Static Semantics

- So far, our parser correctly parses syntactically correct programs
- Semantic consistency in a programming language is usually implemented via typing entities and checking for the consistent use of the entities
- Implementing the Static Semantics requires:
  - implementation of an identifier table to store information about identifiers
  - implementation of a type module which checks the usage of the identifiers
- Information about identifiers is already present in our AST
  - We can determine if an identifier has been declared by looking up the tree to see if we can find any declaration for the identifier
  - There is no real need for an explicit identifier table, except for efficiency in lookups…
  - Likewise type information is also stored about identifiers…

Generic Semantics Implementation

- We can check correct use in our program by simply walking the AST
- Note that the AST implementation makes it easy to propagate information from the root of the tree to the leaf nodes.
  - It is often more difficult to push information from the leaf nodes to the root – even though this is actually what we want.
  - For this reason, we will often need to build an explicit identifier table data structure and type structure.
- So static semantics is:
  - Walk the AST, enter identifiers into the identifier table as you encounter declarations
  - Walk the AST, checking for consistent use
- Question – how do we represent the concept of type?

Representing Type

- Types –
  - Base types of the programming language
  - Type constructors
  - Type rules
    - Type inference rules (integer -> real conversions, for example)
- Base types are enumerable, therefore trivially represented.
- Type constructors mean the type representation is recursive, consider the Pascal array declaration:
  ```
  type example = array [1..10] of T;
  ```
- Type constructors are (thankfully) also enumerable:
  - Finite collections (arrays, records, sets, …)
  - Infinite collections (files, streams, …)
  - Functions (methods)
  - Classes
  - Etc
- It is the compiler writer’s task to figure out how to do this!

Representing Type (cont.)

- The type representation can be “borrowed” from the AST
  - Our AST will have all the necessary information in it
- Once we have a representation for the type, we need to be able to see if entities are being used consistently.
  - We need to be able to compare types
- This is not easy!
  - A type representation will be a recursive tree like structure:
    ```
    Type = array[1..10] of array[-100..1] of char:
    ```

Comparing Types

- There are two basic forms of type comparison:
  - Name equivalent type systems
  - Structure equivalent type systems
- The type rules of the language will define which is appropriate
  - Name equivalence is easy – the two entities must have the same type
  - Structural equivalence is harder – you need to compare the two trees.
- Our type management system will include a boolean valued method called “comptype(t1, t2)” which does the comparison as specified.
  - There will also be a special type “UNKNOWN” that the compiler will implement.
  - Type UNKNOWN is compatible with every other type
  - Used for error recovery in our compiler, when we forget to declare a variable, for example, we will enter it as “type UNKNOWN” so we don’t get too many errors.

Aside - Classes

- In an OO language, the notion of type includes the Object Oriented notion of class and superclass.
- We need to be able to determine which class or superclass method is being referred to – the inheritance hierarchy forms (in general) a lattice.
- Our type comparison mechanism remains basically the same – we walk the lattice looking for the first method whose signature matches.
  - The primary difference for the compiler writer is that the superclass definitions are almost always pre-compiled in some other file
Modifying the AST

- Now we can represent type information, we need to augment our AST in places.
- Consider the assignment statement:
  \[ x := 2 \ast 3; \]
- In our AST, this will appear as follows:

```
Assign
  x
     
2
3
```

- To implement types, we need to know how the type of the arithmetic expression is determined.
- We propagate type information back up the tree

Propagation of Type Information

- This is done by a tree walk
- Traversing a child node returns the type of the child to the parent.
- In arithmetic expressions (for example) we need to rigidly follow the type rules of the language

```
Real
  Assign
    x
     
Int
2
3
Int
```

Booleans in RCL

- There is no explicit type boolean in RCL – so what do you do?
  ```
  short foo;
  ...
  foo := true;
  if foo and true then ...
  ```
- This is legal RCL...
- The rules say:
  > "The type checking should exclude, as far as possible, the application of arithmetic operators to boolean operands, and of boolean operators to arithmetic operands. Thus, the expression (v < 5) and 3 has a type error."
- We can check that but what if “3” was replaced by a short variable whose value was 3?
  - Some checks must be deferred till run-time
- Languages without explicit typing always need this