Where are we at?

• Continuing the whirl-wind trip through a compiler
  – We ended up with parsing the Expression program last time
  – On to static semantics …

• Then we’ll try to connect 311 with 411 by looking at an interpreter in Dr. Racket and Java
Syntax rules alone are not enough to specify the format of well-formed programs.

**Example 1:**
m = 2;
print m + x

**Undefined!**

**Example 2:**
m = 2;
n = m < 4;
print n+1

**Type error!**
Scope Rules

Scope rules regulate visibility of identifiers. They relate every applied occurrence of an identifier to a binding occurrence.

Example 1
m = 2;
r = 10 * m;
print r

Example 2:
m = 2;
print m + x

Terminology:
Static binding vs. dynamic binding
Checking scope rules

• Do:
  – run type sample/badscope.exp
Checking Expression programs

• Look at the implementation:
  – In package typechecker.implementation
    • BuildSymbolTableVisitor
Type Rules

Type rules regulate the expected types of arguments and types of returned values for the operations of a language.

Type rule of $E_1 < E_2$:

$E_1 < E_2$ is type correct and of type Boolean if $E_1$ and $E_2$ are type correct and of type Integer.

Type rule of $E ? V_1 : V_2$:

$E ? V_1 : V_2$ is type correct and of the same type as $V_1$ if $E$ is of type Boolean and $V_1$ and $V_2$ are type correct and have the same type.

Terminology:

Static typing vs. dynamic typing
Checking type rules

• Do:
  – run type sample/badtype.exp
Checking Expression programs

• Look at the implementation:
  – In package typechecker.implementation
  • TypeCheckVisitor
Semantics

Specification of semantics is concerned with specifying the “meaning” of well-formed programs.

Terminology:

Expressions are evaluated and yield values (and may or may not perform side effects)

Statements are executed and perform side effects.

Declarations are elaborated to produce bindings

Side effects:
• change the values of variables
• perform input/output
**Semantics**

**Example:** The (informally specified) semantics of statements in Expression.

*Statements are executed to define constants and perform output.*

The statement $C = E;$ is executed as follows:

- First the expression $E$ is evaluated to yield a value $v$
- Then $v$ is assigned to the constant named $C$

The sequential statement $S_1$ $S_2$ is executed as follows:

- First the statement $S_1$ is executed
- Then the statement $S_2$ is executed

*etc.*
Semantics

Example: The semantics of expressions.

An expression is evaluated to yield a value.

A (literal expression) \( L \) yields the integer value of \( L \)

The (constant name) expression \( C \) yields the value of the constant named \( C \)

The (binary operation) expression \( E_1 \circ E_2 \) yields the value obtained by applying the binary operation \( \circ \) to the values yielded by (the evaluation of) expressions \( E_1 \) and \( E_2 \)

etc.
Semantics

• The ultimate semantics processing in a compiler is translation
• Generate code in the target language that is equivalent in every respect to the program in the source language
• Translation to intermediate code
• Instruction selection
• Register allocation
• Code improvement (optimization)
Expression translation to IR

• Do:
  – run ir sample/cond.exp
  – run ir sample/exp.exp

• Look at the implementation:
  – in package translate.implementation
    • TranslateVisitor

• IR has four forms:
  – raw
  – linear (canonical)
  – basic blocks
  – trace scheduled
  – run vir sample/exp.exp
Optimization (Or more precisely: anti-pessimization)

- A bunch of analysis needs to be done first
  - flow graph
  - liveness

- Then we can perform some correctness-preserving transformations that will improve the code

- I’m not going to show you the code 😞

- Do:
  - run iropt sample/exp.exp
Instruction selection

- Maximal munch (greedy tree matching)
- Express patterns corresponding to machine instructions
- Match the patterns against the (linear) IR
- Works from the leaves up
- Do:
  - run sim sample/cond.exp
  - run sim sample/exp.exp
- Look at the code: (yuck)
  - in backend package codegen.x86_64
    - X64_64Muncher
    - start with CONST and PLUS rules
Register allocation

• A bunch of analysis needs to be done first
  – flow graph
  – liveness
  – interference
• Then we can select registers for every temp
• I’m not going to show you the code 😞
• Do:
  – run flow sample/exp.exp & show the graph (show flow*.dot)
  – run live sample/exp.exp & show the graph (show live*.dot)
  – run interf sample/exp.exp & show the graph (first)
  – run reg sample/exp.exp & show the graph (final)
And then we’re all done

• Well, almost
  – We need to assemble the resulting assembly code
    • use gcc to compile exp.s
  – We need a bit of runtime support
    • getting the program running (main?)
    • implementing println
    • for a bigger language, other “builtin” functionality
      – allocating/freeing objects
      – supporting builtin types
And then we’re all done ...

• Do:
  – run final sample/exp.exp
  – gcc –c sample/exp.s
  – gcc –o exp.exe exp.o runtime/runtime.o
  – ./exp.exe

• Ta Da!!
Where are we going?

• The next 4 weeks or so are:
  – Parsing (Chapters 2, 3, and 4)
  – Scopes (Chapter 5)
  – Type checking (Also Chapter 5)
  – Translation to IR (Chapter 7 with a bit of Chapter 6)

• In the context of a slightly bigger language:
  – Expressions + functions
Course Overview

• Introduction (Chapter 1)

• Compiler Frontend:
  – Lexical Analysis & Parsing (Chapters 2, 3, 4)
  – Semantic Analysis (Chapter 5)
  – Activation Records (Chapter 6)
  – Translation to Intermediate Code (Chapter 7)
  – Basic Blocks and Traces (Chapter 8)

• Compiler Backend:
  – Instruction Selection (Chapter 9)
  – Liveness Analysis (Chapter 10)
  – Register Allocation (Chapter 11)
  – Code Emission (Chapter 12)
Course Overview (continued)

• Optimization
  – Dataflow analysis and optimization (Chapter 17)

• Other stuff
  – Garbage collection (Chapter 13)
  – Object oriented languages (Chapter 14)
Connecting 311 with 411

• Simple interpreters
  – In Racket
  – In Java