Review: Program State

- The state, $\sigma$, of a program is a function from variables \textbf{directly} to values, which we can write as a set:
  \[
  \sigma = \{<i, 13>, <j, -1>\}
  \]

- We define the function $\sigma(exp)$ to be a function that returns the value of the expression, $exp$, given state $\sigma$
  \[
  \sigma(i) = 13 \\
  \sigma(i + j) = 12
  \]

Review: Program Meaning

- Let $\Sigma$ be the set of all program states, $\sigma$
- We let $M$ be a \textbf{meaning function}
  \- Inputs: one or more syntactic symbols, current program state
  \- Output: new program state (changed or not)

- For a language with set of symbols (terminal or non-terminal), $S$, the meaning function is therefore a mapping:
  \[
  M : (S^k \times \Sigma) \rightarrow \Sigma \quad (k \geq 1)
  \]

Program Meaning

- Consider a simple programming language with integer variables that can be declared and assigned:

```
Program -> Declaration* Main
  Main   -> main() { Declaration* Assignment* }
  Declaration -> int Variable;
  Assignment -> Variable = Int;
  Variable   -> Letter Letter*;
  Letter     -> a | b | ... | z
  Int        -> 0 | LeadDigit Digit*
  LeadDigit  -> 1 | 2 | ... | 9
  Digit      -> 0 | 1 | ... | 9
```

Assuming our parser ignores white-space between symbols, a legal program might be:

```
int i;
int j;
main() {
  int mine;
  i = 13;
  j = 1;
  mine = 0;
}
```
Program Meaning

The meaning function gives us rules for interpreting a program:

- **Program** $\rightarrow$ **Declaration** $\ast$ Main
  - **Main** $\rightarrow$ main() $\{$ **Declaration** $\ast$ Assignment $\ast$ $\}$
  - **Declaration** $\rightarrow$ int **Variable**;

- The meaning of a program is given by the value of that mapping, using an initially empty state:
  
  $M(\text{Program}, \emptyset) = M(\text{Declaration}^\ast \text{Main}, \emptyset)$

- This is in turn composed of the meanings of its two parts, according to the semantic rule:
  
  $M(\text{Declaration}^\ast \text{Main}, \emptyset) = M(\text{Main}, M(\text{Declaration}^\ast, \emptyset))$

A Complication: Re-Defined Variables

- We apply the same semantics to analyze the main() block's meaning:
  
  $M(\text{Program}, \emptyset) = M(\text{Declaration}^\ast \text{Main}, \emptyset)$
  
  $= M(\text{Main}, M(\text{Declaration}^\ast, \emptyset))$
  
  $= M(\text{Main}, \sigma_0)$
  
  $= M(\text{Declaration}^\ast \text{Assignment}^\ast, \sigma_0)$
  
  $= M(\text{Assignment}^\ast, M(\text{Declaration}^\ast, \sigma_0))$
  
  $= M(\text{Assignment}^\ast, \sigma_1)$

- Now, $\sigma_1$ is the result of adding all variables declared inside main() to $\sigma_0$.
Further Complications: Variable Shadowing

- **A question**: does the following code have meaning or not?

```
int i;
main() {
    int i;
    i = 3;
}
```

- **Answer**: it depends upon the scoping rules of the language
  - In order to make semantic analysis and specification easier, a language designer and compiler writer may first want to insist upon a scope analysis.
  - Code that fails to obey scope rules properly is in error, and the semantics doesn’t need to account for such cases at all.
  - We can assume that any code we analyze semantically is properly scoped.

Variable Shadowing, I

- Different languages make different decisions about scope and variable shadowing and what is legal or not.
- The following is legal in Java (and similar examples in C++ and many other ALGOL-descendant languages):

```
public class Test {
    private int x = 0;
    private void test() {
        x = 3;
        System.out.printf("Outer x == %d\n", x);
        int x = 1;
        System.out.printf("Inner x == %d\n", x);
    }
}
```

Prints:
- Outer x == 3
- Inner x == 1

Variable Shadowing, II

- The analogue to the previous example is illegal in C#:
  - In Java/C++, a local variable only has scope after it is defined.
  - In C#, it has scope over the entire block in which it is defined.

```
public class Test {
    private int x = 0;
    private void test() {
        x = 3;
        Console.WriteLine("Outer x == {}", x);
        int x = 1;
        Console.WriteLine("Inner x == {}", x);
    }
}
```

Prints:
- Error: Cannot use local variable 'x' before it is declared.

Variable Shadowing, III

- Other languages are even more permissive about shadowing and variable re-use than Java/C++.
- The following is legal in the language Rust, for example:

```
fn main() {
    let x = 1;
    {
        let x = 3;
        println!("Inner x == {}", x);
    }
    println!("Outer x == {}", x);
    let x = "It's Rust!";
    println!("Outer x == {}", x);
}
```

Prints:
- Inner x == 3
- Outer x == 1
- Outer x == It's Rust!
Shadowing in Our Sample Language

- While the syntax of our language does not specify the approach to shadowing, we will assume it is not allowed.
- Thus the program given is syntactically correct, but a (static) scoping check will disallow it before we apply any further semantic analysis.

```
Program → Declaration* Main
Main → main() { Declaration* Assignment* }
Declaration → int Variable;
Assignment → Variable = Int;
Variable → Letter Letter*
Letter → a | b | ... | z
Int → 0 | LeadDigit Digit*
LeadDigit → 1 | 2 | ... | 9
Digit → 0 | 1 | ... | 9

int i;
main() {
  i = 3;
}
```

Semantics of Assignment, I

\[
\text{Assignment} \rightarrow \text{Variable} = \text{Int};
\]

- Once variables are declared, we can assign them values.
- Again, we will assume that all assigned variables are already declared, something that can also be checked during scoping.

\[
M(\text{Assignment}, \sigma) = M(\{(M(\text{Variable}), M(\text{Int}))\}, \sigma)
\]

- We will assume that the meanings of a variable or integer is just its lexical/numerical value:

\[
M(i = 3, \sigma) = M(\{(i, 3)\}, \sigma)
\]

Semantics of Assignment, II

- For state \(\sigma\), let \(\text{var}(\sigma)\) be the set of variables it contains.
- Then for state \(\sigma_t\) of a program at time-step \(t\), a legal assignment \(A\) produces a state \(\sigma_A\) such that:

\[
\text{var}(\sigma_A) \subseteq \text{var}(\sigma_t)
\]

- For any two states, we can define the overriding union:

\[
\sigma_1 \uplus \sigma_2 = \{(\text{var, val}) | (\text{var} \in \text{var}(\sigma_1) \land \text{var} \notin \text{var}(\sigma_2)) \lor (\text{var} \in \text{var}(\sigma_2))\}
\]

- An overriding union of two states can replace the values of some of the variables:

\[
\sigma_1 = \{(x, 1), (y, 1), (z, 3)\}
\]
\[
\sigma_2 = \{(y, 9)\}
\]
\[
\sigma_1 \uplus \sigma_2 = \{(x, 1), (y, 9), (z, 3)\}
\]

- This provides a meaning function for assignment:

\[
M(i = 3, \sigma) = M(\{(i, 3)\}, \sigma)
\]
\[
= \sigma \uplus \{(i, 3)\}
\]
Multiple Assignments

- As we did (less formally) for sequences of multiple variable declarations, we can extend the meaning function to sequences of multiple assignments:

  \[
  \text{Assignment}^* = \langle A_1, A_2, \ldots, A_k \rangle
  \]

  \[
  M(\text{Assignment}^*, \sigma) = \sigma \cup M(A_1) \cup M(A_2) \cup \cdots \cup M(A_k)
  \]

- **Remember**: unlike regular set-theoretic union, the order of overriding union matters, and this must be understood as associating left-to-right

Semantics of Our Simple Program

- We can now provide the set of state-transitions for each sequence of variable declarations and assignments in example program:

  ```
  int i;
  int j;
  main() {
    int mine;
    i = 13;
    j = 1;
    mine = 0;
  }
  ```

  Start state: \( \sigma_0 = \emptyset \)

  **Declarations (1):**

  \( \sigma_1 = \sigma_0 \cup \{(i, \text{undef}), (j, \text{undef})\} \)

  \( = \{(i, \text{undef}), (j, \text{undef})\} \)

  ```
  main() {
    int mine;
    i = 13;
    j = 1;
    mine = 0;
  }
  ```

  **Declarations (2):**

  \( \sigma_2 = \sigma_1 \cup \{(\text{mine}, \text{undef})\} \)

  \( = \{(i, \text{undef}), (j, \text{undef}), (\text{mine}, \text{undef})\} \)

A Slightly More Complex Language

- Suppose we add more complex assignments to our language, allowing the use of integers, variables, or the results of (prefix notation) arithmetic operations on other expressions:

  ```
  Program \rightarrow \text{Declaration}^* \text{Main}
  Main \rightarrow \text{main}() \{ \text{Declaration}^* \text{Assignment}^* \}
  Declaration \rightarrow \text{int} \text{Variable}^
  Assignment \rightarrow \text{Variable} \ast \text{Expression}^
  Expression \rightarrow \text{Int} \mid \text{Variable} \mid \text{BinaryOperation}
  Variable \rightarrow \text{Letter} \text{Letter}^
  Operator \rightarrow \text{a} \mid \text{b} \mid \ldots \mid \text{z}
  Int \rightarrow 0 \mid \text{LeadDigit} \text{Digit}^
  LeadDigit \rightarrow 1 \mid 2 \mid \ldots \mid 9
  Digit \rightarrow 0 \mid 1 \mid \ldots \mid 9
  BinaryOperation \rightarrow \text{Operator} \text{Expression} \text{Expression}
  Operator \rightarrow + \mid - \mid * \mid /
  ```

  ```
  int i;
  int j;
  main() {
    int k;
    i = 10;
    j = i;
    k = + i * j;
  }
  ```
The Meaning of Expressions

Now, since an Expression has three possibilities, we define its meaning using three cases, as well:

\[ M(\text{Expression}, \sigma) = \begin{cases} 
\text{Val}(\text{Int}, \sigma) & \text{if Int} \\
\text{Val}(\text{Variable}, \sigma) & \text{if Variable} \\
M(\text{BinaryOperation}, \sigma) & \text{if BinaryOperation} 
\end{cases} \]

The Val function is an intermediary, defined straightforwardly for integers and variables:

\[ \forall i \in \mathbb{Z}, \forall \sigma, \text{Val}(i, \sigma) = i \]

\[ \forall v, \forall \sigma = \{\ldots, \langle v, x \rangle, \ldots\}, \text{Val}(v, \sigma) = x \]

The Meaning of Assignment

We can again understand assignments as overriding unions:

\[ M(\text{Assignment}, \sigma) = \sigma \cup M(\text{Assignment}) \]

\[ = \sigma \cup (M(\text{Variable} = \text{Expression})) \]

Again, for atomic (integer/variable) expressions, this is quite simple, e.g.:

\[ M(i = 3, \{\langle i, \text{undef} \rangle, \langle j, \text{undef} \rangle\}) = \{\langle i, \text{undef} \rangle, \langle j, \text{undef} \rangle\} \cup \{\langle i, 3 \rangle\} \]

\[ = \{\langle i, 3 \rangle, \langle j, \text{undef} \rangle\} \]

\[ M(j = i, \{\langle i, 3 \rangle, \langle j, \text{undef} \rangle\}) = \{\langle i, 3 \rangle, \langle j, \text{undef} \rangle\} \cup \{\langle j, M(i)\rangle\} \]

\[ = \{\langle i, 3 \rangle, \langle j, \text{undef} \rangle\} \cup \{\langle j, 3 \rangle\} \]

\[ = \{\langle i, 3 \rangle, \langle j, 3 \rangle\} \]

Next Week

Topic: Semantics

Homework 02: Today, 08 March, 5:00 PM on D2L

Midterm Exam: Friday, 15 March, in class

Open book, open notes

Covers ‘formal’ material to denotational semantics (not Scala programming)

Practice exam available by this weekend

Special exam session: Thursday, 14 March, 11:00 AM, 3215 Centennial

Office Hours: Wing 210

Monday: 9:00 AM – 10:30 AM
Tuesday: 3:00 PM – 4:00 PM
Wednesday: 9:00 AM – 10:30 AM
Thursday: 2:00 PM – 3:00 PM
Friday: 9:00 AM – 10:30 AM