CSE450
Translation of Programming Languages
Lecture 10: Flow Control and Lexical Scope

Okay, human.
Huh?
Before you hit `compile`, listen up.

You know when you’re falling asleep, and you imagine yourself walking or something.

And suddenly you misstep, stumble, and jolt awake?

Yeah!

Well, that’s what a segmentation fault feels like.

Double-check your damn pointers, okay?
Flow Control
And, Or, If, Else, While, Break
Boolean Operators - Short Circuit Eval

val x = 1;
val y = 4 || (x = 3)
print(x, ' ', y)

What is the result?
1 4
1 1
3 1
3 4
Truth tables - cond_1 “?” cond_2

<table>
<thead>
<tr>
<th>cond_1</th>
<th>cond_2</th>
<th>And</th>
<th>Mult</th>
<th>Or</th>
<th>Add</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Why have short circuiting?

Reduce unneeded executions.

```scala
val x = 0;
(x == 0) || expensive_calculation();
(x != 0) && expensive_calculation();
```

So we can make a guarantee that the second condition will only be evaluated if needed.
Jumps

Tube Intermediate Code:

... tic instructions ...
jump label_a
... tic instructions ...
label_a:
... tic instructions ...

Tube Intermediate Code:

... tic instructions ...
jump_if_0 s1 label_b
... tic instructions ...
jump_if_n0 s1 label_b
... tic instructions ...
label_b:
... tic instructions ...
Short Circuiting as “if” statements

Short circuiting requires conditional execution (conditional jumps), so that the second clause is only executed if the first fails.

cond_1 && cond_2 becomes something like

if (cond_1) { result = cond_2; }
else { result = 0; }

cond_1 || cond_2 becomes something like

if (cond_1) { result = 1; }
else { result = cond_2; }

However, you can’t just return the second condition, because you must only return 0 (False) or 1 (True).
Flow control commands are statements that are implemented (like boolean short-circuiting) with jumps and labels.

**if (expression) statement**

1. compile, then evaluate expression
2. if false, jump to end_label
3. compile statement
4. label end_label:
“If” and “Else”

if (expression) if_true_statement else if_false_statement

1. compile, then evaluate expression
2. if false, jump to else_label
3. compile if_true_statement
4. unconditionally jump to end_label
5. label else_label:
6. compile if_false_statement
7. label end_label:
Dangling Else

Hold of the color for the code which is correctly indented?

```python
if (x) statement
  if (y) statement
else statement

if (x) statement
  if (y) statement
else statement
```
Dangling Else

Hold of the color for the code which is correctly indented?

if (x) statement
  if (y) statement
else statement

if (x) statement
  if (y) statement
else statement

The second. “else” binds to nearest “if” statement. So be sure else has higher precedence than if.
“While”

while (expression) statement

1. label start_label:
2. compile, then evaluate expression
3. if false, jump to end_label
4. compile statement
5. unconditionally jump to start_label
6. label end_label:
```
val x = 1;
while (x) {
  if (x == 4) {
    break;
  }
  x += 1
}
x += 11;
```

---

```
... pre-while statements ...
start_label:
... expression ...
jump_nequ s4 0 end_label
... statements ...
BREAK??
... more statements ...
end_label:
... post-while statements ...
```
```
val x = 1;
while (x) {
    if (x == 4) {
        break;
    }
    x += 1
}
x += 11;
```
Nested “While” and “Break”

val x = 0;
val y = 0;
while (x+=1) {
    while (y+=1) {
        if (y == 4)
            break;
    }
    if (x==5)
        break;
}

Where does each break jump to?

To the most recent while’s end_label.

How do you keep track?

The SymbolTable should keep a stack of “while scopes.”
Scope

‘{’ and ‘}’
Restricting our Tubular language to a global variables makes for difficult programming due to the inability to reuse variable names.

Instead, we use scope (a form of information hiding) to restrict identifiers to a defined region.

Lexical Scope: (as defined by ALGO 60 from 1960)

“the portion of source code in which a binding of a name with an entity applies”
Introduction to Scope

The scope of an identifier is the portion of a program in which that identifier is accessible.

- The same identifier (e.g. “x”) may refer to different things in different parts of the program.
- Both the value and type of an identifier can vary with scope.

Examples where scope changes:
- Inside conditional blocks or loops
- Inside functions
- Inside statement blocks
Variables in Tubular will exist in scopes

- Default scope will be global

- Each nested block will increment the current scope
- Exiting a block causes all variables declared in that block to become inaccessible to the rest of the program (out of scope)

- All variables in lower scopes remain accessible in the current scope
- Variables in lower scopes can be overridden (shadowed)
Given:

```scala
val a = 123;
if (a == 123) {
    val b = 456;
}
if (a == 123) {
    char b = 'x';
}
```

Is this program legal?

**YES**

**NO**
Given:

```scala
val a = 123;
if (a == 123) {
  val b = 456;
}
if (a == 123) {
  char b = 'x';
}
```

Yes, the two variable b’s have no relationship and aren’t redeclarations.
Given:

```java
val a = 123;
if (a == 123) {
    char a = 'x';
    print(a);
}
print(a);
```

Is this program legal?

YES

NO
Scope Example (2/4)

Given:

```java
val a = 123;
if (a == 123) {
    char a = 'x';
    print(a);
}
print(a);
Yes, shadowing.
```
Scope Example (3/4)

Given:

```scala
val a = 123;
{
  char a = 'x';
  print(a);
}
val a = 47;
```

Is this program legal?

YES

NO
Scope Example (3/4)

Given:

```plaintext
val a = 123;
{
    char a = 'x';
    print(a);
}
val a = 47;
```

NO!!!
Redeclaration Error!
Scope Example (4/4)

Given:

```scala
val a = 123;
{
  print(a);
  char a = 'x';
  print(a);
}
```

Is this program legal?

YES

NO
Given:

```sql
val a = 123;
{
    print(a);
    char a = ‘x’;
    print(a);
}
```

Yes, shadowing.
Implementing Scoping

Scoping can be implemented right within your symbol table(s).

When a variable is declared:
- Check that it has not been previously declared within this scope (but lower scopes are allowed)
- Add it to the table, recording its name, type, etc., along with the scope in which it was created.

When leaving a scope, simply deactivate symbols that are no longer accessible. They can’t be used again in the source program.
Given:

```java
val a = 123;
val b = 44;
if (a == 123) {
    char a = 'x';
    print(a);
}
print(a);
```

SymbolTable[0]:

```
val a
val b
```

SymbolTable[1]:

```
char a
```
## Two Possible Implementations

### Stack of Tables

SymbolTable[0]:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>val</td>
</tr>
<tr>
<td>b</td>
<td>val</td>
</tr>
</tbody>
</table>

SymbolTable[1]:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>char</td>
</tr>
</tbody>
</table>

### One table (to rule them all)

SymbolTable (Depth, Id):

- (0, a) -> val
- (0, b) -> val
- (1, a) -> char
Given:

```plaintext
def f(x):
    return x + 1
```

What is the value of `f(3)`?

1. `4`
2. `5`
3. `6`
4. None of the above.
Given:

```javascript
val a = 1;
{
    print(a);
    #val a;
    a = 2;
}
print(a);
```

What will the code print when compiled and executed:

1 then 1
1 then 2
a then a
Undefined / Error
Scope Question

Given:

```scala
val a = 1;
{
    print(a);
    val a;
    a = 2;
}
print(a);
```

What will the code print when compiled and executed:

1 then 1
1 then 2
a then a
Undefined / Error
A Quiz Question

The CFG (to the right) accepts:

\[ x = (3 + 4); \]

But rejects:

\[ x = y = 3; \]

and

\[ x = 3 + (y = 5 + 4); \]

Hold up the color(s) of the rule(s) that need modification to accept all three strings.
Precedence Reference

The precedence (from lowest to highest) of the operators is:
Assignment Operators (right associative)
Conditional Operator (extra credit, '?' and ':' ; right-associative)
Logical OR (left associative)
Logical AND (left associative)
Relationship Operators (non-associative)
Add/Subtract (left associative)
Multiply/Divide (left associative)
Unary Minus (non-associative)