \text{var } x_1 = 0; \text{var } x_2 = x_1 + x_1; x_1 = x_1 - x_2; \text{return } x_1;
\end{array}
\]
Example Derivation

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

\[
D_2 = \frac{\vdash + : (\text{int, int}) \rightarrow \text{int}}{\text{[ADD]}} \quad \frac{x_1 : \text{int} \in \cdot, x_1 : \text{int}}{\text{[VAR]}} \quad \frac{x_1 : \text{int} \in \cdot, x_1 : \text{int}}{\text{[VAR]}} \quad \frac{G_0; \cdot, x_1 : \text{int} \vdash x_1 + x_1 : \text{int}}{\text{[BOP]}} \quad \frac{G_0; \cdot, x_1 : \text{int} \vdash \text{var } x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int}}{\text{[DECL]}} \quad \frac{G_0; \cdot, x_1 : \text{int} \vdash \text{var } x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int}}{\text{[SDECL]}}
\]
Example Derivation

\[ \mathcal{D}_3 \]

\[
\begin{align*}
\text{x}_1 : \text{int} & \in \cdot, \text{x}_1 : \text{int}, \text{x}_2 : \text{int} ; \\
\text{G}_0 ; \cdot, \text{x}_1 : \text{int}, \text{x}_2 : \text{int} \vdash x_1 : \text{int} & \quad \text{[VAR]} \\
\text{G}_0 ; \cdot, \text{x}_1 : \text{int}, \text{x}_2 : \text{int} \vdash x_2 : \text{int} & \quad \text{[VAR]} \\
\text{G}_0 ; \cdot, \text{x}_1 : \text{int}, \text{x}_2 : \text{int} \vdash x_1 - x_2 : \text{int} & \quad \text{[BOP]} \\
\text{G}_0 ; \cdot, \text{x}_1 : \text{int}, \text{x}_2 : \text{int} ; \text{int} \vdash x_1 = x_1 - x_2 ; \Rightarrow \cdot, \text{x}_1 : \text{int}, \text{x}_2 : \text{int} & \quad \text{[ASSN]} \\
\end{align*}
\]

\[ \mathcal{D}_4 = \]

\[
\begin{align*}
\text{G}_0 ; \cdot, \text{x}_1 : \text{int}, \text{x}_2 : \text{int} \vdash x_1 : \text{int} & \quad \text{[VAR]} \\
\text{G}_0 ; \cdot, \text{x}_1 : \text{int}, \text{x}_2 : \text{int} ; \text{int} \vdash \text{return } x_1 ; \Rightarrow \cdot, \text{x}_1 : \text{int}, \text{x}_2 : \text{int} & \quad \text{[RET]} \\
\end{align*}
\]
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*: $\llbracket C \vdash e : t \rrbracket$
  – 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!

*CIS 341: Compilers

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

- \( C \vdash 341 + 5 : \text{int} \)

What is \([ C \vdash 341 + 5 : \text{int}]\)?

---

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

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

• What is \([C]\)?

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

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

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

• $\llbracket C \rrbracket$ = a map from source identifiers to types and target identifiers

• INVARIANT:
  $x : t \in C$ means that
  
  (1) $\text{lookup } \llbracket C \rrbracket x = (t, \% \text{id}_x)$
  
  (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:
  \[
  \begin{align*}
  & x : t \in L \\
  \implies \quad G; L \vdash x : t \\
  \text{TYP\_VAR}
  \end{align*}
  \]
  as expressions
  (which denote values)

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

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

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

- Statement:
  $[\langle C; \text{stmt} \Rightarrow C' \rangle] = [\langle C' \rangle], \text{stream}$

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

  INVARIANT: stream is of the form:
  
  stream’ @
  
  $[ \%id_x = \text{alloca } [\langle \text{t} \rangle];$
  
  store $[\langle \text{t} \rangle \text{ opn}, [\langle \text{t} \rangle]^* \%id_x ]$

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

- Rest follow similarly
COMPILING CONTROL