College of Engineering
and Computer Science

UNIVERSITY OF CENTRAL FLORIDA

Control Flow

COP-3402 Systems Software
Paul Gazzillo

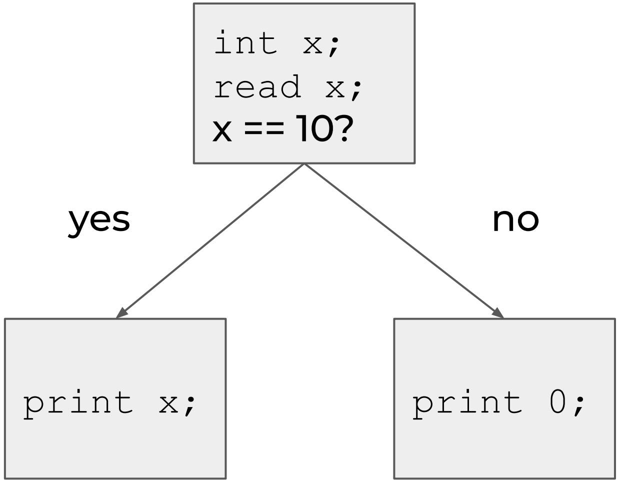
If Statements Encode Decisions

- Uses Boolean logic
- Instructions only executed when conditions are met

```
int x;
read x;
if (x == 10) {
  print x;
} else {
  print 0;
}
```

If Statements as a Flow Chart

```
int x;
read x;
if (x == 10) {
  print x;
} else {
  print 0;
}
```



While Loops Encode Repetition

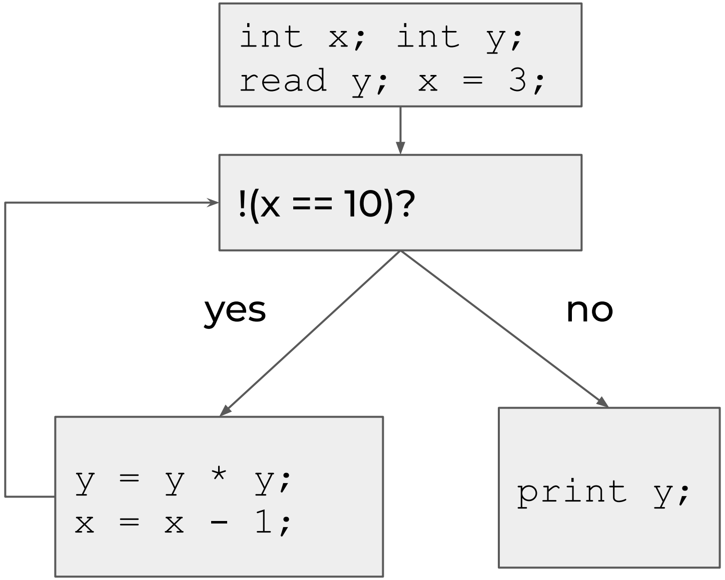
- Repeat instructions until a certain conditional is met
- A unconditional branch plus an if-statement

```
int x;
int y;
read y;
x = 3;
while (!(x == 0)) {
  y = y * y;
  x = x - 1;
}
print y;
```

```
loop:
if (!(x == 0)) {
  y = y * y;
  x = x - 1;
  goto loop;
}
print y;
```

While Loops as a Flow Chart

```
int x;
int y;
read y;
x = 3;
while (!(x == 0)) {
   y = y * y;
   x = x - 1;
}
print y;
```



```
int x; int y;
read y; x = 3;
```

!(x == 10)?

```
y = y * y;
x = x - 1;
```

```
print y;
```

For Loops

- SimpleC does not have a for loop
    - (Optional bonus project)
- Can we still express a for loop?
- How can we write this for loop in SimpleC?

```
int x; int y;
read y;
for (x = 3; !(x = 0); x = x - 1) {
  y = y * y;
}
print y;
```

```
int x;
int y;
read y;
x = 3;
while (!(x == 0)) {
  y = y * y;
  x = x - 1;
}
print y;
```

Reminder: Left Recursion Elimination

- Boolean expression grammar is left recursive

```
expression = expression OR andexpr

andexpr = andexpr AND equalsexpr

equalsexpr = equalsexpr EQUALS addexpr

addexpr
  = addexpr PLUS term
  | addexpr MINUS term
```

```
term
  = term TIMES factor
  | term DIVIDE factor
  | term MOD factor

factor:
  = NOT expression
  | LPAREN expression RPAREN
  | NUMBER
  | IDENTIFIER
```

Reminder: Right Recursion Trick

- Turn the right recursion into a while loop

```
expression():
  andexpr()
  while (lookahead is OR):
    andexpr()

andexpr():
  addexpr()
  while (lookahead is AND):
    addexpr()

addexpr():
  term()
  while (lookahead is PLUS or MULT):
    term()
```

Grammar for If Statements

Control Flow Statement Grammar

- Three new statements
  - If-then-else
  - If-then
  - While
- Plus a new compound statement { … }
  - This allows for structured programming

```
statement
  = IF LPAREN expression RPAREN statement
  | IF LPAREN expression RPAREN statement ELSE statement
  | WHILE LPAREN expression RPAREN statement
  | LCURLY statement* RCURLY
```

Demo: Parse Tree for If Statements

```
if (x) print 0; if (y) print 1; else print 2;
```

The Dangling Else Problem

- The if-statement grammar is ambiguous
  - If lookahead is `else`, which production are we in?
- Preceding `statement` could be an if-statement

```
statement
  = IF LPAREN expression RPAREN statement
  | IF LPAREN expression RPAREN statement ELSE statement
```

Resolving the Dangling Else

- Match `else` to nearest `if`
- First method: make matching explicit in the grammar

```
statement = matched_stmt | unmatched_stmt

matched_stmt
  = IF LPAREN expression RPAREN matched_stmt ELSE matched_stmt
  | // other statements besides if-then

unmatched_stmt
  = IF LPAREN expression RPAREN statement
  | IF LPAREN expression RPAREN matched_stmt ELSE unmatched_stmt
```

Second (Easier) Method

- Always assume `else` is part of current production
- First left factor:

```
statement = if_statement
if_statement = if_prefix else_option
if_prefix = IF LPAREN expression RPAREN statement
else_option = ELSE statement | ε
```

- If lookahead after `if_prefix` is `else` assume `else_option` is not ε

Pseudo-Code for Resolving Dangling Else

```
if_statement():
  consume(IF)
  expression()
  consume(THEN)
  statement()
  if (lookahead == ELSE):
    consume(ELSE)
    statement()
  else:
    // epsilon
```

Demo: Parsing If Statements

```
if (x) print 0; if (y) print 1; else print 2;
```