CSE 3302
Programming Languages
Lecture 3: Semantics

(based on the slides by Chengkai Li)
Leonidas Fegaras
University of Texas at Arlington
Names

• Names: identify language entities
  – variables, procedures, functions, constants, data types, …
• Attributes: properties of names
• Examples of attributes:
  – A data type
    \[
    \text{int } n = 5; \quad \text{(data type name: int)}
    \]
  – A value
    \[
    \text{(value: 5)}
    \]
  – A location
    \[
    \text{int* } y;
    \]
    \[
    y = \text{new int;}
    \]
  – Function parameters, the type of return value
    \[
    \text{int } f(\text{int } n) \{ \ldots \}\]
Binding

- **Binding**: associating attributes to names
  - declarations
  - assignments
  - declarations (prototype) and definition of a function

- The bindings can be explicit or implicit
  
  e.g. `int x;

  - Explicit binding: the data type of \( x \)
  - Implicit binding: the location of \( x \) (static or dynamic, depending on where \( x \) is stored)
Binding Time

• **Binding Time:** the time when an attribute is bound to a name
  
  – **Static binding** (static attribute):
    
    occurs before execution
    
    • *Language definition/implementation time:* The range of the data type `int`
    • *translation time (parsing/semantic analysis):* The data type of a variable
    • *link time:* The code of an external function
    • *load time:* The location of a global variable
  
  – **Dynamic binding** (dynamic attribute):
    
    occurs during execution
    
    • *Between the entry and exit points of a procedure or program:* the value of local variables
Where can declarations happen?

- Blocks ({}), begin-end, …
  - Algol descendants: C/C++, Java, Pascal, Ada, ...
    - e.g., C
      - Function body
      - Anywhere a statement can appear (compound statement)

- External/global
- Structured data type
- Class
const int Maximum = 100;
struct FullName {string Lastname, string FirstName};

class Student {
private:
    struct FullName name; int Age;
public:
    void setValue(const int a, struct FullName name);
    int TStudent();
    ...
};

void Student::setAge(const int a, string lName, string fName) {
    int i;
    Age = a;
    {
        int j;
        name.LastName = lName;
        name.FirstName = fName;
    }
}
...

Scope of Binding

- **Scope of Binding**: the region of the program text where the binding applies (is valid)

- **Block-structured language**

  *lexical scope (static scope)*: from the declaration to the end of the block containing the declaration

  *dynamic scope*: introduced later
Example

```c
int x;
void p(void) {
    char y;
    . . .
    { int i;
        . . .
    }
}

void q(void) {
    double z;
    . . .
}

main() {
    int w[10];
    . . .
}
```
void p(void) {
    int x;
    ...  

    char y;
    ...  
}

Exception in OO languages: Scope of local declarations inside a class declaration includes the whole class

public class {
    public int getValue() { return value; }  
    int value;
}
Scope Hole

- **Scope Hole**: Declarations in nested blocks take precedence over the previous declarations. That is, the binding becomes invisible/hidden.

```c
int x;

void p(void) {
    char x;
    x = 'a';
    ...
}

main() {
    x = 2;
    ...
}
```

- `x` (bound to the character data type)
- `x` (bound to the integer data type)
Accessing Hidden Declarations

- Scope resolution operator :: (C++)

```c
int x;

void p(void) {
    char x;
    x = 'a';
    ::x=42;
    ...
}

main() {
    x = 2;
    ...
}
```

- x (bound with character data type)
- x (bound with integer data type)
- the hidden integer variable x
Hide a Declaration

- File 1: extern int x;
  File 2: int x;

- File 1: extern int x;
  File 2: static int x;
Symbol Table

- Symbol Table: maintains bindings
  - Can be viewed as functions that map names to their attributes
Static vs. Dynamic Scope

- Static scope (lexical scope):
  - the scope is maintained statically (during compilation)
  - it follows the layout of the source code
  - used in most languages
- Dynamic scope:
  - the scope is maintained dynamically (during execution)
  - it follows the execution path
  - few languages use it
    - The bindings cannot be determined statically
    - The result may depend on the user input
      - Lisp: considered a bug by its inventor
      - Perl: can choose lexical or dynamic scope
Static Scope

int x = 1;
char y = 'a';

void p(void) {
    double x = 2.5;
    printf("%c\n", y);
}

void q(void) {
    int y = 42;
    printf("%d\n", x);
    p();
}

main() {
    char x = 'b';
    q();
}
Static Scope

int x = 1;
char y = ‘a’;

void p(void) {
    double x=2.5;
    printf(“%c\n”,y);
}

void q(void) {
    int y = 42;
    printf(“%d\n”,x);
    p();
}

main() {
    char x = ‘b’;
    q();
}

The symbol table in p: the bindings available in p

- double, local to p
- integer, global
- character, global
The symbol table in `q`:
the bindings available in `q`

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X</code></td>
<td>integer, global</td>
</tr>
<tr>
<td><code>Y</code></td>
<td>integer, local to <code>q</code></td>
</tr>
<tr>
<td></td>
<td>character, global</td>
</tr>
</tbody>
</table>

```c
int x = 1;
char y = 'a';

void p(void) {
    double x=2.5;
    printf("%c\n",y);
}

void q(void) {
    int y = 42;
    printf("%d\n",x);
    p();
}

main() {
    char x = 'b';
    q();
}
```
int x = 1;
char y = 'a';

void p(void) {
	double x = 2.5;
	printf("%c\n", y);
}

void q(void) {
	int y = 42;
	printf("%d\n", x);
	p();
}

main() {
	char x = 'b';
	q();
}
Static Scope

- The symbol table is built during compilation
- The bindings are used in type-checking and in generating the machine code
- Result:

  1
  a

- E.g., semantics of `q`

```c
void q(void) {
    int y = 42;
    printf("%d\n", x);
    p();
}
```

The symbol table in `q`:
- the bindings available in `q`

<table>
<thead>
<tr>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer, global</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer, local to <code>q</code></td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>character, global</td>
</tr>
</tbody>
</table>
Practice for Static Scope

```c
int x, y;

void g(void) {
    x = x + 1;
    y = x + 1;
}

void f(void) {
    int x;
    y = y + 1;
    x = y + 1;
    g();
}

main() {
    x = 1;
    y = 2;
    f();
    g();
    printf("x=%d,y=%d\n",x,y);
}
```

Exercise:

Draw the symbol table at the given points in the program using static scope

Question:

What does the program print using static scope?
What if dynamic scope is used?

```c
int x = 1;
char y = 'a';

void p(void) {
    double x=2.5;
    printf("%c\n",y);
}

void q(void) {
    int y = 42;
    printf("%d\n",x);
    p();
}

main() {
    char x = 'b';
    q();
}
```
What if dynamic scope is used?

```c
int x = 1;
char y = 'a';

void p(void) {
    double x=2.5;
    printf("%c\n",y);
}

void q(void) {
    int y = 42;
    printf("%d\n",x);
    p();
}

main() {
    char x = 'b';
    q();
}
```

The symbol table in `main`:
the bindings available in `main`

<table>
<thead>
<tr>
<th>X</th>
<th>character, ‘b’, local to <code>main</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>integer, 1, global</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y</th>
<th>character, ‘a’, global</th>
</tr>
</thead>
</table>
What if dynamic scope is used?

```c
int x = 1;
char y = 'a';

void p(void) {
    double x=2.5;
    printf("%c\n",y);
}

void q(void) {
    int y = 42;
    printf("%d\n",x);
    p();
}

main() {
    char x = 'b';
    q();
}
```

The symbol table in `q`:
the bindings available in `q`

- `character, ‘b’, local to main`
- `integer, 1, global`
- `integer, 42, local to q`
- `character, ‘a’, global`
What if dynamic scope is used?

```c
int x = 1;
char y = 'a';

void p(void) {
    double x = 2.5;
    printf("%c\n", y);
}

void q(void) {
    int y = 42;
    printf("%d\n", x);
    p();
}

main() {
    char x = 'b';
    q();
}
```

The symbol table in `p`:
the bindings available in `p`

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code></td>
<td>double</td>
<td>local to <code>p</code></td>
</tr>
<tr>
<td><code>y</code></td>
<td>character</td>
<td>local to <code>main</code></td>
</tr>
<tr>
<td><code>x</code></td>
<td>integer</td>
<td>global</td>
</tr>
<tr>
<td><code>y</code></td>
<td>integer</td>
<td>local to <code>q</code></td>
</tr>
<tr>
<td><code>x</code></td>
<td>character</td>
<td>global</td>
</tr>
</tbody>
</table>

```
Practice for Dynamic Scope

```c
int x, y;

void g(void) {
    x = x + 1;
    y = x + 1;
}

void f(void) {
    int x;
    y = y + 1;
    x = y + 1;
    g();
}

main() {
    x = 1;
    y = 2;
    f();
    g();
    printf("x=%d,y=%d\n", x, y);
}
```

Exercise:
Draw the symbol table at the given points in the program using dynamic scope

Question:
What does the program print using dynamic scope?
Overloading

• What is overloading?
  – Using the same name for more than one entities

• Why overloading?
  – Convenience
  – Optional function parameters
  – Polymorphism
    • eg. we want + to work on both integers and reals

• What can be overloaded?
Overload Resolution

- Overload Resolution
  - select one entity among all qualified

- Name isn’t sufficient in resolution
  - need extra information (often data types)
Function/Method Overloading

- **C**: no overloading
- **C++/Java/Ada**: resolution by considering the number and types of parameters
  - Perfect if exact match exists
  - No perfect match: different conversion rules
    - Ada: automatic conversions not allowed
    - Java: conversions allowed in certain cases
    - C++: automatic conversions; more flexible
    - e.g.,
      - `int sum(int a, int b) {...}`
      - `double sum(double a, double b) {...}`
      - `double sum(double a, int b) {...}

```
sum(1); sum(1, 2); sum(1.0, 2.0); sum(1, 2.0);
```
### Overload Resolution Example

1. int sum(int, int);
2. double sum(double, int);
3. double sum(double, double);

```c
int x;
double y;
```

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>Java</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = sum(3,4);</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>y = sum(3,4);</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>x = sum(3,4.5);</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>y = sum(3,4.5);</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>x = sum(3.5,4);</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>y = sum(3.5,4);</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>x = sum(3.5,4.5)</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>y = sum(3.5,4.5)</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Environment

- **Location**: one specific attribute of names
- **Environment**: maintains bindings from names to locations
- **Static vs. dynamic**
  - FORTRAN: completely static
  - LISP: completely dynamic
  - Algol-descendants (C, C++, Ada, Java): combination
    - global variables: static
    - local variables: dynamic
Stack-Based Allocations

- **static (global) area**
  - **stack**
  - (unallocated)
  - **heap**

- Automatically-allocated spaces (local variables, procedures) under the control of the runtime system
- Manually-allocated spaces under the control of the programmer
Example

A: { int x;
    char y;
    B: { double x;
        int a;
    }
    C: { char y;
        int b;
        D: { int x;
            double y;
        }
    }
}
Example

A: { int x;
    char y;
 B: { double x;
       int a;
    }  
 C: { char y;
       int b;
 D: { int x;
       double y;
    }  
 }  
 }  

Lifetime is the duration of B
Example

A: { int x;
    char y;
}
B: { double x;
    int a;
}
C: { char y;
    int b;
}
D: { int x;
    double y;
}

Example

A: {  int   x;
    char  y;
  }
B: {  double  x;
    int     a;
  }
C: {  char    y;
    int     b;
  }
D: {  int     x;
    double  y;
  }

\[\begin{array}{c}
x \\
y \\
y \\
b \\
x \\
y \\
\end{array}\]
Example

A: { int x;
    char y;
    B: { double x;
        int a;
    }
    C: { char y;
        int b;
        D: { int x;
            double y;
        }
    }
}
Heap-Based Allocation

- **C**
  ```c
  int *x;
  x = (int *)malloc(sizeof(int));
  free(x);
  ```

- **C++**
  ```cpp
  int *x;
  x = new int;
  delete x;
  ```

- **Java**
  ```java
  Integer x = new Integer(2);
  // no deletion
  // needs garbage collection
  ```
Scope vs. Lifetime

- **Lifetime beyond scope:**
  - alive in scope hole
  - alive outside scope

- **Scope beyond lifetime (unsafe)**
Example: Alive in scope hole

A: {  int  x;
    char  y;
    B: {  double  x;
          int   a;
        }
    C: {  char    y;
          int   b;
    D: {  int     x;
          double  y;
        }
      }
  }

\[\begin{array}{c|c|c|c}
  \hline
  & \text{x} & \text{y} & \text{a} \\
  \hline
  \text{x} & & \text{x} & \\
  \hline
  \text{y} & \text{x} & & \\
  \hline
  \text{x} & & \text{a} & \\
  \hline
\end{array}\]
Example: Alive outside scope

```c
int func(void) {
    static int counter = 0;
    counter += 1;
    return counter;
}

main()
{
    int i;
    int x;
    for (i=0; i<10; i++)
        x=func();
    printf("%d\n", x);
}
```
Example: Scope beyond lifetime

Dangling pointer:

```c
int *x, *y, *z;

x=(int *) malloc(sizeof(int));
*x=2;
y=x;
free(x);

... 

printf("%d\n",*y);
```
Box-and-Circle Diagram for Variables

Name

Value

Location

l-value

r-value

x = y

assignment

x

y

x = y

assignment
Assignment by Sharing

Java:

Student x = new Student("Amy");
Student y = new Student("John");
x.setAge(19);
x = y;
y.setAge(21);
Assignment by Cloning

\[ x = y \]
(1) int *x, *y;
(2) x = (int *)malloc(sizeof(int));
(3) *x = 1;
(4) y = x;
(5) *y = 2;
(6) printf(“%d\n”, *x);

After line 1:
(1) int *x, *y;
(2) x = (int *)malloc(sizeof(int));
(3) *x = 1;
(4) y = x;
(5) *y = 2;
(6) printf("%d\n",*x);

After line 2:
(1) int *x, *y;
(2) x = (int *)malloc(sizeof(int));
(3) *x = 1;
(4) y = x;
(5) *y = 2;
(6) printf("%d\n",*x);

After line 3:
(1) int *x, *y;
(2) x = (int *)malloc(sizeof(int));
(3) *x = 1;
(4) y = x;
(5) *y = 2;
(6) printf("%d\n",*x);

After line 4:
(1) int *x, *y;
(2) x = (int *)malloc(sizeof(int));
(3) *x = 1;
(4) y = x;
(5) *y = 2;
(6) printf("%d\n",*x);

After line 5:
(1) #include <stdio.h>
(2) main()
(3)  int **x;
(4)  int *y;
(5)  int z;
(6)  x = (int**)malloc(sizeof(int*));
(7)  y = (int*)malloc(sizeof(int));
(8)  z = 1;
(9)  *y = 2;
(10) *x = y;
(11) **x = z;
(12) printf("%d\n",*y);
(13) z = 3;
(14) printf("%d\n",*y);
(15) **x = 4;
(16) printf("%d\n",z);
(17) return 0;
(18)}

Exercise:

Draw box-and-circle diagrams of the variables after line 11 and 15

Question 1:
Which variables are aliases at each of those points?

Question 2:
What does the program print?
Dangling References

```c
int *x, *y;
...
x = (int *)malloc(sizeof(int));
...
*x = 2;
...
y = x;
free(x);
/* *y is now a dangling reference */
...
printf("%d\n", *y); /*illegal reference*/
```
Dangling References

```c
{int *x;
 { int y;
  y = 2;
  x = &y;
 }
/* *x is now a dangling reference */
}
```
Dangling References

```c
int* dangle(void)
{
    int x;
    return &x;
}
...

y = dangle();
/* *y is a dangling reference */
```