CMPSC 160
Translation of Programming Languages

Lecture 11: Name Analysis

Semantics Analysis: **Name analysis**

To generate code, a compiler needs to answer many questions:

- Type analysis
  - is x a scalar, an array or a function?
  - is the expression x*y+z type-consistent?
  - in an array reference a[i , j , k], does a have three dimensions?
  - how many arguments does a function take?

- **Name analysis**
  - is x declared? Are there names declared but not used?
  - which declaration of x does each use reference?
    - bind each identifier to the appropriate declaration
- . . .

$Block_0$ → $Block_1$ Assign
| $Block_1$ Decl
| Assign
| Decl
Decl → *Type L*
L → L_1 *Identifier*
→

Assign → *Identifier = Expr ;*
$Expr_0$ → $Expr_1$ + *Term*
| $Expr_1$ – *Term*
| *Term*
$Term_0$ → $Term_1$ * *Factor*
| $Term_1$ / *Factor*
| *Factor*
Type → *int*
boolean
Factor → *(Expr)*
| Number
| Identifier

Ad-hoc Syntax-Directed Translation for just **declaration check**?

<table>
<tr><td>$Block_0$</td><td>→</td><td>$Block_1$ Assign</td><td></td></tr>
<tr><td></td><td>|</td><td>$Block_1$ Decl</td><td></td></tr>
<tr><td></td><td>|</td><td>Assign</td><td></td></tr>
<tr><td></td><td>|</td><td>Decl</td><td></td></tr>
<tr><td>Decl</td><td>→</td><td>Type L</td><td></td></tr>
<tr><td>L</td><td>→</td><td>L_1 Identifier</td><td>i ← hash(Identifier);
Table[i].declared ← true;</td></tr>
<tr><td>Assign</td><td>→</td><td>Identifier = Expr ;</td><td>i ← hash(Identifier);
if (Table[i].declared = false)
then ReportError(“No Declaration”);</td></tr>
<tr><td>$Expr_0$</td><td>→</td><td>$Expr_1$ + Term</td><td></td></tr>
<tr><td></td><td>|</td><td>$Expr_1$ – Term</td><td></td></tr>
<tr><td></td><td>|</td><td>Term</td><td></td></tr>
<tr><td>$Term_0$</td><td>→</td><td>$Term_1$ * Factor</td><td></td></tr>
<tr><td></td><td>|</td><td>$Term_1$ / Factor</td><td></td></tr>
<tr><td></td><td>|</td><td>Factor</td><td></td></tr>
<tr><td>Type</td><td>→</td><td>int</td><td></td></tr>
<tr><td></td><td></td><td>boolean</td><td></td></tr>
<tr><td>Factor</td><td>→</td><td>(Expr)</td><td></td></tr>
<tr><td></td><td>|</td><td>Number</td><td></td></tr>
<tr><td></td><td>|</td><td>Identifier</td><td>i ← hash(Identifier);
if (Table[i].declared = false)
then ReportError(“No Declaration”);</td></tr>
</table>

Key: A global repository, a **symbol table (**insertion and lookup**).**

$Block_0$	→	$Block_1$ Assign	
	|	$Block_1$ Decl	
	|	Assign	
	|	Decl	
Decl	→	*Type L*	
L	→	*L_1 Identifier*	*i← hash*(Identifier)*;* *Table[i].declared ← true;*
Assign	→	*Identifier = Expr ;*	*i← hash*(Identifier)*;* *if (Table[i].declared* = false*)* *then ReportError("No Declaration");* *else Table[i].value = Expr.value*
$Expr_0$	→	*$Expr_1$ + Term*	
	|	*$Expr_1$ – Term*	
	|	*Term*	
$Term_0$	→	*$Term_1$ * Factor*	
	|	*$Term_1$ / Factor*	
	|	*Factor*	
Type	→	*int*	
		boolean	
Factor	→	*(Expr)*	
	|	Number	
	|	Identifier	*i← hash*(Identifier)*;* *if (Table[i].declared* = false*)* *then ReportError("No Declaration");* *else Factor.value = Table[i].value*

What is we also want to get the right **value** for each inference if the identifier has been declared previously?

Values can be replaced with **addresses** if we are focusing on the allocation information or code generation

Use of symbol tables for name analysis

Simplistic 1-pass compiler:

- no tree is built — the name analysis is performed during parsing
- the meaning of a declaration is represented by a small data structure "symbol". For a variable it typically contains type and allocation info. This information is stored in the symbol table
- the name analysis is tangled with other compilation aspects, e.g., type analysis, allocation information, code generation

But **complex cases** may require complex declaration structures and several passes…

Production			Semantic action
$Block_0$	→	Block$_1$ Assign	
	|	Block$_1$ Decl	
	|	Assign	
	|	Decl	
Decl	→	*Type L*	
L	→	*L_1 Identifier*	*i← hash(*Identifier*);* *Table[i].declared ← true;*
Assign	→	*Identifier = Expr ;*	*i← hash(*Identifier*);* *if (Table[i].declared* = false*)* *then ReportError("No Declaration");* *else Table[i].value = Expr.value*
$Expr_0$	→	*$Expr_1$ + Term*	
	|	*$Expr_1$ – Term*	
	|	*Term*	
$Term_0$	→	*$Term_1$ * Factor*	
	|	*$Term_1$ / Factor*	
	|	*Factor*	
Type	→	*int*	
		boolean	
Factor	→	*(Expr)*	
	|	Number	
	|	Identifier	*i← hash(*Identifier*);* *if (Table[i].declared* = false*)* *then ReportError("No Declaration");* *else Factor.value = Table[i].value*

- What are missing here?
- Hint: PLs are much more complicated!!!

Fused Declaration and Definition

```
int z;                              // just a declaration
double square(double);              // just a declaration

void f() {                          // declaration plus definition together
  int x = 3;                        // also a declaration plus definition
  int y = x + z /square(3);         // so is this
}

double square(double w) {           // definition completing earlier declaration
  return w * w;
}
```

Relatively easy to fix!!!

Same Names in Programs

- Multiple variables could have the same names in a program.

```
int a = 4;
{
    int a = 3;
    int b = 3;
    print(a, b);
}
print(a);
```

```
int add(int x, int y){return x + y;}
int subtract(int x, int y){return x - y;}
```

- Multiple functions could have the same names in a program.
  - e.g., function overloading in C++

```
void print(int i) {cout << " Here is int " << i << endl; }
void print(double  f) { cout << " Here is float " << f << endl;}
void print(char const *c) {cout << " Here is char* " << c << endl;}
```

Why Same Names?

- Easy for programming
- Performance.

Name Space (Scope)

- The scope of a declaration: The part of a program where the name of a declaration is visible
  - This means it is only legal to refer to the identifier within its scope. Here identifier refers to function or variable name.

```
int a = 4;
{
    int a = 3;
    int b = 3;
    print(a, b);
}
print(a, b);
```

Any error?

- Usage of Name space (scope):
  - function with local variables.
  - easy for naming: Local names can hide identical, non-local names.
  - local names cannot be seen outside – easy memory management.

Nested Block and Inheritance

- Block:
  - a syntactic unit with declarations and statements
  - may require memory allocation during execution (once or several times)
- Block structure (nesting):
  - a block can have inner blocks (recursively)
  - declarations in a block are visible also in the inner blocks

```
int i ;
void main (int j) {
    int k ;
    {
        int l ;
        int k ;
    }
    {
        int l ;
        int m ;
    }
}
```

```
int a = 4;
{
  int b = 3;
  print(a, b);
}
```

```
int a = 4;
{
  int a = 3;
  int b = 3;
  print(a, b);
}
```

- Shadowing occurs when an identifier declared within a given scope has the same name as an identifier declared in an outer scope.

Nested Block and Inheritance

- Inheritance: declarations in a class are visible also in its subclasses

```
class A { int a; … }
class B extends A {
      print(a);
      int a = 0;
}
```

- Combined block structure and inheritance
  - e.g., a method in a subclass can access instance variables in a superclass

```
class A { int a; … }
class B extends A {
    void m() {
     print(a);
     int a = 0;
     }
}
```

```
class A {
       void m() {}
}
class B extends A {
        void m2() { m(); }
        void m() {}
}
```

Scope rules (visibility rules)

- What are the binding rules?
  - Govern how identifier references are bound to identifier declaration.

- Typical factors (differ in different languages)
  - **Name collisions:** what happens if the same name is declared in many blocks?
  - **Combination:** how can blocks be combined? what happens if a name is not declared in local blocks?
  - **Declaration order:** does it affect the bindings?
  - **Method overloading:** can there be several methods of the same name, but with different argument types?
  - **Parameters:** how do they relate to local variables?
  - **Return values:** are they named explicitly?
  - **Visibility restrictions:** private, public, . . . qualified access access via another name
  - ….

Question Time ☺

```c
// A C program to demonstrate static scoping.
#include<stdio.h>
int x = 10;

// Called by g()
int f()
{
   return x;
}

// g() has its own variable
// named as x and calls f()
int g()
{
   int x = 20;
   return f();
}

int main()
{
  printf("%d", g());
  printf("\n");
  return 0;
}
```

Guess what would be the output?

Lexical Scoping vs. Dynamic Scoping

- Lexical Scoping: Scoping is determined by the program text (static)
  - The scope of a declaration in a block *B* includes *B*
  - If a variable *x* is not declared in *B*, then occurrence of *x* in *B* is in the scope of the declaration of *x* in **enclosing block** *B*ʹ if
    - *B*ʹ has a declaration of *x*
    - *B*ʹ is more **closely** nested around *B* then any other block with declaration of *x*
  - Lexical scoping is used in languages such as Pascal, C

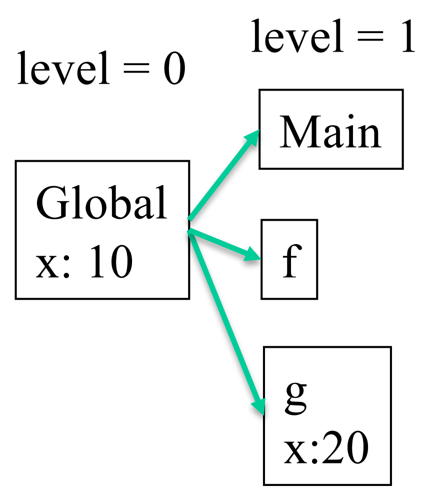
- Dynamic Scoping: Scoping is determined by the run-time behavior
  - A variable that is not declared in the current scope is bound to the variable by that name that was most recently created at *run-time*
  - Dynamic scoping is used in some forms of LISP

Example: Lexical vs. Dynamic Scoping

```c
// A C program to demonstrate static scoping.
#include<stdio.h>
int x = 10;

// Called by g()
int f()
{
   return x;
}

// g() has its own variable
// named as x and calls f()
int g()
{
   int x = 20;
   return f();
}

int main()
{
  printf("%d", g());
  printf("\n");
  return 0;
}
```

With lexical scoping, the output is: 10
If dynamic scoping was used, then the output will be: 20

The execution result you get is 10 for c program.



Example: Lexical vs. Dynamic Scoping

```
program dynamic(input, output)
  var r : real;
  procedure show;
    begin write( r ) end;
  procedure small;
    var r : real;
    begin r := 0.125; show end;
  begin
    r := 0.25
    show; small; writeln;
    show; small; writeln
  end.
```

With lexical scoping, the output is:

```
0.250 0.250
0.250 0.250
```

If dynamic scoping was used, then the output will be:

```
0.250 0.125
0.250 0.125
```

Lexically-Scoped Symbol Tables

The problem

- The compiler needs a distinct record for each declaration
- Nested lexical scopes allow for duplicate declarations

Solution: store the scope information in the symbol table

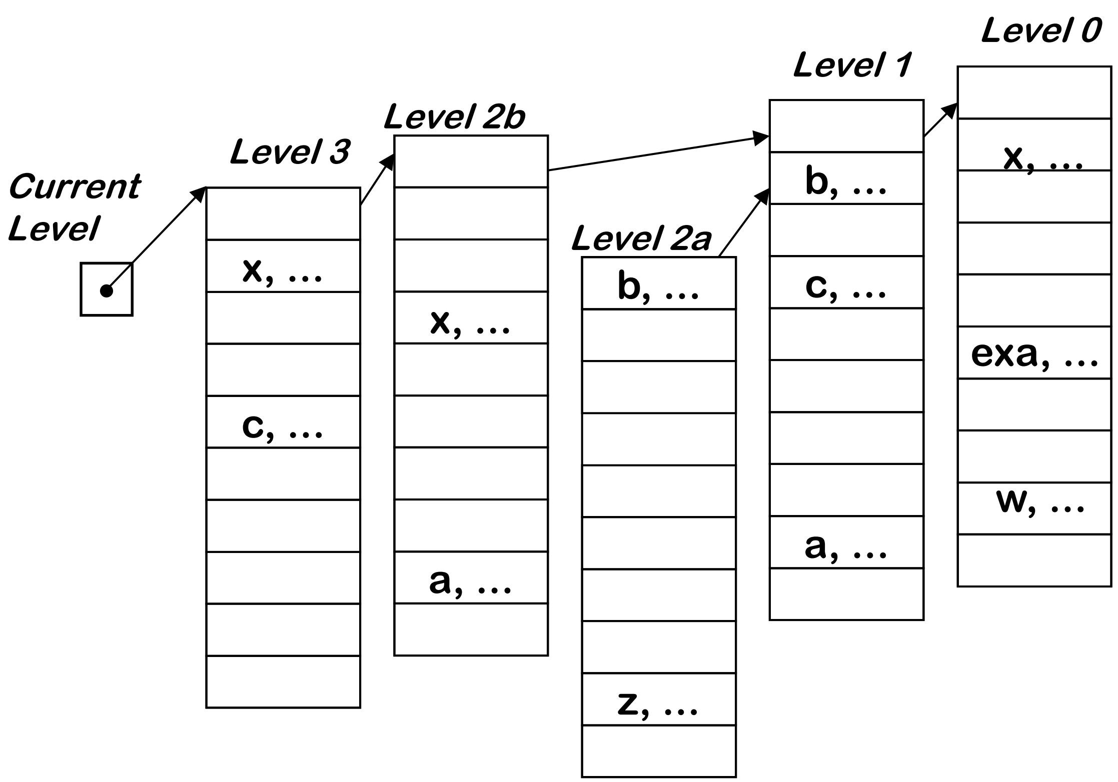
The interface

- Insert(*name,level*) – creates record for *name* at *level*
- Lookup(*name,level*) – returns pointer or index
- OpenScope() – increments the current level and creates a new symbol table for that level
- CloseScope() – changes current level pointer so that it points at the table for the scope surrounding the current level, and then decrements the current level

Lexically-Scoped Symbol Tables

High-level idea

- Create a new table for each scope
- Chain them together for lookup

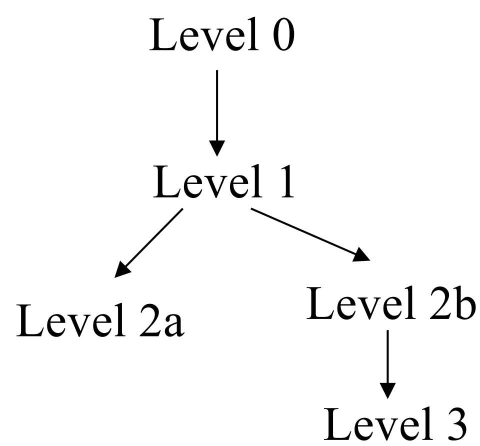


- *Insert*() inserts at the current level
- *LookUp*() walks chain of tables and returns first occurrence of name
- *OpenScope()* creates a new table, connects it to the current level and updates the current level to point to the new table
- *CloseScope()* removes the table which is the top table in the chain

Example in C

```c
int w;                        /* level 0 */
int x;
void example (int a, int b); /* level 1 */
{
  int c;
  {
    int b, z;                /* level 2a */
      ...
  }
  {
    int a, x;               /* level 2b */
      ...
      {
        int c, x;            /* level 3 */
        b = a + b + c + x;
      }
  }
}
```

Level	Names
0	w, x, example
1	a, b, c
2a	b,z
2b	a, x
3	c, x



Nesting relationships

At any scope level, a variable is bound to the declaration that is in that scope level. If there is no declaration at that scope level, then it is bound to the declaration that is in its closest ancestor

Example in C

```
int w;                          /* level 0 */
int x;
void example (int a, int b); /* level 1 */
{
  int c;
  {
    int b, z;                  /* level 2a */
      ...
  }
  {
    int a, x;                  /* level 2b */
      ...
      {
        int c, x;              /* level 3 */
        b = a + b + c + x;
      }
  }
}
```

Generated sequence of calls:

Insert(w)
Insert(x)
Insert(example)
OpenScope()
Insert(a)
Insert(b)
Insert(c)
OpenScope()
Insert(b)
Insert(z)
CloseScope()
OpenScope()
Insert(a)
Insert(x)
OpenScope()
Insert(c)
Insert(x)
Lookup(b)
LookUp(a)
LookUp(b)
LookUp(c)
LookUp(x)
CloseScope()
CloseScope()
CloseScope()

$Block_0$	→	$Block_1$ Assign	
	|	$Block_1$ Decl	
	|	Assign	
	|	Decl	
Decl	→	*Type L*	
L	→	*L_1 Identifier*	*i← hash*(Identifier); *Table[i].declared ← true;*
Assign	→	*Identifier = Expr ;*	*i← hash*(Identifier); *if (Table[i].declared* = false) *then ReportError("No Declaration");*
$Expr_0$	→	*$Expr_1$ + Term*	
	|	*$Expr_1$ – Term*	
	|	*Term*	
$Term_0$	→	*$Term_1$ * Factor*	
	|	*$Term_1$ / Factor*	
	|	*Factor*	
Type	→	*int*	
		boolean	
Factor	→	*(Expr)*	
	|	Number	
	|	Identifier	*i← hash*(Identifier); *if (Table[i].declared* = false) *then ReportError("No Declaration");*

How to change our CFG and semantics rules for block level scoping?

$Block_0$	→	{ $Block_1$ }	*OpenScope() +CloseScope()*
	|	$Block_1$ Assign	
	|	$Block_1$ Decl	
	|	Assign	
	|	Decl	
Decl	→	*Type L*	*i← hash*(Identifier)*;* *Table[i].declared ← true;*
L_0	→	*L_1 Identifier*	
Assign	→	*Identifier = Expr ;*	
$Expr_0$	→	*$Expr_1$ + Term*	
		....	
Factor	→	*(Expr)*	
	|	Number	
	|	Identifier	*i← hash*(Identifier)*;* *if (Table[i].declared* = false*)* *then ReportError("No Declaration");*

Flexibility in Ad-hoc syntax-directed translation:
A → BCD
A → {action} B{action} C {action} D{action}

For this example, we have
Block0 --> { **openScope()** Block1 } **CloseScope()**

Others

- Additional constraints might exist depending on the specific language.

```
...
void main () {
        i = 3;
}
int i ;
...
```

- invalid in C
- valid in Java

- Parameters: Usually, parameters can be seen as special local variables and it is wrong to declare a local variable with the same name

```
void method1(int x, y){
    int s = 2;
    x = s + y;
    ...
}
```

```
void method1(int x, y){
    int x; // multiple declaration of x
    ...
}
```