• Optimizers usually make the code better in terms of
  - less number of instructions
  - faster instructions
  - less space
  - fewer executions (eg. loops)
  - less burden in latter stages of language processing or execution

There is room for optimization at the level of SL, IL or TL.

SC and IC optimizations are front-end tasks; TC optimization is typically architecture-dependent, hence done by the back-end.
• SC optimization: better left to the programmer of SL.
  - If you think that removing recursion from SC is good, imagine what you’d need to know to remove all recursion from SC.

Language processors tend to optimize things that they have control over, and these do not include programming talent (yet).

• IC optimization: better be language-internal and architecture independent.
  - improve loops
  - minimize compiler’s dirty laundry: names and labels invisible to the programmer (temporaries and instruction labels)
- move code around (or replace by simpler code)

- TC optimization: architecture-specific
  - register allocation (less memory reference)
  - instruction selection (less memory/CPU cycles/task)
  - peephole optimization (peek into a moving window of instructions)

- Minimizing the temporaries (IC optimization):

  idea: in $E \to E + E \mid E - E \cdots$

  $E \to E \text{ op } E$’s RHS defines a *lifetime* for temporary to store the $E$’s result.
- keep a counter \( c \)

- if r-value of \( t_c \) is needed, use \( t_c \) and decrement \( c \) by 1. Since temporaries are generated by the grammar in such a way that the same r-value is not referred to again, this temporary can be 'recycled'.

- if a new temporary is needed, use \( t_c \) with current value of \( c \) and increase it by 1 (the next expression will not use that temp so that it’s r-value can be referenced)

**ex:**
\[
x := a*(b+c) - g/h + e
\]
without minimization with min.
\[
\begin{array}{ll}
\text{t0 := b+c} & \text{t0 := b+c} \\
\text{t1 := a*t0} & \text{t0:=a*t0} \\
\text{t2:= g/h} & \text{t1:=g/h} \\
\text{t3:= t1-t2} & \text{t0:=t0-t1} \\
\text{t4:= t3+e} & \text{t0:= t0+e} \\
\text{x:= t4} & \text{x:=t0} \\
\end{array}
\]
\[
\begin{array}{ll}
5 temps & 2 temps \\
\end{array}
\]

in fact, this scheme will use only one temporary if the expression can be evaluated left-to-right, e.g. \(x:=a+b-c+d-e\)

- Algebraic simplification (IC optimization)
  - ex: eliminate \(x:=x+0\) and \(x:=x*1\)
• Peephole optimization (TC opt.)

look at limited sequence of instructions and improve

ex move R0, a

move a, R0

second one is redundant

the statement after an unconditional jump can be removed if it has no label (labels again: this can be done if we generate labels only when they are necessary)
Larger contexts in the peephole:

```
jump L1
...
L1 :  jump L2
```

First statement can be changed to `Jump L2`. Now, if there is no more L1 reference, `L1 :  jump L2` can be eliminated if it's after an unconditional jump.

- Idioms (TC optimization)
  - Certain machines have special instructions for e.g. `x := x+1;`
What does a function call look like in TC?

Usually, a language system will use certain registers for designated functions, e.g.,

R0 : contains the number 0
R1 : stack pointer
R2 : callee’s address
R3 : address of the next instruction

LDA offset(Rx), Ry : load Ry with offset added to the contents of Rx
STI Rx, offset(Ry): store Rx at the address given by offset added to the contents of Ry

BAL Rx, Ry: branch to address in Rx, putting the address of the next instruction in Ry

LDI offset(Rx), Ry: load Ry with the value at the address given by offset added to the contents of Rx
Calling f: assume that the target code of f is at f-addr. Caller does:

```
LDA f-addr(R0),R2 ; addr of f into R2
STI R1,next(R1) ; make space in stack
LDA next(R1),R1 ; increment stack ptr by next
BAL R2,R3 ; jump to R2, save next loc in R3
```

start-of-routine set up by f:

```
STI R3,ret-addr-offset(R1); save ret addr in AR
```
R3 must be saved due to possibility of nested calls

der-of-routine set up by f:

```
LDI ret-addr-offset(R1),R2 ; f’s return address now in R2
LDI 0(R1),R1 ; restore R1 (stack ptr)
BAL R2,R3 ; R2 now has address of caller’s next instr
```