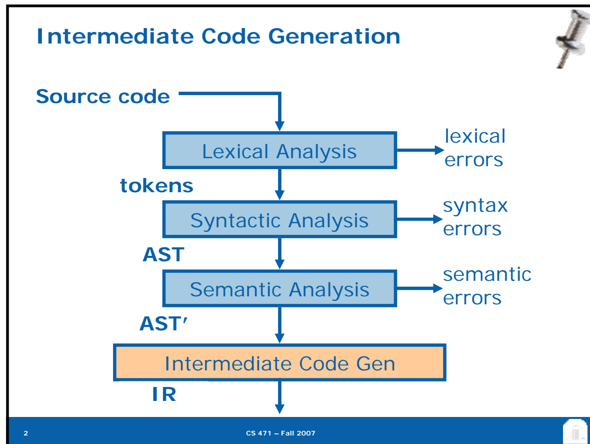
Course Announcements

- **PA5 due Friday**
  - we've got a head start!
  - still a good bit of work to do!
- **I will be at NSF on Monday-Tuesday**
  - Guest lecture on intermediate representations
- **No class on Monday before Thanksgiving**

- **Anonymous feedback?** (Link on the web page)
  - Pace of the lectures
  - Pace of the project
  - Any other feedback

Intermediate Code Generation



Activation Records

The compiler must generate code to handle issues that arise at run time
- Representation of various data types
- Procedure linkage
- Storage organization

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

A procedure creates its own name space
- It can declare local variables
- Local declarations may hide non-local ones
- Local names cannot be seen from outside

Why is this separate from symbol tables?

Handling Control Abstractions

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

This can be modeled with a stack
- Allocate a memory block for each activation
- Maintain a stack of such blocks
- This mechanism can handle recursion

Handling Variable Storage

Static allocation
- object is allocated an address at compile time
- location is retained during execution

Stack allocation
- objects are allocated in LIFO order

Heap allocation
- objects may be allocated and deallocated at any time.

Static Allocation

Objects that are allocated statically include:
- globals
- explicitly declared static variables
- instructions
- string literals
- compiler-generated tables used during run time

Stack Allocation

Follows stack model for procedure activation

A procedure's memory block associated with an activation of the procedure is called an activation record or stack frame

The stack frame is pushed on the stack when the procedure is called and popped (and destroyed) when the procedure terminates

What can we determine at compile time?
- We cannot determine the address of the stack frame
- But we can determine the size of the stack frame and the offsets of various objects within a frame

Heap Allocation

Used for dynamically allocated/resized objects

Managed by special algorithms

General model
- maintain list of free blocks
- allocate block of appropriate size
- handle fragmentation
- handle garbage collection

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)
{
}
```

stack

Stack Pointer →

Stack Pointer →

Stack Pointer →

Stack Pointer →

Stack Pointer →

Stack grows down

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)

	Stack	
	Arg n … Arg 2 Arg 1 Static link	previous frame
fp →	Local vars Ret address Temporaries Saved regs Arg m … Arg 1 Static link	current frame
sp →		next frame

The Frame Pointer

Keeps track of the bottom of the current activation record

```
g(...)
{
  f(a1,...,an);
}
```

- **g is the caller**
- **f is the callee**

	Stack	
	Arg n … Arg 2 Arg 1 Static link	g's frame
fp →	Local vars Ret address Temporaries Saved regs Arg m … Arg 1 Static link	f's frame
sp →		next frame

What if f calls two functions?

Activation Records

- **Handling nested procedures**
  - We must keep a static link (vs. dynamic link)
- **Registers vs. memory**
  - Registers are faster. Why?

```
add %eax,%ebx,%ecx
```

```
ld %ebx, 0[%fp]
ld %ecx, 4[%fp]
add %eax,%ebx,%ecx
```

Cycles?

Registers

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

%eax | %ah | %al | %ax

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

Caller-save vs. Callee-save Registers

What if f wants to use a register? Who must save/restore that register?

```
g(…) {
   f(a1,…,an);
}
```

Stack (g's frame): Arg n, …, Arg 2, Arg 1, Static link — fp

f's frame: Local vars, Ret address, Temporaries, Saved regs, Arg m, …, Arg 1, Static link — sp

next frame

- If g does a save before the call and a restore after the call →**caller-save**
- If f does a save before it uses a register and restore afterward →**callee-save**

What are the tradeoffs?

Caller-save vs. Callee-save Registers

Usually – some registers are marked caller save and some are marked callee save

e.g. MIPS: r16-23 callee save

all others caller save

Optimization – if g knows it will not need a value after a call, it may put it in a caller-save register, but not save it

Deciding on the register will be the task of the register allocator (still a hot research area).

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

Typical Convention
- Usually 4 parameters placed in registers
- Rest on stack
- Why?

Parameter Passing

We often put 4/6 parameters in registers …
What happens when we call another function?

- Leaf procedures – don't call other procedures
- Non-leaf procedures may have dead variables
- Interprocedural register allocation
- Register windows

r32	r33	r34	r35	r32	r33	r34	r35	r32	r33	r34	r35	r32

A() B() C()

Outgoing args of A become incoming args to B

Return Address

Return address - after `f` calls `g`, we must know where to return

➢**Old machines –** return address always pushed on the stack

➢**Newer machines –** return address is placed in a special register (often called the link register `%lr`) automatically during the `call` instruction

Non-leaf procedures must save `%lr` on the stack

Sidenote: Itanium has 8 "branch registers"

Stack maintenance

Calling sequence :

- code executed by the caller before and after a call
- code executed by the callee at the beginning
- code executed by the callee at the end

Stack maintenance

A typical calling sequence :

1. Caller assembles arguments and transfers control
   - evaluate arguments
   - place arguments in stack frame and/or registers
   - save caller-saved registers
   - save return address
   - jump to callee's first instruction

Stack maintenance

A typical calling sequence :

2. Callee saves info on entry
   - allocate memory for stack frame, update stack pointer
   - save callee-saved registers
   - save old frame pointer
   - update frame pointer
3. Callee executes

Stack maintenance

A typical calling sequence :

4. Callee restores info on exit and returns control
   - place return value in appropriate location
   - restore callee-saved registers
   - restore frame pointer
   - pop the stack frame
   - jump to return address
5. Caller restores info
   - restore caller-saved registers

Tiger Compiler Frames

How should our Tiger compiler stack frames be organized?

- Depends whether we want to call C functions
- Actual layout will be architecture dependent

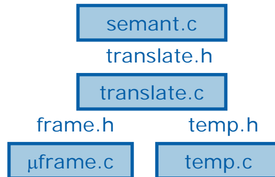
Approach: abstract away the implementation (just as we did with the symbol table)

```
typedef struct F_frame_ *F_frame;
```

Layers of Abstraction

Frame and temp interfaces – machine-independent views of memory-resident and register-resident variables

Translate – generates IR – manages local variables and static function nesting. (For clarity – translate is separated from semant)



Summary

- **Stack frames**
  - Keep track of run-time data
  - Define the interface between procedures
- **Both language and architectural features will affect the stack frame layout and contents**
- **Started to see hints of the back-end optimizer, register allocator, etc.**

- **Next week: Intermediate Representations**