



CSCI 742 - Compiler Construction

Lecture 19
Introduction to Name Analysis
Instructor: Hossein Hojjat

February 26, 2018

Compiler Phases

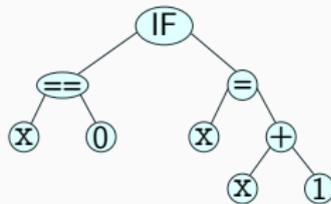
Source Code
(concrete syntax)

```
if (x==0) x=x+1;
```

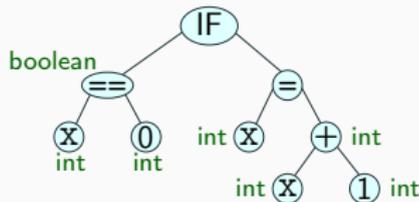
Token Stream

```
if ( x == 0 ) x = x + 1 ;
```

Abstract Syntax Tree
(AST)



Attributed AST



Machine Code

```
16: iload_2  
17: ifne 24  
20: iload_2  
21: iconst_1  
22: iadd  
23: istore_2  
24: ...
```

Lexical Analysis

Syntax Analysis
(Parsing)

Semantic Analysis
(Name Analysis,
Type Analysis, ...)

Code Generation

Error

- Task of a parser:
 - find a derivation of a string in a context-free grammar
- CYK recognizes languages defined by context-free grammars
 - Standard version operates only on Chomsky Normal Form (CNF)
 - cubic time $O(n^3)$
- Restricted forms of CFG can be parsed in linear time:
 - LL (left to right, left-most derivation)
 - LR (left to right, reverse right-most derivation)
- Simple top-down parser: LL(1)
 - Basic recursive-descent implementation
- More powerful parser: LR(1), bottom-up
- An efficiency hack on top of LR(1): LALR(1)

What to expect next?

- Is “x” an array, integer or a function? Is it declared?
- Is the expression “x + z” type-consistent?
- In “x[i]”, is “x” an array? Does it have the correct number of dimensions?
- Where can “x” be stored? (register, local, global, heap, static)
- How many arguments does “f()” take? What about “printf ()” ?

Error detection at different phases

- File input: file does not exist
- Lexer: unknown token, string not closed before end of file, ...
- Parser: syntax error - unexpected token, cannot parse given input string, ...
- Name analyzer: unknown identifier, ...
- Type analyzer: applying function to arguments of wrong type, ...
- Data-flow analyzer: division by zero, loop runs forever, ...

Error detection at different phases

- File input: file does not exist
- Lexer: unknown token, string not closed before end of file, ...
- Parser: syntax error - unexpected token, cannot parse given input string, ...
- **Name analyzer: unknown identifier, ...**
- Type analyzer: applying function to arguments of wrong type, ...
- Data-flow analyzer: division by zero, loop runs forever, ...

Problems detected by Name Analysis

- a class is defined more than once:

```
class A {...} class B {...} class A {...}
```

- a variable is defined more than once:

```
int x; int y; int x;
```

- a field member is overridden (forbidden in eMiniJava)

```
class A {int x; ...}  
class B extends A {int x; ...}
```

- a method is overloaded (forbidden in eMiniJava)

```
class A { void f(B x) {} void f(C x) {} ... }
```

- a method argument is shadowed by a local variable declaration (forbidden in Java)

```
void f (int x) { int x; ...}
```

- two method arguments have the same name (forbidden in many languages)

```
void f (int x, int y, int x) { ... }
```

Problems detected by Name Analysis

- a class name is used as a symbol (as parent class or type, for instance) but is not declared:

```
class A extends Undeclared {}
```

- an identifier is used as a variable but is not declared:

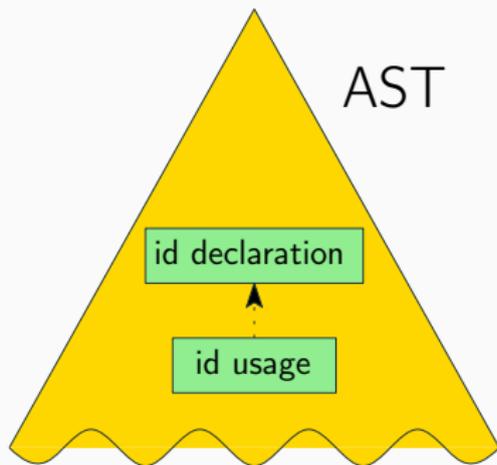
```
int inc (int x, int amount)  
    {return x + ammount; }
```

- the inheritance graph has a cycle:

```
class A extends B {}  
class B extends C {}  
class C extends A
```

Identifier Definition

- **Property:** “Each identifier needs to be declared before usage”
- To check such a property we need “context” information:
the environment where a command executes in
- In **theory** we can use context-sensitive grammars to specify this
- In **practice** we use context-free grammars to specify valid syntax and identify invalid programs using other mechanisms
 - Those mechanisms enforce language properties that cannot be expressed with a CFG
- In order to check the property, we need to find the declaration of each usage of an identifier



```
bool b;  
⋮  
if (b) x = x + 1;
```

- **Name Analysis:** making sense of trees; converting them into graphs: connect identifier uses and declarations

Identifier Mapping

- To make name analysis efficient and clean, we associate mapping from each identifier to the symbol that the identifier represents
- We use Map data structures to maintain this mapping
- The rules that specify how declarations are used to construct such maps are given by “scoping” rules of the programming language

Showing Good Errors with Syntax Trees

- Suppose we have undeclared variable “x” in a program of 100K lines
- Which error message would you prefer to see from the compiler?
 - An occurrence of variable “x” not declared (which variable? where?)
 - An occurrence of variable “x” in procedure P not declared
 - Variable “x” undeclared at line 612, column 21
(and IDE points you there) ✓

Showing Good Errors with Syntax Trees

- How to emit those good error messages if we only have a syntax trees?
- Abstract syntax tree nodes store positions within file
- For identifier nodes: allows reporting variable uses
- Variable “x” in line 612, column 21 undeclared
- For other nodes, supports useful for type errors, e.g. could report for `(x + y) * (!b)`
 - Type error in line 13,
 - expression in line 13, column 11-14, has type `bool`, expected `int` instead

Showing Good Errors with Syntax Trees

Constructing trees with positions:

- Lexer records positions for tokens
- Each subtree in AST corresponds to some parse tree, so it has first and last token
- Get positions from those tokens
- Save these positions in the constructed tree

It is important to save information for leaves

- Information for other nodes can often be approximated using information in the leaves

- **Scope:** The region where an identifier is visible is referred to as the scope of the identifier
- Here identifier refers to function or variable name
- It is only legal to refer to the identifier within its scope
- **Static** property: compiler decides the issue at compile time
- **Dynamic** property: an issue that requires a decision at run-time
- We will study static and dynamic scoping

Scopes

```
class Example {
    boolean x;
    int y;
    int z;
    int compute(int y, int z) {
        int x = 3;
        return x + y + z;
    }
    public void main() {
        int res;
        x = true;
        int y = 10;
        z = 5;
        res = compute(z-1, z+1);
        System.out.println(res);
    }
}
```

- Draw an arrow from occurrence of each identifier to the point of its declaration

Scopes

```
class Example {  
    boolean x;  
    int y;  
    int z;  
    int compute(int y, int z) {  
        int x = 3;  
        return x + y + z;  
    }  
    public void main() {  
        int res;  
        x = true;  
        int y = 10;  
        z = 5;  
        res = compute(z-1, z+1);  
        System.out.println(res);  
    }  
}
```

- Draw an arrow from occurrence of each identifier to the point of its declaration
- Name analysis:
Computes those arrows

Name Analysis Implementation

- For each declaration of identifier, identify where the identifier refers to
- Name analysis:
 - maps, partial functions (math)
 - environments (PL theory)
 - symbol table (implementation)
- Report some simple semantic errors
- We usually introduce symbols for things denoted by identifiers
- Symbol tables map identifiers to symbols

Static Scoping

```
class World {
    int sum;
    int value;
    void add() {
        sum = sum + value;
        value = 0;
    }
    void main() {
        sum = 0;
        value = 10;
        add();
        if (sum % 3 == 1) {
            int value;
            value = 1;
            add();
            println("inner value = " + value);
            println("sum = " + sum);
        }
        println("outer value = " + value);
    }
}
```

- Static Scoping:
Identifier refers to the symbol that was declared “closest” to the place in program structure (thus “static”)
- We will assume static scoping unless otherwise specified

Static Scoping

```
class World {
  int sum;
  int value;
  void add() {
    sum = sum + value;
    value = 0;
  }
  void main() {
    sum = 0;
    value = 10;
    add();
    if (sum % 3 == 1) {
      int value;
      value = 1;
      add();
      println("inner value = " + value); 1
      println("sum = " + sum); 10
    }
    println("outer value = " + value); 0
  }
}
```

- Static Scoping:
Identifier refers to the symbol that was declared “closest” to the place in program structure (thus “static”)
- We will assume static scoping unless otherwise specified

Static Scoping

```
class World {
  int sum;
  int value;
  void add() {
    sum = sum + value;
    value = 0;
  }
  void main() {
    sum = 0;
    value = 10;
    add();
    if (sum % 3 == 1) {
      int value1;
      value1 = 1;
      add(); // cannot change value1
      println("inner value = " + value1); 1
      println("sum = " + sum); 10
    }
    println("outer value = " + value); 0
  }
}
```

- Static Scoping: Identifier refers to the symbol that was declared “closest” to the place in program structure (thus “static”)
- We will assume static scoping unless otherwise specified
- Property of static scoping: Given the entire program, we can rename variables to avoid any shadowing (make all vars unique)

Dynamic Scoping

```
class World {
  int sum;
  int value;
  void add() {
    sum = sum + value;
    value = 0;
  }
  void main() {
    sum = 0;
    value = 10;
    add();
    if (sum % 3 == 1) {
      int value;
      value = 1;
      add();
      println("inner value = " + value); 0
      println("sum = " + sum); 11
    }
    println("outer value = " + value); 0
  }
}
```

- Symbol refers to the variable that was most recently declared within program execution
- Views variable declarations as executable statements that establish which symbol is considered to be the “current one”
 - Used in old LISP interpreters
- Translation to normal code: access through a dynamic environment

Dynamic vs. Static Scoping

- Dynamic Scoping Implementation:
 - Each time a function is called its local variables are pushed on a stack
 - When a reference to a variable is made, the stack is searched top-down for the variable name
- Static scoping is almost universally accepted in modern programming language design
- It is usually easier to reason about and easier to compile
- Static scoping makes reasoning about modular codes easier: binding structure can be understood in isolation

Exercise

Determine the output of the following program assuming static and dynamic scoping. Explain the difference, if there is any.

```
class MyClass{
    int x = 5;
    public int foo(int z) {
        return x + z;
    }
    public int bar(int y) {
        int x = 1;
        int z = 2;
        return foo(y);
    }
    public void main(){
        int x = 7;
        println(foo(bar(3)));
    }
}
```