Run-time Environments

Adapted from Zhendong Su’s slides
Status

• We have covered the front-end phases
  - Lexical analysis
  - Parsing
  - Semantic analysis

• Next are the back-end phases
  - Optimization
  - Code generation

• We’ll do code generation first . . .
Run-time environments

• Before discussing code generation, we need to understand what we are trying to generate
• Place data (global, local, dynamically allocated at runtime)
• Translated instructions
  • Accessing data
  • Arithmetic
  • Control flow (calling relations)
...

• There are a number of standard techniques for structuring executable code that are widely used
Overview of Runtime Organization

- **Code**
  - For most languages, fixed size and read only

- **Static Data**
  - Accessible throughout the program lifetime
  - Fixed size, may be readable or writable

- **Stack – activation record**
  - Destroyed when the lifetime of an AR ends
  - Contain anything whose lifetime is the same as the procedural call
  - Fixed size
  - Try hold (especially, parameter and results) in the register

- **Heap – dynamically allocated memory**
  - Developers manage (e.g., C)
  - Garbage collector
Outline

• Management of run-time resources

• Correspondence between
  - static (compile-time) structures, and
  - dynamic (run-time) structures

• Storage organization
Run-time Resources

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

• When a program is invoked
  - The OS allocates space (virtual memory) for the program
  - The code is loaded into part of the space
    This area contains the generated target code for all procedures in the program. The size of this can be determined statically by the compiler/linker
  - The OS jumps to the entry point (i.e., “main”)
Memory Layout

Memory

Code

Other Space

Low Address

High Address
Notes

• Our pictures of machine organization have
  - Low address at the top
  - High address at the bottom
  - Lines delimiting areas for different kinds of data

• These pictures are simplifications
  - E.g., not all memory need be contiguous

• In some textbooks lower addresses are at bottom
What is Other Space?

• Holds all data for the program

  Other Space = Data Space

• Compiler is responsible for
  - Generating code
  - Orchestrating use of the data area
Code Generation Goals

• Two goals
  - Correctness
    - prove the correctness of the compiler
    - conservativeness
  - Speed
    - Optimizing building process
    - Avoid recompilation

• Most complications in code generation come from trying to be fast as well as correct
Assumptions about Execution

1. Execution is sequential; control moves from one point in a program to another in a well-defined order

2. When a procedure is called, control eventually returns to the point immediately after the call

Do these assumptions always hold?
Activations

• An invocation of proc. $P$ 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 a static concept
Activation Trees

• Assumption (2) requires that when $P$ calls $Q$, then $Q$ returns before $P$ does

• Lifetimes of procedure activations are properly nested

• Activation lifetimes can be depicted as a tree
Example

Class Main {
    g() : Int { 1 };
    f(): Int { g() };
    main(): Int {{ g(); f(); }};
}

Main
  ┌─ g
  │  └─ f
  │      ┌─ g
Example

Class Main {
    g() : Int { 1 };
    f() : Int { g() };
    main() : Int {{ g(); f(); }};
}

Main

Stack

Main
Example

Class Main {
    g(): Int { 1 };  
    f(): Int { g() };  
    main(): Int {{ g(); f(); }};
}

Stack
  Main
  g

Main
  g
Example

Class Main {
    g(): Int { 1 };
    f(): Int { g() };
    main(): Int {{ g(); f(); }};
}
Example

Class Main {
    g(): Int { 1 };
    f(): Int { g() };
    main(): Int {{ g(); f(); }};
}

Stack

Main

<table>
<thead>
<tr>
<th>Main</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
</tr>
<tr>
<td>g</td>
</tr>
</tbody>
</table>

19
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 stack can track currently active procedures
Example 2

Class Main {
    g() : Int { 1 };
    f(x: Int): Int { if x = 0 then g() else f(x - 1) fi};
    main(): Int {{f(3); }};
}

What is the activation tree for this example?
Activation Records

- On many machines, the stack starts at high-addresses and grows towards lower addresses.

- The information needed to manage one procedure activation is called an activation record (AR) or frame.

- If procedure $F$ calls $G$, then $G$’s activation record contains a mix of info 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
  - Actual parameters to G (supplied by F)
  - G’s return value (needed by F)
  - Space for G’s local variables
The Contents of a Typical AR for $G$

- Space for $G$’s return value
- Actual parameters
- Pointer to the previous activation record
  - The *control link*; points to AR of caller of $G$
- Machine status prior to calling $G$
  - Contents of registers & program counter
  - Local variables
- Local variables
Example 2, Revisited

Class Main {
    g() : Int { 1 };
    f(x: Int): Int { if x = 0 then g() else f(x - 1)(**)fi};
    main(): Int {{f(3); (*) }};}

AR for f:

| return address |
| control link   |
| argument       |
| result         |
Stack After Two Calls to f

Stack:

- **
- result
- (*)
- result
- 3
- 2
- f
- main
Notes

• **main** has no argument or local variables and its result is never used; its AR is uninteresting
• (*) and (**) are return addresses 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
  - Would also work for C, Pascal, FORTRAN, etc.
Local variables

Local variables offsets are calculated depending on its size. These offsets can be computed relative to the start of the stack frame, and can be used to specify the layout of local data in the stack frame.
Return value

• The advantage of placing the return value 1st in a frame is that the caller can find it at a fixed offset from its own frame

• There is nothing magic about this organization
  - Can rearrange order of frame elements
  - Can divide caller/callee responsibilities differently
  - An organization is better if it improves execution speed or simplifies code generation
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.

Thus, the AR layout and the code generator must be designed together!
Compiler generates the code to manage AR

How would this happen (push and pop the activation record)?

What makes this happen is known as calling sequence (how to implement a procedure call).

A calling sequence allocates an activation record and enters information into its fields (push the activation record).

On the opposite of the calling sequence is the return sequence.

Return sequence restores the state of the machine so that the calling procedure can continue execution.
AR needs to be accessed fast

• Real compilers hold as much of the frame as possible in registers
  - Especially the method result and arguments
Globals

• All references to a global variable point to the same object
  - Can’t store a global in an activation record

• Globals are assigned a fixed address once
  - Variables with fixed address are “statically allocated”
  - Size can be determined statically at compile time. Static variables are mapped to offsets in the static data area.

• Depending on the language, there may be other statically allocated values
Memory Layout with Static Data

- Memory
  - Code (Low Address)
  - Static Data
  - Stack (High Address)
Variables with Statically Unknown Size

How do we allocate storage for a variable/field with statically unknown size e.g. an array or an adt (abstract Data Type).

By allocating storage separately (on the heap or on top of the AR on the stack) and storing a pointer to the storage in the AR.
Heap Storage

• A value that outlives the procedure that creates it cannot be kept in the AR

```java
method foo() { new Bar }
```

The `Bar` value must survive deallocation of `foo`’s AR

• Languages with dynamically allocated data use a `heap` to store dynamic data
Notes

• Both the heap and the stack grow

• Must take care that they don’t grow into each other

• Solution: start heap and stack at opposite ends of memory and let the grow towards each other
Memory Layout with Heap

Memory

Code

Static Data

Heap

Stack

Low Address

High Address
Memory layout is language specific

Fortran - Static
Pascal - Stack/heap
C - Stack/Heap
Data Layout

• Low-level details of machine architecture are important in laying out data for correct code and maximum performance

• Chief among these concerns is alignment
Alignment

- Modern machines are 32 bit or 64 bit
  - 8 bits in a byte
  - 4 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
  - Or performance penalties for poor alignment
Alignment (Cont.)

• Example: A string
  
  “Hello”
  
  Takes 5 characters (without a terminating \0)

• To word align next datum, add 3 “padding” characters to the string

• The padding is not part of the string, it’s just unused memory