Enoch Hwang

✶ ehwang@cs.ucr.edu ✶

CS201 Compiler
Notes
December 7, 1995

Reaching Definitions

reaches the entrance of P

Upward analysis
Forward analysis

Reachable Uses (Live)

reachable from the exit of P

Upward analysis
Backward analysis

Fresh / Available / Dominates

reaches the entrance of P

Fresh/Available at P if Fresh/Available at A and B
⇒ intersection ⇒ Downward analysis
Forward analysis

Dead / Busy

reachable from the exit of P

Dead/Busy at P if Dead/Busy at A and B
⇒ intersection ⇒ Downwardanalysis
Backward analysis

UnFresh / UnAvailable

reaches the entrance of P

UnFresh/UnAvailable at P if UnFresh/UnAvailable at A or B
⇒ union ⇒ Upward analysis
Forward analysis

Live / UnBusy

reachable from the exit of P

Live/UnBusy at P if Live/UnBusy at A or B
⇒ union ⇒ Upward analysis
Backward analysis

Forward - The close set B is computed in terms of the predecessor sets P.

Backward - The close set B is computed in terms of the successor sets S.

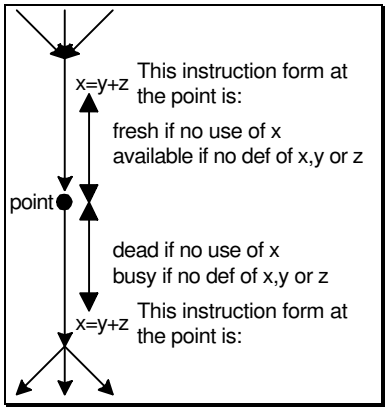
Downward - Starting with everything (i.e. close set has everything), the sets get **smaller** by eliminating elements. (intersection, &)

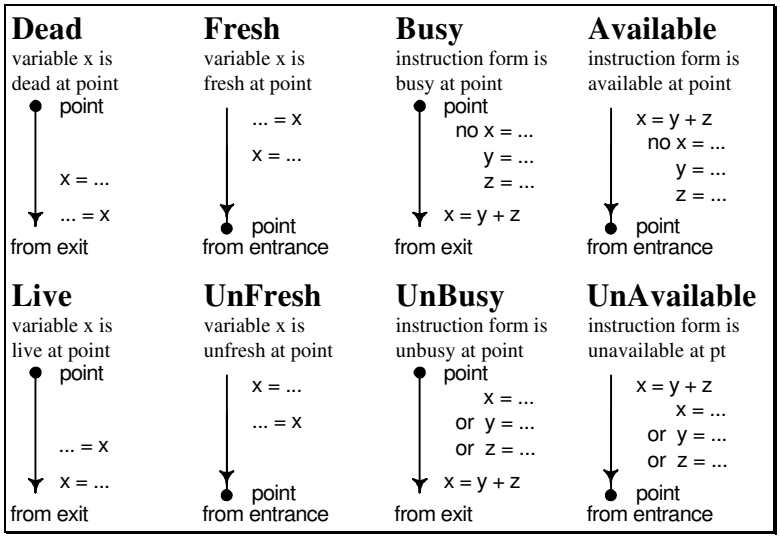
Upward - Starting with nothing (i.e. close set is empty), the sets get **bigger** by adding elements. (union, +)

Ex.
```
while (changes occur)
    for (each block B)
        for (each direct predecessor P of B)          // forward
            B.close += P.gen + (P.close - P.kill)     // upward

while (changes occur)
    for (each block B)
        for (each direct successor S of B)            // backward
            B.close &= S.gen + (S.close - S.kill)     // downward
```

- Upward analysis - include as much as possible (all possible uses) in the GEN set.
- Downward analysis - include as much as possible (all possible kills) in the KILL set.





At a point on a path **p** through the flow graph, a given instruction form, say X = Y + Z, is said to be:

- *p-fresh* if before that point, X is defined after its last use, if any. (X is fresh if it hasn't been use after defining it.) (fresh from entrance.)
- *p-available* if before that point, it has an occurrence without an intervening definition of X, Y, or Z. (Haven't been changed.)
- *p-dead* if after that point, X is defined before its next use, if any. (X is dead if you don't use it after defining it.)
- *p-busy* if after that point, it has an occurrence without an intervening definition of X, Y, or Z. (It will be used.)
- *unfresh* if before point, X is defined before its last use.
- *unavailable* if before point, it has an occurrence with an intervening definition of X, Y, or Z.
- A variable X is *live* at a point if and only if there is a use of the variable reachable from that point. (live from exit). i.e. if after that point, X is defined after its next use, if any.
- *unbusy* if after point, it has an occurrence with an intervening definition of X, Y, or Z.

Reaching Definitions (from entrance; forward; upward)

B.close = the set of all global **definitions** that have reached the **entrance** of B.

```
ALL denotes union {B.def : B is a block}                  // all global definitions
for (each block B)                                        // initialize B.
     B.close is a subset of ALL :
          B.close[<X', B'>] = 0 for all X' and B';        // the empty set.  notation <X,B>
     B.gen is a subset of ALL :                           // ⇒ variable X in block B.
          B.gen[<X', B'>] = 1 iff B' is B;                // i.e. B.gen = B.def
     B.kill is a subset of ALL :                          // i.e. all definitions in block B
          B.kill[<X', B'>] = 1 iff X' is defined in B;

while (changes occur)
     for (each block B)
          for (each direct predecessor P of B)
               B.close += P.gen + (P.close - P.kill)
Second version of the above while loop
do {changing = 0;
     for (each block P)                                   // from top to bottom
```

```
        pass = P.gen + (P.close - P.kill);
        for (each direct successor B of P)              // forward
            if (B.close doesn't include pass)
                changing = 1;
                B.close += pass;                        // upward
} while (changing);
```

Reachable Use (from exit; backward; upward)

B.close = the set of all **uses** of X reachable from the **exit** of B.

```
ALL denotes union {B.use : B is a block}                // all global uses
for (each block B)
    B.close is a subset of ALL :                        // initialize B.
        B.close[<X', B'>] = 0 for all X' and B';        // the empty set
    B.gen is a subset of ALL :
        B.gen[<X', B'>] = 1 iff B' is B;                // B.gen = B.use
    B.kill is a subset of ALL :
        B.kill[<X', B'>] = 1 iff X' is defined in B;

do {changing = 0;
    for (each block S)                                  // from bottom to top
        pass = S.gen + (S.close - S.kill);
        for (each direct predecessor block B of S)      // backward
            if (B.close doesn't include pass)
                changing = 1;
                B.close += pass;                        // upward
} while (changing);
```

Dead Code Analysis (dead at exit; backward; downward)

B.close = the set of all **instructions that are dead** at the exit of each block.

```
for (each block B)
    B.close[x] = 1 for all variables x;
    B.gen[x] = 1 iff x has a definition with no previous use in B;
    B.kill[x] = 1 iff x is used in B;                   // dead code ⇒ "if you don't use it"
                                                        // ⇒ if you use it, you're killing it.
while (changes occur)
    for (each block B)
        for (each direct successor S of B)              // backward
            B.close &= S.gen + (S.close - S.kill);      // downward
```

Dead Code Elimination

```
do reachable use to get B.close;
for (each block B) {
    VariableSet currentlyLive = all X having a use in B.close;
    for (each instruction in B from last to first) {    // backward
        if (currentlyLive[instruction.destination]) {
            currentlyLive −= instruction.destination;
            currentlyLive += instruction.source;
            currentlyLive += instruction.source2;
        } else
            remove instruction;
    }
```

Live (UnDead) Code Analysis (live at exit; backward; upward)

B.close = the set of all **instructions that are live** at the exit of each block.

```
for (each block B)
    B.close[x] = empty;
    B.gen[x] = 1 iff x has a use in B;
    B.kill[x] = 1 iff x is defined in B;

while (changes occur)
    for (each block B)
        for (each direct successor S of B)
            B.close += S.gen + (S.close - S.kill);
```

Fresh code Analysis (fresh at entrance; forward; downward)

an instruction is fresh at the entrance to B iff its destination is in B.close.

```
for (each block B){
    B.close[x] = 1 for all variables x;
    B.gen[x] = 1 iff x has a definition with no subsequent use in B;
    B.kill[x] = 1 iff x is used in B;
}

while (changes occur)
    for (each block B)
        for (each direct predecessor P of B)                // forward
            B.close &= P.gen + (P.close - P.kill);          // downward

for (each block B) B.fresh = B.close;
```

UnFresh code Analysis (unfresh at entrance; forward; upward)

An instruction is fresh at the entrance of B iff its destination is in B.fresh, as computed below:

```
ALL denotes the set of all variables occurring in the program.
for (each block B){
    B.close[x] = 0 for all variables x;
    B.gen[x] = 1 iff x has a use with no subsequent definition in B;
    B.kill[x] = 1 iff x is defined in B;                    // if you define it then it becomes fresh
}

while (changes occur)
    for (each block B)
        for (each direct predecessor P of B)                // forward
            B.close += P.gen + (P.close - P.kill);          // upward

for (each block B) B.fresh = ALL - B.close;                 // B.close = unfresh; B.fresh = fresh
```

Busy Code Analysis (busy at exit; backward; downward)

B.close = the set of all **instructions that are busy** at the exit of each block.

```
All denotes the set of all instruction forms in the program.
for (each block B)
    B.close = ALL;
    B.gen = ALL instructions having an occurrence in B that is reachable from the entrance of B;
    B.kill = ALL instructions having an operand defined in B;

while (changes occur)
    for (each block B)
```

```
        for (each direct successor S of B)
            B.close &= S.gen + (S.close - S.kill);
```

UnBusy (Free) Code Analysis (not busy at exit; backward; upward)

B.close = the set of all **instructions that are not busy** at the exit of each block.

```
All denotes the set of all instruction forms in the program.
for (each block B)
    B.close = empty;
    B.gen = ALL instns having an occurrence in B that is not reachable from the entrance of B;
          = ALL instructions having an operand defined in B.
    B.kill = ALL instructions having an occurrence  in B that is reachable from the entrance of B;

while (changes occur)
    for (each block B)
        for (each direct successor S of B)
            B.close += S.gen + (S.close - S.kill);
```

Available Code Analysis (available at entrance; forward; downward)

B.close = the set of all **instructions available** at the entrance of each block.

```
ALL denotes the set of all instruction forms in the program.
for (each block B){
    B.close = ALL;
    B.gen = ALL instructions having an occurrence in B that reaches an exit of B;
    B.kill = ALL instructions having an operand defined in B;
}
for (each entrance block B) B.close = empty set;
```

First version of the while loop

```
while (changes occur)
    for (each block B)
        for (each direct predecessor P of B)
            B.close &= P.gen + (P.close - P.kill);
```

Second version of the above while loop

```
do {
    changing = 0;
    for (each block P){
        pass = P.gen + (P.close - P.kill);
        for (each direct successor block B of P)          // forward
            if (pass doesn't include B.close){
                changing =  1;
                B.close &= pass;                          // downward
            }
    }
} while (changing)
```

Available Code Elimination

```
for (each instruction i){
    Kd[i] = set of all instruction forms of which i.destination is a destination.
    Ks[i] = set of all instruction forms of which i.destination is a source.
}

for (each block B){
    InstructionSet currentlyAvailable = B.close;
```

```
    for (each instruction occurrence in B from first to last){          // forward
        if (currentlyAvailable[instruction]
            remove instruction;
        else {
            currentlyAvailable −= Kd[instruction.form];
            currentlyAvailable += instruction.form;
            currentlyAvailable −= Ks[instruction.form];
        }
    }
}
```

Unavailable Code Analysis

```
ALL denotes the set of all instruction forms in the program.
for (each block B){
    B.close = emptyL;
    B.gen = ALL instructions having an occurrence in B that does not reach the exit of B;
        = ALL instructions having an operand defined in B.
    B.kill = ALL instructions having an occurrence in B that reaches the exit of B;
}
```

Dominates

A block B dominates a given block B' iff all control paths from each entrance block to B' reach (have to go through) B.

To find which block dominates which: (forward, downward)

```
for (each block B) B.dom = ALL basic blocks
for (each entrance block B0) B0.dom = {B0}
while (changing)
    for (each block B from first to last)                    // forward
        B.dom = {B} + intersection{P.dom : P is a direct predecessor of B}
```

Induction Variables

A *basic induction variable I* is one whose definition in the loop merely increments or decrements *I* by a loop invariant, i.e., they are all of the form: $I = I \pm C$.

A variable X is an *I-based nonbasic induction variable* (of rank *n*) if and only if it has a single definition in the loop, and that one has the form:

$$X = J \; op \; D$$

where

1. *op* is +, −, ×, or ÷.
2. D is loop invariant.
3. J is an I-based induction variable of rank *n*-1. (Assume I itself has rank 0.)
4. If $n > 1$ then J's lone definition dominates this definition of X with respect to the loop entrance and all definitions of I.

LR Parsers

Given the following augmented grammar:

S' → S
1 S → CC
2 C → cC
3 C → d

The set of LR(1) items are as follows: Carry the 2nd component along with the *goto*s.

I_0:	S' → .S, $	goto I_1(S)	close(.S); first(βa) = first(ε$) = first($) = $
	S → .CC, $	goto I_2(C)	close(.C); first(C$) = first(C) = c/d
	C → .cC, c/d	goto I_3(c)	from close(.C)
	C → .d, c/d	goto I_4(d)	from close(.C)
I_1:	S' → S., $	//	
I_2:	S → C.C, $	goto I_5(C)	close(.C); first($) = $
	C → .cC, $	goto I_6(c)	
	C → .d, $	goto I_7(d)	
I_3:	C → c.C, c/d	goto I_8(C)	close(.C);
	C → .cC, c/d	goto I_3(c)	
	C → .d c/d	goto I_4(d)	
I_4:	C → d., c/d	//	
I_5:	S → CC., $	//	
I_6:	C → c.C, $	goto I_9(C)	
	C → .cC, $	goto I_6(c)	
	C → .d, $	goto I_7(d)	
I_7:	C → d., $	//	
I_8:	C → cC., c/d	//	
I_9:	C → cC., $	//	

The LR(1) parsing table is:

State	action			goto	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

All the *shifts* and *gotos* are from the goto edges. All the *reduce* are from the empty states for all the 2nd component terminals.

The set of LALR items are as follows:

I_0:	S' → .S, $	goto I_1(S)	close(.S); first(βa) = first(ε$) = first($) = $
	S → .CC, $	goto I_2(C)	close(.C); first(C$) = first(C) = c/d
	C → .cC, c/d	goto I_3(c)	from close(.C)
	C → .d, cd	goto I_4(d)	from close(.C)
I_1:	S' → S., $	//	
I_2:	S → C.C, $	goto I_5(C)	close(.C); first($) = $
	C → .cC, $	goto I_3(c)	
	C → .d, $	goto I_4(d)	
I_3:	C → c.C, c/d/$	goto I_6(C)	close(.C);
	C → .cC, c/d/$	goto I_3(c)	
	C → .d c/d/$	goto I_4(d)	
I_4:	C → d., c/d/$	//	
I_5:	S → CC., $	//	
I_6:	C → cC., c/d/$	//	

The LALR parsing table is:

State	action			goto	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5			r1		
6	r2	r2	r2		

Given the following augmented grammar:

	S' → S
1	S → L = R
2	S → R
3	L → * R
4	L → id
5	R → L

The set of LALR items are as follows:

I_0:	S' → .S, $	goto I_1(S)	close(.S); first(βa) = first(ε$) = first($) = $
	S → .L = R, $	goto I_2(L)	close(.L); first(βa) = first(=R$) = {=}
	S → .R, $	goto I_3(R)	close(.R); first(βa) = first(ε$) = first($) = $
	L → .* R, =$	goto I_4(*)	
	L → .id, =$	goto I_5(id)	
	R → .L, $	goto I_2(L)	close(.L); first($) = $
I_1:	S' → S., $	//	
I_2:	S → L. = R, $	goto I_6(=)	
	R → L., $	//	
I_3:	S → R., $	//	
I_4:	L → * .R, =$	goto I_7(R)	close(.R); first(=$) = {=$}
	R → .L, =$	goto I_8(L)	close(.L); first(=$) = {=$}
	L → .* R, =$	goto I_4(*)	
	L → .id, =$	goto I_5(id)	
I_5:	L → id., =$	//	
I_6:	S → L = .R, $	goto I_9(R)	close(.R)
	R → .L, $	goto I_8(L)	close(.L); first($) = $
	L → .* R, $	goto I_4(*)	
	L → .id, $	goto I_5(id)	
I_7:	L → * R., =$	//	
I_8:	R → L.., =$	//	
I_9:	S → L = R., $	//	

Given the following augmented grammar:

	E' → E
1	E → E + E
2	E → E * E
3	E → (E)
4	E → id

The set of LALR items are as follows:

I_0:	E' → .E, $	goto I_1(E)	close(.E); first(βa) = first(ε$) = first($) = $
	E → .E + E, $+*	goto I_1(E)	close(.E); first(+E$) = {+}
	E → .E * E, $+*	goto I_1(E)	close(.E); first(*E$) = {*}
	E → .(E), $+*	goto I_2(()	
	E → .id, $+*	goto I_3(id)	
I_1:	E' → E., $	//	

	E → E .+ E, $+*	goto I_4(+)		
	E → E .* E, $+*	goto I_5(*)		
I_2:	E → (.E), $+*)	goto I_6(E)	close(.E); first()) =)	
	E → .E + E,)+*	goto I_6(E)	close(.E); first(+) = +	
	E → .E * E,)+*	goto I_6(E)	close(.E); first(*) = *	
	E → .(E),)+*	goto I_2(()		
	E → .id,)+*	goto I_3(id)		
I_3:	E → id., $+*)	//		
I_4:	E → E + .E, $+*)	goto I_7(E)	close(.E)	The) added from I_6
	E → .E + E, $+*)	goto I_7(E)	close(.E); first(+) = +	
	E → .E * E, $+*)	goto I_7(E)	close(.E); first(*) = *	
	E → .(E), $+*)	goto I_2(()		
	E → .id, $+*)	goto I_3(id)		
I_5:	E → E * .E., $+*)	goto I_8(E)	close(.E); first($+*) = $+*	
	E → .E + E, $+*)	goto I_8(E)	close(.E);	
	E → .E * E, $+*)	goto I_8(E)	close(.E);	
	E → .(E), $+*)	goto I_2(()		
	E → .id, $+*)	goto I_3(id)		
I_6:	E → (E.), $+*)	goto I_9())		
	E → E .+ E,)+*	goto I_4(+)		
	E → E .* E,)+*	goto I_5(*)		
I_7:	E → E + E., $+*)	//	The) added from I_4	
	E → E .+ E, $+*)	goto I_4(+)		
	E → E .* E, $+*)	goto I_5(*)		
I_8:	E → E * E., $+*)	//	The) added from I_5	
	E → E .+ E, $+*)	goto I_4(+)		
	E → E .* E, $+*)	goto I_5(*)		
I_9:	E → (E)., $+*)	//		

The LALR parsing table is:

State	action						goto
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		$r1_{s4}$	$s5_{r1}$		r1	r1	
8		$r2_{s4}$	$r2_{s5}$		r2	r2	
9		r3	r3		r3	r3	

Parse the string "id + id * id"

stack	id + id * id $	action	comment
0	↑	s3	lexical value **id** is pushed on the semantic stack. current state **3** is pushed on the syntactic stack.
0 ~~id3~~	↑	r4: E → id	translation record **E** is pushed on the semantic stack. current state **1** is pushed on the syntactic stack.
0 E1	↑	s4	
0 E1 +4	↑	s3	
0 E1 +4 ~~id3~~	↑	r4: E → id	
0 E1 +4 E7	↑	s5	
0 E1 +4 E7 *5	↑	s3	
0 E1 +4 E7 *5 ~~id3~~	↑	r4: E → id	
0 E1 +4 ~~E7 *5 E8~~	↑	r2: E → E*E	
0 ~~E1 +4 E7~~	↑	r1: E → E+E	the actions are in the reverse order of a
0 E1	↑	accept ✓	rightmost derivation

In the above grammar, we have a shift/reduce conflict.
Use precedence and associativity to resolve parsing action conflicts:

I_7 is E → E + E. ⇒ we have seen E + E, therefore,
if lookahead is a **+**, we want **r1** because + is left associative.
if lookahead is a ***, we want **s5** because * has higher precedence.

I_8 is E → E * E. ⇒ we have seen E * E, therefore,
if lookahead is a **+**, we want **r2** because * has higher precedence.
if lookahead is a ***, we want **r2** because * is left associative.

- for a shift/reduce conflict for the "if c then s" statement, we should resolve in favor of a shift, i.e. we should shift "else" because it is associated with the previous "then".
- for special cases, we introduce an additional production and resolve the conflict by reducing the special-case production.

- Conflict resolution in YACC:
  - **reduce/reduce** conflicts are resolved in favor of the earlier rule in the input sequence.
  - the default is to resolve **shift/reduce** conflicts in favor of the shift. If, however, both rules and the input (lookahead) token have precedence, it resolves in favor of the action of higher precedence, breaking ties on the basis of associativity of the completed item: *left* implies reduce, *right* implies shift, *unassoc* implies error.

Notes on LR-Parsing

- Entries in the parsing (transition) table are of three kinds:
  1. *reduce* - indices of grammar rules.
  2. *shift* - states.
  3. *goto* - states.
- Entries on the stack consist of:
  1. *syntatic part* - state.
  2. *semantic part* - contains either of:
     - *translation record* (left side of a production) - if the corresponding node is an interior node. This is pushed on the stack during a **reduce** action.
     - *lexical value* (yylval) - if the corresponding node is a leaf node. This is pushed on the stack during a **shift** action.
  - on a **shift** action, the **lexical value** is pushed on the **semantic stack**.
    the **current state** is pushed on the **syntatic stack**.
  - on a **reduce** action, the **translation record** is pushed on the **semantic stack**.
    the **current state** is pushed on the **syntatic stack**.
- Comparison of the LR tables:

	LR(0)	LALR	LR(1)
table size	small	same as LR(0) merge states with common core from LR(1). See note below.	large
conflicts	more	little more than LR(1)	less

  - Merging of states with common cores (first component=translation record) can never produce a shift/reduce conflict that was not present in one of the orginal states, because shift actions depend only on the core, not the lookahead. (p237)
  - However, it is possible that a merger will produce a reduce/reduce conflict.

- To compute FOLLOW(B) for all nonterminals B, apply the following rules until nothing can be added to any FOLLOW set:
  1. Place \$ in FOLLOW(S), where S is the start symbol.

     Look at the right side of each production for the nonterminal B:
  2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except for $\in$ is placed in FOLLOW(B).
  3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains $\in$ (i.e., $\beta \overset{*}{\Rightarrow} \in$), then everything in FOLLOW(A) is in FOLLOW(B) (i.e., FOLLOW(B) $\leftarrow$ FOLLOW(A)).