What does the TIC program output?

```
val_copy 4 s1
label:
val_copy label s2
add s1 s2 s3
out_val s3
```

4
5
ERROR
Damn You Josh!!!
Example Definition

define val max(val value1, val value2) {
    if (value1 > value2) return value1;
    return value2;
}

TIC Code for Definition

define val max(val value1, val value2) {
    if (value1 > value2) return value1;
    return value2;
}

#define return s1, return_label s2
#define value1 s3, value2 s4
#define function_max:
#define test_gtr s3 s4 s5
#define jump_if_0 s5 if_else_1:
#define if_else_1:
#define if_end_2:
#define val_copy s4 s1
#define jump s2
Example Function Call

val high_score = max(score1, score2);
print("And the high score is: ", high_score);

Or

max(max(score1, score2), score3);

Or

print(max(8, 9));
print(max(8, 9));

val_copy 8 s6
val_copy 9 s7
val_copy s6 s3
val_copy s7 s4
val_copy function_return_0 s2

jump function_max
function_return_0:
val_copy s1 s8
out_val s8
out_char '\n'

# return s1, return_label s2
# value1 s3, value2 s4
function_max:
  test_gtr s3 s4 s5
  jump_if_0 s5 if_else_1:
  val_copy s3 s1
  jump s2

if_else_1:
  if_end_2:
val_copy s4 s1
jump if_end_2
What things does the function call need to know about the function being called?

1. The parameter variables
2. The return variable
3. The return label variable
4. The function body label
What does the program output?

val x = 4;

define val biggerThanX(val i) {
    return i > x;
}

x = 0;

print(biggerThanX(2));

0
1
ERROR

Damn You Josh!!!
Is this recursion?

define val max(val i, val j) {
    if (i > j) return i;
    return j;
}
max(max(1, 3), 2);
Is this recursion?

define val countdown(val i) {
    print(i);
    if (i == 0) return 0;
    countdown(i - 1);
    return 0;
}
countdown(4);
Is this recursion?

```plaintext
define val sub_one(val i) {
    return sub(i, 1);
}
define val sub(val x, val y) {
    return x - y;
}
sub_one(4);
```

Yes
No
Maybe?
Recur-what?
Is this recursion?

```plaintext
define val collatz_even(val i) {
    print(i);
    if (i == 1) return 0;
    return collatz_odd(i / 2);
}

define val collatz_odd(val i) {
    print(i);
    return collatz_even(i * 3 + 1);
}

collatz_odd(21);
```
define val fib(val position) {
    if (position <= 1) return position;
    return fib(position - 1) + fib(position - 2);
}

print("The 7th Fibonacci number is: ", fib(7));
Implementation (See Project 7)

1. Update the lexer
2. Update the symbol table
3. Update the parser
4. Update semantic checking
5. Build intermediate code to handle functions WITHOUT a call stack
6. Use the new Intermediate Code instructions to implement a call stack
7. Convert new intermediate code instructions into TubeCode Assembly
Update the lexer

Really simple, just add the two new keywords:

“define” and “return”
Many different implementations, but the reference compiler does it this way:

1. Functions are entries added to a new (function) symbol table (like variables)

2. Each function (whose lookup is by name) has a pointer to a mini AST that contains its body

3. The function symbol table needs to track the function’s name, return type, and the type of its arguments
Update the Parser

1. Add grammar for definitions: They must be in the global scope, but the body (including the arguments) need to be in a narrower scope.

2. Add grammar for function calls: They are expressions and can have zero or more arguments
1. A function call must have the same number and type as the definition.

2. There is no overloading (functions with the same name, but different arguments), unless you want extra credit.

3. The function call’s type is the return type. The call’s location, definition’s return type and the return statement should all match.

4. The “return” keyword is only allowed within a function.
Build Basic Function Call w/o Stack

1. (Function Body) Make sure function definition is not executed by default
2. (Function Body) Execute code
3. (Function Body) Return copies expression to return variable and jumps to return_label variable
4. (Function Call) Setup parameter values (move them to appropriate parameter variables)
5. (Function Call) Make a return label and put it in return_label variable
6. (Function Call) Do Call! Jump to Function Body
7. (Function Call) Drop label and use the return variable as expression.
Call Stack

Why to we need it?
What needs to go on it?
How to we construct it?
New Intermediate Code

“push” “pop” “ar_push” “ar_pop”

They push (or pop) a value from a call stack.

Why?

We need a way to protect local variables from a recursive call. So we push them to the stack, where future calls can't affect them. At the conclusion of the call, we pop them off the stack to restore the previous state.
New TubeCode

Convert the “push” / “pop” (and “ar_push” / “ar_pop”) Intermediate Code to TubeCode Assembly.

No new instructions needed.

After the normal compilation, setup the call stack by using regH to point to the top of the call stack (for instance position 5,000)

A push stores the value at the top and increments it, a pop decrements and gets the value there.