Compiler Internals

15-745 Optimizing Compilers
Spring 2006

Peter Lee

Reminders

* Get on the course mailing list
* Check out the web site
  * http://www.cs.cmu.edu/afs/cs/academic/class/15745-s06/web
  * subscribe to the RSS feed
* Strongly consider buying the textbook
  * Read 4.1-4.4, 4.9, 12.4
* Task 0 is available, due on Thursday

Registering for the course

* If you are not registered for the course, please see me and send me an email explaining your situation
* I will not clear you off the waiting list until you do that

Course infrastructure

* A compiler for a small, safe, imperative language, called L3, into x86 assembler
* A "toy", in the sense that it compiles, but does not provide debugging support, multiple targets, error recovery, heap tracing, ...
* But for optimization, the problems will be more than realistic enough, even for some research purposes

Task 0

- To get warmed up, T0 involves simply learning the L3 language by writing some working programs in it
- Also, installing and building the (unoptimizing) L3 compiler
- Do this one on your own; but find a partner for Tasks 1 (next week), 2, and 3

The L3 Language

The L3 syntax

⟨program⟩	::=	[⟨struct⟩ | ⟨function⟩]⋆
⟨stuct⟩	::=	**struct** ⟨ident⟩ **{** ⟨ident⟩ **:** ⟨type⟩ [**;** ⟨ident⟩ **:** ⟨type⟩]⋆ [**;**] **} ;**
⟨function⟩	::=	⟨type⟩ ⟨ident⟩ **(** ⟨paramlist⟩ **)** ⟨body⟩ | **void** ⟨ident⟩ **(** ⟨paramlist⟩ **)** ⟨body⟩
⟨paramlist⟩	::=	ε | ⟨ident⟩ **:** ⟨type⟩ [**,** ⟨ident⟩ **:** ⟨type⟩]⋆
⟨body⟩	::=	**{** ⟨decl⟩⋆ ⟨stmt⟩⋆ **}** | **foreign**
⟨decl⟩	::=	**var** ⟨ident⟩ [**,** ⟨ident⟩]⋆ **:** ⟨type⟩ **;**
⟨type⟩	::=	**bool** | **int** | ⟨ident⟩ | ⟨type⟩⋆
⟨stmt⟩	::=	⟨simp⟩ **;** | ⟨control⟩ | **;**
⟨simp⟩	::=	⟨lval⟩ ⟨asop⟩ ⟨exp⟩ | ⟨exp⟩ | **return** ⟨exp⟩ | **return**
⟨control⟩	::=	**if (** ⟨exp⟩ **)** ⟨block⟩ [**else** ⟨block⟩] |
		for ([⟨lval⟩ ⟨asop⟩ ⟨exp⟩] **;** ⟨exp⟩ **;** [⟨simp⟩] **)** ⟨block⟩ |
		while (⟨exp⟩ **)** ⟨block⟩ | **continue** | **break**
⟨block⟩	::=	⟨stmt⟩ | **{** ⟨stmt⟩⋆ **}**
⟨exp⟩	::=	**(** ⟨exp⟩ **)** | ⟨const⟩ | ⟨lval⟩ | ⟨unop⟩ ⟨exp⟩ | ⟨exp⟩ ⟨binop⟩ ⟨exp⟩ |
		& ⟨lval⟩ | **alloc (** ⟨exp⟩ **,** ⟨type⟩ **)** | **offset (** ⟨exp⟩ **)** | **size (** ⟨exp⟩ **)** |
		⟨ident⟩ **(** [⟨exp⟩ [**,** ⟨exp⟩]⋆] **)**
⟨lval⟩	::=	**(** ⟨lval⟩ **)** | ⟨ident⟩ | ***** ⟨exp⟩ | ⟨exp⟩ **[** ⟨exp⟩ **]** |
		⟨lval⟩ **.** ⟨ident⟩ | ⟨exp⟩ **->** ⟨ident⟩ | **(** ⟨lval⟩ **)**
⟨const⟩	::=	⟨intconst⟩ | **true** | **false** | **NULL**
⟨ident⟩	::=	[**A-Z_a-z**][**0-9A-Z_a-z**]⋆
⟨intconst⟩	::=	[**0-9**][**0-9**]⋆
⟨asop⟩	::=	**=** | **+=** | **-=** | ***=** | **/=** | **%=**
⟨binop⟩	::=	**+** | **-** | ***** | **/** | **%** | **<** | **<=** | **==** | **!=** | **>** | **>=** |
		&& | **||** | **^^** | **&** | **|** | **^** | **<<** | **>>** | ***+** | ***-** | ***==** | ***!=**
⟨unop⟩	::=	**!** | **~** | **-**

L3 is a safe language

- L3 is **syntactically similar to C**
  - has (non-nested) recursive procedures and the usual control constructs (except setjmp/longjmp)
- L3 is **strongly typed** and has only **safe access to memory**
- Please study the sample test programs we have provided, and try to write "interesting" L3 programs of your own

A sample L3 program

```
int find_primes(list : int*) {
  var p : int;
  var sum : int;
  sum = 2;

  list = list *- offset(list);
  list[0] = 2;
  list = list *+ 1;
  p = 3;

  while(offset(list) < size(list)) {
    if(isPrime(p, list)) {
      sum += p;
      *list = p;
      list = list *+ 1;
    }
    p += 2;
  }
  return sum;
}
```

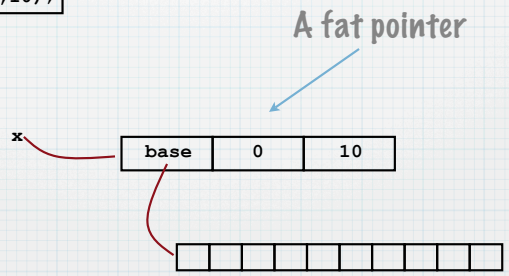
Compiled L3 Programs

Compiled L3 programs

* In addition to studying L3, take a close look at the target assembly code
* One interesting feature is that the L3 implementation uses **fat pointers** (reminiscent of those used by CCured and Cyclone)

Fat pointers

```
var x : int*;
…
x = alloc(int,10);
```



Fat pointer implementations

* In the unoptimized implementations, fat pointers are themselves heap-allocated data structures

base	offset	length

"relative" representation

start	end	current

"absolute" representation

Why memory safety?

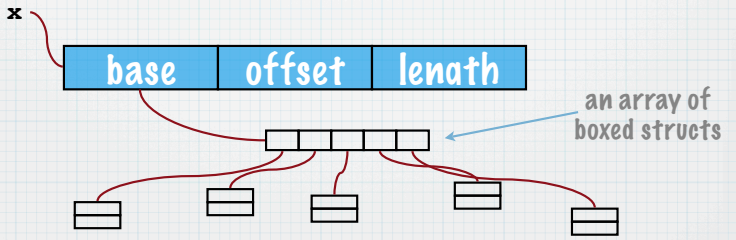
* Obviously, memory safety will provide many opportunities for optimization
* These optimizations are relevant to modern languages such as ML, Java, Cyclone, etc.
* And anyway, life (and this course!) is too short to spend it on uncivilized languages ;-)

Boxed objects

* In an unoptimized implementation of L3, all "large" values (i.e., structs) are heap-allocated
  * thus, struct are always represented by fat pointers
  * this is called the **boxed** representation

Boxed representation

```
struct list {value : int; next : list*;}
var x : list*;
 …
x = alloc(5, list);
```



Boxing

* Obviously, boxed representations are not always the most efficient
* We may look at the "unboxing" optimization later...

Doing better

* In more realistic implementations, both larger and smaller representations of pointers are possible
* E.g., if a pointer is only incremented by positive numbers:

A topic for a future lecture...

Pointer dereferences

* To dereference a pointer, the following precondition must be established:
  * offset < length (in relative), or
  * start <= current < end (in absolute)

Pointer arithmetic

* Pointer arithmetic is always OK, because dereferences are always checked
* Note that, in general, pointer arithmetic results in the creation of a new fat pointer in the heap

What happens here?

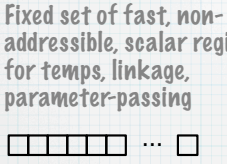
```
…
var x : int*;
var y : int*;
…
x = alloc(10,int);
…
y = x;
…
  … x + 1 …
```

Null

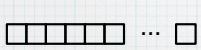
- Null is probably a mistake! But we'll live with it since L3 has it...
- How should null be represented?
  - can use offset>=length (or current>end)
- Null-equality checking to be infrequent
  - so can use a statically allocated null

Registers, stack, and heap

Fixed set of fast, non-addressible, scalar registers for temps, linkage, parameter-passing

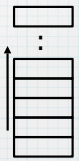


Run-time stack (grows downwards), for spillage, caller/callee saves, excess parameters



Fixed set of fast, non-addressible, floating-point registers for temps, linkage, parameter-passing

Heap (grows upwards), for dynamically allocated objects

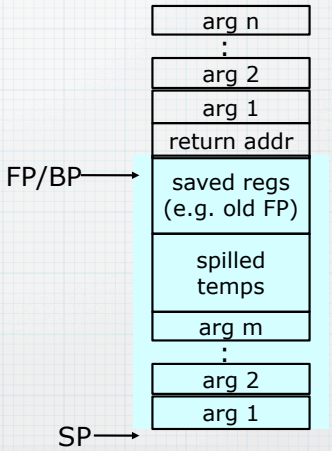


Activation records

The run-time stack is organized as a stack of activation records

Each record contains values pertinent to a single active procedure

Most modern architectures transmit the first few procedure arguments in registers, and the rest on the stack

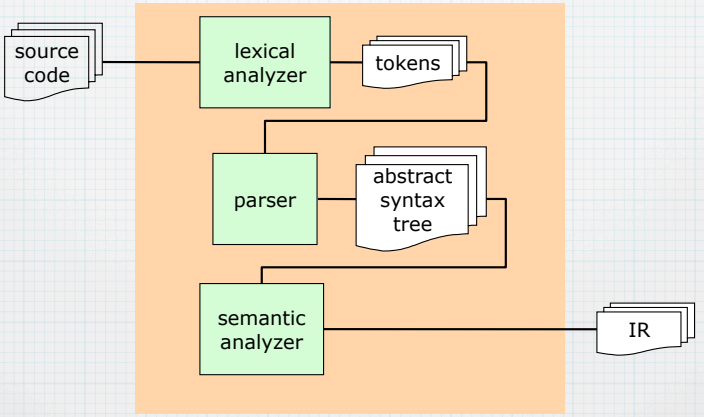


C calling convention

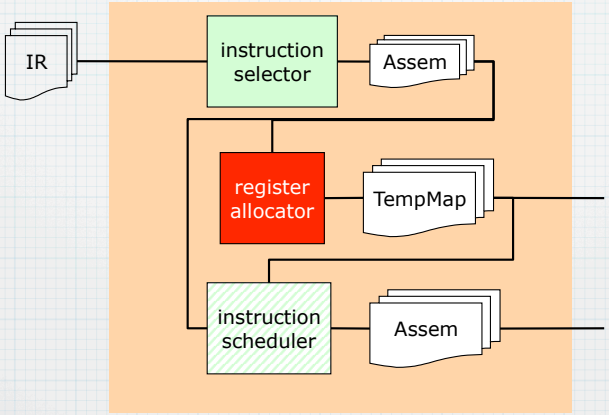
- Specifies the procedure linkage protocol, including caller/callee-save registers
- On x86:
  - caller-save: %eax, %ecx, %edx, eflags, fflags, and all FP registers
  - callee-save: %ebp, %esp, %ebx, %esi, % edi, and FP register stack top

L3 Compiler Internals

Front-end structure



Back-end structure



Main internal structures

- For our purposes, there are two main internal data structures
  - intermediate representation (**IR Tree**)
  - assembly instructions (**Assem**)

Intermediate representations

* Many different approaches
  * 3-address code
  * expression trees
  * static single-assignment form
  * program dependence graphs
  * continuation-passing style

Common IR features

* Simple syntax and semantics
* Arguments/operands are "simple" (i.e., constant-time computations)
* Assembler-like conditional and call instructions

```
...
t2 = mem(t1)
t1 = t1 + 4
...
t3 = t3 + 1
t5 = t3 < t4
if t5 goto L1
...
```

Sample 3-address code

Terminology alert

* A **temp** (aka, "pseudo-register") is a variable in the IR
* It can be thought of as a machine register
  * infinite supply
  * non-addressible
  * global scope

Tree representation

* Flat representations such as 3-address code is widespread but often inconvenient
* Even simple source expressions get translated into 3-address codes that use a lot of temporaries
* Often, the high-level structure of trees provides flexibility

Syntax of the L3 Tree IR

```
exp ::= const
      | label
      | temp
      | binop(exp, exp)
      | MEM(exp)
      | CALL(symbol, [exp, exp, ...])
      | ESEQ(stm, exp)

stm ::= COMMENT(string)
      | MOVE(exp, exp)
      | EXP(exp)
      | JUMP(label)
      | CJUMP(relop, exp, exp, label, label)
      | SEQ(stm, stm)
      | LABEL(label)
      | NOP
```

Example

source code

```
if (3<4 && 5<6) x += 1;
else y = 1;
```

abstract syntax tree

```
IfElse(OpExp(AndOp,
             OpExp(LtOp,ConstExp 3,ConstExp 4),
             OpExp(LtOp,ConstExp 5,ConstExp 6)),
       [Assign(Var x, OpExp(PlusOp,LVal(Var x),ConstExp 1))],
       [Assign(Var y, ConstExp 1)])
```

abstract syntax tree

```
IfElse(OpExp(AndOp,
             OpExp(LtOp,ConstExp 3,ConstExp 4),
             OpExp(LtOp,ConstExp 5,ConstExp 6)),
       [Assign(Var x, OpExp(PlusOp,LVal(Var x),ConstExp 1))],
       [Assign(Var y, ConstExp 1)])
```

```
Nx(SEQ(SEQ(CJUMP(Less,Int 3,Int 4,L_1,L_3),
           SEQ(LABEL(L_1),
               CJUMP(Less,Int 5,Int 6,L_2,L_3))),
       SEQ(LABEL(L_2),
        SEQ(MOVE(TEMP x,BINOP(Plus,TEMP x,Int 1)),
         SEQ(JUMP(L_4),
          SEQ(LABEL(L_3),
           SEQ(MOVE(TEMP y,Int 1),
               LABEL(L_4)))))))
```

tree IR

From trees to linear code

- Trees are convenient for many purposes, but actual machine code is
  - linear (no nesting of SEQs, etc)
  - only single-labeled conditional branches
- Some optimizations are also expressed more easily on linear code instead of trees

Terminology alert

- A tree is **canonical** if
  - it contains no SEQ or ESEQ
  - every CALL is in one of these contexts:
    - MOVE(TEMP t, CALL(...))
    - EXP(CALL(...))
    - optionally: all args to binops are either constant or temp
  - i.e., similar to an assembly instruction

```
Nx(SEQ(SEQ(CJUMP(Less,Int 3,Int 4,L_1,L_3),
         SEQ(LABEL(L_1),
             CJUMP(Less,Int 5,Int 6,L_2,L_3))),
     SEQ(LABEL(L_2),
      SEQ(MOVE(TEMP x,BINOP(Plus,TEMP x,Int 1)),
       SEQ(JUMP(L_4),
        SEQ(LABEL(L_3),
         SEQ(MOVE(TEMP y,Int 1),
             LABEL(L_4)))))))
```

tree IR

Linear sequence of canonical trees, aka "statements" or "instructions"

```
CJUMP(Less,Int 3,Int 4,L_1,L_3)
LABEL(L_1)
CJUMP(Less,Int 5,Int 6,L_2,L_3)))
LABEL(L_2)
MOVE(TEMP x,BINOP(Plus,TEMP x,Int 1))
JUMP(L_4)
LABEL(L_3)
MOVE(TEMP y,Int 1)
LABEL(L_4)
```

```
LABEL(L_start)
CJUMP(Less,Int 3,Int 4,L_1,L_3)
LABEL(L_1)
CJUMP(Less,Int 5,Int 6,L_2,L_3)))
LABEL(L_2)
MOVE(TEMP x,BINOP(Plus,TEMP x,Int 1))
JUMP(L_4)
LABEL(L_3)
MOVE(TEMP y,Int 1)
LABEL(L_4)
JUMP(L_end)
LABEL(L_end)
```

Some "administrative" labels and jumps added (in red)

```
LABEL(L_start)
CJUMP(Less,Int 3,Int 4,L_1,L_3)
```

```
LABEL(L_1)
CJUMP(Less,Int 5,Int 6,L_2,L_3)))
```

```
LABEL(L_2)
MOVE(TEMP x,BINOP(Plus,TEMP x,Int 1))
JUMP(L_4)
```

```
LABEL(L_3)
MOVE(TEMP y,Int 1)
JUMP(L_4)
```

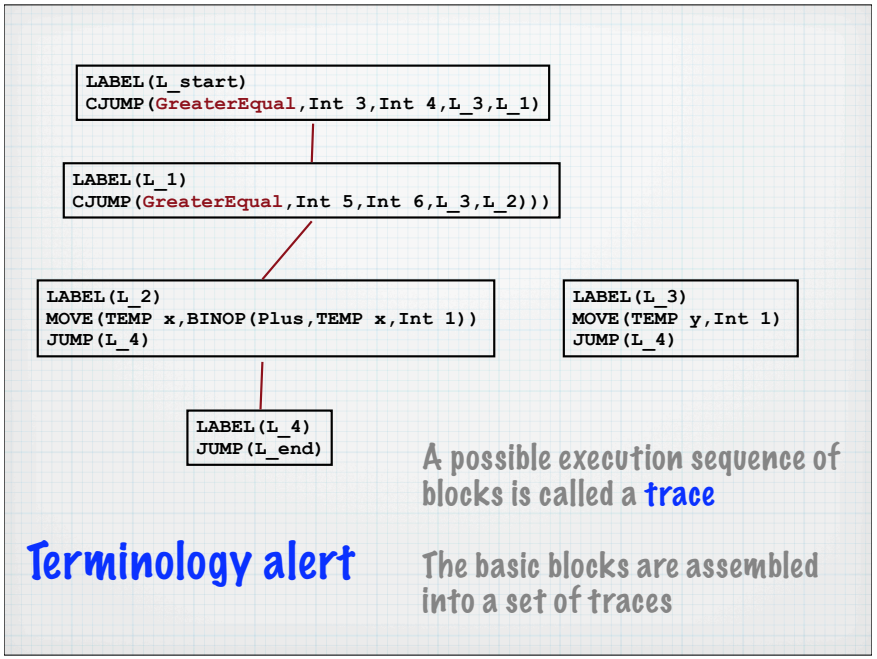
```
LABEL(L_4)
JUMP(L_end)
```

Terminology alert

Each consecutive sequence of instructions that contains a unique entry point and ends with a JUMP or CJUMP is a **basic block**

The IR is divided into an unordered set of basic blocks

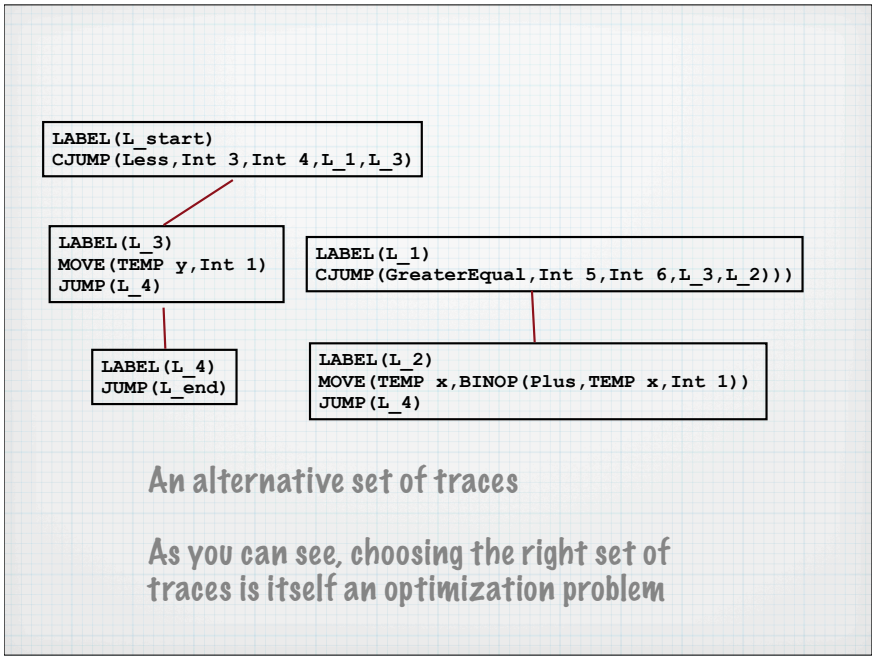
While this step is theoretically optional, we will see later that it will make dataflow analysis run much faster



A possible execution sequence of blocks is called a **trace**

Terminology alert

The basic blocks are assembled into a set of traces



An alternative set of traces

As you can see, choosing the right set of traces is itself an optimization problem

Conditional jumps

- Linearized code often constrains the use of conditional jumps, too
  - Any CJUMP followed by its false label is OK
  - Any CJUMP followed by its true label should be switched and its condition negated
  - Any CJUMP(c,x,y,lt,lf) followed by neither label should be rewritten as:

```
CJUMP(c,x,y,lt,newLf)
LABEL newLf
JUMP lf
```

Assem representation

source and target temps

```
…
   movl     $2, t7
   movl     t7, t11
   imull    t7, t11
   movl     t11, t10
   imull    $37, t10
   movl     t10, t8
   movl     t7, t17
   addl     $1, t17
   movl     $33, t1
   cltd
   idivl    t17
…
```

```
…
OPER("movl\t$2, `d0",    [],     [t7])
MOVE("movl\t`s0, `d0",   [t7],   [t11])
OPER("imull\t`s0, `d0",  [t7],   [t11])
MOVE("movl\t`s0, `d0",   [t11],  [t10])
OPER("imull\t$37, `d0",  [],     [t10])
MOVE("movl\t`s0, `d0",   [t10],  [t8])
MOVE("movl\t`s0, `d0",   [t7],   [t17])
OPER("addl\t$1, `d0",    [],     [t17])
OPER("movl\t$33, `d0",   [],     [%eax])
OPER("cltd",             [%eax], [%edx])
OPER("idivl `s0", [t17], [%eax,%edx])
…
```

represented as

`tempMap: temp -> string`

Allocator's view

```
…
OPER("movl\t$2, `d0",    [],     [t7])
MOVE("movl\t`s0, `d0",   [t7],   [t11])
OPER("imull\t`s0, `d0",  [t7],   [t11])
MOVE("movl\t`s0, `d0",   [t11],  [t10])
OPER("imull\t$37, `d0",  [],     [t10])
MOVE("movl\t`s0, `d0",   [t10],  [t8])
MOVE("movl\t`s0, `d0",   [t7],   [t17])
OPER("addl\t$1, `d0",    [],     [t17])
OPER("movl\t$33, `d0",   [],     [%eax])
OPER("cltd",             [%eax], [%edx])
OPER("idivl `s0",  [t17], [%eax,%edx])
…
```

```
…
t7   <- ⊗()
t11  := t7
t11  <- ⊗(t7)
t10  := t11
t10  <- ⊗()
t8   := t10
t17  := t7
t17  <- ⊗()
%eax <- ⊗()
%edx <- ⊗(%eax)
%eax,%edx <- ⊗(t17)
…
```

Register allocator's view

Register allocation

The register allocator assigns each temp to a machine register

If that fails, the register allocator keeps rewriting the code so that it can succeed

Much more on register allocation later in the semester

```
…
t7   <- ⊗()
t11  := t7
t11  <- ⊗(t7)
t10  := t11
t10  <- ⊗()
t8   := t10
t17  := t7
t17  <- ⊗()
%eax <- ⊗()
%edx <- ⊗(%eax)
%eax,%edx <- ⊗(t17)
…
```

```
…
t7  : %ebx
t8  : %ecx
t10 : %eax
t11 : %eax
t17 : %esi
…
```

The TempMap

Remember…

* Get on the course mailing list
* Check out the web site
  * http://www.cs.cmu.edu/afs/cs/academic/class/15745-s06/web
  * subscribe to the RSS feed
* Strongly consider buying the textbook
* Task 0 is available, so **play with L3 and the compiler!**