CS571: Programming Languages

Flex tutorial

- ftp://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_mono/flex.html

Assignment 1
Due: 11:59pm Feb. 16 (Friday)

Assignment 1

- Done by a group of 2 students
- Goal:
  - Learn how to write a C makefile: http://www.delorie.com/djgpp/doc/ug/larger/makefiles.html
  - Acquaint yourself with flex and bison.

Assignment 1

- You will extend **calc.l** and **calc.y** to parse programs whose syntax is defined below.

Prog → main() {Stmts}
Stmts → ε | Stmt; Stmts
Stmt → float Id | Id = E | print Id | {Stmts}
E → Float | Id | E - E | E * E | (-Float)
Float → digit+ . Digit+

Assignment 1

Prog → main() {Stmts}
Stmts → ε | Stmt; Stmts
Stmt → float Id | Id = E | print Id | {Stmts}
E → Float | Id | E - E | E * E | (-Float)
Float → digit+ . Digit+

- **Prog:** a program that contains one **main** function
- **Stmts:** empty or a sequence of statements separated using ;

Assignment 1

Prog → main() {Stmts}
Stmts → ε | Stmt; Stmts
Stmt → float Id | Id = E | print Id | {Stmts}
E → Float | Id | E - E | E * E | (-Float)
Float → digit+ . Digit+

- Id: identifier that starts with a lower-case letter followed by zero or more (lower-case or capital) letters or digits.
- A new variable gets **0.0** as its initial value.

Assignment 1

Prog → main() {Stmts}
Stmts → ε | Stmt; Stmts
Stmt → float Id | Id = E | print Id | {Stmts}
E → Float | Id | E - E | E * E | (-Float)
Float → digit+ . Digit+

- **Expression E**
  - A floating point (positive/negative)
  - An identifier
  - An infix arithmetic expression with "-" and "*" only
  - - and * are left associative.
  - * has higher precedence than -

Assignment 1

Prog → main() {Stmts}
Stmts → ε | Stmt; Stmts
Stmt → float Id | Id = E | print Id | {Stmts}
E → Float | Id | E - E | E * E | (-Float)
Float → digit+ . Digit+

- **Id = E** assigns the value of an expression **E** to **Id**
- **print Id** outputs the value of **Id**.

Assignment 1

Prog → main() {Stmts}
Stmts → ε | Stmt; Stmts
Stmt → float Id | Id = E | print Id | {Stmts}
E → Float | Id | E - E | E * E | (-Float)
Float → digit+ . Digit+

- **{Stmt}:** a block that contains a sequence of statements
  - similar to blocks in C, C++, and Java
  - Local variables in different blocks may have the same name.
  - Blocks may be nested to arbitrary depth

Assignment 1

- Tokens may be separated by any number of white spaces, tabs or new lines.
- If an input does not match any token, output lexical analysis error: <input>, where <input> is the input.
- If there is a parse error, you need to report the error.
  - Your error messages must contain the line number where the error was found.

Compilation and Execution

- Compile your program:
  flex –l calc.l
  bison -dv calc.y
  gcc -o calc calc.tab.c lex.yy.c –lfl
- Execution (example):.
  ./calc < input

  input is the name of the input file

Assignment 1: Example

Program 1:
main() {float x; print x;}
Output:

Program 2:
main() {float x; x = 3.0;}
Output:

Program 3:
main() {float x; x = 3.0; print x;}
Output:

Program 4:
main() {float x; x=4.0; x = x-3.0, print x;}
Output:

Assignment 1: Example

Program 1:
main() {float x; print x;}
Output: 0.0

Program 2:
main() {float x; x = 3.0;}
Output:

Program 3:
main() {float x; x = 3.0; print x;}
Output:

Program 4:
main() {float x; x=4.0; x = x-3.0, print x;}
Output:

Assignment 1: Example

Program 1:
main() {float x; print x;}
Output: 0.0

Program 2:
main() {float x; x = 3.0;}
Output:

Program 3:
main() {float x; x = 3.0; print x;}
Output: 3.0

Program 4:
main() {float x; x=4.0; x = x-3.0, print x;}
Output:

Assignment 1: Example

Program 1:
main() {float x; print x;}
Output: 0.0

Program 2:
main() {float x; x = 3.0;}
Output:

Program 3:
main() {float x; x = 3.0; print x;}
Output: 3.0

Program 4:
main() {float x; x=4.0; x = x-3.0, print x;}
Output: 1.0

Assignment 1: Example

Program 5:
main() {float x; x = 2.0+1.0;}
Output:

Program 6:
float x;
Output:

Program 7:
main() {float 1x;}
Output:

Assignment 1: Example

Program 5:
main() {float x; x = 2.0+1.0;}
Output: Lexical analysis error: +

Program 6:
float x;
Output:

Program 7:
main() {float 1x;}
Output:

Assignment 1: Example

Program 5:

```
main() {float x; x = 2.0+1.0;}
```

Output: Lexical analysis error: +

Program 6:

```
float x;
```

Output: Parsing error: line 1

Program 7:

```
main() {float 1x;}
```

Output:

Assignment 1: Example

Program 5:

```
main() {float x; x = 2.0+1.0;}
```

Output: Lexical analysis error: +

Program 6:

```
float x;
```

Output: Parsing error: line 1

Program 7:

```
main() {float 1x;}
```

Output: Lexical analysis error: 1x

Assignment 1: Example

Program 8:

```
main() {
    float x;
}
```

Output:

Program 9:

```
main() {
        float x;
        x = 3.0;
print   x;
}
```

Output:

Assignment 1: Example

Program 8:

```
main() {
    float x;
}
```

Output:

Program 9:

```
main() {
        float x;
        x = 3.0;
print   x;
}
```

Output: 3.0

Assignment 1: Example

Program 10:

```
main{
        float x;
        x = 1.0;
        print x;
        {
                float x;
                x = 3.0;
                print x;

        };
        print x;
}
```

Output:

Assignment 1: Example

Program 10:

```
main{
        float x;
        x = 1.0;
        print x;
        {
                float x;
                x = 3.0;
                print x;

        };
        print x;
}
```

Output:

1.0
3.0
1.0

Assignment 1: Example

Program 11: **Output:**

```
main(){
        float x;
        float y;
        float z;
        x = 1.0;
        {  float x;
           x = 5.0;
           y = x-3.0;
           print y;
           {  float y;
              y = 6.0-x;
              print y;
           }; };
        z= y-x;
        print z;
}
```


Assignment 1: Example

Program 11: **Output:**

```
main(){
        float x;
        float y;
        float z;
        x = 1.0;
        {  float x;
           x = 5.0;
           y = x-3.0;
           print y;
           {  float y;
              y = 6.0-x;
              print y;
           }; };
        z= y-x;
        print z;
}
```

2.0
1.0
1.0

Assignment 1: Example

Program 12:

```
main() {float x; x = 3.0-(-1.0); print x;}
```

Output:

Assignment 1: Example

Program 12:

```
main() {float x; x = 3.0-(-1.0); print x;}
```

Output: 4.0

Program 13:

```
main() { }
```

Output:

Submission Guideline

- Hand in your source code and a makefile electronically (do not submit .o or executable code).
- Each group uploads only ONE copy of the assignment
- Make sure that this code compiles and runs correctly on bingsuns.binghamton.edu. The makefile must give the executable code the name calc
- Write a README file (text file, do not submit a .doc file) which contains
  - Names and email addresses of group members
  - Whether your code was tested on bingsuns.
  - How to execute your program.
  - (optional) Briefly describe anything special about your submission that the TA should take note of.

Submission Guideline

- Place all your files under one directory with a unique name (p1-[userid] for assignment 1, e.g. p1-pyang).
- Tar the contents of this directory using the command

  tar -cvf p1-[userid].tar p1-[userid]

  E.g. tar -cvf p1-pyang.tar p1-pyang/
- Upload the tared file you created above on mycourses.

Grading Guideline

- Readme (must be a text file), correct executable name (calc): 4'
- Correct makefile (all files are compiled when typing make): 8'
- Correctness of the program: 84'

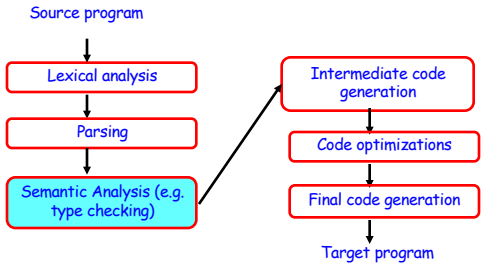
Academy Integrity

- We will use moss to detect to plagiarism in this assignment.

Online Resources

- Unix and C tutorial: http://heather.cs.ucdavis.edu/~matloff/unix.html

Phase of Compilation



Type Checking

- Detect type errors and undeclared variables
  - E.g. an operator is applied to incompatible operands

```
main() { int x; int a[100]; a = x; }
> testerror.c: In function 'main':
> testerror.c:1: error: incompatible types in assignment
```

- Determine which operators to apply
  - E.g. in x + y, "+" is integer addition if x and y are integers
- Recognize when to convert from one representation to another
  - E.g. in x + y, if x is a float while y is an integer, convert y to a float value before adding.

Static Type Checking

- Catch errors at compile time instead of run time
- Allows many errors to be caught early.
- C, C++, Java, Fortran, Pascal, Haskell, and C# (multi-paradigm programming language that encompasses functional, imperative and object oriented programming disciplines)

Dynamic Type Checking

- Type checking is performed at run-time (also known as "late-binding") as opposed to at compile-time.
- Dynamic typing may allow compilers to run more quickly.
- JavaScript, Lisp, Perl (scripting), Prolog, Python (dynamic OO), Ruby (dynamic OO) and Smalltalk.

A Simple Type Checking System

P → Ds;S
Ds → Ds;D
D → **id**:T { addtype(id.entry,T.type) }
T → char { T.type=char }
T → int { T.type=int }
T → float { T.type=float}
T → ↑T_1 { T.type=pointer(T_1.type) }
T → array[**intnum**] of T_1 { T.type=array(1..intnum.val,T_1.type) }

Type Checking: Statements

S → **id** = E { if (id.type==E.type then S.type=void
else S.type=type-error }

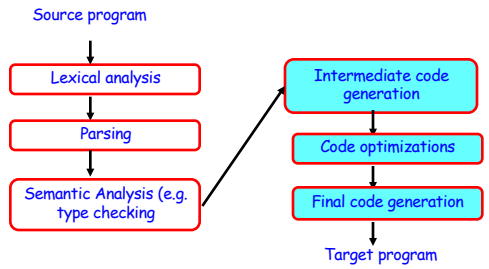
S → if E then S_1 { if (E.type==boolean then S.type=S_1.type
else S.type=type-error }

S → while E do S_1 { if (E.type==boolean then S.type=S_1.type
else S.type=type-error }

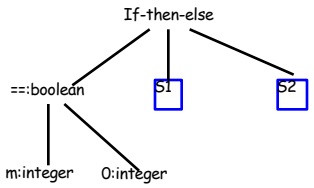
Type Checking: Expressions

E → **id** { E.type=lookup(id.entry) }
E → **charliteral** { E.type=char }
E → **intliteral** { E.type=int }
E → **floatliteral** { E.type=float}
E → **booleanliteral** { E.type=boolean}
E → E_1 + E_2
{ if (E_1.type=int and E_2.type=int) then E.type=int
else if (E_1.type=int and E_2.type=float) then E.type=float
else if (E_1.type=float and E_2.type=int) then E.type=float
else if (E_1.type=float and E_2.type=float) then E.type=float
else E.type=type-error }
E → E_1 [E_2]
{ if (E_2.type=int and E_1.type=array(s,t)) then E.type=t
else E.type=type-error }
E → E_1 ↑
{ if (E_1.type=pointer(t)) then E.type=t
else E.type=type-error }

Phase of Compilation



Intermediate Code Generation




```
    loadint m
    loadimmed 0
    intequal
    jmpz .L1
    jmp .L2
.L1:
    …. Code for S1
    jmp .L3
.L2:
    …. Code for S2
    jmp .L3
.L3
```

Code Optimization

- Apply a series of transformations to improve the time and space efficiency of the generated code
  - Reorder, remove or add instructions to change the structure of generated code

Code Optimization: Example

```
    loadint m
    loadimmed 0
    intequal
    jmpz .L1
    jmp .L2
.L1:
    …. Code for S1
   jmp .L3
.L2:
    …. Code for S2
   jmp .L3
.L3
```


Code Optimization: Example

```
    loadint m                    loadint m
    loadimmed 0                  jmpnz .L2
    intequal                .L1:
    jmpz .L1                    …. Code for S1
    jmp .L2                   jmp .L3
.L1:                    ➔   .L2:
    …. Code for S1               …. Code for S2
   jmp .L3                  .L3
.L2:
    …. Code for S2
   jmp .L3
.L3
```


Final Code Generation

- Map instructions in the intermediate code to specific machine instructions
- Generates sufficient information to enable debugging

Section 5

Basic Semantics

Names (Identifiers)

- Names in a programming language can be:
  - Variables
    e.g.: int x = 100;
  - Procedures
    e.g.: int fact (int n) {….}
  - Constants
    e.g.: const int n = 100;

Attributes

- Attributes are associated with names.
- Attributes describe the meaning or semantics of names.
- Examples:
  - For a variable: value, data type, memory location

int x;	There is a variable named x, which is of type integer
int y = 2;	There is a variable named y, which is of type integer, whose initial value is 2
Set s = new Set();	There is a variable named s, which is of type Set and refers to an object class Set.

Attributes (Cont.)

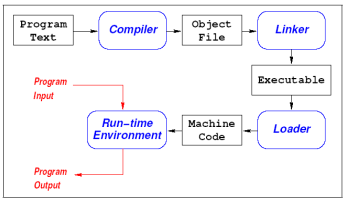
- For a function
  - Number, names, types of its parameters
  - Type of returned value
  - Body of a function or method

int fact (int n) {…}

  - The name of the function is fact
  - 1 parameter with name n and type int
  - Type of returned value int
  - The body of the code … in {…}.

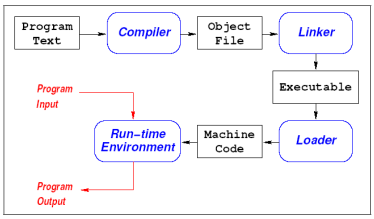
Binding Time

- Binding: the process of associating an attribute to a name



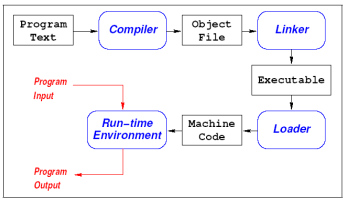
- Binding Time
  - Language Design (the design of syntax, primitive types and semantics):
    - e.g. meaning of type int
  - Language Implementation: max identifier length.
  - Compile time
    - e.g. value of n in const int n = 5

Binding Time (Cont.)



- Link time
  - A program is usually not complete until the various modules are joined together by a linker
  - Compile different modules of a program at a different time.
  - E.g. the body of function f in extern int f();

Binding Time (Cont.)



- Load time
  - Refers to the point at which the operating system loads the program into memory so that it can run.
  - E.g. the location of a global variable
- Execution time (Run time)
  - E.g. the location of a local variable

Static vs. Dynamic Binding

- Binding made prior to execution is called static binding.
- Binding made at run-time is called dynamic binding.

Language definition time
Language implementation time
Translation time (compile time)
Link time
Load time

} Static binding

Execution time

} Dynamic binding

Static vs. Dynamic Binding (Cont.)

- Languages differ substantially in which attributes are bound statically and which are bound dynamically.
  - Functional languages have more dynamic bindings than imperative languages
  - Binding time also depends on the translator

Static vs. Dynamic Binding: Example

- Value of an expression:
- Data type of an identifier:
- Maximum number of digits in an integer:
- Location of a variable:
- body of a function or method:

Static vs. Dynamic Binding: Example

- Value of an expression:
  - Execution or translation (constant expression).
- Data type of an identifier:
- Maximum number of digits in an integer:
- Location of a variable:
- body of a function or method:

Static vs. Dynamic Binding: Example

- Value of an expression:
  - Execution or translation (constant expression).
- Data type of an identifier:
  - Translation time (Java) or execution time (Smalltalk, Lisp).
- Maximum number of digits in an integer:
- Location of a variable:
- body of a function or method:

Static vs. Dynamic Binding: Example

- Value of an expression:
  - Execution or translation (constant expression).
- Data type of an identifier:
  - Translation time (Java) or execution time (Smalltalk, Lisp).
- Maximum number of digits in an integer:
  - Language definition time or language implementation time.
- Location of a variable:
- body of a function or method:

Static vs. Dynamic Binding: Example

- Value of an expression:
  - Execution or translation (constant expression).
- Data type of an identifier:
  - Translation time (Java) or execution time (Smalltalk, Lisp).
- Maximum number of digits in an integer:
  - Language definition time or language implementation time.
- Location of a variable:
  - Load (global variable) or execution time.
- body of a function or method:

Static vs. Dynamic Binding: Example

- Value of an expression:
  - Execution or translation (constant expression).
- Data type of an identifier:
  - Translation time (Java) or execution time (Smalltalk, Lisp).
- Maximum number of digits in an integer:
  - Language definition time or language implementation time.
- Location of a variable:
  - Load (global variable) or execution time.
- body of a function or method:
  - Translation time or link time (external function)

Static vs. Dynamic Binding: Example

- int x;
  - The type of x
  - The value of x
  - The location of x

Static vs. Dynamic Binding: Example

- int x;
  - The type of x
    statically determined
  - The value of x
  - The location of x

Static vs. Dynamic Binding: Example

- int x;
  - The type of x
    statically determined
  - The value of x
    dynamically determined
  - The location of x

Static vs. Dynamic Binding: Example

- int x;
  - The type of x
    statically determined
  - The value of x
    dynamically determined
  - The location of x
    dynamically or statically (if x is a global variable) determined

Section 5.2 Blocks and Scope

Blocks and Scope

- Usually, a name refers to an entity within a given context.
- The context is specified by "Blocks"
  - Delimited by "{" and "}" in C, C++, and Java
  - Delimited by "begin" and "end" in Pascal, Algol and Ada

```
int x; /*global*/
void p ()
{ double r;  /*the block of p*/
  …
  { int x, y; /*nested block*/
    x = 2;
    y = 0;
  }
  …
}
```


Scope

- Scope: where in a program a variable is accessible
- Scope is typically indicated implicitly by the position of the declaration in the code.
- The same name may be involved in several different declarations, each with a different scope.

```
void p( )
  { int x;
    …
  }
void q( )
  { char x;
    …
  }
```


Scope

- Declarations in nested blocks take precedence over previous declaration.

```
int x;
void p( )
{ char x;
  x = 'a';
}
main ()
{ x = 2;
}
```


Scope

- Declarations in nested blocks take precedence over previous declaration.

```
int x;
void p( )
{ char x;
  x = 'a'; /* assigns to char x */
}
main ()
{ x = 2; /* assigns to global x*/
}
```

The global declaration of x is said to have a scope hole inside p

Symbol Table

- Dealing with scopes: maintain a separate symbol table for each scope.
- Symbol table: in a compiler, only static attributes can be computed.

  Symbol Table: Names → Static Attributes

The Symbol Table

- Lexical analysis time
  - Lexical Analyzer scans program
  - Finds Symbols
  - Adds Symbols to symbol table
- Syntactic analysis time/type checking
  - Information about each symbol is filled in

The Symbol Table

- Can be implemented using different data structures.
- Allows efficient name lookup operations in the presence of scope changes.
- A scope stack that keeps track of the current scope and its surrounding scopes.
  - The topmost element in the scope stack corresponds to the current (i.e., innermost) scope
  - The bottommost element corresponds to the outermost (i.e., global) scope.

Static (Lexical) vs. dynamic scoping

- Static/Lexical scoping: scope is managed statically (prior to execution).
  - The scope of a binding is limited to the block in which its declaration appears.
  - C, C++, Java use static scoping
- Dynamic scoping: scope is managed directly during execution (run-time).
  - The scope of a binding depends on the order in which subroutines are called
  - The current binding for a given name is the one encountered most recently during execution.
  - Emacs lisp use dynamic scoping
  - Perl and Common Lisp support both static or dynamic scoping.

Example

```
float y = 1.0
void f() {
   y = 1;
}
void g() {
   int y;
   f();
}
void h() {
    float y = 10.0;
    f();
}
```

- Static/Lexical scoping:

Example

```
float y = 1.0
void f() {
   y = 1;
}
void g() {
   int y;
   f();
}
void h() {
    float y = 10.0;
    f();
}
```

- Static/Lexical scoping: the name y in function f always refer to the global name y
- Dynamic scoping:

Example

```
float y = 1.0
void f() {
   y = 1;
}
void g() {
   int y;
   f();
}
void h() {
    float y = 10.0;
    f();
}
```

- Static/Lexical scoping: the name y in function f always refer to the global name y
- Dynamic scoping:
  - If f is called from h, then y refers to the float variable declared in h.
  - If f is called from g, then y refers to the integer variable y defined in g

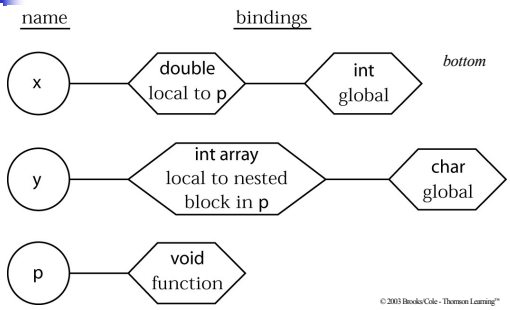
Static (Lexical) Scoping

- On entry into a block, all declarations of that block are processed and the corresponding bindings are added to the symbol table.
- On exit from the block, the bindings provided by the declarations are removed, restoring any previous bindings that may have existed.
- We view symbol table schematically as collection of names, each of which has a stack of declaration associated with it (independent of the data structures).

Static Scoping: Example

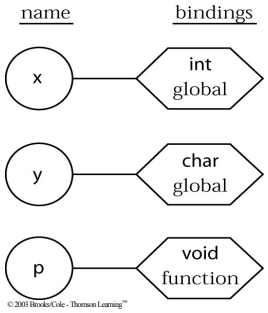
```
(1) int x;
(2) char y;

(3) void p()
(4) { double x:
(5)    x=123;
(6)    { int y[10];
(7)      …
(8)    }
(9)   …
(10) }

(11) void q()
(12) { int y;
(13)   …
(14) }

(15) main()
(6)   { char x;
(17)     …
(18) }
```


Static Scoping: Example



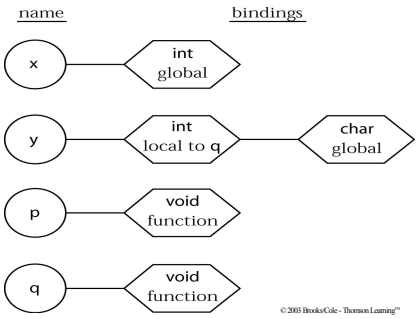
Symbol Table Structure at Line 7

Static Scoping: Example



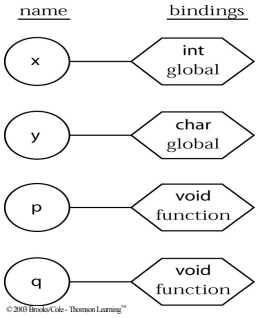
Symbol Table Structure at Line 10

Static Scoping: Example

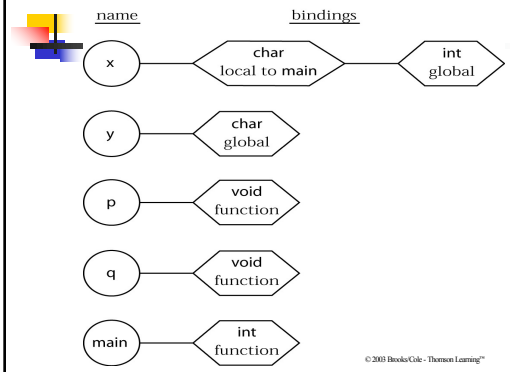


Symbol Table Structure at Line 13

Static Scoping: Example



Symbol Table Structure at Line 14



Symbol Table Structure at Line 17

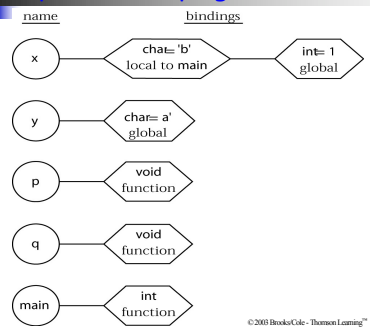
Dynamic Scoping

- Dynamic scoping: scope is managed directly during execution (run-time).
  - The scope of a binding depends on the order in which subroutines are called
  - The current binding for a given name is the one encountered most recently during execution.

Dynamic Scoping

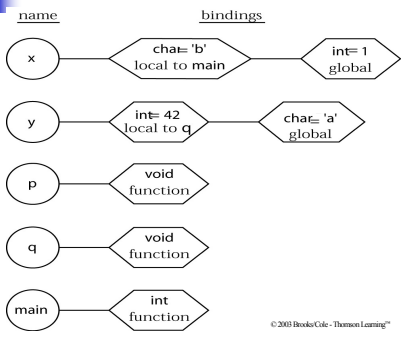
```
(1) #include <stdio.h>

(2) int x = 1;
(3) char y = 'a';

(4) void p()
(5) { double x = 2.5;
(6)   printf("%d\n",y);
(7)   { int y[10];
(8)   }
(9) }

(10) void q()
(11) { int y = 42;
(12)   printf("%c\n",x);
(13)   p();
(14) }

(15) main()
(16) { char x = 'b';
(17)   q();
(18)   return 0;
(19) }
```


Dynamic Scoping



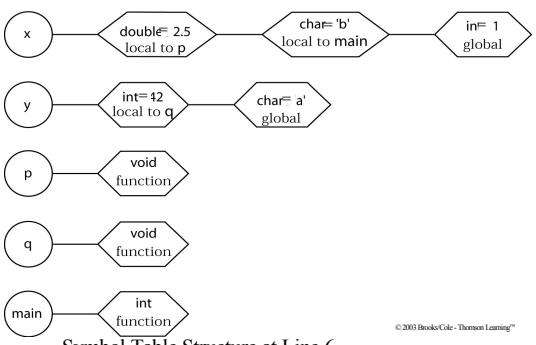
Symbol Table Structure at Line 17

Dynamic Scoping



Symbol Table Structure at Line 12

Dynamic Scoping



Symbol Table Structure at Line 6

Example

```
public class Scope {
    public static int x = 2;
    public static void f(){
        System.out.println(x);
    }
    public static void main(String[] args){
        int x = 3;
        f();
    }
}
```

- What will be printed out under
  - Static scoping?
  - Dynamic scoping?

Example

```
public class Scope {
    public static int x = 2;
    public static void f(){
        System.out.println(x);
    }
    public static void main(String[] args){
        int x = 3;
        f();
    }
}
```

- What will be printed out under
  - Static scoping? 2
  - Dynamic scoping?

Example

```
public class Scope {
    public static int x = 2;
    public static void f(){
        System.out.println(x);
    }
    public static void main(String[] args){
        int x = 3;
        f();
    }
}
```

- What will be printed out under
  - Static scoping? 2
  - Dynamic scoping? 3

Dynamic Scoping Evaluated

- Almost all languages (C/C++/Java/SML) use static scoping
  - With dynamic scoping, the meaning of a variable cannot be known until execution time – no static type checking
- Originally used in Lisp. Lisp and Perl support both static and dynamic scoping.