CMPT 379
Compilers

Anoop Sarkar

http://www.cs.sfu.ca/~anoop

Code Generation

- Instruction selection
- Register allocation
- Stack frame allocation √
- Static or global allocation √
- Basic blocks and Flow graphs
- Transformations on Basic blocks

Code Generation

- Produce code that is correct
- Produce code that is of high quality (size and speed)
- Mathematically, the problem of generating optimal code is undecidable
- In practice, we need heuristics that generate good, but perhaps not optimal, code

Instruction Costs

- Since optimal code generation is not possible a useful way to think about the problem is as an *optimization* problem
- Each instruction can be assigned a cost
  - For complex instruction sets some instructions can be more preferable than others
- Using registers have zero cost, while using memory locations is costlier
- If each instruction is equally expensive, this will minimize the number of instructions as well

Register Allocation

- Code generation either directly to assembly or from 3-address code (TAC)
- For each location, we have to find a register to store values or temporary values
  - Problem: limited number of registers
- Compiler has to find optimal assignment of locations to registers
  - Register use can involve stacked temporaries or other ways to reuse registers
- If no more registers available, we *spill* a location into memory

Register Allocation

- Bind locations to registers for all or part of a function
- Dynamic Optimization Problem
  - Not compile-time, but run-time frequency is what counts
- Heuristics
  - Allocate registers for variables likely to be used frequently
  - Keep temporaries in registers → minimize their number

Basic Blocks

- Functions transfer control from one place (the caller) to another (the called function)
- Other examples include any place where there are branch instructions
- A *basic block* is a sequence of statements that enters at the start and ends with a branch at the end
- Remaining task of code generation is to create code for basic blocks and branch them together

Blocks

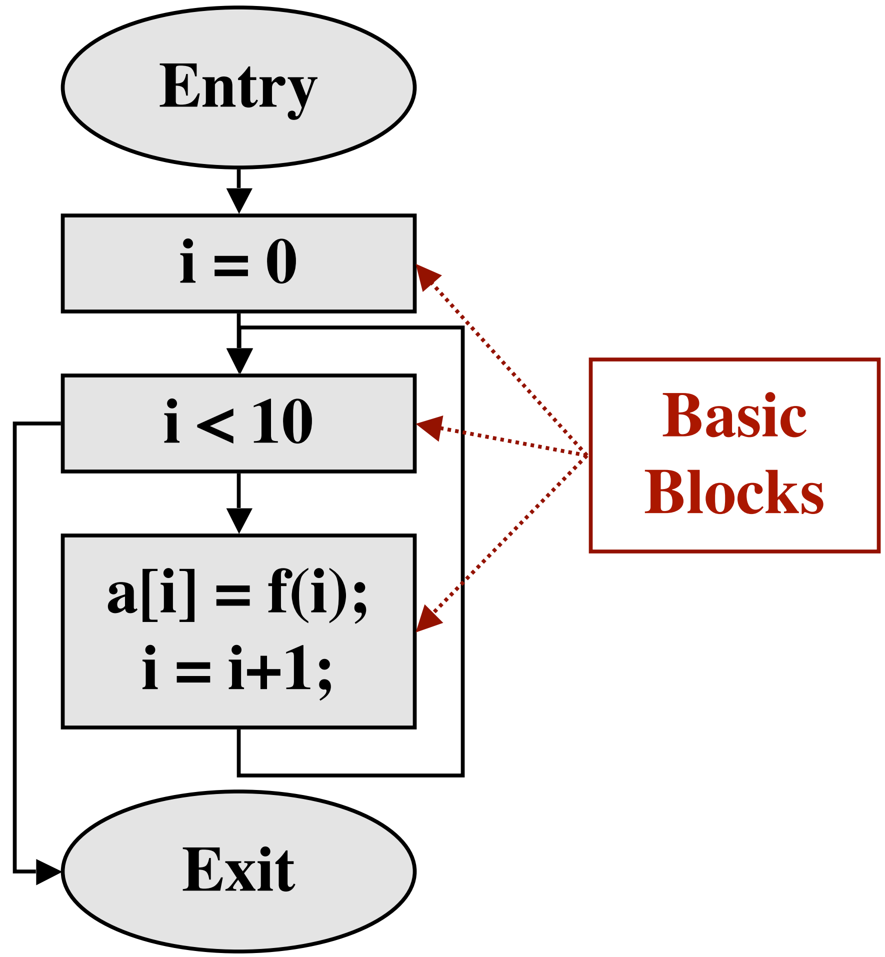
```
main()
{
    int a = 0; int b = 0;
    {
        int b = 1;
        {
            int a = 2; printf(“%d %d\n”, a, b);
        }
        {
            int b = 3; printf(“%d %d\n”, a, b);
        }
        printf(“%d %d\n”, a, b);
    }
    printf(“%d %d\n”, a, b);
}
```

Partition into Basic Blocks

- Input: sequence of TAC instructions
  1. Determine set of leaders, the 1st statement of each basic block
     a) The 1st statement is a leader
     b) Any statement that is the target of a conditional jump or goto is a leader
     c) Any statement immediately following a conditional jump or goto is a leader
  2. For each leader, the basic block contains all statements upto the next leader

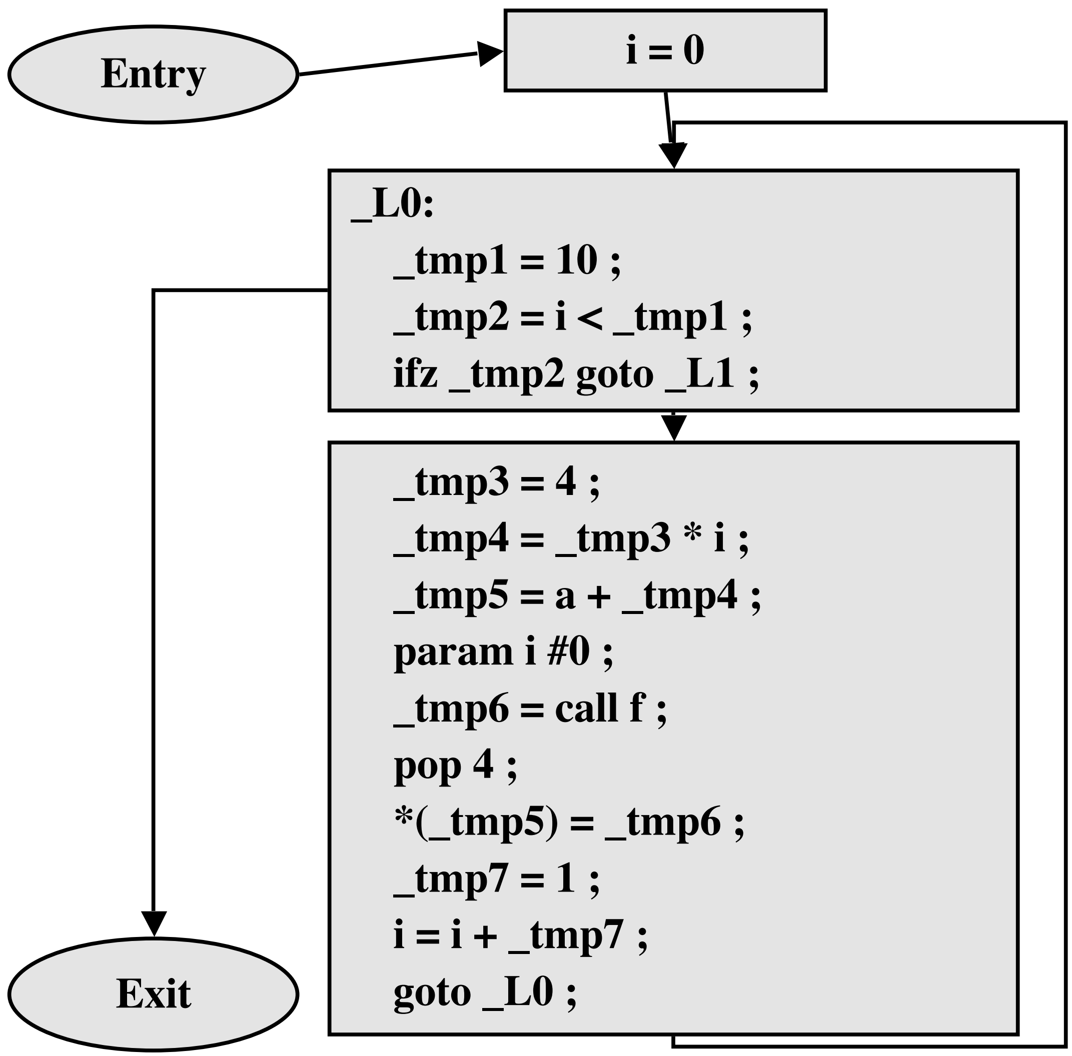
Control Flow Graph (CFG)

```
int main() {
 extern int f(int);
 int i;
 int *a;
 for (i = 0;
         i < 10;
            i = i + 1)
     { a[i] = f(i); }
}
```



Control Flow Graph in TAC

```
main:
   BeginFunc 72 ;
   i = 0 ;
_L0:
   _tmp1 = 10 ;
   _tmp2 = i < _tmp1 ;
   IfZ _tmp2 Goto _L1 ;
   _tmp3 = 4 ;
   _tmp4 = _tmp3 * i ;
   _tmp5 = a + _tmp4 ;
   param i #0 ;
   _tmp6 = call f ;
   pop 4 ;
   *(_tmp5) = _tmp6 ;
   _tmp7 = 1 ;
   i = i + _tmp7 ;
   goto _L0 ;
_L1:
   EndFunc ;
```



Dataflow Analysis

- Compute Dataflow Equations over Control Flow Graph
  - Reaching Definitions (Forward)

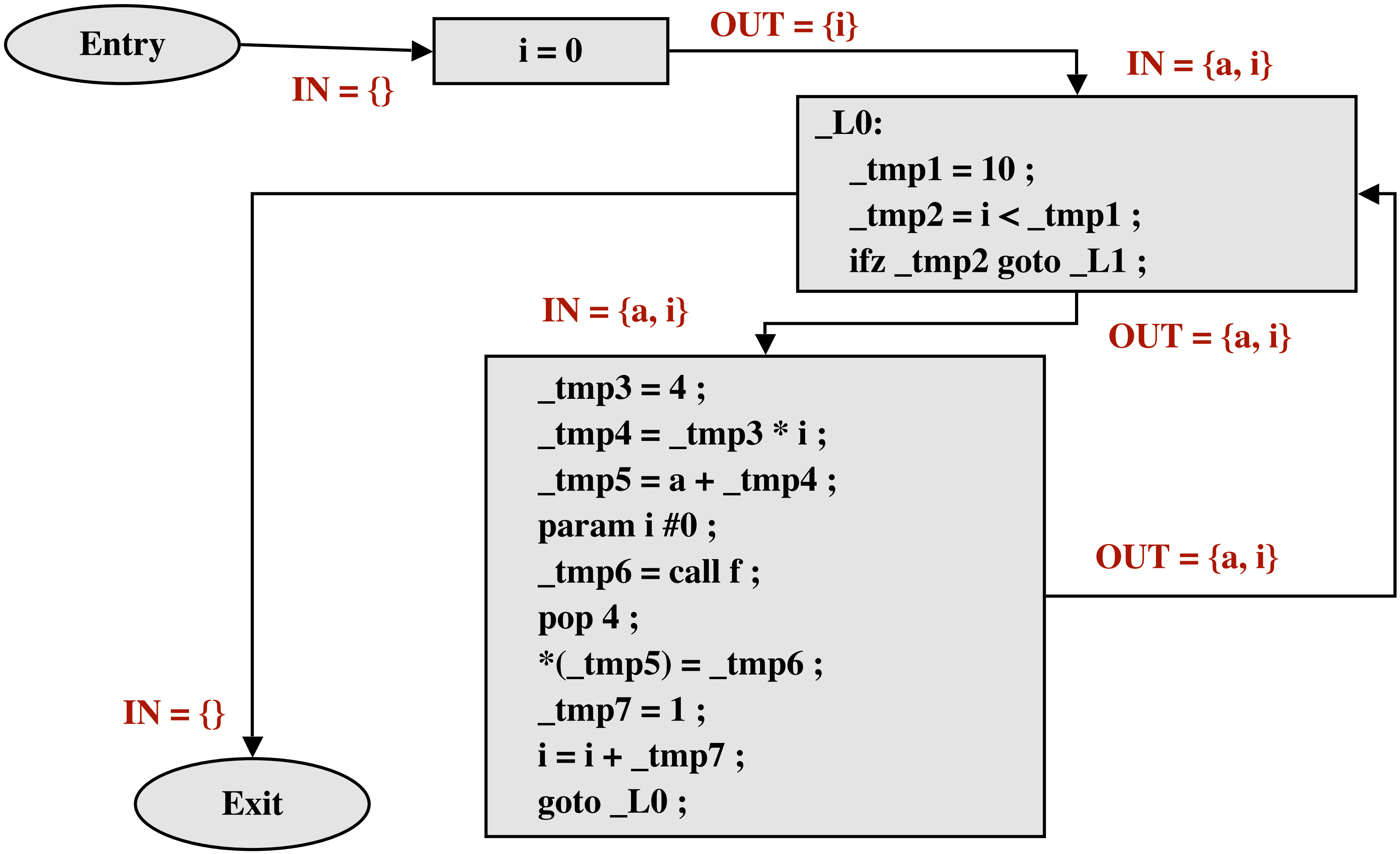
    out[BB] := gen[BB] ∪ (in[BB] – kill[BB])

    in[BB] := ∪ out[s] : forall s ∈ pred[BB]
  - Liveness Analysis (Backward)

    in[BB] := use[BB] ∪ (out[BB] – def[BB])

    out[BB] := ∪ in[s] : forall s ∈ succ[BB]
- Computation by fixed-point analysis

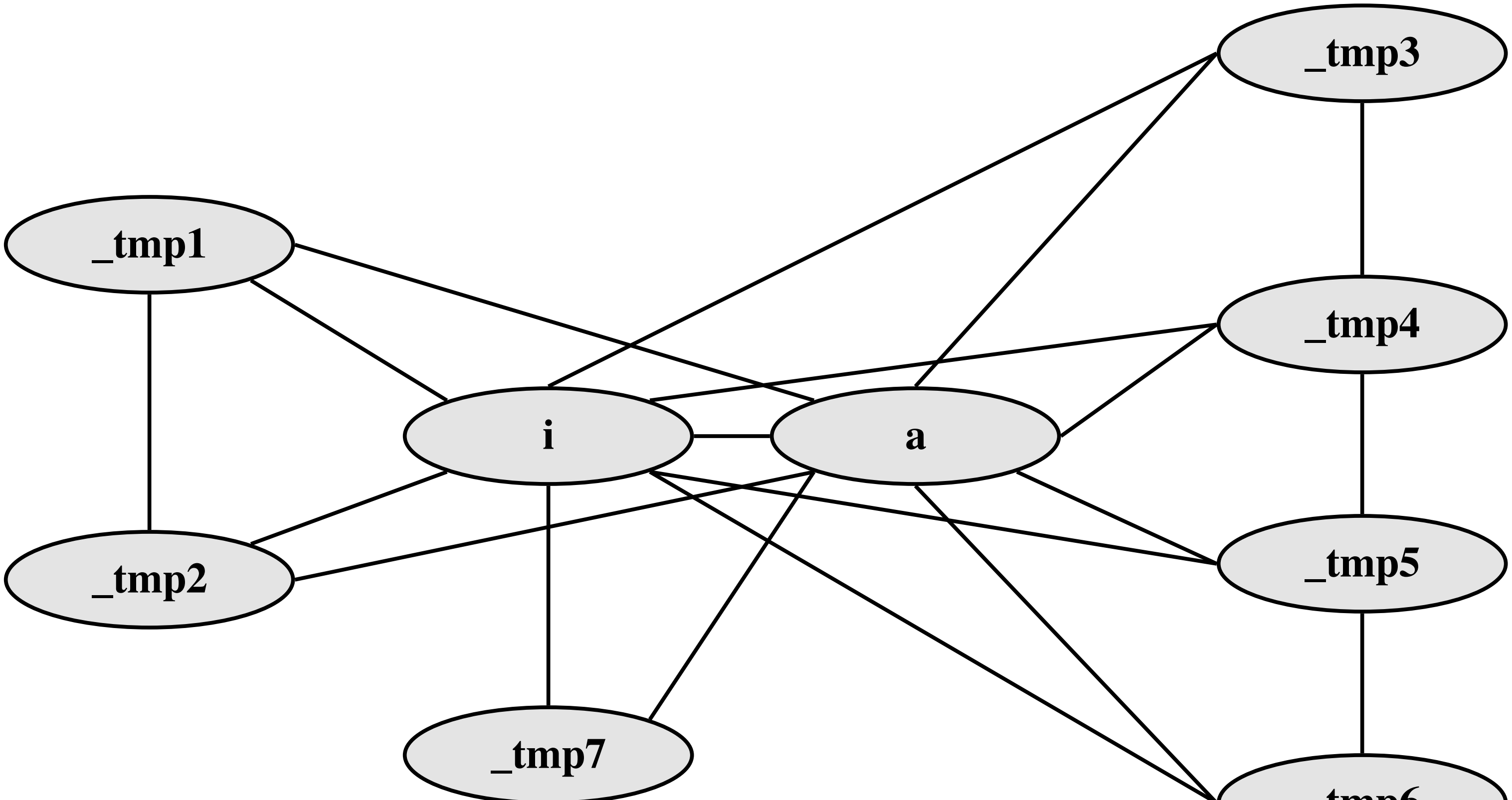
Liveness Analysis



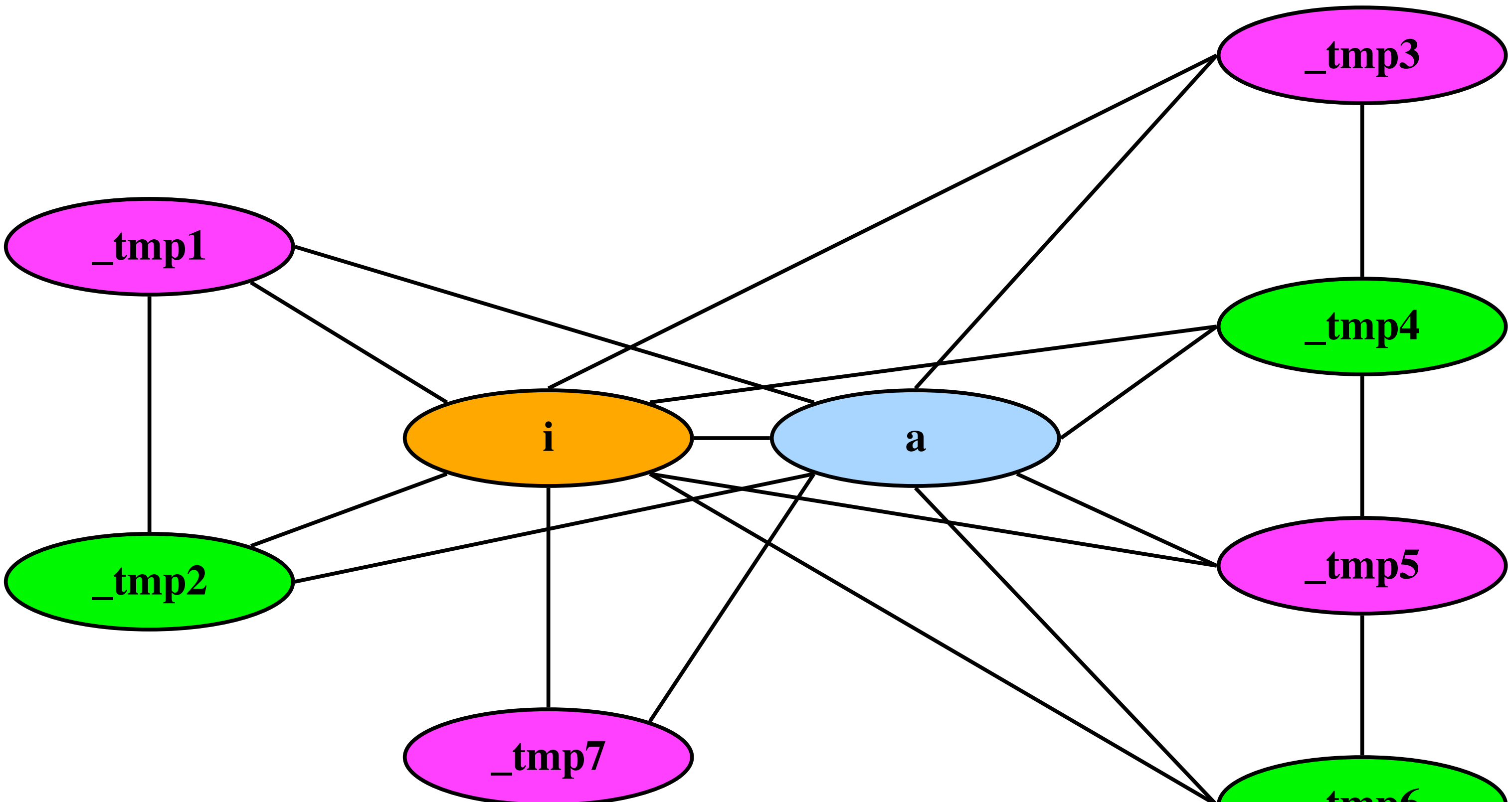
Register Allocation

- Do liveness analysis on Control Flow Graph
  - Straightforward (iteration-less) computation within basic block
  - Compute live ranges for each location
- Build interference graph
  - Two locations are connected if their live ranges overlap
- Color interference graph
  - Result is register assignment

Interference Graph



Colored Interference Graph



Register Allocation as Graph Coloring

- First pass: use as many symbolic registers as needed including registers for stack pointers, frame pointers, etc.
- Second pass: assign physical registers to symbolic ones
  - Construct a ***register interference graph*** (nodes are symbolic registers and edge denotes that they cannot be assigned to the same physical register)
  - Attempt to ***k*-color** the interference graph, where *k* is the number of available registers
  - *k*-coloring a graph is NP-complete

Register Allocation as Graph Coloring

- Algorithm for solving whether a graph G is *k*-colorable:
- Pick any node *n* with fewer than *k* neighbours
- Remove *n* to create a new graph G'
- *k*-coloring of G' can be extended to *k*-coloring to G by assigning to *n* a color that is not assigned to any of *n*'s neighbours
- If we cannot extend G' to G, then *k*-coloring of G is not possible

Register Allocation as Graph Coloring

- If a node *n* in G has more than *k* neighbours, *k*-coloring of G is not possible
- Take the node *n* and spill into memory, remove it from the graph and continue *k*-coloring
- Many different heuristics for picking a node n to spill
  - E.g. avoid introducing spilling symbolic registers that are inside loops or heavily visited regions of code

Transformations on Basic Blocks

- Structure preserving transformations
  1. Common subexpression elimination

     _a := _b + _c

     _b := _a - _d

     _c := _b + _c

     _d := _a - _d ($\Rightarrow$ _b)
  2. Dead-code elimination
  3. Renaming temporary variables
  4. Interchange of statements

Transformations on Basic Blocks

- Algebraic transformations

  _d := _a + 0 (⇒ _a)

  _d := _d * 1 (⇒ *eliminate*)

- Reduction of strength

  _d := _a ** 2 (⇒ _a * _a)

Code-generation Generators

- Code generation by Tree Rewriting
  - Use output of syntax parsing
  - Write tree patterns that match portions of the parse tree
  - Each tree pattern is associated with code
  - Also uses costs (similar to instruction costs) for optimizing code generation
  - Several tools available: *burs*, *iburg*, *lcc*

Code-generation Generators

- Code generation by Tree Parsing
  - Take the prefix representation of the syntax tree
  - E.g. (+ (* c_1 r_1) (+ m_a c_2)) in prefix representation uses an inorder traversal to get + * c_1 r_1 + m_a c_2
  - Write CFG rules that match substrings of the above representation and non-terminals are registers or memory locations
  - The attribute syntax-directed defn is used to produce the code