CSE 4251: Compiler Construction

Ryan Stansifer
Department of Computer Sciences
Florida Institute of Technology
Melbourne, Florida USA 32901

http://www.cs.fit.edu/~ryan/

computer-presentation edition

March 26, 2019
Review

• Regular expressions to NFA
• NFA to DFA
• Nullable, first, and follow
• LL(1) parsing
• LR(0), SLR, LR(1), LALR(1) parsing
More Generally

- Definition of formal language, regular expression
- Recursive descent parsers
- Scanners versus recognizers
- Definition of grammars, parse trees, ambiguity
- Hierarchy of formal languages
Overview of Assignment 5

- Study AST for MiniJava
- Add to JavaCC parser semantic actions to create AST
- Understand visitor pattern
- Design symbol table
- Code visitor to create symbol table
- Code visitor to perform semantic checking.
Chapter 6: Activation Records
Activation Records

Because of recursion, several invocations of a subroutine may exist simultaneously.

Each invocation of a subroutine has its own instances of the local variables lasting from subroutine entry to return.

A subroutine invocation requires the runtime system maintain information about the invocation including the values of local variables (and much more: return address, saved state of the machine, temporaries values, access to the non-local environment, \ldots).

This collection of information is called an activation record.
A subroutine returns only after all the function it has called have returned. Thus subroutine invocations behave in a LIFO manner. So keep activation records in a stack.
Activation Records

Appel, 2nd edition, page 118. “For historical reasons, the run-time stack usually starts at a high memory address . . .”

Runtime Organization of a Program in Memory

- **High Memory**: The stack (local variables)
- **Low Memory**: The heap (dynamically allocated variables), globals variables, program code (read only)
Stack of Activation Records

high memory

FFFF

previous

current

next

low memory

........

0000
Static link
Dynamic link
Return address
Parameters
Local variables
Register save area
Usual Layout of Activation Record for SPARC architecture
On an architecture with register windows, such as SPARC . . . if we choose the stack pointer to be one of the out registers and the frame point to be the corresponding in register, as the SPARC UNIX System V API specifies, then the save and the restore instructions can be used to perform the entry and exit operations, with saving registers to memory and restoring left to the register-window spill and fill trap handlers.

Muchnick, page 113.
SPARC calling sequence

1. Caller puts arguments in %o0, %o1, ..., %o5. For example:
   
   mov 1,%o0 ! pass argument 0 in the register %o0
   set LABEL,%o1 ! pass argument 1 in the register %o1
   mov %l4,%o2 ! pass argument 2 in the register %o2

2. If more than six arguments, then the caller puts them in the (caller’s) argument build area. For example:
   
   mov 6,[%sp+4*(16+6)] ! pass argument 6 on the stack
   set LABEL,[%sp+4*(16+7)] ! pass argument 7 on the stack

3. Caller then executes call instruction, which saves the pc in %o7 and jumps to a label. For example:
   
   call f
   nop
4. Callee executes **save** instruction, which shifts the register window set and reserves space for new stack frame. For example:

```
save %sp, -4*(16+1+t+x)&-8, %sp
```

where \( t \) is the number of locals and temporaries, and \( x \) is the maximum arguments needed by any subroutines of the callee. Stack pointers must be double word aligned.

5. If needed, the callee saves the incoming arguments (0 through 5) into the argument build area of the caller’s frame. For example:

```
st %i0, [%fp+4*(16+0)] ! store arg 0 into stack
st %i1, [%fp+4*(16+1)] ! store arg 1 into stack
```

Local variables and temporaries of the current frame are accessed with negative offsets from the %fp. For example:

```
ld [%fp-4*(1+0)], %l0 ! load local/temp 0
ld [%fp-4*(1+1)], %l1 ! load local/temp 1
```
SPARC return sequence

1. Callee puts (integer) return value in %i0. For example:
   mov 1,%i0  ! put return value in %i0

2. Callee executes restore instruction, which resets the register window set and pops frame.

3. Callee then executes ret instruction, which jumps to %i7+8 (just past the call and its delay slot in the caller). For example:
   ret      ! "jmp1 %i7+8, %g0" after next instruction
   restore  ! restore register window set
Nonlocal variable access

1. Whenever a function \( f \) is called, it can be passed a pointer to the frame of the function statically enclosing \( f \); this pointer is the *static link*.

2. A global array can be maintained, containing – in position \( i \) – a pointer to the frame of the most recently entered procedure whose *static nesting depth* is \( i \). This array is called a *display*.

3. When \( g \) calls \( f \), each variable of \( g \) that is actually accessed by \( f \) (or by any function nested inside \( f \)) is passed to \( f \) as an extra argument. This is called *lambda lifting*. 
Nonlocal Variable Access

A local variable is accessed as follows:

\[ r := \text{fp} \quad ! \text{relative to frame pointer} \]
\[ r := M[r+offset] \quad ! \text{access local variable} \]

A non-local variable at static distance two is accessed as follows:

\[ r := \text{fp} \quad ! \text{relative to frame pointer} \]
\[ r := M[r+sl] \quad ! \text{follow first static link} \]
\[ r := M[r+sl] \quad ! \text{follow second static link} \]
\[ r := M[r+offset] \quad ! \text{access the variable at sd=2} \]
Nonlocal Variable Access

\[ M[fp + (k)] \]

\[ MEM (BINOP (PLUS, \new TEMP (frame.FP()), CONST ((k)))) \]
line 21. prettyprint calls show
   (show’s SL is set to prettyprint’s own frame pointer)

line 15. show calls indent
   (indent’s SL is set to show’s own frame pointer)

line 13. indent calls write

line 17. show calls itself recursively

line 15. show calls indent

line 13. indent calls write
line 14. indent accesses the variable output
An abstract structure to represent frame information

```java
package frame;

/* Appel, 2nd edition, Chapter 6, page 127 */
public abstract class Frame {
    public final Label instructionPointer;
    public final List<Access> formals; // including implicit static

    public Frame (Label i, List<? extends Access> f) {
        instructionPointer = i; formals = f;
    }

    public int getNumberOfFormals () { return formals.size(); }
    abstract public Frame newFrame (Label name, List formals);
    abstract public Access allocateLocal (boolean escape);
    public Access allocateLocal () { return allocateLocal(true); }

    public abstract temp.Temp FP(); /* Chapter 7, page 143 */
}
```
package frame;
import tree.Exp;

/*
*/
public abstract class Access {
    public abstract Exp access (Exp framePtr);
}
The class Frame holds information about formal parameters and local variables allocated in this frame. To make a new frame for a function $f$ with $k$ formal parameters, call \texttt{newFrame}(f, l), where $l$ is a list of $k$ booleans: \texttt{true} for each parameter that escapes and \texttt{false} for each parameter that does not. (In MiniJava, no parameters ever escape.) The result will be a Frame object.


Calling \texttt{newFrame()} and getting a new frame might change the caller’s frame. For example, the maximum size of the argument build area may increase.
package temp;
public class Temp {}
public class Label {}

new temp.Label ("ClassName"+$"+"methodName")
package translate;
import frame.Frame;
import translate.Access;
import java.util.LinkedList;

public class Level {
    public final Level parent;
    public final Frame frame;
    public Level (Level p, String name, LinkedList formals);
    public Access allocateLocal (boolean escape);
    public Access allocateLocal () { return allocateLocal(true); }
    public Access allocateLocal () { return allocateLocal(true); }
}

// Obsolete?
// Appel, 1st edition, page 147
Overview of Assignment 6

- Review the visitor pattern
- Review AST for MiniJava
- Study the given IR code in chapter 7
- Fix the design of the symbol table
- Detect missing semantics errors
- Study chapter 6
- Advise: use a \texttt{sparc} package, but ignore the books abstract classes
- Code visitor to translate to IR code
- Use or don’t use “generic IR trees”
Chapter 7: Translation to Intermediate Code
Intermediate Representation (IR)

- IR should be easy to convert from the abstract syntax; easy to convert to assembly. It should support machine-independent optimizations.

- Often compilers use several IRs.

$$
syntax \Rightarrow IR_1 \Rightarrow IR_2 \ldots \Rightarrow IR_k \Rightarrow \text{assembly code}
$$

Figure 6.2: A compiler might use a sequence of intermediate representations

- MiniJava compiler uses only one IR: `tree.Exp`
An intermediate language helps modularize the back end of the compiler.

7.1. INTERMEDIATE REPRESENTATION TREES

Diagram showing compilers for five languages and four target machines: (a) without an IR, (b) with an IR.
Figure 10.3: A middle-end and its ILs simplify construction of a compiler suite that must support multiple source languages and multiple target architectures.

GCC includes two ILs, one that represents source programs at a relatively high level and another that represents machine instructions abstractly. The Microsoft compiler suite uses Common Intermediate Language (CIL) as an IL and the Common Language Runtime (CLR) as a generic interpreter of CIL.
abstract class Exp
    class CONST (int value)
    class NAME (tree.Label label)
    class TEMP (tree.NameOfTemp temp)
    class BINOP (int binop, Exp left, Exp right)
    class MEM (Exp exp)
    class CALL (Exp fun, ExpList args)
    class ESEQ/RET (Stm stm, Exp exp)

abstract class Stm
    class MOVE (Exp dst, Exp src)
    class EVAL (Exp exp)
    class JUMP (Exp exp, List<tree.Label> targets)
    class CJUMP (int rel, Exp l, Exp r, Label t, Label f)
    class SEQ (Stm left, Stm right)
    class LABEL (tree.NameOfLabel label)

[RET is more pronouncible than ESEQ.]
It is almost possible to give a formal semantics to the Tree language. However, there is no provision in this language for procedure and function definitions – we can specify only the body of each function. The procedure entry and exit sequences will be added after as special “glue” that is different for each target machine.

Appel, 2nd, Section 7.1, page 139.
Intermediate Representation

Expressions which stand for the computation of some value (possibly with side effects):

CONST(i) The integer constant i.

NAME(n) The symbolic constant n corresponding to an assembly language label.

Temp_As_Exp(t) Temporary t. A temporary in the abstract machine is similar to a register in a real machine. However, the abstract machine has an infinite number of temporaries.

BINOP() The application of binary operator o to operands e1 and e2. Subexpression e1 is evaluated before e2.
Intermediate Representation

The remaining expressions which stand for the computation of some value (possibly with side effects):

\textbf{MEM(e)} The contents of a word of memory at address \textit{e}.

\textbf{CALL()} Procedure call.

\textbf{RET(s,e)} The statement \textit{s} is evaluated for side effects, then \textit{e} is evaluated for a result. Called \textbf{ESEQ(s,e)} by Appel.
Intermediate Representation

The statements of the intermediate representation which perform side effects and control flow:

\textbf{MOVE(TEMP \ t,e)} Store the results of evaluating \( e \) into the temporary \( t \).

\textbf{MOVE(MEM \ e_1,e_2)} Store the results of evaluating \( e_2 \) at the address \( e_1 \).

\textbf{EVAL(e)} Evaluate \( e \) for its side effects and discard the result.
(Called \textsc{EXP} by Appel.)

\textbf{JUMP(e,l)} Transfer control to address \( e \).
Intermediate Representation

The remaining statements which perform side effects and control flow:

**CJUMP** Evaluate $e_1$ and $e_2$ in that order; compare using relational operator $o$. If the result is true, jump to $t$, otherwise jump to $f$.

**SEQ**($s_1,s_2$) The statement $s_1$ followed by $s_2$.

**LABEL**($n$) Define name to be the current machine code address.
Example

new SEQ (new MOVE (temp,1), // temp := 1;
new SEQ (new CJUMP(<,x,5,T,F), // if x<5 goto T else F
new SEQ (new LABEL(F), // F:
new SEQ (new MOVE (temp,0), // temp := 0;
    new LABEL(T) // T:
)))))
Example 1

Abstract Syntax

\text{OpExp}(\text{PLUS,}\
\text{IntExp}(3),\
\text{IntExp}(4))

Intermediate Trees

\text{BINOP}(\text{PLUS,}\
\text{CONST}~3,\
\text{CONST}~4)
Example 2

Appel’s factorial program

```java
public int ComputeFac(int num) {
    int num_aux = 0;
    if (num < 1)
        num_aux = 1;
    else
        num_aux = num * (this.ComputeFac(num-1));
    return num_aux;
}
```

Factorial-07.txt
Lazy IR Trees

How do you generate good IR code bottom-up in a tree traversal?

The problem is the code you want to produce depends on the context in which it is used. The boolean conjunctions are the most notable example. In some contexts the code must produce a 0 (false) or a 1 (true). In other contexts the code is used to control a test.
Lazy IR Trees

A solution

Do not produce the IR tree directly. Instead use a class that will make the tree later, when it is known which one of three contexts is desired.
package translate;
class LazyIRTree { 
    abstract Exp asExp();
    abstract Stm asStm();
    Stm asCond (Label t, Label f) { throw new UnsupportedOperation(); }
    public String toString () {
        return String.format ("IR: %s", asStm().toString());
    }
}

Three views of the code:

1. as an expression
2. as a statement
3. as a conditional
“The whole point of the Cx representation is that conditional expressions can be combined easily with the MiniJava operator &&.” Appel, 2nd, Section 7.2, page 149.
Lazy IR Trees

class ExpIRTree extends LazyIRTree

class StmIRTree extends LazyIRTree

abstract class Cx extends LazyIRTree
    class RelCx extends Cx /* Page 149. */
Lazy IR Trees

class ExpIRTree extends LazyIRTree {
    private final tree.Exp exp;
    ExpIRTree (tree.Exp e) { exp = e; }
    tree.Exp asExp() { return exp; }
    tree.Stm asStm() { return new tree.EVAL(exp); }
    // asCond not implemented
}

class StmIRTree extends LazyIRTree {
    private final tree.Stm stm;
    StmIRTree (tree.Stm s) { stm = s; }
    tree.Stm asStm() { return stm; }
    // asExp, asCond not implemented
}
abstract class Cx extends LazyIRTree {
    tree.Exp unEx() { /* Program 7.2, page 161.*/ } 
    abstract tree.Stm unCx (Label t, Label f); 
    // unNx "left as exercise"

}
Lazy IR Trees

“Making ‘simple’ Cx expressions from Absyn comparison operators is easy with the CJUMP operator.” Appel, 2nd, Section 7.2, page 149.

class RelCx extends Cx { /* Page 149. */
    final private int relop;
    final private tree.Exp left, right;

    RelCx (int op, tree.Exp l, tree.Exp r) {
        relop = op; left = l; right = r;
    }

    public tree.Stm asCond (Label t, Label f) {
        // new tree.CJUMP
    }
}
If $i=\text{translate}("x<5")$, then $i\text{.asCond}(t,f)$ should be $\text{CJUMP}(\text{LT},x,5,t,f)$. 
class Rel_LIRT extends Cond_LIRT {
    final private int relop;
    final private Exp left, right;
    Rel_LIRT (final int op, final Exp l, final Exp r) {
        relop = op; left = l; right = r;
    }
    @Override
    public Stm asCond (NameOfLabel t, NameOfLabel f) {
        return new CJUMP (relop, left, right, t, f);
    }
    // This conditional test is used only for its side effects!
    public Stm asStm() {
        return new SEQ(new EVAL(left),new EVAL(right));
    }
}
This sketch of the class appears in Appel, 2nd, Section 7.2, page 150.

class IfThenElseExp extends LazyIRTree {
    private final LazyIRTree cond, e2, e3;
    Label t,f,join;
    IfThenElseExp (LazyIRTree c, LazyIRTree thenClause, LazyIRTree elseClause) {
        assert c!=null; assert thenClause!=null;
        cond = c; e2 = thenClause; e3 = elseClause;
    }
    public Exp asExp() { /* ... */ }
    public Stm asStm() { /* ... */ }
    public Stm asCond (Label tt, Label ff) { /* ... */ }
}
A Simpler Translate

“To simplify the implementation of Translate, you may do without the Ex, Nx, Cx constructors. The entire Translate module can be done with ordinary value-expressions.” Appel, page 178.
Fragments

syntax. Program

symbol table

Chapter 5

fragments

Chapter 7

Chapter 4
Fragments

syntax. Program

Chapter 4

fragments

tree. Stm

Chapter 7

tree. Stm
Exercises

7.1 Draw a picture of the IR tree that results from each of the following expressions. Assume all variables escape.

b.

let

    var i := 8
    type intArray = array of int
    var b := intArray[10] of 0

in

    b[i+1]

end
Chapter 8: Basic Blocks and Traces
Translation of a Method

Stm \xrightarrow{\text{Seq, PSeq}} \text{Stm list} \xrightarrow{\text{linearize}} \text{Inst list} \xrightarrow{\text{emit}}

\text{chap 7} \xrightarrow{} \text{chap 8} \xrightarrow{} \text{chap 9}
The constructs of the intermediate representation trees of package 

**tree** are crafted to match the capabilities of real machines. This 
facilitates the translation from the abstract syntax. Yet some 
aspects do not correspond exactly to a real machine.

- **CJUMP** can jump to one of two labels, but real machine 
  instructions fall through to the next instruction if the condition 
  is false

- **ESEQ/RET** within expressions require attention to the order of 
  evaluating the subtrees; the left subtree must always be 
  evaluated first

\[
2*(x:=3)+x
\]

\[
\text{BINOP(+, BINOP(*,2,ESEQ/RET(x:=3,3)), x)}
\]

- **CALL** within expressions

\[
\text{BINOP(+, callf2(1,2), callf3(1,2,3))}
\]

- **CALL** within **CALL** complicates passing parameters in a fixed set 
  of registers

\[
\text{call f1(callf2 (1,2), callf3(1,2,3))}
\]
We can take any tree and rewrite it to an equivalent tree without the troublesome cases. The SEQ constructs will all move to the top of the tree and become unnecessary. The tree can be replaced by a list of the other constructs.

1. The tree is flattened by removing SEQ and ESEQ
2. The list is broken into basic blocks containing no internal jumps
3. The basic blocks are ordered into a set of traces in which every CJUMP is followed by its false label.

The package canon does all of this.

List<tree.Stmt> canon.Main.transform (tree.Stmt body)
Chapter 8: Basic Blocks and Traces

Section 8.1: Canonical Trees
A *canonical tree* is one in which

1. there are no **SEQ** or **ESEQ[=RET]**
2. the parent of each **CALL** is either **EVAL** or **MOVE(TEMP t, ...)**.

How can the **ESEQ/RET** nodes be eliminated? The idea is to lift them higher and higher in the tree, until they can become **SEQ** nodes.

Appel, 2nd, page 164.
See Appel, 2nd, Figure 8.1, page 165.
\[
\begin{equation}
\text{ESEQ}(s_1, \text{ESEQ}(s_2, e)) \quad \Rightarrow \quad \text{ESEQ}(\text{SEQ}(s_1, s_2), e)
\end{equation}
\]
\[(2)\]

\[
\begin{align*}
\text{BINOP}(op, \text{ESEQ}(s, e_1), e_2) &= \text{ESEQ}(s, \text{BINOP}(op, e_1, e_2)) \\
\text{MEM}(\text{ESEQ}(s, e_1)) &= \text{ESEQ}(s, \text{MEM}(e_1)) \\
\text{JUMP}(\text{ESEQ}(s, e_1)) &= \text{SEQ}(s, \text{JUMP}(e_1)) \\
\text{CJUMP}(op, \text{ESEQ}(s, e_1), e_2, l_1, l_2) &= \text{SEQ}(s, \text{CJUMP}(op, e_1, e_2, l_1, l_2))
\end{align*}
\]
if \( s, e_1 \) commute

\[
\begin{align*}
\text{BINOP}(op, e_1, \text{ESEQ}(s, e_2)) &= \text{ESEQ}(s, \text{BINOP}(op, e_1, e_2)) \\
\text{CJUMP}(op, e_1, \text{ESEQ}(s, e_2), l_1, l_2) &= \text{SEQ}(s, \text{CJUMP}(op, e_1, e_2, l_1, l_2))
\end{align*}
\]
(3) \[
\begin{align*}
\text{BINOP}(op, e_1, \text{ESEQ}(s, e_2)) &= \text{ESEQ}(\text{MOVE}(\text{TEMP } t, e_1), \\
&\phantom{=} \text{ESEQ}(s, \text{BINOP}(op, \text{TEMP } t, e_2))) \\
\text{CJUMP}(op, e_1, \text{ESEQ}(s, e_2), l_1, l_2) &= \text{SEQ}(\text{MOVE}(\text{TEMP } t, e_1), \\
&\phantom{=} \text{SEQ}(s, \text{CJUMP}(op, \text{TEMP } t, e_2, l_1, l_2)))
\end{align*}
\]
The statement $s$ is said to *commute* with an expression $e$, if the value of $e$ is not changed where $s$ executed immediately before $e$.

\[
\text{BINOP}(\text{op}, e_1, \text{ESEQ/RET}(s, e_2)) = \\
\text{ESEQ/RET}(s, \text{BINOP}(\text{op}, e_1, e_2)) = 
\]
static boolean commute (Stm a, Exp b) {
    return isNop(a) ||
            b instanceof NAME ||
            b instanceof COSNT;
}

static boolean isNop (Stm a) {
    return a instanceof EVAL &&
            ((EVAL)a).exp instanceof CONST;
}
Move calls to the top level. Create temps to hold values of calls.
linearize creates java.util.List¡Tree.Stmt¿
Chapter 8: Basic Blocks and Traces

Section 8.2: Taming Conditional Branches
A basic block is a sequence of statements that is always entered at the beginning and exited at the end:

- The first statement is a LABEL
- The last statement is a JUMP or CJUMP
- There are no other LABELs, JUMPs or CJUMPs.

Basic blocks can be arranged in any order, and the result of executing the program will be the same.
Schedule a trace.

Put blocks in list Q
while Q is not empty
    Start a new (empty) trace, call it T
    Remove the first block b from Q
    while b is not marked
        mark b
        append b to the end of T
    let s be the set of blocks to which b branches (if any)
    if there is any successor c in s
        b := c
    end the current trace T
MOVE(
    MEM (ESEQ (LABEL s, CONST 1828)),
    CONST 2718)

After linearize
1: LABEL s
2: MOVE (MEM (CONST 1828), CONST 2718)

After trace scheduling
1: LABEL s
2: MOVE (MEM (CONST 1828), CONST 2718)
3: JUMP (NAME BB$1)
MOVE(
  MEM (CONST 1828),
  ESEQ (LABEL s, CONST 2718))

After linearize
1: LABEL s
2: MOVE (MEM (CONST 1828), CONST 2718)

After trace scheduling
1: LABEL s
2: MOVE (MEM (CONST 1828), CONST 2718)
3: JUMP (NAME BB$1)
EVAL(
    ESEQ(
        LABEL s,
        CONST 2718))

After linearize
1: LABEL s

After trace scheduling
1: LABEL s
2: JUMP (NAME BB$1)
EVAL(
    CALL(
        ESEQ (LABEL s, CONST 2718),
        CONST 1828,
        MEM( NAME s$c$1)
    )
)

After linearize
1: LABEL s
2: EVAL ( CALL (CONST 2718, CONST 1828, MEM (NAME s$c$1)))
MOVE(
    TEMP t,
    CALL(
        ESEQ(
            LABEL s,
            CONST 2718),
            CONST 1828,
            MEM( NAME s$c$1))
)

After linearize
1: LABEL s
2: MOVE (TEMP t, CALL (CONST 2718, CONST 1828, MEM (NAME s$c$1)))
MOVE(
   MEM(
      ESEQ(
         SEQ(
            CJUMP(LT,
            TEMP t,
            CONST 0,
            out,ok),
            LABEL ok),
            TEMP t)),
      CONST 2718)

After linearize
1: CJUMP (LT,  TEMP t,  CONST 0,  out,ok)
2: LABEL  ok
3: MOVE(  MEM(  TEMP t),  CONST 2718)
m := 0; v := 0;
if v ≥ n goto 6
r := v; s := 0;
if r < n goto 4
v := v + 1; goto 2
x := M[r]; s := s + 1; if s ≤ m goto 8
m := 3;
r := r + 1; goto 4
return m
Optimal traces

prolog
while 1>N do
  body_stmts
od
epilog

begin:
  jump test
test:
    cjump 1>N done, body_label
body_label:
  body_stmts
  jump test
done:
  epilog
  jump elsewhere
Chapter 9: Instruction Selection
Instruction Selection

Problem: How to assign the machine register to all the temporaries?

Solution: Allocate registers after instruction selection—generate instructions templates.

1. Generate abstract instructions—instructions with holes for registers.
2. Allocate registers; spilling may require inserting some instructions.
3. Generate procedure entry and exit sequences.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Assembler Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>$r_i \leftarrow r_j + r_k$</td>
<td>add $%r_j,%r_k,%r_i$</td>
</tr>
<tr>
<td>MUL</td>
<td>$r_i \leftarrow r_j \times r_k$</td>
<td>smul $%r_j,%r_k,%r_i$</td>
</tr>
<tr>
<td>SUB</td>
<td>$r_i \leftarrow r_j - r_k$</td>
<td>sub $%r_j,%r_k,%r_i$</td>
</tr>
<tr>
<td>ADDI</td>
<td>$r_i \leftarrow r_j + c$</td>
<td>add $%r_j,c,%r_i$</td>
</tr>
<tr>
<td>SUBI</td>
<td>$r_i \leftarrow r_j - c$</td>
<td>sub $%r_j,c,%r_i$</td>
</tr>
<tr>
<td>LOAD</td>
<td>$r_i \leftarrow M[r_j + c]$</td>
<td>ld $[%r_j,%c],%r_i$</td>
</tr>
<tr>
<td>STORE</td>
<td>$r_i \rightarrow M[r_j + c]$</td>
<td>st $%r_i,[%r_j+%c]$</td>
</tr>
<tr>
<td>MOVEM</td>
<td>$M[r_i] \rightarrow M[r_j]$</td>
<td>ld $[%r_j],%r$; st $%r,[%j_i]$</td>
</tr>
</tbody>
</table>
ld [%r1+%r2],%r3  !  r3 := MEM[r1+r2]
ld [%r1+c],%r3  !  r3 := MEM[r1+c]
ld [%r1-c],%r3  !  r3 := MEM[r1-c]
ld [%r1],%r3  !  r3 := MEM[r1]
st %r1, [%r2+%r3]  !  MEM[r2+r3] := r1
st %r1, [%r2+c]  !  MEM[r2+c] := r1
st %r1, [%r2-c]  !  MEM[r2-c] := r1
st %r1, [%r2]  !  MEM[r2] := r1
add %r1, %r2, %r3  !  r3 := r1 + r2
sub %r1, %r2, %r3  !  r3 := r1 - r2
smul %r1, %r2, %r3  !  (%y)r3 := r1 * r2 signed, integer multiplication
sdiv %r1, %r2, %r3  !  r3 := (%y)r1 / r2 signed, integer division
package assem;
public abstract class Instruction {
    // A concrete subclass should override these
    public List<Temp> use() { return null; }
    public List<Temp> def() { return null; }
    public List<Label> jumps() { return null; }
    public final String assem;
    protected Instruction (String a) { assem=a; }
    public String toString () { return assem; }
    public String format () {
        return format (DEFAULT_MAP);
    }
    public String format (Map<Temp,String> map) {
        /* body provided */
    }
}

Abstract Instruction Format

The string of an instr may refer to source registers \(s_0, s_1, \ldots, s_{(k-1)}\), and destination registers \(d_0, d_1, \text{ etc.}\). Jumps are OPER instructions that refer to labels \(j_0, j_1, \text{ etc.}\) conditional jumps, which may branch away or fall through, typically have two labels in the jump list but refer to only one of them in the assem string.

(Later in dataflow analysis, these will be used variables and defined variables, respectively. And the jump list will be the successors of the statement.)
Abstract Instruction Format

Calling `i.format(m)` formats an assembly instruction as a string; `m` is an object implementing the `Map<Temp, String>` interface. The “parameters,” e.g., ’s0, ’d1, ’j2, are replaced with real registers and labels from the map.
Type of Instructions

The concrete subclasses of Instruction:

abstract Instruction (String assem)

Comment (String text)

LabelInstruction (NameOfLabel l)

OperationInstruction (String assem, String comment,
   NameOfTemp d, NameOfTemp s1, NameOfTemp s2)

MoveInstruction (String assem, String comment,
   NameOfTemp d, NameOfTemp s)
Abstract Instruction Format

OperationInstruction ("LOAD 'd0 <- M['s0+8]", new Temp(), frame.FP())

OperationInstruction ("ld ['s0+8],’d0", new Temp(), frame.FP())

ld [%fp+8],%l0
Pattern match trees and possibly generate more temps.

\[ + \left( (\text{TEMP} \ t87, \ \text{CONST} \ 4), \ \text{MEM} \ (\text{TEMP} \ t92) \right) \]

\[
\begin{align*}
\text{MULI} & \quad \text{'}d0 \leftarrow \text{'}s0 \times 4 & t908 & t87 \\
\text{LOAD} & \quad \text{'}d0 \leftarrow \text{M}['s0+0] & t909 & t92 \\
\text{ADD} & \quad \text{'}d0 \leftarrow \text{'}s0 \times \text{'}s1 & t910 & t908, t909 \\
\text{sll} & \quad \text{'}s0, \ 2, \text{'}d0 & t908 & t87 \\
\text{ld} & \quad ['s0+0], \text{'}d0 & t909 & t92 \\
\text{add} & \quad \text{'}s0, \text{'}s1, \text{'}d0 & t910 & t908, t909
\end{align*}
\]
TempList L(Temp h, TempList t) {return new TempList(h, t);}

munchStmt(SEQ(a,b))
    {munchStmt(a); munchStmt(b);}  
munchStmt(MOVE(MEM(BINOP(PLUS,e1,CONST(i)),e2)))
    emit(new OPER("STORE M[`s0" + i + "'] <- `s1\n",
                   null, L(munchExp(e1), L(munchExp(e2), null))));
    munchStmt(MOVE(MEM(BINOP(PLUS,CONST(i),e1)),e2))
    emit(new OPER("STORE M[`s0" + i + "'] <- `s1\n",
                   null, L(munchExp(e1), L(munchExp(e2), null))));
    munchStmt(MOVE(MEM(e1),MEM(e2)))
    emit(new OPER("MOVE M[`s0] <- M[`s1]\n",
                   null, L(munchExp(e1), L(munchExp(e2), null))));
    munchStmt(MOVE(MEM(CONST(i)),e2))
    emit(new OPER("STORE M[r0" + i + "'] <- `s0\n",
                   null, L(munchExp(e2), null)));
    munchStmt(MOVE(MEM(e1),e2))
    emit(new OPER("STORE M[`s0] <- `s1\n",
                   null, L(munchExp(e1), L(munchExp(e2), null))));
    munchStmt(MOVE(TEMP(i), e2))
    emit(new OPER("ADD `d0 <- `s0 + r0\n",
                   L(i,null), L(munchExp(e2), null)));
    munchStmt(LABEL(lab))
    emit(new Assem.LABEL(lab.toString() + ":
", lab));
munchExp(MEM(BINOP(PLUS, e1, CONST(i))))
    Temp r = new Temp();
    emit(new OPER("LOAD 'd0 <- M['s0+' + i + ']
          L(r,null), L(munchExp(e1),null)");
    return r;

munchExp(MEM(BINOP(PLUS, CONST(i), e1)))
    Temp r = new Temp();
    emit(new OPER("LOAD 'd0 <- M['s0+' + i + ']
          L(r,null), L(munchExp(e1),null)");
    return r;

munchExp(MEM(CONST(i)))
    Temp r = new Temp();
    emit(new OPER("LOAD 'd0 <- M[r0+]
          L(r,null), L(munchExp(e1),null)");
    return r;

munchExp(MEM(e1))
    Temp r = new Temp();
    emit(new OPER("LOAD 'd0 <- M['s0+0]
          L(r,null), L(munchExp(e1),null)");
    return r;

munchExp(BINOP(PLUS, e1, CONST(i)))
    Temp r = new Temp();
    emit(new OPER("ADDI 'd0 <- 's0+' + i + '
          L(r,null), L(munchExp(e1),null)");
    return r;

munchExp(BINOP(PLUS, CONST(i), e1))
    Temp r = new Temp();
    emit(new OPER("ADDI 'd0 <- 's0+' + i + '
          L(r,null), L(munchExp(e1),null)");
    return r;

munchExp(CONST(i))
    Temp r = new Temp();
    emit(new OPER("ADDI 'd0 <- r0+
          L(null), L(munchExp(e1),null)");
    return r;

munchExp(BINOP(PLUS, e1, e2))
    Temp r = new Temp();
    emit(new OPER("ADD 'd0 <- 's0+ 's1'
          L(r,null), L(munchExp(e1),L(munchExp(e2),null)");
    return r;

munchExp(TEMP(t))
    return t;

**PROGRAM 9.6.** Assem-instructions for munchExp.
Chapter 10: Liveness Analysis
Two temporaries can be assigned to the same register, if they are never “in use” at the same time. It is the purpose of this chapter to explain how this is determined.

A variable is *live* if it holds a value that may be needed in the future.

*Liveness analysis* is the determination by the compiler of the variables that may potentially be read after some point in the program before their next write update.

In addition to register allocation, liveness analysis can be used to remove statements that assign to a variable whose value is not used afterward, and find uninitialized variables.
Liveness analysis is just one kind of *dataflow analysis*.

In forward-flow analyses, the entry state of a node is a function of the exit state of its predecessors.

\[
in[n] = \text{meet}_{p \in \text{pred}[n]}(out[p]) \\
out[n] = \text{transfer}(in[n])
\]

In backward-flow analyses, the exit state of a node is a function of the entry state of its successors.

\[
out[n] = \text{meet}_{s \in \text{succ}[n]}(in[s]) \\
in[n] = \text{transfer}(out[n])
\]

Each particular type of dataflow analysis has its own specific transfer function and join operation.
In “any path” analysis, the state of a node is determined by the union (meet operation is set union).

In “all paths” analysis, the state of a node is determined by the intersection (meet operation is set intersection).
Liveness analysis for the purpose of (i.a) register allocation.

Reaching definitions for the purpose of (i.a) constant propagation.

Available-expressions analysis for the purpose of (i.a.) common subexpression elimination.

Busy expressions (or anticipated expressions) for the purpose of (i.a.) code hoisting.
To perform analyses of this kind on a program, it is usually necessary to make a *control-flow graph*. Each statement in the program is a node in the flow graph; an edge indicates that one statement potentially follows another in the execution of the program.
As a reminder, here are the forms/representations of our statements in the back-end of the MiniJava compiler.

\[
\begin{align*}
\text{a} & \;:=\; \text{n}; \\
\text{L1: } \text{b} & \;:=\; \text{a}+\text{m};
\end{align*}
\]

MOVE (TEMP(a), CONST(n))
LABEL ("L1")
MOVE (TEMP(b), BINOP (PLUS, TEMP(b), CONST(m)))

"set n,'d0" src/use=[], dst/def=TEMP(a), jump=[]
LABEL ("L1")
"add 's0,m,'d0" src/use=TEMP(a), dst/def=TEMP(b), jump=[]
A flow-graph node has out-edges that lead to successor nodes, and in-edges that come from predecessor nodes. The set $\text{pred}[n]$ is all the predecessors of node $n$, and $\text{succ}[n]$ is all the successors.

An assignment to a variable or temporary defines that variable. An occurrence of a variable on the right-hand side of an assignment uses the variable. We can speak of the def of a node as the set of variables that it defines; and, similarly, the use of a node is the set of variables that it uses.

$\text{use}[n]$ is the set of variables used by node $n$, and $\text{def}[n]$ is the set of variables defined (set) by note $n$.

(The set $\text{use}[n]$ are what are labeled $\text{src}$ in the assem package, and the set $\text{def}[n]$ are what are labeled $\text{dst}$.)
Equations 10.3. Dataflow equations for liveness analysis.

1. A variable is live on entry to node (statement) \( n \), if it is in \( \text{use}[n] \).

2. A variable is live on entry to node (statement) \( n \), if it is not in \( \text{def}[n] \), but it is live on exit to node \( n \), i.e., the value is needed later.

3. A variable is live on exit from node (statement) \( n \), if it is live on entry to some successor node (statement) \( s \).
It is easy to solve these equations by iteration.
for each \((n)\)
\((\mathit{in}[n] := \emptyset)\)
\((\mathit{out}[n] := \emptyset)\)
end;

repeat
  for each \((n)\)
  \((\mathit{out}[n] := \bigcup_{s \in \mathit{succ}(n)} \mathit{in}[s])\)
  \((\mathit{in}[n] := \mathit{use}[n] \cup (\mathit{in}[n])\)
end;
until no changes;

Computing out before in propagates the changes faster. Order the nodes backward along the control flow also propagates the changes faster: “Go with the flow.”
Observations

Monotonic: variables are added, never subtracted, by the algorithm. Bigger out and only make bigger in, and vice versa.

Basic blocks: nodes with only one predecessor and one successor are not interesting and can be merged with the predecessors and successors.

Representation of sets? Bit-vector.

Complexity: $O(n^4)$ (but usually much faster)

Least fixed points. Actually more than one solution. Algorithm always computes the least fixed point.

Conservative. Impossible to decide if a “value will actually be used in the future.”
Example of Liveness Analysis

<table>
<thead>
<tr>
<th>flowgraph</th>
<th>stm</th>
<th>use</th>
<th>def</th>
<th>succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a := 0)</td>
<td>1</td>
<td>(\emptyset)</td>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>(b := a + 1)</td>
<td>2</td>
<td>a</td>
<td>b</td>
<td>3</td>
</tr>
<tr>
<td>(c := c + b)</td>
<td>3</td>
<td>b,c</td>
<td>c</td>
<td>4</td>
</tr>
<tr>
<td>(a := b \times 2)</td>
<td>4</td>
<td>b</td>
<td>a</td>
<td>5</td>
</tr>
<tr>
<td>(a &lt; N)</td>
<td>5</td>
<td>a</td>
<td>(\emptyset)</td>
<td>2,6</td>
</tr>
<tr>
<td>return c</td>
<td>6</td>
<td>c</td>
<td>(\emptyset)</td>
<td>none</td>
</tr>
</tbody>
</table>
Initialize all *in* and *out* to the empty set.

<table>
<thead>
<tr>
<th>flowgraph</th>
<th>use</th>
<th>def</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:=0</td>
<td>∅</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>b:=a+1</td>
<td>a</td>
<td>b</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>c:=c+b</td>
<td>b,c</td>
<td>c</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>a:=b*2</td>
<td>b</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>a&lt;N</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>return c</td>
<td>c</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>
\[ \text{flowgraph} \quad \text{use} \quad \text{def} \quad \text{in} \quad \text{out} \]

\[
\begin{array}{|l|l|l|l|l|}
\hline
a := 0 & \emptyset & a & ? & \emptyset \\
\hline
b := a + 1 & a & b & \emptyset & \emptyset \\
\hline
c := c + b & b, c & c & \emptyset & \emptyset \\
\hline
a := b * 2 & b & a & \emptyset & \emptyset \\
\hline
a < N & a & \emptyset & \emptyset & \emptyset \\
\hline
\text{return} \ c & c & \emptyset & \emptyset & \emptyset \\
\hline
\end{array}
\]

\[ \text{in}[1] = \text{use}[1] \cup (\text{out}[1] \setminus \text{def}[1]) \]
\[
\text{flowgraph} \quad \text{use} \quad \text{def} \quad \text{in} \quad \text{out} \\
\begin{array}{cccccc}
\text{a:=0} & \emptyset & a & \emptyset & \emptyset & \text{in}_1 = \text{use}_1 \cup (\text{out}_1 \setminus \text{def}_1) \\
\text{b:=a+1} & a & b & \emptyset & \emptyset & = \emptyset \\
\text{c:=c+b} & b,c & c & \emptyset & \emptyset & \\
\text{a:=b*2} & b & a & \emptyset & \emptyset & \\
\text{a<N} & a & \emptyset & \emptyset & \emptyset & \\
\text{return c} & c & \emptyset & \emptyset & \emptyset & \text{return c} \\
\end{array}
\]
Liveness Analysis—compute $out[1]$ 

<table>
<thead>
<tr>
<th>flowgraph</th>
<th>use</th>
<th>def</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a := 0$</td>
<td>$\emptyset$</td>
<td>$a$</td>
<td>$\emptyset$</td>
<td>?</td>
</tr>
<tr>
<td>$b := a + 1$</td>
<td>$a$</td>
<td>$b$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$c := c + b$</td>
<td>$b, c$</td>
<td>$c$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$a := b * 2$</td>
<td>$b$</td>
<td>$a$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$a &lt; N$</td>
<td>$a$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>return $c$</td>
<td>$c$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

$out[1] = \bigcup_{s \in succ[1]} in[s]$
### Liveness Analysis—compute $out[1]$}

The flowgraph for the given program is shown below, with the analysis results for each node:

<table>
<thead>
<tr>
<th>Node</th>
<th>use</th>
<th>def</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a := 0$</td>
<td>$\emptyset$</td>
<td>$a$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$b := a + 1$</td>
<td>$a$</td>
<td>$b$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$c := c + b$</td>
<td>$b, c$</td>
<td>$c$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$a := b \times 2$</td>
<td>$b$</td>
<td>$a$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$a &lt; N$</td>
<td>$a$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>return $c$</td>
<td>$c$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

The outflow equation is given by:

$$ out[1] = \bigcup_{s \in \text{succ}[1]} in[s] $$

- $in[2] = \emptyset$
- $out[1] = \bigcup_{s \in \text{succ}[1]} in[s] = \emptyset$
Liveness Analysis
In the flowgraph, we have:

- **a:=0**
  - Use: \( \emptyset \)
  - Def: \( a \)
  - In: \( \emptyset \)
  - Out: \( \emptyset \)

- **b:=a+1**
  - Use: \( a \)
  - Def: \( b \)
  - In: \( b \)
  - Out: \( a \)

- **c:=c+b**
  - Use: \( b,c \)
  - Def: \( c \)
  - In: \( c \)
  - Out: \( \emptyset \)

- **a:=b*2**
  - Use: \( b,a \)
  - Def: \( a \)
  - In: \( a \)
  - Out: \( \emptyset \)

- **a<N**
  - Use: \( a \)
  - Def: \( \emptyset \)
  - In: \( \emptyset \)
  - Out: \( \emptyset \)

- **return c**
  - Use: \( c \)
  - Def: \( \emptyset \)
  - In: \( \emptyset \)
  - Out: \( \emptyset \)

The calculation for \( \text{in}[2] = \text{use}[2] \cup (\text{out}[2] \setminus \text{def}[2]) \) is:

- For **b:=a+1**:
  - \( a \) is used, \( b \) is defined.
  - \( \text{in}[2] = \{a\} \cup (\emptyset \setminus \{b\}) = \{a\} \)

This completes the liveness analysis for the given flowgraph.
\[
\begin{array}{c|c|c|c|c}
\text{flowgraph} & \text{use} & \text{def} & \text{in} & \text{out} \\
\hline
a := 0 & \emptyset & a & \emptyset & \emptyset \\
\downarrow & & & & \\
b := a + 1 & a & b & a & ? \\
\downarrow & & & & \\
c := c + b & b, c & c & \emptyset & \emptyset \\
\downarrow & & & & \\
a := b \ast 2 & b & a & \emptyset & \emptyset \\
\downarrow & & & & \\
a < N & a & \emptyset & \emptyset & \emptyset \\
\downarrow & & & & \\
\text{return } c & c & \emptyset & \emptyset & \emptyset \\
\end{array}
\]

\[\text{out}[2] = \bigcup_{s \in \text{succ}[2]} \text{in}[s]\]
<table>
<thead>
<tr>
<th>flowgraph</th>
<th>use</th>
<th>def</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:=0</td>
<td>∅</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>b:=a+1</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>∅</td>
</tr>
<tr>
<td>c:=c+b</td>
<td>b,c</td>
<td>c</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>a:=b*2</td>
<td>b</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>a&lt;N</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>return c</td>
<td>c</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>

\[
out[2] = \bigcup_{s \in \text{succ}[2]} \text{in}[s]
\]

\[
in[3] = \emptyset
\]

\[
= \emptyset
\]
<table>
<thead>
<tr>
<th>flowgraph</th>
<th>use</th>
<th>def</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:=0</td>
<td>∅</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>b:=a+1</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>∅</td>
</tr>
<tr>
<td>c:=c+b</td>
<td>b,c</td>
<td>c</td>
<td>?</td>
<td>∅</td>
</tr>
<tr>
<td>a:=2*b</td>
<td>b</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>a&lt;N</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>

\[ \text{in}[3] = \text{use}[3] \cup (\text{out}[3] \setminus \text{def}[3]) \]
flowgraph  use  def  in  out

<table>
<thead>
<tr>
<th>a:=0</th>
<th>∅</th>
<th>a</th>
<th>∅</th>
<th>∅</th>
</tr>
</thead>
<tbody>
<tr>
<td>b:=a+1</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>∅</td>
</tr>
<tr>
<td>c:=c+b</td>
<td>b,c</td>
<td>c</td>
<td>b,c</td>
<td>∅</td>
</tr>
<tr>
<td>a:=b*2</td>
<td>b</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>a&lt;N</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>return c</td>
<td>c</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>

\[\text{in}[3] = \text{use}[3] \cup (\text{out}[3] \setminus \text{def}[3]) = \{b, c\} \cup (\emptyset \setminus \{c\}) = \{b, c\}\]
<table>
<thead>
<tr>
<th>flowgraph</th>
<th>use</th>
<th>def</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:=0</td>
<td>∅</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>b:=a+1</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>∅</td>
</tr>
<tr>
<td>c:=c+b</td>
<td>b,c</td>
<td>c</td>
<td>b,c</td>
<td>?</td>
</tr>
<tr>
<td>a:=b*2</td>
<td>b</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>a&lt;N</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>return c</td>
<td>c</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>

\[
\text{out}[3] = \bigcup_{s \in \text{succ}[3]} \text{in}[s]
\]
flowgraph  use  def  in  out

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{a:=0} & \emptyset & a & \emptyset & \emptyset \\
\hline
\text{b:=a+1} & a & b & a & \emptyset \\
\hline
\text{c:=c+b} & b,c & c & b,c & \emptyset \\
\hline
\text{a:=b*2} & b & a & \emptyset & \emptyset \\
\hline
\text{a<N} & a & \emptyset & \emptyset & \emptyset \\
\hline
\text{return c} & c & \emptyset & \emptyset & \emptyset \\
\hline
\end{array}
\]

\[
\text{out}[3] = \bigcup_{s \in \text{succ}[3]} \text{in}[s]
\]

\[
= \text{in}[4]
\]

\[
= \emptyset
\]
Liveness Analysis
<table>
<thead>
<tr>
<th>flowgraph</th>
<th>use</th>
<th>def</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>a := 0</td>
<td>∅</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>b := a + 1</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>∅</td>
</tr>
<tr>
<td>c := c + b</td>
<td>b,c</td>
<td>c</td>
<td>b,c</td>
<td>∅</td>
</tr>
<tr>
<td>a := b * 2</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>∅</td>
</tr>
<tr>
<td>a &lt; N</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>

\[
in[4] = \text{use}[4] \cup (\text{out}[4] \setminus \text{def}[4]) = \{b\} \cup (\emptyset \setminus \{a\}) = \{b\}\]

Liveness Analysis
<table>
<thead>
<tr>
<th>flowgraph</th>
<th>use</th>
<th>def</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:=0</td>
<td>∅</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>b:=a+1</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>∅</td>
</tr>
<tr>
<td>c:=c+b</td>
<td>b,c</td>
<td>c</td>
<td>b,c</td>
<td>∅</td>
</tr>
<tr>
<td>a:=b*2</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>?</td>
</tr>
<tr>
<td>a&lt;N</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>return c</td>
<td>c</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>

\[ out[4] = \bigcup_{s \in \text{succ}[4]} in[s] \]
A flowgraph with variables and assignments:

1. `a := 0`  
   - Use: ∅  
   - Def: a  
   - In: ∅  
   - Out: ∅  

2. `b := a + 1`  
   - Use: a  
   - Def: a  
   - In: ∅  
   - Out: ∅  

3. `c := c + b`  
   - Use: b, c  
   - Def: b, c  
   - In: c  
   - Out: ∅  

4. `a := b * 2`  
   - Use: b  
   - Def: b  
   - In: a  
   - Out: ∅  

5. `a < N`  
   - Use: a  
   - Def: ∅  
   - In: ∅  
   - Out: ∅  

6. `return c`  
   - Use: c  
   - Def: ∅  
   - In: ∅  
   - Out: ∅  

Liveness analysis:

\[ \text{out}[4] = \bigcup_{s \in \text{succ}[4]} \text{in}[s] = \text{in}[5] = \emptyset \]
<table>
<thead>
<tr>
<th>flowgraph</th>
<th>use</th>
<th>def</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:=0</td>
<td>∅</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>b:=a+1</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>∅</td>
</tr>
<tr>
<td>c:=c+b</td>
<td>b,c</td>
<td>c</td>
<td>b,c</td>
<td>∅</td>
</tr>
<tr>
<td>a:=b*2</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>∅</td>
</tr>
<tr>
<td>a&lt;N</td>
<td>a</td>
<td>∅</td>
<td>?</td>
<td>∅</td>
</tr>
</tbody>
</table>

\[ \text{in}[5] = \text{use}[5] \cup (\text{out}[5] \setminus \text{def}[5]) \]

Liveness Analysis
<table>
<thead>
<tr>
<th>flowgraph</th>
<th>use</th>
<th>def</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:=0</td>
<td>∅</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>b:=a+1</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>∅</td>
</tr>
<tr>
<td>c:=c+b</td>
<td>b,c</td>
<td>c</td>
<td>b,c</td>
<td>∅</td>
</tr>
<tr>
<td>a:=b*2</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>∅</td>
</tr>
<tr>
<td>a&lt;N</td>
<td>a</td>
<td>∅</td>
<td>a</td>
<td>∅</td>
</tr>
</tbody>
</table>

\[
\text{in}[5] = \text{use}[5] \cup (\text{out}[5] \setminus \text{def}[5])
\]
\[
= \{a\} \cup (\emptyset \setminus \emptyset)
\]
\[
= \{a\}
\]

Liveness Analysis
<table>
<thead>
<tr>
<th>flowgraph</th>
<th>use</th>
<th>def</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a:=0</code></td>
<td>∅</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td><code>b:=a+1</code></td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>∅</td>
</tr>
<tr>
<td><code>c:=c+b</code></td>
<td>b,c</td>
<td>c</td>
<td>b,c</td>
<td>∅</td>
</tr>
<tr>
<td><code>a:=b*2</code></td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>∅</td>
</tr>
<tr>
<td><code>a&lt;N</code></td>
<td>a</td>
<td>∅</td>
<td>a</td>
<td>?</td>
</tr>
<tr>
<td><code>return c</code></td>
<td>c</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>

\[ \text{out}[5] = \bigcup_{s \in \text{succ}[5]} \text{in}[s] \]
<table>
<thead>
<tr>
<th>flowgraph</th>
<th>use</th>
<th>def</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:=0</td>
<td>∅</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>b:=a+1</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>∅</td>
</tr>
<tr>
<td>c:=c+b</td>
<td>b,c</td>
<td>c</td>
<td>b,c</td>
<td>∅</td>
</tr>
<tr>
<td>a:=b*2</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>∅</td>
</tr>
<tr>
<td>a&lt;N</td>
<td>a</td>
<td>∅</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>return c</td>
<td>c</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>

\[
\text{out}[5] = \bigcup_{s \in \text{succ}[5]} \text{in}[s] = \text{in}[2] \cup \text{in}[6] = \{a\} \cup \emptyset = \{a\}
\]
flowgraph | use | def | in | out
--- | --- | --- | --- | ---
\[a:=0\] | $\emptyset$ | $a$ | $\emptyset$ | $\emptyset$
\[b:=a+1\] | $a$ | $b$ | $a$ | $\emptyset$
\[c:=c+b\] | $b,c$ | $c$ | $b,c$ | $\emptyset$
\[a:=b*2\] | $b$ | $a$ | $b$ | $\emptyset$
\[a<N\] | $a$ | $\emptyset$ | $a$ | $a$
\[return\ c\] | $c$ | $\emptyset$ | $?$ | $\emptyset$

flowgraph  use  def  in  out

a:=0  \emptyset  a \emptyset \emptyset

b:=a+1  a b a \emptyset

c:=c+b  b,c c b,c \emptyset

a:=b*2  b a b \emptyset

a<N  a \emptyset a a

return c  c \emptyset c \emptyset

\textit{in}[6] = \textit{use}[6] \cup (\textit{out}[6] \setminus \textit{def}[6])
= \{c\} \cup (\emptyset \setminus \emptyset)
= \{c\}
flowgraph  use  def  in  out

a:=0  ∅  a  ∅  ∅

b:=a+1  a  b  a  ∅

c:=c+b  b,c  c  b,c  ∅

a:=b*2  b  a  b  ∅

a<N  a  ∅  a  a

return c  c  ∅  c  ?  

out[6] = \bigcup_{s \in succ[6]} in[s]

Liveness Analysis—compute out[6]
### Flowgraph

<table>
<thead>
<tr>
<th>Flowgraph</th>
<th>Use</th>
<th>Def</th>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a := 0</code></td>
<td>∅</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td><code>b := a + 1</code></td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>∅</td>
</tr>
<tr>
<td><code>c := c + b</code></td>
<td>b,c</td>
<td>c</td>
<td>b,c</td>
<td>∅</td>
</tr>
<tr>
<td><code>a := b * 2</code></td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>∅</td>
</tr>
<tr>
<td><code>a &lt; N</code></td>
<td>a</td>
<td>∅</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td><code>return c</code></td>
<td>c</td>
<td>∅</td>
<td>c</td>
<td>∅</td>
</tr>
</tbody>
</table>

*out[6] = \( \bigcup_{s \in \text{succ}[5]} \text{in}[s] \)

\[ = \emptyset\]
End of the First Iteration

<table>
<thead>
<tr>
<th>flowgraph</th>
<th>use</th>
<th>def</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:=0</td>
<td>∅</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>b:=a+1</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>∅</td>
</tr>
<tr>
<td>c:=c+b</td>
<td>b,c</td>
<td>c</td>
<td>b,c</td>
<td>∅</td>
</tr>
<tr>
<td>a:=b*2</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>∅</td>
</tr>
<tr>
<td>a&lt;N</td>
<td>a</td>
<td>∅</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>return c</td>
<td>c</td>
<td>∅</td>
<td>c</td>
<td>∅</td>
</tr>
</tbody>
</table>
Liveness Analysis—compute \( \text{in}[1] \)
flowgraph  use  def  in  out

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a:=0</td>
<td>∅</td>
<td>a</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b:=a+1</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>∅</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c:=c+b</td>
<td>b,c</td>
<td>c</td>
<td>b,c</td>
<td>∅</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a:=b*2</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>∅</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a&lt;N</td>
<td>a</td>
<td>∅</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return c</td>
<td>c</td>
<td>∅</td>
<td>c</td>
<td>∅</td>
</tr>
</tbody>
</table>

\[ \text{in}[1] = \text{use}[1] \cup (\text{out}[1] \setminus \text{def}[1]) \]

\[ = \emptyset \cup (\emptyset \setminus a) \]

\[ = \emptyset \]
\[
\text{out}[1] = \bigcup_{s \in \text{succ}[1]} \text{in}[s]
\]

Liveness Analysis—compute \textit{out}[1]

<table>
<thead>
<tr>
<th>flowgraph</th>
<th>use</th>
<th>def</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:=0</td>
<td>$\emptyset$</td>
<td>a</td>
<td>$\emptyset$</td>
<td>?</td>
</tr>
<tr>
<td>b:=a+1</td>
<td>a</td>
<td>b</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>c:=c+b</td>
<td>b,c</td>
<td>c</td>
<td>b,c</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>a:=b*2</td>
<td>b</td>
<td>a</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>a&lt;N</td>
<td>a</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>return c</td>
<td>c</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
The flowgraph shows the following assignments and operations:

1. `a := 0` (uses `∅` and defines `a`)
2. `b := a + 1` (uses `a` and defines `b`, `∅`)
3. `c := c + b` (uses `b, c` and defines `b, c, ∅`)
4. `a := b * 2` (uses `b` and defines `a`, `∅`)
5. `a < N` (uses `a` and defines `∅`)
6. `return c` (uses `c` and defines `∅`)

The `out[1]` is computed as:

\[ out[1] = \bigcup_{s \in \text{succ}[1]} in[s] \]

This results in:

\[ = in[2] \]

\[ = a \]

Liveness Analysis—compute `out[1]`
Completed Liveness Calculation

<table>
<thead>
<tr>
<th>flowgraph</th>
<th>stm</th>
<th>use</th>
<th>def</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>a := 0</td>
<td>1</td>
<td>∅</td>
<td>a,c</td>
<td>a,c</td>
<td></td>
</tr>
<tr>
<td>b := a + 1</td>
<td>2</td>
<td>a</td>
<td>b</td>
<td>a,c</td>
<td>b,c</td>
</tr>
<tr>
<td>c := c + b</td>
<td>3</td>
<td>b,c</td>
<td>c</td>
<td>b,c</td>
<td>b,c</td>
</tr>
<tr>
<td>a := b * 2</td>
<td>4</td>
<td>b</td>
<td>a</td>
<td>b,c</td>
<td>a,c</td>
</tr>
<tr>
<td>a &lt; N</td>
<td>5</td>
<td>a</td>
<td>∅</td>
<td>a,c</td>
<td>a,c</td>
</tr>
<tr>
<td>return c</td>
<td>6</td>
<td>c</td>
<td>∅</td>
<td>c</td>
<td>∅</td>
</tr>
</tbody>
</table>
Live Range of a

<table>
<thead>
<tr>
<th>flowgraph</th>
<th>stm</th>
<th>use</th>
<th>def</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:=0</td>
<td>1</td>
<td>∅</td>
<td>a</td>
<td>c</td>
<td>a,c</td>
</tr>
<tr>
<td>b:=a+1</td>
<td>2</td>
<td>a</td>
<td>b</td>
<td>a,c</td>
<td>b,c</td>
</tr>
<tr>
<td>c:=c+b</td>
<td>3</td>
<td>b,c</td>
<td>c</td>
<td>b,c</td>
<td>b,c</td>
</tr>
<tr>
<td>a:=b*2</td>
<td>4</td>
<td>b</td>
<td>a</td>
<td>b,c</td>
<td>a,c</td>
</tr>
<tr>
<td>a&lt;N</td>
<td>5</td>
<td>a</td>
<td>∅</td>
<td>a,c</td>
<td>a,c</td>
</tr>
<tr>
<td>return c</td>
<td>6</td>
<td>c</td>
<td>∅</td>
<td>c</td>
<td>∅</td>
</tr>
</tbody>
</table>

a live from 1–2, 4–5, and 5–2.
**Live Range of b**

<table>
<thead>
<tr>
<th>flowgraph</th>
<th>stm</th>
<th>use</th>
<th>def</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:=0</td>
<td>1</td>
<td>∅</td>
<td>a,c</td>
<td>a,c</td>
<td></td>
</tr>
<tr>
<td>b:=a+1</td>
<td>2</td>
<td>a</td>
<td>b</td>
<td>a,c</td>
<td>b,c</td>
</tr>
<tr>
<td>c:=c+b</td>
<td>3</td>
<td>b,c</td>
<td>c</td>
<td>b,c</td>
<td>b,c</td>
</tr>
<tr>
<td>a:=b*2</td>
<td>4</td>
<td>b</td>
<td>a</td>
<td>b,c</td>
<td>a,c</td>
</tr>
<tr>
<td>a&lt;N</td>
<td>5</td>
<td>a</td>
<td>∅</td>
<td>a,c</td>
<td>a,c</td>
</tr>
<tr>
<td>return c</td>
<td>6</td>
<td>c</td>
<td>∅</td>
<td>c</td>
<td>∅</td>
</tr>
</tbody>
</table>

b live from 2–3, 3–4.
### Live Range of c

<table>
<thead>
<tr>
<th>flowgraph</th>
<th>stm</th>
<th>use</th>
<th>def</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:=0</td>
<td>1</td>
<td>∅</td>
<td>a</td>
<td>c</td>
<td>a,c</td>
</tr>
<tr>
<td>b:=a+1</td>
<td>2</td>
<td>a</td>
<td>b</td>
<td>a,c</td>
<td>b,c</td>
</tr>
<tr>
<td>c:=c+b</td>
<td>3</td>
<td>b,c</td>
<td>c</td>
<td>b,c</td>
<td>b,c</td>
</tr>
<tr>
<td>a:=b*2</td>
<td>4</td>
<td>b</td>
<td>a</td>
<td>b,c</td>
<td>a,c</td>
</tr>
<tr>
<td>a&lt;N</td>
<td>5</td>
<td>a</td>
<td>∅</td>
<td>a,c</td>
<td>a,c</td>
</tr>
<tr>
<td>return</td>
<td>6</td>
<td>c</td>
<td>∅</td>
<td>c</td>
<td>∅</td>
</tr>
</tbody>
</table>

c live into 1 and from 1–2, 2–3, 3–4, 4–5, 5–6, and 5–2.
### Live Ranges of the Variables

<table>
<thead>
<tr>
<th>flowgraph</th>
<th>stm</th>
<th>use</th>
<th>def</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:=0</td>
<td>1</td>
<td>∅</td>
<td>a</td>
<td>c</td>
<td>a,c</td>
</tr>
<tr>
<td>b:=a+1</td>
<td>2</td>
<td>a</td>
<td>b</td>
<td>a,c</td>
<td>b,c</td>
</tr>
<tr>
<td>c:=c+b</td>
<td>3</td>
<td>b,c</td>
<td>c</td>
<td>b,c</td>
<td>b,c</td>
</tr>
<tr>
<td>a:=b*2</td>
<td>4</td>
<td>b</td>
<td>a</td>
<td>b,c</td>
<td>a,c</td>
</tr>
<tr>
<td>a&lt;N</td>
<td>5</td>
<td>a</td>
<td>∅</td>
<td>a,c</td>
<td>a,c</td>
</tr>
<tr>
<td>return c</td>
<td>6</td>
<td>c</td>
<td>∅</td>
<td>c</td>
<td>∅</td>
</tr>
</tbody>
</table>
Interference

At any node that defines a variable $x$, $x$ interferes with $y$ for any variable $y$ which is live-out.

$x$ interferes with $y$ if $x \in \text{def}[n]$ and $y \in \text{out}[n]$. 
Conclusions

<table>
<thead>
<tr>
<th>Flowgraph</th>
<th>Stmt</th>
<th>Use</th>
<th>Def</th>
<th>In</th>
<th>Out</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:=0</td>
<td>1</td>
<td>∅</td>
<td>a</td>
<td>c</td>
<td>a,c</td>
<td>a interferes with c</td>
</tr>
<tr>
<td>b:=a+1</td>
<td>2</td>
<td>a</td>
<td>b</td>
<td>a,c</td>
<td>b,c</td>
<td>b interferes with c</td>
</tr>
<tr>
<td>c:=c+b</td>
<td>3</td>
<td>b,c</td>
<td>c</td>
<td>b,c</td>
<td>b,c</td>
<td>c interferes with b</td>
</tr>
<tr>
<td>a:=b*2</td>
<td>4</td>
<td>b</td>
<td>a</td>
<td>b,c</td>
<td>a,c</td>
<td>a interferes with c</td>
</tr>
<tr>
<td>a&lt;N</td>
<td>5</td>
<td>a</td>
<td>∅</td>
<td>a,c</td>
<td>a,c</td>
<td></td>
</tr>
<tr>
<td>return c</td>
<td>6</td>
<td>c</td>
<td>∅</td>
<td>c</td>
<td>∅</td>
<td></td>
</tr>
</tbody>
</table>
Interference Matrix, Interference Graph

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Liveness Analysis

Liveness information is used for several kinds of optimizations in a compiler.

Appel, 2nd, page 212.

1. register allocation
2. detecting uninitialized variables
3. dead code elimination
Graph Coloring

An interference graph can be created by taking the nodes of the graph to be program variables or temporaries. There is an edge between two nodes if the variables interfere. The most common form of interference is caused by overlapping live ranges.

We can think of the map from temporaries to registers as coloring the temporaries. Each register has its own color $r_1, r_2, r_3$ etc. Coloring the vertices of the interference graph with as few colors as possible corresponding to mapping the temporaries to as few registers as possible.

Two colors suffice.
Interference

Recall that we said a variable $x$, $x$ interferes with $y$ for any variable $y$ which is live-out.

It is important not to create any unnecessary interferences between variables (temporaries). A move/copy instruction $x := z$ may introduce such an unnecessary interference. In fact, assigning $x$ and $z$ the same register would make the move instruction unnecessary. For this reason, we define interference for move statements specifically.

At any move statement $x := z$, $x$ interferes with $y$ for any variable $y$ which is not equal to $z$ and is live-out.
Yet, we can allocate the same register for $t$ and $s$. (As long as there is no future, non-move, definition of $t$, e.g., $t:=3$.)
Chapter 11: Register Allocation
RISC vs CICS

1. 32 registers,

2. only one class of integer/pointer registers,

3. arithmetic operations only between registers,

4. “three-address” instructions of the form
   \[ r_1 \leftarrow r_2 \oplus r_3 \]

5. load and store instructions with only the M[reg+const] addressing mode,

6. every instruction exactly 32 bits long,

7. one result or effect per instruction.
RISC vs CICS

1. few registers (16, or 8, 6),

2. registers divided into different classes, with some operations available only on certain registers,

3. arithmetic operations can access registers or memory through “addressing modes,”

4. “two-address” instructions of the form
   \[ r_1 \leftarrow r_1 \oplus r_2 \]

5. several different addressing modes,

6. variable-length instructions, formed from variable-length opcode plus variable-length addressing modes,

7. instructions with side effects such as “auto-increment” addressing modes.
How many registers does an `tree.Exp` need?

**CONST** (int value) 0 registers

**TEMP** 0 registers

**BINOP**
- $\max(l,r) + 1$ if $(l=r)$
- $\max(l,r)$ otherwise

**MEM** (Exp exp) 1

**CALL** () 0

**NAME** () [unused]

**ESEQ** (Stm stm, Exp exp) [unused]
Sethi-Ullman Algorithm

Left subtree is name or constant.

1. \( R := \text{top (rstack)} \)

2. \text{emit: } R := \text{name (move or load instruction)}
**Sethi-Ullman Algorithm**

Left subtree \( t_1 \) requires more registers than right subtree \( t_2 \).

1. \( \text{gencode} \ (t_1) \)

2. \( R := \text{pop} \ (rstack) \)

3. \( \text{gencode} \ (t_2) \)

4. \( R' := \text{top} \ (rstack) \)

5. \( \text{emit: } R := R \ast R' \)

6. \( \text{push} \ (R, rstack) \)
Sethi-Ullman Algorithm

Right subtree $t_2$ requires as many or more registers as left subtree $t_1$.

1. $R := \text{top (rstack)}$

2. swap (rstack)

3. genode ($t_2$)

4. $R' := \text{pop (rstack)}$

5. genode ($t_1$)

6. emit: $R := R \star R'$

7. push (R', rstack)

8. swap (rstack)
Register Spilling

Together the subtrees requires more registers than are available...

1. \( R := \text{top (rstack)} \)

2. \( \text{gencode (} t_2 \text{)} \)

3. \( T := \text{pop (tstack)} \)

4. \( \text{emit: } T := R \)

5. \( \text{gencode (} t_1 \text{)} \)

6. \( \text{emit: } R := R \times T \)

7. \( \text{push (} T, \text{ tstack)} \)

Keep a stack of available temporaries to be stored in memory (stack).
gencode((t\sb4\)) [%l0 %l1 %l2 %l3]
  swap
gencode((t\sb3\)) [%l1 %l0 %l2 %l3]
  swap
gencode((t\sb2\)) [%l0 %l1 %l2 %l3]
  swap
gencode(d) [%l1 %l0 %l2 %l3]
  emit:  ld %l1,[d]
pop %l1
gencode(c) [%l0 %l2 %l3]
  emit:  ld %l0,[c]
  emit:  add %l1,%l0,%l0  ! %l0:=%l1+%l0
  push %l1;  swap
pop %l0
gencode(e) [%l1 %l2 %l3]
  emit:  ld %l1,[e]
  emit:  sub %l1,%l0,%l1  ! %l1:=%l1-%l0
push %l0; swap
pop %l1
gencode (\(t\{sb1\}\) ) [%l0 %l2 %l3]
swap
gencode (b) [%l2 , %l0 %l3]
emit: ld %l2 , [b]
pop %l2
gencode (c) [%l0 %l3]
emit: ld %l0 , [c]
emit: add %l2 ,%l0 ,%l0 ! %l0:=%l2+%l0
push %l2; swap
emit: sub %l1 ,%l0 ,%l0 ! %l0:=%l1-%l0
Example

emit:   ld %l1,[d]
emit:   ld %l0,[c]
emit:   add %l1,%l0,%l0 ! %l0 := %l1+%l0
emit:   ld %l1,[e]
emit:   sub %l1,%l0,%l1 ! %l1 := %l1-%l0
emit:   ld %l2,[b]
emit:   ld %l0,[c]
emit:   add %l2,%l0,%l0 ! %l0 := %l2+%l0
emit:   sub %l1,%l0,%l0 ! %l0 := %l1-%l0
Spilling

\textbf{add} \quad t19, t21, t21 \quad ! \quad t21 := t19 + t21

If no registers for \( t19 \) and \( t21 \), we must allocate them in the stack frame.

\textbf{ld} \quad [%fp-36],%g1 \quad ! \quad %g1 := t19 \ (\text{local at offset} \ -36) \ ; \text{spill}

\textbf{ld} \quad [%fp-44],%g2 \quad ! \quad %g2 := t21 \ (\text{local at offset} \ -44) \ ; \text{spill}

\textbf{add} \quad %g1, \ %g2, \ %g1 \quad ! \quad %g1 := %g1 + %g2

\textbf{st} \quad %g1,[%fp-44] \quad ! \quad t21 := %g1 \ (\text{local at offset} \ -44) \ ; \text{spill}