


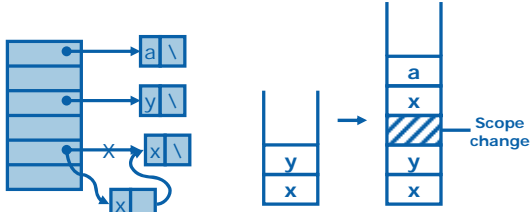
Activation Records

CS 471
October 22, 2007



Last Time ... PA5 Workshop

We have two symbol tables: **tenv** and **venv**



CS 471 – Fall 2007

We will write

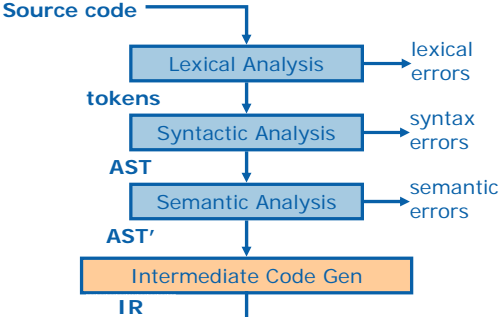
Recursive functions to operate on the AST

```

transTy:
  A_type * tenv → Ty_ty
transVar:
  A_var * tenv * venv → Ty_ty (+ int code)
transExp:
  A_exp * tenv * venv → Ty_ty (+ int code)
transDec:
  A_dec * tenv * venv → () /* side effect symtab */
  
```

CS 471 – Fall 2007

Intermediate Code Generation



CS 471 – Fall 2007

Run time vs. Compile time

The compiler must generate code to handle issues that arise at run time

- Representation of various data types
- Procedure linkage
- Storage organization

CS 471 – Fall 2007

Control abstraction

A procedure is a control abstraction

- it associates a name with a chunk of code
- that piece of code is regarded in terms of its purpose and not of its implementation.

Big issue #1: Allow separate compilation

- Without it we can't build large systems
- Saves compile time
- Saves development time
- We must establish conventions on memory layout, calling sequences, procedure entries and exits, interfaces, etc.

CS 471 – Fall 2007

Control abstraction

Procedures must have a well defined call mechanism

In many languages:

- a call creates an instance (activation) of the procedure
- on exit, control returns to the call site, to the point right after the call.

Use a call graph to see set of potential calls

6

CS 471 – Fall 2007

Control abstraction

Generated code must be able to

- preserve current state
 - save variables that cannot be saved in registers
 - save specific register values
- establish procedure environment on entry
 - map actual to formal parameters
 - create storage for locals
- restore previous state on exit

7

CS 471 – Fall 2007

Local Variables

- Functions have **local variables**
 - created upon entry
- Several invocations may exist
- Each invocation has an instantiation
- Local variables are (often) destroyed upon function exit

- Happens in a LIFO manner
- What else operates in a LIFO manner?

8

CS 471 – Fall 2007

Stacks

Work languages with **nested functions**

- Functions declared inside other functions
 - Inner functions can use outer function's local vars
 - Doesn't happen in C
 - Does happen in Tiger! (and Pascal)
- Work with languages that support function pointers

ML, Scheme have higher-order functions:

- nested functions AND
- functions as returnable values

So ... ML, Scheme cannot use stacks for local vars!

9

CS 471 – Fall 2007

Stack Frames

Basic operations: push, pop

Happens too frequently!

Local variables can be pushed/popped in large batches (on function entry/exit)

Instead, use a big array with a **stack pointer**

- Garbage beyond the end of the sp
- A **frame pointer** indicates the start (for this procedure)

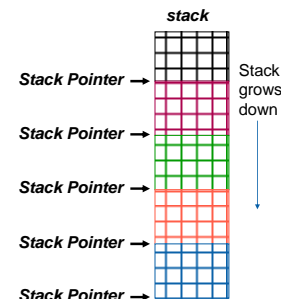
10

CS 471 – Fall 2007

Stack

• Last In, First Out (LIFO) data structure

```
main ()
{ a(0);
  void a (int m)
  { b(1);
    void b (int n)
    { c(2);
      void c (int o)
      { d(3);
        void d (int p)
        {
        }
      }
    }
  }
}
```



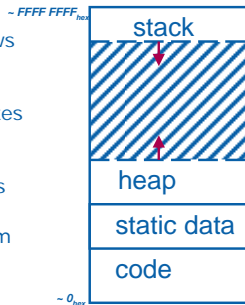
11

CS 471 – Fall 2007

Review: Normal C Memory Management

A program's **address space** contains 4 regions:

- **stack**: local variables, grows downward
- **heap**: space requested for pointers via `malloc()`; resizes dynamically, grows upward
- **static data**: variables declared outside main, does not grow or shrink
- **code**: loaded when program starts, does not change



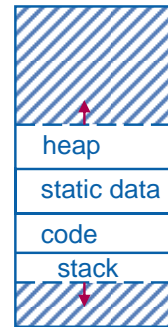
12

CS 471 – Fall 2007

Intel x86 C Memory Management

A C program's **x86 address space**:

- **heap**: space requested for pointers via `malloc()`; resizes dynamically, grows upward
- **static data**: variables declared outside main, does not grow or shrink
- **code**: loaded when program starts, does not change
- **stack**: local variables, grows downward



13

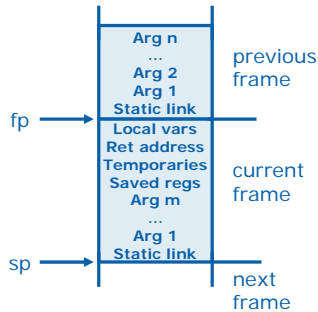
CS 471 – Fall 2007

Stack Frames

Activation record or stack frame stores:

- local vars
- parameters
- return address
- temporaries
- (...etc)

(Frame size not known until late in the compilation process)



14

CS 471 – Fall 2007

Stack Frame Contents

- **Parameters** (outgoing) in reverse order (Why?)
- **Local variables** – declared in the source
- **Temporary values** – result of intermediate computations
- **Return value** – kept in caller's frame (Why?)
- **Stack pointer**
- **Frame pointer**
- **Return address** – saved PC
- **Dynamic Link** – reference to the frame of the caller
- **Static Link** – reference to the nearest frame of the lexically enclosing procedure

15

CS 471 – Fall 2007

Handling nested procedures

Some languages allow nested procedures

- Example:

```

proc A() {
  proc B() {
    call C()
  }
  proc C() {
    proc D() {
      proc E() {
        call B()
      }
      call E()
    }
    call D()
  }
  call B()
}
    
```

call sequence:

A
|
B
|
C
|
D
|
E
|
B

B can call C
B cannot call D or E
E can access C's locals
C cannot access B's locals

16

CS 471 – Fall 2007

Handling nested procedures

- In order to implement the "closest nested scope" rule we need access to the frame of the lexically enclosing procedure

Solution: static links

- Reference to the frame of the lexically enclosing procedure
- Static chains of such links are created.
- How do we use them to access non-locals?
 - The compiler knows the scope `s` of a variable
 - The compiler knows the current scope `t`
 - Follow `s-t` links

17

CS 471 – Fall 2007

Handling nested procedures

Setting the links:

- if callee is nested directly **within caller**
 - set its static link to point to the caller's frame pointer

```
proc A()  
  proc B()
```

- if callee has the **same nesting level** as the caller
 - set its static link to point to wherever the caller's static link points

```
proc A()  
proc B()
```

18

CS 471 – Fall 2007

Storage – Registers vs. Memory

- Access to registers is much faster than access to memory

• Goal: store as much data as possible in registers

• Limitations/considerations:

- Small number of registers, e.g. if there are no registers available or their contents need to be spilled, we must have a place in memory where this data can be saved
- Some data cannot be stored in registers, e.g. variables whose address needs to be accessed, or variables that are too large to fit in a register

• Things we'd like to store in registers:

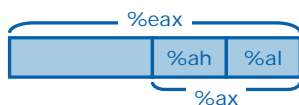
- Frequently used variables
- Function parameters
- Function return values

19

CS 471 – Fall 2007

Registers

- Depending on the architecture, may have more or fewer registers (typically 32)



- Always faster to use registers (remember the memory hierarchy)
- Want to keep local variables in registers (when possible ... why can't we decide now?)

20

CS 471 – Fall 2007

Parameter passing

By value

- actual parameter is copied

By reference

- address of actual parameter is stored

By value-result

- call by value, AND
- the values of the formal parameters are copied back into the actual parameters

21

CS 471 – Fall 2007

Summary

- Stack frames keep track of run-time data
- Both language and architectural features will affect the stack frame layout and contents
- Next time: How this all relates to your project

Class Notices:

- PA5 is due on Friday – please start early

22

CS 471 – Fall 2007