Declarations

- In programming languages, a <u>declaration</u> specifies the identifier, type, and other properties (e.g. 'static') of language elements such as variables and functions. [1]
- A declaration is used to announce the <u>existence</u> of the language element as part of the semantics of the language.
- Many languages (such as C and Java) require variables to be declared before use.

[1] loosely quoted from http://en.wikipedia.org/wiki/Type_declaration

Declarations

Let's extend our IMP programming language with variable declarations. We continue to assume that the only type we have in our language is the integer type. This means the only job of the variable declaration at this point is to announce the existence of a variable.

We will enforce two rules:

- We need to declare variables before using them ('$\mathbf{var}(x)$' syntax), using an undeclared variable will result in an error.
- Declaring the same variable twice is not allowed.

Consider the following programs, determine if they are valid or not valid according to the rules:

$$\mathbf{assign}(x, 3)$$

$$\mathbf{var}(x)\ \mathbf{seq}\ \mathbf{assign}(x, 3)$$

$$\mathbf{var}(x)\ \mathbf{seq}\ \mathbf{assign}(x, 3)\ \mathbf{seq}\ \mathbf{var}(x)\ \mathbf{seq}\ \mathbf{assign}(x, 4)$$

$$\mathbf{var}(x)\ \mathbf{seq}\ \mathbf{assign}(x, 3)\ \mathbf{seq}\ \mathbf{assign}(x, \mathbf{add}(y, 1))$$

$$\mathbf{var}(x)\ \mathbf{seq}\ \mathbf{assign}(x, 3)\ \mathbf{seq}\ \mathbf{var}(y)\ \mathbf{seq}\ \mathbf{assign}(x, \mathbf{add}(y, 1))$$

Declaration Semantics

We add the '**var**(x)' command to the syntax:

```
A ::= n
   |  x
   |  add(A,A)
   |  sub(A,A)
   |  mult(A,A)

B ::= true
   |  false
   |  eq(A,A)
   |  le(A,A)
   |  not(B)
   |  and(B,B)
   |  or(B,B)
```

```
C ::= skip
   |  var(x)
   |  assign(x,A)
   |  seq(C,C)
   |  if(B,C,C)
   |  whiledo(B,C)
```

Declaration Semantics

From a semantics perspective, the '**var**(x)' command needs to remember that the variable x was declared in the program. This gives rise to the following rule,

```
(var(X),State) -->> OState :-          % decl, if lookup is successful
    lookup(X,State,_),!,         % then var(x) must fail, only
    fail.        % one var declaration allowed

(var(X),State) -->> OState :-          % decl, we have shown that lookup is not
    put(X,0,State,OState),!.           % successful, therefore enter the new var
```

This implies that our lookup needs to fail in the initial state,

```
lookup(_,s0,_) :- !,fail.
```

Declaration Semantics

Assignments can only succeed if the variable on the left side was declared,

```
(assign(X,A),State) -->> OState :-      % assignment
    lookup(X,State,_),                 % only allowed to assign to variables
    (A,State) -->> ValA,                % that have been declared
    put(X,ValA,State,OState),!.
```

Since this is the only semantic rule for assignments, if the lookup fails, the program will fail. Expressions with variables can only be evaluated if the variable has been declared,

```
(X,State) -->> Val :-                 % variables
    atom(X),
    lookup(X,State,Val),!.
```

Note: Nothing has changed with this semantic rule, except that lookup fails if the X is not declared.

Declaration Semantics

```
?- ['sem-decl.pl'].
%   xis.pl compiled 0.00 sec, 6,920 bytes
%  preamble.pl compiled 0.00 sec, 8,084 bytes
%  xis.pl compiled 0.01 sec, 148 bytes
% sem-decl.pl compiled 0.01 sec, 14,948 bytes
true.

?- (assign(x, 3),s0)-->>V.
false.

?- ((var(x) seq assign(x, 3)),s0)-->>V.
V = state([bind(3, x), bind(0, x)], s0).

?- ((var(x) seq assign(x, 3) seq var(x) seq assign(x, 4)),s0)-->>V.
false.

?- ((var(x) seq assign(x, 3) seq assign(x, plus(y, 1))),s0)-->>V.
false.

?- ((var(x) seq assign(x, 3) seq var(y) seq assign(x, add(y, 1))),s0)-->>V.
V = state([bind(1, x), bind(0, y), bind(3, x), bind(0, x)], s0).

?-
```

Types

Types and type systems are fundamental in modern programming languages. Typed variables and expressions in programs allow the language system to assist the programmer by detecting *illegally typed expressions* which usually constitutes a logic/programming error.

We define a type as follows:

> *A type is a set of values.*

This means the type ‘`real`’ constitutes the set of all real values and the type ‘`int`’ constitutes the set of all integer values.

Types

When we combine the notion of a type and variable declarations we restrict what we are allowed to store in the variable. For example, the declaration in C,

```
int v;
```

restricts the values that are allowed to be stored in the variable 'v' to the set of integer values.

Limiting the kind of values a variable is allowed to assume will allow the system to catch errors. Consider the C code snippet,

```
int i = "1";
```

The compiler will reject this with a type error.[2]

[2] However, in C the statement `int i = '1'` with `'1'` being a character constant is legal – the set of character constants is a *subtype* of the integers.

Types

Our notion of a type as a set of values extends to more complex types. Consider the array declaration,

```
int a[5];
```

This declaration limits the values that the variable ‘a’ can assume to arrays of size 5 with integer elements. Here are some example values from that set,

$$\{[1,2,3,4,5],[102,4026,798,2,999],[-22,4,56,-654,0],\ldots\}$$

Types

Type errors can also appear in expressions. Consider the C statement,

```
String s = "hello world" + 3.0;
```

However, many languages allow for certain type combinations to appear in expressions. Consider the C code,

```
int i = 3;
float f = i * 5.5;
```

Here, even though the operands of the multiplication operator are of different types, C will allow this kind of expression. Mixed type expressions are usually allowed as long as the types involved have a *subtype/supertype* relationship.

Types

Interestingly, the notion of a type as a set of values also extends to object oriented languages if we view objects as values in a particular set of object (a particular type!). Consider the following Java snippet,

```
class Foobar {...};

Foobar o = new Foobar();
```

Here the class statement introduces the new type 'Foobar' as a set of objects that can be instantiated from the class. The next statement declares a variable 'o' of type 'Foobar' and thereby restricts the variable to only accept values (objects) from the set 'Foobar'.

The following code would fail in a Java program:

```
class Foobar {...};
class Goobar {...};

Foobar o = new Goobar();
```

It is precisely these kinds of errors that type systems are designed to catch.

Types

Our view of a type as a set allows us to develop the notion of a *subtype*:

> *If the values of a type are fully contained within another type, then we call the former a subtype of the latter.*

More precisely, let A and B be types and interpreting these types as sets, then A is a *subtype* of B if

$$A \subset B.$$

Or conversely we call B a *supertype* of A.

In Java and C we have the following *type hierarchy*:

$$\texttt{char} \subset \texttt{short} \subset \texttt{int} \subset \texttt{float} \subset \texttt{double}$$

Not all programming languages support type hierarchies. The language ML, for example, has no notion of subtype. Here, all types are completely separate sets, subset inclusion is not allowed.

Types

If a language supports subtypes then we can convert the types of expressions along those subtype/supertype relationships.

- **Widening conversion** – here we convert the value of an expression from a *subtype to its supertype*. This is often also referred to as *type promotion*. Consider the code snippet,

  ```
  float f = 3;
  ```

  To make this statement work the language system will promote the integer constant 3 to a *float* value and then assign it to the variable f.

- **Narrowing conversion** – here we convert the value of an expression from a *supertype to a subtype*. Consider,

  ```
  int i = 3.6;
  ```

  The programming language C will simply truncate the value to turn the floating point value to an integer value.

Types

Expressions that have types which are not related along subtype/supertype relations cannot be converted and therefore typically generate errors in a language system. Consider the C program snippet from before,

```
String s = "hello world" + 3.0;
```

In C, $\texttt{String} \not\subset \texttt{float}$ and $\texttt{float} \not\subset \texttt{String}$, therefore the above statement cannot be executed.

Typed Arithmetic

We experiment with a very simple type system. It only has two types, namely, `int` and `real`.

We assume that these two types are related via a subtype/supertype relationship:

$$\texttt{int} \subset \texttt{real}.$$

This will allow us to implement type promotion and narrowing conversion in our type system.

Typed Arithmetic

We introduce a new syntactic domain

$$\mathbf{Type} = \{\mathbf{int}, \mathbf{real}\}$$

We can now have declarations of the form

$$\begin{aligned} C &::= \mathbf{var}(x, T) \\ T &::= \mathbf{int} \mid \mathbf{real} \end{aligned}$$

In addition we introduce the syntactic domain of floating point values $\mathbf{R}$ (with the semantic denotation of $\mathbb{R}$) such that

$$A ::= v$$

where $v \in \mathbf{I} \cup \mathbf{R}$ can be either an integer or floating point constant.

Type Declaration Semantics

Putting this all together,

```
A ::= v
    |  x
    |  add(A,A)
    |  sub(A,A)
    |  mult(A,A)

B ::= true
    |  false
    |  eq(A,A)
    |  le(A,A)
    |  not(B)
    |  and(B,B)
    |  or(B,B)
```

```
T ::= int  |  real

C ::= skip
    |  var(x,T)
    |  assign(x,A)
    |  seq(C,C)
    |  if(B,C,C)
    |  whiledo(B,C)
```

Typed Arithmetic

Our semantics needs to be able to deal with the following programs:

1. **var**$(x,$ **real**$)$ **seq** **assign**$(x, 3)$ (type promotion)
2. **var**$(y,$ **int**$)$ **seq** **assign**$(y, 3.5)$ (narrowing conversion)
3. **var**$(x,$ **real**$)$ **seq** **assign**$(x,$ **add**$(3.5, 2))$ (type promotion in expressions)
4. **var**$(x,$ **real**$)$ **seq** **var**$(y,$ **int**$)$ **seq** **assign**$(y, 1)$ **seq** **assign**(x, y) (type promotion of variable values in expressions)

Typed Arithmetic

Semantics:

- We think about the types as sets, however, in our semantics the type names can just be viewed as tags attached to variable declarations. Since we know that there is a subtype/supertype relation between the types we can use the tags to infer type promotions or narrowing conversions.
- We attach type tag names to variable binding terms.
- We do all our arithmetic in floating point, truncating the value if we need to.

Typed Arithmetic

The semantic rule for a variable declaration,

```
(var(X,int),State) -->> _ :-                % var %decl%
    lookup(X,_,State,_),!,
     fail.

(var(X,int),State) -->> OState :-       % var %decl%
    put(X,int,0,State,OState),!.

(var(X,real),State) -->> _ :-           % var %decl%
    lookup(X,_,State,_),!,
    fail.

(var(X,real),State) -->> OState :-      % var %decl%
    put(X,real,0.0,State,OState),!.
```

Typed Arithmetic

The semantic rules for an assignment statement,

```
(assign(X,A),State) -->> OState :-      % assignment to real var  %decl%
    lookup(X,real,State,_),
    (A,State) -->> ValA,
    FValA xis float(ValA),
    put(X,real,FValA,State,OState),!.

(assign(X,A),State) -->> OState :-     % assignment to int var  %decl%
    lookup(X,int,State,_),
    (A,State) -->> ValA,
    IValA xis truncate(ValA),
    put(X,int,IValA,State,OState),!.
```

Typed Arithmetic

The semantic rules for constants and variables,

```
(C,_) -->> FVal :-                 % int constants %decl%
    int(C),
    FVal xis float(C),!.             % promote from int to real

(C,_) -->> C :-                    % real constants %decl%
    real(C),!.

(X,State) -->> FVal :-             % int variables %decl%
    atom(X),
    lookup(X,int,State,IVal),
    FVal xis float(IVal),!.

(X,State) -->> FVal :-             % real variables %decl%
    atom(X),
    lookup(X,real,State,FVal),!.
```

Typed Arithmetic

```
?- ['sem-type.pl'].
%   xis.pl compiled 0.01 sec, 7,792 bytes
%  preamble.pl compiled 0.01 sec, 8,956 bytes
%  xis.pl compiled 0.00 sec, 148 bytes
% sem-type.pl compiled 0.01 sec, 16,828 bytes
true.

?- ((var(x, real) seq assign(x, 3)),s0) -->> V.
V = state([bind(3.0, real, x), bind(0.0, real, x)], s0).

?- ((var(y, int) seq assign(y, 3.5)),s0) -->> V.
V = state([bind(3, int, y), bind(0, int, y)], s0).

?- ((var(x, real) seq assign(x, add(3.5, 2))),s0) -->> V.
V = state([bind(5.5, real, x), bind(0.0, real, x)], s0).

?- ((var(x, real) seq var(y, int) seq assign(y,1) seq assign(x, y)),s0) -->> V.
V = state([bind(1.0, real, x), bind(1, int, y), bind(0, int, y), bind(0.0, real, x)], s0).

?-
```

Typed Arithmetic

What kind of changes would we have to make to the semantic specification if we wanted to keep integer arithmetic as integer arithmetic and only promote the type when necessary?

Typed Arithmetic

Assume that the subtype/supertype relationship does not exist, i.e., `int` $\not\subset$ `real` . Further, assume that we have two new additional operators as part of our programming language syntax:

$$\text{A} ::= \textbf{promote}(\text{A}) \mid \textbf{narrow}(\text{A})$$

where the first operator promotes the type of an arithmetic expression from `int` to `real` and the second operator narrows the type of an arithmetic expression from `real` to `int`. Since there is not subtype/supertype relationship between the types all mixed type expression will fail unless we insert our explicit type conversion operators,

1. $\textbf{var}(x, \textbf{real})\ \textbf{seq}\ \textbf{assign}(x, \textbf{promote}(3))$
2. $\textbf{var}(y, \textbf{int})\ \textbf{seq}\ \textbf{assign}(y, \textbf{narrow}(3.5))$
3. $\textbf{var}(x, \textbf{real})\ \textbf{seq}\ \textbf{assign}(x, \textbf{add}(3.5, \textbf{promote}(2)))$
4. $\textbf{var}(x, \textbf{real})\ \textbf{seq}\ \textbf{var}(y, \textbf{int})\ \textbf{seq}\ \textbf{assign}(x, \textbf{promote}(y))$

What kind of changes do you envision for our type system specification?