Incorrect Programs

- Lexically and syntactically correct programs may still contain other errors!
- Lexical and syntax analysis are not powerful enough to ensure the correct usage of variables, objects, functions, statements, etc.
- Example: lexical analysis does not distinguish between different variable or function identifiers (it returns the same token for all identifiers)

\[
\begin{align*}
\text{int } a; & \quad \text{int } a; \\
a = 1; & \quad b = 1;
\end{align*}
\]

Incorrect Programs

- Example 2: syntax analysis does not correlate the declarations with the uses of variables in the program:

\[
\begin{align*}
\text{int } a; & \\
a = 1; &
\end{align*}
\]

- Example 3: syntax analysis does not correlate the types from the declarations with the uses of variables:

\[
\begin{align*}
\text{int } a; & \quad \text{int } a; \\
a = 1; & \quad a = 1.0;
\end{align*}
\]

Goals of Semantic Analysis

- Semantic analysis ensure that the program satisfies a set of rules regarding the usage of programming constructs (variables, objects, expressions, statements)
- Examples of semantic rules:
  - Variables must be defined before being used
  - A variable should not be defined multiple times
  - In an assignment statement, the variable and the assigned expression must have the same type
  - The test expr. of an if statement must have boolean type
- Two main categories:
  - Semantic rules regarding types
  - Semantic rules regarding scopes

Type Information

- Type information describes what kind of values correspond to different constructs: variables, statements, expressions, functions

\[
\begin{align*}
\text{variables: } & \text{int } a; & \text{integer} \\
\text{expressions: } & (a+1) == 2 & \text{boolean} \\
\text{statements: } & a = 1.0 & \text{floating-point} \\
\text{functions: } & \text{int pow(int n, int m)} & \text{int } \times \text{int } \rightarrow \text{int}
\end{align*}
\]
Type Checking

- Type checking = set of rules which ensures the type consistency of different constructs in the program

- Examples:
  - The type of a variable must match the type from its declaration
  - The operands of arithmetic expressions (+, *, -) must have integer types; the result has integer type
  - The operands of comparison expressions (==, !=) must have integer or string types; the result has boolean type

Scope Information

- Scope information = characterizes the declaration of identifiers and the portion of the program where it is allowed to use each identifier
  - Example identifiers: variables, functions, objects, labels
  - Lexical scope = textual region in the program
    - Statement block
    - Formal argument list
    - Object body
    - Function or method body
    - Module body
    - Whole program (multiple modules)
  - Scope of an identifier: the lexical scope its declaration refers to

Scope Information

- Scope of formal arguments of functions:
  ```java
  int factorial(int n) {
      ...  // scope of argument n
  }
  ```

- Scope of labels:
  ```java
  void f() {
      ...  // scope of label l
      ...  // scope of label l
  }
  ```

Scope Information

- Scope of variables in statement blocks:
  ```java
  { int a;  // scope of variable a
    ...  // scope of variable b
  }
  ```

- Scope of global variables: current module
- Scope of external variables: whole program

Scope Information

- Scope of object fields and methods:
  ```java
  class A {
      private int x;
      public void g() { x += 1; }  // scope of field x
      ...  // scope of method f
  }
  ```

  class B extends A {
      ...  // scope of method f
  }

  public int h() { g(); }  // scope of method f
  ```
Semantic Rules for Scopes

- Main rules regarding scopes:
  Rule 1: Use each identifier only within its scope
  Rule 2: Do not declare identifiers of the same kind with identical names more than once in the same lexical scope
- Can declare identifiers with the same name with identical or overlapping lexical scopes if they are of different kinds

```c
class X {
  int x;
  void X(int x) {
    int y;
    x = for(1);
    X x = 1;
  }
};
```

Not Recommended!

Symbol Tables

- Semantic checks refer to properties of identifiers in the program — their scope or type
- Need an environment to store the information about identifiers — symbol table
- Each entry in the symbol table contains:
  - the name of an identifier
  - additional information: its kind, its type, if it is constant, ...

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Kind</th>
<th>Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>func</td>
<td>int</td>
<td>arg, int, bool, extern</td>
</tr>
<tr>
<td>m</td>
<td>arg</td>
<td>int</td>
<td>const</td>
</tr>
<tr>
<td>n</td>
<td>arg</td>
<td>int</td>
<td>const</td>
</tr>
<tr>
<td>int</td>
<td>var</td>
<td>bool</td>
<td>const</td>
</tr>
</tbody>
</table>
```

Scope Information

- How to capture the scope information in the symbol table?
- Idea:
  - There is a hierarchy of scopes in the program
  - Use a similar hierarchy of symbol tables
  - One symbol table for each scope
  - Each symbol table contains the symbols declared in that lexical scope

Example

```c
int x;
void f(int m) {
    float x, y;
    ...
    { int i, j; ... }
    { int x; k = ... }
}
int g(int n) {
    bool t;
    ...
}
```

Identifiers With Same Name

- The hierarchical structure of symbol tables automatically solves the problem of resolving name collisions (identifiers with the same name and overlapping scopes)
- To find which is the declaration of an identifier that is active at a program point:
  - Start from the current scope
  - Go up in the hierarchy until you find an identifier with the same name

Example

```c
int x;
void f(int m) {
    float x, y;
    ...
    { int i, j; x = 1; }
    { int k; k = 2; }
}
int g(int n) {
    bool t;
    x = 3;
    ...
}
```
Catching Semantic Errors

```c
int x;
void f(int m) {
    float x, y;
    ...
    { int i; x = 1; }
    { int x; i = 2; }
}
int g(int n) {
    bool t;
    x = 3;
    x = 1
    i = 2
    }
```

Symbol Table Operations

- Two operations:
  - To build symbol tables, we need to insert new identifiers in the table.
  - In the subsequent stages of the compiler, we need to access the information from the table: use a lookup function.
- Cannot build symbol tables during lexical analysis.
- Hierarchy of scopes encoded in the syntax.
- Build the symbol tables:
  - While parsing, using the semantic actions.
  - After the AST is constructed.

List Implementation

- Simple implementation = list.
  - One cell per entry in the table.
  - Can grow dynamically during compilation.

Hash Table Implementation

- Efficient implementation = hash table.
  - It is an array of lists (buckets).
  - Uses a hashing function to map the symbol name to the corresponding bucket. `hashfunc : string -> int`.
  - Good hash function = even distribution in the buckets.

Example:

```c
class A {
    int m() { return n(); }
    int n() { return 1; }
}
```

Summary

- Semantic checks ensure the correct usage of variables, objects, expressions, statements, functions, and labels in the program.
- Scope semantic checks ensure that identifiers are correctly used within the scope of their declaration.
- Type semantic checks ensure the type consistency of various constructs in the program.
- Symbol tables: a data structure for storing information about symbols in the program.
  - Used in semantic analysis and subsequent compiler stages.
  - Next time: type-checking.