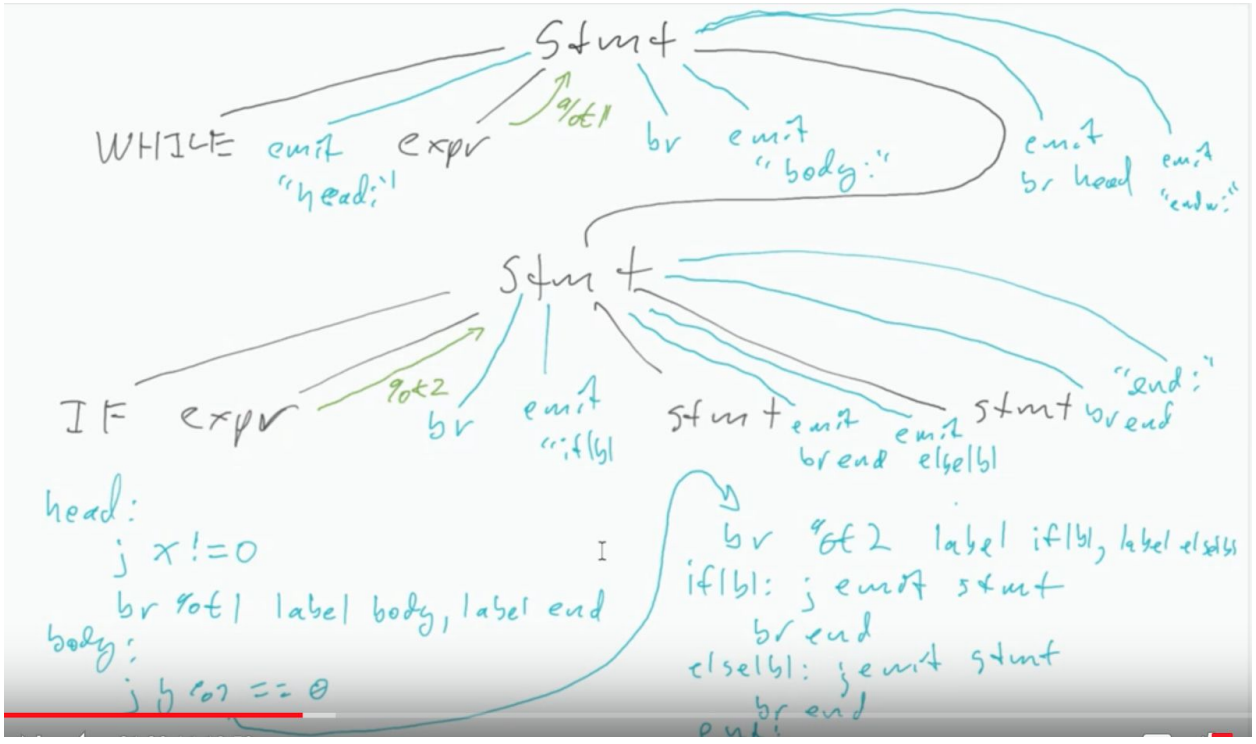
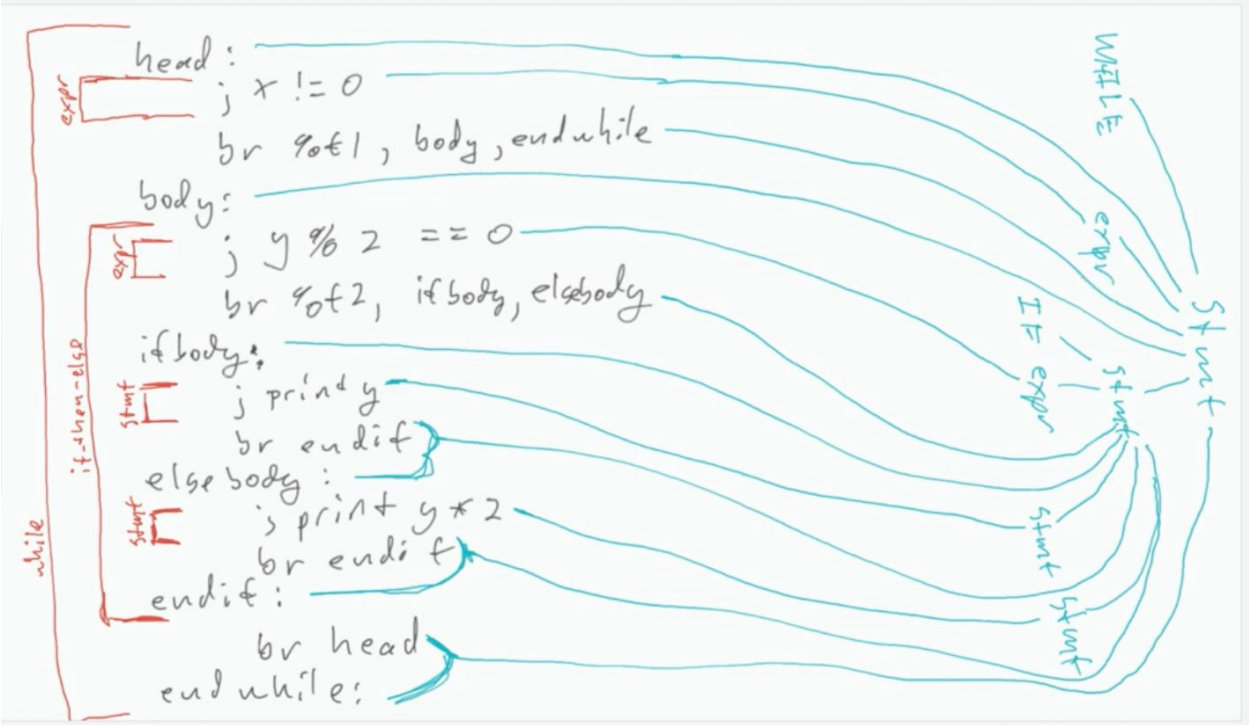
COP-3402 Systems Software
10.31 Thu

Ex: Code generation for a nested if-then-else statement inside a while-loop.

```
int x;
int y;
x = 3;
read y;
while (x != 0) {
  if (y % 2 = 0)
    print y;
  else
    print y * 2;
  x = x - 1;
  y = y + 1;
}
```

We consume the ‘while’ keyword and LPAREN then we generate a new label, and we emit that label immediately because we want to have a label on the expression computation so we can jump back to that and we can recompute the while-loop’s condition. We consume RPAREN, and generate labels for the body and the end and then generate the branch. The body of while-loop is another statement that will include if-else-statement. Next, please refer to the pseudo-code for the if-then-else statement to generate code for it.





Other Bonus Project Ideas

- Implement constant propagation and folding
- Add a Boolean type
- Perform dead code elimination
- Add inequalities in conditional expressions
- Implement for loops
- Check that arithmetic and Boolean operations are not mixed
- Check that all paths of a function have a return statement
- Check for unreachable code in a function (after a return statement)

Please refer to the link below for the grammar, objectives, and semantics of project#4.
https://github.com/cop3402fall19/syllabus/blob/master/projects/project4.md

Functions:
Functions in SimpleC:
Similar to C
Functions have a name, parameters as inputs, parameter types, output.

Ex:

```
int square(int x) {
        return x * x;
}
```

Grammar for Project#4

```
program
  = declaration* function* statement*

function
  = INT IDENTIFIER LPAREN ( IDENTIFIER (COMMA IDENTIFIER)* )? RPAREN LCURLY
declaration* statement* RCURLY

declaration
  = INT IDENTIFIER SEMI

statement
  = PRINT expression SEMI
  | READ IDENTIFIER SEMI
  | IDENTIFIER ASSIGN expression SEMI
  | IF LPAREN expression RPAREN statement
  | IF LPAREN expression RPAREN statement ELSE statement
  | WHILE LPAREN expression RPAREN statement
  | LCURLY statement* RCURLY
  | RETURN expression SEMI

expression
  = expression PLUS expression
  | expression MINUS expression
  | expression TIMES expression
  | expression DIVIDE expression
  | expression MOD expression
  | expression EQUALS expression
  | expression LT expression
  | expression AND expression
  | expression OR expression
  | NOT expression
  | LPAREN expression RPAREN
  | NUMBER
  | IDENTIFIER
  | IDENTIFIER LPAREN ( expression (COMMA expression)* )? RPAREN
```

Functions abstract away computation.
Functions for recursive descent:
Carefully define each parsing function; inputs, outputs, positions of file pointers (fgetc), etc.
Composable by making sure all parsing functions follow the same rule.

Versions of C allow local variables inside of while-loops, SimpleC does not. SimpleC allows local variables in functions.

Static Scoping

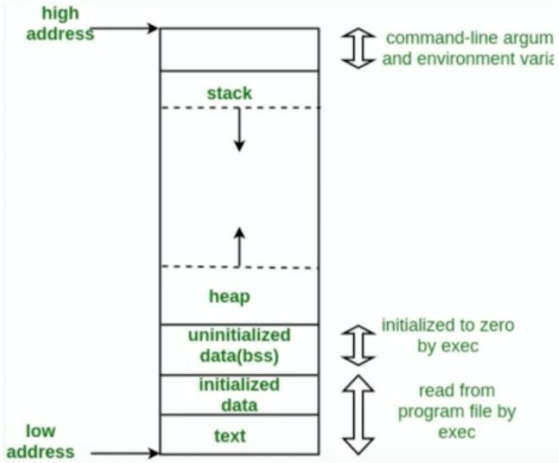
- Region of code where variable is valid
  - Compound statement
  - Nested scopes, e.g., int x
- Local variables
  - Declared in nested scopes
  - Function parameters
- Out-of-scope variables
  - e.g., print y

```
int x;
int f(int x) {
  int y;
  read y;
  return x * y;
}
print x;
print y;
```

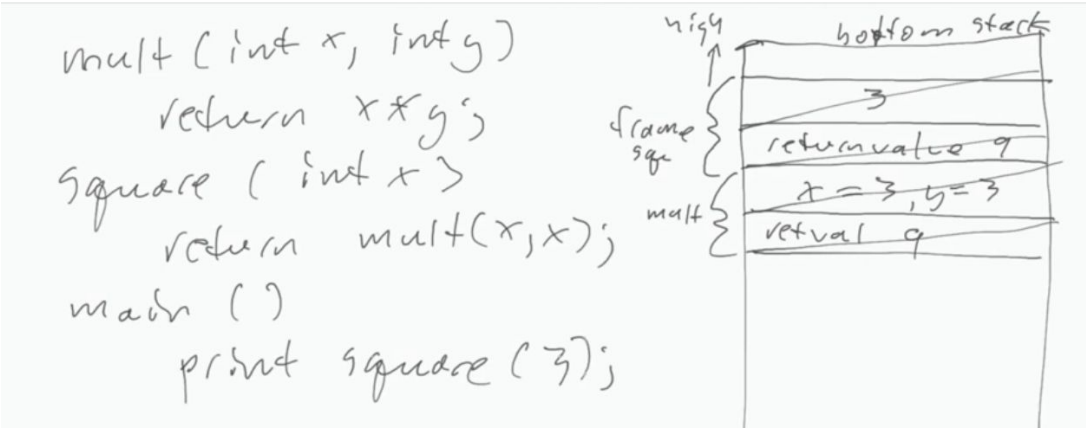
We can use stack to pass values:

Freezing Function State Using the Stack

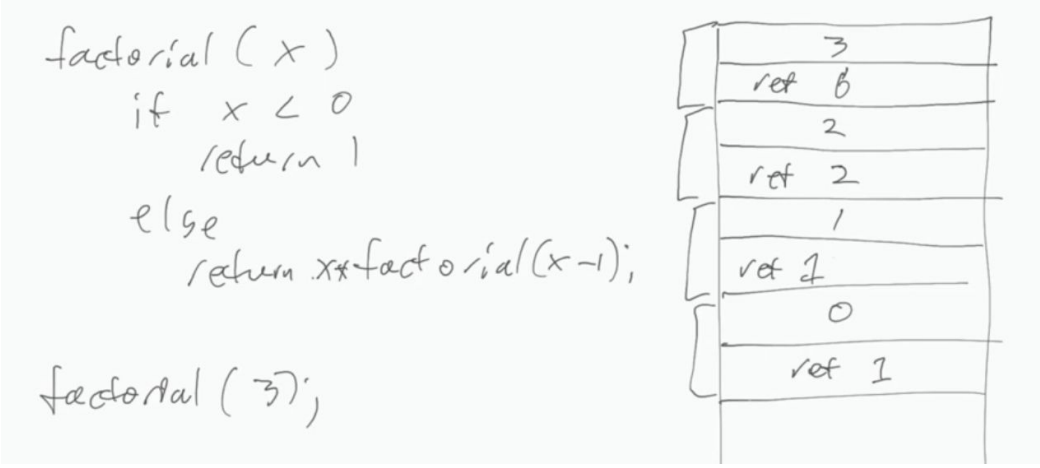
- Stack holds local variables



We have this address base, zero at the bottom, high addresses at the top. The text is the actual code (binary version of it), global values, heap which we have access with malloc, and then stack which is memory automatically managed by the compiler. We push values onto the stack when we want to call a new function, we push its parameters onto the stack, pop them off and then push room for return value onto the stack. We use stack because the order of calls matters.



Let's say we have a recursive function, factorial:



We push the address onto the stack return to, so we know where to jump back to. It is called the dynamic link if you want to refer to the dragon book. And then, we pull that address off of the stack and jump to it, so the branch gets filled in with values in the stack.

```c
int f(int x) {
  return x;
}

int main() {
  f(2);
}
```

```
  .file "test.c"
  .text
  .p2align 4,,15
  .globl  f
  .type f, @function
f:
.LFB0:
  .cfi_startproc
  movl  %edi, %eax
  ret
  .cfi_endproc
.LFE0:
  .size f, .-f
  .section  .text.startup,"ax",@progbits
  .p2align 4,,15
  .globl  main
  .type main, @function
main:
.LFB1:
  .cfi_startproc
  xorl  %eax, %eax
  ret
  .cfi_endproc
.LFE1:
  .size main, .-main
  .ident  "GCC: (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0"
  .section  .note.GNU-stack,"",@progbits
```