Today’s Topic

Runtime - Variables
Storage and Access of Variables

Three types of data memory (variables)

- **Globals**, **locals**, and dynamic/heap (new)
- Focus on first two for now
- How does compiler manage allocation of, reading of, and storing into variables?

Continue with **base + offset** strategy

- Scope vars allocated in-order on Sparc
- Oberon scope $\rightarrow$ Sparc base pointer (register)
- Oberon var decl $\rightarrow$ offset from base pointer
- Offset is sum of sizes of preceding variables

**Modelling** of Oberon, compiler, Sparc memory
Global Variables

VAR z : INTEGER; VAR w : REAL;
PROC… split … w + z… END; (* code *)
VAR A : ARRAY 5 OF INTEGER;

• Compiler symbol table & scopes?
• Sparc Memory?
• What is address of:
  z
  w
  A
Global Variables

Rewrite as C code and compile “cc –S”

```
.seg "data"
.comm _z, 0x4,"data"
.comm _w, 0x4,"data"
...
set _w, %o0 ; set address
ld [%o0], %o0 ; load from address
set _z, %o1
ld [%o1], %o1
```

More consistent with handling of locals:

```
set _z, %g7 ; base is zero offset from first global
...
ld [%g7+4], %o0 ; true base + offset style
ld [%g7+0], %o1
```
Local Vars – hard due to recursion

PROCEDURE quicksort(
    VAR A : ARRAY OF INTEGER;
    low, high : INTEGER)
VAR mid : INTEGER;
BEGIN
    IF low < high THEN
        mid := split(A, low, high);
        quicksort(A, low, mid);
        quicksort(A, mid+1, high);
    END;
END quicksort;

What is the address of
A
low
high
mid

Call frame

Locals & temps
State w/sv’d fp/sp/pc
Params & return

Locals & temps
State w/sv’d fp/sp/pc
Params & return

Locals & temps
State w/sv’d fp/sp/pc
Params & return

0x07FFFFFF (bottom)

Call stack with 3 frames
Design Considerations

“Code to specification” to get good interface
  • Approach top-down from use, incremental (start with INTEGER)
  • Implies “base” and “offset” methods for STO
Design Considerations

“Code to specification” to get good interface
• Approach top-down from use, incremental (start with INTEGER)
• Implies “base” and “offset” methods for STO

*Machine* details deserve their own *class*
• Machine architecture could change, use differently
• A *façade* class on range of machine details
  – Register classes, allocation; machine instructions…
Design Considerations

“Code to specification” to get good interface

• Approach top-down from use, incremental (start with INTEGER)
• Implies “base” and “offset” methods for STO

*Machine details deserve their own class*

• Machine architecture could change, use differently
• A *façade* class on range of machine details
  – Register classes, allocation; machine instructions…

```java
String base() { // ExprSTO
  if (this.isLocal()) { // “Local” property inherited from scope
    return Machine.localBase();
  }
  else if …
} // Now what about offset()?
```
Design Considerations

“Code to specification” to get good interface
• Approach top-down from use, incremental (start with INTEGER)
• Implies “base” and “offset” methods for STO

*Machine* details deserve their own *class*
• Machine architecture could change, use differently
• A *façade* class on range of machine details
  – Register classes, allocation; machine instructions…

```java
String base() { // ExprSTO
    if (this.isLocal()) {
        // “Local” property inherited from scope
        return Machine.localBase();
    }
    else if ...
} // Now what about offset()?

int size() { // ExprSTO
    return this.type().size();
}
```
Design Considerations

“Code to specification” to get good interface
- Approach top-down from use, incremental (start with INTEGER)
- Implies “base” and “offset” methods for STO

**Machine** details deserve their own *class*
- Machine architecture could change, use differently
- A *façade* class on range of machine details
  - Register classes, allocation; machine instructions…

String base() {  // ExprSTO
  if (this.isLocal()) { // “Local” property inherited from scope
    return Machine.localBase();
  }
  else if ...
}  // *Now what about offset()*?

int size() {  // ExprSTO
  return this.type().size();
}

int size()  // IntegerType
{ return Machine.intSize(); }  // *rep bool as int, BTW*