Review: The Meaning of Expressions

Now, since an Expression has three possibilities, we define its meaning in 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

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

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

Review: 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 → Declaration* Main
  Main → main() { Declaration* Assignment* }
Declaration → int Variable;
Assignment → Variable = Expression;
Expression → Int | Variable | BinaryOperation
Variable → Letter Letter*;
Letter → a | b | \ldots | z
Int → 0 | LeadDigit Digit*
LeadDigit → 1 | 2 | \ldots | 9
Digit → 0 | 1 | \ldots | 9
BinaryOperation → Operator Expression Expression
Operator → * | - | \* | /
```

```
int i;
int j;
main() {
  int k;
i = 10;
  j = i;
k = i + j;
}
```

Review: 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.:

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

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

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

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

\[ M(j = 1, \langle i, 3 \rangle, \langle j, \text{undef} \rangle) = \langle i, 3 \rangle, \langle j, \text{undef} \rangle \cup \{ (j, M(1)) \} \]

\[ = \langle i, 3 \rangle, \langle j, \text{undef} \rangle \cup \{ (j, 1) \} \]

\[ = \langle i, 3 \rangle, \langle j, 1 \rangle \]

\[ M(k = i + j, \langle i, 3 \rangle, \langle j, 3 \rangle) = \langle i, 3 \rangle, \langle j, 3 \rangle \]

\[ = \langle i + j, 3 \rangle, \langle j, 3 \rangle \]
The Meaning of Expressions, Continued

**BinaryOperation** \(\rightarrow\) **Operator Expression Expression**

**Operator** \(\rightarrow\) + | - | * | /

- The meaning of the third type of **Expression**, **BinaryOperation**, is more complex to define.

- We can then define an intermediary function, **ApplyOp**, that gives the semantics of integer arithmetic:

\[
\text{ApplyOp} : \text{Operator} \times \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}
\]

\[
\text{ApplyOp}(op, z_1, z_2) = \begin{cases} 
    z_1 + z_2 & \text{if } op == + \\
    z_1 - z_2 & \text{if } op == - \\
    z_1 \times z_2 & \text{if } op == * \\
    \frac{z_1}{z_2} & \text{if } op == / 
\end{cases}
\]

We can now evaluate the meaning of the complex assignment to \(i\)

- (Infix version: \(i = j + k * 2\))

---

**Meaning of Arithmetic Expressions**

```c
main() {
    int i;
    int j;
    int k;
    j = 2;
    k = 3;
    i = j + k * 2;
}
```

- **start state**: \(\sigma_0 = \emptyset\)

- **Declarations**:

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

\[
= \{(i, \text{undef}), (j, \text{undef}), (k, \text{undef})\}
\]

- **Assignments**:

\[
\sigma_2 = \sigma_1 \cup \{(j, 2)\} \cup \{(k, 3)\}
\]

\[
= \{(i, \text{undef}), (j, 2), (k, \text{undef})\} \cup \{(k, 3)\}
\]

\[
= \{(i, \text{undef}), (j, 2), (k, 3)\}
\]

- **Assignment**:

\[
\sigma_3 = \sigma_2 \cup \{(i, M(\ast (j * k) 2, \sigma_2))\}
\]

\[
= \sigma_2 \cup \{(i, 8)\}
\]

\[
= \{(i, 8), (j, 2), (k, 3)\}
\]

**Meaning of Arithmetic Expressions, Continued**

\[
M(\ast (j * k) 2, \sigma_2)
\]

\[
= \text{ApplyOp}(\ast, M(j, \sigma_2), M(\ast k 2, \sigma_2))
\]

\[
= \text{ApplyOp}(\ast, \text{Val}(j, \sigma_2), M(\ast k 2, \sigma_2))
\]

\[
= \text{ApplyOp}(\ast, 2, M(\ast k 2, \sigma_2))
\]

\[
= \text{ApplyOp}(\ast, 2, \text{ApplyOp}(\ast, M(k, \sigma_2), M(2, \sigma_2)))
\]

\[
= \text{ApplyOp}(\ast, 2, \text{ApplyOp}(\ast, \text{Val}(k, \sigma_2), \text{Val}(2, \sigma_2)))
\]

\[
= \text{ApplyOp}(\ast, 2, \text{ApplyOp}(\ast, 3, 2))
\]

\[
= \text{ApplyOp}(\ast, 2, 6)
\]

\[
= 8
\]
A Complication: Variable Instantiation

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

Two clauses in the meaning function for an expression can involve the appearance of variables—either assigned directly, or in some sub-expression of a binary operation application.

For the latter case, the $ApplyOp$ function assumes that the meaning of each of the sub-expressions will be an integer value.

**A question:** is this code legal?

```java
main() {
    int k;
    k = + k 1;
}
```

Uninitialized Variables, I

- In many languages, variables must be initialized before use.
- In Java (and similarly in languages like C++), *instance* variables are given default initial values if necessary, but *local* variables must be initialized explicitly before use.

```java
public class Test {
    private int x;
    private void test() {
        System.out.printf("Instance: \%dn", x + 1);
    }
}
```

Legal: Would print 1.
Illegal: Won’t compile.

Uninitialized Variables, II

- This is **not** true of all languages: the C++/Java restriction was a decision to distinguish from languages like C, where uninitialized variables have indeterminate value.
- Depending upon what a compiler writer decides, such variables may end up taking on the values of arbitrary memory registers when first used without explicit initialization.

```java
void test() {
    int y;
    printf("%d\n", y + 1);
}
```

Legal: May print 1.
May print something else altogether!

Uninitialized Variables, III

- Other languages give default values to **all** variables.
- The language D re-engines C++ with features of some other languages like Python and Ruby, and gives all variables some initial value, no matter what (even if not always very useful).

```java
void test() {
    int x;
    bool y;
    double z;
    printf("%d", x);
    printf("%s", y);
    printf("%f", z);
}
```

Legal: Prints: 0 false nan
No Uninitialized Variables

- In our example language, we will have very strict rules.
- Every variable must be explicitly initialized \textit{before} use on the right-hand side of any assignment.
- All of the following are \textit{illegal} and the meaning function does not apply to any of them:

```
main() {
  int k;
  main() {
    k = + k 1;
  }
}
int k;
main() {
  int j;
  j = k;
}
```

Adding Functions to Our Language

- Our current language allows global/local declarations, and a \texttt{main()} function with assignments and arithmetic.
- We will add the ability to call other functions (void and non-void).
- We will also allow variables to be initialized at the time of declaration, and for method bodies to combine various types of statements in whichever order we like (no longer having to do all declarations first).

```
int i = 0;
void addtoi(int x) {
  i = i + x;
}
int sum(int x, int y) {
  int sum = x + y;
  return sum;
}
main() {
  int j = 2;
  addtoi(j);
  int k = sum(i, j);
}
```

A More Complex Language

```
Program → Declaration* Function* Main
Function → Void | NonVoid
  Void → void Name(Parameters) { Statement* }
  NonVoid → int Name(Parameters) { Statement* Return }
Name → Letter Letter*
Parameters → ε | int Variable(, int Variable)*
Return → return Expression;
Main → main() { Statement* }
Statement → Declaration | Assignment | Call
Declaration → int Variable; | int Assignment
Assignment → Variable = Expression;
Expression → Int | Variable | BinaryOperation
Call → Name(Arguments);
Arguments → ε | Argument(, Argument)*
Argument → Variable | Int
```

Note: unlike the other braces and parentheses, the ones in \textit{Parameters} and \textit{Arguments} are part of the formal grammar, not part of programming language itself.

Rest of the language (Variable, Int, etc.) is the same as it was before.

Functions in the New Language

```
Function → Void | NonVoid
  Void → void Name(Parameters) { Statement* }
  NonVoid → int Name(Parameters) { Statement* Return }
Name → Letter Letter*
```

- The language allows the specification of other functions, before main(), with or without a return value, and with any number of integer parameters.

```
int i = 0;
void f() {
  i = + 1;
}
int i = 0;
void g() {
  return i + 1;
}
int i = 0;
void h(int x) {
  i = + x;
}
```
Function Calls

Statement → Declaration | Assignment | Call
Expression → Int | Variable | BinaryOperation | Call
Call → Name(Arguments);
Arguments → ε | Argument(, Argument)∗
Argument → Variable | Int

We can now call a function as an imperative statement, assign it to a variable as an expression, or use it as in a return statement.

```c
int i = 0;
void f(int x) {
    i = i + 1 x;
}
void g() {
    f(3);
}
```

```c
int f(int x) {
    return * x 2;
}
void g() {
    int i = f(2);
    int f(int y) {
        return f(y);
    }
}
```

Other Features of the Language

Declaration → int Variable;
Assignment → Variable = Expression;

The language is now more flexible, allowing both stand-alone variable declaration and assignment, as well as combining the two into a single step.

```c
int i = 0;
int j;
main() {
    j = 3;
    int k = 1;
    i = + i k;
}
```

Other Features of the Language

Void → void Name(Parameters) {Statement∗}
NonVoid → int Name(Parameters) {Statement∗ Return}
Main → main() {Statement∗}
Statement → Declaration | Assignment | Call

The language also allows us to write main() and other functions with statements in any order we want, rather than having all declarations first.

```c
int f(int x) {
    return * 2 x;
}
main() {
    int i = 0;
    i = f(5);
}
```

```c
int f(int x) {
    return * x 1;
}
int g(int y) {
    return f(y);
}
```

Semantic Pre-Processing for the Function-Language

As usual, a number of new semantic scoping and pre-processing rules are defined for the new language.

Each rule deals with situations for which the formal semantic model has nothing to say.

These rules are enforced before we worry about the meaning of any program attempts.

```c
int i = 0;
void addtoi(int x) {
    i = i + x;
}
main() {
    int j = 2;
    addtoi(j);
    int k = sum(i, j);
}
```
Scoping and Identifier Rules

1. All scoping is outside-in, using standard Java-style rules
2. All identifiers in a scope must be unique
3. No global shadowing (function variables same as global ones) allowed
4. No parameter shadowing (function variables same as parameters) allowed

```java
int i = 0;
void f(int x) {
    int j = +i x;
}
main {
    int k = j;
}
```

```java
int f(int x) {
    return * x 2;
}
main {
    int i = f(3);
}
```

```java
int i = 0;
int f(int x) {
    return +i x;
}
void f() {
    int i = f(2);
}
main {
    int i = f(3);
}
```

This Week

- **Topic:** Semantics
- **Midterm Exam:** this Friday in class
  - Open book, open notes
  - Covers "formal" material to denotational semantics (not Scala)
  - Practice exam available, answer key tomorrow; some review in class on Wednesday. If you have questions about the practice exam
  - **Special exam session:** Thurs., 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