Compiler
--- Intermediate Code Generation

Zhang Zhizheng
seu_zzz@seu.edu.cn
School of Computer Science and Engineering,
Software College
Southeast University

Role I

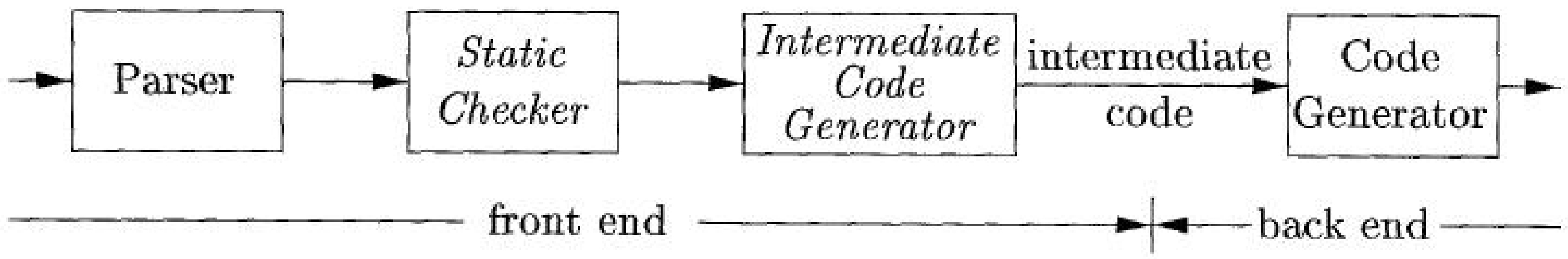
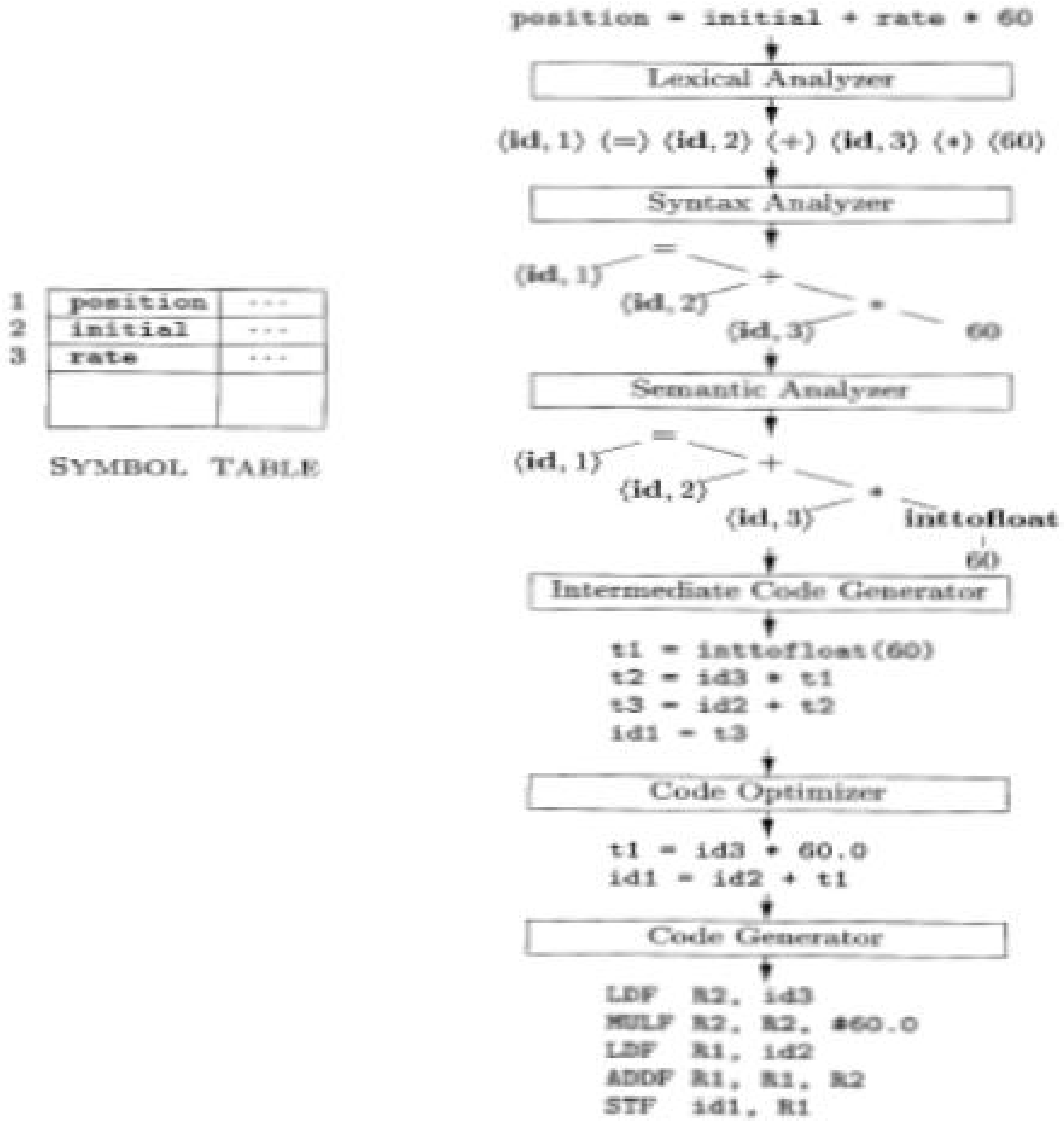


Figure 6.1: Logical structure of a compiler front end

Role II



1	position	···
2	initial	···
3	rate	···

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

Why Use Intermediate code

- Facilitate Portability
- Facilitate Opertimization

➢ $E \rightarrow E1 * Digit$ has E.val=E1.val × Digit.val

Digit.type=E1.type

E.code=E1.code '*' Digit.code

How to implement

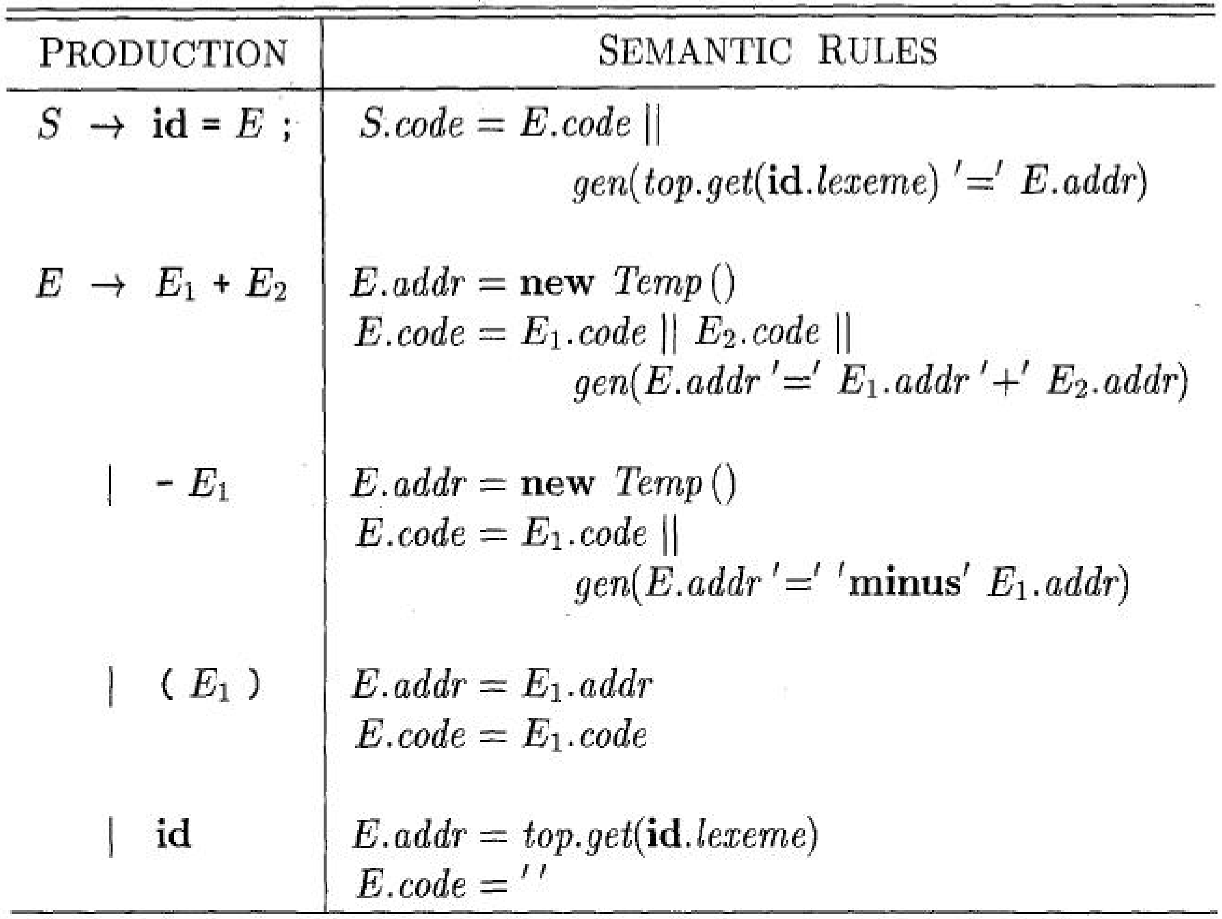
- Utilize semantics rule to evaluate the "code" attribute

➢ $E \rightarrow E1 * Digit$ has E.val=E1.val × Digit.val

Digit.type=E1.type

E.code=E1.code '∗' Digit.code

Example



PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \;\Vert\; gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp\,()$ $E.code = E_1.code \;\Vert\; E_2.code \;\Vert\; gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ -E_1$	$E.addr = \textbf{new}\ Temp\,()$ $E.code = E_1.code \;\Vert\; gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \textbf{id}$	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

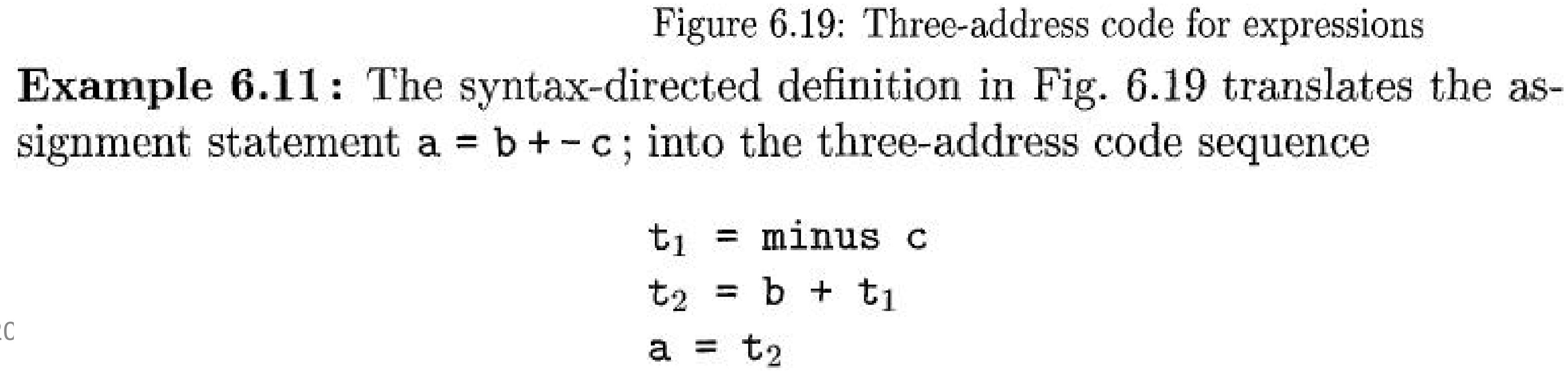


Figure 6.19: Three-address code for expressions

Example 6.11: The syntax-directed definition in Fig. 6.19 translates the assignment statement `a = b + - c`; into the three-address code sequence

```
t₁ = minus c
t₂ = b + t₁
a  = t₂
```

Intermediate Representation: Three address code

1. Assignment instructions of the form $x = y \; op \; z$, where *op* is a binary arithmetic or logical operation, and x, y, and z are addresses.

2. Assignments of the form $x = op \; y$, where *op* is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert an integer to a floating-point number.

3. *Copy instructions* of the form $x = y$, where x is assigned the value of y.

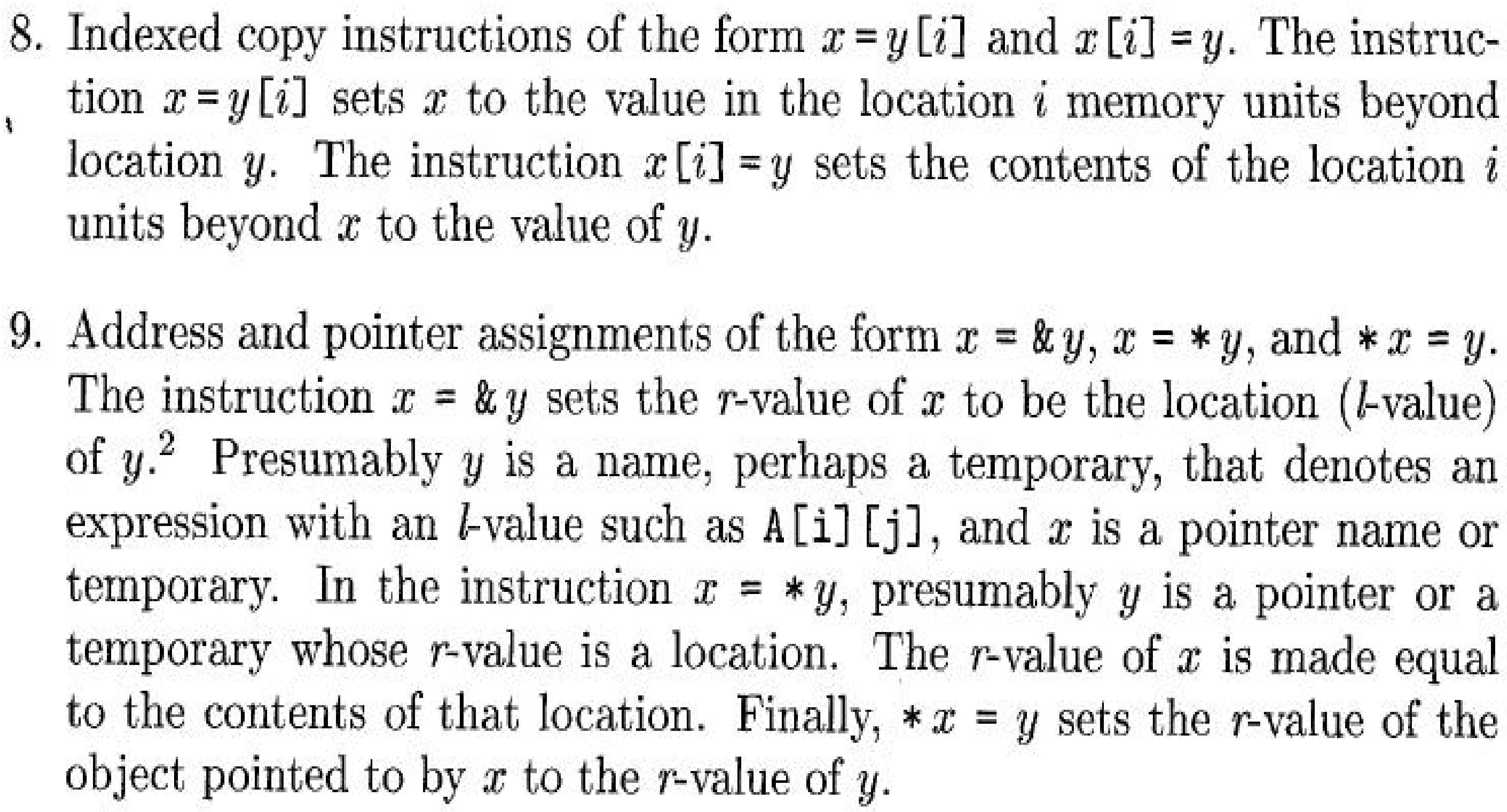
4. An unconditional jump `goto` L. The three-address instruction with label L is the next to be executed.

5. Conditional jumps of the form `if` x `goto` L and `ifFalse` x `goto` L. These instructions execute the instruction with label L next if x is true and false, respectively. Otherwise, the following three-address instruction in sequence is executed next, as usual.

6. Conditional jumps such as `if` x *relop* y `goto` L, which apply a relational operator (<, ==, >=, etc.) to x and y, and execute the instruction with label L next if x stands in relation *relop* to y. If not, the three-address instruction following `if` x *relop* y `goto` L is executed next, in sequence.

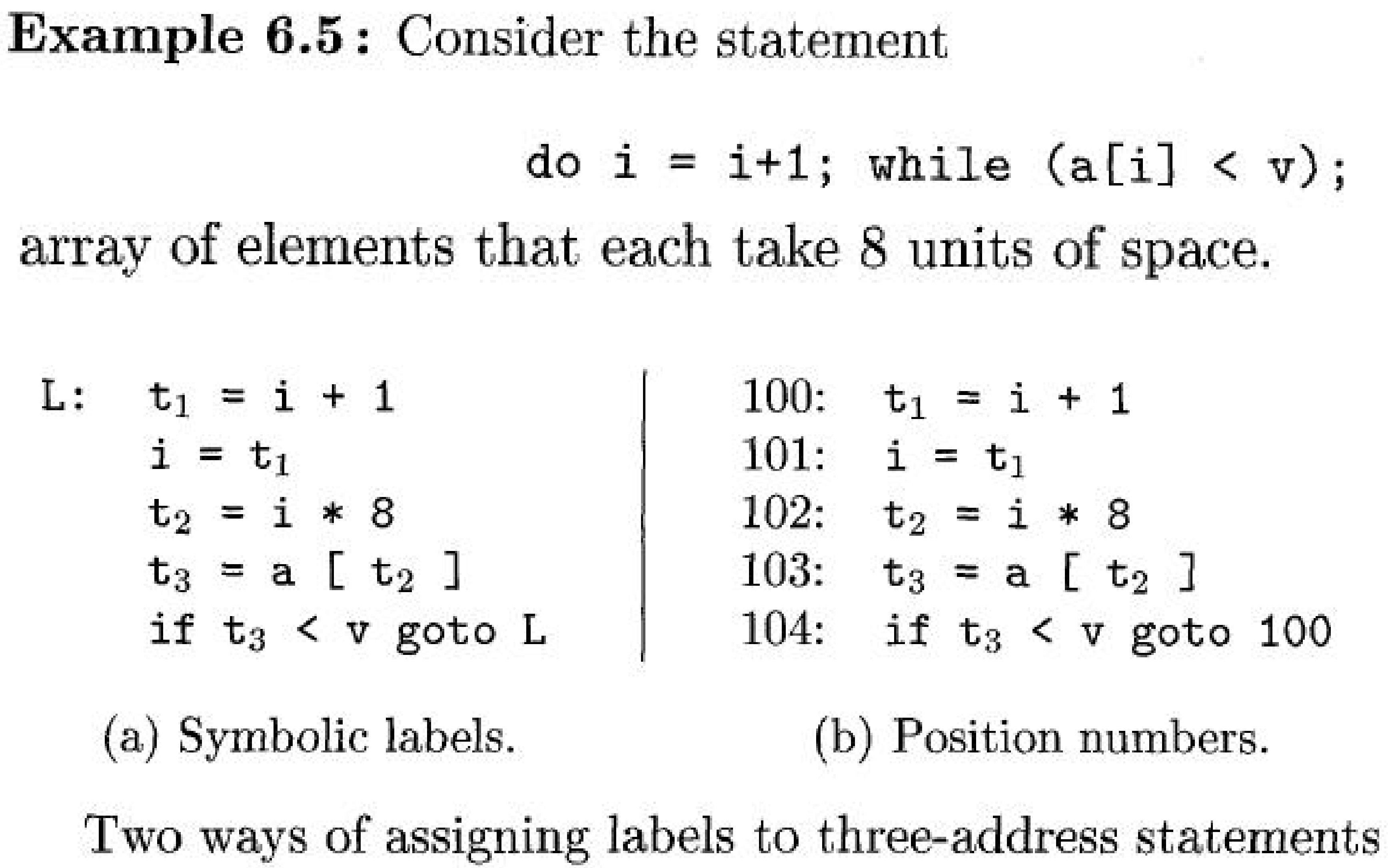
7. Procedure calls and returns are implemented using the following instructions: `param` x for parameters; `call` p, n and y = `call` p, n for procedure and function calls, respectively; and `return` y, where y, representing a returned value, is optional. Their typical use is as the sequence of three-address instructions

```
param x1
param x2
...
param xn
call p, n
```



8. Indexed copy instructions of the form $x = y[i]$ and $x[i] = y$. The instruction $x = y[i]$ sets x to the value in the location i memory units beyond location y. The instruction $x[i] = y$ sets the contents of the location i units beyond x to the value of y.

9. Address and pointer assignments of the form $x = \& y$, $x = * y$, and $* x = y$. The instruction $x = \& y$ sets the r-value of x to be the location (l-value) of y.[2] Presumably y is a name, perhaps a temporary, that denotes an expression with an l-value such as `A[i][j]`, and x is a pointer name or temporary. In the instruction $x = * y$, presumably y is a pointer or a temporary whose r-value is a location. The r-value of x is made equal to the contents of that location. Finally, $* x = y$ sets the r-value of the object pointed to by x to the r-value of y.


Example 6.5: Consider the statement

```
do i = i+1; while (a[i] < v);
```

array of elements that each take 8 units of space.

```
L:   t1 = i + 1
     i = t1
     t2 = i * 8
     t3 = a [ t2 ]
     if t3 < v goto L
```

(a) Symbolic labels.

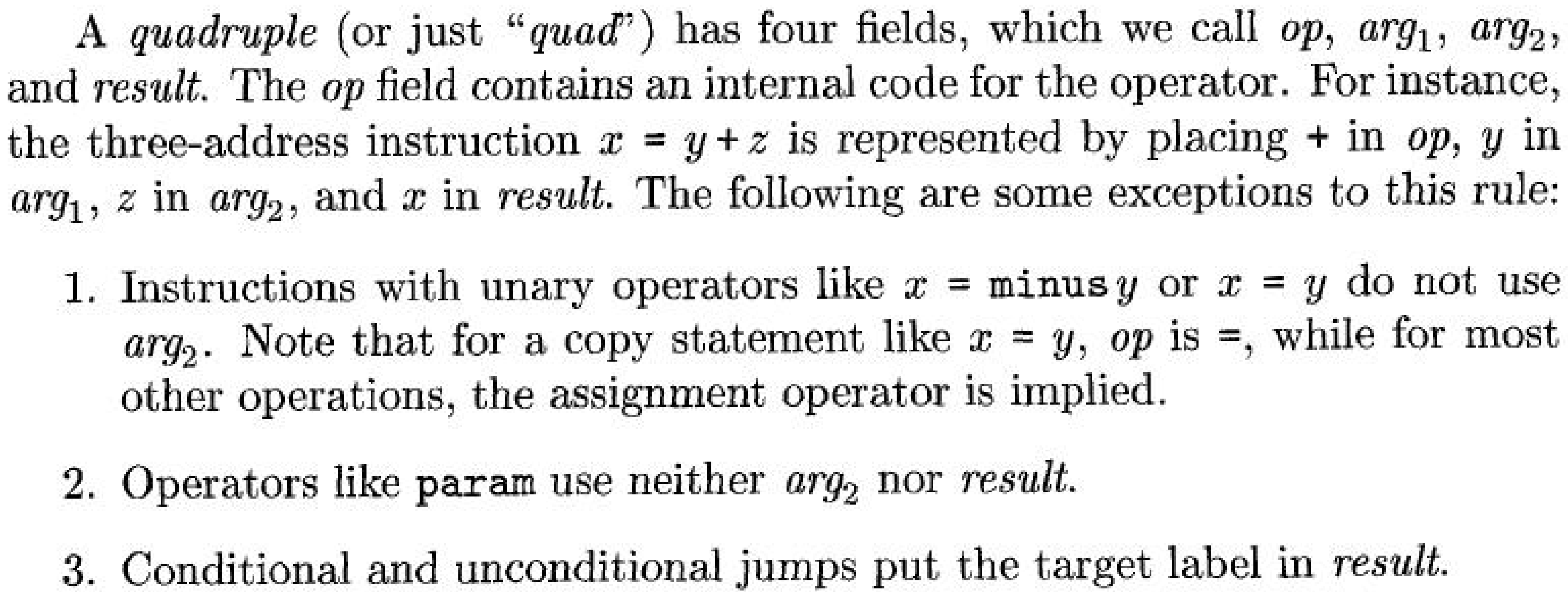
```
100:  t1 = i + 1
101:  i = t1
102:  t2 = i * 8
103:  t3 = a [ t2 ]
104:  if t3 < v goto 100
```

(b) Position numbers.

Two ways of assigning labels to three-address statements

Data Structure of Three Address Code I

- Quadruples



A *quadruple* (or just "*quad*") has four fields, which we call *op*, arg_1, arg_2, and *result*. The *op* field contains an internal code for the operator. For instance, the three-address instruction $x = y + z$ is represented by placing + in *op*, y in arg_1, z in arg_2, and x in *result*. The following are some exceptions to this rule:

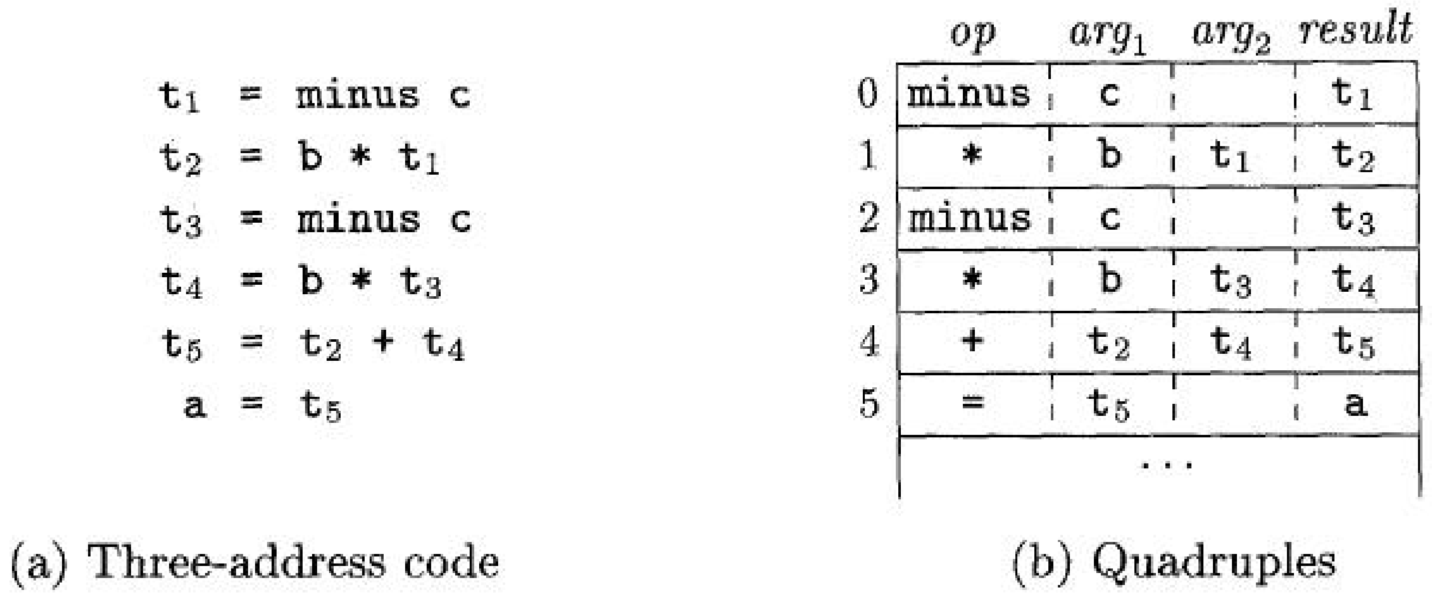
1. Instructions with unary operators like x = `minus` y or $x = y$ do not use arg_2. Note that for a copy statement like $x = y$, *op* is =, while for most other operations, the assignment operator is implied.

2. Operators like `param` use neither arg_2 nor *result*.

3. Conditional and unconditional jumps put the target label in *result*.

Data Structure of Three Address Code I

- Example



```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code

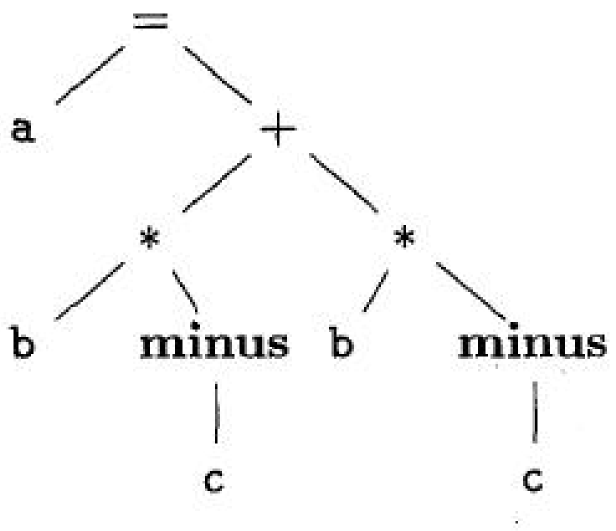
	op	*arg*$_1$	*arg*$_2$	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		. . .		

(b) Quadruples

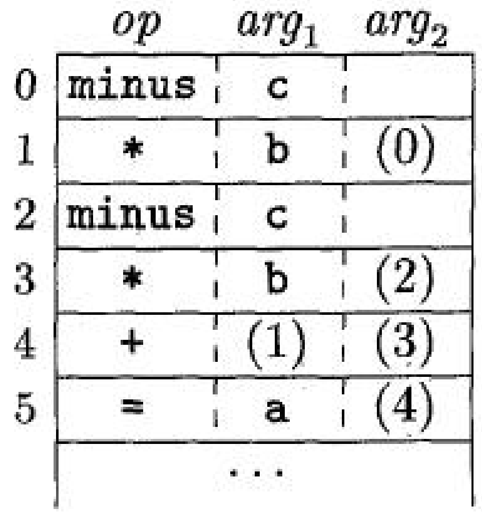
Figure 6.10: Three-address code and its quadruple representation

Data Structure of Three Address Code II

- Triples



(a) Syntax tree



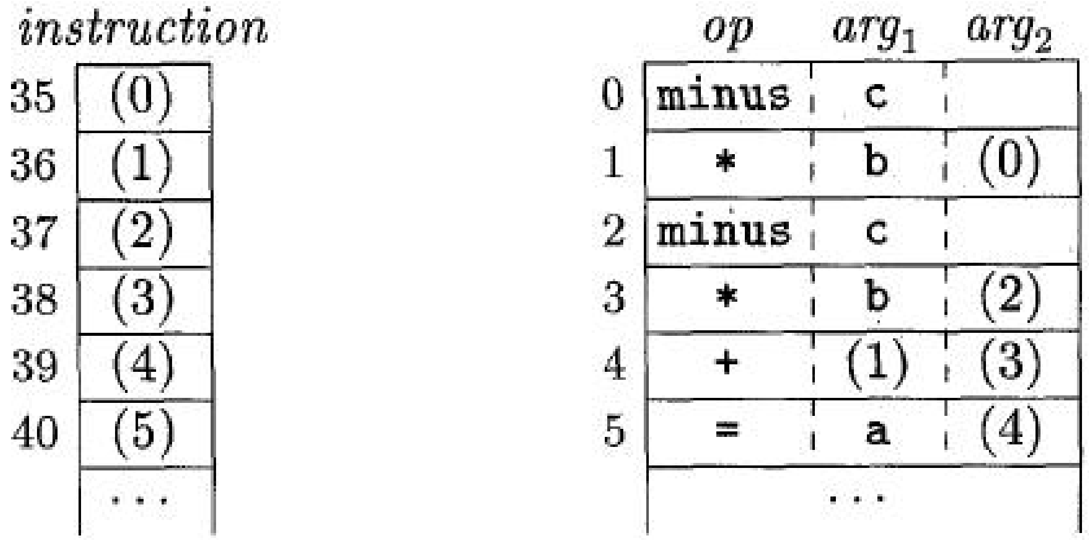
	op	*arg*$_1$	*arg*$_2$
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

(b) Triples

Figure 6.11: Representations of $a + a * (b - c) + (b - c) * d$

Data Structure of Three Address Code III

- Indirect Triples



	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	*arg*$_1$	*arg*$_2$
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

Figure 6.12: Indirect triples representation of three-address code

Data Structure of Three Address Code I

- Example

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code

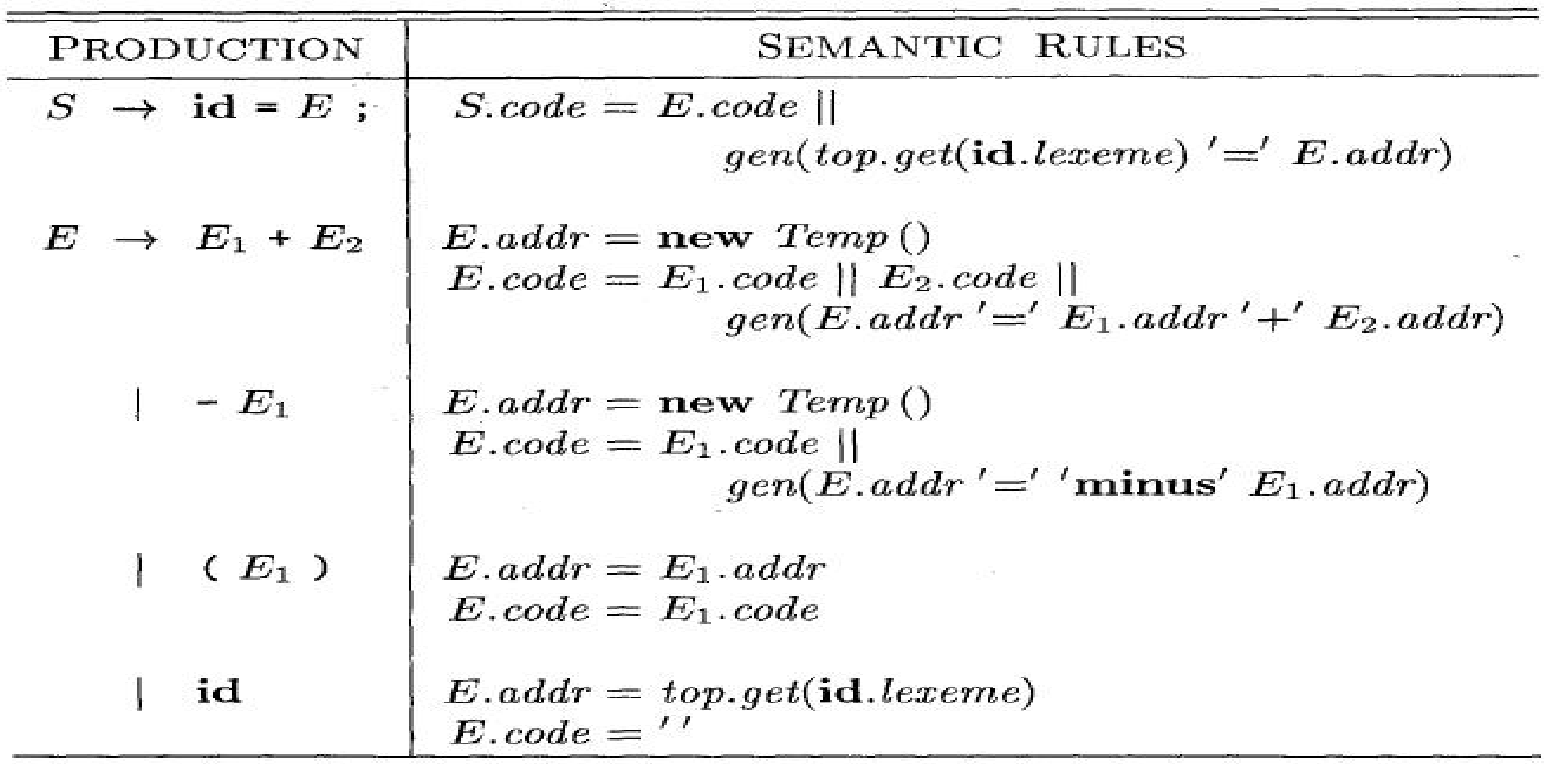
	op	*arg*$_1$	*arg*$_2$	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		. . .		

(b) Quadruples

Figure 6.10: Three-address code and its quadruple representation

Translation of common statements I

- Expression

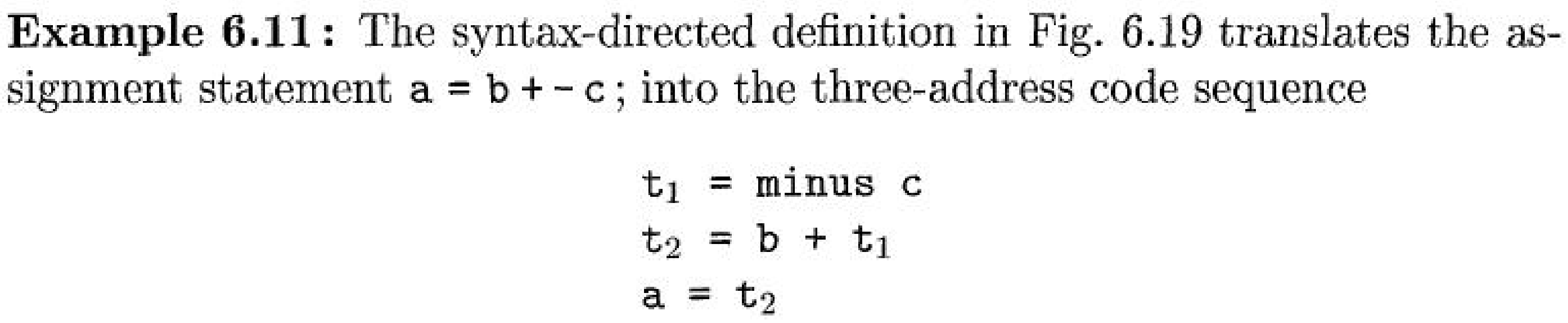


PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \mid\mid$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp\,()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ -E_1$	$E.addr = \mathbf{new}\ Temp\,()$ $E.code = E_1.code \mid\mid$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

Figure 6.19: Three-address code for expressions

Translation of common statements I

Example

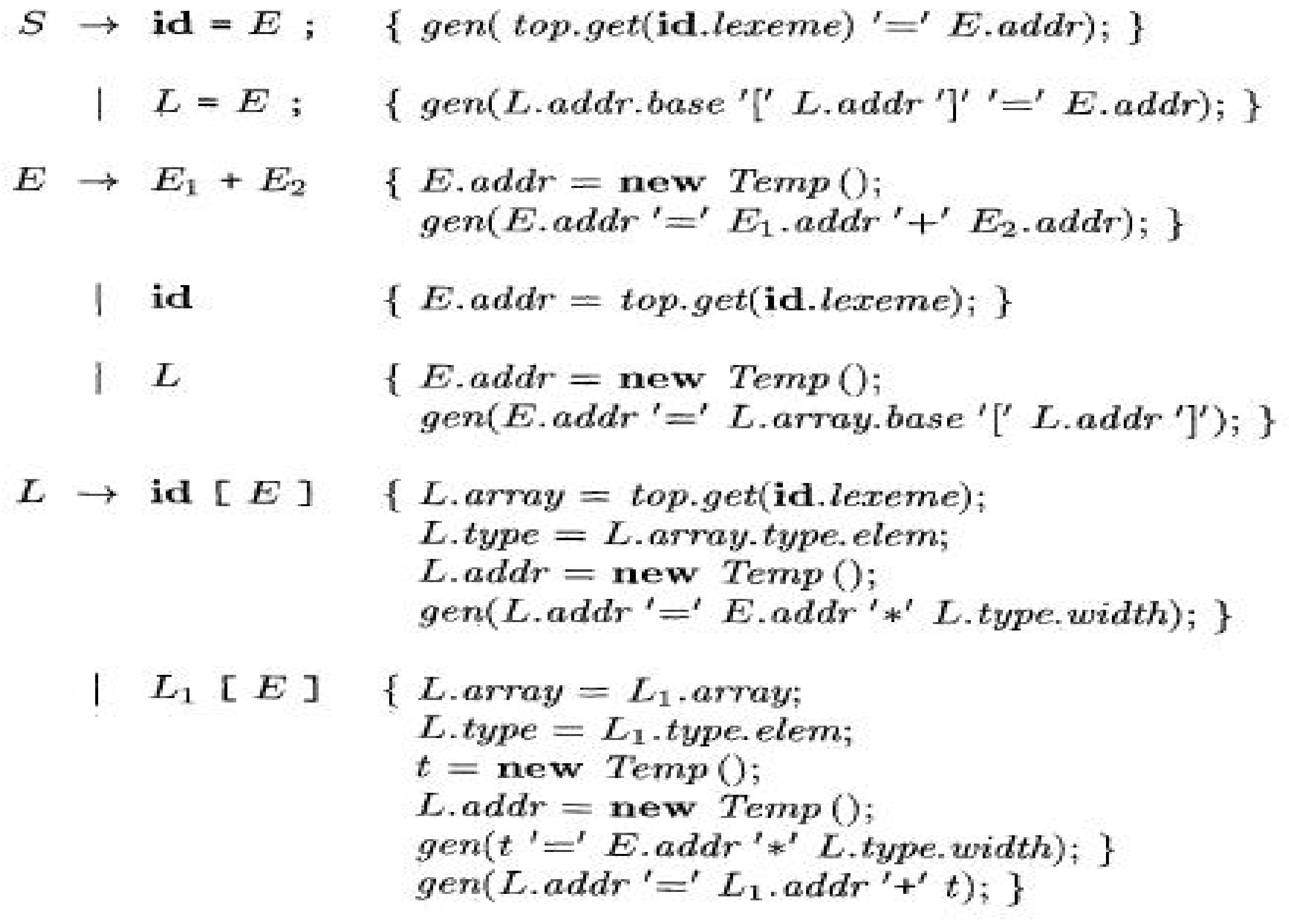


Example 6.11: The syntax-directed definition in Fig. 6.19 translates the assignment statement `a = b + - c ;` into the three-address code sequence

```
t1 = minus c
t2 = b + t1
a = t2
```

Translation of common statements II

- Array

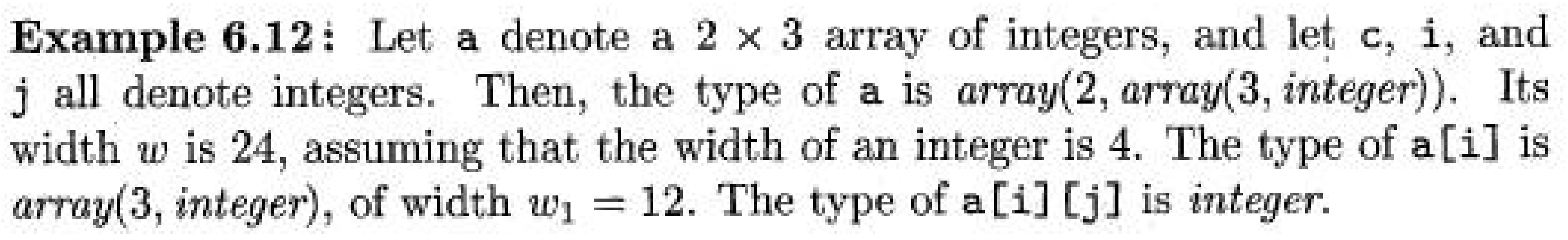


```
S → id = E ;     { gen( top.get(id.lexeme) '=' E.addr); }

  | L = E ;      { gen(L.addr.base '[' L.addr ']' '=' E.addr); }

E → E1 + E2      { E.addr = new Temp();
                   gen(E.addr '=' E1.addr '+' E2.addr); }

  | id           { E.addr = top.get(id.lexeme); }

  | L            { E.addr = new Temp();
                   gen(E.addr '=' L.array.base '[' L.addr ']'); }

L → id [ E ]     { L.array = top.get(id.lexeme);
                   L.type = L.array.type.elem;
                   L.addr = new Temp();
                   gen(L.addr '=' E.addr '*' L.type.width); }

  | L1 [ E ]     { L.array = L1.array;
                   L.type = L1.type.elem;
                   t = new Temp();
                   L.addr = new Temp();
                   gen(t '=' E.addr '*' L.type.width); }
                   gen(L.addr '=' L1.addr '+' t); }
```

Figure 6.22: Semantic actions for array references

Translation of common statements II

Example



Example 6.12: Let `a` denote a 2×3 array of integers, and let `c`, `i`, and `j` all denote integers. Then, the type of `a` is $array(2, array(3, integer))$. Its width w is 24, assuming that the width of an integer is 4. The type of `a[i]` is $array(3, integer)$, of width $w_1 = 12$. The type of `a[i][j]` is $integer$.

Translation of common statements II

Example (Cont.)

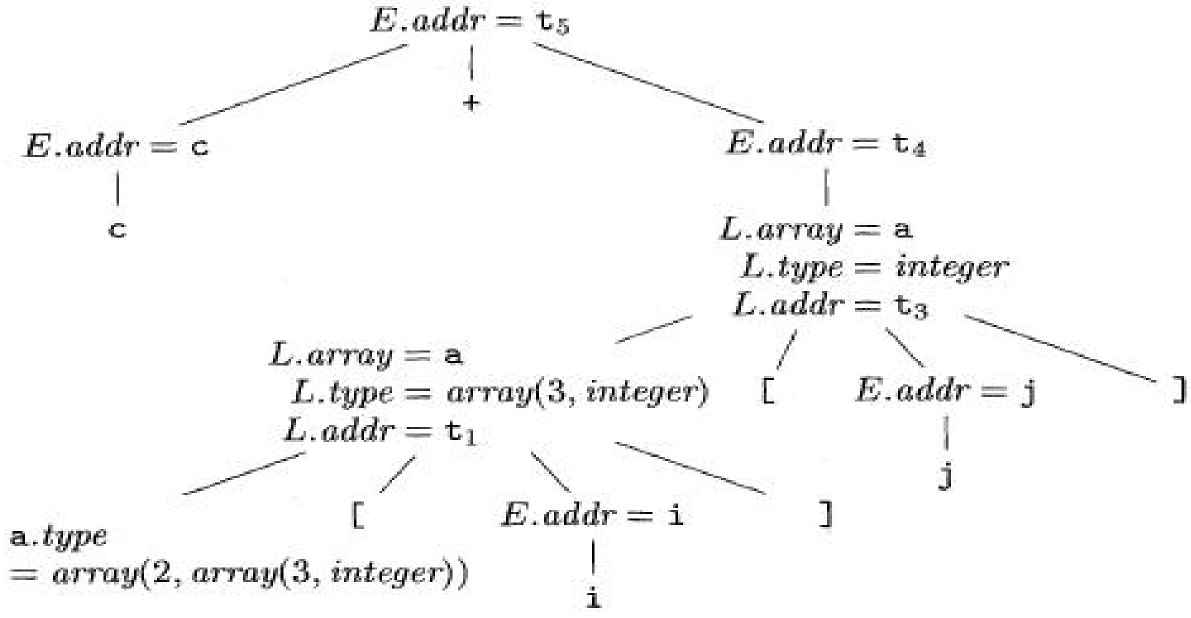


Figure 6.23: Annotated parse tree for c+a[i][j]

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```

Figure 6.24: Three-address code for expression c+a[i][j]

Translation of common statements III

- Flow-of-Control statements

$$\begin{array}{lcl} S & \rightarrow & \textbf{if} \ (\ B \ ) \ S_1 \\ S & \rightarrow & \textbf{if} \ (\ B \ ) \ S_1 \ \textbf{else} \ S_2 \\ S & \rightarrow & \textbf{while} \ (\ B \ ) \ S_1 \end{array}$$

$$B \rightarrow B \ || \ B \ \mid \ B \ \&\& \ B \ \mid \ ! \, B \ \mid \ (\ B \ ) \ \mid \ E \ \textbf{rel} \ E \ \mid \ \textbf{true} \ \mid \ \textbf{false}$$

Translation of common statements III

□Coding Approaches

Two ways:

1)**Short Circuit(Jumping) codes**, where .code is managed as inherited attribute.

2)**Backpatching codes**, where .code is managed as synthesized attribute.

Translation of common statements III

- Short Circuit Encoding I

① For Boolean Expression

In *short-circuit* (or *jumping*) code, the boolean operators &&, ||, and ! translate into jumps. The operators themselves do not appear in the code; instead, the value of a boolean expression is represented by a position in the code sequence.

Translation of common statements III

- Short Circuit Encoding I

① For Boolean Expression Example

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

```
      if x < 100 goto L2
      ifFalse x > 200 goto L1
      ifFalse x != y goto L1
L2:   x = 0
L1:
```

ifFalse is allowed

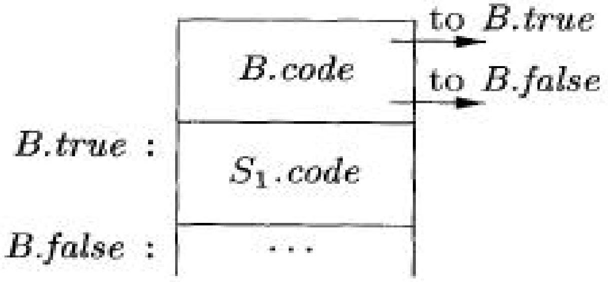
```
      if x < 100 goto L2
      goto L3
L3:   if x > 200 goto L4
      goto L1
L4:   if x != y goto L2
      goto L1
L2:   x = 0
L1:
```

ifFalse is not allowed

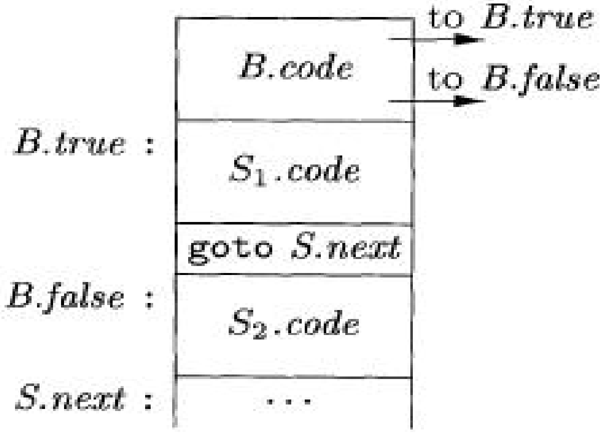
Translation of common statements III

- Short Circuit Encoding II

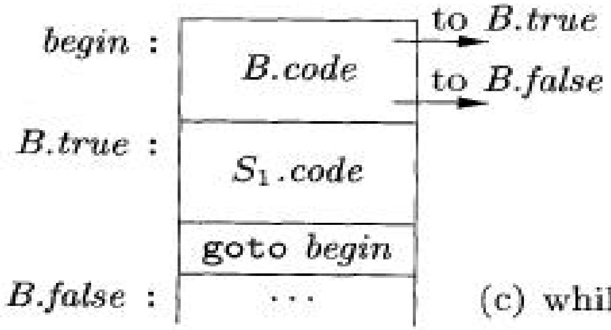
② For Flow Control



(a) if



(b) if-else



(c) while

Translation of common statements III

- Short Circuit Encoding II

② For Flow Control example

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

```
        if x < 100 goto L2
        goto L3
L3:     if x > 200 goto L4
        goto L1
L4:     if x != y goto L2
        goto L1
L2:     x = 0
L1:
```

ifFalse is not allowed

Translation of common statements III

- Short Circuit Encoding III

③ Semantics Rules

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \;\|\|\; label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \;\|\|\; label(B.true) \;\|\|\; S_1.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\quad \|\|\; label(B.true) \;\|\|\; S_1.code$ $\quad \|\|\; gen(\texttt{'goto'}\ S.next)$ $\quad \|\|\; label(B.false) \;\|\|\; S_2.code$
$S \rightarrow \textbf{while}\ (\ B\ )\ S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \;\|\|\; B.code$ $\quad \|\|\; label(B.true) \;\|\|\; S_1.code$ $\quad \|\|\; gen(\texttt{'goto'}\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \;\|\|\; label(S_1.next) \;\|\|\; S_2.code$

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \; \&\& \; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow \; ! \; B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \; \mathbf{rel} \; E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\quad \mid\mid gen(\text{'}\texttt{if}\text{'} \; E_1.addr \; \mathbf{rel}.op \; E_2.addr \; \text{'}\texttt{goto}\text{'} \; B.true)$ $\quad \mid\mid gen(\text{'}\texttt{goto}\text{'} \; B.false)$
$B \rightarrow \mathbf{true}$	$B.code = gen(\text{'}\texttt{goto}\text{'} \; B.true)$
$B \rightarrow \mathbf{false}$	$B.code = gen(\text{'}\texttt{goto}\text{'} \; B.false)$

Case Analysis

```
i=0;
tag=0;
while (tag==0 && i<=10) do
{
   j=1;
  while (tag==0 && j<=10) do
     if (a[i,j]==x) tag=1;
     else j=j+1;
  if (tag==0) i=i+1
}
```

```
i=1
tag=0
L1:  if (tag==0) goto L2
     goto L3
L2:  if (i<=10) goto L4
     goto L3
L4:  j=1
L5:  if (tag==0) goto L6
     goto L7
L6:  if (j<=10) goto L8
     goto L7
L8: t1=addrA-11
     t2=i*10
     t3=t2+j
     t4=t1[t3]
     if (t4==x) goto L9
     goto L10
L9: tag=1
     goto L11
L10: t5=j+1
     j=t5
L11: goto L5
L7: if (tag==0) goto L12
     goto L13
L12: t6=i+1
     i=t6
L13: goto L1
L3:
```

Jumping codes

(1) (=,0,_,i)

(2) (=,0,_,tag)

(3) (j==,tag,0,(5))

(4) (j,_,_,~~0~~(28))

(5) (j<=,i,10,(7))

(6) (j,_,_,~~4~~(28))

(7) (=,1,_,j)

(8) (j==,tag,0,(10))

(9) (j,_,_,~~0~~(23))

(10) (j<=,j,10,(12))

(11) (j,_,_,~~9~~(23))

(12) (-,addrA,11,t1)

(13) (*,i,10,t2)

(14) (+,t2,j,t3)

(15) (=[],t1[t3],_,t4)

(16) (j==,t4,x,(18))

(17) (j,_,_,(20))

(18) (=,1,_,tag)

(19) (j,_,_,(22))

(20) (+,j,1,t5)

(21) (=,t5,_,j)

(22) (j,_,_,(8))

(23) (j==,tag,0,(25))

(24) (j,_,_,(27))

(25) (+,i,1,t6)

(26) (=,t6,_,i)

(27) (j,_,_,(3))

(28)

Backpatching

Written Assignment

Please translate the following program fragment into three-address code using the form of short circuit code.

```
i=2;
loop=0;
while (loop==0 && i<=10) {
  j=1;
  while (loop ==0 && j<i)
        if (a[i,j] == x)
              loop=1;
        else j=j+1;
  if (loop==0) i=i+1;
}
```

Notes: Here we assume that the declaration of array A is array [1..10,1..10], each data element of array A would **use 2 storage units,** and the start address of array A's storage area is addrA.

Appendix Backpatching

1.Why and what is backpatching?

- When generating code for boolean expressions and flow-of-control statements , we may not know the labels that control must go to.
- We can get around this problem by generating a series of branching statement with the targets of the jumps temporarily left unspecified.
- Each such statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined.
- This subsequent filling in of labels is called backpatching

2.Functions to manipulate lists of labels related to backpatching

- Makelist(i)
  - Creates a new list containing only i, an index into the array of quadruples; makelist returns a pointer to the list it has made.
- Merge(p1,p2)
  - Concatenates the lists pointed to by p1 and p2, and returns a pointer to the concatenated list.
- Backpatch(p,i)
  - Inserts i as the target label for each of the statements on the list pointed to by p.

3.Boolean expression

1)Modify the grammar

$E \rightarrow E^A E \mid E^0 E \mid \text{not } E \mid (E) \mid i \mid E_a \text{ rop } E_a$

$E^A \rightarrow E \text{ and}$

$E^0 \rightarrow E \text{ or}$

2)Semantic Rules

(1) $E \rightarrow i$ {E•TC=NXQ; E•FC=NXQ+1;
GEN(jnz,ENTRY(i),_,0);
GEN(j,_,_,0)}

3.Boolean expression

2)Semantic Rules

(2) $E \rightarrow E_a$ rop E_a

{$E \bullet TC=NXQ$; $E \bullet FC=NXQ+1$;

GEN(jrop, $E_a^{(1)} \bullet PLACE$, $E_a^{(2)} \bullet PLACE$,0);

GEN(j,_,_,0)}

(3) $E \rightarrow (E^{(1)})$

{$E \bullet TC= E^{(1)} \bullet TC$; $E \bullet FC= E^{(1)} \bullet FC$}

(4) $E \rightarrow$ not $E^{(1)}$

{$E \bullet TC= E^{(1)} \bullet FC$; $E \bullet FC= E^{(1)} \bullet TC$}

3.Boolean expression

2)Semantic Rules

(5)$E^A \rightarrow E^{(1)}$ and {BACKPATCH($E^{(1)}$•TC,NXQ);

E^A•FC= $E^{(1)}$•FC;}

(6) $E \rightarrow E^A E^{(2)}$

{E•TC= $E^{(2)}$•TC;

E•FC=MERG(E^A•FC,$E^{(2)}$•FC}

3.Boolean expression

2)Semantic Rules

(7)$E^0 \rightarrow E^{(1)}$ or

{BACKPATCH($E^{(1)}\bullet$FC,NXQ);

$E^0\bullet$TC= $E^{(1)}\bullet$TC;}

(8) $E \rightarrow E^0 E^{(2)}$

{$E\bullet$FC= $E^{(2)}\bullet$FC;

$E\bullet$TC=MERG($E^0\bullet$TC,$E^{(2)}\bullet$TC}

Translate A and B or not C

INPUT	SYM	TC	FC	quadruple
A and B or not C#	#	-	-	
and B or not C#	#i	--	--	
and B or not C#	#E	-1	-2	1.(jnz,a,-,(3))
B or not C#	#E and	-1-	-2-	2.(j,-,-(5))
B or not C#	# E^A	--	-2	
or not C#	# E^Ai	---	-2-	

INPUT	SYM	TC	FC	quadruple
or not C	# $E^{A}E$	— —3	—2 4	3.(jnz,B,—,0)
#or not C#	#E	—3	—4	4.(j,—,—(5))
or not C#	#E or	—3—	—4—	
not C#	# E^{0}	—3	— —	
C#	# E^{0} not	—3—	— — —	
#	# E^{0} not i	—3—	— — — —	
#	# E^{0} notE	═3—5	— — —6	5.(jnz,C,—,0)
#	# E^{0}E	—3 6	— —5	6.(j,—,—,3)
#	#E	—6	—5	
success				

4.Flow of control statements

1) modify the grammar

$S \rightarrow$ if E then $S^{(1)}$ else $S^{(2)}$ ➔

  $C \rightarrow$ if E then

  $T \rightarrow C\ S^{(1)}$ else

  $S \rightarrow T\ S^{(2)}$

$S \rightarrow$ if E then $S^{(1)}$ ➔

  $C \rightarrow$ if E then

  $S \rightarrow C\ S^{(1)}$

4.Flow of control statements

2) Semantic Rules

$C \rightarrow$ if E then {BACKPATCH(E•TC,NXQ);
C•CHAIN=E•FC;}

$T \rightarrow C\ S^{(1)}$ else {q=NXQ; GEN(j,—,—0);
BACKPATCH(C•CHAIN,NXQ);
T •CHAIN=MERG($S^{(1)}$•CHAIN,q)}

$S \rightarrow T\ S^{(2)}$ {S•CHAIN=MERG(T•CHAIN,$S^{(2)}$•CHAIN)}

$S \rightarrow C\ S^{(1)}$ {S•CHAIN=MERG(C•CHAIN,$S^{(1)}$•CHAIN)}

e.g.

```
If a then
   if b then
       A:=2
   else A:=3
Else if  c then
   A=4
Else a=5
```

```
(1) (jnz,a,_,0)
(2) (j,_,_,0)
```

```
If a then
   if b then
      A:=2
   else A:=3
Else if  c then
   A=4
Else a=5
```

(1)(jnz,a,_,(3))
(2)(j,_,_,0)
(3)(jnz,b,_,0)
(4)(j,_,_,0)

Ca•CHAIN->2

```
If a then
   if b then
      A:=2
   else A:=3
Else if  c then
   A=4
Else a=5
```

```
(1)(jnz,a,_,(3))
(2)(j,_,_,0)
(3)(jnz,b,_,(5))
(4)(j,_,_,0)
(5)(:=,2,_,A)
```

Ca•CHAIN->2

Cb•CHAIN->4

```
If a then
   if b then
       A:=2
   else A:=3
Else if  c then
   A=4
Else a=5
```

```
(1)(jnz,a,_,(3))
(2)(j,_,_,0)
(3)(jnz,b,_,(5))
(4)(j,_,_,(7))
(5)(:=,2,_,A)
(6)(j,_,_,0)
```

Ca•CHAIN->2
Cb•CHAIN->6

Answer

(1)(jnz,a,_,(3))

(2)(j,_,_,(9))

(3)(jnz,b,_,(5))

(4)(j,_,_,(7))

(5)(:=,2,_,A)

(6)(j,_,_,0)

(7)(:=,3,_,A)

(8)(j,_,_,6)

(9)(jnz,c,_,(11))

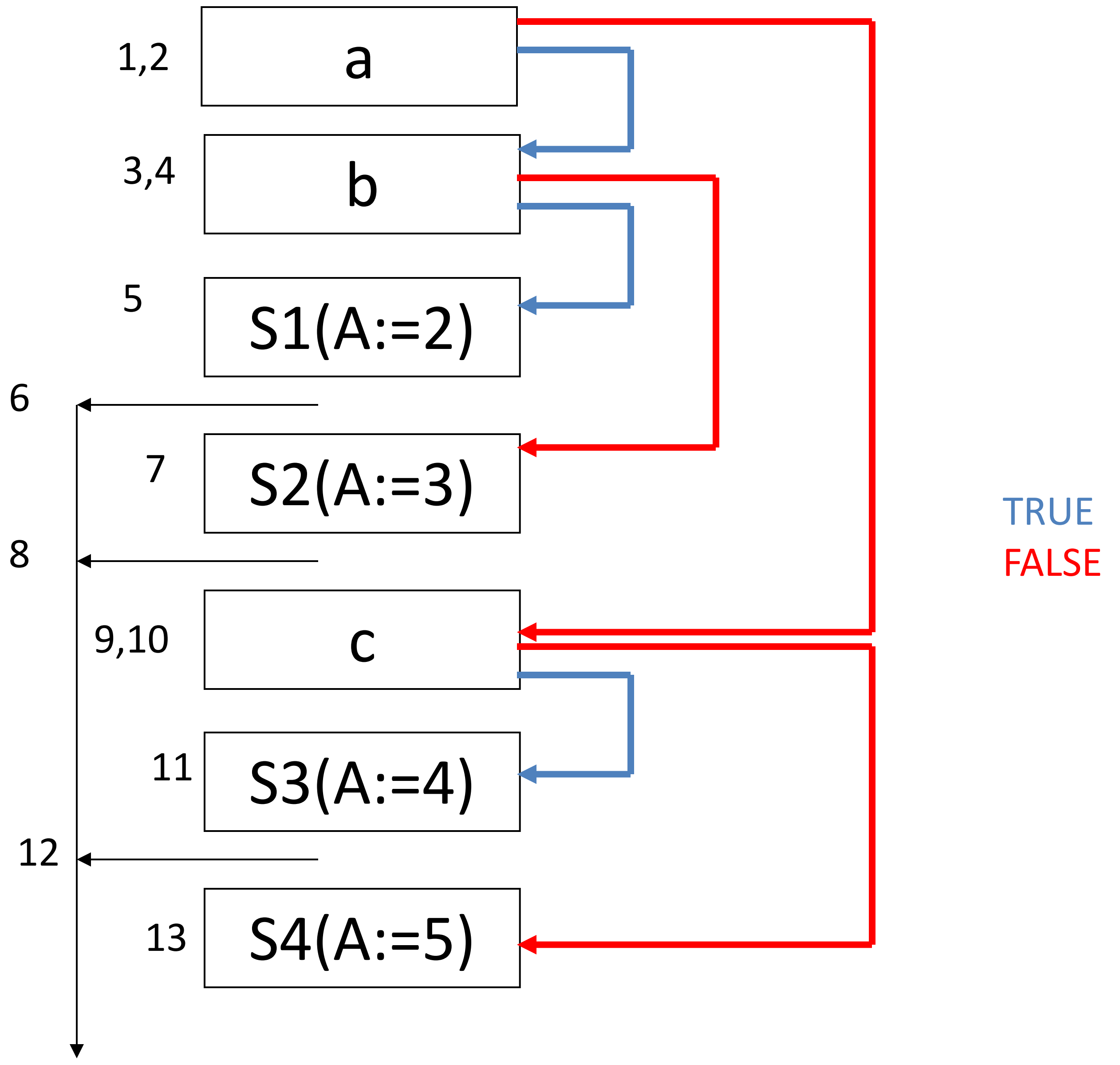
(10)(j,_,_,(13))

(11)(:=,4,_,A)

(12)(j,_,_,8)

(13)(:=,5,_,A)

S•CHAIN->6->8->12


1,2
a
3,4
b
5
S1(A:=2)
6
7
S2(A:=3)
8
9,10
c
11
S3(A:=4)
12
13
S4(A:=5)
TRUE
FALSE

4.Flow of control statements

3) While statement

$S \rightarrow$ while E do $S^{(1)}$ ➔

$W \rightarrow$ while

$W^{d} \rightarrow W$ E do

$S \rightarrow W^{d}\, S^{(1)}$

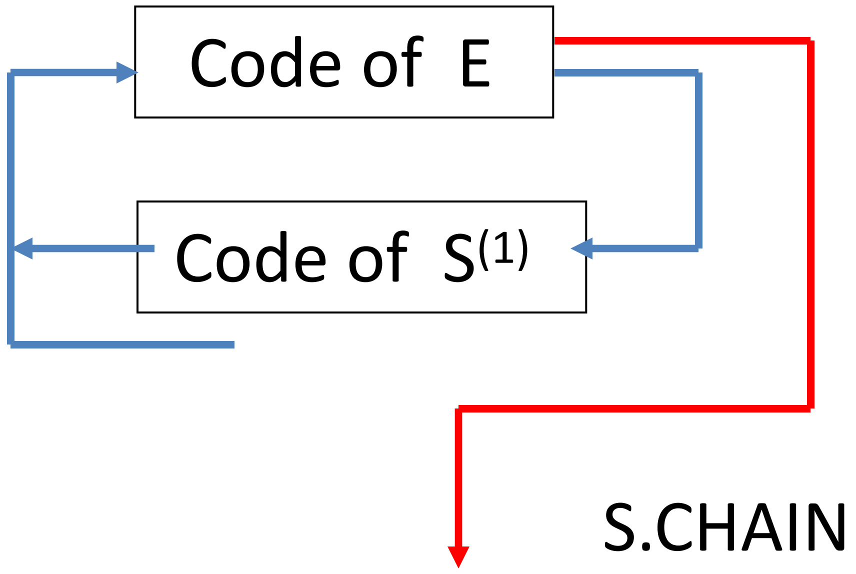
4.flow of control statements

3) While statement

```
W →while        {W•QUAD=NXQ}
Wᵈ →W E do      {BACKPATCH(E•TC,NXQ);
                 Wᵈ•CHAIN=E•FC;
                 Wᵈ•QUAD=W•QUAD;}
S → Wᵈ S⁽¹⁾{BACKPATCH(S⁽¹⁾•CHAIN, Wᵈ•QUAD);
                GEN(j,_,_, Wᵈ •QUAD);
                S • CHAIN= Wᵈ•CHAIN}
```

4.flow of control statements

3) While statement



e.g.

```
While (A<B) do
  if (C<D)  then
   X:=Y+Z;
```

➔ (100) (j<,A,B,0)

(101)(j,_,_,0)

e.g.

```
While (A<B) do
   if (C<D)  then
    X:=Y+Z;
```

```
  ➔:   (100) (j<,A,B,(102))
       (101)(j,_,_,0)
       (102)(j<,C,D,0)
       (103)(j,_,_,(100))
```

e.g.

```
While (A<B) do
  if (C<D)  then
   X:=Y+Z;
```

➔: (100) (j<,A,B,(102))

(101)(j,_,_,0)

(102)(j<,C,D,(104))

(103)(j,_,_,(100))

(104)(+,Y,Z,T_1)

(105)(:=, T_1,_,X)

e.g.

```
While (A<B) do
  if (C<D)  then
   X:=Y+Z;
```

➔:

```
(100) (j<,A,B,(102))        (106)(j,_,_,(100))
(101)(j,_,_,(107))
(102)(j<,C,D,(104))
(103)(j,_,_,(100))
(104)(+,Y,Z,T1)
(105)(:=, T1,_,X)
```