COMP 520 - Compilers

Lecture 13 (Tue Mar 19)

Run-time organization

• Reading
  – Chapter 6, section 6.1 - 6.5 (pp 173 - 219)
Where are we?

- We have completed discussion of the compiler “front-end”
  - scanning
  - parsing
  - contextual analysis

- Semantics of a program
  - defined in terms of its decorated AST
  - could be executed by an AST “interpreter”

- Next is “back end”
  - code generation
  - translate the AST semantics to operations in a target machine

- Approach
  - first try to understand the target machine
    - (lower-level) storage and execution model connects front end to back end
  - then study the translation
    - challenge: what is done at compile time and what is done at run time?
Topics in code generation (multiple lectures)

• **Overview of run-time issues**
  – memory model and organization
  – representation of values
  – evaluation of expressions
  – procedures and functions
    • activation records
    • non-local variable access
    • parameter passing
  – runtime resources and management system

• **Run-time organization for object-oriented languages**
  – creation of and access to objects
  – inheritance and virtual methods
Target machine model

- Machine model
  - physical
    - MIPS as studied in computer organization class
    - Intel Architecture (IA-32 or x86-64) or ARM
  - abstract
    - Triangle Abstract Machine (TAM) from our text
    - Java Virtual Machine (JVM)

- Application Binary Interface (ABI)
  - a set of conventions
    - how values are represented (integers, floating point values, byte order)
    - where values are stored (stack, heap)
    - basic runtime facilities (memory allocation, garbage collection)
  - examples
    - MIPS
    - TAM, JVM, .NET Microsoft Common Language Runtime (CLR)
ABI: MIPS memory organization

- ABI defines fixed addresses and usage conventions

- Key areas
  - Reserved
    - for use by operating system
  - Text segment
    - compiler-generated program is loaded here
  - Stack segment
    - procedure invocation and expression evaluation stack
      - expands downwards
  - Data segment
    - static constants and variables are placed at the bottom
      - their locations are known by the compiler
    - dynamically allocated values are allocated beyond this part
      - their locations cannot be predicted by the compiler
      - expands upwards
      - memory for deleted values can be reused
TAM memory organization

- Two separate memories
  - Code store
    - compiler-generated program is loaded into code segment
    - predefined runtime functions are located in the primitive segment
    - TAM can not write into code store
  - Data store
    - static constants and variables are loaded into global segment
    - procedure invocation and expression evaluation use execution stack
      - expands downwards
    - dynamically allocated values are allocated on the heap
      - expands upwards
      - memory for deleted values can be reused
- ABI defines fixed addresses and usage conventions
  - various locations in memories are accessed relative to machine registers (CB, SB, HT, etc.)
Representation of values in memory

- Values of a given type must have a well-defined representation
  - ex: double, int, char, boolean (predefined)
  - typically represented as a 64-bit, 32-bit, 16-bit, or 8-bit binary values
  - chosen to match underlying machine hardware
  - attractive for compiler if all values of a type have the same size

- For aggregate values (records, arrays)
  - compiler must know how to access components
  - aggregate values may have static size or dynamic size
  - indirect representation of dynamic size values
    - fixed sized pointer to location of dynamic sized value
Execution model: Stack machine

- **Stack machine**
  - all operations take place at stack top
  - implementations
    - Burroughs 5500 (hardware interpreter)
    - TAM (software interpreter)

- **Stack operations**
  
<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORE $addr$</td>
<td>pop value off stack top and store at address $addr$</td>
</tr>
<tr>
<td>LOAD $addr$</td>
<td>push value at address $addr$ onto top of stack</td>
</tr>
<tr>
<td>LOADL $c$</td>
<td>push literal value $c$ onto top of stack</td>
</tr>
<tr>
<td>ADD, SUB, …</td>
<td>perform operation at stack top: pop operands, push result</td>
</tr>
<tr>
<td>CALL $foo$</td>
<td>execute $foo$: $foo$ receives its arguments at the stack top, consumes them, and returns its result at stack top</td>
</tr>
</tbody>
</table>
Expression evaluation on stack machines

- Given expression AST, construct code for expression evaluation on stack
  - via postorder traversal of AST
    - generate code for children of node (l to r) then generate code for node
    - leaf action: load value
    - non-leaf action: perform operation
- ex:

  \[ x + 2 + 3 \times x \]

AST

Stack machine code

- \[ LOAD \ x \]
- \[ LOADL \ 2 \]
- \[ ADD \]
- \[ LOADL \ 3 \]
- \[ LOAD \ x \]
- \[ MUL \]
- \[ ADD \]
Triangle code generation

- Triangle Abstract Machine (TAM)
  - Implements a stack machine

Triangle

```plaintext
let
  var n: Integer
  var c: Char
in
  begin
    c := '&';
    n := n + 1
  end
```

TAM instructions

```
PUSH  2      // space for n, c
LOADL 38     // ascii code ‘&’
STORE 1[SB]  // store in c
LOAD  0[SB]  // load n
LOADL 1
CALL  add
STORE 0[SB]  // update n
POP   2      // delete space
HALT
```
Execution model: Register machine

- **Register machine**
  - all operations take place in limited collection of registers
  - implementations
    - RISC architectures, such as MIPS

- **Register machine operations**
  - STORE \( r, \text{addr} \) store value in register \( r \) at address \( \text{addr} \)
  - LOAD \( r, \text{addr} \) load value at address \( \text{addr} \) into register \( r \)
  - LOADL \( r, c \) load literal value \( c \) into register \( r \)
  - ADD \( r3, r1, r2 \) \( r3 = r1 + r2 \) (\( r1, r2, r3 \) registers)
  - SUB \( r3, r1, r2 \) \( r3 = r1 - r2 \) (\( r1, r2, r3 \) registers)
  ...

- A register machine can simulate a stack machine
  - part of Application Binary Interface (ABI),
    - e.g. a fixed register is designated as the stack pointer
  - sometimes supported in hardware (e.g. ia-32 floating point, sparc)
Expression evaluation on register machines

• Naive strategy
  – simulate stack machine
  – load values at stack top into registers for operations, and save result back onto stack

• Better strategy
  – some values can be kept in registers rather than on the stack
    • ex: $x + 2 + 3 \ast x$
    • finding optimal solution with fixed number of registers is NP-hard
Procedures and functions

• The procedure and function abstraction
  – allows us to build large programs and reuse code
  – invocation:
    • call from within a statement or expression
    • return (possibly with result) to point of call
  – local variables have separate instantiations for each invocation
    • enables recursive invocation

• Implementation of procedure and function invocation
  – a convention
  – machine dependent and possibly hardware assisted
  – division of responsibility between caller and callee
    • caller: set up arguments and space for result
    • callee: create space for locals, execute body, clean up space, and return
  – debuggers rely on the convention being followed
Definitions (Triangle)

```
func fib(n : Integer) : Integer ~
if n <= 2
  then 1
else fib(n-1) + fib(n-2)
```

- **function definition**
- **function name**
- **parameter**
- **function body**
- **(recursive) function call**
- **argument**
More definitions

- **The lifetime** of an activation of procedure $P$ is:
  - all steps taken from the start of execution of $P$ until its return to the point of call
  - includes the lifetimes of procedures that $P$ calls

- **Dynamic concept**
  - may depend on parameters

- **Important fact**
  - Given two activations A and B, their lifetimes are either disjoint or properly nested
Activation trees

Fibonacci function

```plaintext
func f (n : Integer) : Integer ~
  if n <= 2
    then 1
    else f(n-1) + f(n-2)
```

• Depends on runtime behavior

• May be different for each program input
Execution stack and activation records

- Activation tree suggests the use of a stack to keep track of currently active procedures
  - superficially similar to nested scope
  - but activation tree is dynamic and scope is static

- Stack usually laid out in contiguous storage
  - each entry on the stack is a procedure or function activation record
    - Information needed to manage one procedure activation
    - In our text, an activation record is known as a “frame”

- If F calls G, then G’s activation record contains
  - information to resume execution of F
  - arguments from F to G (often viewed as part of F)
  - local variables of G
  - result of G to F (often viewed as part of F)
Components of a frame

• Register conventions
  – the frame pointer LB (FP in MIPS) contains address of start of the frame
  – the stack pointer ST (SP in MIPS) contains address of the end of the frame and is the top of the execution stack

Dynamic link
  – points at start of frame of calling procedure

Static Link
  – points at start of AR of statically enclosing procedure (more later)

Return address
  – address to resume execution of caller
Who’s Doing What and When?

- Code: Generated by compiler
- Static data: Space reserved by compiler
- Stack: Activation records generated at run time
- Heap: Managed by run-time dynamic allocation
- Free memory
Recall TAM memory organization

- **Two separate memories**
  - **Code store**
    - compiler-generated program is loaded into code segment
    - predefined runtime functions are located in the primitive segment
    - TAM can not write into code store
  - **Data store**
    - static constants and variables are loaded into global segment
    - procedure invocation and expression evaluation uses execution stack
      - expands downwards
    - dynamically allocated values are allocated on the heap
      - expands upwards
      - memory for deleted values can be reused
- **ABI defines fixed addresses and usage conventions**
  - various locations in memories are accessed relative to machine registers (CB, SB, HT, etc.)
Procedures and functions

• Run-time access to variables
  – local variables
  – global variables
  – non-local, non-global variables (Triangle)
  – object instances and members (miniJava)

• Procedures and functions
  – frame maintenance
  – parameter passing
  – result value

• Heap-allocated variables
  – runtime heap management
Runtime access to local variables

- The value of a local variable may be stored
  - in an activation record on the execution stack
    - if the variable lifetime = procedure lifetime
  - in the heap
    - if the variable lifetime can exceed procedure lifetime
      - E.g. a reference returned from a procedure

- Two models of scoped variables
  - local/global scope (C)
    - all procedures (logically) declared at global scope
    - variables are declared at global scope or at local scope
    - variable references are local or global
      - where do we find the value at runtime? EASY!
  - nested procedures (Pascal, Triangle)
    - procedure definitions may be nested
    - variables are declared in procedures (including a special “global” procedure)
    - variable references may refer to a declaration in a surrounding procedure
      - where do we find the value at runtime? HARD!
Local/global scope: access to variables

- **A variable x can be declared**
  - at global scope
    - the compiler knows $d_G(x)$, the offset of x relative to the stack base SB
    - to get the value of x at stack top in TAM
      - LOAD $d_G(x)[SB]$
    - to store value at stack top into x
      - STORE $d_G(x)[SB]$
  - at local scope in procedure P
    - the variable is allocated in the frame (activation record) for P and is available only while P is executing.
    - the compiler knows $d(x)$, the offset of x relative to LB
    - to get the value of x at stack top
      - LOAD $d(x)[LB]$
    - to store value at stack top into x
      - STORE $d(x)[LB]$
Object access (preview)

• Classes
  
  ```java
  class A {int x; void p(){x = 3;}}
  ```
  
  – Information known to the compiler
    • class A:  \( S_A \) = size of class A (# fields)
    • field x:  \( d_x \) = displacement of field x within A

• Objects
  
  – instances are created on the heap
    ```java
    A a = new A();
    ```
  
  – access to members
    ```java
    a.x = 2;
    a.p();
    ```

mJAM runtime layout

object instance in heap

activation record on stack
Local/global scope: frame maintenance

• Procedure declaration
  ```
  int p(int x, int y) { return x+y; }
  ```

• Procedure call
  ```
  int x = p(3,4) + 1;
  ```

• Steps taken at point of call
  1. evaluate each argument expression at stack top
     • one value on the stack for each argument
  2. create new frame at stack top
     • space for dynamic link and return address (static link unused)
  3. Jump to procedure at appropriate address in instruction store

• Steps taken by the callee
  – save caller LB into dynamic link
  – save caller return address
  – set LB to start of current frame
  – execute body
  – restore LB of caller
  – restore ST of caller, popping parameters of caller and pushing result from callee
  – return to caller at RA
Nested procedures and non-local variables (Triangle)

```
let
  var a: Integer;
  proc F(b: Integer) ~
    let
      var b2: Integer;
      proc G(c: Integer) ~
        begin
          ... a, b, b2, c,
          F( ... ), G( ... )
        end
    in
      ... a, b, b2, F( ... ), G( ... )
  in
    ... a, F( ... )
```
Nested procedures and non-local variables (Triangle)

let

var a: Integer;
proc F(b: Integer) ~

let

var b2: Integer;
proc G(c: Integer) ~

begin

... a, b, b2, c,
F( ... ), G( ... )
end

in

... a, b, b2, F( ... ), G( ... )

in

... a, F( ... )
Nested procedures: access to variables

- **Procedure nesting level**
  - defined as the number of enclosing procedure or function declarations at a given point in a program

- **Given a reference to a variable** \( x \)
  - Declaration level \( d_x \)
    - procedure nesting level at point of declaration of \( x \)
  - Reference level \( r_x \)
    - procedure nesting level at reference of \( x \)
  - \( r_x \geq d_x \)

- **To find value of** \( x \) **at reference level** \( r_x \) **in TAM**
  - assume \( x \) is stored at offset \( h \) in frame of declaring procedure
  ```
  LOAD h[LB] // if \( r_x - d_x = 0 \)
  LOAD h[L1] // if \( r_x - d_x = 1 \)
  LOAD h[L2] // if \( r_x - d_x = 2 \)
  LOAD h[L3] // if \( r_x - d_x = 3 \)
  ...
  ```
  ```
  L1 = LOAD 0[LB] (the static link)
  L2 = { LOAD 0[LB]; LOADI }
  L3 = { LOAD 0[LB]; LOADI; LOADI }
  ...
Nested Procedures: frame maintenance

- At procedure call
  - call P occurs at reference level $r_P$
  - P is declared at declaration level $d_P$
  - $r_P - d_P \geq 0$

- Check some examples
  - F calls F, F calls G, G calls F
  - how do we set the dynamic link? how do we set the static link?

- Steps taken by the caller
  - establish static link at start of frame (current stack top)

- Steps taken by the callee
  - save caller LB into dynamic link
  - execute body
  - restore LB of caller
  - restore ST of caller
  - pop caller arguments off stack
  - push result on the stack
  - return to caller
Parameter passing mechanisms

- **Definition**
  - argument
    - passed into a procedure or function
  - parameter
    - stands for something passed into a procedure or function

- **Questions**
  - when are arguments of function calls evaluated?
    - most languages evaluate arguments at the point of call
  - to what are the parameters bound?
    - values?
    - addresses?
    - functions?
Call-by-value

- Frequently used (e.g. C, Java)
  - ex
    
    let
    
    proc g(x: Integer) ~ x := x + 1
    
    var y: Integer
    
    in
    
    y := 5; g(y); print(y)

  - implementation
    - argument is evaluated at stacktop (= value of y) by caller
    - callee parameter x is directly before activation record of g
    - modifications to x have no effect on y, because argument is popped on return by callee

- Call-by-value-result
  - x is copied back into y on termination of g
Call-by-reference

• the address is passed instead of a value (e.g. Pascal var parameter)
  – ex
    
    let
    proc g(var x: Integer) ~ x := x + 1
    var y: Integer
    in
    y := 5; g(y); print(y)
  
  – implementation
    • argument must be a variable
    • argument is evaluated at stacktop (= address of y) by caller
    • callee parameter x is directly before activation record of g
    • a reference to x requires dereference of the pointer
    • a change to x changes y
  – why do this?

• Aliasing
  – two parameters may refer to the same location or a parameter may refer to the same location as a global variable
  – is this a problem?
Values that live on the heap

- need to be allocated and deallocated at run-time

<table>
<thead>
<tr>
<th>language</th>
<th>allocation</th>
<th>deallocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>malloc</td>
<td>free</td>
</tr>
<tr>
<td>C++</td>
<td>new</td>
<td>delete</td>
</tr>
<tr>
<td>Java</td>
<td>new</td>
<td>(garbage collection)</td>
</tr>
<tr>
<td>Matlab</td>
<td>(implicit)</td>
<td>(reference counting)</td>
</tr>
</tbody>
</table>

- values on the heap are always passed as a reference
  - for performance reasons
  - but can lead to extensive aliasing
Run-time API

• alloc($k$)
  – locates a block of at least $k$ bytes in free space pool, removes it from pool, returns its address

• free($p$)
  – places the block pointed to by $p$ back in free space pool
  – Not needed if unused storage is automatically reclaimed
Issues in Heap Management

• Wasted space
  – If \textit{alloc} returns blocks larger than requested, excess space is wasted

• Fragmentation
  – After a series of \textit{alloc/free} commands, free space pool becomes fragmented, preventing allocation of large blocks

• Speed
  – \textit{alloc} and \textit{free} should be inexpensive