Compiler Development (CMPSC 401)
Semantic Analysis

Janyl Jumadinova

March 12, 2019
Where we are now

Program is lexically well-formed:
- Identifiers have valid names.
- Strings are properly terminated.
- No stray characters.
Where we are now

Program is lexically well-formed:
  • Identifiers have valid names.
  • Strings are properly terminated.
  • No stray characters.

Program is syntactically well-formed:
  • Class declarations have the correct structure.
  • Expressions are syntactically valid.
Where we are now

Program is lexically well-formed:
- Identifiers have valid names.
- Strings are properly terminated.
- No stray characters.

Program is syntactically well-formed:
- Class declarations have the correct structure.
- Expressions are syntactically valid.

Does this mean that the program is legal?
A short Decaf Program

class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x = new string;

        x[5] = myInteger * y;
    }

    void doSomething2() {

    }

    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }
}
A short Decaf Program

class MyClass implements MyInterface {
    string myInteger;
    
    void doSomething() {
        int[] x = new string; // Can't multiply strings
        x[5] = myInteger * y; // Variable not declared
    }

    void doSomething() { // Can't redefine functions
    }

    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1); // Can't add void
    }
}

No main function
Semantic Analysis

- Ensure that the program has a well-defined meaning.
Semantic Analysis

- Ensure that the program has a well-defined meaning.
- Verify properties of the program that aren’t caught during the earlier phases:
  - Variables are declared before they are used.
  - Expressions have the right types.
  - Arrays can only be instantiated with NewArray.
  - Classes don’t inherit from non-existent base classes
  - ...

Once we finish semantic analysis, we know that the user’s input program is legal.
Semantic Analysis

- Ensure that the program has a well-defined meaning.

- Verify properties of the program that aren’t caught during the earlier phases:
  - Variables are declared before they are used.
  - Expressions have the right types.
  - Arrays can only be instantiated with NewArray.
  - Classes don’t inherit from non-existent base classes
  - ...

- Once we finish semantic analysis, we know that the user’s input program is legal.
Semantic Analysis

- **Static semantics**: can be analyzed at compile-time.
- **Dynamic semantics**: analyzed at runtime.
Semantic Analysis

- **Static semantics**: can be analyzed at compile-time.
- **Dynamic semantics**: analyzed at runtime.

- Not a clear distinction or boundary.
- Theory says that while some problems can be found at compile-time, not all can.
- So, must have run-time semantic checks.
Challenges in Semantic Analysis

- Reject the largest number of incorrect programs.
- Accept the largest number of correct programs.
Challenges in Semantic Analysis

- Reject the largest number of incorrect programs.
- Accept the largest number of correct programs.
- And do this quickly.
Semantic Analyzer

Role in compilers varies

- Strict boundary between parsing, analysis and synthesis.
- Generally some interleaving of three activities.
- Some compilers perform semantic analysis on intermediate forms.
Other Goals of Semantic Analysis

Gather useful information about program for later phases:

- Determine what variables are meant by each identifier.
- Build an internal representation of inheritance hierarchies.
- Count how many variables are in scope at each point.
Why can’t we just do this during parsing?
Limitations of CFG

- How would you prevent duplicate class definitions?
- How would you differentiate variables of one type from variables of another type?
- How would you ensure classes implement all interface methods?
Limitations of CFG

- How would you prevent duplicate class definitions?
- How would you differentiate variables of one type from variables of another type?
- How would you ensure classes implement all interface methods?
- For most programming languages, these are provably impossible.
  - Use the pumping lemma for context-free languages, or Ogden’s lemma.
Compiler Phases

1. Source Program
2. Lexical Analyzer
3. Syntax Analyzer
4. Abstract syntax tree
5. Semantic Analyzer
6. Intermediate Code Generator
7. AST and symbol tables
8. Code Optimizer
9. IR
10. Back end
11. Target Program
Implementing Semantic Analysis

- **Attribute Grammars**
  - Augment cup/bison/... rules to do checking during parsing.

- **Recursive Abstract Syntax Tree (AST) Walk**
  - Construct the AST, then use virtual functions and recursion to explore the tree.
  - AST: abstract representation of source program (including source program type info).
  - Common for parser to generate AST for analysis.
Today:

Scope Checking
Today:

Scope Checking

Next Time: Type Checking
The same name in a program may refer to fundamentally different things:

This is perfectly legal Java code:

```java
public class A {
    char A;
    A A(A A) {
        A.A = 'A';
        return A((A) A);
    }
}
```
The same name in a program may refer to fundamentally different things:

This is perfectly legal C++ code:

```cpp
int Awful() {
    int x = 137;
    {
        string x = "Scope!"
        if (float x = 0)
            double x = x;
    }
    if (x == 137) cout << "Y";
}
```
The scope of an entity is the set of locations in a program where that entity’s name refers to that entity.
Scope

- The **scope** of an entity is the set of locations in a program where that entity’s name refers to that entity.
- The introduction of new variables into scope may hide older variables.
- How do we keep track of what’s visible?
A symbol table is a data structure used by the compiler to keep track of identifiers used in the source program.
This is a compile-time data structure. Not used at run-time.
0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:     printf("%d,%d,%d\n", x, y, z);
4:     {
5:         int x, z;
6:         z = y;
7:         x = z;
8:         {
9:             int y = x;
10:             {
11:                 printf("%d,%d,%d\n", x, y, z);
12:             }
13:             printf("%d,%d,%d\n", x, y, z);
14:         }
15:     printf("%d,%d,%d\n", x, y, z);
16: }
0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:     printf("%d,%d,%d\n", x@2, y@2, z@1);
4:     {
5:         int x, z;
6:         z@5 = y@2;
7:         x@5 = z@5;
8:     {
9:         int y = x@5;
10:         {
11:             printf("%d,%d,%d\n", x@5, y@9, z@5);
12:         }
13:         printf("%d,%d,%d\n", x@5, y@9, z@5);
14:     }
15:         printf("%d,%d,%d\n", x@5, y@2, z@5);
16:     }
17: }
Symbol Table Operations

- Typically implemented as a **stack of maps**.
- Each map corresponds to a particular scope.
- Stack allows for easy “enter” and “exit” operations.
Symbol Table Operations

- Typically implemented as a **stack of maps**.
- Each map corresponds to a particular scope.
- Stack allows for easy “enter” and “exit” operations.
- **Symbol table operations are:**
  - **Push scope** : Enter a new scope.
  - **Pop scope** : Leave a scope, discarding all declarations in it.
  - **Insert symbol** : Add a new entry to the current scope.
  - **Lookup symbol** : Find what a name corresponds to.
Using a symbol table

To process a portion of the program that creates a scope (block statements, function calls, classes, etc.):

- Enter a new scope.
- Add all variable declarations to the symbol table.
- Process the body of the block/function/class.
- Exit the scope.
Using a symbol table

To process a portion of the program that creates a scope (block statements, function calls, classes, etc.):

- Enter a new scope.
- Add all variable declarations to the symbol table.
- Process the body of the block/function/class.
- Exit the scope.

Much of the semantic analysis is defined in terms of recursive AST traversals like this.
Scoping in Practice
Scoping in C++ and Java

```
class A {
public:
    /* ... */

private:
    B* myB
};

class B {
public:
    /* ... */

private:
    A* myA;
};
```

```
class A {
    private B myB;
};

class B {
    private A myA;
};
```
Scoping in C++ and Java

class A {
    public:
        /* ... */
    private:
        B* myB;
};

class B {
    public:
        /* ... */
    private:
        A* myA;
};

class A {
    private B myB;
};

class B {
    private A myA;
};

Error: B not declared

Perfectly fine!
Our predictive parsing methods always scan the input from left-to-right. LL(1), LR(1), etc. Since we only need one token of lookahead, we can do scanning and parsing simultaneously in one pass over the file. Some compilers can combine scanning, parsing, semantic analysis, and code generation into the same pass. These are called single-pass compilers. Other compilers rescan the input multiple times. These are called multi-pass compilers.
Our predictive parsing methods always scan the input from left-to-right. LL(1), LR(1), etc.

Since we only need one token of lookahead, we can do scanning and parsing simultaneously in one pass over the file.
Our predictive parsing methods always scan the input from left-to-right. LL(1), LR(1), etc.

Since we only need one token of lookahead, we can do scanning and parsing simultaneously in one pass over the file.

Some compilers can combine scanning, parsing, semantic analysis, and code generation into the same pass. These are called single-pass compilers.
Our predictive parsing methods always scan the input from left-to-right. LL(1), LR(1), etc.

Since we only need one token of lookahead, we can do scanning and parsing simultaneously in one pass over the file.

Some compilers can combine scanning, parsing, semantic analysis, and code generation into the same pass. These are called single-pass compilers.

Other compilers rescan the input multiple times. These are called multi-pass compilers.
Some languages are designed to support single-pass compilers. (e.g. C, C++).
Some languages require multiple passes. (e.g. Java, Decaf).
Most modern compilers use a huge number of passes over the input.
Scoping in Multi-Pass Compilers

- Completely parse the input file into an abstract syntax tree (first pass).
Scoping in Multi-Pass Compilers

- Completely parse the input file into an abstract syntax tree (first pass).
- Walk the AST, gathering information about classes (second pass).
Scoping in Multi-Pass Compilers

- Completely parse the input file into an abstract syntax tree (first pass).
- Walk the AST, gathering information about classes (second pass).
- Walk the AST checking other properties (third pass).
Scoping in Multi-Pass Compilers

- Completely parse the input file into an abstract syntax tree (first pass).
- Walk the AST, gathering information about classes (second pass).
- Walk the AST checking other properties (third pass).
- Could combine some of these, though they are logically distinct.
Static and Dynamic Scoping

- The scoping we have seen so far is called **static scoping** and is done at compile-time.
- Some languages use **dynamic scoping**, which is done at runtime.
int x = 137;
int y = 42;
void Function1() {
    Print(x + y);
}
void Function2() {
    int x = 0;
    Function1();
}
void Function3() {
    int y = 0;
    Function2();
}
Function1();
Function2();
Function3();
Dynamic Scoping in Practice

- Examples: Perl, Common LISP.
- Often implemented by preserving symbol table at runtime.
- Often less efficient than static scoping.
- Compiler cannot “hardcode” locations of variables.
- Names must be resolved at runtime.