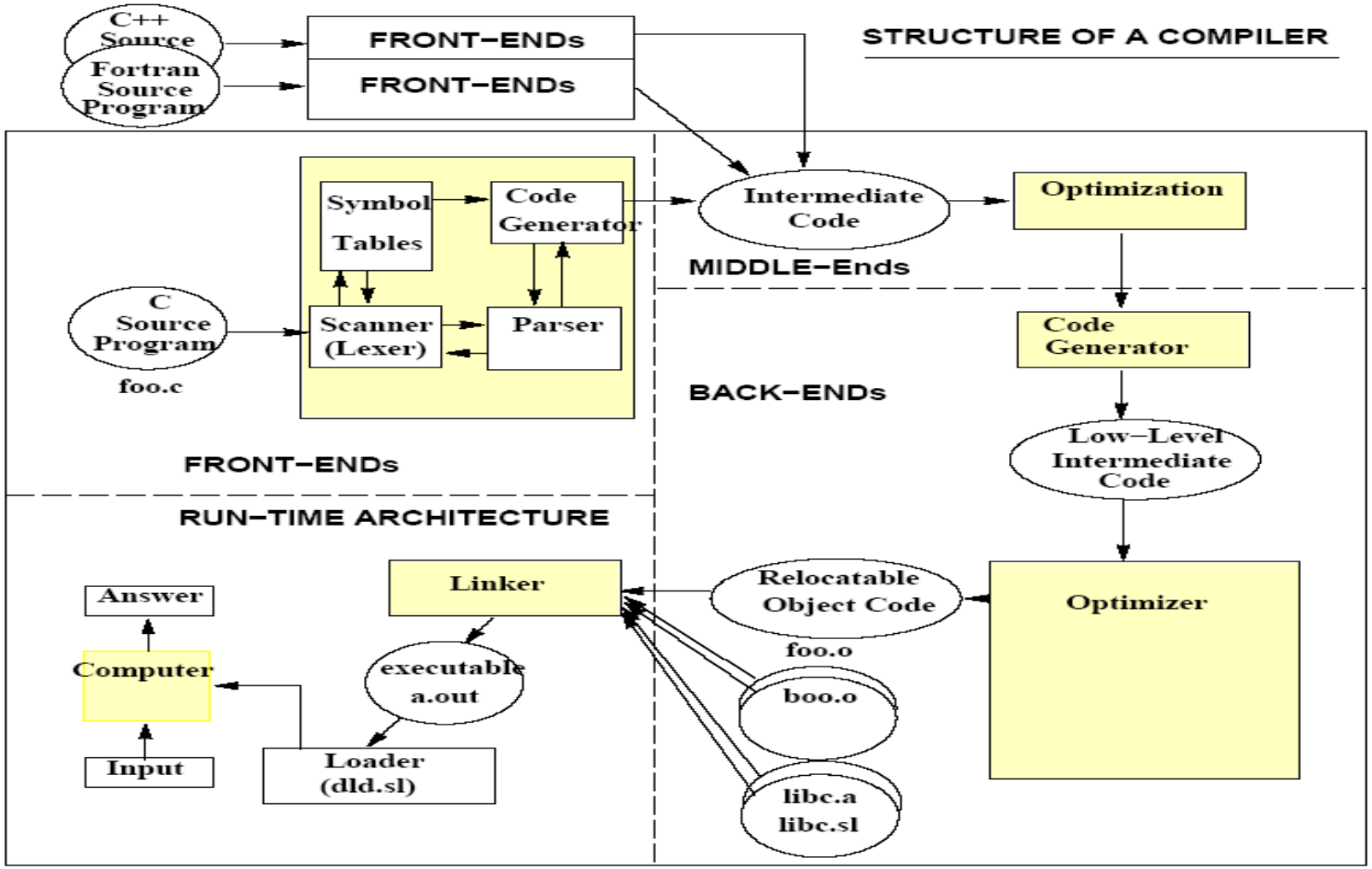
Code Generation

Dragon: Ch 7. 8. (Just part of it)

Holub: Ch 6.

Compilation Processes Again

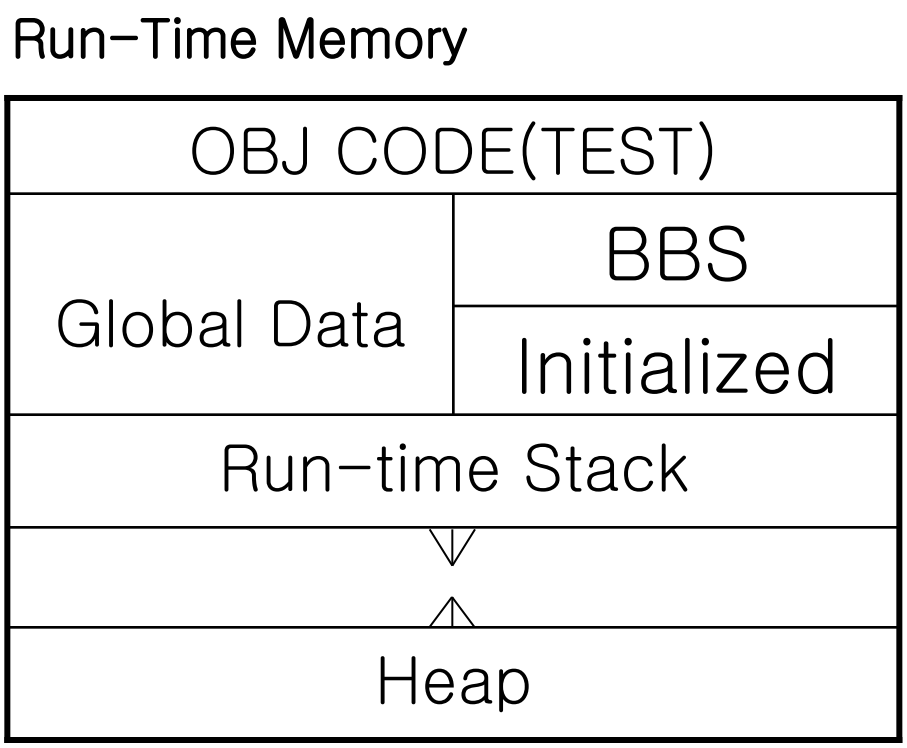


Choice of Intermediate Code Representation (IR)

- IR examples
  - Parse tree
  - Three address code (e.g., x := y op z)
  - Abstract stack machine code
- We will generate stack-machine code in our project
  - Operand stack is used for storing temporary results
    - E.g., `c=a+b;` => `push a; push b; add; store c`
- Problems of code generation
  - Data Allocation
  - Data Access
  - Code Generation

Data Allocation: Different Types of Storages

- Global data area
  - Global Objects, Local Static Objects, Constants
- Run-time stack
  - Local Objects
  - Parameters
  - Operand stack
    - For stack machine
- Heap
  - Dynamic Objects

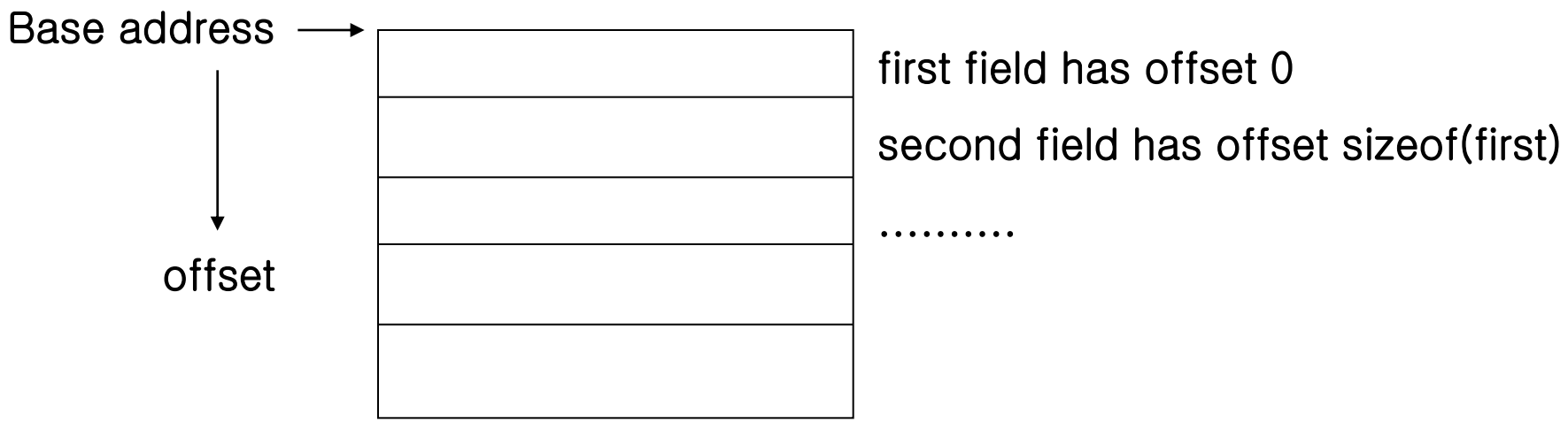


Data Allocation: Size of Data Objects

- How big is an object of a given type for a byte-addressed machine
- Basic Objects
  - int: 4 bytes, float: 4 bytes, char: 1 byte
- Compound Objects: compute recursively based on type expressions
  - pointer: 4 bytes
  - `struct`: sum of its field sizes
  - array: array length × element size

Data Allocation: Data Layout

- `Struct` has internal parts; how the compiler lay them out?
- Simple issues: each field has a size and an offset
  - When processing a `struct` declaration, the total size of previous fields is maintained as a variable and this gives you the location (i.e., the offset) for the next field



Some Data Layout Issues

- More complicated issues
  - Alignment (e.g., locate `int` at the word boundary)
  - Packing
  - Reorder to optimize
- For our programming assignments, we can get around lots of these problems by making everything a nice size (e.g., word)

Accessing Data

- Basically, every data object access must first be converted into an arithmetic access expression
  - Has a form of base pointer + offset
  - base pointer depends on where the object is allocated
    - Global variables: global pointer
    - Local variables: stack pointer or frame pointer
  - offset is composed of two parts
    - object offset: the offset of the object from the base pointer
    - item offset: for a compound object such as struct or array, the offset of specific item from its beginning (0 for singleton object)
  - So, it is actually base pointer + object offset + item offset

Base Pointer

- Where does the base pointer come from?
  - Global data area: global pointer (compile-time constant)
  - Parameters and local variables:
    stack & frame pointer is maintained at run-time
  - Parameter & Local variables of enclosing procedure:
    static links, access links

Base_pointer + object offset

- For a global object x:
  - Global_Pointer + x_offset
- For a parameter or local object x
  - stack_frame_ptr + x_offset
  - stack frame is maintained by run-time system during procedure call

Item Offset

- Item offset is also defined recursively
- For example, for A[i+1].f[2*j] we need to sum up
  - Offset of A[i+1] from the beginning of A[]
  - Offset of f[] from the beginning of A[i+1]
  - Offset of f[2*j] from the beginning of f[]
- So we build up an item offset recursively in bottom-up
  - If α is the beginning an object and we want to access part of it
  - `struct`: α + field offset
  - array : α + index × element-size
  - pointer: want the contents of an address;
    - fetch(α): gets the contents of locations α

Example: Assignments

- Variable: associated with location and value
- Location is a box that holds value of the variable
- Example: x = y
  - x is converted into a location
  - y is also converted into a location but we get its value using a command (instruction) called fetch()
  - for global variables x and y:
    Global_Pointer + x_offset = fetch(Global_Pointer + y_offset)
  - for local variables x and y:
    Stack_pointer + x_offset = fetch(Stack_pointer + y_offset)
  - For a different example: *x = **y;
    fetch(x_location) = fetch(fetch(fetch(y_location)));

Example: Complex Assignment

```
struct foo {
    int f3;
    int f4;
};

struct boo {
    int f1[3];
    struct foo f2;
};
struct boo a[5];

a[3].f2.f4 = 1;  ⇒ Base_pointer + a_offset + 3 * 20 + 12 + 4 = 1;
```

- Final address will be Base_pointer + some_constant, which will be used for code generation

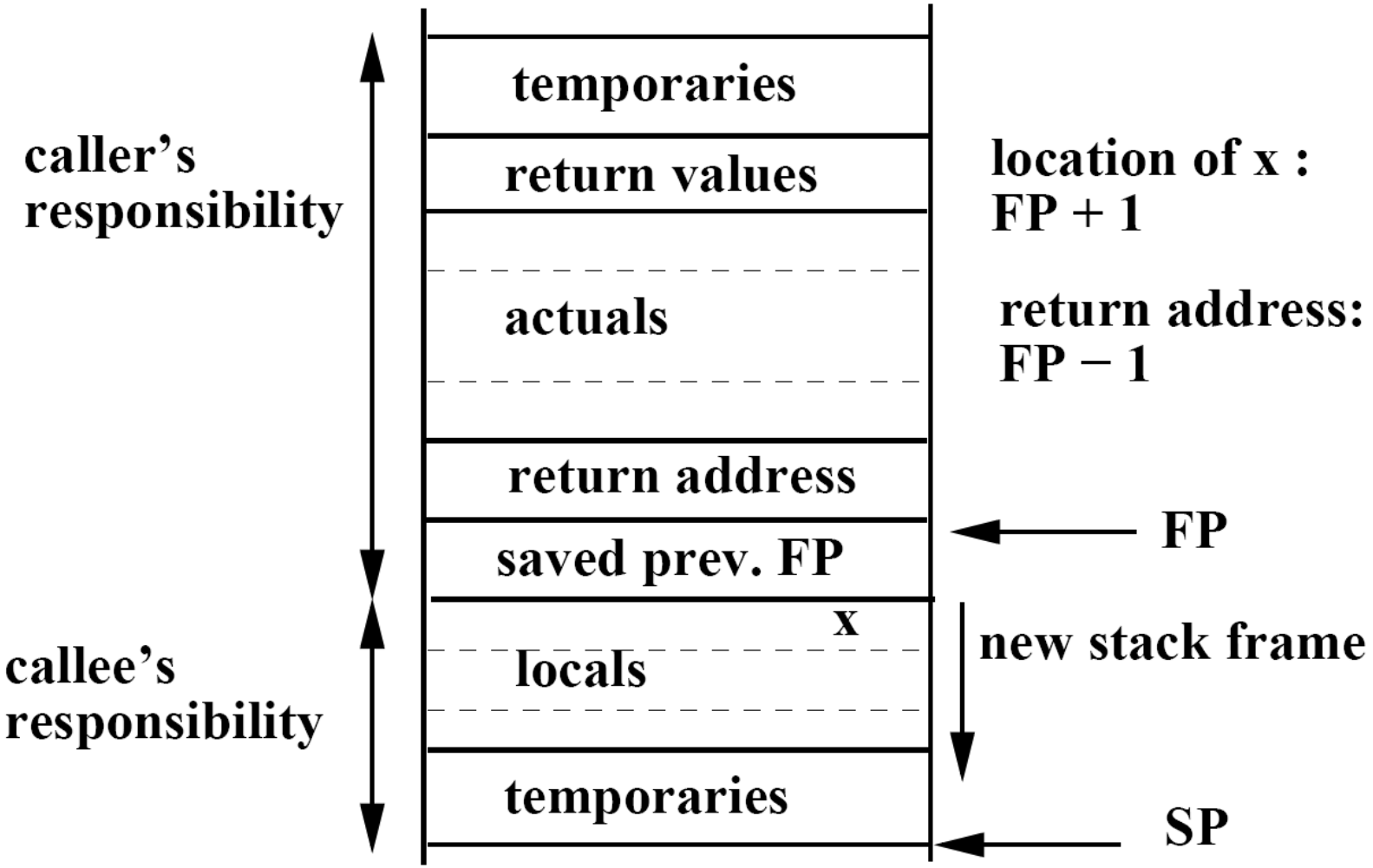
Procedure Calls

- Compile-time vs. run-time
  - Must allocate storage for each activation's information and store activation record on the run-time stack
  - Lots of choices about what goes inside the activation record and who puts in them
    - Format of the activation record
    - Caller/Callee responsibilities
  - ISA (Instruction Set Architecture) may help (e.g., VAX)

The Case in a Simple Abstract Stack Machine

- A single run-time stack holds an activation record as well as an operand stack for temporaries in evaluating expressions
- Two stack pointers in our stack machine
  - SP: top of the stack
  - FP : keep track of activation records for accessing parameters, locals, return values

STACK FRAME Structures



Handling Expressions

- Temporaries: a place to store intermediate values; most machines do not have an `a+b*c` instruction

```
t = b * c;
t += a;
```

- Efficient allocation of temporaries is a major issue for compilers which requires optimizations
- For stack machines, store temporaries on stack

```
push b; push c; mul; push a; add
```

Procedure Call: Caller Responsibility

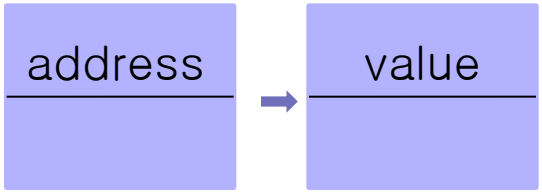
- Call generation code should do
  - push a hole for return value
  - push the actual parameters on the stack
  - push the return address
  - push the old FP
  - FP = SP
  - Then, jump
- On return, clear the allocated space

Procedure Call: Callee Responsibility

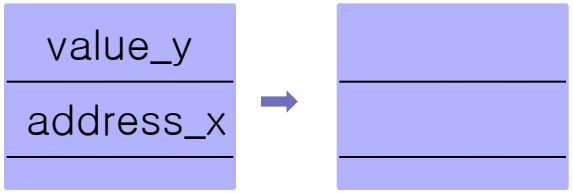
- Callee code should do inside a procedure
  - push space on stack for locals
  - execute the body of the procedure
  - put return value in the hole created
  - SP = FP
  - pop the old FP into FP
  - pop and jump to return address
- Variations
  - Arguments/return values could be passed in registers
  - Change the caller/callee responsibilities

Variable Access

- In stack machine, you need to fetch the value of a variable onto the stack before any computation
  - First push its address on the stack
  - Then, execute `fetch`
  - Compute using the value(s)



- For code generation for assignment (e.g, x = y)
  - Push address of x and value of y
  - Then, execute `assign`



Variable Address

- A variable's address depends on its scope and offset, which can be found from the symbol table
  - Base_pointer: Lglob (global), FP-based pointer (local)
  - *id_offset*:
    - Offset from the beginning of the "record" of all global variables
    - Offset from the beginning of the local variable area or parameter area in the stack

Pushing Variable Address

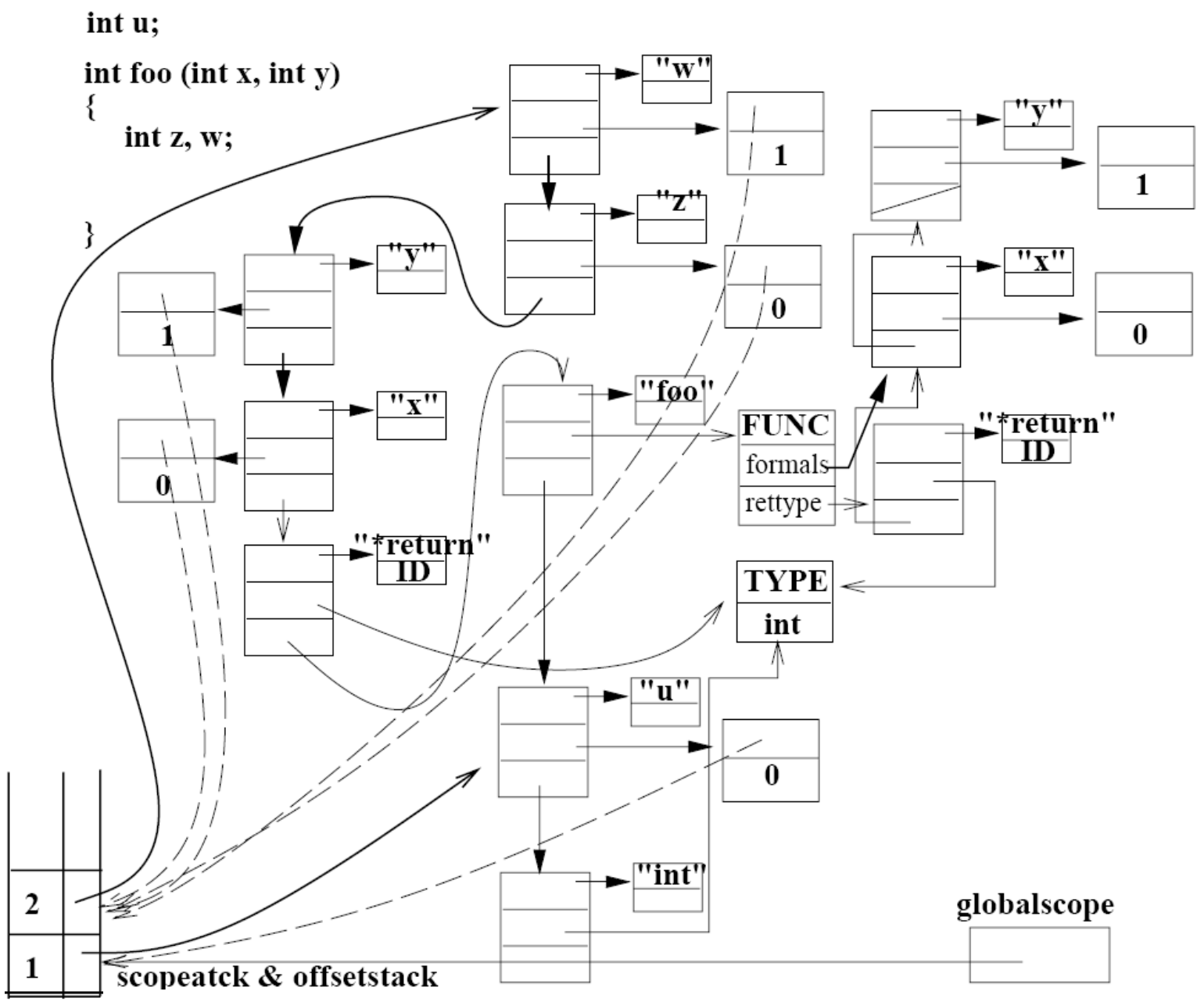
- Global variables:
  - Location: Lglob + *id_offset*
  - Intermediate code: `push_const` Lglob + *id_offset*
- Local variables:
  - Location: FP + 1 + *id_offset*
  - Intermediate code:
    `push_reg` FP
    `push_const` 1 + *id_offset*
    `add`

Pushing Variable Address

- Parameters
  - Location: FP −1 − num_args + *id_offset*
  - Intermediate Code:
    ```
    push_reg FP
    push_const id_offset − num_args − 1
    add
    ```
- Variable access code in YACC
  - Push the variable address on the stack as above
  - When it becomes an expression, then it is an r-value and emit `fetch` operation which get the contents of the location, pop the address itself and push the contents on the stack
  - For l-values, leave the address as it is
  - When you reduce an assignment statement, use `assign`

Computation of id_offset

- We first include action code in YACC specification that computes the sizes and offsets of variables or record fields, and save them in the symbol table
  - Add offset and size to `struct decl`
  - Provide offset-stack in parallel to scope-stack to keep track of offsets of objects in the current scope
  - The top of offset-stack should contain the sum of sizes of objects that have already been declared in the current scope.
    This value is then the offset of the next object to be declared.
  - pushscope() starts off a new offset and initialize it to zero
  - popscope() just remove the current scope


int u;
int foo (int x, int y)
{
int z, w;
}
"w"
1
"z"
0
"y"
1
"x"
0
"y"
1
"x"
0
"foo"
FUNC
formals
rettype
"*return"
ID
"*return"
ID
TYPE
int
"u"
0
"int"
2
1
scopeatck & offsetstack
globalscope

Global Variables

- The `globalscope` pointer is maintained to check if a variable belongs to the global scope; if so the base pointer of its address is Lglob, otherwise FP
  - How do we distinguish between local variables and parameters?
- After all global variables have been processed, we know the size of the whole global data area, which can be used to declare Lglob. data *num* assembly directives
- In our homework, there are no global variables declared in the middle of file, so that you can declare Lglob. right after you process global variables at the file header

Processing Control Structures

if <expr> stmt_seq1 optelse
Compiling:

```
        else_label              = new_label();
        after_every_label    = new_label();
        <generate code for the expr>
        “branch-false else_label”
        <generate code for stmt_seq1>
        “jump after_every_label”
        <emit else_label>
        <generate code for optelse>
        <emit after_every_label>
```

YACC for Control Structures

```
stmt : if <expr>    { $<intval>$ = new_label();
                      printf("branch-false L%d₩n", $<intval>$);
                    }
       comp-stmt    { $<intval>$ = new_label();
                      printf("jump L%d₩n", $<intval>$);
                      printf("L%d: ₩n", $<intval>3);
                    }
       <optelse>    { printf("L%d: ₩n", $<intval>5); }
```