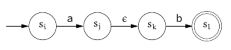
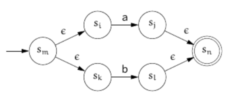
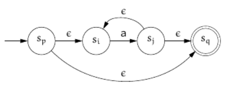
0.1 Scanner



NFA for *ab*



NFA for *a* | *b*



NFA for *a**

The following table sketches the steps that the subset construction algorithm follows.

Set Name	DFA States	NFA States	ϵ-**closure**(**Delta**$(q, \circ)$)		
			a	b	c
q_0	d_0	n_0	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	$\emptyset$	$\emptyset$
q_1	d_1	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	$\emptyset$	$\{n_5, n_8, n_9, n_3, n_4, n_6\}$	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$
q_2	d_2	$\{n_5, n_8, n_9, n_3, n_4, n_6\}$	$\emptyset$	q_2	q_3
q_3	d_3	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$	$\emptyset$	q_2	q_3

The algorithm takes the following steps

1. Initialization separates accepting from nonaccepting states: $\{\{s_3, s_5\}, \{s_0, s_1, s_2, s_4\}\}$
2. Step 1 examines $\{s_3, s_5\}$: neither state has an exiting transition, *i.e.*, no split occurs
3. Step 2 examines $\{s_0, s_1, s_2, s_4\}$: on charachter e, it splits $\{s_2, s_4\}$ out of the set
4. Step 3 examines $\{s_0, s_1\}$: on charachter f, it splits $\{s_1\}$ out of the set
5. Step 4 makes a final pass over the current partition $\{\{s_3, s_5\}, \{s_0\}, \{s_1\}, \{s_2, s_4\}\}$: no more splits occur and, therefore, the fix-point is reached

Note We assume that the various loops in the algorithm iterate over the sets of P and over the characters in $\Sigma = \{e, f, i\}$ in order.

The following summarizes the significant steps that occur in minimizing this DFA.

Step	Current	Examines		
	Partition	Set	Char	Action
0	$\{\{s_3, s_5\}, \{s_0, s_1, s_2, s_4\}\}$	—	—	—
1	$\{\{s_3, s_5\}, \{s_0, s_1, s_2, s_4\}\}$	$\{s_3, s_5\}$	*all*	*none*
2	$\{\{s_3, s_5\}, \{s_0, s_1, s_2, s_4\}\}$	$\{s_0, s_1, s_2, s_4\}$	e	*split* $\{s_2, s_4\}$
3	$\{\{s_3, s_5\}, \{s_0, s_1\}, \{s_2, s_4\}\}$	$\{s_0, s_1\}$	f	*split* $\{s_1\}$
4	$\{\{s_3, s_5\}, \{s_0\}, \{s_1\}, \{s_2, s_4\}\}$	*all*	*all*	*none*

0.2 LL(1) Grammar

Formally, a **Context-Free Grammar** G is a quadruple (T, NT, S, P) where

T	is the set of terminal symbols, or words, in the language $L(G)$.
NT	is the set of nonterminal symbols that appear in the productions of G.
S	is a nonterminal designated as the **goal symbol** or **start symbol** of the grammar. S represents the set of sentences in $L(G)$.
P	is the set of productions or rewrite rules in G. Each rule in P has the form $NT \rightarrow (T \cup NT)^+$, *i.e.*, it replaces a **single nonterminal** with a string of one or more grammar symbols.

$$\begin{array}{rcl} A & \rightarrow & A\alpha \\ & | & \beta \end{array} \qquad \begin{array}{rcl} A & \rightarrow & \beta A' \\ A' & \rightarrow & \alpha A' \\ & | & \epsilon \end{array}$$

The translation from (direct) left recursion to right recursion is mechanical
- introduce a new nonterminal A' and transfer the recursion onto A'
- add a rule $A' \rightarrow \epsilon$, where ϵ represents the empty string

So far, we have only tackled **direct** left recursion. There can also be **indirect** left recusion, which is caused by chains of "transitive" productions.

$$\alpha \rightarrow \beta, \beta \rightarrow \gamma, \text{and } \gamma \rightarrow \alpha\delta \implies \alpha \rightarrow^+ \alpha\delta$$

Indirect left recursion can be obscured by a long chain of productions. Therefore, we need a **more systematic approach** to convert indirect left recursion into right recursion.

We can eliminate all left recursion from a grammar using **two** simple techniques
- forward substitution to convert indirect left recursion into direct left recursion
- rewriting direct left recursion as right recursion

```
impose an arbitrary order on nonterminals $A_1, A_2, \dots, A_n$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $i-1$ do
        if there is a production $A_i \rightarrow A_j\gamma$ and $A_j \rightarrow \delta_1|\delta_2|\dots|\delta_n$ then
            replace $A_i \rightarrow A_j\gamma$ with a set of productions $A_i \rightarrow \delta_1\gamma|\delta_2\gamma|\dots|\delta_n\gamma$
    rewrite the productions to eliminate any direct left recursion on $A_i$
```

Note This algorithm assumes that the original grammar has no cycles ($A \rightarrow^+ A$) and no ϵ-productions.

FIRST

For a grammar symbol α, FIRST(α) is the set of terminals that can appear at the start of a sentence derived from α.

The domain of FIRST is the set of grammar symbols, $T \cup NT \cup \{\epsilon, \texttt{eof}\}$ and its range is $T \cup \{\epsilon, \texttt{eof}\}$.

$\alpha \in T \cup \{\epsilon, \texttt{eof}\}$	FIRST(α) has exactly one member α
$A \in NT$	FIRST(A) contains all terminal symbols that can appear as the leading symbol in any sentential form derived from A

Dealing with ϵ-productions
- parser should apply the ϵ-production if the lookahead symbol is **not a member** of the FIRST set of any other alternative
- to differentiate between **legal input** and **syntax errors**, it needs to know which words can appear as the leading symbol after a valid application of an ϵ-production

FOLLOW

For a nonterminal A, FOLLOW(A) contains the set of words that can occur immediately after A in a sentence.

Backtrack-Free Grammar

For a production $A \rightarrow \beta$, we define its augmented FIRST set, FIRST$^+$.

$$\text{FIRST}^+(A \rightarrow \beta) = \begin{cases} \text{FIRST}(\beta) & \epsilon \notin \text{FIRST}(\beta) \\ \text{FIRST}(\beta) \cup \text{FOLLOW}(A) & \textit{otherwise} \end{cases}$$

A grammar is **backtrack-free** if the following property holds for any nonterminal A with multiple right-hand sides, *i.e.*, $A \rightarrow \beta_1|\beta_2|\dots|\beta_n$.

$$\forall\, 1 \le i, j \le n, i \ne j : \text{FIRST}^+(A \rightarrow \beta_i) \cap \text{FIRST}^+(A \rightarrow \beta_j) = \emptyset$$

We can left factor any set of rules that has alternate right-hand sides with a common prefix.

$$A \rightarrow \alpha\beta_1|\alpha\beta_1|\dots|\alpha\beta_n|\gamma_1|\gamma_2|\dots|\gamma_m$$

The transformation introduces a new nonterminal B to represent the alternate suffixes for α and rewrites the original productions according to the following pattern.

$$\begin{array}{rcl} A & \rightarrow & \alpha B|\gamma_1|\gamma_2|\dots|\gamma_m \\ B & \rightarrow & \beta_1|\beta_1|\dots|\beta_n \end{array}$$

To left factor a complete grammar, we must inspect each nonterminal, discover common prefixes, and apply the transformation in a **systematic** way.

0.3 Code Shape

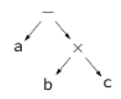
Arithmetic Operators

If the expression is represented in a tree-like IR, this process fits into a **postorder** tree walk.

`base`
- returns the name of a register holding the base address for an identifier
- if needed, it emits code to get that address into a register

`offset`
- returns the name of a register holding the identifier's offset
- offset is relative to the address returned by `base`

```
procedure expr($n$)
    if $n \in \{+, -, \times, \div\}$ then
        $t_1 \leftarrow$ expr($n$.left)
        $t_2 \leftarrow$ expr($n$.right)
        $r \leftarrow$ NextRegister()
        emit($n$, $t_1$, $t_2$, $r$)
    else if $n$ = ident then
        $t_1 \leftarrow$ base($n$)
        $t_2 \leftarrow$ offset($n$)
        $r \leftarrow$ NextRegister()
        emit(loadAO, $t_1$, $t_2$, $r$)
    else if $n$ = num then
        $r \leftarrow$ NextRegister()
        emit(loadI, $n$, _, $r$)
    return $r$
```



```
loadI   @a          ⇒  r1
loadAO  r_arp, r1   ⇒  r2
loadI   @b          ⇒  r3
loadAO  r_arp, r3   ⇒  r4
loadI   @c          ⇒  r5
loadAO  r_arp, r5   ⇒  r6
mult    r4, r6      ⇒  r7
add     r2, r7      ⇒  r8
```

```
loadI   @A0          ⇒  r_@A0    ...adjusted base address of A
multI   ri, len2     ⇒  r1       ...i ×len2
add     r1, rj       ⇒  r2       ...+ j
multI   r2, 4        ⇒  r3       ...× element length (4)
loadAO  r_@A0, r3    ⇒  ra       ...value of A[i,j]
```

```
if (a < b)
{
    statement1;
}
else
{
    statement2;
}
```

```
     comp    ra, rb  ⇒  cc1
     cbr_LT  cc1     →  L1, L2
L1:  code for statement1
     jumpI           →  L3
L2:  code for statement2
     jumpI           →  L3
L3:  nop
```

For Loops

To map a `for` loop into code, the compiler follows the general schema from before.

```
for (i = 1; i <= 100; i++) {
    loop body
}
next statement
```

```
     loadI   1        ⇒  ri          ...Step 1
     loadI   100      ⇒  r1          ...Step 2
     cmp_GT  ri, r1   ⇒  r2
     cbr     r2       →  L1, L2
L1:  loop body                       ...Step 3
     addI    ri, 1    ⇒  ri          ...Step 4
     cmp_LE  ri, r1   ⇒  r3
     cbr     r3       →  L1, L2
L2:  next statement                  ...Step 5
```

The compiler can also shape the loop to have only **one copy** of the test. In this form, Step 4 evaluates e_3 and then jumps to Step 2, *i.e.*, replace `cmp_LE` and `cbr` with `jumpI`.

While Loops

A `while` loop can also be implemented with the loop schema. Since it has no initialization, the code is even more compact.

```
while (x < y) {
    loop body
}
next statement
```

```
     cmp_LT  rx, ry  ⇒  r1          ...Step 2
     cbr     r1      →  L1, L2
L1:  loop body                      ...Step 3
     cmp_LT  rx, ry  ⇒  r2          ...Step 4
     cbr     r2      →  L1, L2
L2:  next statement                 ...Step 5
```

Replicating the test in Step 4 creates the possibility of a loop with a **single basic block**.

Until Loops

An `until` loop iterates as long as the controlling expression is `false`.

It checks the controlling expression **after** each iteration. Thus, it always enters the loop and performs at least one iteration and produces a particularly simple loop structure.

```
{
    loop body
} until (x < y)
next statement
```

```
L1:  loop body                      ...Step 3
     cmp_LT  rx, ry  ⇒  r1          ...Step 4
     cbr     r1      →  L1, L2
L2:  next statement                 ...Step 5
```

Note The do loop known from C, C++, and Java is similar to the `until` loop with the difference that it iterates as long as the controlling expression is `true`.

0.4 Instruction selection

	Production	Cost	Code Template
1	Goal → Assign	0	–
2	Assign → ←(Reg$_1$, Reg$_2$)	1	store r$_2$ ⇒ r$_1$
3	Assign → ←(+(Reg$_1$, Reg$_2$), Reg$_3$)	1	storeAO r$_3$ ⇒ r$_1$, r$_2$
4	Assign → ←(+(Reg$_1$, Num$_2$), Reg$_3$)	1	storeAI r$_3$ ⇒ r$_1$, n$_2$
5	Assign → ←(+(Num$_1$, Reg$_2$), Reg$_3$)	1	storeAI r$_3$ ⇒ r$_2$, n$_1$
6	Reg → Lab$_1$	1	load l$_1$ ⇒ r$_{new}$
7	Reg → Val$_1$	0	–
8	Reg → Num$_1$	1	load n$_1$ ⇒ r$_{new}$
9	Reg → ◆(Reg$_1$)	1	load r$_1$ ⇒ r$_{new}$
10	Reg → ◆(+(Reg$_1$, Reg$_2$))	1	loadAO r$_1$, r$_2$ ⇒ r$_{new}$
11	Reg → ◆(+(Reg$_1$, Num$_2$))	1	loadAI r$_1$, n$_2$ ⇒ r$_{new}$
12	Reg → ◆(+(Num$_1$, Reg$_2$))	1	loadAI r$_2$, n$_1$ ⇒ r$_{new}$
13	Reg → ◆(+(Reg$_1$, Lab$_2$))	1	loadAI r$_1$, l$_2$ ⇒ r$_{new}$
14	Reg → ◆(+(Lab$_1$, Reg$_2$))	1	loadAI r$_2$, l$_1$ ⇒ r$_{new}$
15	Reg → +(Reg$_1$, Reg$_2$)	1	add r$_1$, r$_2$ ⇒ r$_{new}$
16	Reg → +(Reg$_1$, Num$_2$)	1	addI r$_1$, n$_2$ ⇒ r$_{new}$
17	Reg → +(Num$_1$, Reg$_2$)	1	addI r$_2$, n$_1$ ⇒ r$_{new}$
18	Reg → +(Reg$_1$, Lab$_2$)	1	addI r$_1$, l$_2$ ⇒ r$_{new}$
19	Reg → +(Lab$_1$, Reg$_2$)	1	addI r$_2$, l$_1$ ⇒ r$_{new}$
20	Reg → −(Reg$_1$, Reg$_2$)	1	sub r$_1$, r$_2$ ⇒ r$_{new}$
21	Reg → −(Reg$_1$, Num$_2$)	1	subI r$_1$, n$_2$ ⇒ r$_{new}$
22	Reg → −(Num$_1$, Reg$_2$)	1	rsubI r$_2$, n$_1$ ⇒ r$_{new}$
23	Reg → ×(Reg$_1$, Reg$_2$)	1	mult r$_1$, r$_2$ ⇒ r$_{new}$
24	Reg → ×(Reg$_1$, Num$_2$)	1	multI r$_1$, n$_2$ ⇒ r$_{new}$
25	Reg → ×(Num$_1$, Reg$_2$)	1	multI r$_2$, n$_1$ ⇒ r$_{new}$

0.5 Instruction scheduling

0.5.1 Local

```
Cycle ← 1
Ready ← leaves of D
Active ← ∅
while Ready ∪ Active ≠ ∅ do
    foreach op ∈ Active do
        if S(op) + delay(op) < Cycle then
            remove op from Active
            foreach successor s of op in D do
                if s is ready then
                    Ready ← Ready ∪ {s}
    if Ready ≠ ∅ then
        remove an op from Ready
        S(op) ← Cycle
        Active ← Active ∪ {op}
    Cycle ← Cycle + 1
```

The algorithm maintains a simulation clock `Cycle` to track time.

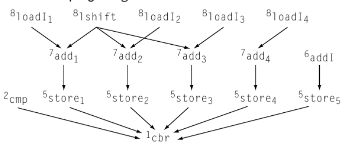
`Ready` holds all the operations that can execute in the current cycle.

`Active` holds all operations that were issued in an earlier cycle but have not yet finished.

At each time step, it accounts for any operations completed in the previous cycle, it schedules an operation for the current cycle, and it increments `Cycle`.

Forward versus Backward List Scheduling

Example Below the dependence graph for a basic block found in the SPEC 95 benchmark program go is shown.



Opcode	Latency
loadI	1
lshift	1
add	2
addI	1
cmp	1
store	4

Note The compiler added dependences from the `store` operations to the block-ending branch to ensure that the memory operations complete before the next block begins.

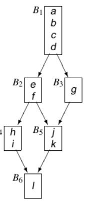
	Integer	Integer	Memory
1	loadI$_1$	lshift	–
2	loadI$_2$	loadI$_3$	–
3	loadI$_4$	add$_1$	–
4	add$_2$	add$_3$	–
5	add$_4$	addI	store$_1$
6	cmp	–	store$_2$
7	–	–	store$_3$
8	–	–	store$_4$
9	–	–	store$_5$
10	–	–	–
11	–	–	–
12	–	–	–
13	cbr	–	–

	Integer	Integer	Memory
1	loadI$_4$	–	–
2	addI	lshift	–
3	add$_4$	loadI$_3$	–
4	add$_3$	loadI$_2$	store$_5$
5	add$_2$	loadI$_1$	store$_4$
6	add$_1$	–	store$_3$
7	–	–	store$_2$
8	–	–	store$_1$
9	–	–	–
10	–	–	–
11	cmp	–	–
12	cbr	–	–
13	–	–	–

0.5.2 Global

An **extended basic block** (EBB) consists of a set of blocks $B_1, B_2, \dots, B_n$ in which B_1 has multiple predecessors and every other block B_i has exactly one predecessor, some B_j in the EBB.

Example The CFG on the right has one large EBB, $\{B_1, B_2, B_3, B_4\}$, and two trivial EBBs, $\{B_5\}$ and $\{B_6\}$.

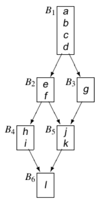


To obtain a **larger context** for list scheduling, the compiler can treat paths in an EBB, such $\{B_1, B_2, B_4\}$, as if they were single blocks.

When scheduling paths, the compiler needs to accounts for

- shared path prefixes, *e.g.*, B_1, which occurs in $\{B_1, B_2, B_4\}$ and $\{B_1, B_3\}$
- premature exits, *e.g.*, $B_1 \to B_3$ and $B_2 \to B_5$

To see how **shared prefixes** and **premature exits** complicate list scheduling, consider the possibilities for code motion in $\{B_1, B_2, B_4\}$.



Move an operation **forward**, *e.g.*, c from B_1 to B_2

- might speed execution along the path $\{B_1, B_2, B_4\}$
- changes the computation performed along the path $\{B_1, B_3\}$
- to fix this problem, the scheduler must insert a copy of c into B_3

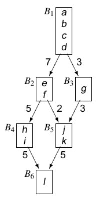
Move an operation **backward**, *e.g.*, f from B_2 to B_1

- might speed execution along the path $\{B_1, B_2, B_4\}$
- inserts a computation of f into the path $\{B_1, B_3\}$, which might produce incorrect code along this path
- the scheduler must rewrite the code to undo that effect in B_3

Trace scheduling constructs maximal-length acyclic paths through the CFG and applies the list-scheduling algorithm to those paths, or **traces**.

To build a trace, a simple **greedy approach** can be used that stops when it either runs out of possible edges or encounters a loop-closing branch.

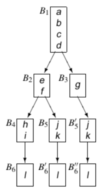
Example The trace for the example on the right is $\{B_1, B_2, B_4, B_6\}$.



With respect to EBB scheduling, one **additional** opportunity for compensation code occurs: a trace may have interim entry points, *i.e.*, blocks in mid-trace that have **multiple** predecessors.

Note EBB scheduling can be considered a degenerate case of trace scheduling in which interim entries to the trace are prohibited.

Join points in the CFG **limit the opportunities** for either EBB scheduling or trace scheduling.



The compiler can **clone blocks** to create longer join-free paths.

- for EBB scheduling, it increases the size of the EBB and the length of some of the paths through the EBB
- for trace scheduling, it avoids the complications caused by interim entry points in the trace

After cloning, the entire graph forms **one single** EBB.

The compiler can combine blocks that are linked by an edge where the source has no other successors and the sink has no other predecessors.

0.6 Register Allocation

0.6.1 Local Top-Down

If the block uses **fewer than** k virtual registers, allocation is trivial and the compiler can simply assign each `vr` to its own physical register.

If the block uses **more than** k virtual registers, the compiler applies the following simple algorithm.

1. compute a priority for each virtual register
2. sort the virtual registers into priority order
3. assign the first $k - \mathcal{F}$ virtual registers to physical registers in priority order
4. rewrite the code
    - replace virtual register names with physical register names
    - replace virtual register names with no allocated physical register with code that uses one of the reserved register and performs the appropriate load or store

Example Assume a target machine with four physical registers r_1, r_2, r_3, and r_4 as well as two spill registers f_1 and f_2. Both r_{arp} and r_i are **live** upon entry into the block.

```
loadAI   rarp, 12  ⇒  ra
loadAI   rarp, 16  ⇒  rb
add      ri, ra    ⇒  rc
sub      rb, ri    ⇒  rd
mult     rc, rd    ⇒  re
multI    rb, 2     ⇒  rf
add      re, rf    ⇒  rg
storeAI  rg        ⇒  rarp, 8
```

r_{arp}	r_a	r_b	r_c	r_d	r_e	r_f	r_g	r_i
3	2	3	2	2	2	2	2	2

```
loadAI   rarp, 12  ⇒  r1
loadAI   rarp, 16  ⇒  r2
add      r4, r1    ⇒  r3
sub      r2, r4    ⇒  f1
storeAI  f1        ⇒  rarp, 20
mult     r3, f1    ⇒  f1
storeAI  f1        ⇒  rarp, 24
multI    r2, 2     ⇒  f1
storeAI  f1        ⇒  rarp, 28
loadAI   rarp, 24  ⇒  f2
add      f2, f1    ⇒  f1
storeAI  f1        ⇒  rarp, 8
```

0.6.2 Local Bottom-Up

The primary weakness of top-down allocation is that it dedicates a physical register to one virtual register for the **entire** basic block.

The key idea behind the **bottom-up local allocator** is to focus on the details of how values are defined and used on an operation-by-operation basis.

1. start with an empty register set
2. allocate registers on demand
3. if no register is available, free one
    - keep values that will be "used soon" in registers
    - spill register whose next use is farthest in the future

```
foreach op_i vr_i1, vr_i2 ⇒ vr_i3, in order i = 1...N do
    r_x ← Ensure(vr_i1, class(vr_i1))
    r_y ← Ensure(vr_i2, class(vr_i2))
    if vr_i1 is not needed after op_i then
        Free(r_x, class(r_x))
    if vr_i2 is not needed after op_i then
        Free(r_y, class(r_y))
    r_z ← Allocate(vr_i3, class(vr_i3))
    rewrite op_i as op_i r_x, r_y ⇒ r_z
    if vr_i1 is needed after op_i then
        class.Next[r_x] ← Dist(vr_i1)
    if vr_i2 is needed after op_i then
        class.Next[r_y] ← Dist(vr_i2)
    class.Next[r_z] ← Dist(vr_i3)
```

The bottom-up allocator iterates over the operations in the block, making allocation decisions on demand.

By considering vr_{i_1} and vr_{i_2} in order, the allocator avoids using two physical registers for an operation with a repeated operand.

Trying to free r_x and r_y before allocating r_z avoids spilling a register to hold an operation's result when the operation actually frees a register.

The function `Dist(vr)` returns the index in the block of the next reference to `vr`.

```
procedure Ensure(vr, class)
    if vr is already in class then
        result ← vr's physical register
    else
        result ← Allocate(vr, class)
        emit code to move vr into result
    return result
```

`Ensure` takes two arguments: a virtual register `vr` holding the desired value, and a representation for the appropriate register class `class`.

- if `vr` already occupies a physical register, `Ensure`'s job is done
- otherwise, it allocates a physical register for `vr` and emits code to move `vr`'s value into that physical register

In either case, it returns the physical register.

```
procedure Allocate(vr, class)
    if class.StackTop ≥ 0 then
        i ← pop(class)
    else
        i ← j that maximizes class.Next[j]
        store contents of j
    class.Name[i] ← vr
    class.Next[i] ← -1
    class.Free[i] ← false
    return i
```

```
struct Class{
  int Size;
  int Name[Size];
  int Next[Size];
  int Free[Size];
  int Stack[Size];
  int StackTop;
}
```

`Allocate` returns a physical register from the free list of class, if one exists.

Otherwise, it selects the value stored in class that is used farthest in the future, spills it, and reallocates the corresponding physical register to `vr`.

```
procedure Free(vr, class)
    i ← push(class)
    class.Name[i] ← -1
    class.Next[i] ← ∞
    class.Free[i] ← true
```

`Free` simply pushes the freed register onto the stack and resets its fields to their initial values.

Example Assume a target machine with four physical registers r_1, r_2, r_3, and r_4 as well as two spill registers f_1 and f_2. Both r_{arp} and r_i are **live** upon entry into the block.

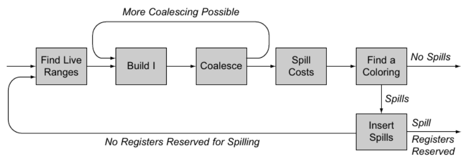
```
loadAI   r_arp, 12  ⇒  r_a                loadAI   r_arp, 12  ⇒  r_1
loadAI   r_arp, 16  ⇒  r_b                loadAI   r_arp, 16  ⇒  r_2
add      r_i, r_a   ⇒  r_c                add      r_4, r_1   ⇒  r_1
sub      r_b, r_i   ⇒  r_d                sub      r_2, r_4   ⇒  r_4
mult     r_c, r_d   ⇒  r_e                mult     r_1, r_4   ⇒  r_1
multI    r_b, 2     ⇒  r_f                multI    r_2, 2     ⇒  r_2
add      r_e, r_f   ⇒  r_g                add      r_1, r_2   ⇒  r_1
storeAI  r_g        ⇒  r_arp, 8           storeAI  r_1        ⇒  r_arp, 8
```

Note Due to freeing operand registers before allocating result registers (*cf.* Lines 4–7 on Slide 516), the bottom-up allocator avoids all spills.

0.6.3 Global

Comparing Top-Down and Bottom-Up Global Allocators

Both the top-down and the bottom-up coloring allocators have the **same basic structure**.



Coalescing Copies to Reduce Degree

Example The original code appears on the left, with lines to the right that indicate the regions where each of the relevant values, LR_a, LR_b, and LR_b are live.

```
add LR_t , LR_u ⇒ LR_a  ┬ a                 add LR_t , LR_u ⇒ LR_ab  ┬ ab
     ...                │                        ...                 │
i2i LR_a        ⇒ LR_b  │ ┬ b                                        │
i2i LR_a        ⇒ LR_c  ┴ │ ┬ c               i2i LR_ab      ⇒ LR_c   │ ┬ c
     ...                  │ │                      ...                 │ │
add LR_b , LR_w ⇒ LR_x    ┴ │               add LR_ab , LR_w ⇒ LR_x  ┴ │
add LR_c , LR_y ⇒ LR_z      ┴               add LR_c , LR_y ⇒ LR_z     ┴
```

On the right, the result of coalescing LR_a and LR_b to produce LR_{ab} is shown. Since LR_c is defined by a copy from LR_{ab}, they do not interfere.

At any point in the code, only live values need registers. LIVEOUT sets encode precisely this knowledge.

LIVEOUT

For each block b, the set LIVEOUT(b) contains all the variables that are live on exit from b.

Any value in LIVEOUT(b) must be **stored** to its assigned location in memory after its last definition in b to ensure that the correct value is available in a subsequent block.

The fundamental effect that a global register allocator must model is the **competition** among values for space in the processor's register set.

Interference

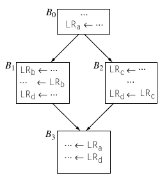
Two live ranges, LR_i and LR_j interfere if one is live at the definition of the other and they have different values.

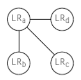
To model the allocation problem, the compiler can build an **interference graph** $I = (N, E)$
- nodes in N represent individual live ranges
- edges in E represent interferences between live ranges

An undirected edge $(n_i, n_j) \in I$ exists if and only if the corresponding live ranges LR_i and LR_j interfere.

Interferences and the Interference Graph

Example On the left, the code (rewritten to use live ranges) from the example on Slide 529 is shown. The corresponding interference graph is shown on the right.





The interference graph shows that LR_a interferes with **each** of the other live ranges.

The rest of the live ranges, however, **do not** interfere with each other.

Once the allocator has built global live ranges and annotated each block in the code with its LIVEOUT set, it can construct the interference graph in a linear pass over each block.

```
foreach $LR_i$ do
    create a node $n_i \in N$
foreach basic block $b$ do
    LIVENOW ← LIVEOUT($b$)
    foreach operation $op_n, op_{n-1} \ldots op_1$ in $b$ with form $op_i$ $LR_a$, $LR_b \Rightarrow LR_c$ do
        foreach $LR_i \in$ LIVENOW do
            $E \leftarrow E \cup \{(LR_c, LR_i)\}$
        LIVENOW ← LIVENOW − $LR_c$
        LIVENOW ← LIVENOW ∪ $\{LR_a\} \cup \{LR_b\}$
```

Top-Down

Top-down allocators try to color live ranges in an order given by a **ranking function**.

The **priority-based**, **top-down allocators** assign each node a rank that is the estimated runtime savings that accrue from keeping that live range in a register.
1. consider live ranges in rank order and attempt to assign a color to each of them
2. if no color is available, invoke the spilling or splitting mechanism

To improve the process, the allocator can partition the live ranges into two sets: **constrained** live ranges (k or more neighbors) and **unconstrained** live ranges.
- first, constrained live ranges are colored in rank order
- then, unconstrained live ranges are colored in any order

Because an unconstrained live range has fewer than k neighbors, the allocator can always find a color for it: no assignment of colors to its neighbors can use all k colors.

To spill LR_i, the allocator inserts a store after **every definition** of LR_i and a load before **each use** of LR_i.

If memory operations need registers, the allocator can **reserve registers** to handle them.
- the number of registers needed is determined by the instruction set architecture
- reserving these registers simplifies spilling

An alternative to reserving registers is to **look for free colors** at each definition and use.
- if no color is available, retroactively spill a previously colored live range
- this scheme has the potential to spill previously colored live ranges recursively

This feature has led most implementors of top-down, priority-based allocators to reserve spill registers instead.

Bottom-Up

The algorithm computes the coloring order for a graph $I = (N, E)$ as follows.

```
initialize stack to empty
while $N \neq \emptyset$ do
    if $\exists n \in N : deg(n) < k$ then
        node ← $n$
    else
        node ← $n$ picked from $N$
    remove node and its edges from $I$
    push node onto stack
```

It uses two **distinct mechanisms** to select the node to remove next.
- the first clause takes a node that is **unconstrained** in the graph from which it is removed
- the second clause, invoked only when every remaining node is **constrained**, picks a node using some external criteria

The loop halts when the graph is empty. At that point, the stack contains all the nodes in order of removal.

To color the graph, the allocator rebuilds the interference graph in the order represented by the stack, *i.e.*, in **reverse order** of removing them from the graph.

```
while stack ≠ ∅ do
    pop node from stack
    insert node and its edges into $I$
    color node
```

The allocator tallies the colors of node's neighbors in the current approximation to I and assigns node an unused color.

If no color remains, node is left uncolored.

When the stack is empty, I has been rebuilt.
- if every node has a color, the allocator declares success and rewrites the code
- if nodes remain uncolored, the allocator spills or splits the corresponding live range

At this point, the classic bottom-up allocators rewrite the code to reflect the spills and splits and **repeat** the entire process.