Live Range Analysis

The main idea behind Live Range Analysis for register allocation is to determine when a variable is alive, i.e. when it is born (defined), and when it dies (last used). Performing live range analysis within a basic block is very straight-forward. We will see how this is accomplished in a later unit. However, to achieve global register allocation or any global optimization, live range analysis must be performed over the entire procedure, not just a single basic block. This will require *data-flow information* that a global register allocator collects by a process known as *data-flow analysis*. Data-flow information can be collected by setting up and solving systems of equations that relate information at various points in a program. A typical equation for live variable calculation might look like

$$\mathit{in}\,[B] = \mathit{use}\,[B] \cup (\mathit{out}\,[B] - \mathit{def}\,[B])$$

This equation can be read as the variables that have values at the beginning of a basic block are dependent on either the values that are generated in the sequence of statements executed or the variables that had values at the end of the statements being executed minus those variables that are defined as the control flows through the basic block. This is a typical *data-flow equation*.

Graphically we have

More formally, let

Def: $\mathit{in}\,[B] = \{ V_i \mid V_i \text{ is live when the basic block B is about to be executed} \}$

Informally, *in* [B] is the set of variables that are live at the point immediately before basic block B is executed.

Def: $\mathit{out}\,[B] = \{ V_i \mid V_i \text{ is live when the basic block B has finished executing} \}$

Informally, out [B] is the set of variables that are live at the point immediately after basic block B has executed.

Def: $\mathit{def}\,[B] = \{ V_i \mid V_i \text{ is assigned a value in basic block B prior to any use of } V_i \text{ in B} \}$

Informally, *def* [B] is the set of variables that assigned values, i.e. appear to the left of an assignment operation, in basic block B before any use of that variable in B.

Def: *use* [B] = { V_i | V_i is used in basic block B prior to any definition of V_i }

Informally, *use* [B] is the set of variables that are used, i.e. appear to the right of an assignment operation, in basic block B. Those variables may be used prior to any definition of the variable in B.

The equations that relate *def* and *use* to the unknowns *in* and *out* are:

$$in[B] = use[B] \cup (out[B] - def[B])$$

and

$$out[B] = \bigcup_{\substack{S\ a \\ successor \\ of\ B}} in[S]$$

The first equation says that a variable is live coming into a block if either it is used before redefinition in the block or it is live coming out of the block and is not redefined in the block.

The second equation says that a variable is live coming out of a block if and only if it is live coming into one of its successors.

The algorithm below computes the variables that are live for each basic block in a program.

Live Variable Analysis Algorithm

Input: A flow control graph with def and use computed for each block.

Output: *out* [B], the set of variables live on exit from each basic block B in the control flow graph.

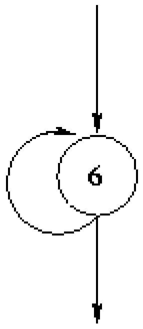
```
for each block B do
    $in[B] = $
while changes to any of the in's occur do
    for each block B do
        $out[B] = \bigcup_{S \text{ a successor of } B} in[S]$
        $in[B] = use[B] \cup (out[B] - def[B])$
```

Example

In the previous unit, we had the code segment

```
50   L5     label
51   I$30   +      c      d
52   I$31   +      e      I$30          use [6]  = { c,  d,  e,  i,  k }
53   s      =s     I$31   k
54   I$32   +      10     i
55   s      =s     I$32   e             def [6]  = { s }
56   I$33   +      1      e
57   k      =      I$33
58   B$34   <      k      20
59   L5     JPC    B$34
```

The following graph depicts this code segment



Assume that k is live at the exit of block 6, i.e. *in* [7] = { k }. Initially we have that

in [6] =

iteration 1:

out [6] = { k }

in [6] = *use* [6] U (*out* [6] - *def* [6]) = { c, d, e, i, k } U { { k } - { s }) = { c, d, e, i, k }

iteration 2:

out [6] = *in* [6] U *in* [7] = { c, d, e, i, k } U { k } = { c, d, e, i, k }

in [6] = *use* [6] U (*out* [6] - *def* [6]) = { c, d, e, i, k } U { { c, d, e, I, k } - { s })
= { c, d, e, i, k }

Iteration 3:

out [6] = *in* [6] U *in* [7] = { c, d, e, i, k } U { k } = { c, d, e, i, k }

in [6] = *use* [6] U (*out* [6] - *def* [6]) = { c, d, e, i, k } U { { c, d, e, I, k } - { s })
= { c, d, e, i, k }

After the third iteration the *out*s and *in*s do not change. Hence, the algorithm terminates. Note that when the entire control flow graph is considered, there may be more iterations.

Let's consider another example. Suppose that we have the following program segment:

```
DO
    a ← b + c ;
    d ← - a ;
    e ← d + f ;
    IF ( a < d ) THEN
        f ← 2 * e ;
    ELSE
        b ← d + e ;
        e ← e - 1 ;
    END
    b ← f + c ;
UNTIL ( b > 10 ) END
```

The 4-tuples that we would get from the above code segment are:

1.	L$0	LABEL		
2.	I$0	IADD	b	c
3.	a	STORE	I$0	
4.	I$1	IMINUS	0	a
5.	d	STORE	I$1	
6.	I$2	IADD	d	f
7.	e	STORE	I$2	
8.	B$0	LSTH	a	d
9.	L$1	CJUMPF	B$0	
10.	I$3	IMULT	2	e
11.	f	STORE	I$3	
12.	L$2	JUMP		
13.	L$1	LABEL		
14.	I$4	IADD	d	e
15.	b	STORE	I$4	
16.	I$5	ISUB	e	1
17.	e	STORE	i$5	
18.	l$2	LABEL		
19.	I$6	IADD	f	c
20.	b	STORE	I$6	
21.	B$1	GRTH	b	10
22.	L$0	CJUMPF	B$1	

Next we would identify the set of leaders

{   }

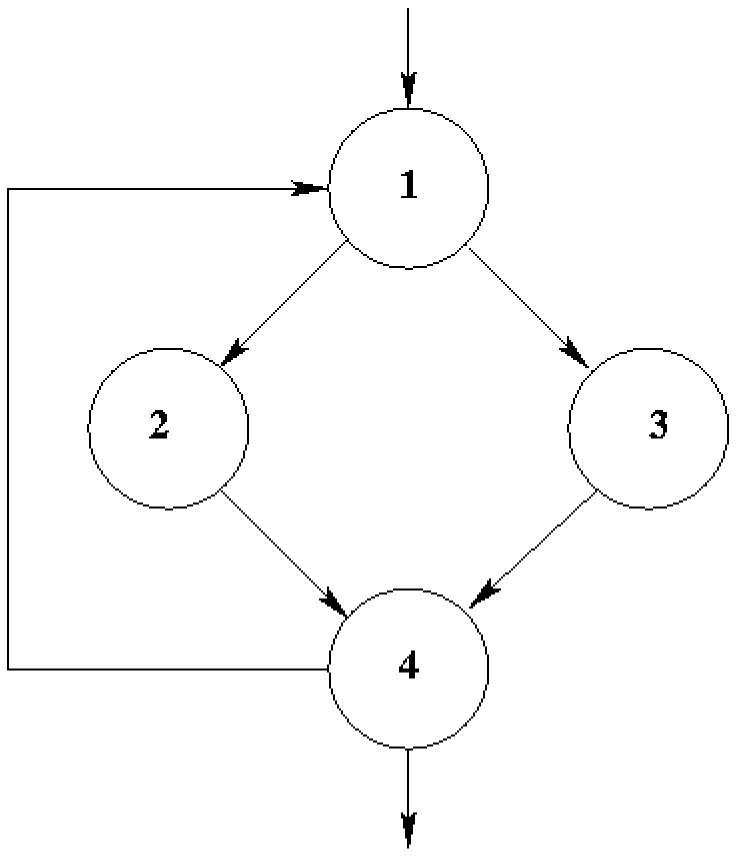
Click here for the set of leaders.

Basic Block	Beginning 4-tuple	Ending 4-tuple	Next 4-tuples
1	1		
2			
3			

4

Click here for the answer.

The control flow graph for the above program segment is



We have the following 4-tuples, *use*, and *def* for the basic blocks:

Basic Block 1:

1.	L$0	LABEL			
2.	I$0	IADD	b	c	*use* [1] = { b, c, f }
3.	a	STORE	I$0		
4.	I$1	IMINUS	0	a	
5.	d	STORE	I$1		
6.	I$2	IADD	d	f	*def* [1] = { a, d, e }
7.	e	STORE	I$2		
8.	B$0	LT	e	a	
9.	L$1	CJUMPF	B$0		

Basic Block 2:

```
10.   I$3   IMULT    2     e          use [2]  = { e }
11.   f     STORE    I$3
12.   L$2   JUMP                      def [2]  = { f }
```

Basic Block 3:

```
13.   L$1   LABEL
14.   I$4   IADD     d     e          use [3]  = { d,  e }
15.   b     STORE    I$4
16.   I$5   ISUB     e     1          def [3]  = { b  }
17.   e     STORE    i$5
```

Basic Block 4:

```
18.   l$2   LABEL
19.   I$6   IADD     f     c          use [4]  = { c,  f }
20.   b     STORE    I$6
21.   B$1   GT       b     10         def [4]  = { b }
22.   L$0   CJUMPF   B$1
```

We will now compute the *in*s and *out*s for the basic blocks. Assuming that b is live after basic block 4, we initially have

in [1] = *in* [2] = *in* [3] = *in* [4] =

Iteration 1:

out [4] =

in [4] = *use* [4] U (*out* [4] - *def* [4]) =

out [3] = *in* [4] =

in [3] = *use* [3] U (*out* [3] - *def* [3]) =

out [2] = *in* [4] =

in [2] = *use* [2] U (*out* [2] - *def* [2]) =

out [1] = *in* [2] U *in* [3] =

in [1] = *use* [1] U (*out* [1] - *def* [1]) =

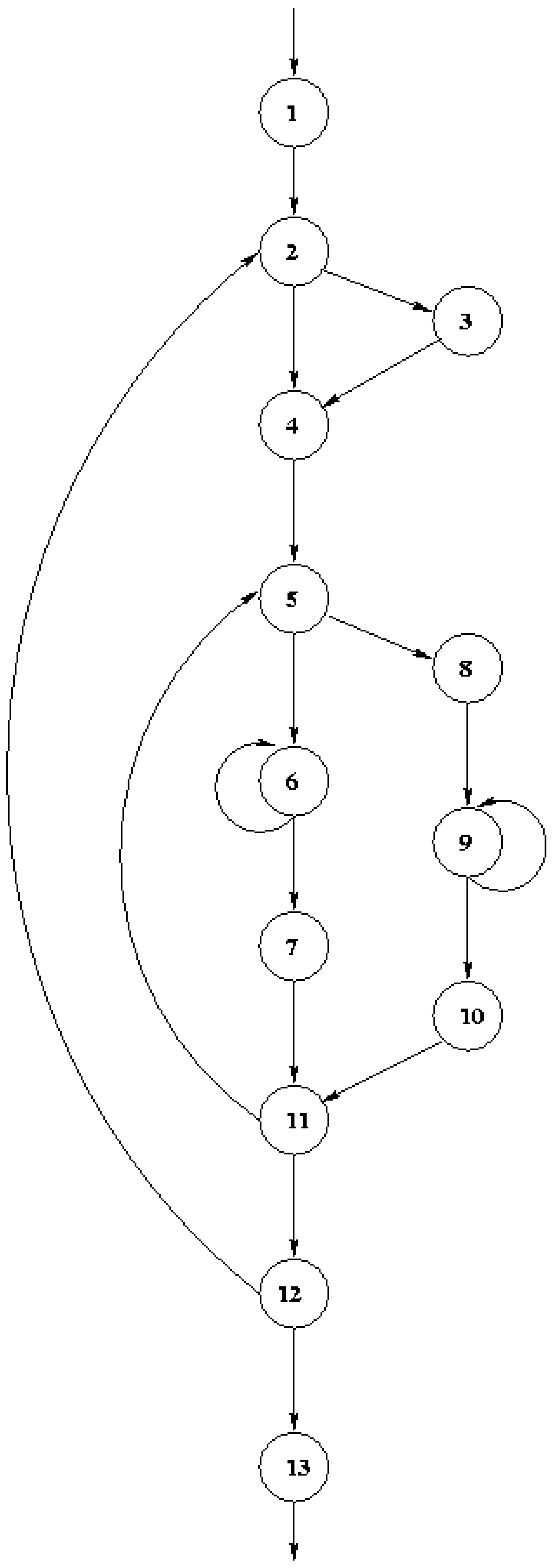
Click here for the rest of the calculations.

Homework:

Using a segment depicted below from the control flow graph in the previous unit, we have

1	i	=	1		50	L5	label		
2	a	=	0		51	I$30	+	c	d
3	b	=	10		52	I$31	+	e	I$30
4	L1	label			53	s	$=_s$	I$31	k
5	I$1	*	1	b	54	I$32	+	10	i
6	I$2	+	a	I$1	55	s	$=_s$	I$32	e
7	I$3	*	i	I$2	56	I$33	+	1	e
8	c	=	I$3		57	k	=	I$33	
9	I$4	+	1	i	58	B$34	<	k	20
10	I$5	*	b	I$4	59	L5	JPC	B$34	
11	d3	=	I$5		60	k	=	25	
12	I$6	+	1	i	61	L6	JP		
13	I$7	*	c	I$6	62	L4	label		
14	d	=	I$7		63	k	=	15	
15	I$8	+	1	i	64	L7	label		
16	I$9	*	c	I$8	65	I$35	+	10	i
17	I$10	+	a	I$9	66	I$36	*	b	I$35
18	e	=	I$10		67	d	=	I$36	
19	I$11	*	a	I$9	68	I$37	+	3	k
20	d1	=	I$11		69	s	$=_s$	I$37	k
21	B$12	<	c	10	70	I$38	+	1	k
22	L2	JPC	B$12		71	k	=	I$38	
23	I$13	+	e	i	72	B$39	<	k	30
24	e	=	I$13		73	L7	JPC	B$39	
25	I$14	*	b	i	74	k	=	25	
26	d1	=	I$14		75	L6	label		
27	I$15	+	1	b	76	I$40	*	3	a
28	I$16	LD_s	s	I$15	77	I$41	+	k	I$40
29	I$17	+	2	b	78	e	=	I$41	
30	I$18	LD_s	s	I$17	79	I$42	+	1	i
31	I$19	+	I$16	I$18	80	I$43	*	c	I$42
32	I$20	+	1	a	81	k	=	I$43	
33	s	$=_s$	I$19	I$20	82	B$44	≠	k	30
34	I$31	*	b	d1	83	L3	JPC	B$44	
35	I$22	*	i	I$21	84	I$45	*	a	e
36	c	=	I$22		85	I$46	*	b	I$45
37	L2	label			86	d	=	I$46	
38	I$23	+	d	e	87	I$47	+	c	d
39	e	=	I$23		88	I$48	+	i	I$47
40	L3	label			89	k	=	I$48	
41	I$24	LD_s	s	i	90	I$49	+	1	i
42	I$25	+	c	I$24	91	i	=	I$49	
43	d2	=	I$25		92	B$50	<	i	10
44	I$26	+	1	I	93	L1	JPC	B$50	
45	I$27	*	b	I$26	94	I$51	+	c	k
46	c	=	I$27		95	s	$=_s$	I$51	b
47	k	=	10		96	I$25	+	i	d3
48	B$28	<	c	20	97	i	=	I$52	
49	L4	JPC	B$28						

Assume that i is alive after 4-tuple 97.



ANSWERS:

{ 1, 10, 13, 18 }

Return

Basic Block	Beginning 4-tuple	Ending 4-tuple	Next 4-tuples
1	1	9	10, 13
2	10	12	18
3	13	17	18
4	18	22	

Return

Assuming that b is live after the code segment, we have

in [1] = *in* [2] = *in* [3] = *in* [4] =

Iteration 1:

out [4] = { b }

in [4] = *use* [4] U (*out* [4] - *def* [4]) = { c, f } U ({ b } - { b }) = { c, f }

out [3] = *in* [4] = { c, f }

in [3] = *use* [3] U (*out* [3] - *def* [3]) = { d, e } U ({ c, f } - { b }) = { c, d, e, f }

out [2] = *in* [4] = { c, f }

in [2] = *use* [2] U (*out* [2] - *def* [2]) = { e } U ({ c, f } - { f }) = { c, e }

out [1] = *in* [2] U *in* [3] = { c, e } U { c, d, e, f } = { c, d, e, f }

in [1] = *use* [1] U (*out* [1] - *def* [1]) = { b, c, f } U ({ c, d, e, f } - { a, d, e }) = { b, c, f }

Iteration 2:

out [4] = in [1] U { b } = { b, c, f } U { b } = { b, c, f } *in* [4] = *use* [4] U (*out* [4] - *def* [4]) = { c, f } U ({ b, c, f } - { b }) = { c, f }

out [3] = *in* [4] = { c, f }

in [3] = *use* [3] U (*out* [3] - *def* [3]) = { d, e } U ({ c, f } - { b }) = { c, d, e, f }

out [2] = *in* [4] = { c, f }

in [2] = *use* [2] U (*out* [2] - *def* [2]) = { e } U ({ c, f } - { f }) = { c, e }

out [1] = *in* [2] U *in* [3] = { c, e } U { c, d, e, f } = { c, d, e, f }

in [1] = *use* [1] U (*out* [1] - *def* [1]) = { b, c, f } U ({ c, d, e, f } - { a, d, e }) = { b, c, f }

Note: Those lines in **red** indicate that the set of live variables have not changed from the previous iteration.

Since the *out* [4] did change in the second iteration we would need to do one more iteration before the algorithm terminates.

Return