CSE P 501 – Compilers

Instruction Selection

Hal Perkins

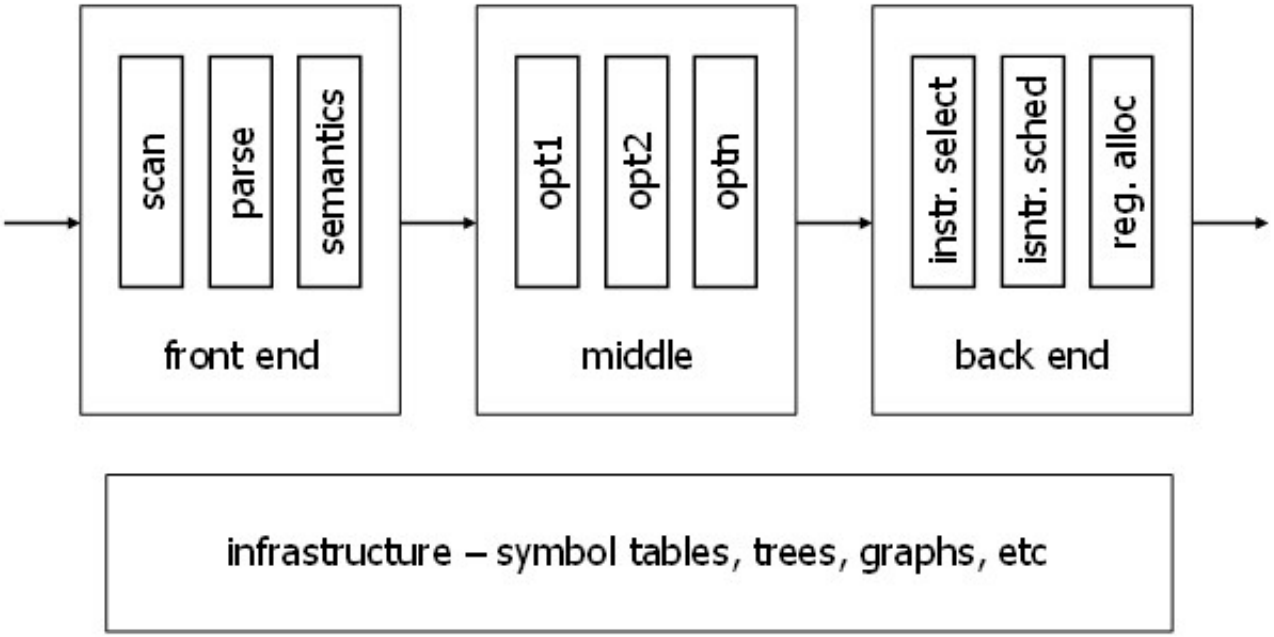
Spring 2018

Agenda

- Compiler back-end organization
- Instruction selection – tree pattern matching

- Credits: Slides by Keith Cooper (Rice); Appel ch. 9; burg/iburg slides by Preston Briggs, CSE 501 Sp09
- Burg/iburg paper: "Engineering a Simple, Efficient Code Generator", Fraser, Hanson, & Proebsting, ACM LOPLAS v1, n3 (Sept. 1992)

Compiler Organization



Big Picture

- Compiler consists of lots of fast stuff followed by hard problems
  - Scanner: O(n)
  - Parser: O(n)
  - Analysis & Optimization: ~ O(n log n)
  - Instruction selection: fast or NP-Complete
  - Instruction scheduling: NP-Complete
  - Register allocation: NP-Complete

IR for Code Generation

- Assume a low-level RISC-like IR
  - 3 address, register-register instructions + load/store

    r1 <- r2 op r3
  - Could be tree structure or linear
  - Expose as much detail as possible
- Assume "enough" (i.e., ∞) registers
  - Invent new temporaries for intermediate results
  - Map to actual registers later

Overview: Instruction Selection

- Map IR into assembly code
- Assume known storage layout and code shape
  - i.e., the optimization phases have already done their thing
- Combine low-level IR operations into machine instructions (take advantage of addressing modes, etc.)

Overview: Instruction Scheduling

- Reorder operations to hide latencies – processor function units; memory/cache
  - Originally invented for supercomputers (1960s)
  - Now important everywhere
    - Even non-RISC machines, i.e., x86
    - Even if processor reorders on the fly
- Assume fixed program

Overview: Register Allocation

- Map values to actual registers
  - Previous phases change need for registers
- Add code to spill values to temporaries as needed, etc.
- Usually worth doing another instruction scheduling pass afterwards if spill code inserted

How Hard?

- Instruction selection
  - Can make locally optimal choices
  - Global is undoubtedly NP-Complete
- Instruction scheduling
  - Single basic block – quick heuristics
  - General problem – NP Complete
- Register allocation
  - Single basic block, no spilling, interchangeable registers – linear
  - General – NP Complete

Conventional Wisdom

- We typically lose little by solving these independently
  - But not always, of course (iterating phases on x86[-64] can help because of limited registers, memory operands)
- Instruction selection
  - Use some form of pattern matching
  - ∞ virtual registers – create as needed
- Instruction scheduling
  - Within a block, list scheduling is close to optimal
  - Across blocks: EBBs or trace scheduling if list scheduling not good enough
- Register allocation
  - Start with unlimited virtual registers and map "enough" to K

An Simple Low-Level IR (1)

- Details not important for our purposes; point is to get a feeling for the level of detail involved
  - This example is from Appel
- Expressions
  - CONST(i) – integer constant i
  - TEMP(t) – temporary t (i.e., register)
  - BINOP(op,e1,e2) – application of op to e1,e2
  - MEM(e) – contents of memory at address e
    - Means value when used in an expression
    - Means address when used on left side of assignment
  - CALL(f,args) – apply function f to argument list args

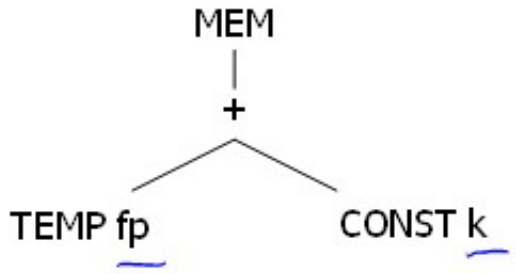
Simple Low-Level IR (2)

- Statements
  - MOVE(TEMP t, e) – evaluate e and store in temporary t
  - MOVE(MEM(e1), e2) – evaluate e1 to yield address a; evaluate e2 and store at a
  - EXP(e) – evaluate expressions e and discard result
  - SEQ(s1,s2) – execute s1 followed by s2
  - NAME(n) – assembly language label n
  - JUMP(e) – jump to e, which can be a NAME label, or more complex (e.g., switch)
  - CJUMP(op,e1,e2,t,f) – evaluate e1 op e2; if true jump to label t, otherwise jump to f
  - LABEL(n) – defines location of label n in the code

Low-Level IR Example (1)

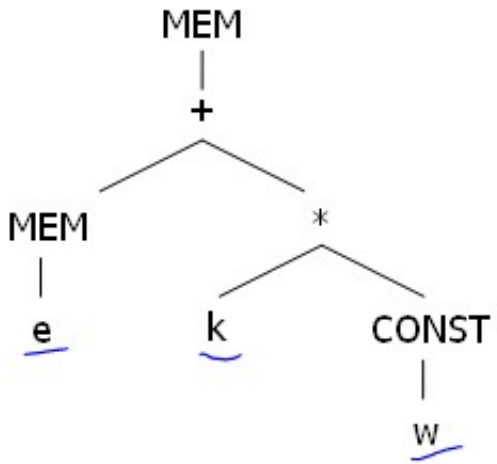
- Expr for a local variable at a known offset k from the frame pointer fp
  - Linear

    MEM(BINOP(PLUS, TEMP fp, CONST k))
  - Tree



Low-Level IR Example (2)

- For an array element e(k), where each element takes up w storage locations



Generating Low-Level IR

- Assuming initial IR is an AST, a simple treewalk can be used to generate the low-level IR
  - Can be done before, during, or after optimizations in the middle part of the compiler
    - Typically AST is lowered to some lower-level IR, but maybe not final lowest-level one used in instruction selection
- Create registers (temporaries) for values and intermediate results
  - Value can be safely allocated to a register when only 1 name can reference it
    - Trouble: pointers, arrays, reference parameters
  - Assign a virtual register to anything that can go into one
  - Generate loads/stores for other values

Instruction Selection Issues

- Given the low-level IR, there are many possible code sequences that implement it correctly
  - e.g. to set %rax to 0 on x86, among others…

```
movq  $0,%rax        xorq  %rax,%rax
subq  %rax,%rax      imulq  %rax,0
```

  - Many machine instructions do several things at once – e.g., register arithmetic and effective address calculation
    - leaq offset(%rbase,%rindex,scale),%rdst

Instruction Selection Criteria

- Several possibilities
  - Fastest
  - Smallest
  - Minimize power consumption (ex: don't use a function unit if leaving it powered-down is a win)
- Sometimes not obvious
  - e.g., if one of the function units in the processor is idle and we can select an instruction that uses that unit, it effectively executes for free, even if that instruction wouldn't be chosen normally
    - (Some interaction with scheduling here…)

Implementation

- Problem: We need some representation of the target machine instruction set that facilitates code generation
- Idea: Describe machine instructions using same low-level IR used for program
- Use pattern matching techniques to pick machine instructions that match fragments of the program IR tree
  - Want this to run quickly
  - Would like to automate as much as possible

Matching: How?

- Tree IR – pattern match on trees
  - Tree patterns as input
  - Each pattern maps to target machine instruction (or sequence)
    - and at least one simple target match for each kind of tree node so we can always generate something
  - Various algorithms – max. munch, dynamic programming, …
- Linear IR – some sort of string matching
  - Strings as input
  - Each string maps to target machine instruction sequence
  - Use text matching or peephole matching (parsing!, but with a way, way ambiguous grammar for target machine)
- Both work well in practice; algorithms are quite different

An Example Target Machine (1)

- Arithmetic Instructions – result in reg.

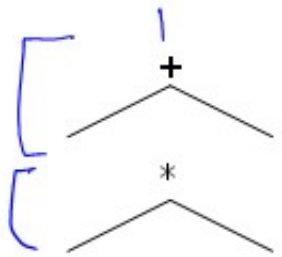
  (unnamed) ri

  ADD ri <- rj + rk

  MUL ri <- rj * rk

  SUB and DIV are similar

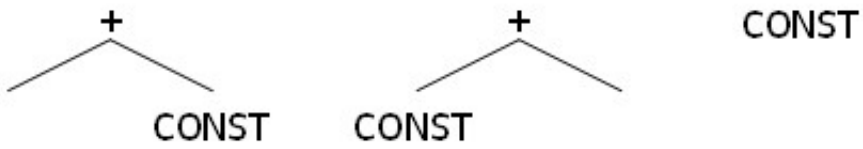
TEMP



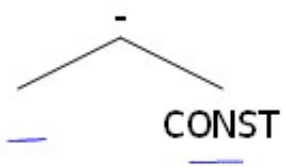
An Example Target Machine (2)

- Immediate Instructions

ADDI ri <- rj + c



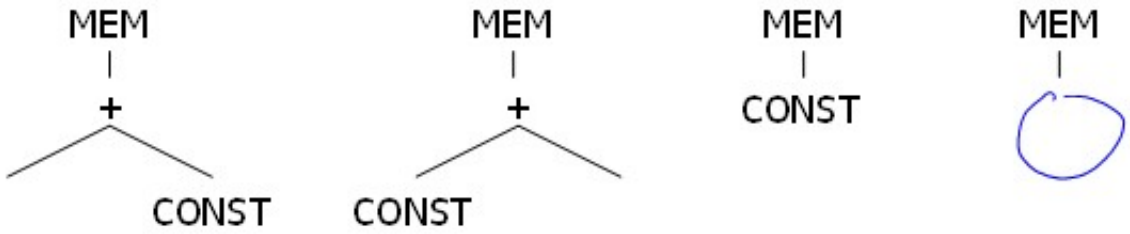
SUBI ri <- rj - c



An Example Target Machine (3)

- Load

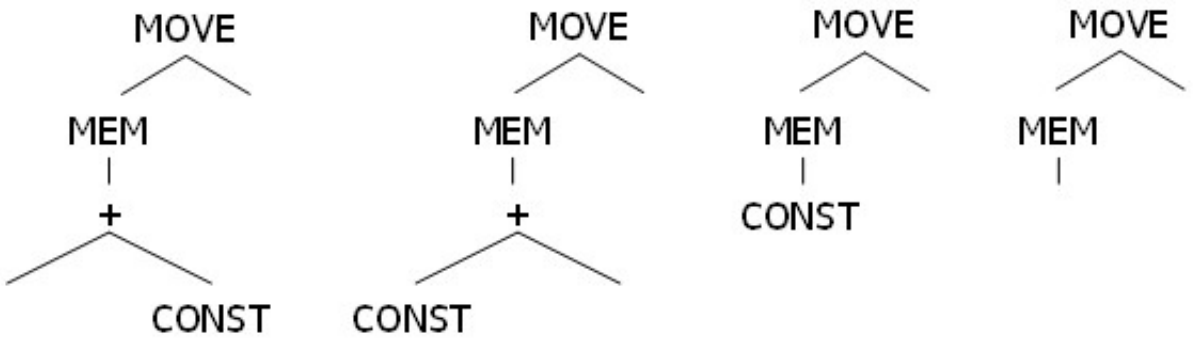
  LOAD ri <- M[rj + c]



An Example Target Machine (4)

- Store – not an expression; no result

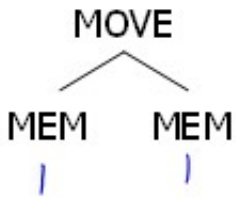
  STORE M[rj + c] <- ri



An Example Target Machine (5)

- mem->mem copy – also not an expression

  MOVEM M[rj] <- M[ri]



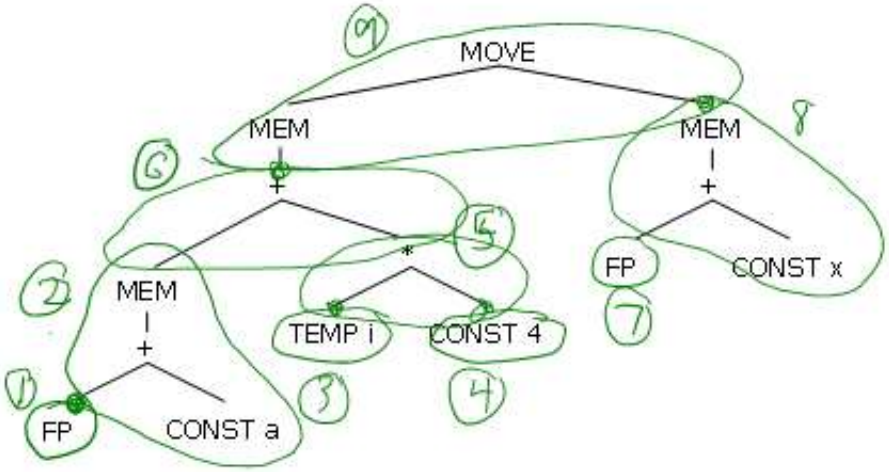
Tree Pattern Matching (1)

- Goal: Tile the low-level tree with operation (instruction) trees
- A *tiling* is a collection of <node,op> pairs
  - node is a node in the tree
  - op is an operation tree
  - <node,op> means that op could implement the subtree at node

Tree Pattern Matching (2)

- A tiling "implements" a tree if it covers every node in the tree and the overlap between any two tiles (trees) is limited to a single node
  - If <node,op> is in the tiling, then node is also covered by a leaf in another operation tree in the tiling – unless it is the root
  - Where two operation trees meet, they must be compatible (i.e., expect the same value in the same location)

Example – Tree for a[i]:=x



2. LOAD $r1 \leftarrow mem[fp + a_{off}]$
4. ADDI $r2 \leftarrow 4 + r0$
5. MUL $r2 \leftarrow r2 * r_i$
6. ADD $r1 \leftarrow r1 + r2$
8. LOAD $r2 \leftarrow MEM[fp + x_{off}]$
9. STORE $M[r1 + 0] \leftarrow r2$

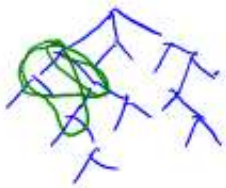
Generating Code

- Given a tiled tree, to generate code
  - Postorder treewalk; node-dependant order for children
  - Emit code sequences corresponding to tiles in order
  - Connect tiles by using same register name to tie boundaries together

Tiling Algorithms (1)

- Maximal Munch
  - Start at root of tree, find largest tile that fits. Cover the root node and possibly other nearby nodes. Then repeat for each subtree
  - Generate instruction as each tile is placed
    - Generates instructions in reverse order
  - Generates an optimum tiling – tiles sum to lowest possible cost
    - But not optimal – there may be another tiling where two adjacent tiles can be combined into one of lower cost

Tiling Algorithms (2)



- Dynamic Programming
  - There may be many tiles that could match at a particular node
  - Idea: Walk the tree and accumulate the set of all possible tiles that could match at that point – Tiles(n)
  - Then: Select minimal cost for subtrees (bottom up), and go top-down to select and emit lowest-cost instructions

Tile(Node n)

```
Tiles(n) <- empty;
if n has two children then
    Tile(left child of n)
    Tile(right child of n)
    for each rule r that implements n
      if (left(r) is in Tiles(left(n)) and right(r) is in Tiles(right(n)))
      Tiles(n) <- Tiles(n) + r
else if n has one child then
    Tile(child of n)
    for each rule r that implements n
      if(left(r) is in Tiles(child(n)))
      Tiles(n) <- Tiles(n) + r
else /* n is a leaf */
    Tiles(n) <- { all rules that implement n }
```

Tools

- iburg and burg and others use a combination of dynamic programming and tree pattern matching to find optimal translations
- Product of years of research going back to peephole optimizers
- Not really a research area now (just like parsing), but still room for newer/better tools

- Following slides skipped 18sp but retained for anyone who is interested in exploring further (see burg/iburg papers)

Peephole Optimization

- Idea: Find pairs of instructions (not necessarily adjacent) and replace them with something better
- Instead of

```
t2 = &c
t3 = *t2
```

use

```
t3 = c
```

Which Pairs?

- Chris Fraser noticed that useful pairs were connected by DU chains, so
- Plan: consider each instruction together with instructions that feed it
- DU chains reach across blocks, so instruction selector can work globally
  - Works great with SSA too

Patterns

- Reduce pairs of instructions to schematics

  t5 = 4

  t6 = t4 + t5

  becomes

  %reg1 = %con

  %reg3 = %reg2 + %reg1

- Find in hash table; if found, replace

  %reg3 = %reg2 + %con

Tree-Based Representation

- The tree makes the DU chains explicit
- Each definition in the tree is used once
- Typically a basic block would have a sequence of expression trees

Example

Code for i = c + 4
[c a char; i an int]

t1 = &i

t2 = &c

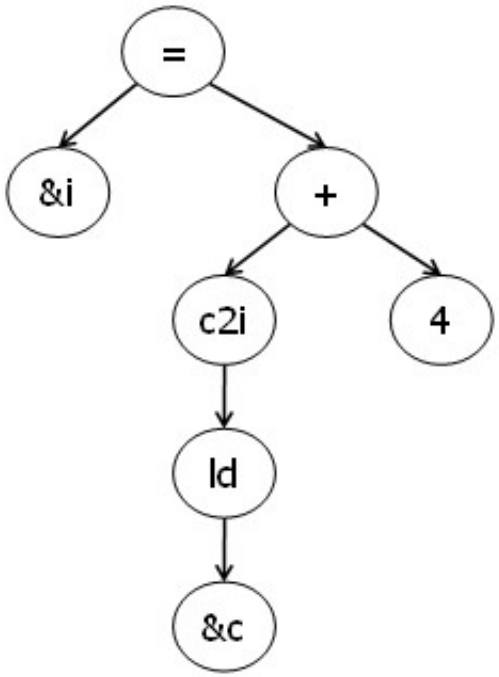
t3 = *t2

t4 = cvci(t3)

t5 = 4

t6 = t4+t5

*t1 = t6



Tree Patterns

- An iburg spec is a set of rules. BNF:

  rule = nonterm : tree = integer (cost)

  tree = term (tree , tree)

  | term (tree)

  | term

  | nonterm

- Terminals are IL operations
- Nonterminals name sets of rules

Rules

- Rules are numbered to the right of the = sign
- The cost of a rule is its cost (often 0) plus the cost of any subtrees
- A rule may be nested to define a pattern that matches more than one level in a tree

Example

```
stmt    : ASSIGN(addr, reg) = 1 (1)
stmt    : reg = 2 (0)
reg     : ADD(reg, rc) = 3 (1)
reg     : ADD(rc, reg) = 4 (1)
reg     : LD(addr) = 5 (1)
reg     : C2I(LD(addr)) = 6 (1)
reg     : addr = 7 (1)
reg     : con = 8 (1)
addr    : ADD(reg, con) = 9 (0)
addr    : ADD(con, reg) = 10 (0)
addr    : ADDRLP = 11 (0)   // addr of local var
rc      : con = 12 (0)
rc      : reg = 13 (0)
con     : CNST = 14 (0)
```

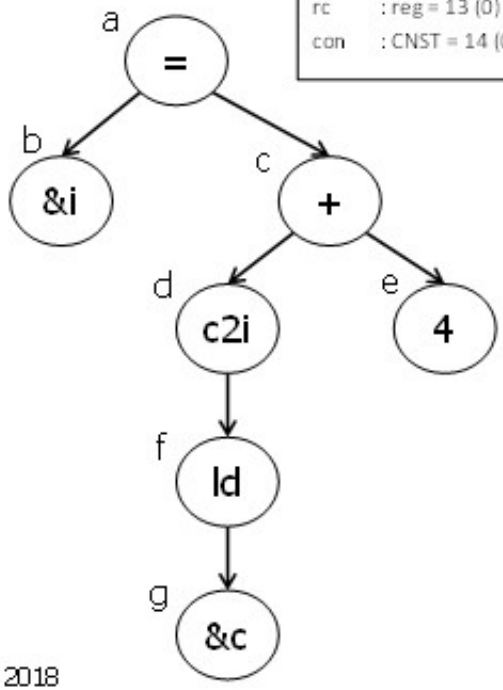
Algorithm

- Pass I: bottom up
  - Label each node with lowest cost rule to produce result from available operands
    (cost = cost of rule + cost of subtrees)
  - Label is: (r, c) = best is rule r, cost is c
- Pass II: top down
  - Find cheapest node that generates result needed by parent tree
  - Emit instructions in reverse order as choices are made

Bottom-up (Labeling)

op	stmt	reg	addr	rc	con
a					
b					
c					
d					
e					
f					
g					

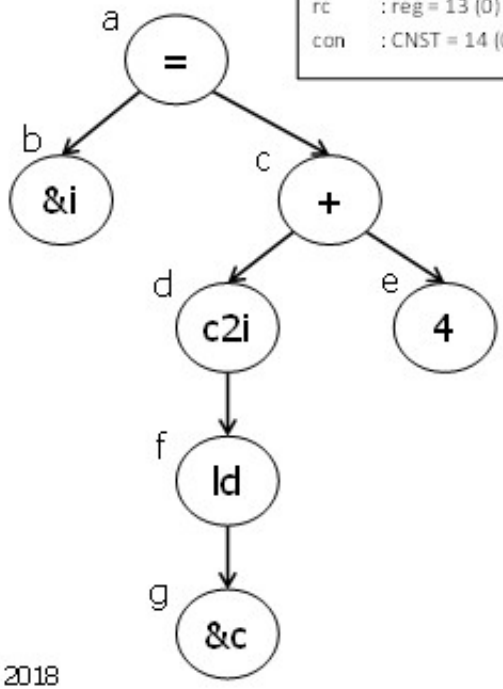
```
stmt : ASSIGN(addr, reg) = 1 (1)
stmt : reg = 2 (0)
reg  : ADD(reg, rc) = 3 (1)
reg  : ADD(rc, reg) = 4 (1)
reg  : LD(addr) = 5 (1)
reg  : C2I(LD(addr)) = 6 (1)
reg  : addr = 7 (1)
reg  : con = 8 (1)
addr : ADD(reg, con) = 9 (0)
addr : ADD(con, reg) = 10 (0)
addr : ADDRLP = 11 (0)
rc   : con = 12 (0)
rc   : reg = 13 (0)
con  : CNST = 14 (0)
```



Top-Down (Reduction)

```
stmt : ASSIGN(addr, reg) = 1 (1)
stmt : reg = 2 (0)
reg  : ADD(reg, rc) = 3 (1)
reg  : ADD(rc, reg) = 4 (1)
reg  : LD(addr) = 5 (1)
reg  : C2I(LD(addr)) = 6 (1)
reg  : addr = 7 (1)
reg  : con = 8 (1)
addr : ADD(reg, con) = 9 (0)
addr : ADD(con, reg) = 10 (0)
addr : ADDRLP = 11 (0)
rc   : con = 12 (0)
rc   : reg = 13 (0)
con  : CNST = 14 (0)
```

op	stmt	reg	addr	rc	con
a	(1,3)				
b	(2,1)	(7,1)	(11,0)	(13,1)	
c	(2,2)	(3,2)	(9,1)	(13,2)	
d	(2,1)	(6,1)		(13,1)	
e	(2,1)	(8,1)		(12,0)	(14,0)
f	(2,1)	(5,1)		(13,1)	
g	(2,1)	(7,1)	(11,0)	(13,1)	



Coming Attractions

- Instruction Scheduling
- Register Allocation
- And more….