Runtime Organization

- We have covered the front-end phases: lexing, parsing, semantic analysis
- Now back-end: optimization, code generation
- Before that: we need to talk about runtime organization --- what do we need to generate. Later we will talk about how to generate it (code generation)

Management of runtime resources

- Correspondence between static (compile-time) and dynamic (run-time) structures
  - What is done by the compiler, and what is deferred to the runtime
- Storage organization

Execution of a program is initially under the control of the operating system

- The OS allocates space for a program
- Code is loaded into part of the space
- OS jumps to the entry point (e.g., "main")

Draw a picture of memory with "code" and "other space". Code is usually loaded at one end of the memory space (e.g., low or high end).

- Will draw memory as a rectangle with low address at bottom and high address at top
- Lines delimiting different areas of the memory
- Simplification: assume contiguous memory (need not be necessary)

- Other space = Data space
- Compiler is responsible for
  - Generating code
  - Orchestrating use of data area

Activations

- Two goals
  - Correctness
  - Speed
- Complications in code generation from trying to be fast as well as correct
- Over time, fairly elaborate structures have been developed on how to do code generation and runtime structures

Two assumptions:

- Execution is sequential; control moves from one point in a program to another in a well-defined order
  - Violated in face of concurrency
- When a procedure is called, control always returns to the point immediately after the call
  - Violated in catch/throw style exceptions (an exception may escape multiple procedures before it is caught)
  - Call/cc: call with current continuation
- Even such violating constructs depend on the ideas discussed here

- An invocation of procedure `P` is an *activation* of `P`
- The *lifetime* of an activation of `P` is
  - All the steps to execution `P`
  - Including all the steps in procedures `P` calls
- The *lifetime* of a variable `x` is the portion of execution in which `x` is defined
- Note that
  - Lifetime is a dynamic (run-time) concept
  - Scope is a static concept

- Observation
  - When P calls Q, then Q returns before P returns
- Lifetimes of procedure activations are properly nested
- Activation lifetimes can be depicted as a tree
- Consider an example:

```
int g() { return 1; }
int f() { return g(); }

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

  Draw the tree of activation lifetimes
  - The example shows nesting and the fact that different calls to the same procedure will have different activations
- Another example

```
int g() { return 1; }
int f(int x) { if (x == 0) then return g(); else return f(x- 1); }

int
main() {
return f(3);
}
```

  Draw the activation tree
- The activation tree depends on run-time behaviour (may be different for different inputs).
- **Since activations are properly nested, a stack can track the currently active activations**
  - The stack will not keep track of the entire activation tree
  - It will only keep track of the currently active activations
  - Work the first example showing the tree and the stack incrementally over runtime
- Let's look at the memory layout, where the first portion is occupied by the code. One of the important structures that goes in the "data area" is the stack of actie activations. The stack typically grows in one direction

Activation Records

What information to keep in an activation? That is what we will call the activation record.

- The information needed to manage one procedure activation is called an *activation record* (AR) or *frame*
- If procedure `F` calls procedure `G`, then `G`'s activation record contains a mix of info about `F` and `G`
- `F` is "suspended" until `G` completes, at which point `F` resumes
- `G's` AR contains information needed to
  - Complete execution of `G`
  - Resume execution of `F`
- Example

```
int g() { return 1; }
int f(int x) { if (x == 0) then return g(); else return f(x- 1); }

int
main() {
return f(3);
}
```

  AR: (1) result , (2) arguments , (3) control-link : pointer to the caller's activation , (4) return address

- Executing this program by hand, show the AR for each function (except main). There can be two return addresses for `f`, and show them as "*" and "**" respectively. These represent addresses of the instructions in memory.
- This stack is not as abstract as a general stack. This stack has an array-like layout, and hence compiler writers will often play tricks to exploit this fact that activation records are adjacent in memory.
- This is only one of many possible AR designs. e.g., many compilers don't use a control link, they rely on the fact that the stack is laid out linearly as an array. Similarly, the return address may be in a register (and not in the stack). The compiler determines the AR. Different compilers may employ different ARs. Some ARs are more efficient than others. Some ARs are easier to generate code for.

- The advantage of placing the result in the first word of the AR is that the caller can find the result at a fixed offset from its own activation (size of its AR + 1).
- Can divide responsibility between caller/callee differently
- Real compilers make careful tradeoffs on which part of the activation frame should be in registers and which part in memory
- The compiler must determine, at compile-time, the layout of activation records and generate code that correctly accesses locations in the activation record. Thus AR layout and the code generator must be designed together!

Globals and Heap

- A global variable's lifetime transcends all procedure lifetimes. It has a global lifetime
  - Can't store a global in an activation record
- Global s are assigned a fixed address once
  - Variables with fixed address are "statically allocated"
- Depending on the language, there may be other statically allocated values
- Memory layout: Code, static data, stack (grows downwards)

Heap objects

- A value that outlives the procedure that creates it cannot be kept in the AR
- e.g., new A should survive deallocation of the current procedure's AR.

Usually

- The code area contains object code: fixed size and read only
- Static area contains data (not code) with fixed addresses: fixed size, may be readable or writeable
- Stack contains an AR for each currently active procedure. Each AR usually fixed size, contains locals
- Heap contains all other data: In C, heap is managed through *malloc* and *free*

Both heap and stack grow. Must take care they don't grow into each other. Simple solution: put them at opposite ends of memory and let them grow towards each other. If the two regions start touching each other, then the program is out of memory (or it may request more memory, etc.). This design allows different programs to use these different areas as they see fit, e.g., some programs may have large stacks, other may have large static data regions.

Alignment

Low-level but important detail that every compiler writer needs to be aware of.

- Most modern machines are 32 or 64 bit
  - 8 bits in a byte
  - 4 or 8 bytes in a word
  - Machines are either byte or word addressable
- Data is *word aligned* if it begins at a word boundary
- Most machines have some alignment restrictions
  - E.g., undefined behaviour if access misaligned data

    - Or dramatic performance penalties for poor alignment (can be up to 10x depending on the machine)
- Example: a string "Hello" takes 5 characters (without a terminating '\0').
    - Show an array of bytes with word-boundaries marked with darker lines
    - To word align next word, add 3 "padding" characters to the string
    - The padding is not part of the string, it's just unused memory

Stack machines

Begin talking about code generation. Stack machines are the simplest model of code generation

In a stack machine

- Only storage is a stack
- An instruction

```
r = F(a1, ..., an):
```

    - Pop the top `n` words off the stack
    - Apply `f`
    - Push the result on the stack
- Example: two instructions `push i` and `add`
- Location of the operands/result is not explictly stated
    - Always the top of the stack
- In contrast to a *register machine*
    - `add` instead of `add r1,r2,r3`
    - More compact programs
    - One reason that Java bytecode uses stack evaluation model
        - In early days, Java was meant for mobility and memory was scarce
- There is an intermediate point between a pure stack machine and a pure register machine
- An *n-register stack machine*: Conceptually, keep the top *n* locations of the stack in the registers
- A one-register stack machine: the one register is called an *accumulator*.
- Advantage of a one-register stack machine:
    - In a pure stack machine
        - An `add` does 3 memory operations (load two arguments and store one result
    - In a one-register stack machine
        - The add does `acc <-- acc + top_of_stack`
        - Just one memory read
- Consider an expression `op(e1,...,en)`
    - Note `e1,..,en` are subexpressions
- For each `ei`, compute `ei` with the result in the accumulator; push the result on the stack
- For `en`, just evaluate into accumulator, do not push on stack.
- Evaluate `op` using values on stack and `en` in accumulator to produce result in accumulator
- e.g., `7+5`: push 7; acc <-- 5; acc <-- acc + [top]
- Invariant: after evaluating an expression `e`, the accumulator holds the value of `e` and the stack is unchanged
    - Important property: expression evaluation preserves the stack
- Another example: 3 + (7 + 5)
    - First evaluate 3
    - Save 3
    - Evaluate 7
    - Save 7
    - Evaluate 5
    - Add
    - Add
- Another important property: the expressions are being evaluated left-to-right. i.e., the evaluation order is left to right which also determines the order of the operands on the stack.

- Some code generation strategies may depend on the evaluation order like this one
- Others may not, e.g., where we had named identifiers (e.g., registers) storing the result for each intermediate expression