Last time we started on backend and p-code machine
it is not technically a part of the compiler.
machine code is really similar to RISK.
we are not going to turn this code to binary code. the interpreter can do that.
assemblers takes the text of machine code and translates it to binary. we will talk about it a little bit, but we are not going to implement it.
Questions?
pcode:
it is important to know how the instructions work. think about what is the pseudo code for each instruction.

Today:
Functions?
for the first half of the semester we went over the front end of the compiler.
Now we got this great AST. We can traverse it and it has types.
Backend takes that nice instruction and translate it to machine code.

we are converting pl0 to machine code.
Functions is one of the most fundumental part of the language.
it can help us to abstract machine code.
compilers generates codes that can simulate functions.
when we call a function in another function it is like the first function is wailting for the second function to get completed, so the first function is kind of freezed in memory.

Stack Frame (activation record)
Function state stored on stack in activation records.
Functionstate:
parameters: we need to know what are the parameters of the function
return value
pointer to parent function(static scoping)
dynamic link
return address
local variables
…

we usually put the bottom of the stack on top!
so we start from top to down.

frame pointer: is pointing to current running function stack frame.

in the callers code sets up all of the first part, so frame pointer usually starts from the middle of the stack, and the callee sets up the rest of the stack (after frame pointer).

Dynamic link: we want to be able to restore all of that frozen function.
Every new function call is a new activation record on the stack.

because we saved dynamic link we can go back to the correct stack frame and because we have return address we can find the address of the instruction (instruction pointer).

we have an example in the slides: function bob()
and function f() function joe()

static link helps you how to find the parent scope.

dynamic link always pointing to the previous function (where we called the current function)
static link always points to stack frame of parent scope. (so it is always the same for the function, no matter where we call the functions.)

it becomes more important when we have more nested functions.

to implement static scoping: function state stored in stack frame

How to find parent function stack frame? we record it in the current stack frame.

There is another example in the slides which is more complicated.

```
function f()
        function g()
                return h()
        function h()
        g()
f()
```

global
f
g
h

dynamic: h -> g -> f -> global
static: h->f, g -> f, f-> global

constructing stack frames:
At compile time we know the number of parameters, and local variables, we know where we called the functions, and we know the parent scope of each function
we also know the scope of the symbol.

we always reserve 4 spaces for these information, so we use frame pointer - 4 most of the time.

ershov numbers is for register allocation and we will talk about it next time.
Today's is only about functions.

push and pop are more complicated because they update something in the memory and also in the registers.

next time we will have machine code, so this time let's take a look at the diagrams.

```
function bob(x : int) : int
        begin
                write x
        end
bob(2)
```

(at the end)fp ~~fp~~ Global ~~sp~~ sp(at the end)
x=2
return=0 (assume)
static = 0
(br) ret address =(IP + 1)
~~fp~~ dynamic = 0
x = 0 ~~sp~~

Let's run the actual code machine for this function:

very important: when you generate the code the order can be very different from when you run the code. it can be very confusing.