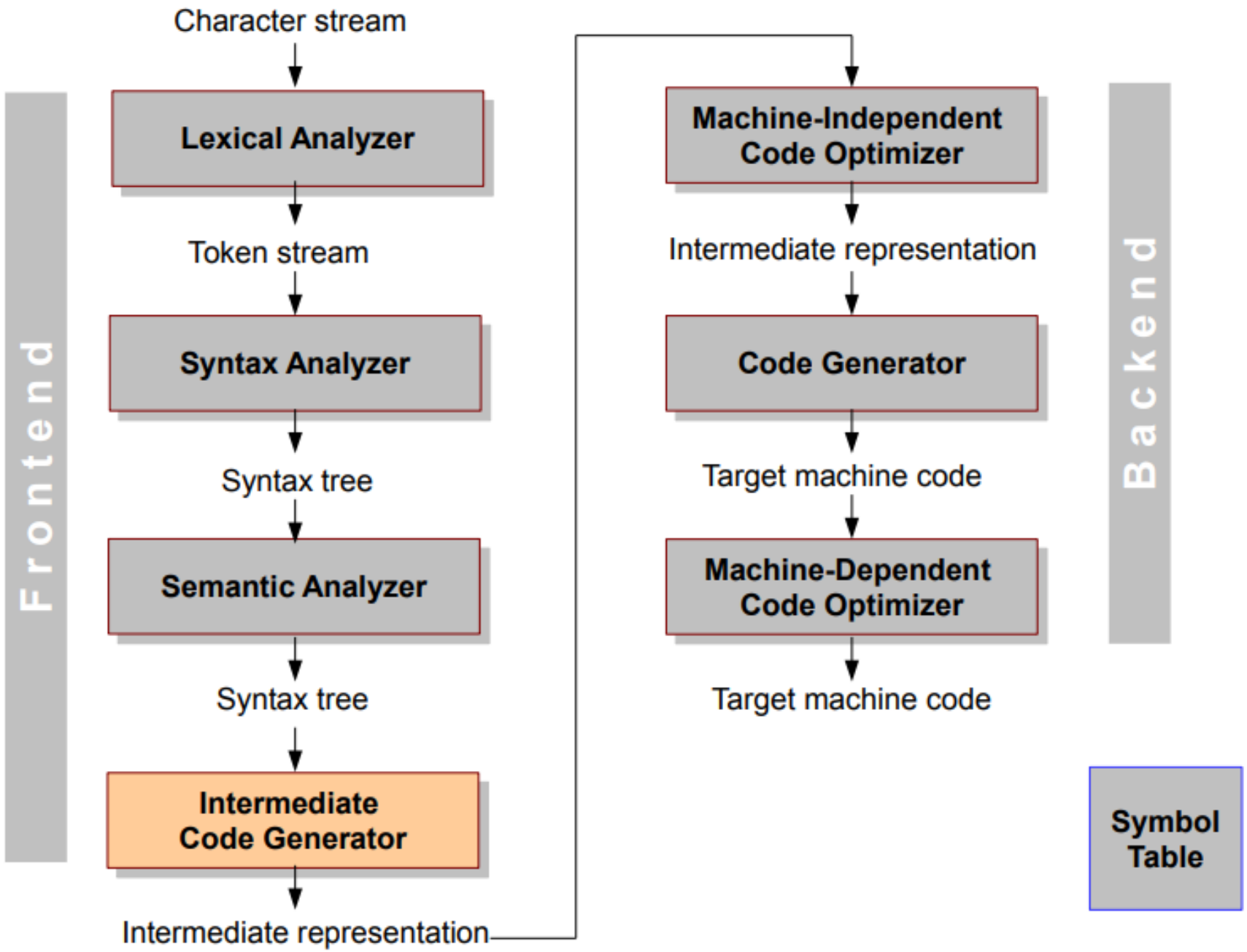
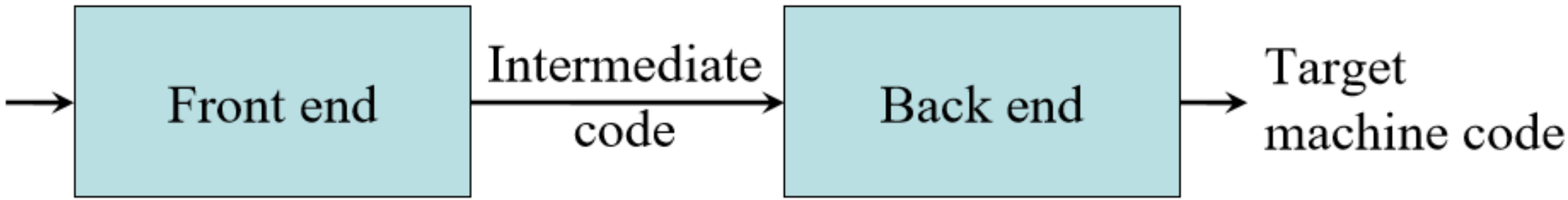
Intermediate Code Generation

Reading: chapter 8

How the syntax-directed methods can be used to translate into an intermediate form of programming language.


Character stream
Lexical Analyzer
Token stream
Syntax Analyzer
Syntax tree
Semantic Analyzer
Syntax tree
Intermediate Code Generator
Intermediate representation
Frontend
Machine-Independent Code Optimizer
Intermediate representation
Code Generator
Target machine code
Machine-Dependent Code Optimizer
Target machine code
Backend
Symbol Table

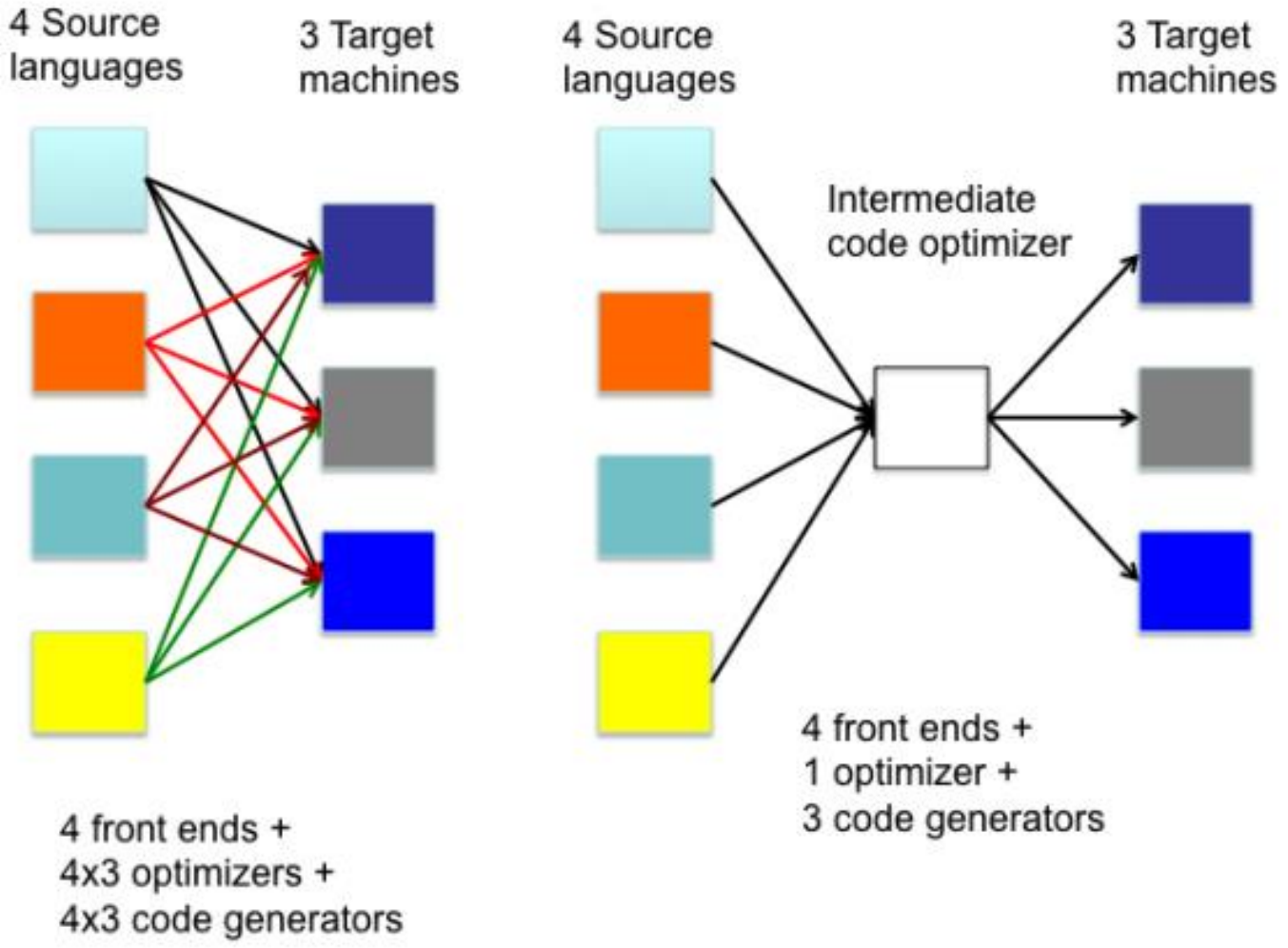
Intermediate Code Generation



The front end translate the source program into an intermediate representation from which the backend generates target code

Intermediate codes are machine independent codes, but they are close to machine instructions.

Why Intermediate Code ?



Intermediate Representations

There are three kinds of intermediate representations:

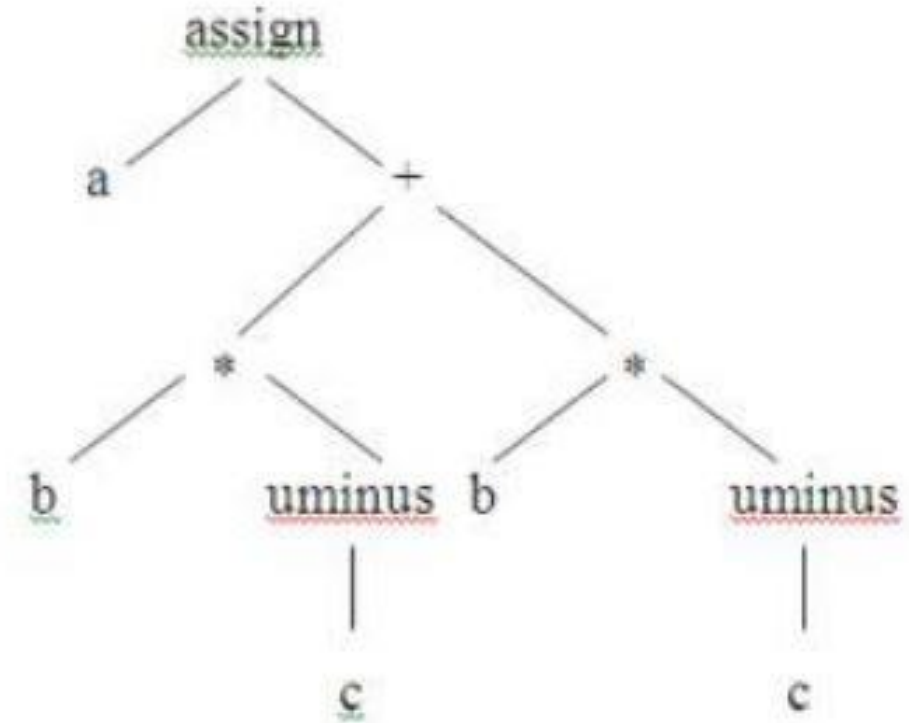
1. *Graphical representations* (e.g. Syntax tree or Dag)

2. *Postfix notation*: operations on values stored on operand stack (similar to JVM bytecode)

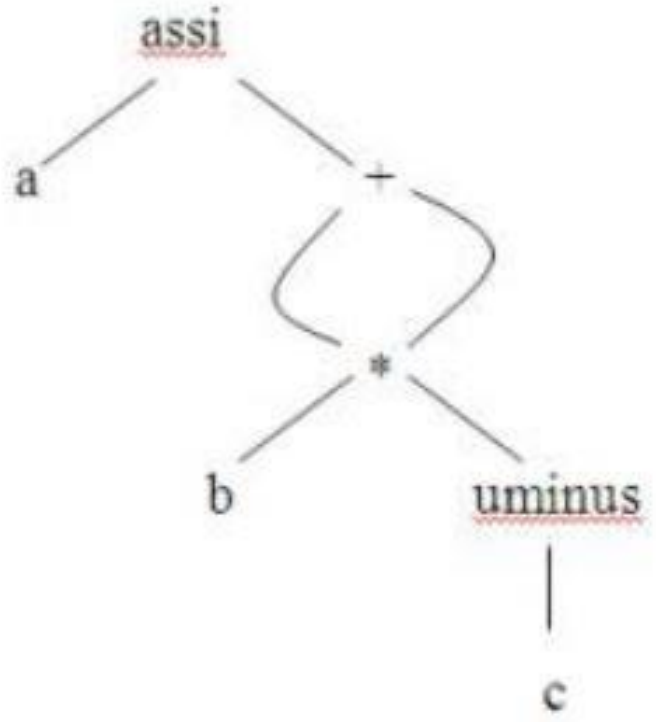
3. *Three-address code*: (e.g. triples and quads)Sequence of statement of the form

$x = y$ op z

Note: we discuss only three-address code representation



(a) Syntax tree



(b) Dag

Fig. Graphical representation of a:=b*- c+b*- c

Three-Address Code (Quadruples)

A quadruple is: $x = y$ ***op*** z

where x, y and z are names, constants or compiler-generated temporaries and ***op*** is any operator. (only one operator on the right side of the statement)

Postfix notation (much better notation because it looks like a machine code instruction)

op *y,z,x* apply operator *op* to y and z, and store the result in x.

We use the term "three-address code" because each statement usually contains three addresses (two for operands, one for the result).

Thus the source language like $x + y * z$ might be translated into a sequence

$t1 = y * z$

$t2 = x + t1$

where *t1* and *t2* are the compiler generated temporary name.

3-address code

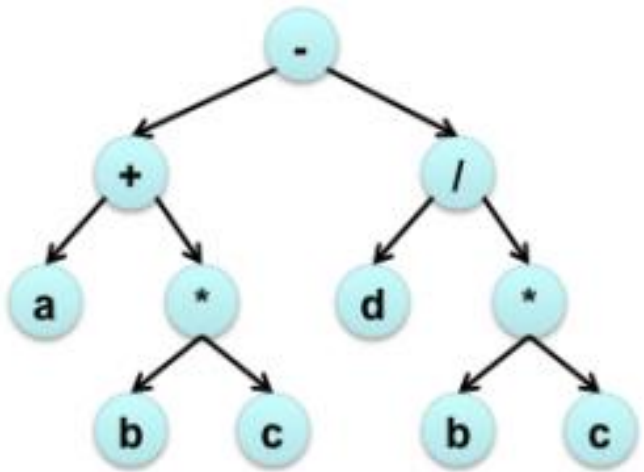
```
1  t1 = b*c
2  t2 = a+t1
3  t3 = b*c
4  t4 = d/t3
5  t5 = t2-t4
```

Quadruples

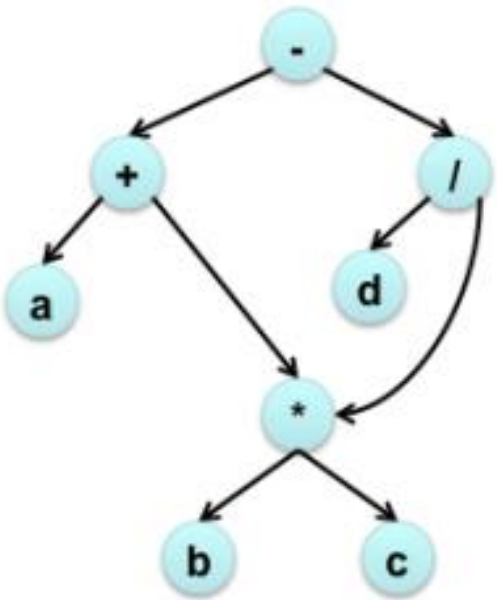
op	arg_1	arg_2	result
*	b	c	t1
+	a	t1	t2
*	b	c	t3
/	d	t3	t4
-	t2	t4	t5

Triples

	op	arg_1	arg_2
0	*	b	c
1	+	a	(0)
2	*	b	c
3	/	d	(2)
4	-	(1)	(3)



Syntax tree



DAG

Three-Address Statements

- Assignment statements: $x = y$ *op* z, *op is binary*
- Assignment statements: $x =$ *op* y, *op is unary*
- Indexed assignments: $x = y[i]$, $x[i] = y$
- Pointer assignments: $x =$ **&**y, $x =$ *y, *$x = y$
- Copy statements: $x = y$
- Unconditional jumps: **goto** *label*
- Conditional jumps: **if** x *relop* y **goto** *label*
- Function calls: **param** x… **call** p, n **return** y

Syntax-Directed Translation into Three-Address Code

Productions	Synthesized attributes:	
$S \rightarrow$ **id** = E	S.code	three-address code for evaluating S
| **while** E **do** S	S.begin	label to start of S or nil
$E \rightarrow E + E$	S.after	label to end of S or nil
| $E * E$	E.code	three-address code for evaluating E
| - E	E.place	a name that holds the value of E
| (E)		
| **id**		
| **num**		

To represent three address statements

$gen(E.\text{place} \text{ ‘=’ } E_1.\text{place} \text{ ‘+’ } E_2.\text{place})$

Code generation → `t3 = t1 + t2`

Syntax-directed Translation to produce three-address code for assignments

Syntax-Directed Translation into Three-Address Code

Productions	Semantic rules
$S \rightarrow \textbf{id} = E$	S.code = E.code || *gen*(**id**.place '=' E.place); S.begin = S.after = nil
$S \rightarrow \textbf{while}\ E$ **do** S_1	*(see next slide)*
$E \rightarrow E_1 + E_2$	*Returns a new temporary name* E.place = *newtemp*(); E.code = E_1.code || E_2.code || *gen*(E.place '=' E_1.place '+' E_2.place)
$E \rightarrow E_1 * E_2$	E.place = *newtemp*(); E.code = E_1.code || E_2.code || *gen*(E.place '=' E_1.place '*' E_2.place)
$E \rightarrow - E_1$	E.place = *newtemp*(); E.code = E_1.code || *gen*(E.place '=' 'uminus' E_1.place)
$E \rightarrow (E_1)$	E.place = E_1.place E.code = E_1.code
$E \rightarrow \textbf{id}$	E.place = **id**.name E.code = ''
$E \rightarrow \textbf{num}$	E.place = *newtemp*(); E.code = *gen*(E.place '=' **num**.value)

Write a grammar with semantic rules that converts C like while statement into 3 address code.

Syntax-Directed Translation into Three-Address Code

Production	Semantic rule
$S \rightarrow$ **while** E **do** S_1	S.begin = *newlabel*() S.after = *newlabel*() S.code = *gen*(S.begin ':') || E.code || *gen*('if' E.place '=' '0' 'goto' S.after) || S_1.code || *gen*('goto' S.begin) || *gen*(S.after ':')

Returns a new label

S.begin:	E.code
	if E.place = **0 goto** S.after
	S.code
	goto S.begin
S.after:	...

Practice 3-ADDRESS CODE

r= 7 + 2 * 3
t1=2
t2=3
t1=t1 *t2
t3=7
t1=t1 +t3
r=t1

a= -b *c
t1=-b
t2 = t1*c
a=t2

a:= b*-c + b*-c
t1:=-c
t2:=b*t1
t5:=t2+t2
a:=t5

a or b and not c
t1 = not c
t2 = b and t1
t3 = a or t2

if a>b then x =y +z.
if a>b then goto L1
goto L2
L1: t1 = y + z
x = t1
L2: ....

if a>b then x =y +z
else x = y-z
if a>b then goto L1
goto L2
L1: t1= y+z
x =t1
goto L3
L2: t1 = y-z
x = t1
L3:...

while a>b
do x =y +z.
L1: if a> b then goto L2
goto L3
L2: t1 = y+z
x = t1
goto L1
L3: .....

i = 2 * n + k
while i do
i = i – k
t1 =2
t2 = t1 *n
t3 = t2 +k
L1: if i =1 then goto L2
goto L3
L2: t4 = i-k
i = t4
goto L1
L3:...

while a<b do
if c< d then
x = y + z
else
x = y – z
L1: if a<b then GOTO L2
GOTO LNEXT
L2: if c<d then GOTO L3
GOTO L4
L3: t1 = y + z
x = t1
GOTO L1
L4: t1= y - z
x = t1
GOTO L1
LNEXT:

- Homework
- **Write the importance of 3AC in intermediate representation.**
- **Write a grammar with semantic rules that converts C like while statement into 3 address code.**