COMP 181

Lecture 13

Procedures and methods

October 17, 2006

Midterm

- Scanners
  - Regular expressions, regular languages
  - Thompson's construction
  - Subset construction
  - Table representation
- Grammars
  - Context-free grammars
  - Sentence derivation, sentential forms
  - Rightmost, leftmost derivation
  - Precedence
  - Ambiguity

Midterm

- Top-down parsing
  - Left recursion
  - Predictive parsing
    - Lookahead
    - FIRST and FOLLOW sets
    - LL(1) property
  - Recursive descent
  - Left factoring
  - Table-driven predictive parsing

Midterm

- Bottom-up parsing
  - Handles
  - Reductions
  - Shift-reduce parsing
    - General skeleton
    - Shift-reduce, reduce-reduce conflicts
  - LR parser construction
    - LR items
    - Closure and goto functions

Midterm

- Syntax-directed translation
  - Attribute grammars
    - Inherited, synthesized attributes
    - Evaluation
  - Ad-hoc SDT – integration into LR parsing
- Static checking
  - Type systems
  - Type checking rules
  - Type equivalence

Midterm

- Procedures
  - Namespaces and symbol tables
  - Activation records
  - Storage for variables
- Object-oriented programming
  - Objects and classes
  - Implementing inheritance using prefixing
    - For objects: prefixing instance variables
    - For code: handling method overriding

Prelude

- What is this?
  The medal that accompanies a Nobel Prize

- Why did Alfred Nobel create the prize?
  Apparently, guilt over his invention of dynamite
- What are the five Nobel Prize categories?
  Physics, chemistry, medicine, literature, and peace
- What about math and computer science?
  Fields medal, Turing award

Procedures

- Three part abstraction:
  - Control abstraction
    - Single entry, single exit
    - Return-to-callsite semantics
  - Namespace abstraction
    - An isolated "bubble" (scope)
    - Local storage – per procedure activation
  - External interface
    - System-wide procedure call protocol
- All of this is created by the compiler

Job of the compiler

- Control abstraction
  Generate code to:
  - Jump to procedure
  - Remember return address, and jump back
- Namespace abstraction
  - Resolve references (using symbol table)
  - Figure out where to store variables
  - Generate code to access variable storage
- External interface
  Generate code to:
  - Pass parameters
  - Save and restore caller state (e.g., registers)

Where do we store variables?

Non-local variables:

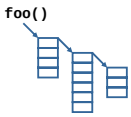
```
int global;
static int local_global;
extern int ext_global;

int foo()
{
  static int hidden_static;
}
```

- Global variables
  - Lifetime is entire execution
  - Store in a global data area
- Static (limited scope)
  - Lifetime is entire execution
  - Procedure scope ⇒ storage area tied to procedure name
  - File scope ⇒ storage area tied to file name

What about locals?

- Compiler responsibilities
  - Which "x"?
    Solved by symbol table
  - Where is "x" in memory at run-time?
- Simplistic model
  - Associate some storage space with each procedure
  - Organize like the symbol table
    - One "sheaf" for each scope
    - Store variable's value in each slot



- What's the problem?

Recursion

- **Problem**:
  - Multiple "x"s may be active at a time
  - Need separate storage for each "x"
- Solution: ***activation record***
  - Block of memory for each *dynamic* instance of a procedure invocation
  - Compiler can store other information
    - Return address, return value
    - Other data related to invocation

Example

```
int global;
static int local_global;
extern int ext_global;
int foo(int x, float y)
{
  int a, b;
  for (int i = 0; i<10; i++) {
    float temp = foo(x-1, y);
  }
  while (y < 3.141) {
    int temp = ...;
  }
  return ...;
}
```

Whole program

`global, local_global foo`

foo() `x, y, a, b`

for `i, temp`

while `temp`

Addressing variables

- Global and static
  - Fixed addresses determined at compile-time
- Local variables?
  - Procedure may be called from many places
  - Recursion: multiple activation records
  - How do we generate code that will work in all these cases?
    How do we generate an address for "x"?
- **Idea**:
  - We can place "x" at a fixed offset from the start of the activation record
  - Computed offset using symbol table structure

Example

Whole program

`global, local_global foo`

foo() `x, y, a, b`

for `i, temp`

while `temp`

Fixed place in address space of program

Fixed offsets relative to activation record of procedure

Example

Base address

foo() `x, y, a, b`

for `i, temp`

while `temp`

x
y
a
b

x at offset 0
y at offset +4
a at offset +12
b at offset +16

i	temp

temp at offset +20

i at offset +20
temp at offset +24

Example

- Code for **`foo()`**
  - Assume base address in common location
    Stack pointer – "sp" – often a register
  - Access to variable is base+offset

Statements like

```
b = a + x;
```

become

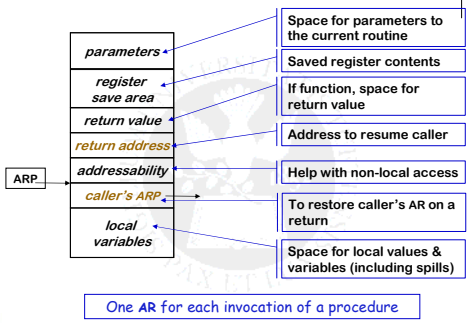
```
*(sp+16) = *(sp+12) + *(sp)
```


Activation records

Space for:

- Local variables
  - Locally declared variables
  - Possibly formal parameters
- Saved calling context
  - Return address, return value
  - Pointer to caller's activation record
  - Saved environment of caller (e.g., registers)

Activation record



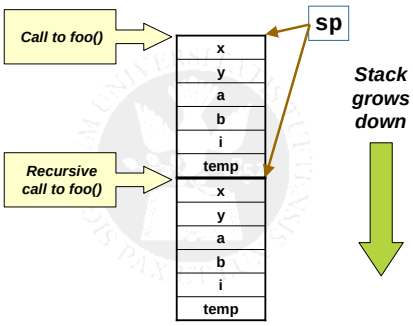
One AR for each invocation of a procedure

Activation record storage

- Where do activation records live?
  - If lifetime of AR matches lifetime of invocation, *AND*
  - If code normally executes a "return"
  - ⇒ Keep ARs on a stack
  - If a procedure can outlive its caller, *OR*
  - If it can return an object that can reference its execution state
  - ⇒ ARs must be kept in the heap
  - If a procedure makes no calls
  - ⇒ AR can be allocated statically

Efficiency prefers static, stack, then heap

At run-time…



Variable-length data

- Support for variable length local variables
  - FORTRAN and Pascal

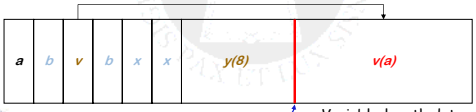
```
subroutine foo(size)
  integer size
  real temp(size)
  real x
  do i = 1, size
    temp(i) = arr(i) + ...
```

  - Where is temp stored?
    - Keeping in mind, we need to know the address of x

Variable-length Data

```
B0:
{
  int a, b
  … assign to a
B1:
  {
    int v(a), b, x
B2:
    {
      int x, y(8)
      ….
    }
```

Arrays
- → If size is fixed at compile time, store in fixed-length data area ←
- → If size is variable, store descriptor in fixed length area, with pointer to variable length area
- → Variable-length data area is assigned at the end of the fixed length area for block in which it is allocated



Variable-length data

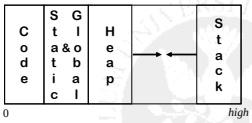
Includes variable length data for all blocks in the procedure …

Variable stack data in C

- C does not support variable array declarations (like Fortran)
- Run-time call to allocate on stack
  - **`void *alloca( size_t size);`**
  - The **`alloca`** function allocates **`size`** bytes of space in the stack frame of the caller. This temporary space is automatically freed on return.

Run-time memory layout

Classic Organization

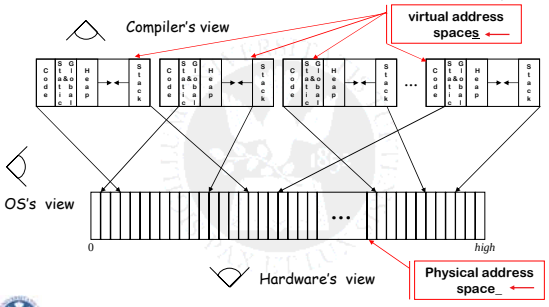


Single Logical Address Space

- Code, static, & global data have known size
  - Use symbolic labels in the code
- Heap & stack both grow & shrink over time
- This is a virtual address space

- Better utilization if stack & heap grow toward each other
- Very old result (Knuth)
- Code & data separate or interleaved
- Uses address space, not allocated memory

How Does This Really Work?

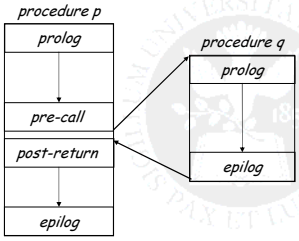


Procedure linkages

- Procedure call implementation
  - What does the compiler have to work with?
    - Load, store
    - Jump
  - Where are the caller and callee?
    - Different compilation units
    - Library code
- Procedure call is a system-wide protocol
  - Orchestrated by compiler
  - Add code to caller and callee

Procedure Linkages

Standard procedure linkage



Procedure has
- standard prolog
- standard epilog

Each call involves a
- pre-call sequence
- post-return sequence

These are completely predictable from the call site ⇒ depend on the number & type of the actual parameters

Procedure linkages

- **Pre-call** sequence
  - Sets up callee's basic AR
  - Helps preserve its own environment
- The details
  - Allocate space for the callee's AR ***(excluding local vars)***
  - Evaluates each parameter & stores value or address
  - Saves return address, caller's ARP into callee's AR
  - Save any caller-save registers ***(save in caller's AR)***
  - Jump to address of callee's prolog code

Procedure linkages

- **Post-return** sequence
  - Finish restoring caller's environment
  - Place any value back where it belongs
- The details
  - Copy return value from callee's AR, if necessary
  - Free the callee's AR
  - Restore any caller-save registers
  - Restore any call-by-reference parameters to registers
    - Also copy back call-by-value/result parameters
  - Continue execution after the call

Procedure linkages

- **Prolog code**
  - Finish setting up callee's environment
  - Preserve parts of caller's environment that will be disturbed
- The details
  - Preserve any callee-save registers
  - Allocate space for local variables
    - Easiest scenario is to extend the AR
  - Find any static data areas referenced in the callee
  - Handle any local variable initializations

With heap allocated AR, may need to use a separate heap object for local variables

Procedure linkages

- **Epilog Code**
  - Wind up the business of the callee
  - Start restoring the caller's environment
- The details
  - Store return value?
    - Some implementations do this on the return statement
    - Others have return assign it & epilog store it into caller's AR
  - Restore callee-save registers
  - Free space for local data, if necessary (on the heap)
  - Load return address from AR
  - Restore caller's ARP
  - Jump to the return address

If ARs are stack allocated, this may not be necessary. (Caller can reset stacktop to its pre-call value.)

Back to activation records

- Stored on the stack
  - Easy to extend — simply bump top of stack pointer
  - Caller & callee share responsibility
    - Caller can push params, space for registers, return value slot, return address, addressability info, & its own ARP
    - Callee can push space for local variables
- Stored on the heap
  - Hard to extend
  - Caller often passes everything it can in registers
  - Callee allocates AR & stores register contents into it
  - Extra parameters stored in caller's AR !
- Static is easy

Object-Oriented Languages

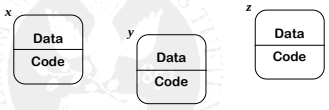
- What is an OOL?

  How is it different from an ALL? ***(Algol-like language)***
  - Data-centric view
    - Objects communicating by "messages" – ***methods***
  - Polymorphism
    - Meaning of message depends on receiver
  - Inheritance
  - What was first OOL?
- Term is almost meaningless today

Object-Oriented Languages

An object is an abstract data type that encapsulates data, operations and internal state behind a simple, consistent interface.

The Concept:

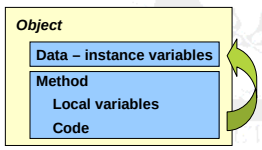


Issues for compilation:

- Each object needs local storage for its attributes
- Access control – notion of privacy
- More complex linkage convention – dynamic aspect

Object structure

- **Idea**:
  - Natural extension of scope
  - Methods can see data in the object, in addition to usual local scopes




```
class Foo {
  int x;
  float y;
  void someMethod(int z)
  {
    int w = z + x;
  }
}
```

OO abstractions

- Object-oriented name spaces
  - Storage associated with methods
    - Local values inside a method
    - Global/static variables
  - Local storage in objects
  - Methods associated with object
- Classes
  - Objects with the same state and methods:
    - Same instance variables, & methods
    - Class variables are static, shared among all objects of same class
  - Allows code reuse at both source & implementation level

There are OO languages that don't use classes: "prototype" based

Implementation

So, what can an executing method see?

- The object's own public & private variables
  - Smalltalk terminology: *instance variables*
- The public & private variables of the class that defines it
  - Smalltalk terminology: *class variables*
- Any object defined in the global name space (or scope)

⇨ Compiler must generate code for any of these

Added twist:

- Most OOLs support a notion of inheritance
  Some OOLs support multiple inheritance

Implementation

- OO slogan:
  If B is a subclass of A, then an object of class B can be used wherever an object of class A is expected
- What does this mean for the implementation?
  - Goal: code reuse
  - How are objects represented?
    - Methods in A must work on B objects
  - How are method calls resolved?
    - Which method is called at o.f()?

Java name space

Given a method M for an object O in a class C, M can see:

- Local variables declared within M (*lexical scoping*)
- All instance variables and class variables of the class C
- All public and protected variables of any superclass of C
- Classes in the same package or in an imported package
  - public class variables and instance variables of imported classes
  - package class and instance variables in the package containing C
- Class declarations can be nested

Java symbol tables

To compile code in method M for an object O within a class C, the compiler needs:

- Lexically scoped symbol table for block and class nesting
  - Just like ALL — inner declarations hide outer declarations
- Chain of symbol tables for inheritance
  - Need mechanism to find the class and instance variables of all superclasses
- Symbol tables for all global classes (package scope)
  - Entries for all members with visibility
  - Need to construct symbol table for imported packages

Java Symbol Tables

To find the address associated with a variable reference in method M for an object O within a class C, the compiler must

- For an unqualified use (i.e., x):
  - Search the scoped symbol table for the current method
  - Search the chain of symbol tables for the class hierarchy
  - Search global symbol table (current package and imported)
  - In each case check access control attribute of x
- For a qualified use (i.e.: Q.x):
  - Search stack-structured global symbol table for Q
  - Check access control attribute of x

OOL Storage Layout (Java)

Class variables

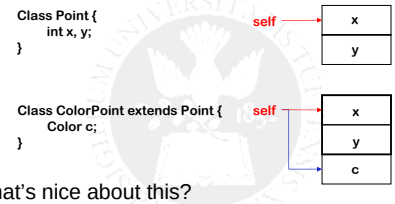
- Static class storage accessible by global name (*class C*)
  - Accessible via linkage symbol &_C
  - Nested classes are handled like blocks in ALLs
  - Method code put at fixed offset from start of class area

Object Representation

- Object storage is heap allocated
  - Fields at fixed offsets from start of object storage
- Methods
  - Code for methods is stored with the class
  - Methods accessed by offsets from code vector
  - Method local storage in object (no calls) or on stack

Dealing with Single Inheritance

- Use prefixing of storage



- What's nice about this?
  - Polymorphism "for free"
  - Code generated for Point will work on ColorPoint

Implementation

Mapping message names to methods

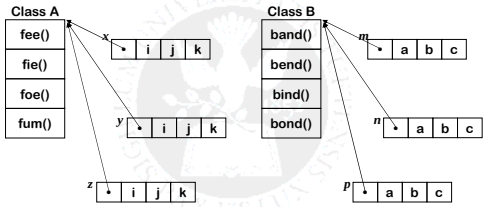
- Static mapping, known at compile-time (Java, C++)
  - Fixed offsets & indirect calls
- Dynamic mapping, unknown until run-time (Smalltalk)
  - Look up name in class' table of methods

This is really a data-structures problem

- Build a table of function pointers
- Use a standard invocation sequence

Implementation

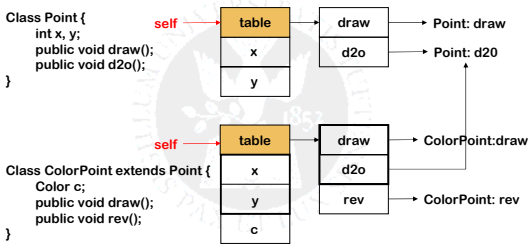
With static, compile-time mapped classes



Message dispatch becomes an indirect call through a function table

Single Inheritance and Dynamic Dispatch

- Use prefixing of tables



The Inheritance Hierarchy

- Two distinct philosophies

Static class structure	Dynamic class structure
• Can map name to code at compile time	• Cannot map name to code at compile time
• Leads to 1-level jump vector	• Multiple jump vector (1/class)
• Copy superclass methods	• Must search for method
• Fixed offsets & indirect calls	• Run-time lookups caching
• Less flexible & expressive	• Much more expensive to run

- In essence, OOL differs from ALL in the shape of its name space ***AND*** in the mechanism used to bind names to implementations

Next time…

- Intermediate representations and intro to code generation
  - (Not on midterm)
- Tuesday, Oct 24: Midterm
  I'll be at a conference, Noah will proctor