Syntax directed translation and Semantic analysis

Semantic Analysis

- Semantic rules can be attached to grammar to perform type checking.
- Semantic rules are a collection of procedures called at appropriate times by the parser.
- Semantic rules can be applied to the grammar by attaching attributes to the CFG.
- CFG + sematic rules Attribute grammar

Syntax-directed translation: analysis and synthesis

- Translation process driven by the syntactic structure of the program, as generated by the parser.

- In syntax-directed translation, the semantic analysis and translation steps of the compilation process is divided in two parts:
  - analysis (syntactic, semantic)
  - synthesis (code generation and optimization)

- The semantic analysis becomes the link between analysis and synthesis: code generation (synthesis) is conditional to positive semantic analysis.

- The syntax-directed translation process is inducing very strong coupling between the syntax analysis phase, the semantic checking phase, and the translation phase.

Syntax-directed translation: semantic actions

- Some semantic actions (implemented as ***semantic routines***) do the analysis phase by performing semantic checking in productions that need such checks, depending on the semantic rules defined by the language specification, e.g. *type checking*.

- Semantic actions assemble information in order to validate and generate a meaning (i.e. translation) for the program elements generated by the productions.

- They are the starting point for code generation (synthesis).
- Thus, the semantic routines are the heart of the compiler.

Syntax-directed translation and attribute grammars

- Semantic routines can be formalized using **attribute grammars**.
- Attribute grammars augment ordinary context-free grammars with **attributes** that represent semantic properties such as type, value or correctness used in semantic analysis (checking) and code generation (translation).

- It is useful to keep checking and translation facilities distinct in the semantic routines' implementation.

- Semantic checking is machine-independent and code generation is not, so separating them gives more flexibility to the compiler (front/back end).

Attributes

- An ***attribute*** is a property of a programming language construct, including *data type, value, memory location/size*, *translated code,* etc.

- Implementation-wise, they are also called ***semantic records***.

- The process of computing the value of an attribute is called ***binding***. *Static binding* concerns binding that can be done at compile-time, and *dynamic binding* happens at run-time, e.g. for polymorphism.

Attributes migration

- Static attribute binding is done by ***gathering***, ***propagating***, and ***aggregating*** attributes while traversing the parse tree.

- Attributes are *gathered* at tree leaves, *propagated* across tree nodes, and *aggregated* at some parent nodes when additional information is available.

- This can be done as the program is being parsed using *syntax-directed translation*.

- **Synthetized attributes** : attributes gathered from a child in the syntax tree
- **Inherited attributes** : attributes gathered from a sibling in the syntax tree

Attributes

- Attribute grammar
    - E □E + T {E.value = E.value + T.value}

- Based on the way the attributes obtain their values they are divided into two categories:

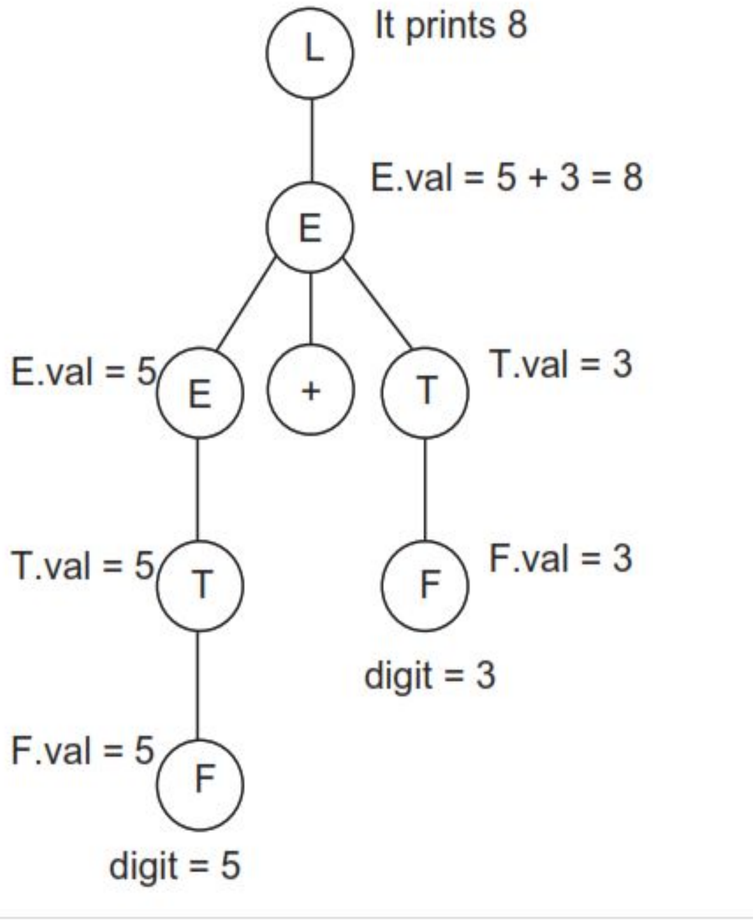
    - **Synthesized** – obtain the values from child nodes
    - **Inherited -** obtain the values from parents or siblings

Synthesized attributes

- The attributes that obtain values from the attribute values of their child nodes

Production	Semantic Rules
L□E	print(E.val)
E□E+T	E.val=E1.val + T.val
E□ T	E.val = T.val
T□T*F	T.val = T1.val * F.val
T□ F	T.val = F.val
F□digit	F.val = digit.lexval

Example:



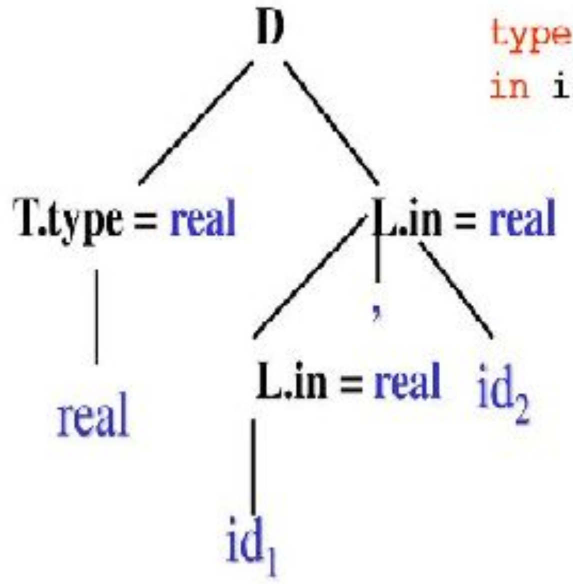
Inherited attributes

Example:

- Inherited attributes take values

from parents and/or siblings.

$D \rightarrow T\ L$	$\{ L.in := T.type \}$
$T \rightarrow int$	$\{T.type := integer \}$
$T \rightarrow real$	$\{T.type := real \}$
$L \rightarrow L_1 , id$	$\{L_1.in := L.in ; addtype (id.entry, L.in)\}$
$L \rightarrow id$	$\{addtype (id.entry, L.in)\}$



type is a synthesized attribute
in is an inherited attribute

Intermediate Code Generation

Many compilers convert the source code to an intermediate representation.

The benefits of using machine-independent intermediate code are as follows:

- It reduces the number of optimizers and code generators.
- It is easy to generate and translate code into the target program.
- It enhances portability.
- It is easy to optimize as compared to machine-dependent code.
- The representation of intermediate code can be directly executed by compile Or the interpreter.

Intermediate Code Representation

- Graphical representations can be **parse trees**, **abstract syntax trees**, **DAG**, etc.
- Linear representations are non-graphical **like three-address code (TAC), static single assignment (SSA**), etc.
- Representation of TACs
  - Quadruples
  - Triples
  - Indirect triples

Intermediate Code Representation- Example

Represent the expression a = (b + c) * −c in quadruple, triple and indirect triple representation

- TAC:

```
T1 = b + c
T2 = −c
T3 = T1* T2
a = T3
```

TAC:

T1 = b + c

T2 = −c

T3 = T1* T2

a = T3

Quadruple

	op	x (operand1)	y (operand2)	z (result)
(1)	+	b	c	T1
(2)	-	c		T2
(3)	*	T1	T2	T3
(4)	=	a	T3	

Triple

	op	x (operand1)	y (operand2)
(1)	-	b	c
(2)	-	c	
(3)	*	(1)	(2)
(4)	=	a	(3)

Indirect Triple

			op	x (operand1)	y (operand2)
(1)		(1)	-	b	c
(2)		(2)	-	c	
(3)		(3)	*	(1)	(2)
(4)		(4)	=	a	(3)

Syntax-directed Translation into Three-address Code

- To translate any construct of a programming language, its syntax structure must be specified.

- Semantic actions should be defined in the production rules of the grammar.

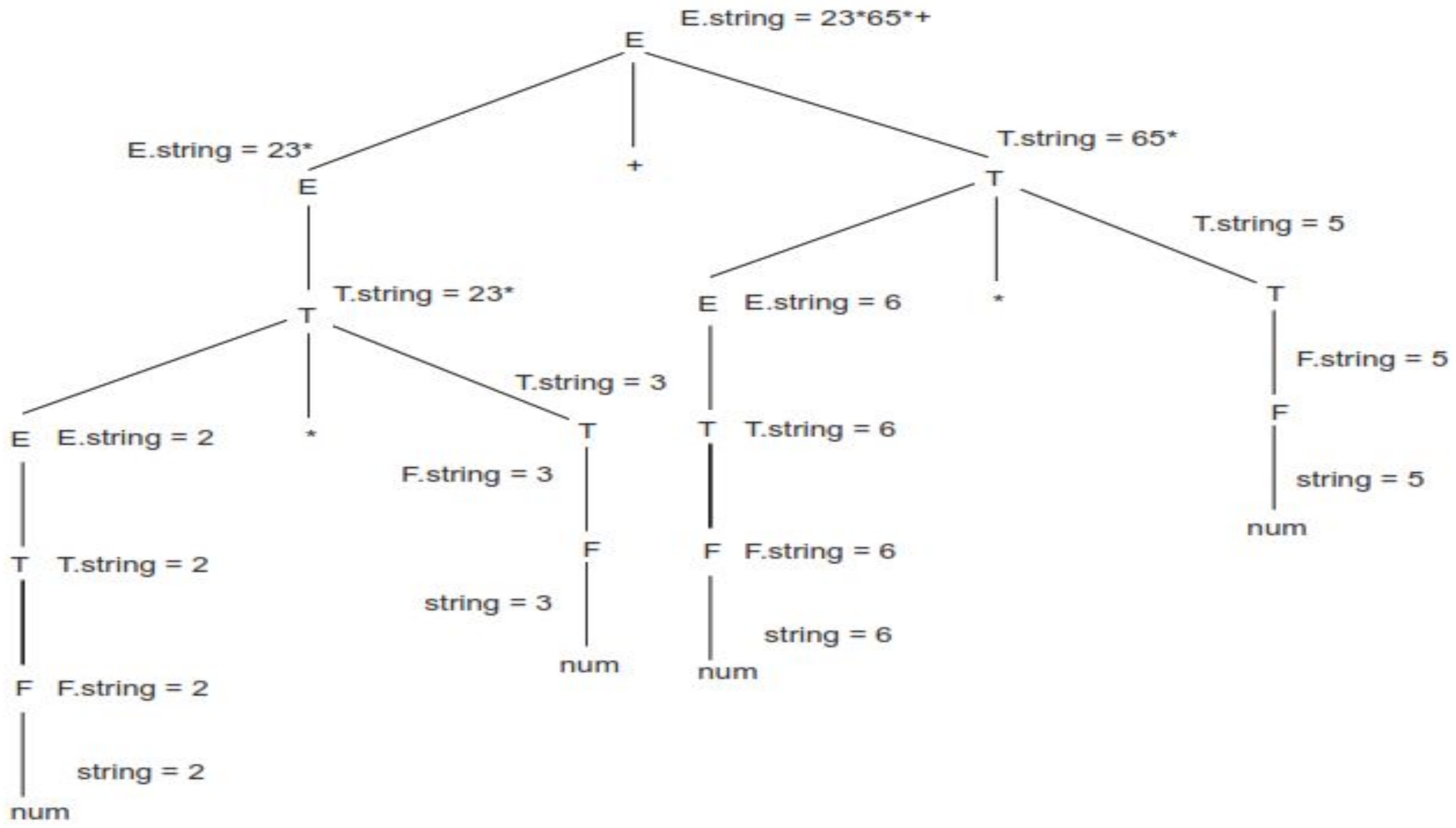
- The syntax-directed translation (SDT) scheme is used to generate the TAC.

Syntax-directed translation scheme to convert **infix to postfix**

Grammar	**Semantic rule**
E1□E2+T	E1.string = E1.string || T.string || ‘+’
E1□ T	E1.string = T.string
T1□T2*F	T1.string = T2.string || F.string || ‘*’
T□F	T.string = F.string
F□(E)	F.string = E.string
F□num	F.string = num.string

Syntax-directed translation scheme to convert infix to postfix

Annotated parse tree for the input string 2*3+6*5



(5+3)* 12 + 7

Grammar	Semantic rule
E1□E2+T	E1.string = E1.string || T.string || ‘+’
E1□ T	E1.string = T.string
T1□T2*F	T1.tring = T2.string || F.string || ‘*’
T□F	T.string = F.string
F□(E)	F.string = E.string
F□num	F.string = num.string

SDTS of Assignment Statement into Three Address Code (TAC)

A□ id = E

E □E+E | E*E | E-E | (E) | id

Production	Semantic Rule
A □id = E	gen(id.place ‘ =’ E . place)

E□ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘+’ E2.place)

E□ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘*’ E2.place)

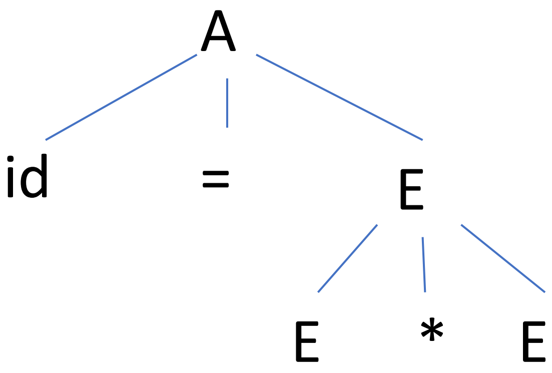
E□ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘-’ E2.place)

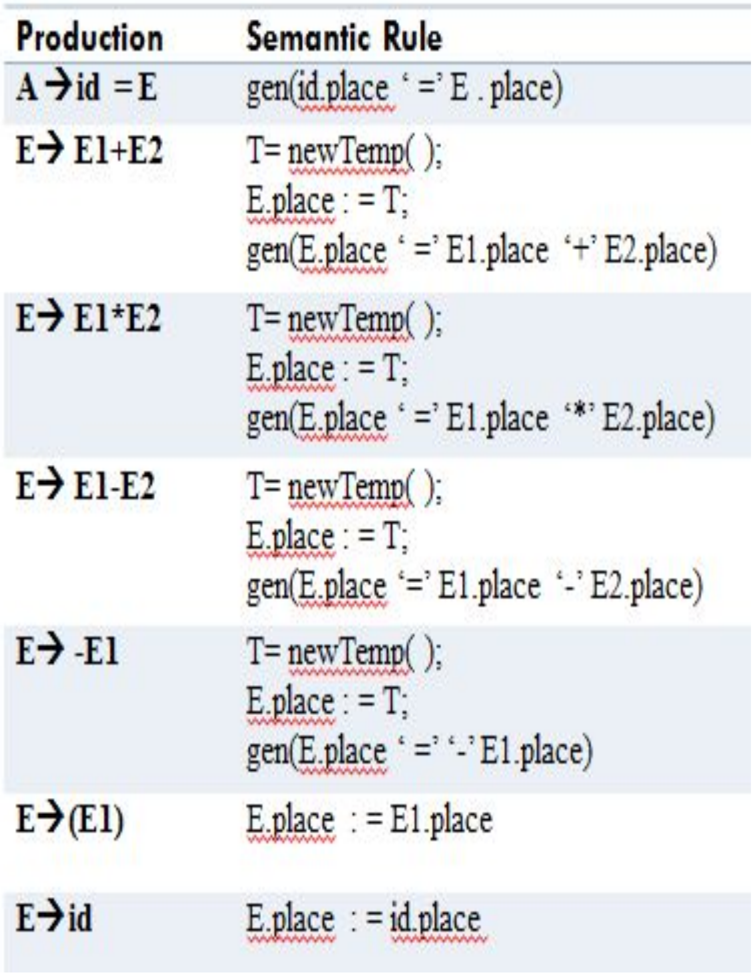
E□ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ ‘-’ E1.place)

E□(E1)	E.place : = E1.place
E□id	E.place : = id.place

Example 1

p=(q+r)*(s+t)





Production	Semantic Rule
A→id =E	gen(id.place ‘ =’ E . place)

E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘+’ E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘*’ E2.place)

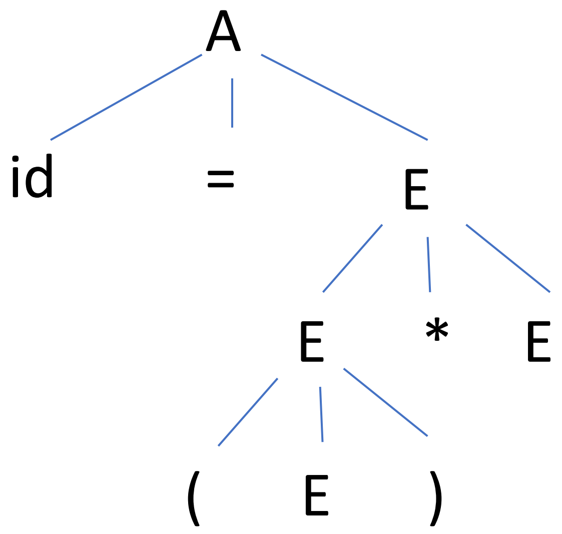
E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘-’ E2.place)

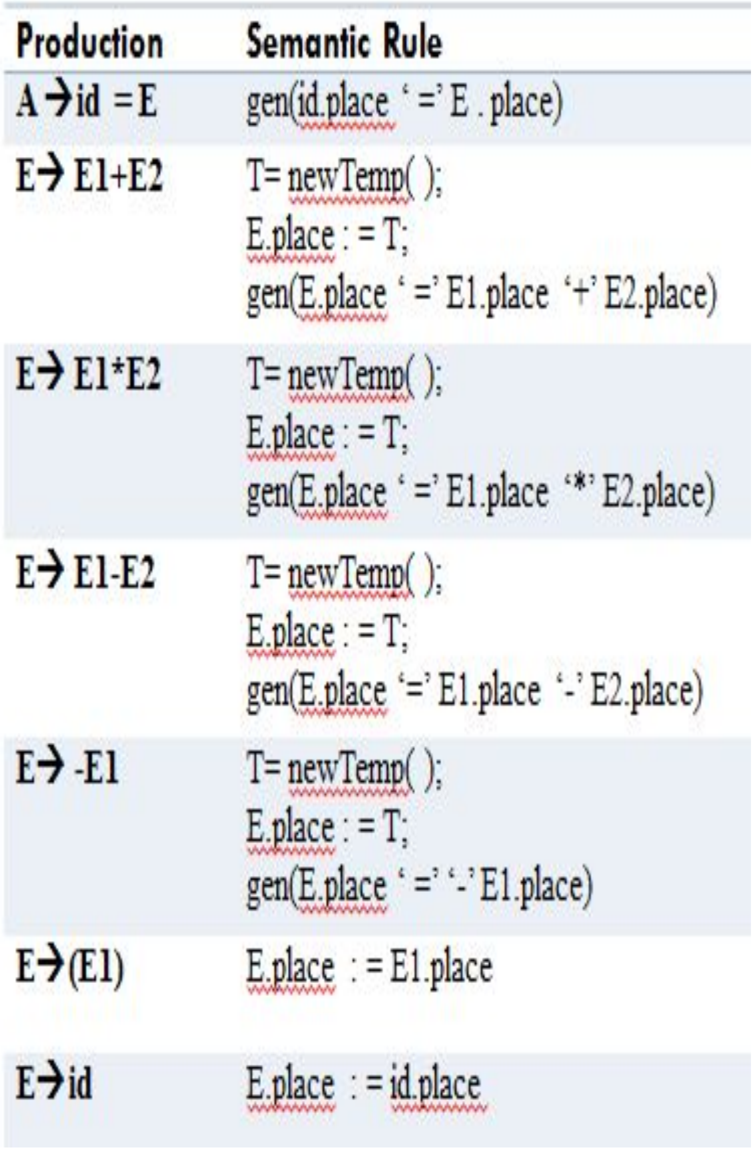
E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ ‘-’ E1.place)

E→(E1)	E.place : = E1.place
E→id	E.place : = id.place

Example 1

p=(q+r)*(s+t)





Production	Semantic Rule
A→id =E	gen(id.place ‘=’ E . place)

E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘+’ E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘*’ E2.place)

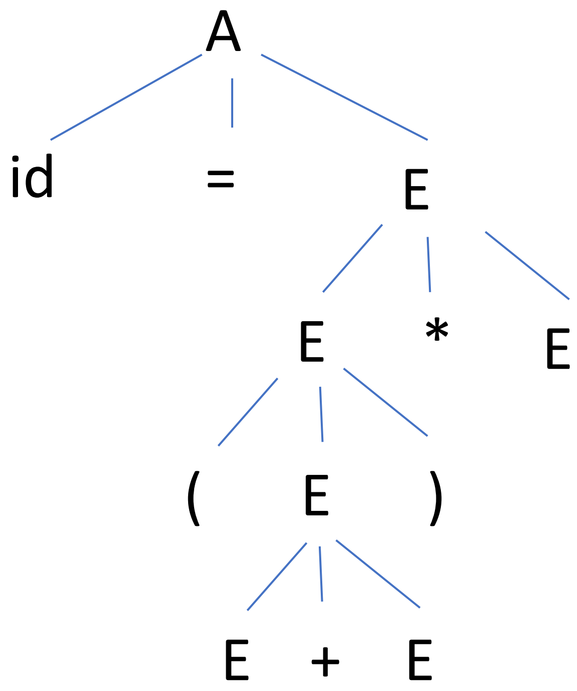
E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘-’ E2.place)

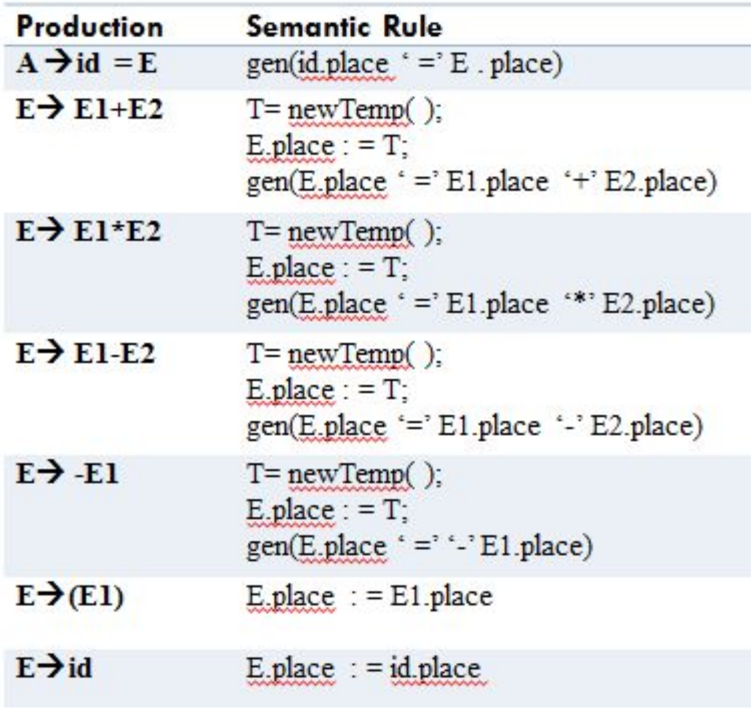
E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ ‘-’ E1.place)

E→(E1)	E.place : = E1.place
E→id	E.place : = id.place

Example 1

p=(q+r)*(s+t)





Production	Semantic Rule
A→id =E	gen(id.place ‘ =’ E . place)

E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘+’ E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘*’ E2.place)

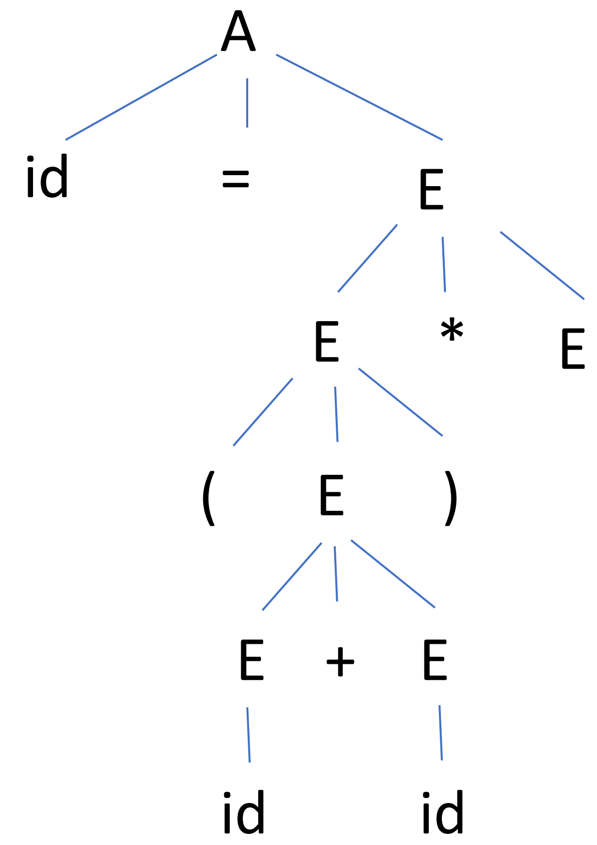
E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘-’ E2.place)

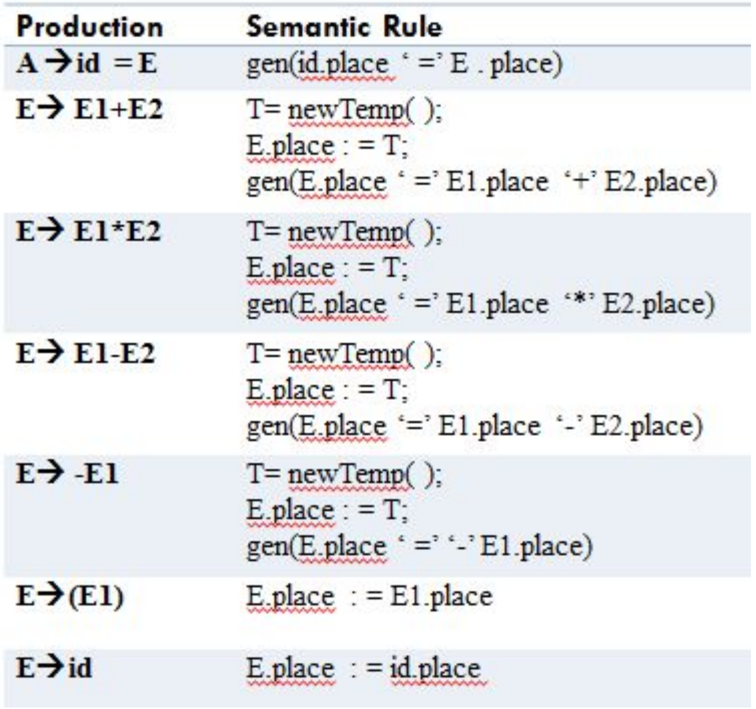
E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ ‘-’ E1.place)

E→(E1)	E.place : = E1.place
E→id	E.place : = id.place

Example 1

p=(q+r)*(s+t)





Production	Semantic Rule
A→id =E	gen(id.place ‘ =’ E . place)

E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘+’ E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘*’ E2.place)

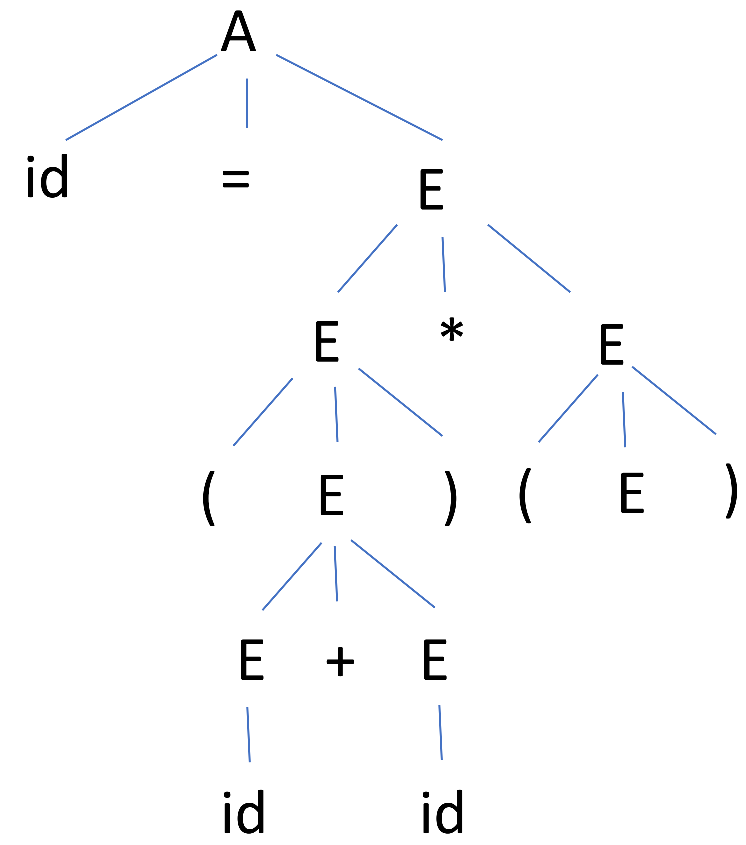
E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘-’ E2.place)

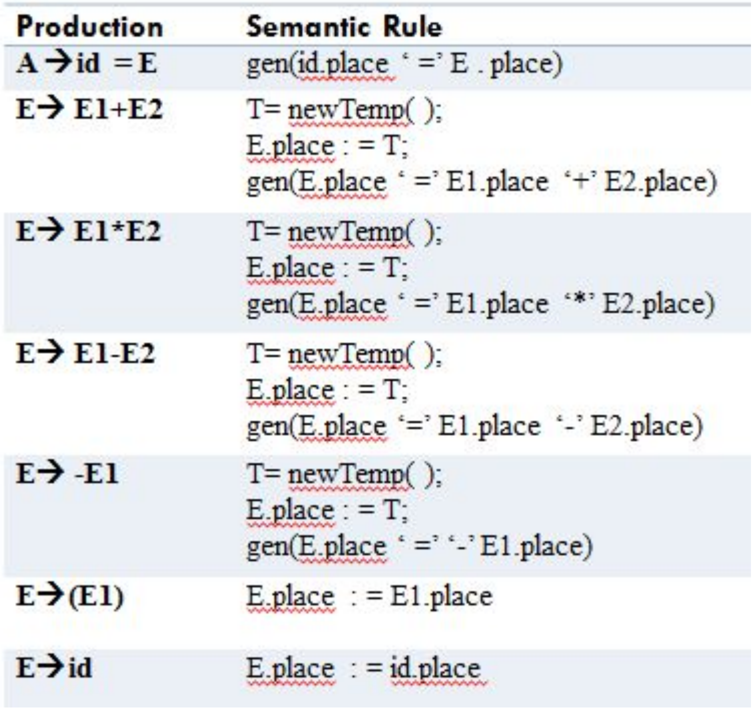
E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ ‘-’ E1.place)

E→(E1)	E.place : = E1.place
E→id	E.place : = id.place

Example 1

p=(q+r)*(s+t)





Production	Semantic Rule
A→id =E	gen(id.place ‘ =’ E . place)

E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘+’ E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘*’ E2.place)

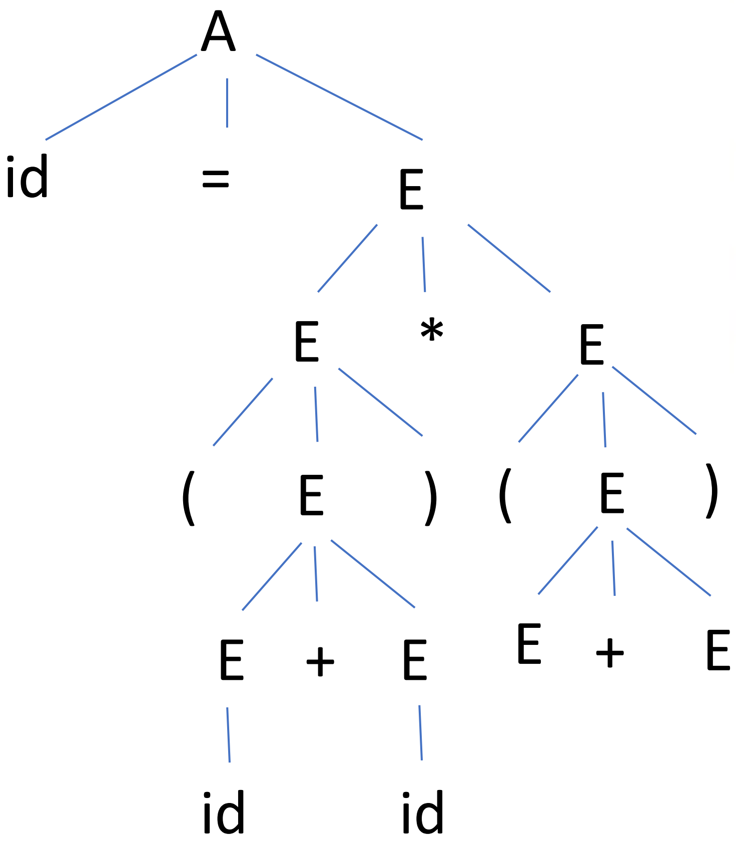
E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘-’ E2.place)

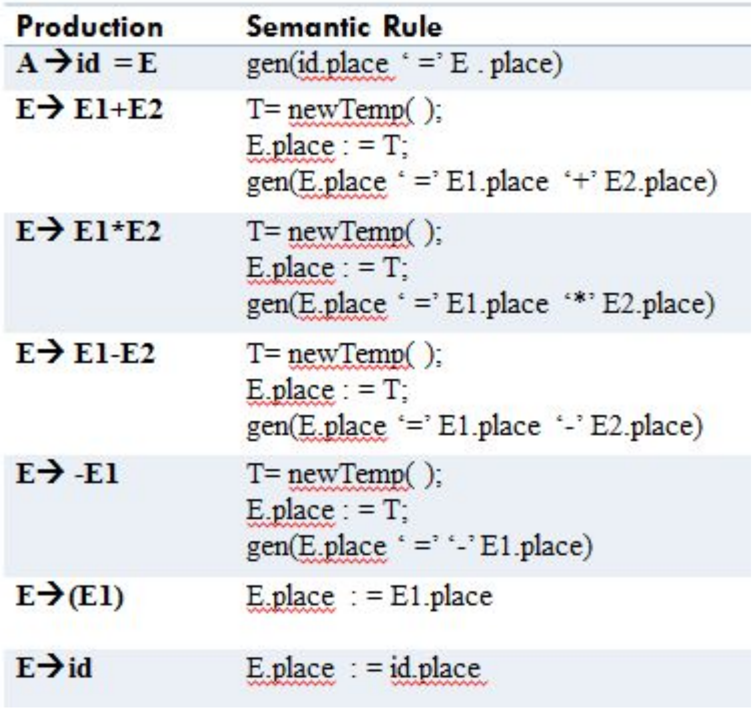
E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ ‘-’ E1.place)

E→(E1)	E.place : = E1.place
E→id	E.place : = id.place

Example 1

p=(q+r)*(s+t)





Production	Semantic Rule
A→id =E	gen(id.place ‘ =’ E . place)

E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘+’ E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘*’ E2.place)

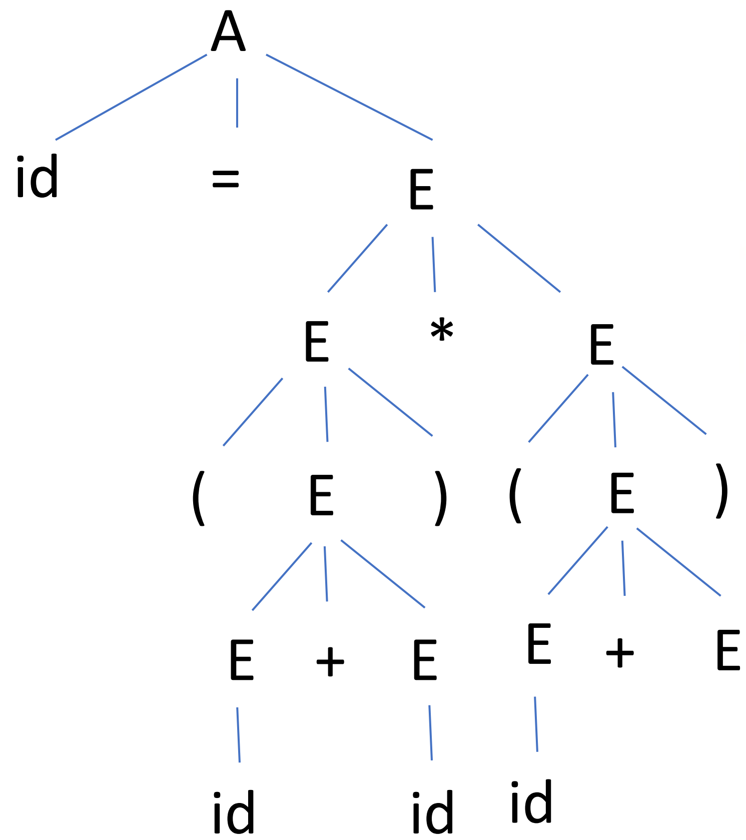
E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘-’ E2.place)

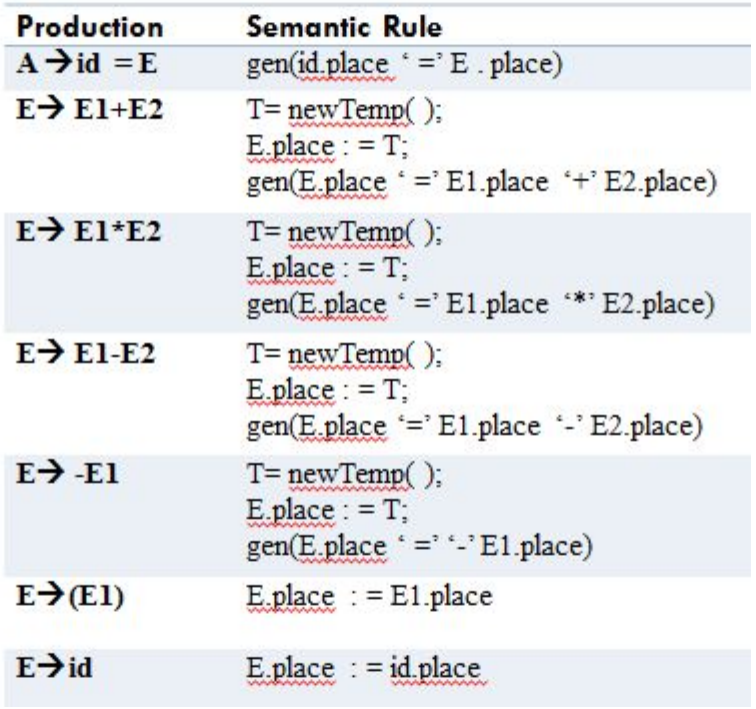
E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ ‘-’ E1.place)

E→(E1)	E.place : = E1.place
E→id	E.place : = id.place

Example 1

p=(q+r)*(s+t)





Production	Semantic Rule
A→id =E	gen(id.place ‘ =’ E . place)

E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘+’ E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘*’ E2.place)

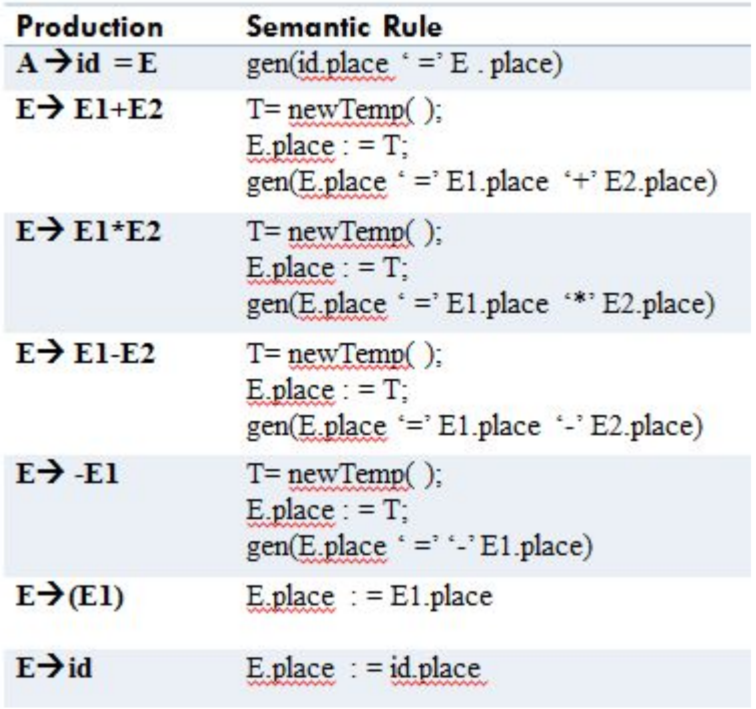
E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘-’ E2.place)

E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ ‘-’ E1.place)

E→(E1)	E.place : = E1.place
E→id	E.place : = id.place

Example 1

p=(q+r)*(s+t)



Production	Semantic Rule
A→id =E	gen(id.place ‘ =’ E . place)

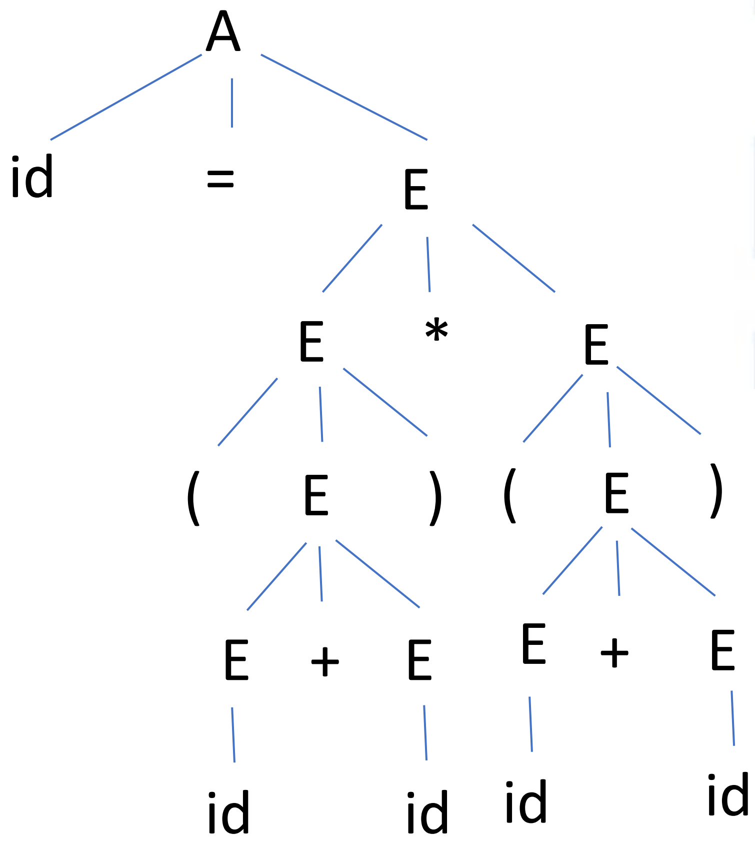
E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘+’ E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘*’ E2.place)

E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘-’ E2.place)

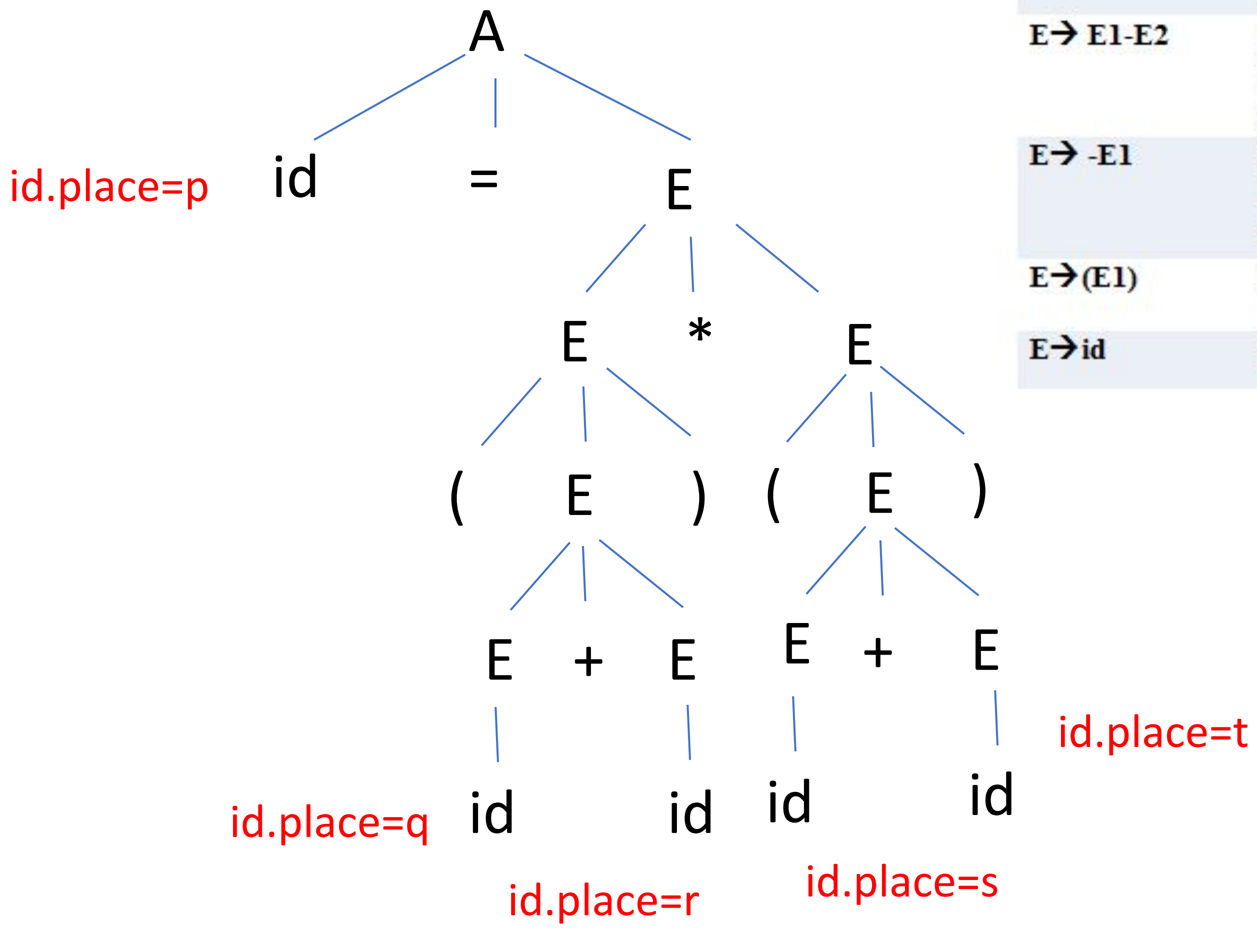
E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ ‘-’ E1.place)

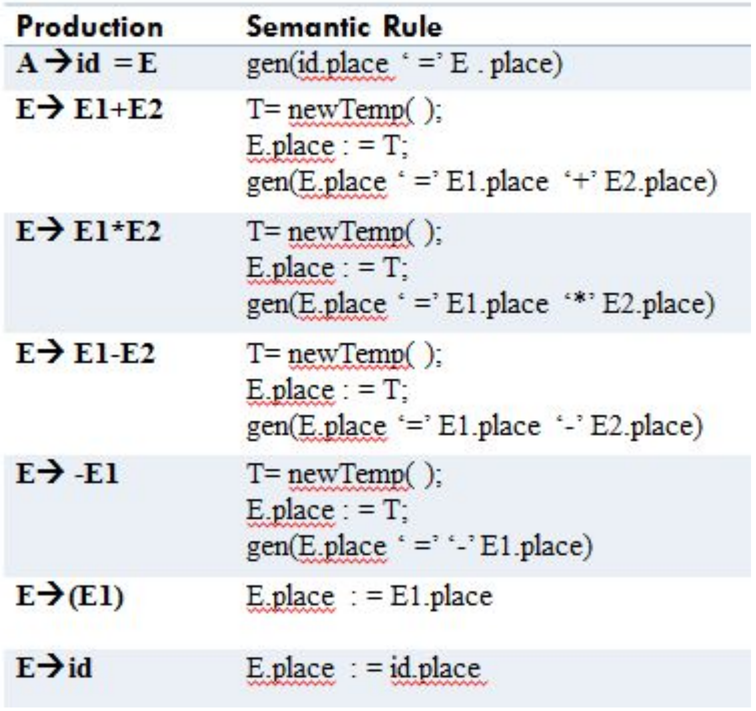
E→(E1)	E.place : = E1.place
E→id	E.place : = id.place



Example 1

p=(q+r)*(s+t)





Production	Semantic Rule
A→id =E	gen(id.place ‘ =’ E . place)

E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘+’ E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘*’ E2.place)

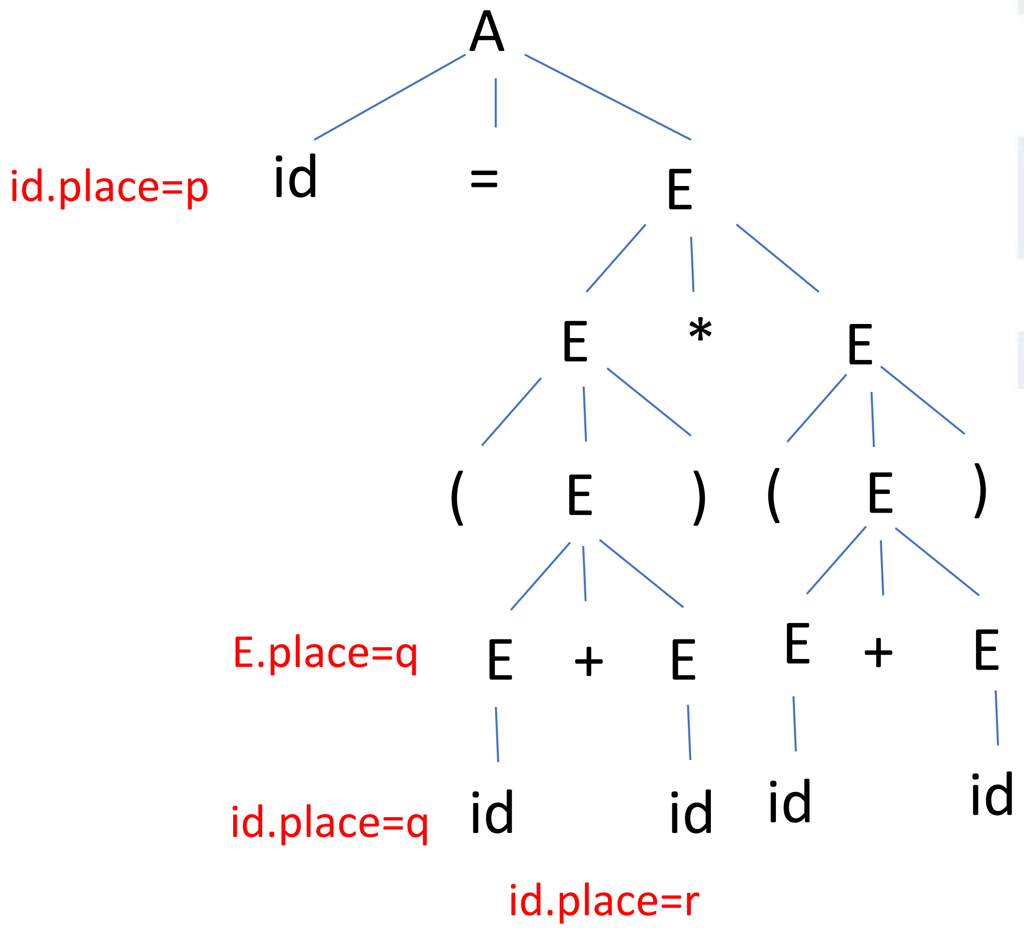
E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘-’ E2.place)

E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ ‘-’ E1.place)

E→(E1)	E.place : = E1.place
E→id	E.place : = id.place

Example 1

p=(q+r)*(s+t)

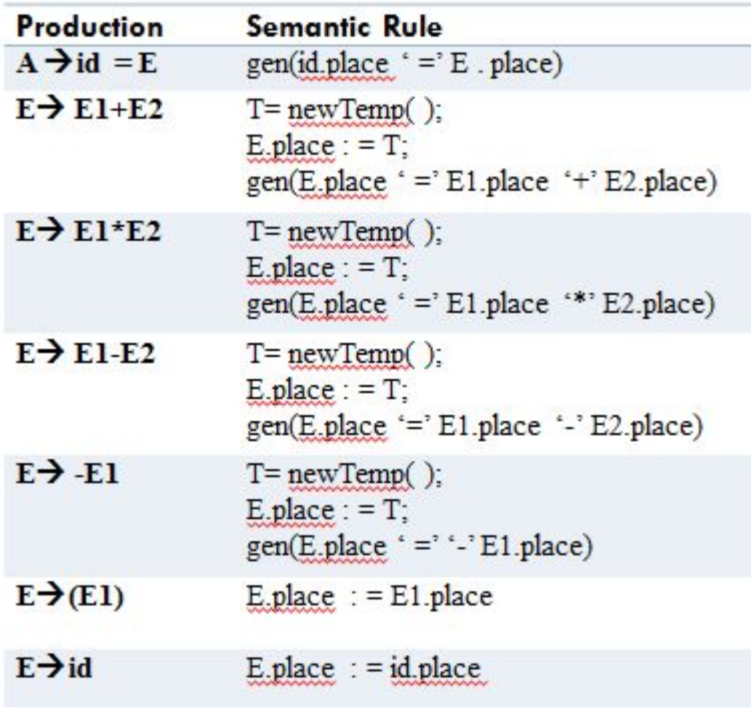


id.place=p

E.place=q

id.place=q

id.place=r



Production	Semantic Rule
A→id =E	gen(id.place ‘ =’ E . place)

E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘+’ E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘*’ E2.place)

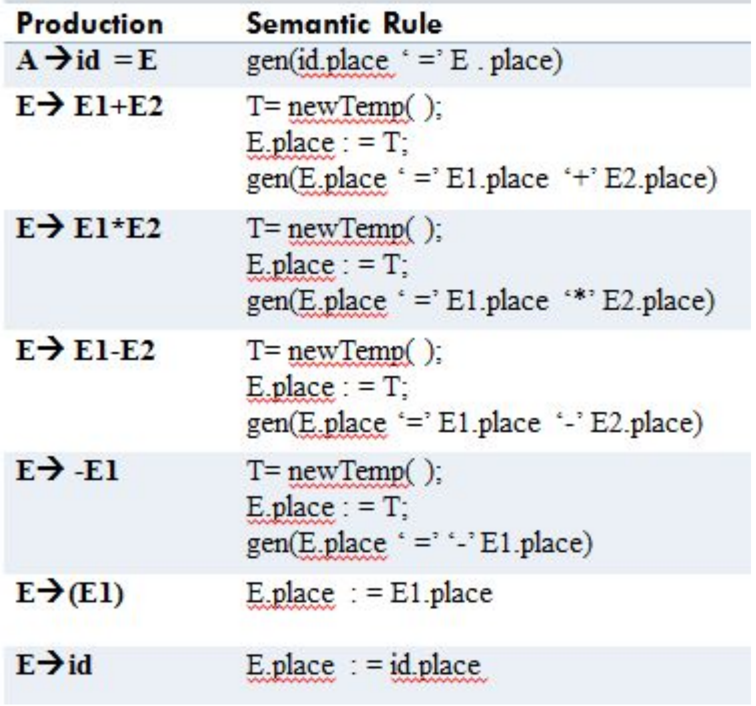
E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘-’ E2.place)

E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ ‘-’ E1.place)

E→(E1)	E.place : = E1.place
E→id	E.place : = id.place

Example 1

p=(q+r)*(s+t)



Production	Semantic Rule
A→id =E	gen(id.place ‘ =’ E . place)

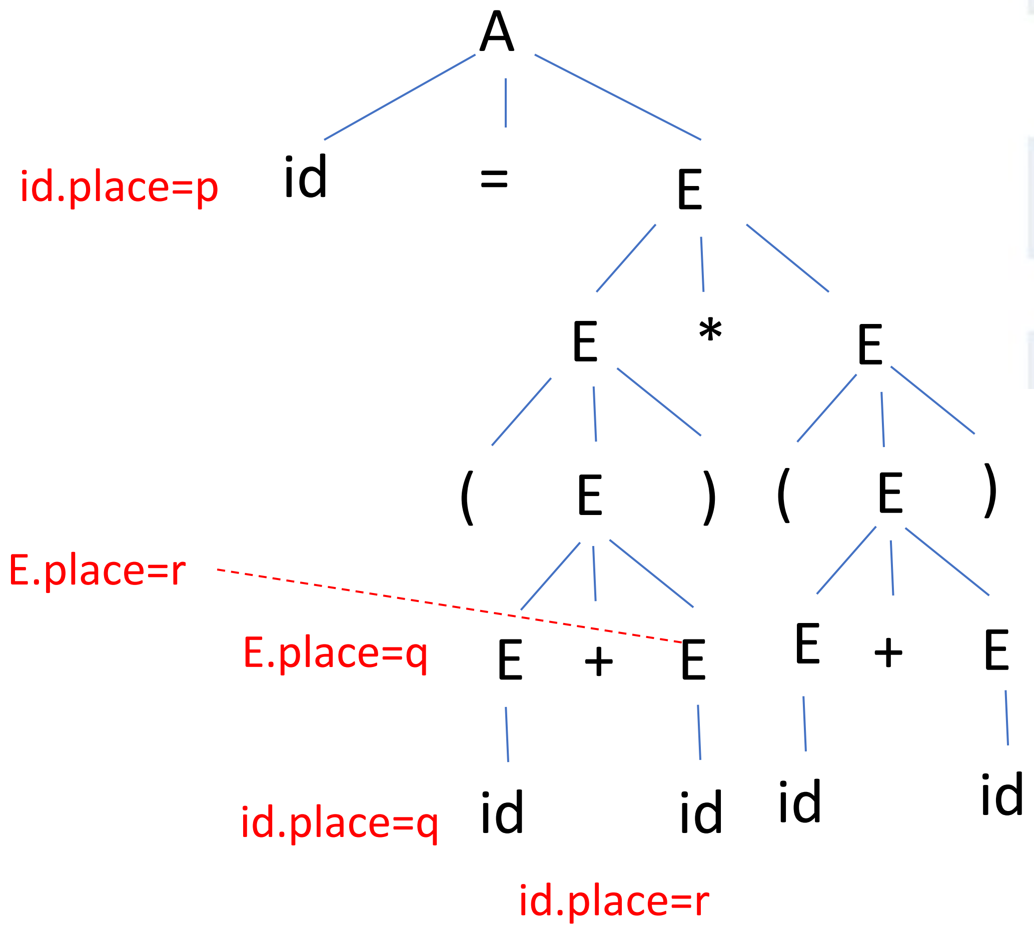
E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘+’ E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘*’ E2.place)

E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘-’ E2.place)

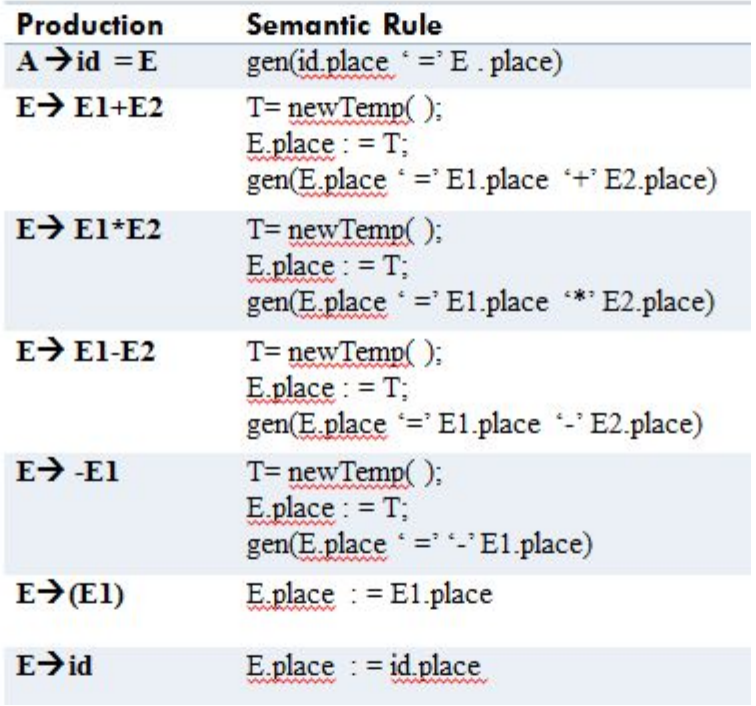
E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ ‘-’ E1.place)

E→(E1)	E.place : = E1.place
E→id	E.place : = id.place



Example 1

p=(q+r)*(s+t)



Production	Semantic Rule
A→id =E	gen(id.place ‘ =’ E . place)

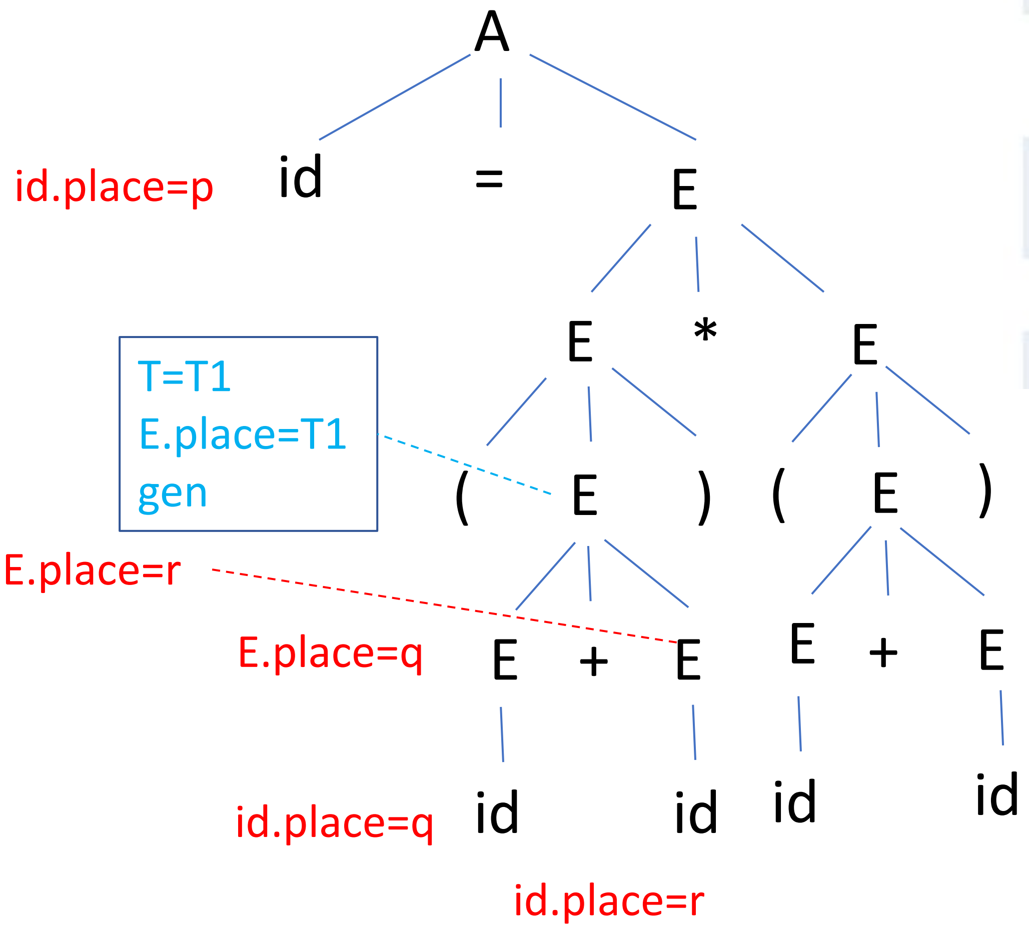
E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘+’ E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘*’ E2.place)

E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘-’ E2.place)

E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ ‘-’ E1.place)

E→(E1)	E.place : = E1.place
E→id	E.place : = id.place



id.place=p

T=T1
E.place=T1
gen

E.place=r

E.place=q

id.place=q

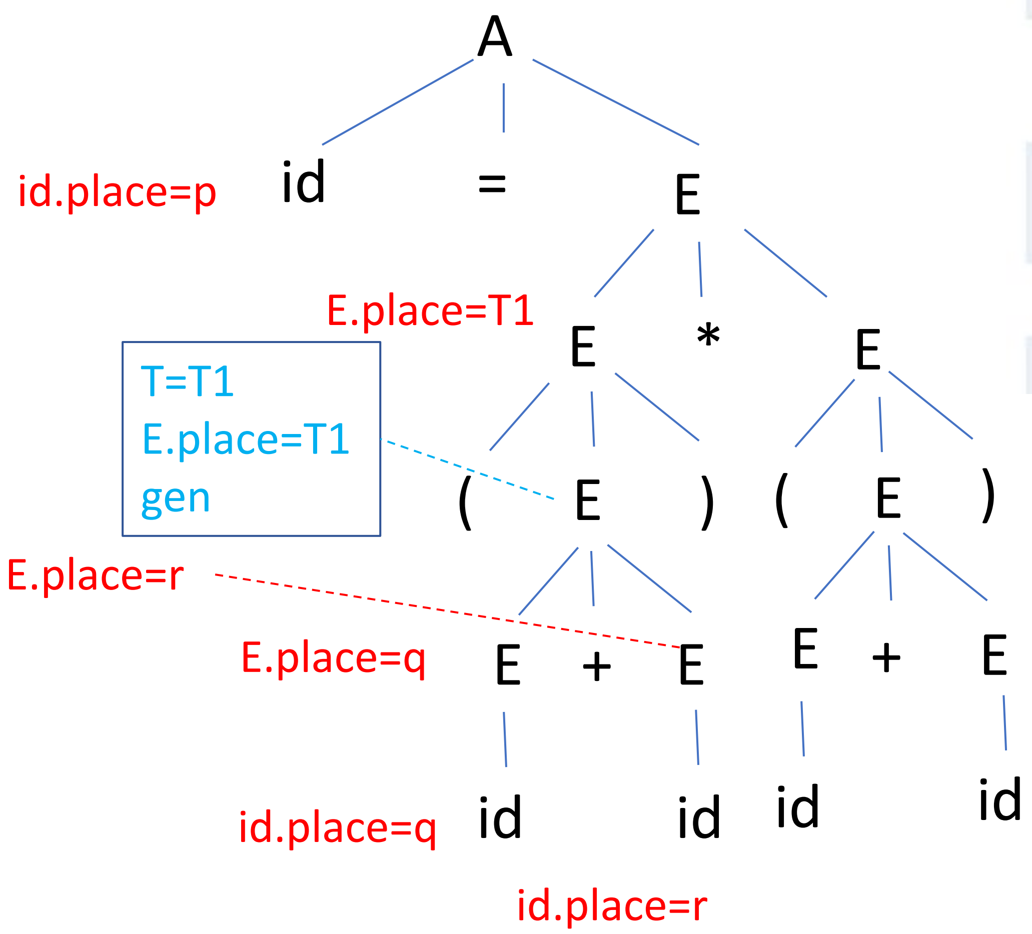
id.place=r

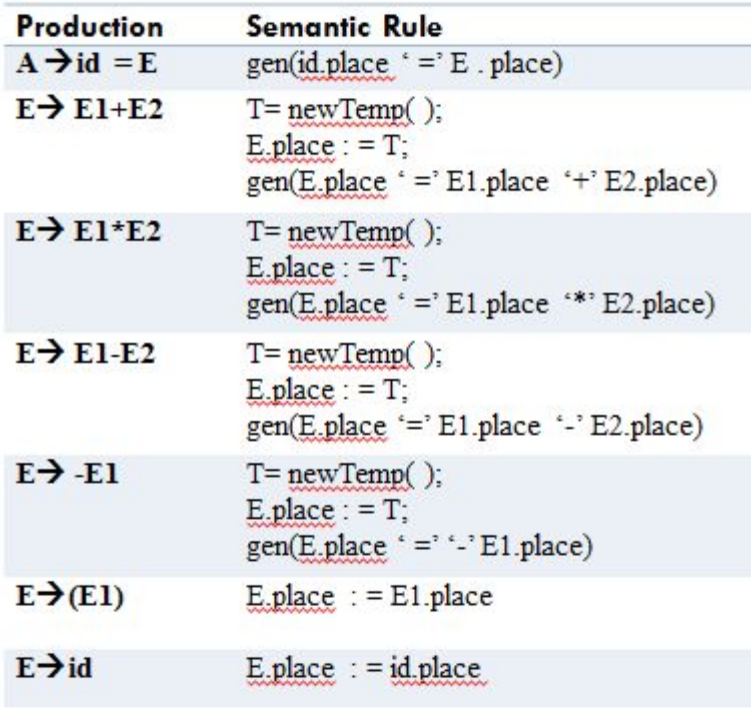
Three address code:

100) T1=q+r

Example 1

p=(q+r)*(s+t)





Production	Semantic Rule
A→id =E	gen(id.place ‘ =’ E . place)

E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘+’ E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘*’ E2.place)

E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘-’ E2.place)

E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ ‘-’ E1.place)

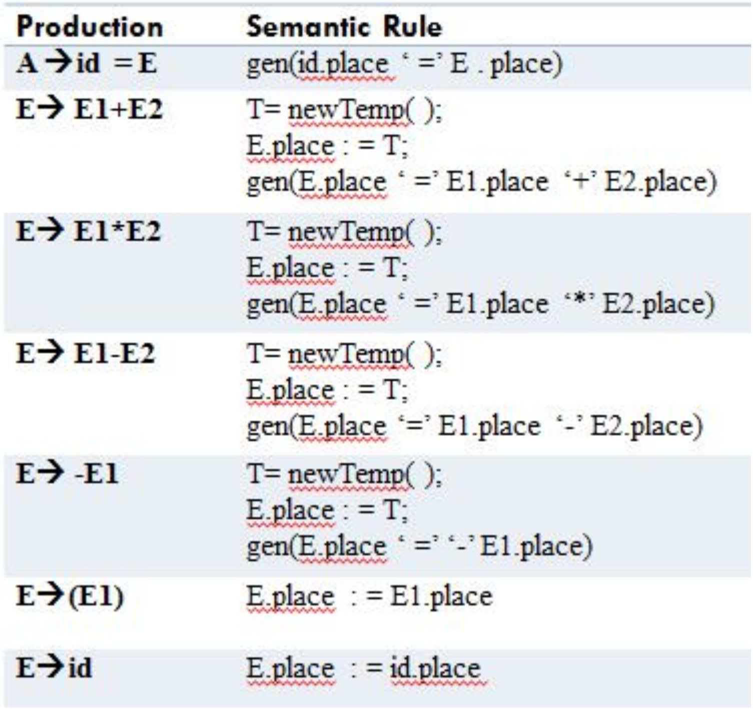
E→(E1)	E.place : = E1.place
E→id	E.place : = id.place

Three address code:

100) T1=q+r

Example 1

p=(q+r)*(s+t)



Production	Semantic Rule
A→id =E	gen(id.place ‘ =’ E . place)

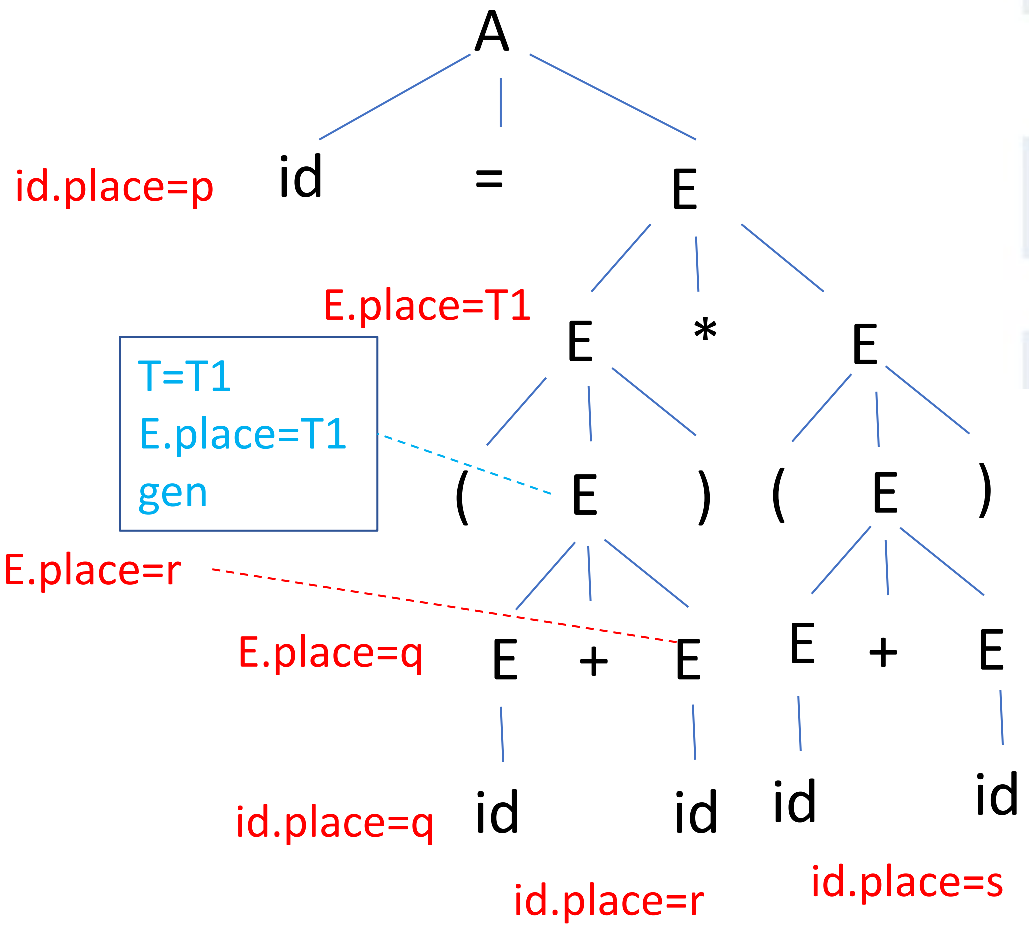
E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘+’ E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘*’ E2.place)

E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘-’ E2.place)

E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ ‘-’ E1.place)

E→(E1)	E.place : = E1.place
E→id	E.place : = id.place

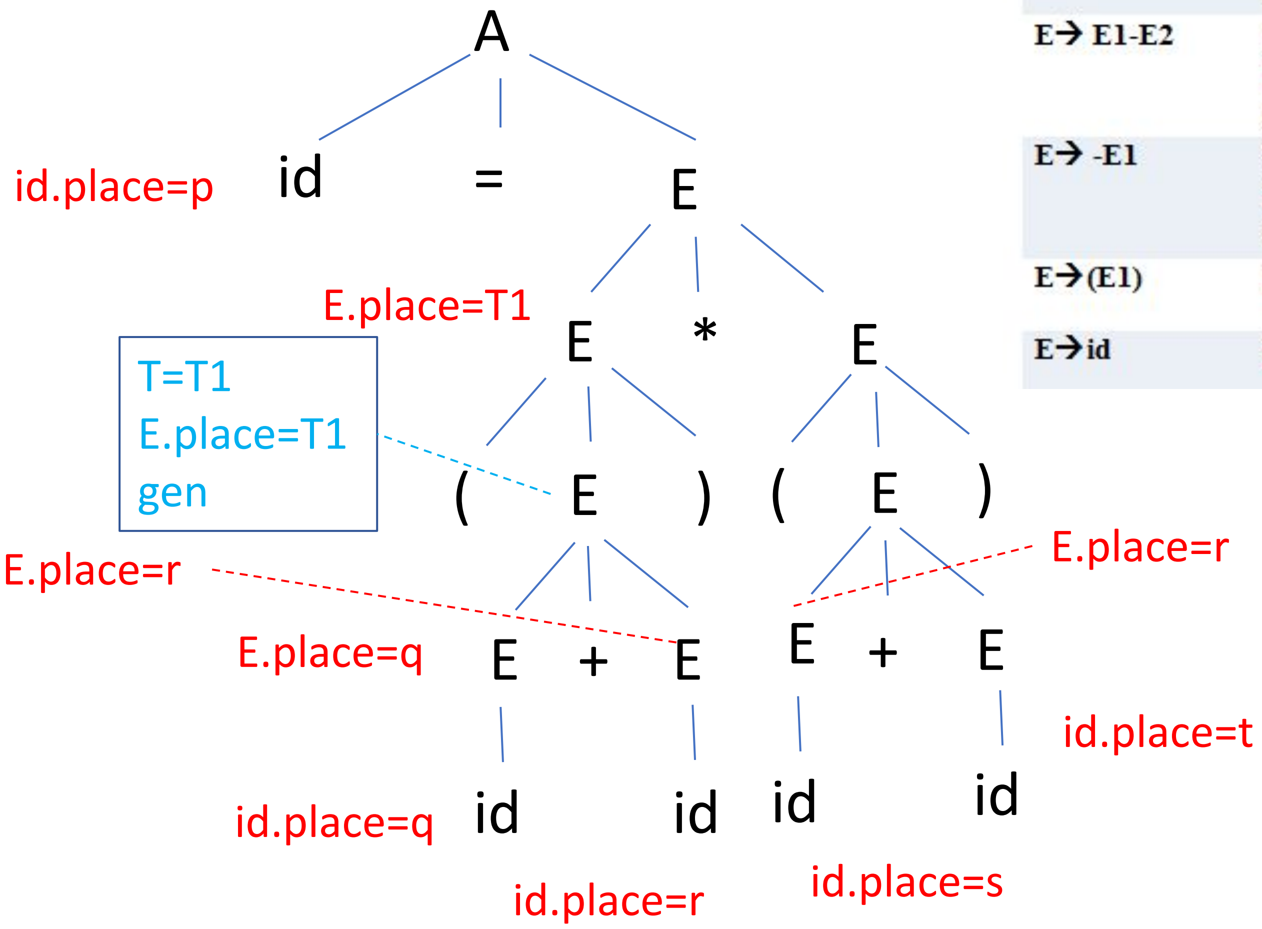


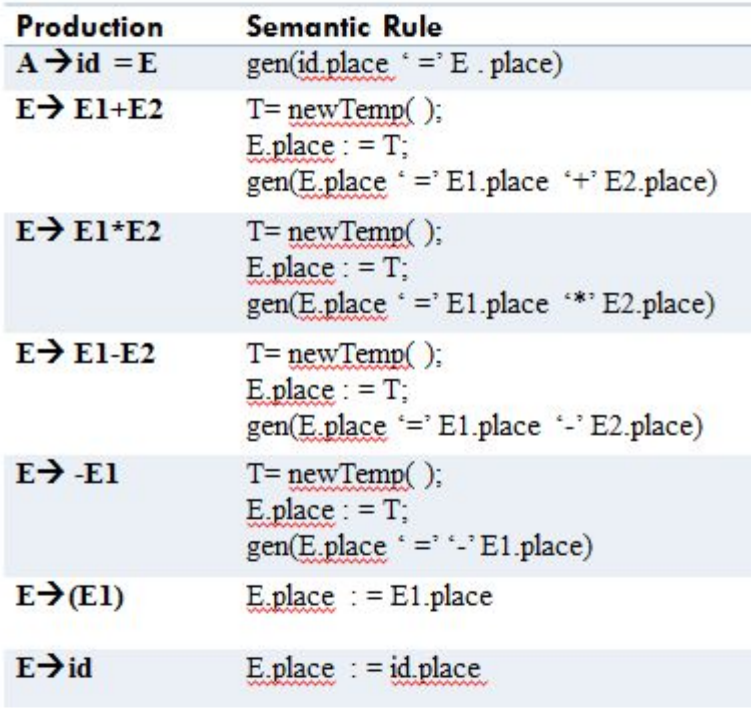
Three address code:

100) T1=q+r

Example 1

p=(q+r)*(s+t)





Production	Semantic Rule
A→id =E	gen(id.place '=' E . place)

E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place '=' E1.place '+' E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place '=' E1.place '*' E2.place)

E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place '=' E1.place '-' E2.place)

E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place '=' '-' E1.place)

E→(E1)	E.place : = E1.place
E→id	E.place : = id.place

Three address code:

100) T1=q+r

Example 1

p=(q+r)*(s+t)

Production	Semantic Rule
A→id =E	gen(id.place ‘ =’ E . place)

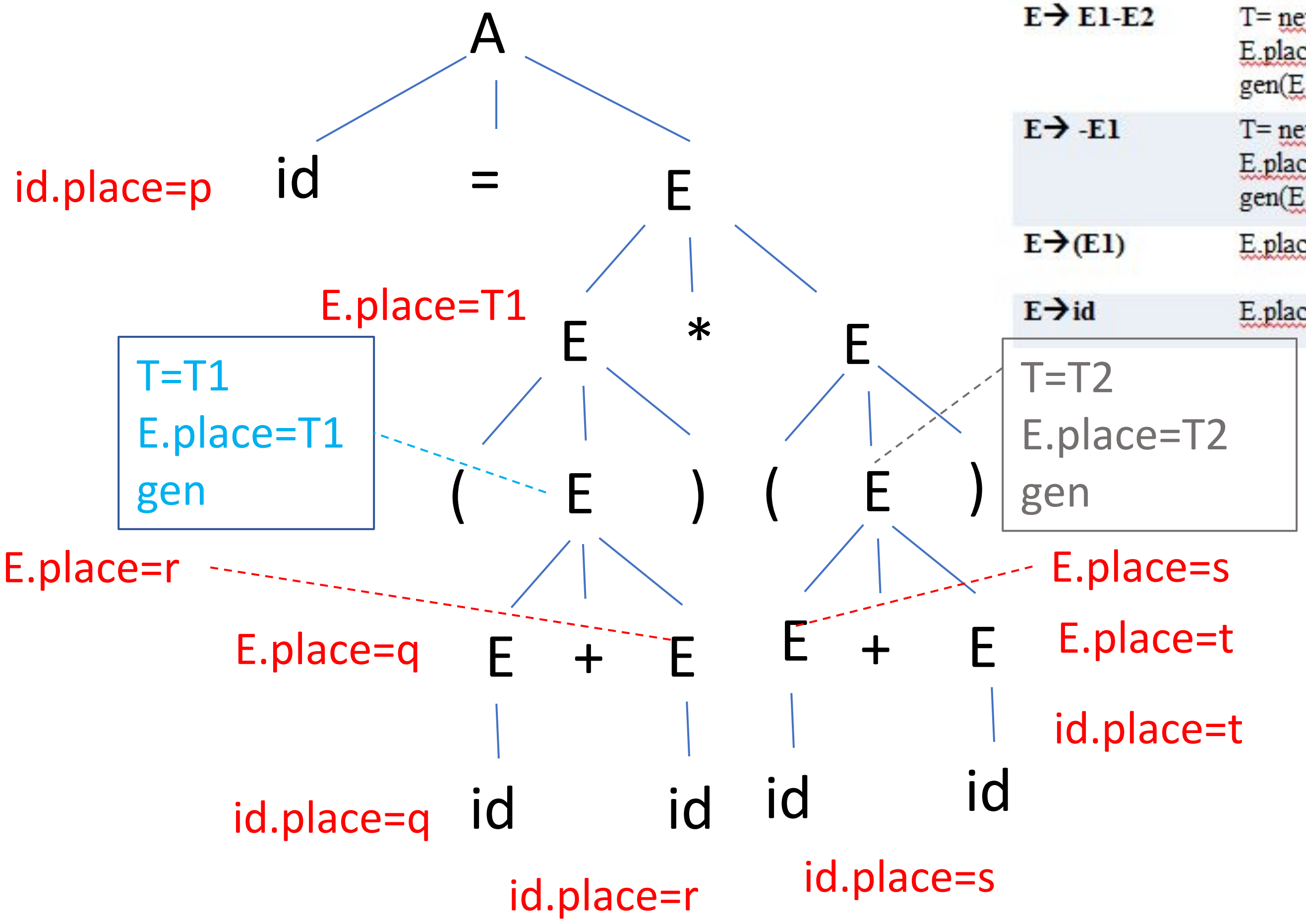
E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘+’ E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘*’ E2.place)

E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘-’ E2.place)

E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ ‘-’ E1.place)

E→(E1)	E.place : = E1.place
E→id	E.place : = id.place



Three address code:

100) T1=q+r

101) T2=s+t

Example 1

p=(q+r)*(s+t)

Production	Semantic Rule
A→id =E	gen(id.place ‘ =’ E . place)

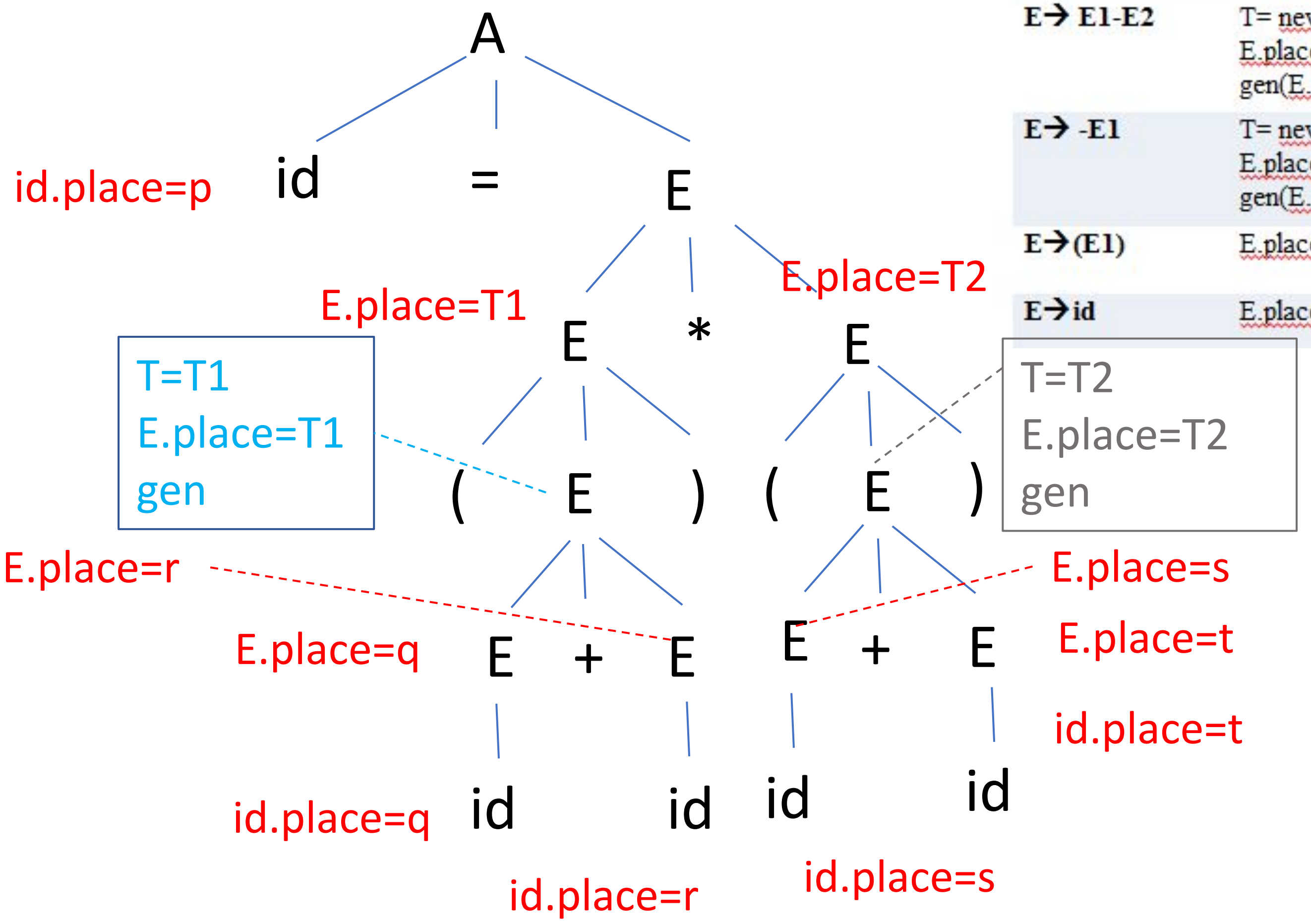
E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘+’ E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘*’ E2.place)

E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘-’ E2.place)

E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ ‘-’ E1.place)

E→(E1)	E.place : = E1.place
E→id	E.place : = id.place



id.place=p

E.place=T1

E.place=T2

T=T1
E.place=T1
gen

T=T2
E.place=T2
gen

E.place=r

E.place=q

E.place=s

E.place=t

id.place=t

id.place=q

id.place=r

id.place=s

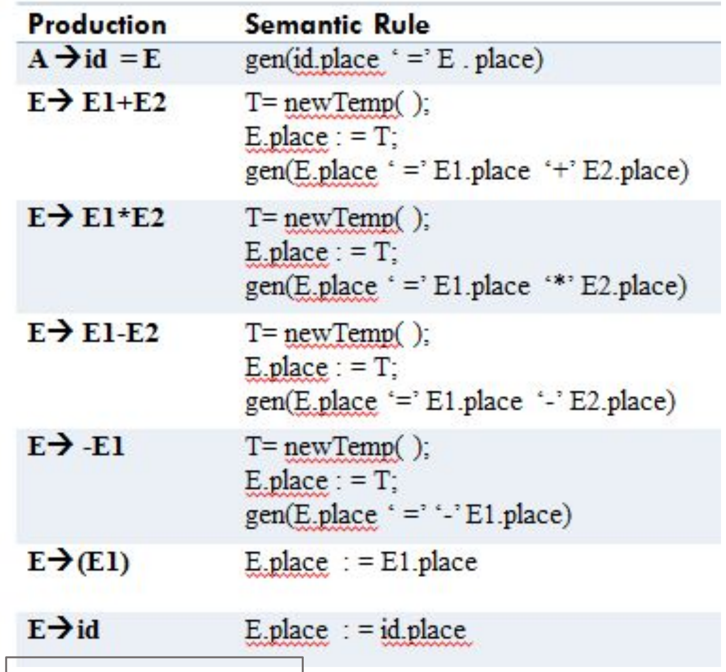
Three address code:

100) T1=q+r

101) T2=s+t

Example 1

p=(q+r)*(s+t)



Production	Semantic Rule
A→id =E	gen(id.place '=' E . place)

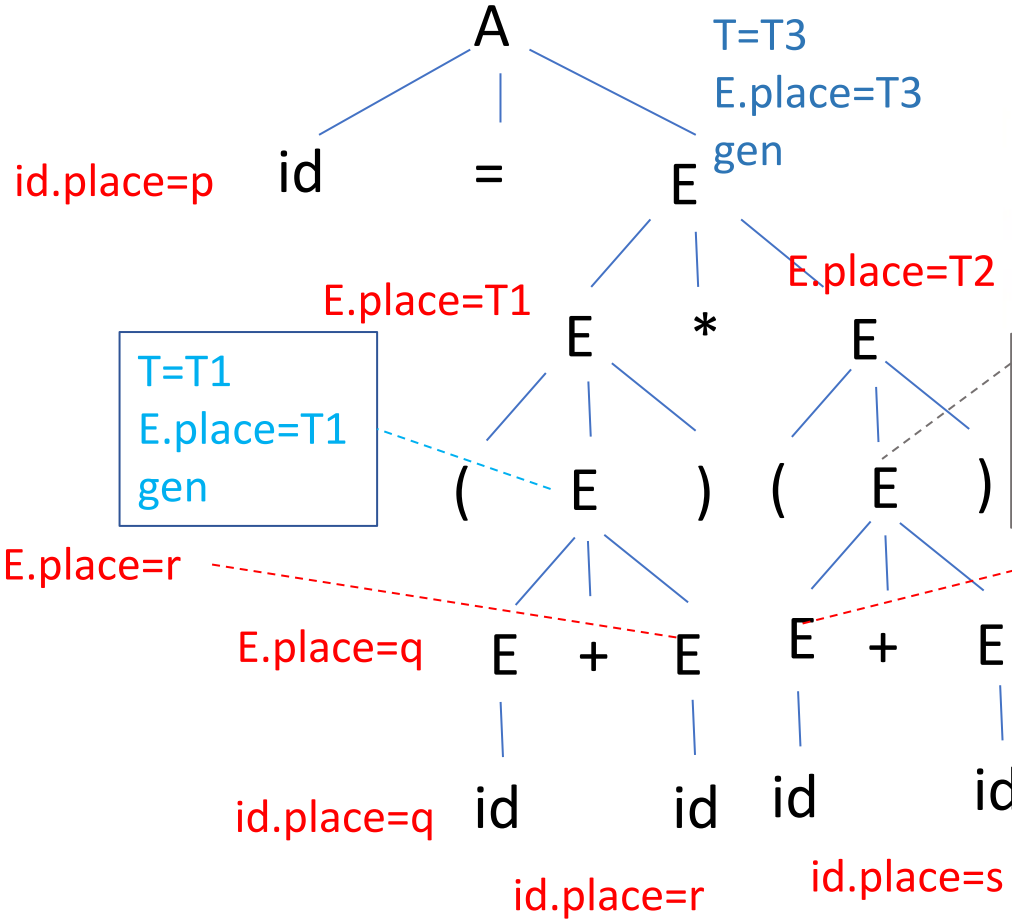
E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place '=' E1.place '+' E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place '=' E1.place '*' E2.place)

E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place '=' E1.place '-' E2.place)

E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place '=' '-' E1.place)

E→(E1)	E.place : = E1.place
E→id	E.place : = id.place



T=T2
E.place=T2
gen

E.place=s

E.place=t

id.place=t

Three address code:

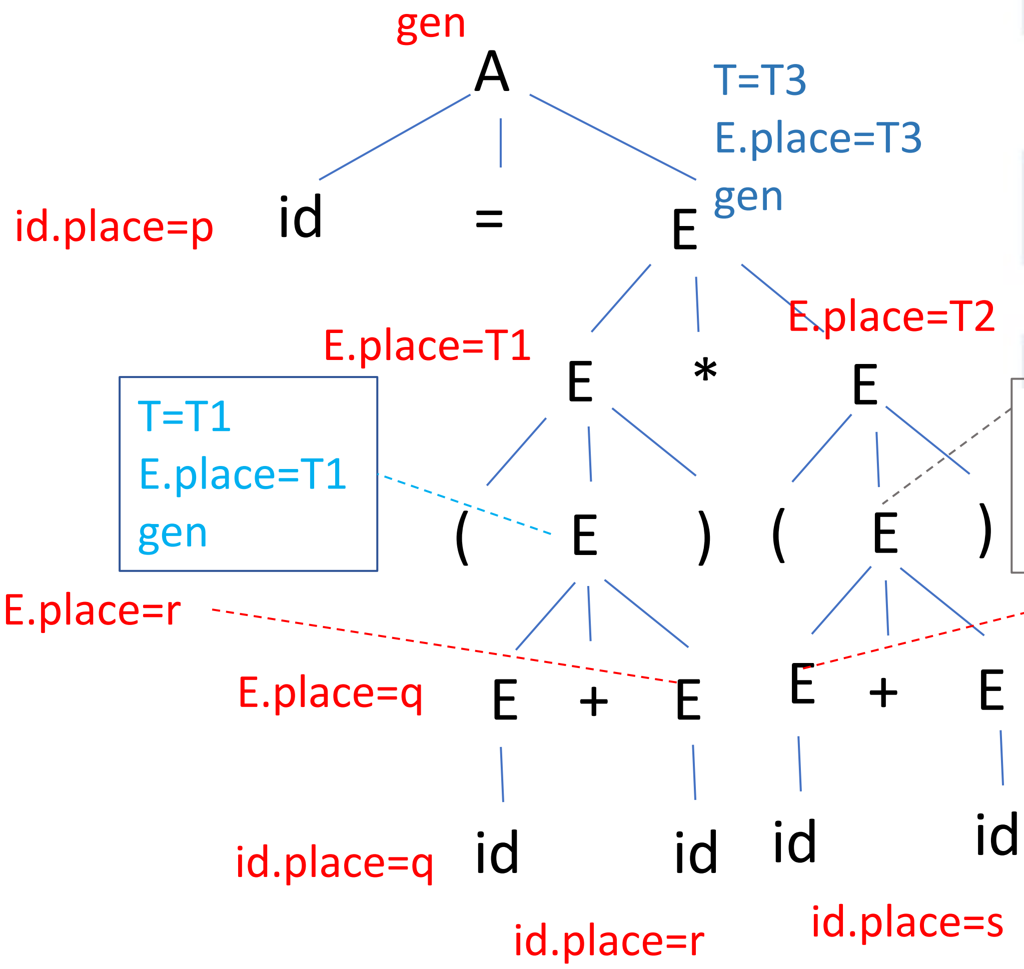
100) T1=q+r

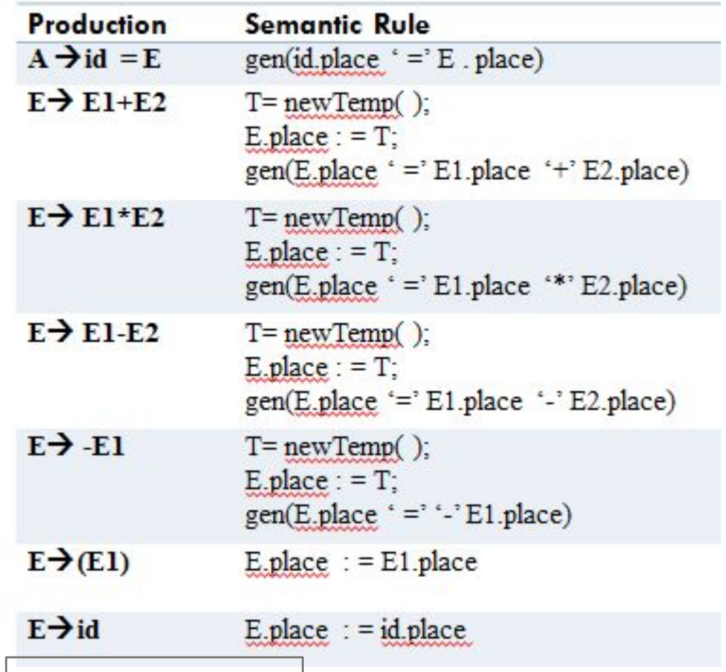
101) T2=s+t

102) T3=T1*T2

Example 1

p=(q+r)*(s+t)





Production	Semantic Rule
A→id =E	gen(id.place ‘ =’ E . place)

E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘+’ E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘*’ E2.place)

E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘-’ E2.place)

E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ ‘-’ E1.place)

E→(E1)	E.place : = E1.place
E→id	E.place : = id.place

Three address code:

100) T1=q+r

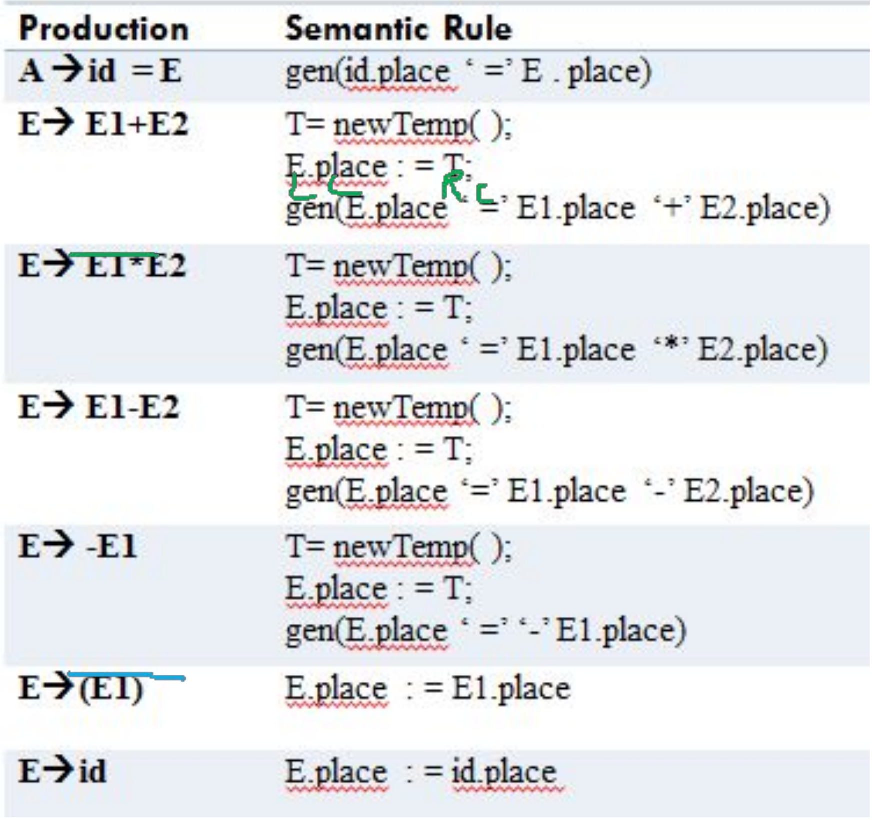
101) T2=s+t

102) T3=T1*T2

103) p=T3

Example 2

a= -c*(d+e)



Production	Semantic Rule
A→id =E	gen(id.place ‘ =’ E . place)

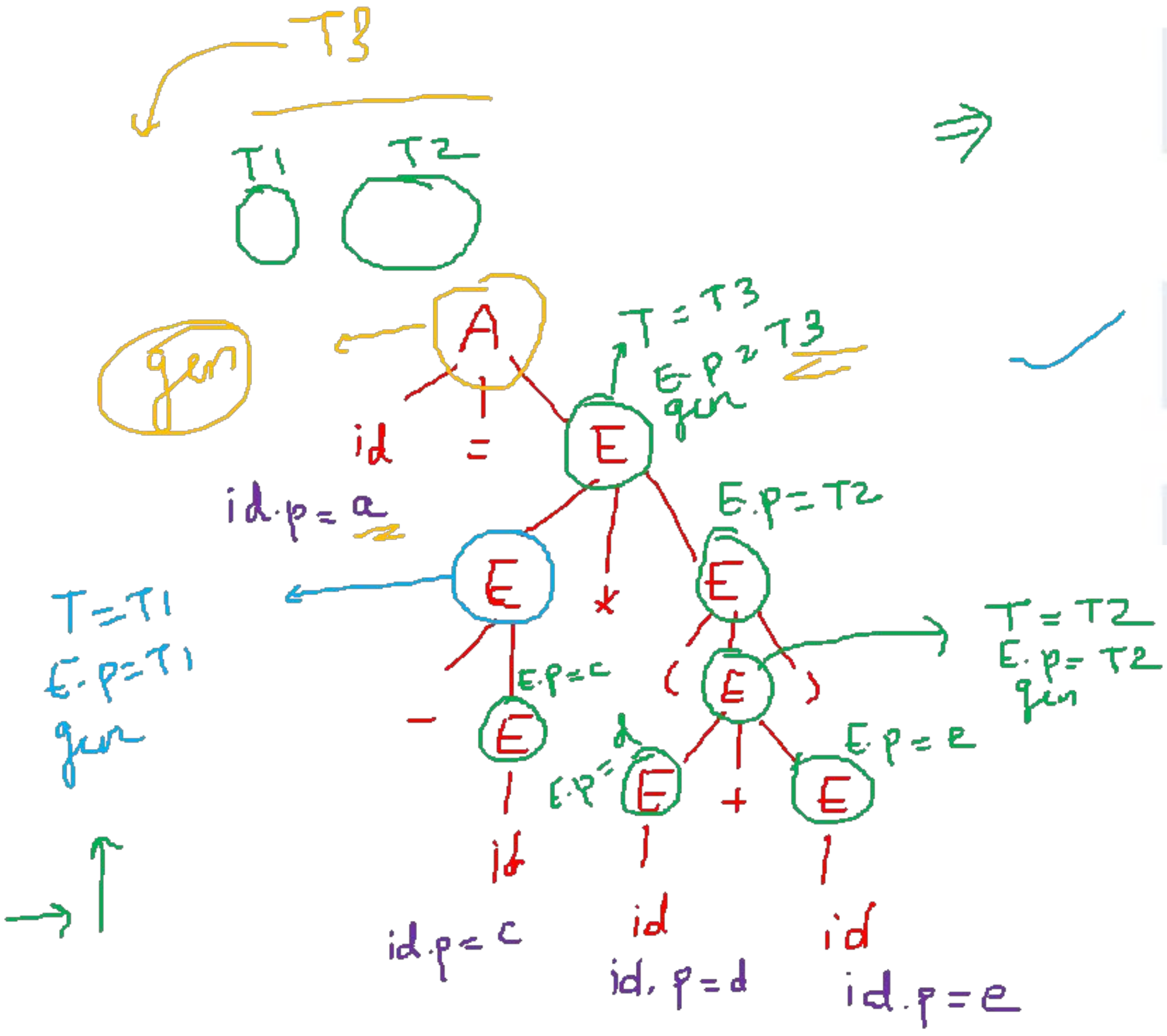
E→ E1+E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘+’ E2.place)

E→ E1*E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ E1.place ‘*’ E2.place)

E→ E1-E2	T= newTemp( );
	E.place : = T;
	gen(E.place ‘=’ E1.place ‘-’ E2.place)

E→ -E1	T= newTemp( );
	E.place : = T;
	gen(E.place ‘ =’ ‘-’ E1.place)

E→(E1)	E.place : = E1.place
E→id	E.place : = id.place



TAC :-

100] T1 = -c

101] T2 = d + e

102] T3 = T1 * T2

103] a = T3

SDTS for Boolean expressions

Numerical Representation

Production	Semantic Rule

E□ id1 relop id2	T= newTemp();
	E.place= T;
	gen(if id1.place relop id2.place goto nextquad+3);
	gen (T=0);
	gen (goto nextquad+2);
	gen(T=1);

E□ E1 or E2	T= newTemp();
	E.place= T;
	gen(T= E1.place or E2.place)

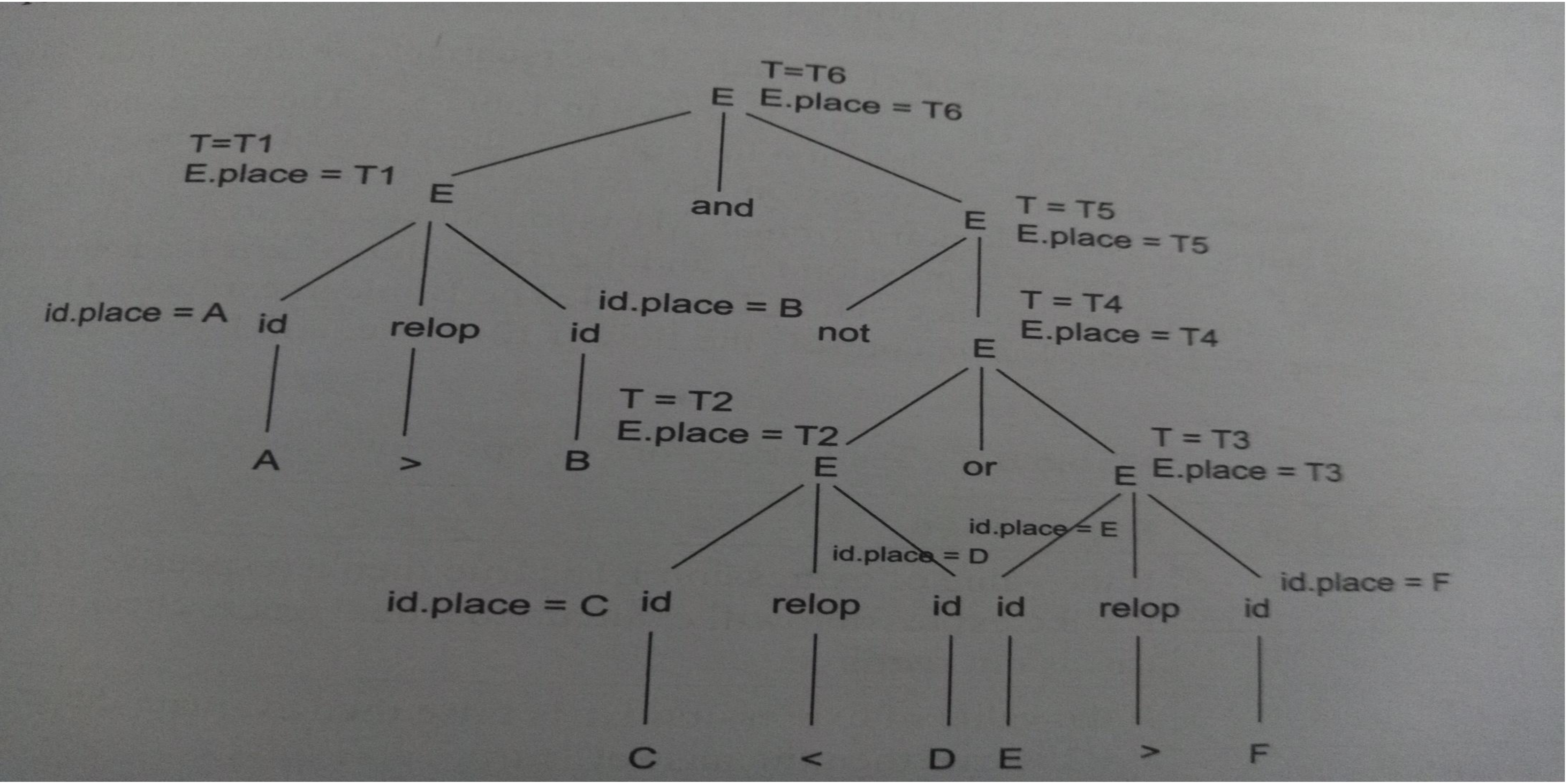
E□ E1 and E2	T= newTemp();
	E.place= T;
	gen(T= E1.place and E2.place)

E□ not E1	T= newTemp();
	E.place= T;
	gen(T= not E1.place)

E□ id	E.place= id.place
	E.code= null

relop □ <	{ relop.val= ‘<‘ }
relop □ >	{ relop.val= ‘>‘ }
relop □ <=	{ relop.val= ‘<=‘ }
relop □ >=	{ relop.val= ‘>=‘ }
relop □ = =	{ relop.val= ‘ = = ‘ }
relop □ ! =	{ relop.val= ‘! =‘ }

A>B and not(C > D or E> F)



TAC generated using SDTs

i)	If A > B goto i+3
i+1)	T1=0
i+2)	goto i+4
i+3)	T1=1
i+4)	If C > D goto i+7
i+5)	T2=0
i+6)	*goto i+8*
i+7)	T2=1
i+8)	If E > F goto i+11
i+9)	T3=0
i+10)	goto i+12
i+11)	T3=1
i+12)	T4= T2 or T3
i+13)	T5= not T4
i+14	T6= T1 and T5

SDTS for Boolean expressions

Short circuit code

Production	Semantic Rule

E→ E1 or M1 E2
M→ €
	Backpatch(E1.false, M.quad);
	E.false= E2.false
	E.True = merge (E1.true, E2.true);
	M.Quad =nextquad

E→ E1 and M1 E2
M→ €
	Backpatch(E1.true, M.quad);
	E.true= E2.true
	E.false = merge (E1,false, E2.false);
	M.Quad =nextquad

E→ not E1	E.True = E1.false;
	E.false=E,true;

SDTS for Boolean expressions

Short circuit code

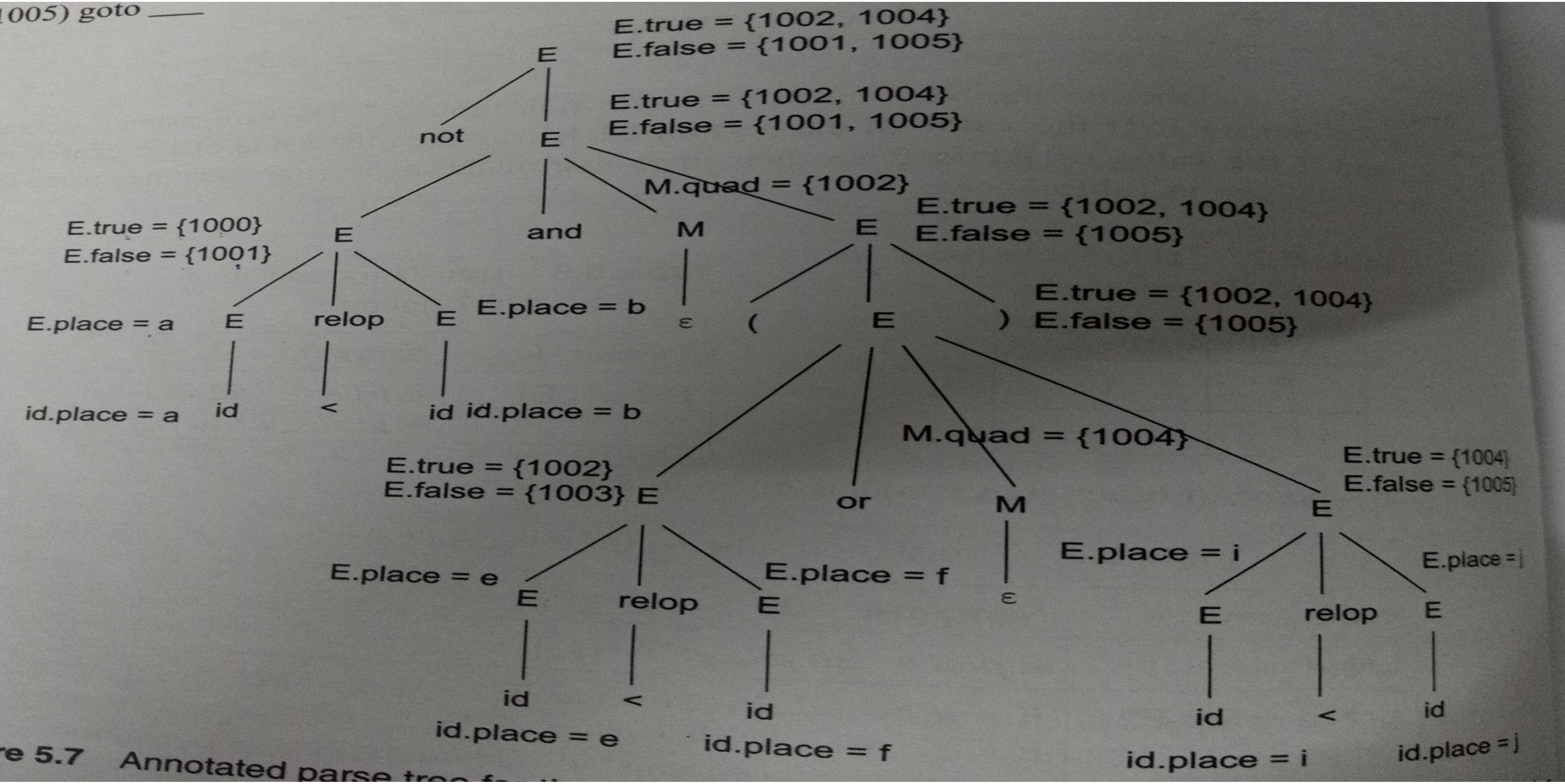
Production	Semantic Rule
E□ id	E.place=id.place;

E□ E1 relop E2	E.true= makelist(nextquad);
	E.false= makelist(nextquad+1);
	gen(if E1.place relop,val E2.place goto ____________);
	gen (goto_________)

E□ (E)	E,true = E1. true;
	E’.false = E1.false;

not(a < b and (e > f or I < j))

005) goto ____



re 5.7 Annotated parse tree fo

TAC generated using SDTs

1000	**If a < b goto 1002**
1001	goto ________________
1002	If e < f goto __________
1003	goto 1004
1004	If I > j goto _________________
1005	goto ______________

SDTS for IF, IF-else

Production	Semantic Rule

S□if E then M S1	backpatch(E.true,M.quad);
	S.next= merge(E.false,S1.next);

S□if E then M1 S1 N else M2 S2	backpatch(E.true,M1.quad);
	backpatch(E.false,M2.quad);
	S.next=merge(S1.next,merge(N.next, S2.next);

S□while M1 E do M2 S1	backpatch(S1.next, M1.quad);
	backpatch(E.true,M2.quad);
	S.next= E.false; gen(‘goto’ M1.quad)

S□do M1 S1 while M2 E	backpatch(S1.next,M2.Quad);
	backpatch(E.true = M1.Quad);
	S.next = E.false;

M□ε	M.quad = nextquad;

N□ε	N.next= makelist(nextquad),
	gen(goto __)

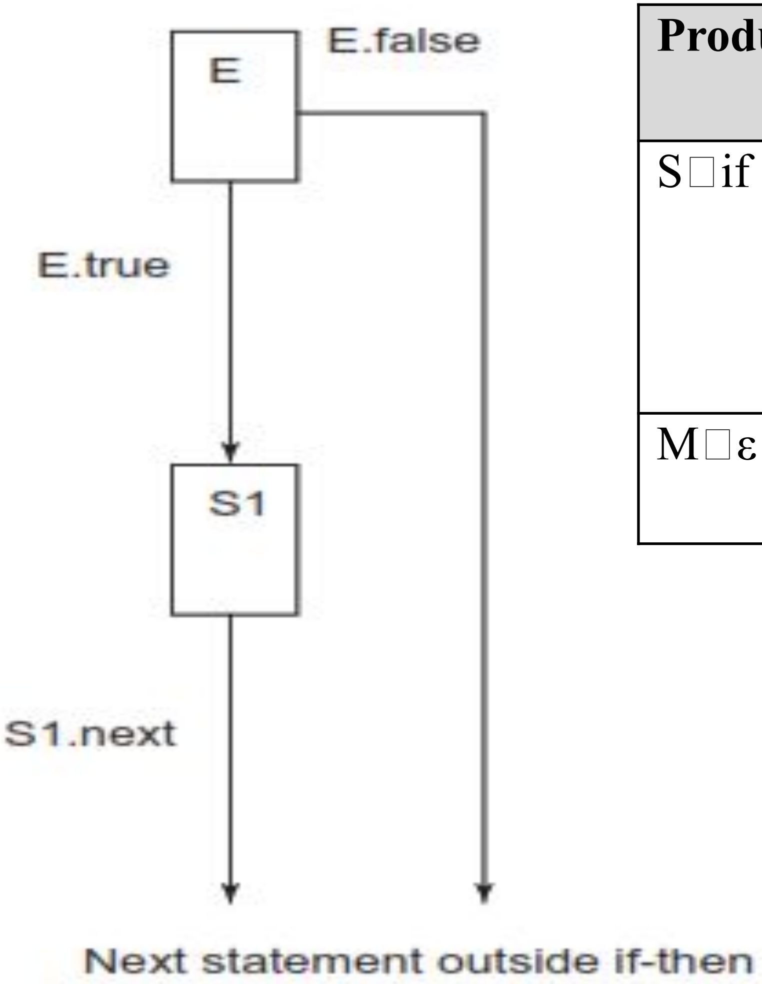
Extra **Statements**

Production	**Semantic Rule**
S□begin L end	S.next = L.next;
S□A	S.next = nil;

L□L1; M S	backpatch(L1.next , M.quad);
	L.next = S.next;

L□S	L.next = S.next;
L □e	L . next = S.next;

SDTS for If-then Construct



Grammar: if E then S1

Production	**Semantic Rule**

S□if E then M S1	backpatch(E.true,M.quad);
	S.next= merge(E.false,S1.next);

M□ε	M.quad = nextquad;

SDTS for If-then Construct Example

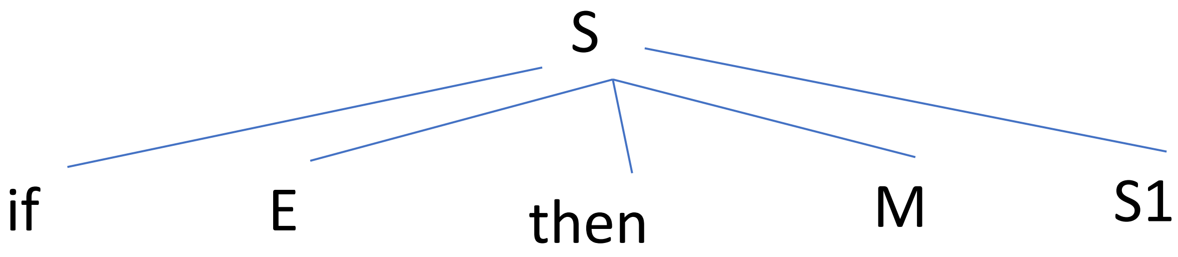
Q1. if (a>10) then p=q + r

if (a>10) then p=q+r

Production	Semantic Rule

S□if E then M S1	backpatch(E.true,M.quad);
	S.next= merge(E.false,S1.next);

M□ε	M.quad = nextquad;

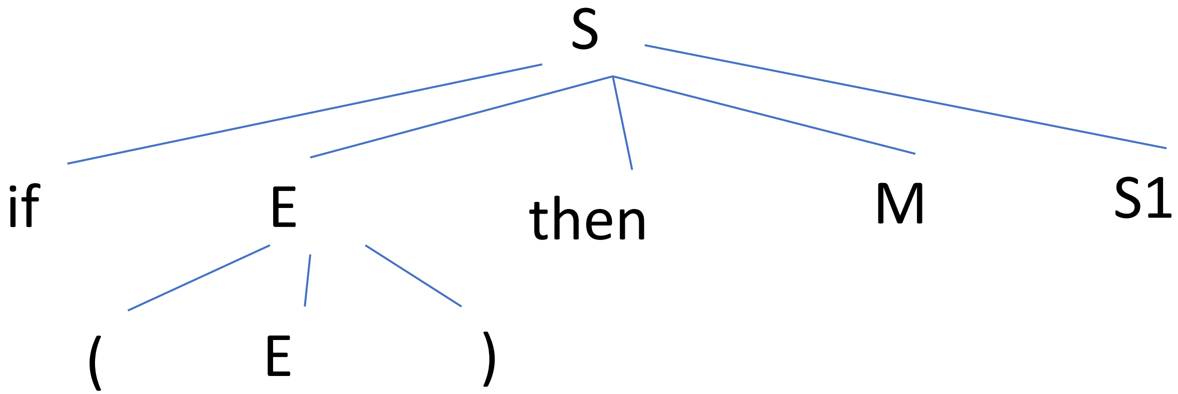


if (a>10) then p=q+r

Production	Semantic Rule

S□if E then M S1	backpatch(E.true,M.quad);
	S.next= merge(E.false,S1.next);

M□ε	M.quad = nextquad;



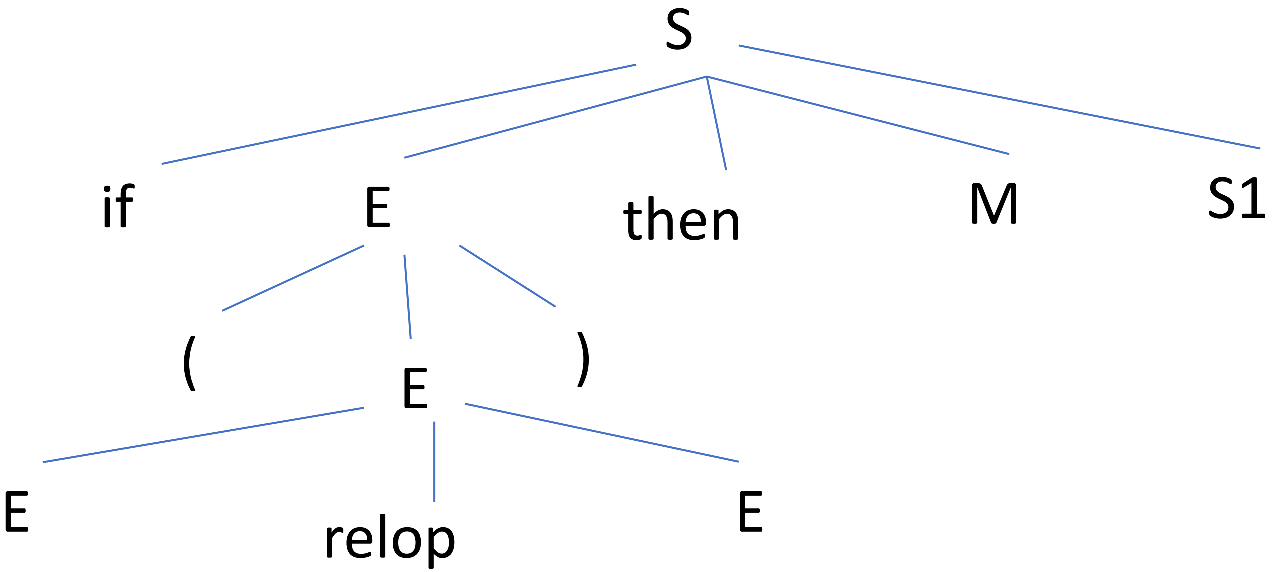
E□E1relop E2	E.true = makelist(nextquad);
	E.false= makelist(nextquad+1);
	gen (if E1.place relop.val E2.place goto_____);
	gen(goto _____);

if (a>10) then p=q+r

Production	Semantic Rule

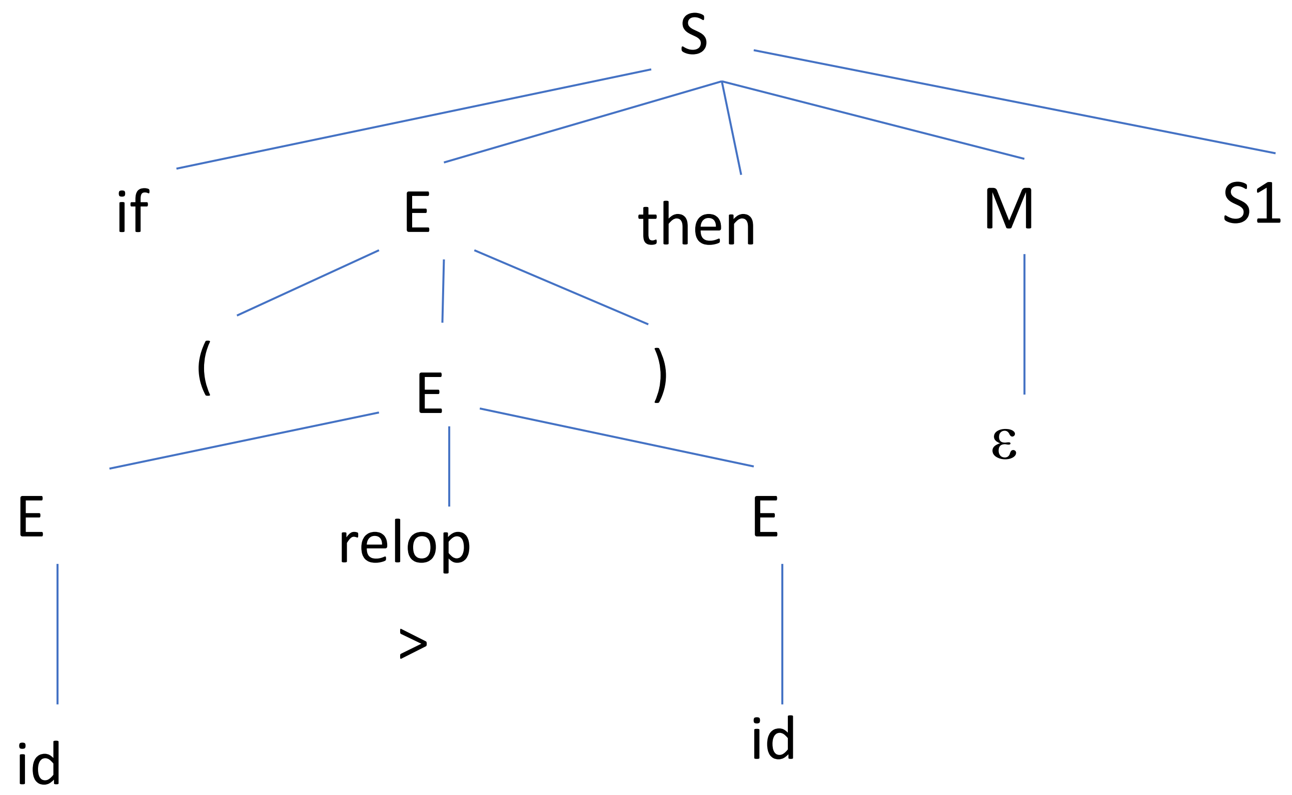
S□if E then M S1	backpatch(E.true,M.quad);
	S.next= merge(E.false,S1.next);

M□ε	M.quad = nextquad;

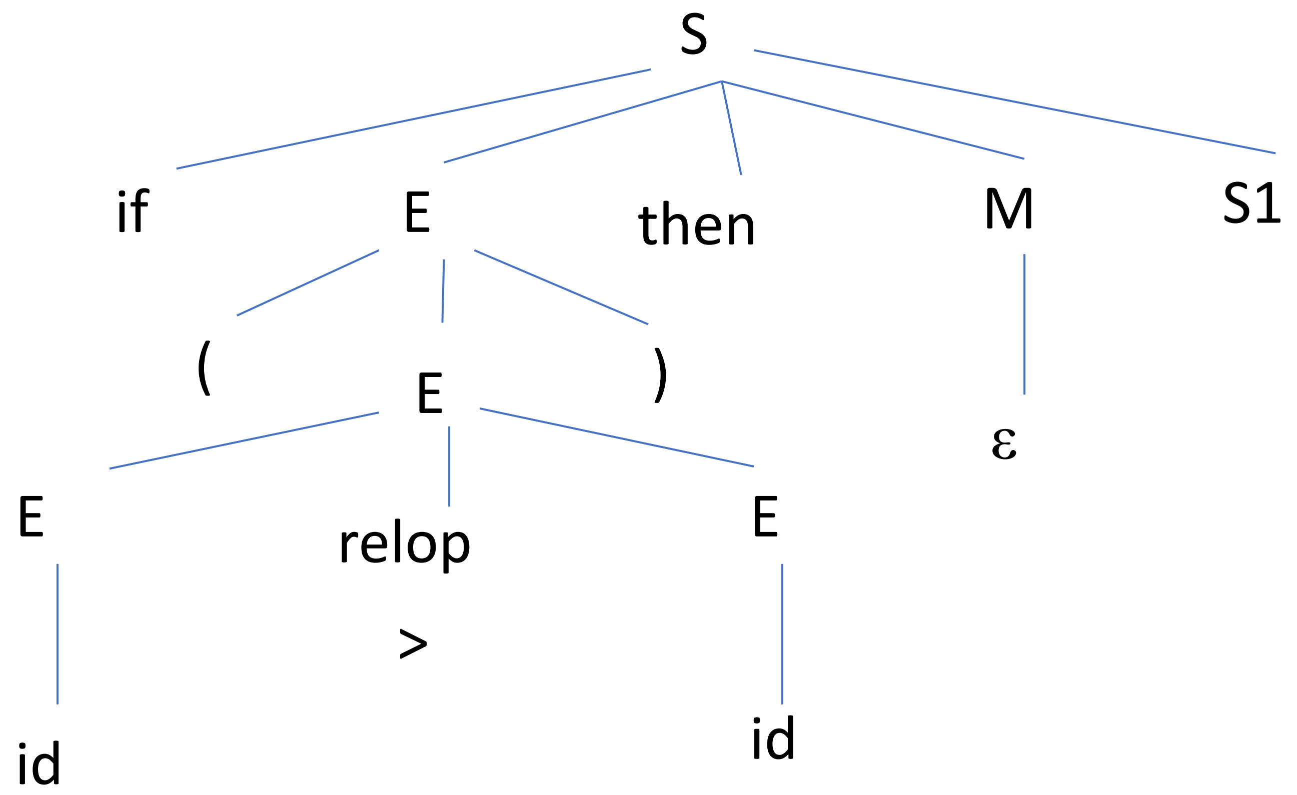


E□E1relop E2	E.true = makelist(nextquad);
	E.false= makelist(nextquad+1);
	gen (if E1.place relop.val E2.place goto_____);
	gen(goto _____);

if (a>10) then p=q+r

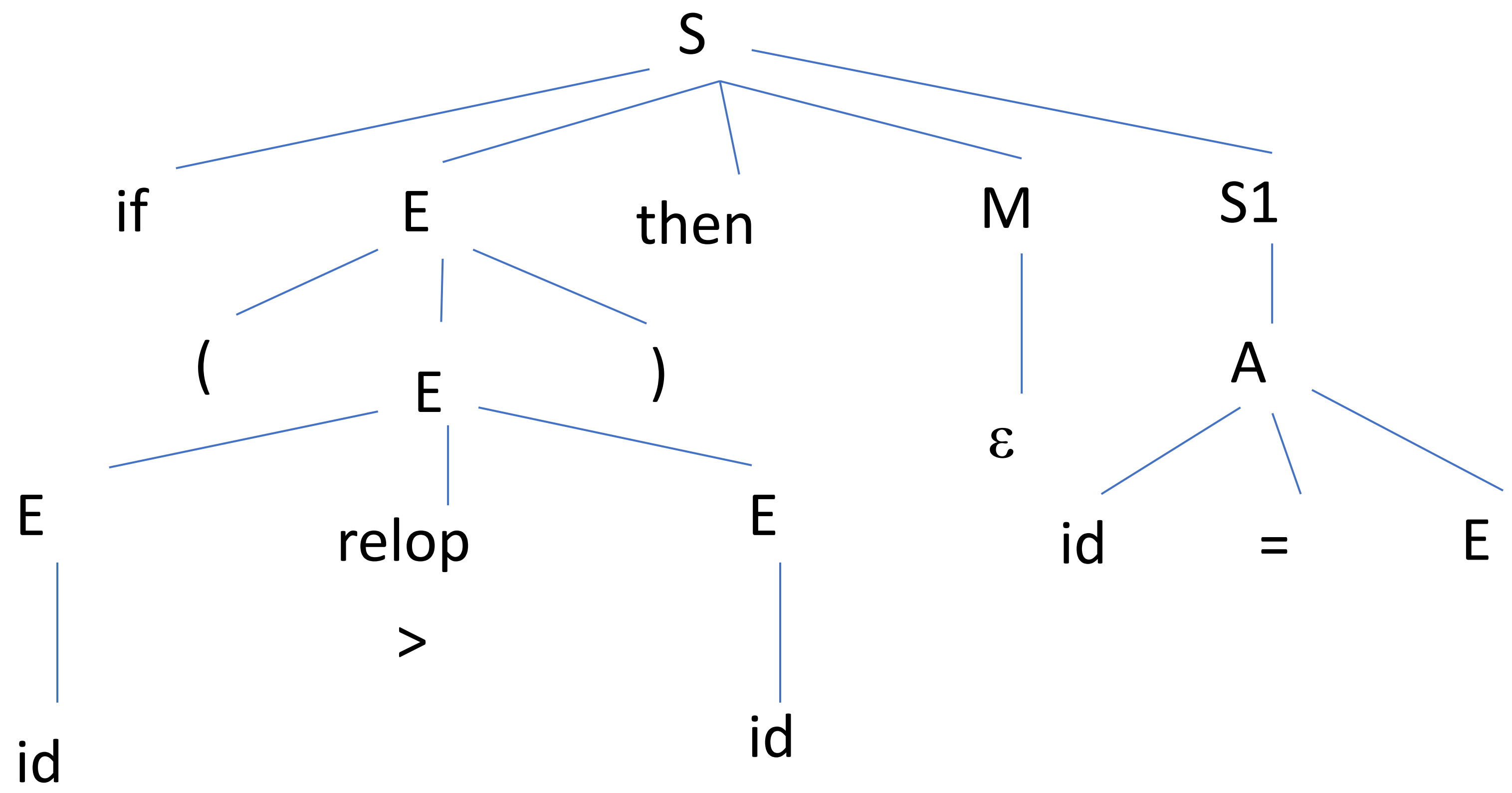


if (a>10) then p=q+r



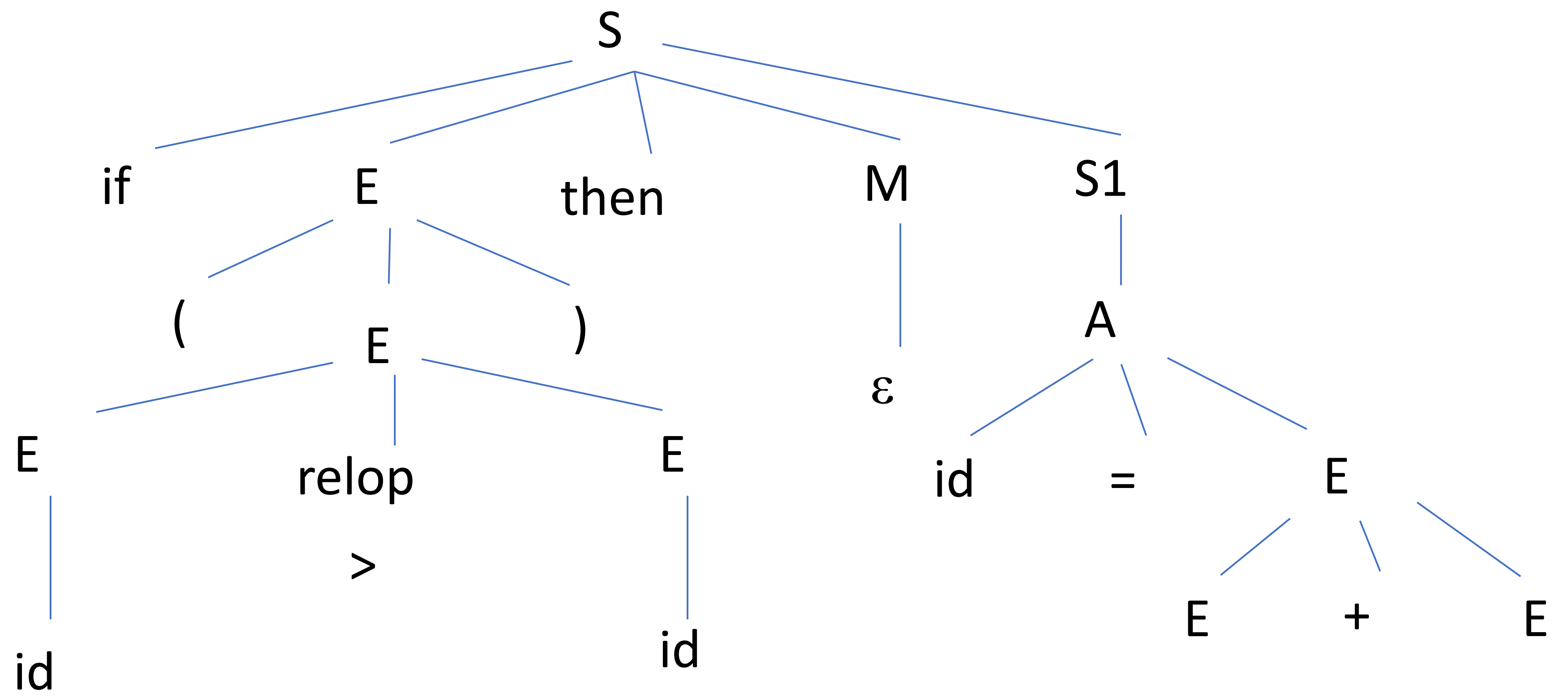
Production	Semantic Rule
A □id : = E	A .code = E. code || gen(id.place ‘ =’ E . place)
S□A	**S.next = nil;**

if (a>10) then p=q+r

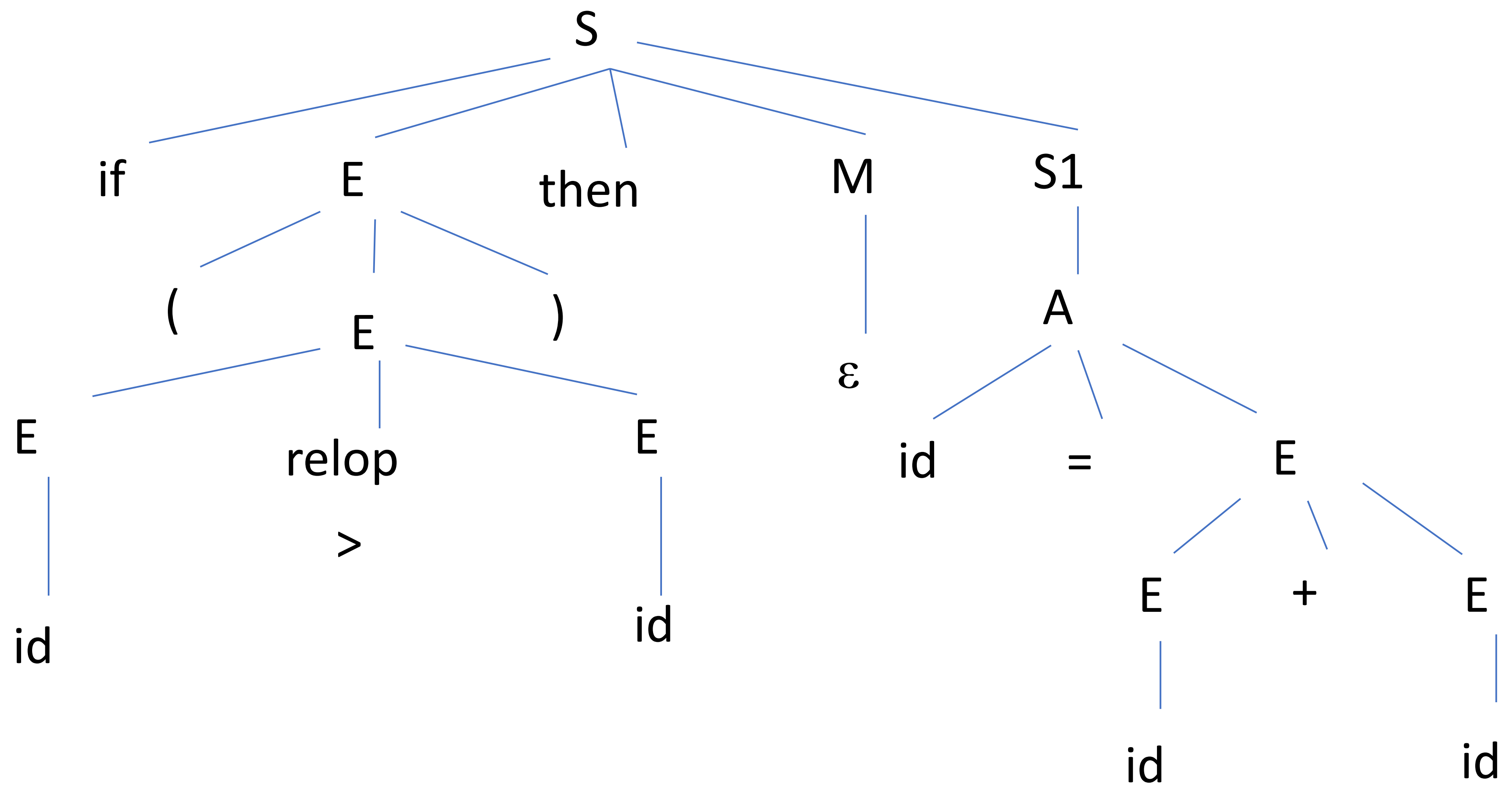


Production	Semantic Rule
A □id : = E	A .code = E. code || gen(id.place ‘ =’ E . place)
S□A	**S.next = nil;**

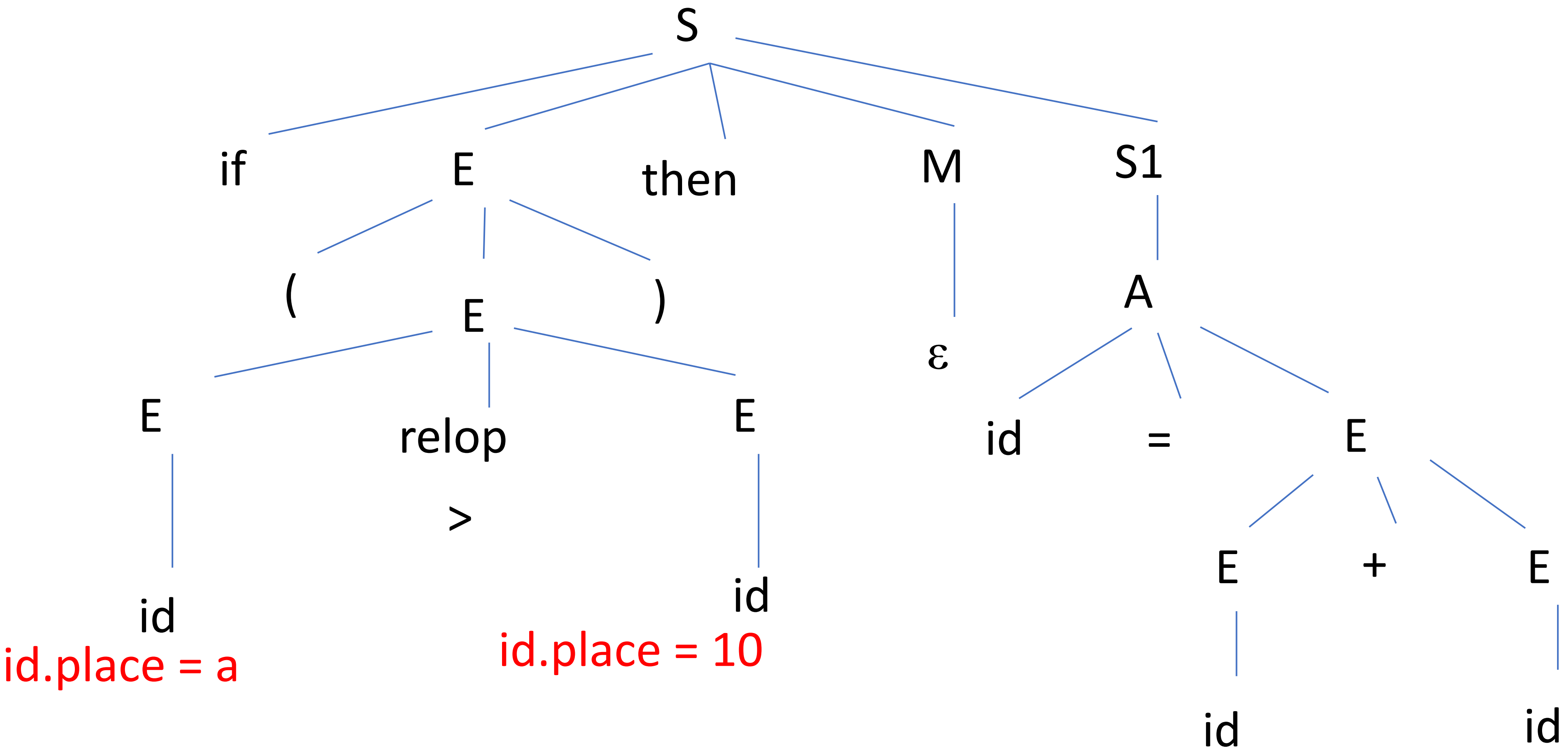
if (a>10) then p=q+r



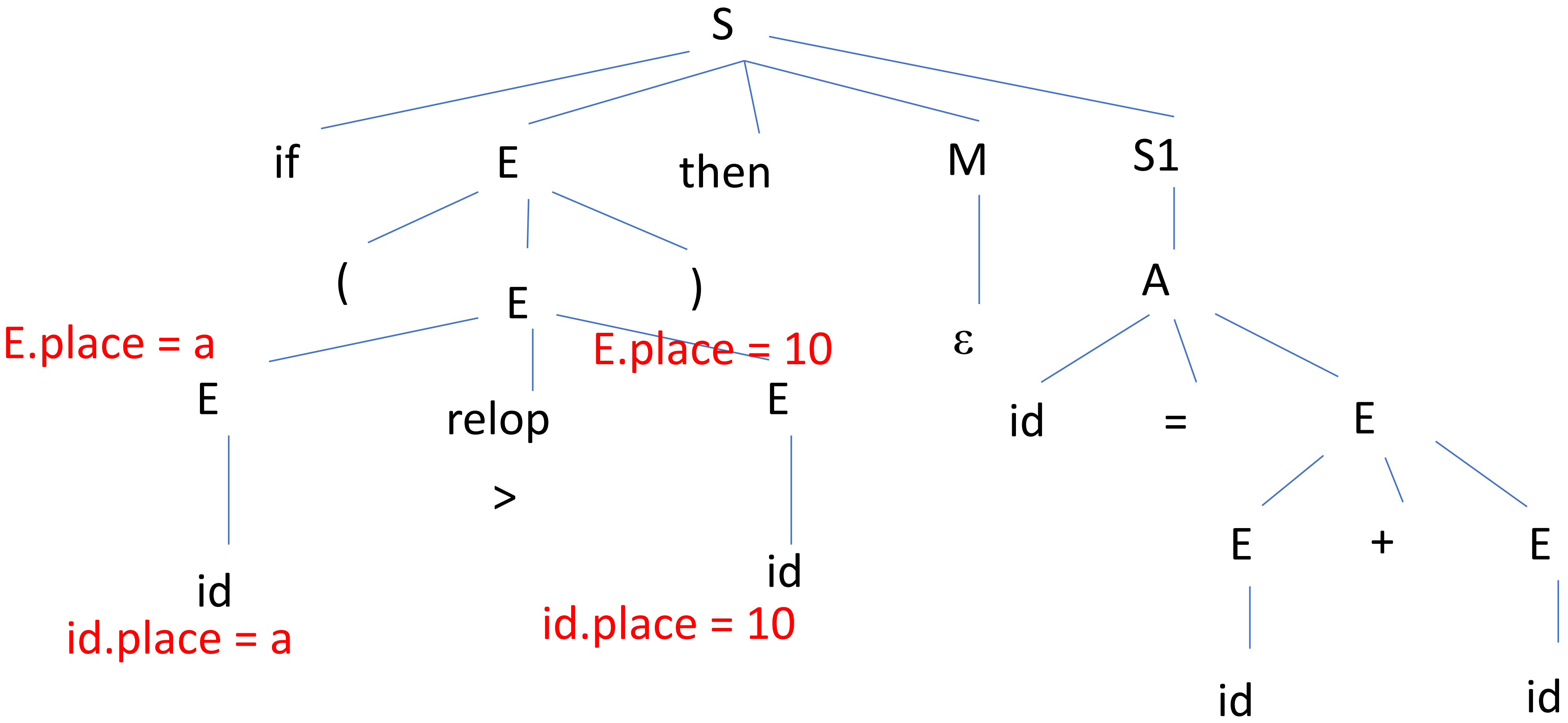
if (a>10) then p=q+r



if (a>10) then p=q+r

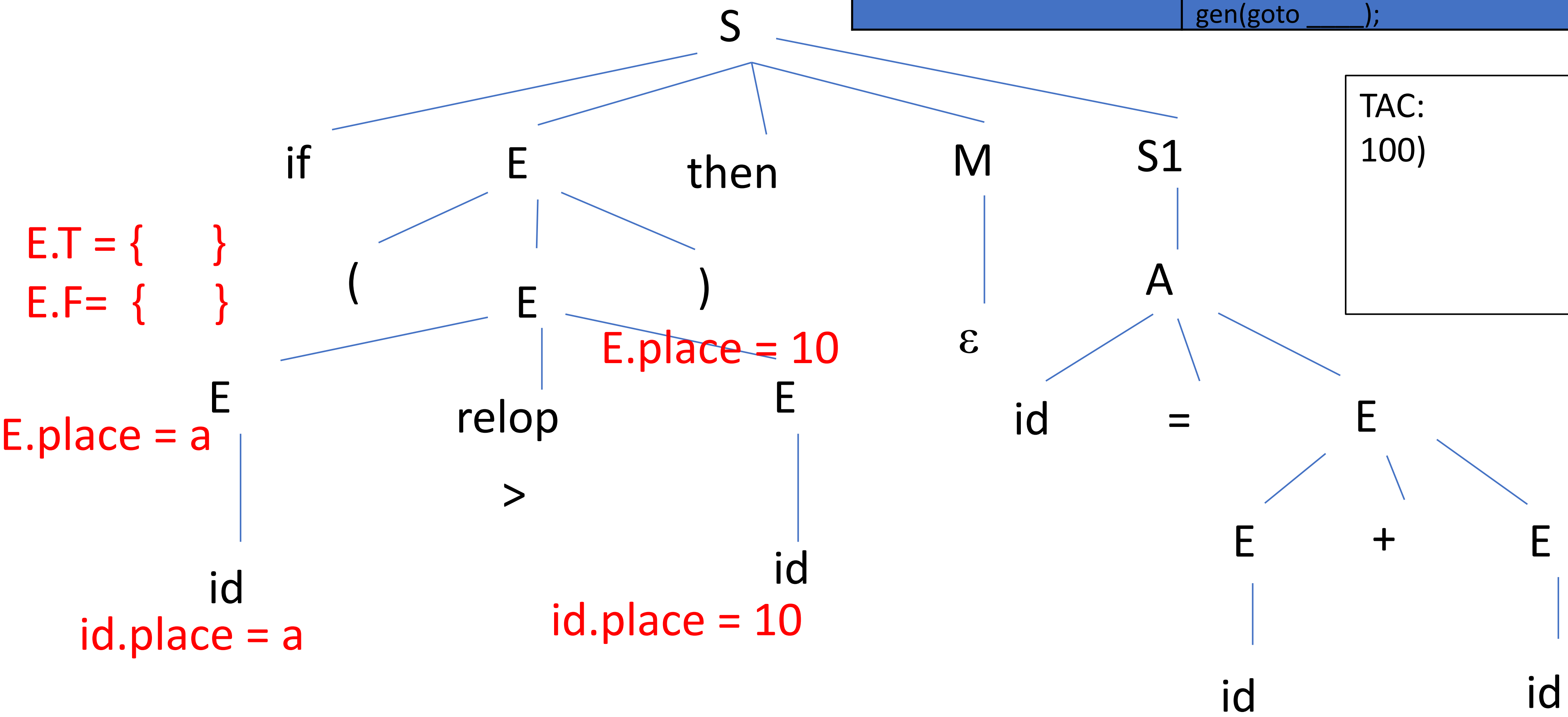


if (a>10) then p=q+r



if (a>10) then p=q+r

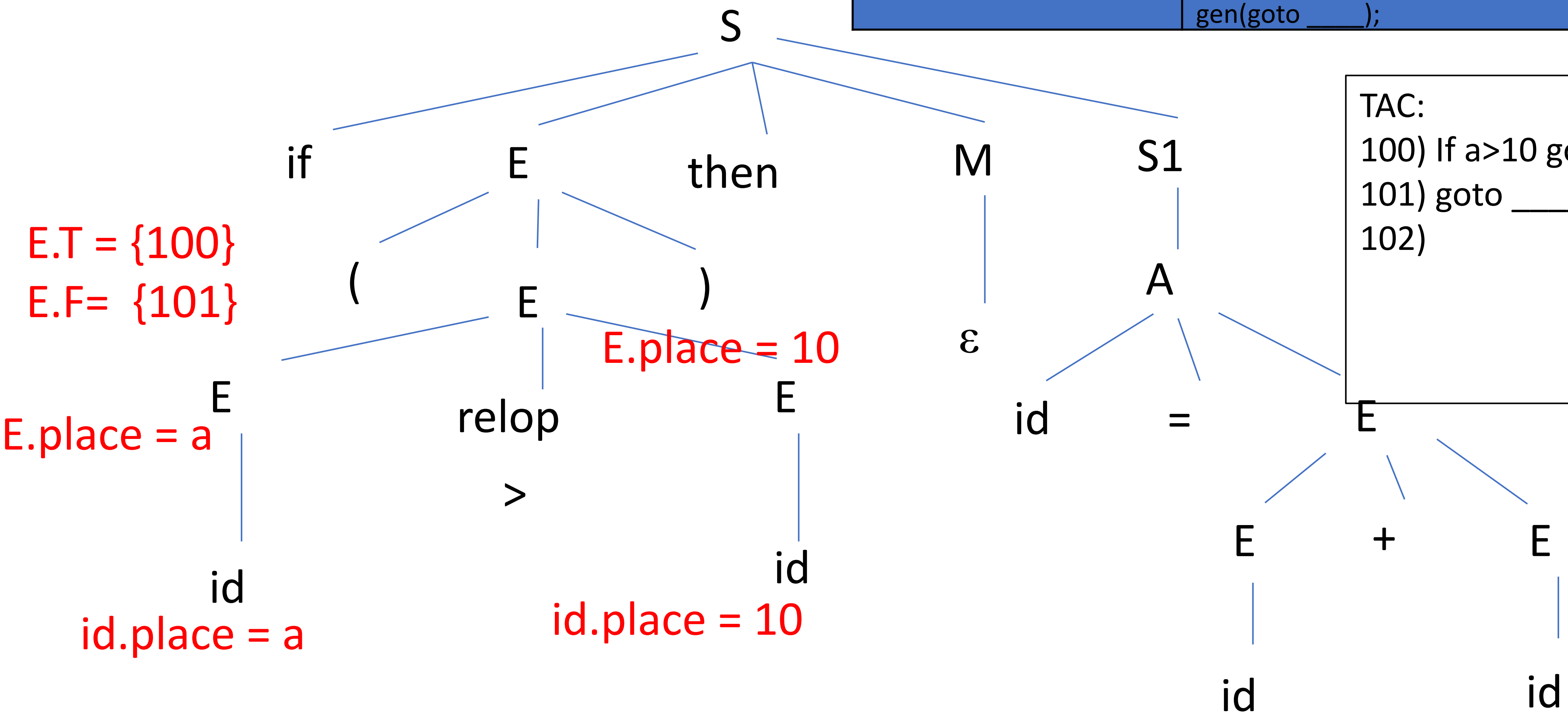
E□E1 relop E2	E.true = makelist(nextquad);
	E.false= makelist(nextquad+1);
	gen (if E1.place relop.val E2.place goto____);
	gen(goto _____);



TAC:
100)

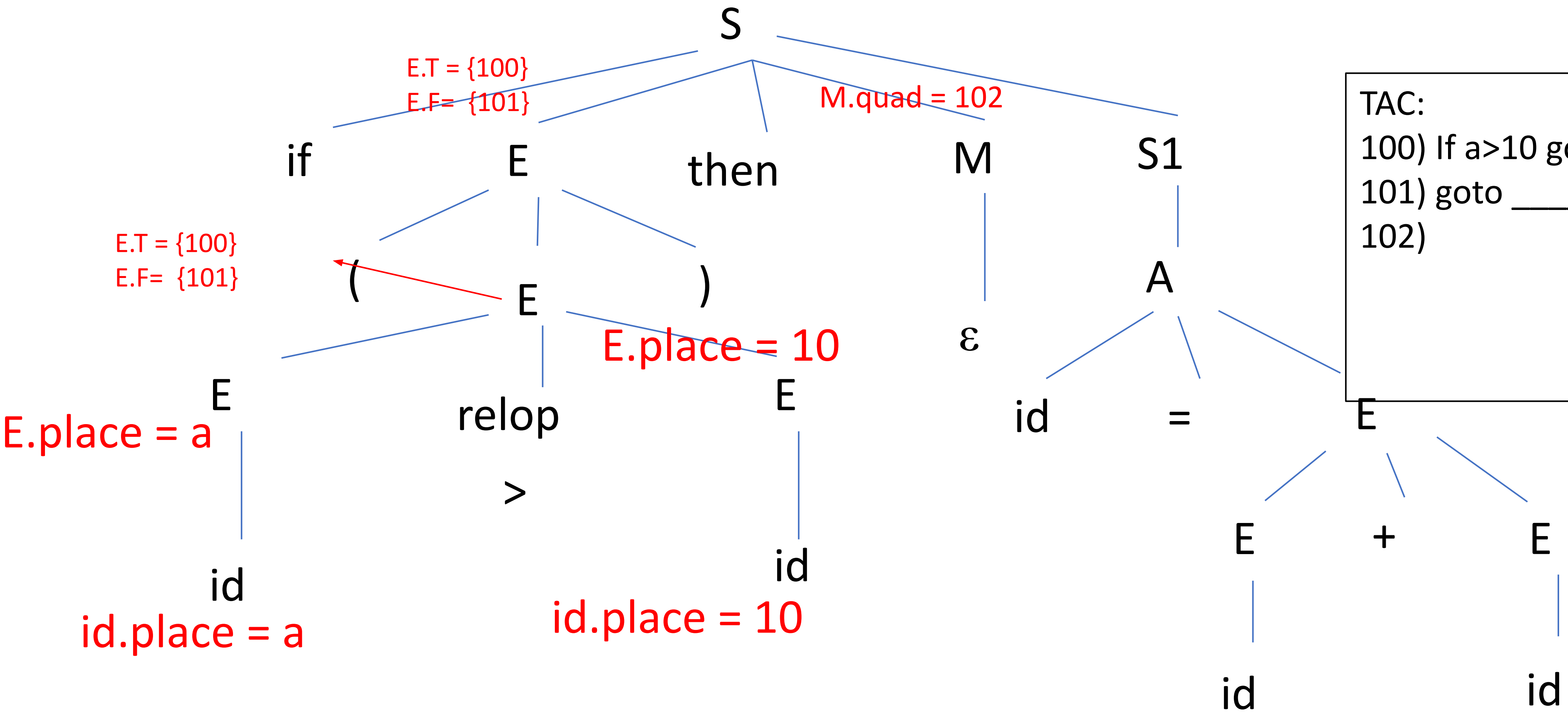
if (a>10) then p=q+r

E□E1 relop E2	E.true = makelist(nextquad);
	E.false= makelist(nextquad+1);
	gen (if E1.place relop.val E2.place goto____);
	gen(goto _____);



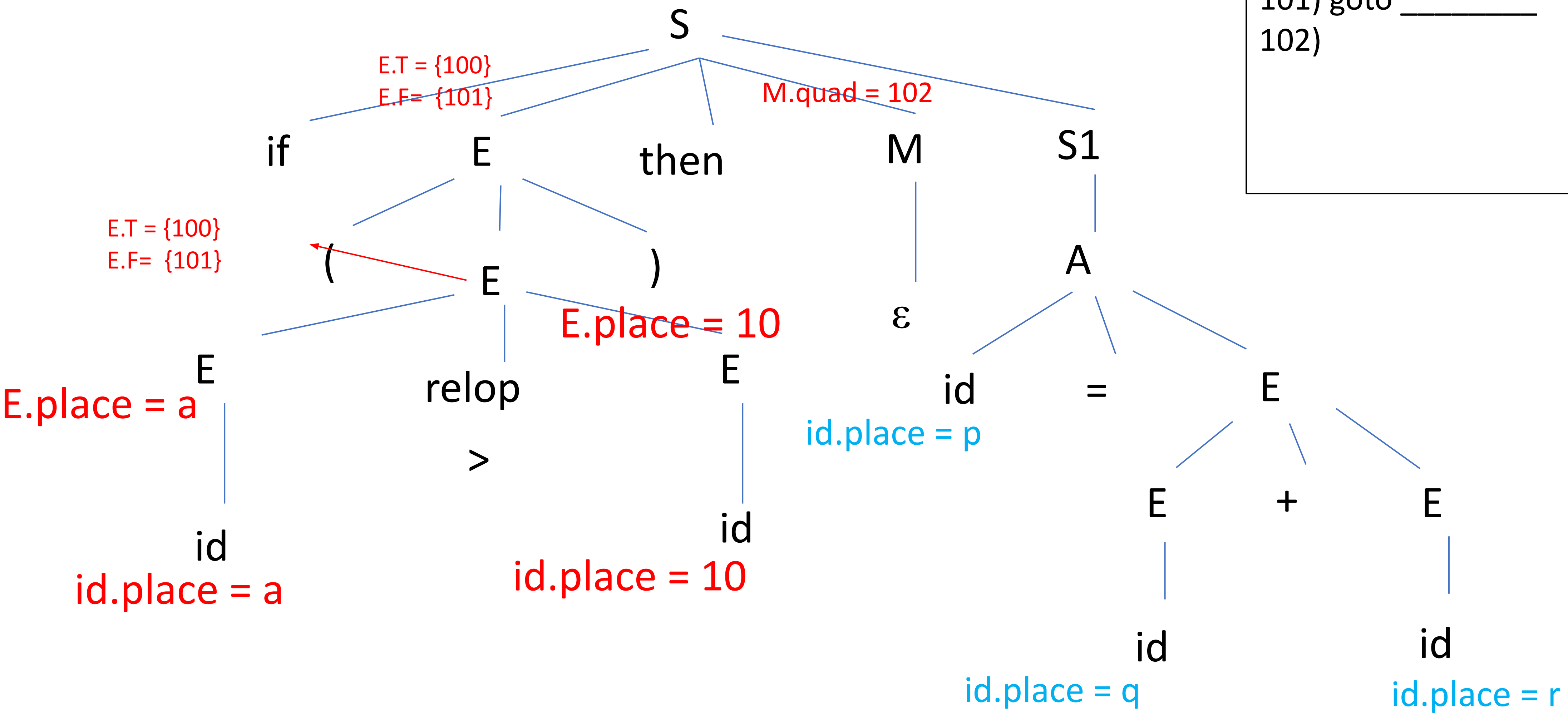
TAC:
100) If a>10 goto _______
101) goto _________
102)

if (a>10) then p=q+r



TAC:
100) If a>10 goto ______
101) goto ________
102)

if (a>10) then p=q+r



TAC:

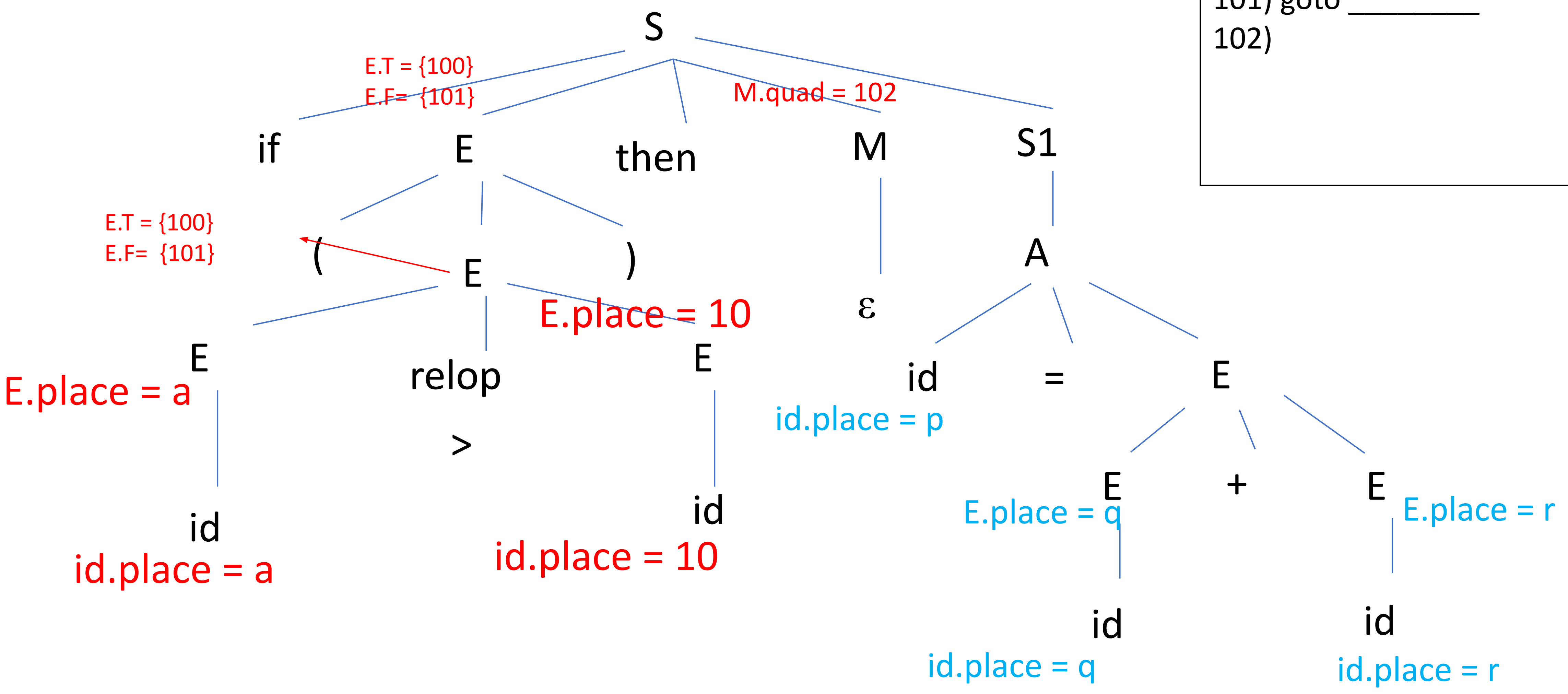
100) If a>10 goto _______

101) goto _________

102)

if (a>10) then p=q+r

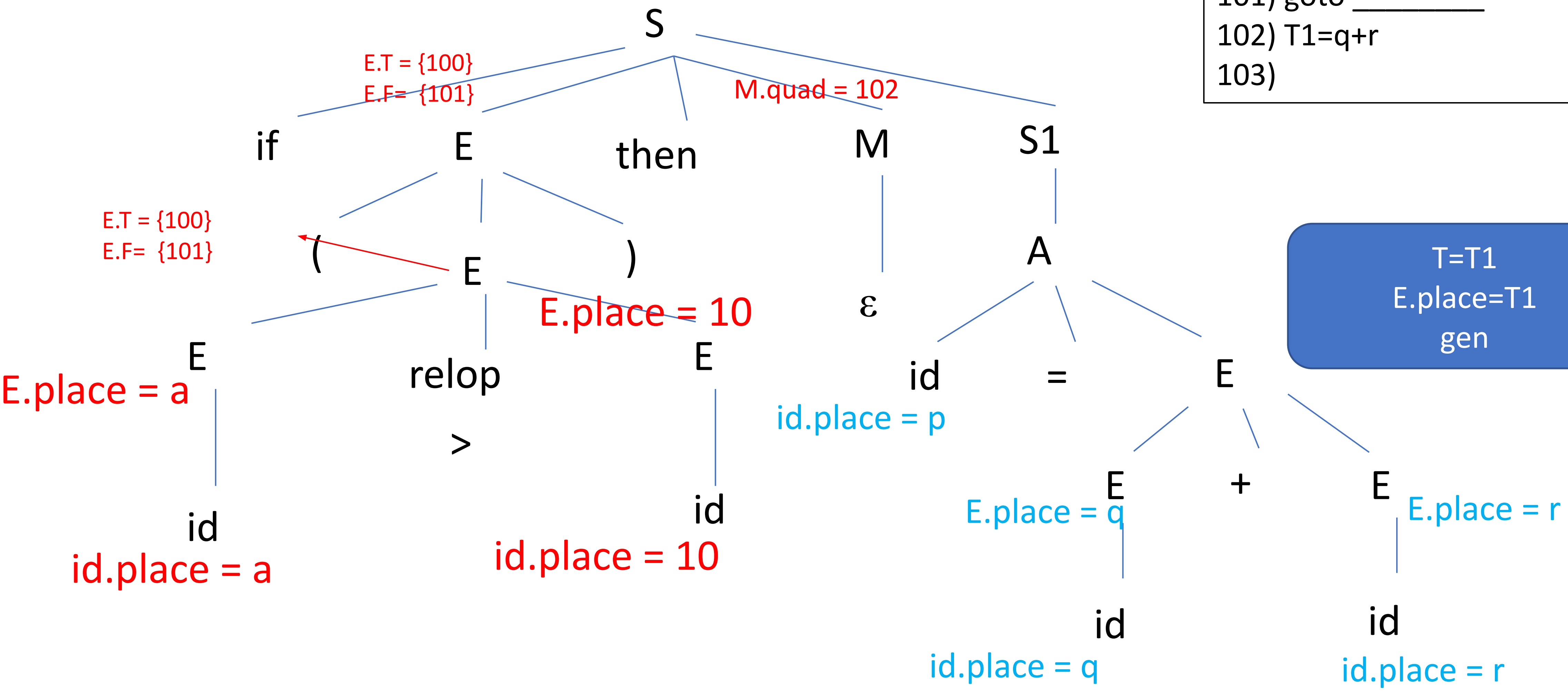
TAC:
100) If a>10 goto _______
101) goto _________
102)



E → E1+E2	T= newTemp( );
	E.place : = T;
	E.code : = E1.code || E2.code || gen(E.place ‘ =’ E1.place ‘+’ E2.place)

if (a>10) then p=q+r

TAC:
100) If a>10 goto _______
101) goto _________
102) T1=q+r
103)

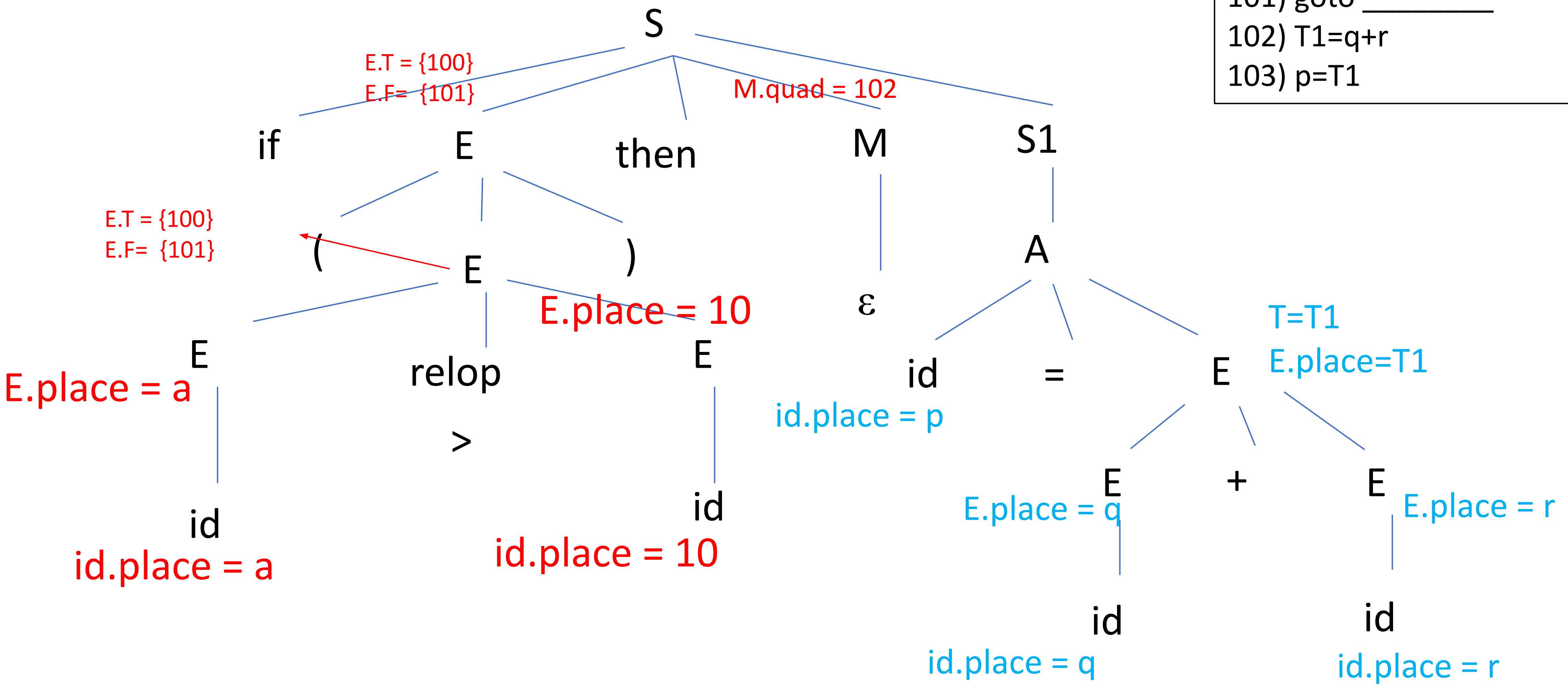


T=T1
E.place=T1
gen

E□ E1+E2	T= newTemp( );
	E.place : = T;
	E.code : = E1.code || E2.code || gen(E.place ‘ =’ E1.place ‘+’ E2.place)

if (a>10) then p=q+r

TAC:
100) If a>10 goto _______
101) goto _________
102) T1=q+r
103) p=T1



Production	Semantic Rule
A id : = E	A .code = E. code || gen(id.place ‘ =’ E . place) (THIS IS GENERATE)

if (a>10) then p=q+r

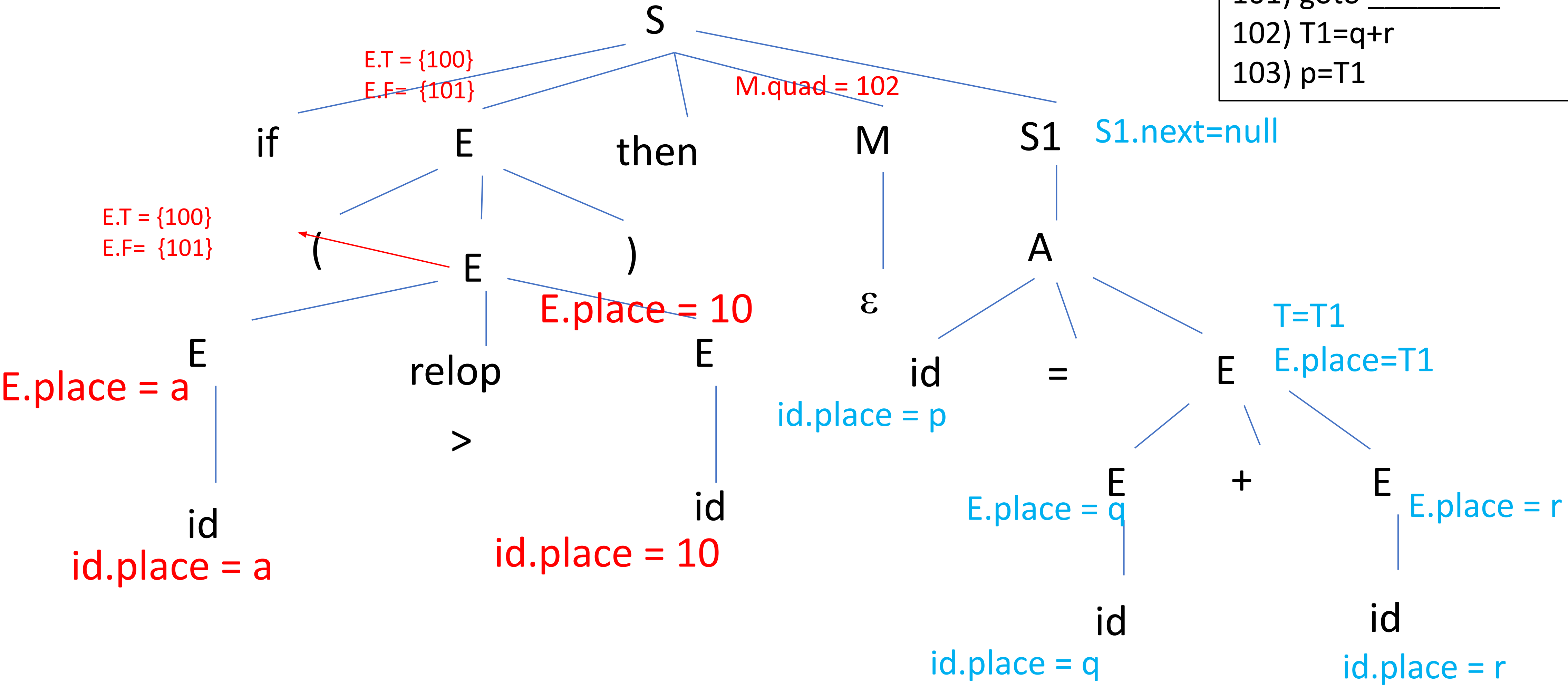
TAC:

100) If a>10 goto ______

101) goto ________

102) T1=q+r

103) p=T1



S□A	**S.next = nil;**

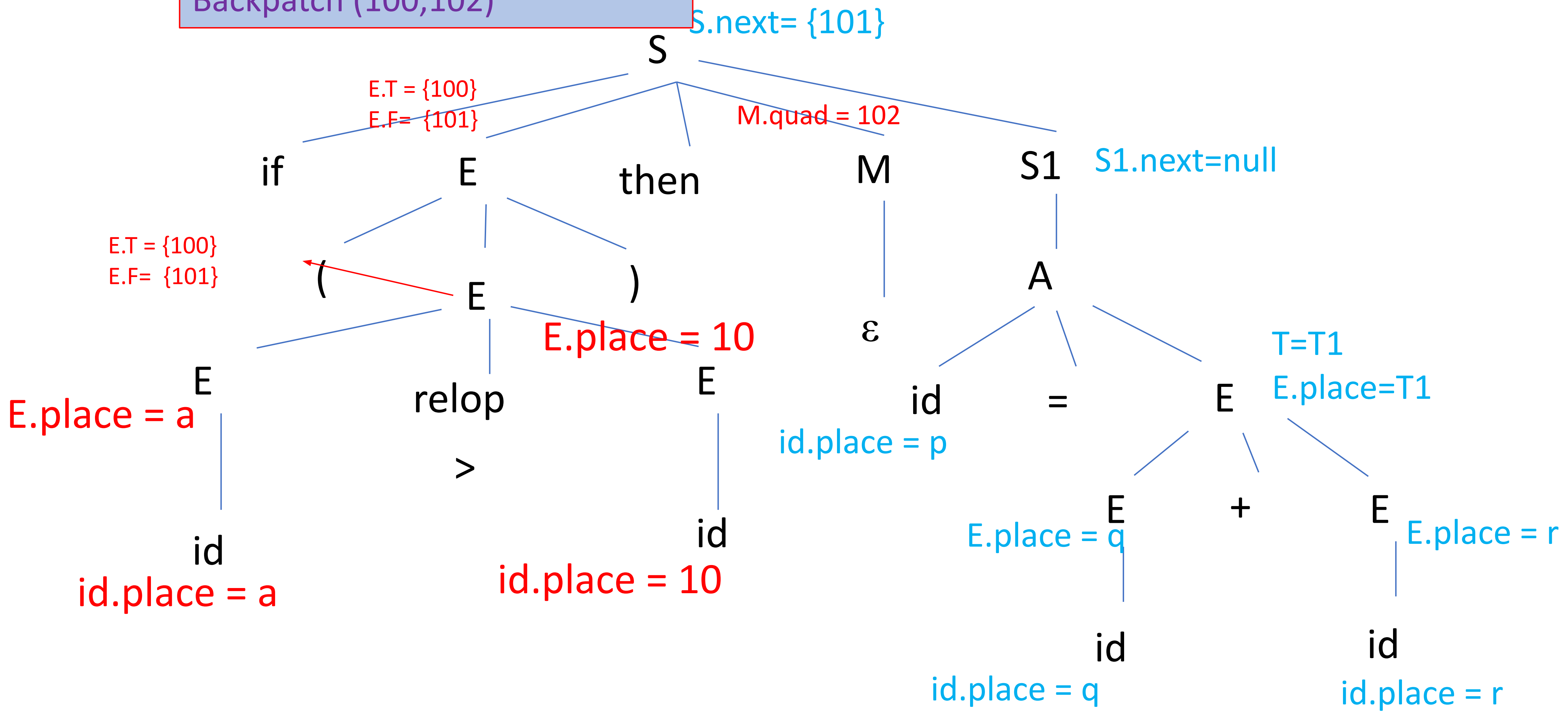
if (a>10) then p=q+r

Production	Semantic Rule

S□if E then M S1	backpatch(E.true,M.quad);
	S.next= merge(E.false,S1.next);

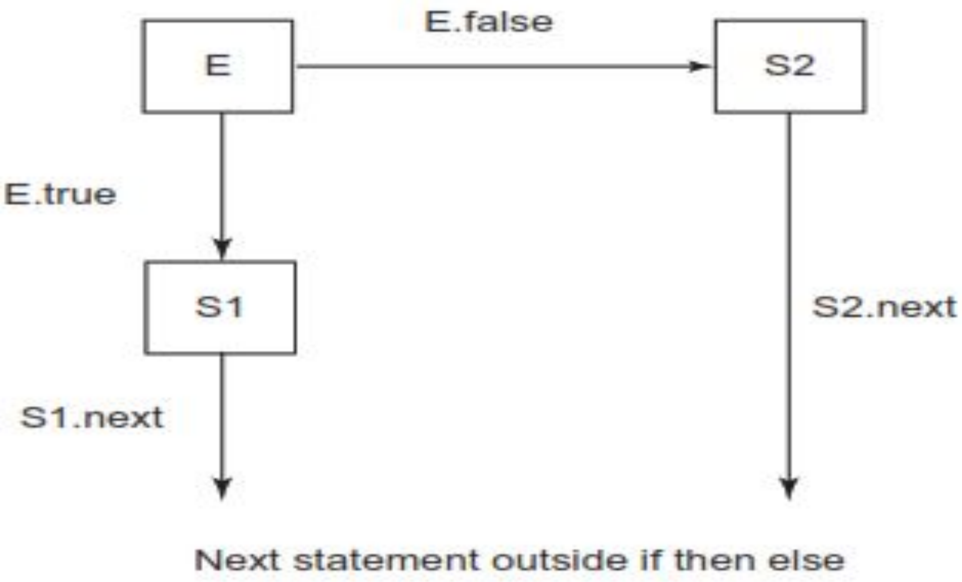
TAC:
100) If a>10 goto _102_
101) goto _________
102) T1=q+r
103) p=T1

Backpatch (100,102)



SDTS for IF-THEN-ELSE

Grammar: S□ if E then M1 S1 N else M2 S2



Production	**Semantic Rule**

S□if E then M1 S1 N else M2 S2	backpatch(E.true,M1.quad);
	backpatch(E.false,M2.quad);
	S.next=merge(S1.next, merge(N.next, S2.next);

M□ε	M.quad = nextquad;

N□ε	N.next= makelist(nextquad),
	gen(goto __)

Exercise 1

```
if(a<b or c>d) then
    x= y +z
else
    x=y -z
```

Production	Semantic Rule
S□begin L end	S.next = L.next;
S□A	S.next = nil;

L□L1; M S	backpatch(L1.next , M.quad);
	L.next = S.next;

L□S	L.next = S.next;
L □€	L . next = S.next;