CSC 425 - Principles of Compiler Design I

Run-time Environments

Status

- We have covered the front-end phases
  - Lexical analysis
  - Parsing
  - Semantic analysis
- Next are the back-end phases
  - Code generation
  - Optimization
  - Register allocation
  - Instruction scheduling
- In this course, we will examine code generation

Run-time Environments

- Before discussing code generation, we need to understand what we are trying to generate
- There are a number of standard techniques for structuring executable code that are widely used.

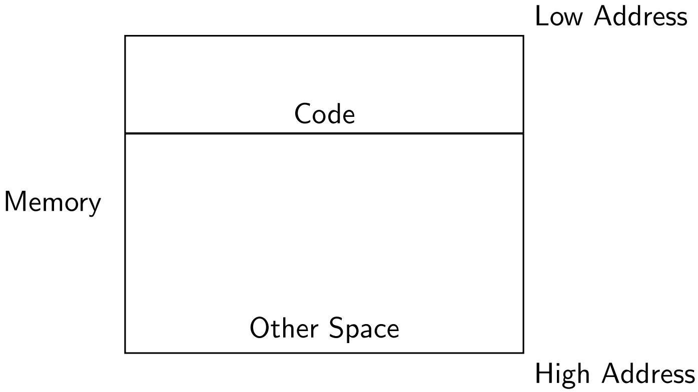
Outline

- Management of run-time resources
- Correspondence between static (compile-time) and dynamic (run-time) structures
- Storage organization

Run-time Resources

- Execution of a program is initially under the control of the operating system (OS)
- When a program is invoked:
  - The OS allocates space for the program
  - The code is loaded into part of this space
  - The OS jumps to the entry point of the program, that is, to the beginning of the "main" function

Memory Layout

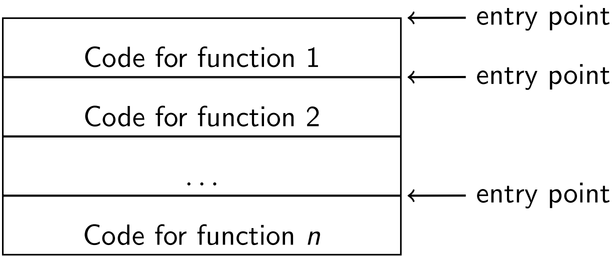


Notes

- By tradition, pictures of run-time memory organization have:
  - Low addresses at the top
  - High addresses at the bottom
  - Lines delimiting areas for different kinds of data
- These pictures are simplifications
  - For example, not all memory need be contiguous

Organization of Code Space

- Usually, code is generated one function at a time; the code area has the form:



- Careful layout of code within a function can improve instruction-cache utilization and give better performance
- Careful attention in the order in which functions are processed can also improve instruction-cache utilization

What is Other Space?

- Holds all data needed for the program's execution
- Other space is data space
- Compiler is responsible for:
  - generating code
  - orchestrating the use of the data area

Code Generation Goals

- Two main goals:
  - Correctness
  - Speed
- Most complications in code generation come from trying to be fast as well as correct

Assumptions about Flow of Control

- (1) Execution is sequential; at each step, control is at some specific program point and moves from one point to another in a well-defined order
- (2) When a procedure is called, control eventually returns to the point immediately following the place where the call was made
- Question: do these assumptions always hold?

Language Issues that Affect the Compiler

- Can procedures be recursive?
- What happens to the values of the locals on return from a procedure?
- Can a procedure refer to non-local variables?
- How are parameters to a procedure passed?
- Can procedures be passed as parameters?
- Can procedures be returned as results?
- Can storage be allocated dynamically under program control?
- Must storage be deallocated explicitly?

Activations

- An invocation of procedure P is an activation of P
- The lifetime of an activation of P is:
  - All the steps to execute P
  - Including all the steps in procedures that P calls

Lifetimes of Variables

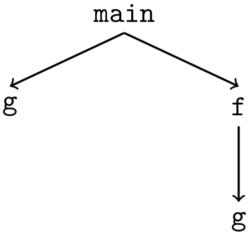
- The lifetime of a variable x is the portion of execution in which x is defined
- Note that:
  - Lifetime is a dynamic (run-time) concept
  - Scope is (usually) a static concept

Activation Trees

- Assumption (2) requires that when P calls Q, then Q returns before P does
- Lifetimes of procedure activations are thus either disjoint or properly nested
- Activation lifetimes can be depicted as a tree

Example 1

```
let
  function g(): int = (42)
  function f(): int = g()
  function main() = (g(); f();)
in
  main()
end
```



Example 2

```
let
  function g(): int = (42)
  function f(x:int): int =
    if x = 0 then g()
    else f(x-1)
  function main() = f(3)
in
  main()
end
```

- What is the activation tree for this example?

Notes

- The activation tree depends on run-time behavior
- The activation tree may be different for every program input
- Since activations are properly nested, a (control) stack can track currently active procedures
  - push information about an activation at the procedure entry
  - pop the information when the activation ends, that is, at the return from the call

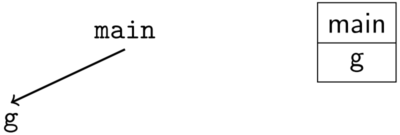
Example

```
let
  function g(): int = (42)
  function f(): int = g()
  function main() = (g(); f();)
in
  main()
end
```

`main` | main |

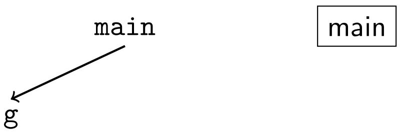
Example

```
let
  function g(): int = (42)
  function f(): int = g()
  function main() = (g(); f();)
in
  main()
end
```



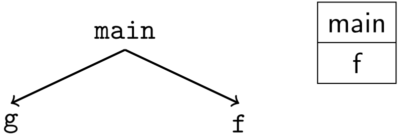
Example

```
let
  function g(): int = (42)
  function f(): int = g()
  function main() = (g(); f();)
in
  main()
end
```



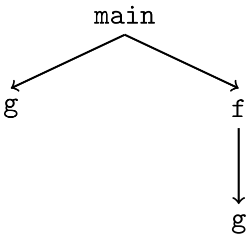
Example

```
let
  function g(): int = (42)
  function f(): int = g()
  function main() = (g(); f();)
in
  main()
end
```



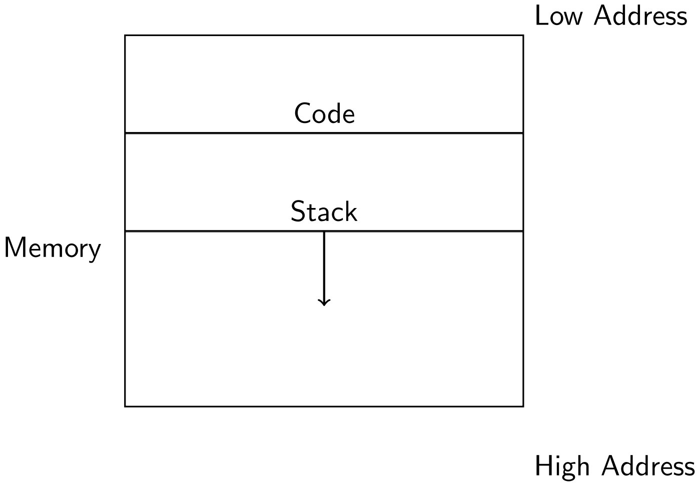
Example

```
let
  function g(): int = (42)
  function f(): int = g()
  function main() = (f(); g())
in
  main()
end
```



main
f
g

Revised Memory Layout



Activation Records

- The information needed to manage a single procedure activation is called an activation record (AR) or a stack frame
- If a procedure F calls G, then G's activation record contains a mix of information about F and G

What is in G's AR when F calls G?

- F is "suspended" until G completes, at which point F resumes.
- G's AR contains information needed to resume execution of F
- G's AR may also contain:
  - G's return value (needed by F)
  - Actual parameters to G (supplied by F)
  - Space for G's local variables

The Contents of a Typical AR for G

- Space for G's return value
- Actual parameters
- (Optional) Control link, a pointer to the previous activation record
- (Optional) Access link for access to non-local names; points to the AR of the function that statically contains G
- Machine status prior to calling G
  - return address, values of registers, and program counter
  - local variables
- Other temporary values used during evaluation

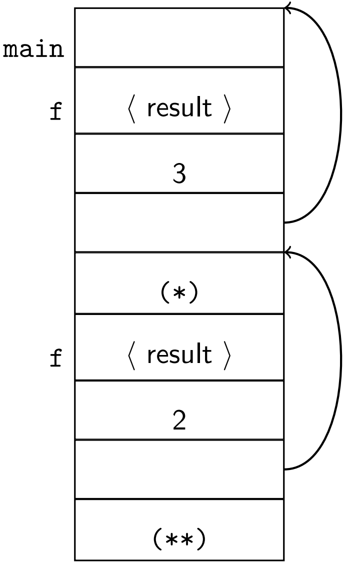
Example 2

```
let
  function g(): int = (42)
  function f(x:int): int =
    if x = 0 then g()
    else f(x-1) (**)
  function main() = f(3) (*)
in
  main()
end
```

- AR for `f`

result
argument
control link
return address

Stack After Two Calls to `f`



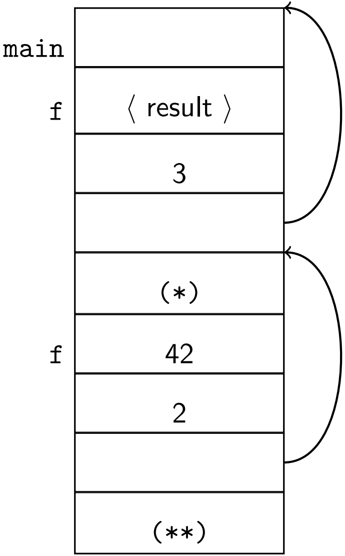
Notes

- `main` has no argument or local variables and returns no result; its AR is not interesting
- `(*)` and `(**)` are return addresses (continuation points) of the invocations of `f`
- The return address is where execution resumes after a procedure call finishes
- This is only one of many possible AR designs

The Main Point

- The compiler must determine, at compile-time, the layout of activation records and generate code that correctly accesses locations in the activation record
- Key point: the AR layout and the code generator must be designed together

Stack After Second Call to f Returns



Discussion

- The advantage of placing the return value first in an AR is that the caller can find it at a fixed offset from its own AR
- There is nothing special about this run-time organization
  - Can rearrange order of frame elements
  - Can divide caller/callee responsibilities
  - An organization is better if it improves execution speed or simplifies code generation
- It is beneficial for a compiler to hold as much of the AR as possible in registers

Storage Allocation Strategies for Activation Records

- Static Allocation (Fortran 77)
  - Storage for all data objects is laid out at compile time
  - Can only be used if the size of data objects and constraints on their position in memory can be resolved at compile time
  - Recursive procedures are restricted since all activations of a procedure must share the same locations for local names

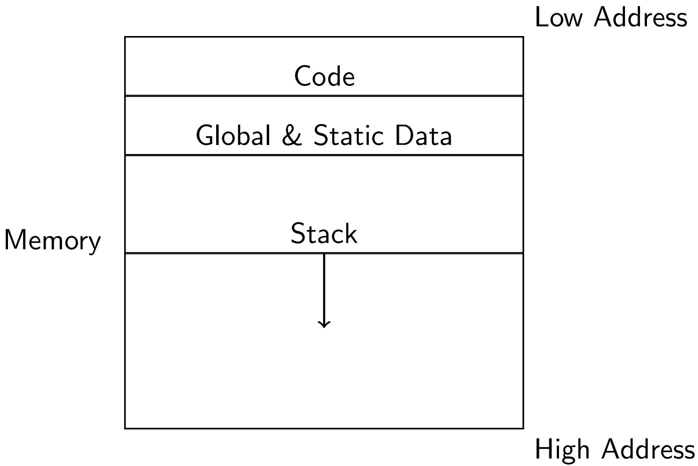
Storage Allocation Strategies for Activation Records

- Stack Allocation (Pascal, C)
  - Storage is organized as a stack
  - Activation record pushed when activation begins and popped when it ends
  - Cannot be used if the values of local names must be retained when the evaluation ends or if the called invocation outlives the caller
- Heap Allocation (Lisp, ML)
  - Activation records may be allocated and deallocated in any order
  - Some form of garbage collection is needed to reclaim free space

Globals

- All references to a global variable point to the same object; a global cannot be stored in an activation record
- Globals are assigned a fixed address once; variables with fixed addresses are "statically allocated"
- Depending on the language, there may be other statically allocated values

Memory Layout with Static Data



Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the activation record
- Languages with dynamically allocated data use a heap to store dynamic data

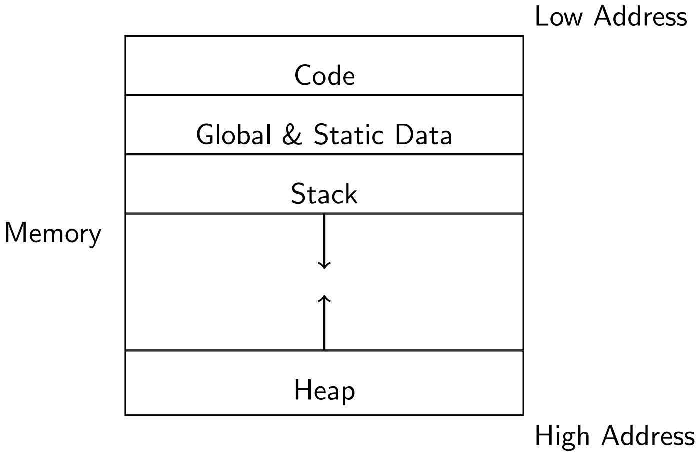
Review of Runtime Organization

- The code area contains object code; for most languages, fixed size and read only
- The static area contains data (not code) with fixed addresses; fixed size my be readable or writable
- The stack contains an AR for each currently active procedure; each AR usually has a fixed size
- The heap contains all other data

Notes

- Both the heap and the stack grow
- We must take care so that they do not grow into each other
- Solution: start the heap and the stack at opposite ends of memory and let them grow towards each other

Memory Layout with Heap



Data Layout

- Low-level details of computer architecture are important in laying out data for correct code and maximum performance
- One of these concerns is data alignment
  - most modern machines are 32 or 64 bit; this defines a word
  - data is word-aligned if it begins at a word boundary
  - Most machines have some alignment restrictions