UNIT III-INTERMEDIATE CODE GENERATION

6. TYPE SYSTEM

- A type System is a collection of rules for assigning type expressions to the various parts of the program.
- A type checker implements a type system. It is specified in a syntax-directed manner.

6.1 Type checking:

A compiler must check that the source program should follow the syntactic and semantic conventions of the source language and it should also check the type rules of the language. The type-checker determines whether these values are used appropriately or not.

Types of Type Checking:

There are two kinds of type checking:

1. Static Type Checking.
2. Dynamic Type Checking.

6.1.1 Static Type Checking:

Static type checking is defined as type checking performed at compile time. It checks the type variables at compile-time, which means the type of the variable is known at the compile time.

Examples of Static checks include:

Type-checks: A compiler should report an error if an operator is applied to an incompatible operand. For example, if an array variable and function variable are added together.

The flow of control checks: Checking the control flow in our looping and branching statements. For example, a break statement in C causes control to leave the smallest enclosing while, for, or switch statement, an error occurs if such an enclosing statement does not exist.

Uniqueness checks: There are situations in which an object must be defined only once. For example, labels in a case statement must be distinct

Name-related checks: Sometimes the same name may appear two or more times. For example in Ada, a loop may have a name that appears at the beginning and end of the construct. The compiler must check that the same name is used at both places.

The Benefits of Static Type Checking:

1. Runtime Error Protection.
2. It catches syntactic errors and wrong names.
3. Detects incorrect argument types.
4. It catches the wrong number of arguments.
5. It catches wrong return types.

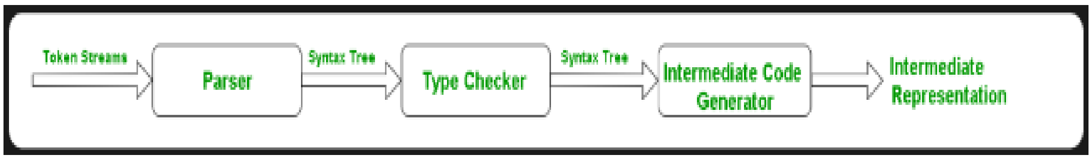
6.1.2 Dynamic Type Checking:

Dynamic Type Checking is defined as the type checking being done at run time. In Dynamic Type Checking, types are associated with values, not variables.

The design of the type-checker depends on

1. Syntactic Structure of language constructs.
2. The Expressions of languages.
3. The rules for assigning types to construct (semantic rules).

The Position of the Type checker in the Compiler:



Type checking in Compiler

6.2 Overloading:

An Overloading symbol is one that has different operations depending on its context.

Overloading is of two types

1. Operator Overloading
2. Function Overloading

6.2.1 Operator Overloading: In Mathematics, the arithmetic expression "x+y" has the addition operator '+' is overloaded because '+' in "x+y" have different operators when 'x' and 'y' are integers, complex numbers, reals, and Matrices.

6.2.2 Function Overloading: The Type Checker resolves the Function Overloading based on types of arguments and Numbers.

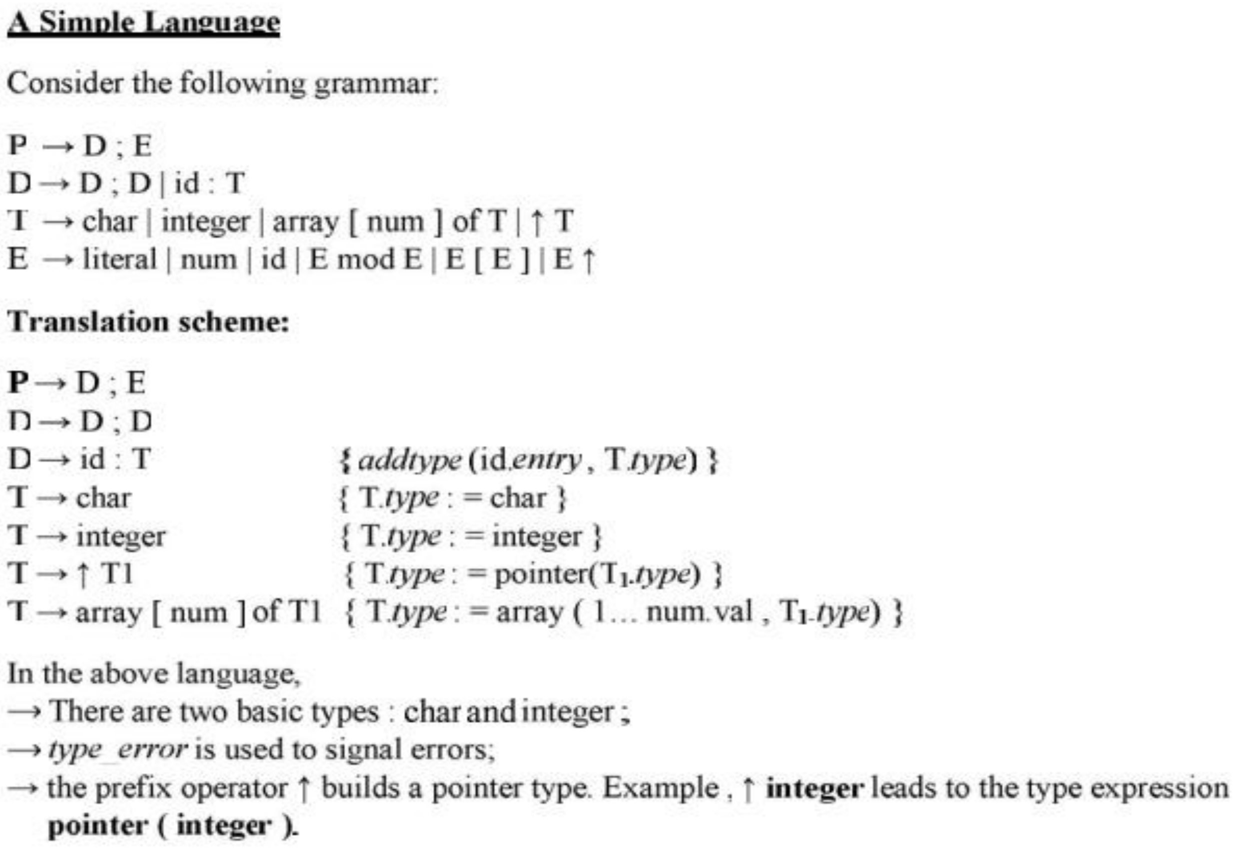
Example:

```
E-->E1(E2)
          {
            E.type:= if E2.type = s
            E1.type = s -->t then t
                      else type_error
          }
```

6.3 SPECIFICATION OF A SIMPLE TYPE CHECKER

- Specification of a simple type checker for a simple language in which the type of each identifier must be declared before the identifier is used. The type checker is a translation scheme that synthesizes the type of each expression from the types of its sub expressions. The type checker can handle arrays, pointers, statements, and functions.
- Specification of a simple type checker includes the following:

- **A Simple Language**
- **Type Checking of Expressions**
- **Type Checking of Statements**
- **Type Checking of Functions**



A Simple Language

Consider the following grammar:

P → D ; E
D → D ; D | id : T
T → char | integer | array [num] of T | ↑ T
E → literal | num | id | E mod E | E [E] | E ↑

Translation scheme:

P → D ; E	
D → D ; D	
D → id : T	{ *addtype* (id.*entry* , T.*type*) }
T → char	{ T.*type* : = char }
T → integer	{ T.*type* : = integer }
T → ↑ T1	{ T.*type* : = pointer(T_1.*type*) }
T → array [num] of T1	{ T.*type* : = array (1… num.val , T_1.*type*) }

In the above language,
→ There are two basic types : char and integer ;
→ *type_error* is used to signal errors;
→ the prefix operator ↑ builds a pointer type. Example , ↑ **integer** leads to the type expression **pointer (integer).**

Type checking of expressions

In the following rules, the attribute *type* for E gives the type expression assigned to the expression generated by E.

1. $E \rightarrow$ **literal** { E.*type* : = *char* }
   $E \rightarrow$ **num** { E.*type* : = *integer* }
   Here, constants represented by the tokens **literal** and **num** have type *char* and *integer*.

2. $E \rightarrow$ **id** { E.*type* : = *lookup* (**id**.*entry*) }
   lookup (e) is used to fetch the type saved in the symbol table entry pointed to by e.

3. $E \rightarrow E_1$ **mod** E_2 { *E.type* : = **if** E_1. *type* = *integer* **and**
   E_2. *type* = *integer* **then** *integer*
   else *type_error* }
   The expression formed by applying the mod operator to two subexpressions of type integer has type integer; otherwise, its type is *type_error*.

4. $E \rightarrow E_1 [E_2]$ { *E.type* : = if E_2.*type* = *integer* **and**
   E_1.*type* = *array(s,t)* **then** *t*
   else *type_error* }
   In an array reference $E_1 [E_2]$, the index expression E_2 must have type integer. The result is the element type *t* obtained from the type *array(s,t)* of E_1.

5. $E \rightarrow E_1 \uparrow$ { *E.type* : = **if** E_1.*type* = *pointer (t)* **then** *t*
   else *type_error* }

   The postfix operator ↑ yields the object pointed to by its operand. The type of E ↑ is the type *t* of the object pointed to by the pointer E.

Type checking of statements

Statements do not have values; hence the basic type *void* can be assigned to them. If an error is detected within a statement, then *type_error* is assigned.

Translation scheme for checking the type of statements:

1. Assignment statement:

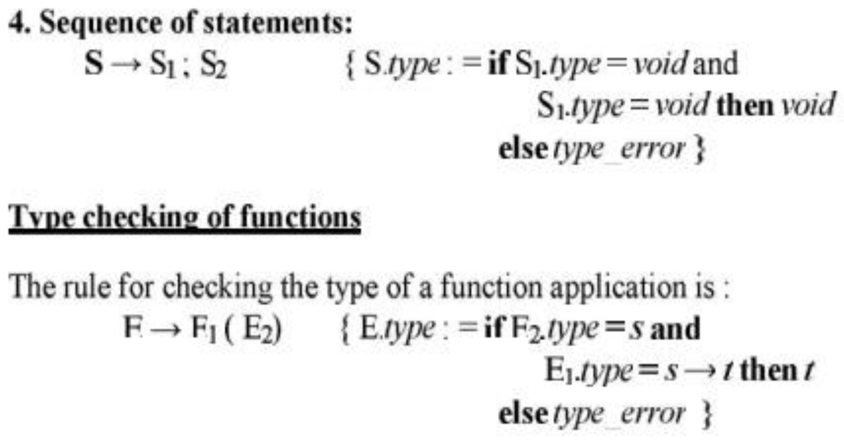
$S \rightarrow$ **id** : = E { S.*type* : = **if id**.*type* = E.*type* **then** *void*
else *type_error* }

2. Conditional statement:

$S \rightarrow$ **if** E **then** S_1 { S.*type* : = **if** E.*type* = *boolean* **then** S_1.*type*
else *type_error* }

3. While statement:

$S \rightarrow$ while E do S_1 { S.*type* : = **if** E.*type* = *boolean* **then** S_1.*type*
else *type_error* }



4. Sequence of statements:

$S \rightarrow S_1 \; ; \; S_2$ { $S.type$:= **if** $S_1.type = void$ and $S_1.type = void$ **then** $void$ **else** $type\_error$ }

Type checking of functions

The rule for checking the type of a function application is :

$F \rightarrow F_1 (E_2)$ { $E.type$:= **if** $F_2.type = s$ **and** $E_1.type = s \rightarrow t$ **then** t **else** $type\_error$ }

Equivalence of Type Expressions

If two type expressions are equal then return a certain type else return type_error.

When checking equivalence of named types, we have two possibilities.

- Structural Equivalence
- Names Equivalence

Structural Equivalence of Type Expressions

- Type expressions are built from basic types and constructors, a natural concept of equivalence between two type expressions is structural equivalence.
- Example: The constructed type array(n1 , t1) and array(n2 , t2) are equivalent if n1 = n2 and t1 =t2

Names for Type Expressions

- In some languages, types can be given names (Data type name). For example, in the Pascal program fragment.

Type Conversions

- Conversion of one data type to another automatically by the compiler is called "Type Conversion".
- For Example, Consider expressions like x + i, where x is of type float and i is of type integer.

the compiler may need to convert one of the operands of + to ensure that both operands are of the same type when the addition occurs.

Implicit Conversion:

- Implicit type conversion is done automatically by the compiler, without being specified by the user.
- For example, the integer 2 is converted to a float in the code for the expression 2 * 3 .14:

Explicit Conversion:

- Explicit type casting has to be explicitly defined by the user to temporarily change the datatype of a variable to some other datatype.

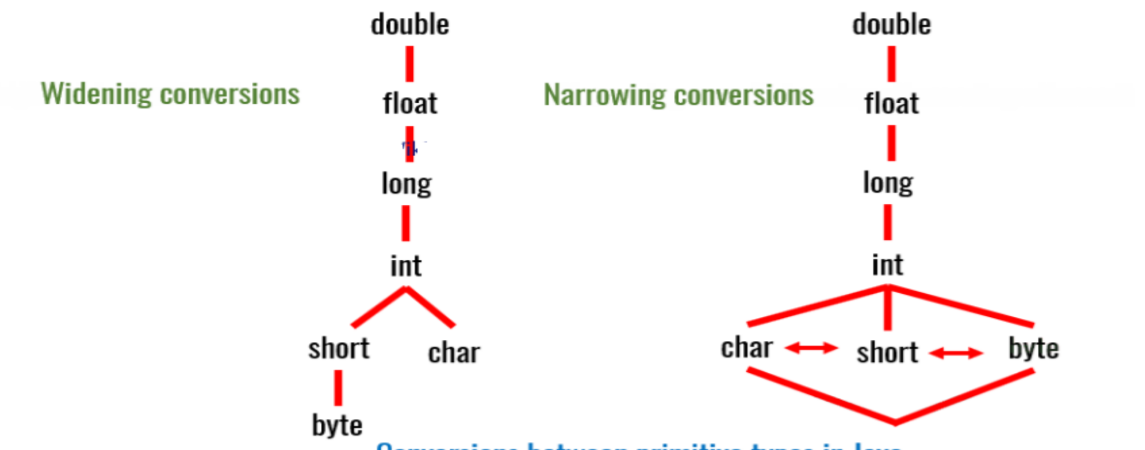
Example:

t1 = (float) 2

t2 = t1 * 3.14

The rule associated with E → >E1 + E2 builds on the pseudocode

```
if (E1.type = integer and E2.type = integer)          E.type = integer;
else if (E1.type = float and E2. type = integer)      E.type = float;
else if (E1.type = integer and E2. type = float)      E.type = float;
else if (E1.type = float and E2. type = float)        E.type = float;
```



Conversions between primitive types in Java

1. SYNTAX DIRECTED DEFINITIONS

Syntax-Directed Translations

- Translation of languages guided by CFGs
- Information associated with programming language constructs
  - Attributes attached to grammar symbols
  - Values of attributes computed by “semantic rules” associated with grammarproductions
- Two notations for associating semantic rules
  - Syntax-directed definitions
  - Translation schemes

Semantic Rules

- Semantic rules perform various activities:
  - Generation of code
  - Save information in a symbol table
  - Perform Semantic Check
  - Issue error messages
  - Other activities
- Output of semantic rules is the translation of the token stream

Syntax-Directed Translation Schemes (SDT)

- SDT embeds program fragments called semantic actions within production bodies. The position of semantic action in a production body determines the order in which the action is executed.

<u>SYNTAX DIRECTED DEFINITIONS</u>

- Syntax Directed Definitions are a generalization of context-free grammars in which:

  1. Grammar symbols have an associated set of Attributes;

  2. Productions are associated with Semantic Rules for computing the values of attributes.

  - **Annotated Parse-Trees** where each node of the tree is a record with a field for each attribute (e.g., X.a indicates the attribute a of the grammar symbol X).
  - The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.

There are two kinds of attributes:

1. **Synthesized Attributes(S-Attribute)**: They are computed from the values of the attributes of the children nodes.
2. **Inherited Attributes(L-Attribute)**: They are computed from the values of the attributes of both the siblings and the parent nodes

1.Synthesized Attribute

The Syntax Directed Definition associates to each non terminal a synthesized attribute called val.

Production	**Semantic Rules**
$L \rightarrow E$ **return**	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ **digit**	F.val = **digit**.lexval

Definition: An S-Attributed Definition is a Syntax Directed Definition that uses only synthesized attributes.

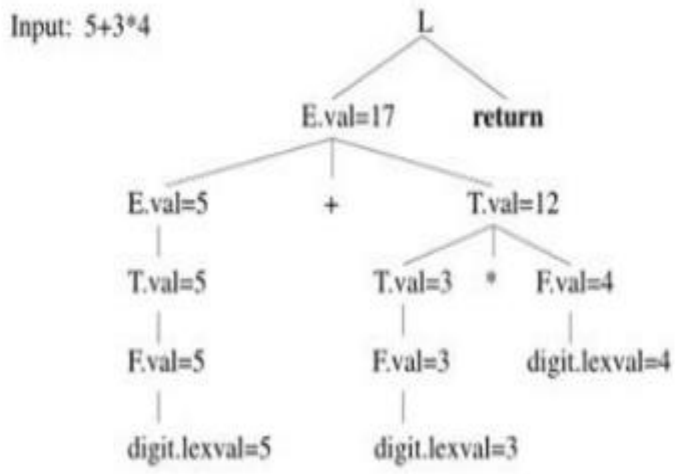
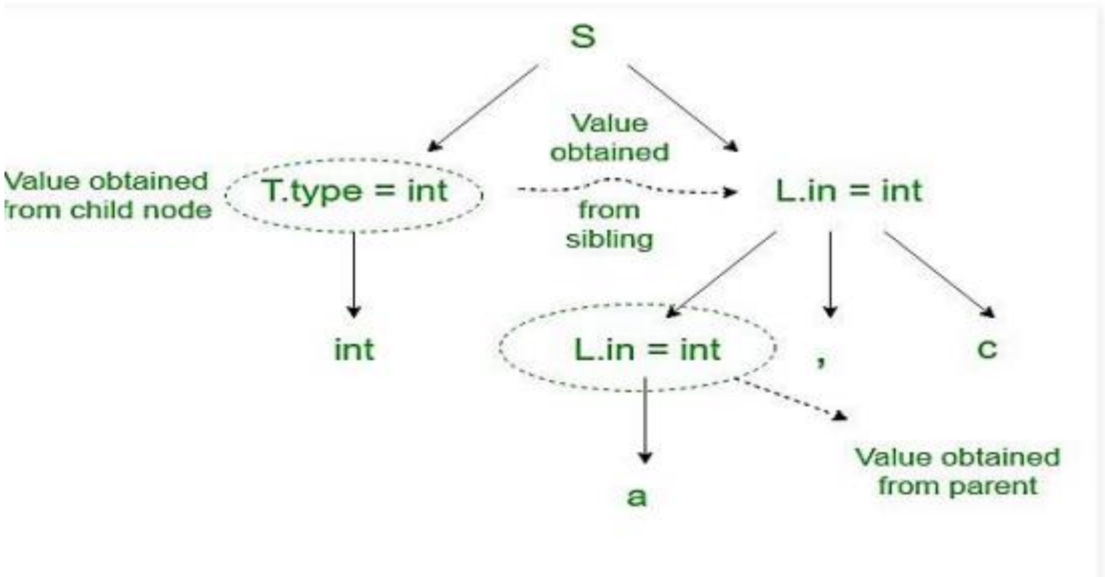


Fig 3.1 Annotated Parse Tree

2.Inherited Attribute

Production	Semantic Rules
S → T L	L.in := T.type
T → int	T.type := integer
T → real	T.type := real

L □ L_1 , id	L_1.in := L.in addtype(id.entry, L.in)
L □ id	addtype(id.entry, L.in)



Annotated Parse Tree

BOTTOM-UP EVALUATION OF S-ATTRIBUTED DEFINITIONS

- We put the values of the synthesized attributes of the grammar symbols into a parallelstack.
  - When an entry of the parser stack holds a grammar symbol X (terminal or non- terminal), the corresponding entry in the parallel stack will hold the synthesized attribute(s) of the symbol X.
- We evaluate the values of the attributes during reductions.

Bottom-Up Evaluation Example

Production	Code Fragment
(1) L □ E \n	Print(val[top])
(2) E □ E_1 + t	val[ntop] := val[top-2] + val[top]
(3) E □ T	

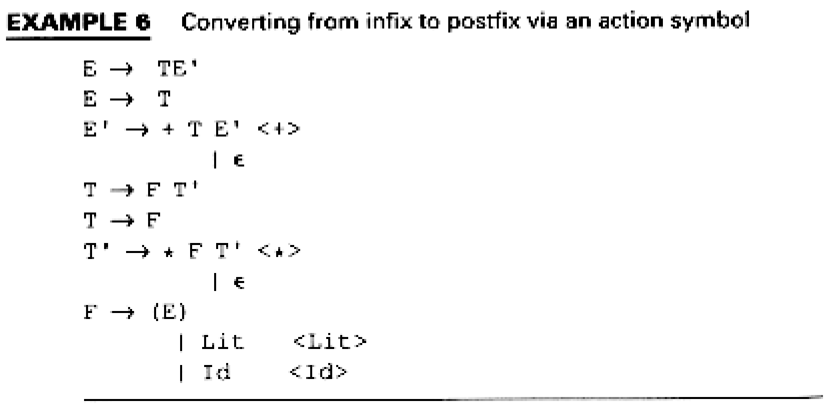
(4) T □ `T$_1$ * F	val[ntop] := val[top-2] * val[top]
(5) T □ F	
(6) F □ (E)	val[ntop] := val[top-1]
(7) F □ digit	

Evaluation

Input	State	Val	Rule
3*5+4\n	---	---	
*5+4\n	3	3	
*5+4\n	F	3	(7)
*5+4\n	T	3	(5)
5+4\n	T*	3_	
+4\n	T*5	3_5	
+4\n	T*F	3_5	(7)
+4\n	T	3_5	(4)
+4\n	E	15	(3)
4\n	E+	15_	
\n	E+4	15_4	
\n	E+F	15_4	(7)
\n	E+T	15_4	(5)
\n	E	19	(2)
	E\n	19_	
	L	19	(1)

TOP DOWN EVALUATION OF L-ATTRIBUTED DEFINITION

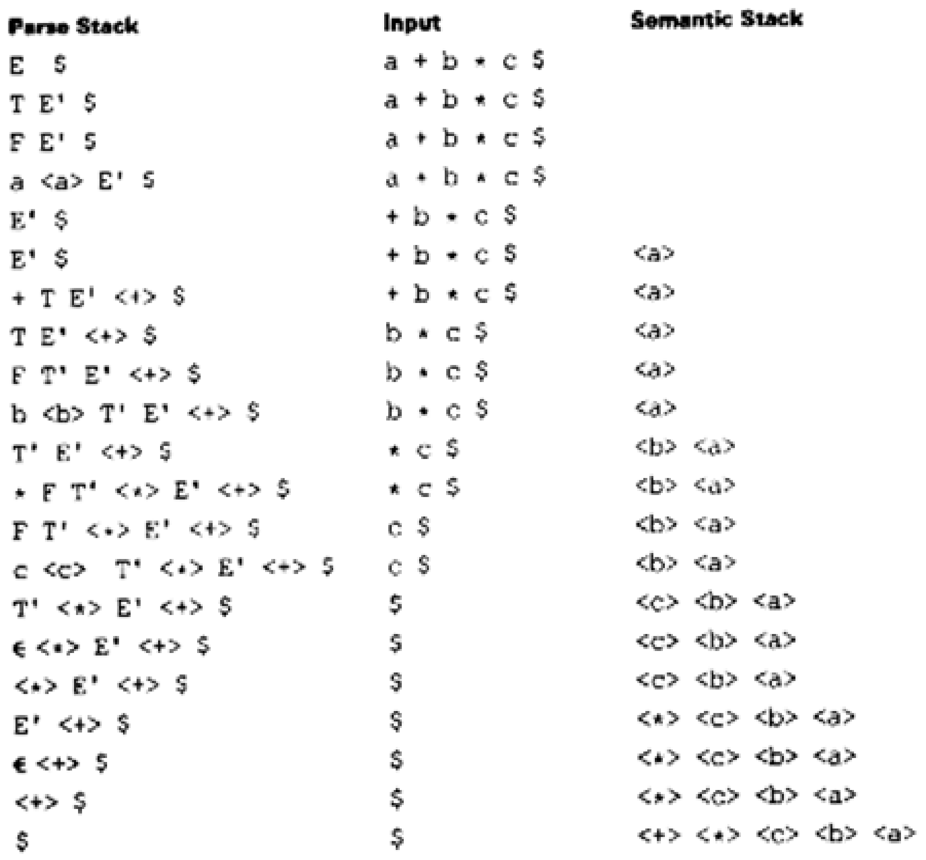
A top-down parser can evaluate attributes as it parses if the attribute values can be computed in atop-down fashion. Such attribute grammars are termed *L-Attributed*.



EXAMPLE 6 Converting from infix to postfix via an action symbol

```
E → TE'
E → T
E' → + T E' <+>
       | ε
T → F T'
T → F
T' → * F T' <*>
       | ε
F → (E)
      | Lit   <Lit>
      | Id    <Id>
```

We parse and translate *a* + *b* * *c*. The top is on the left for both stacks.

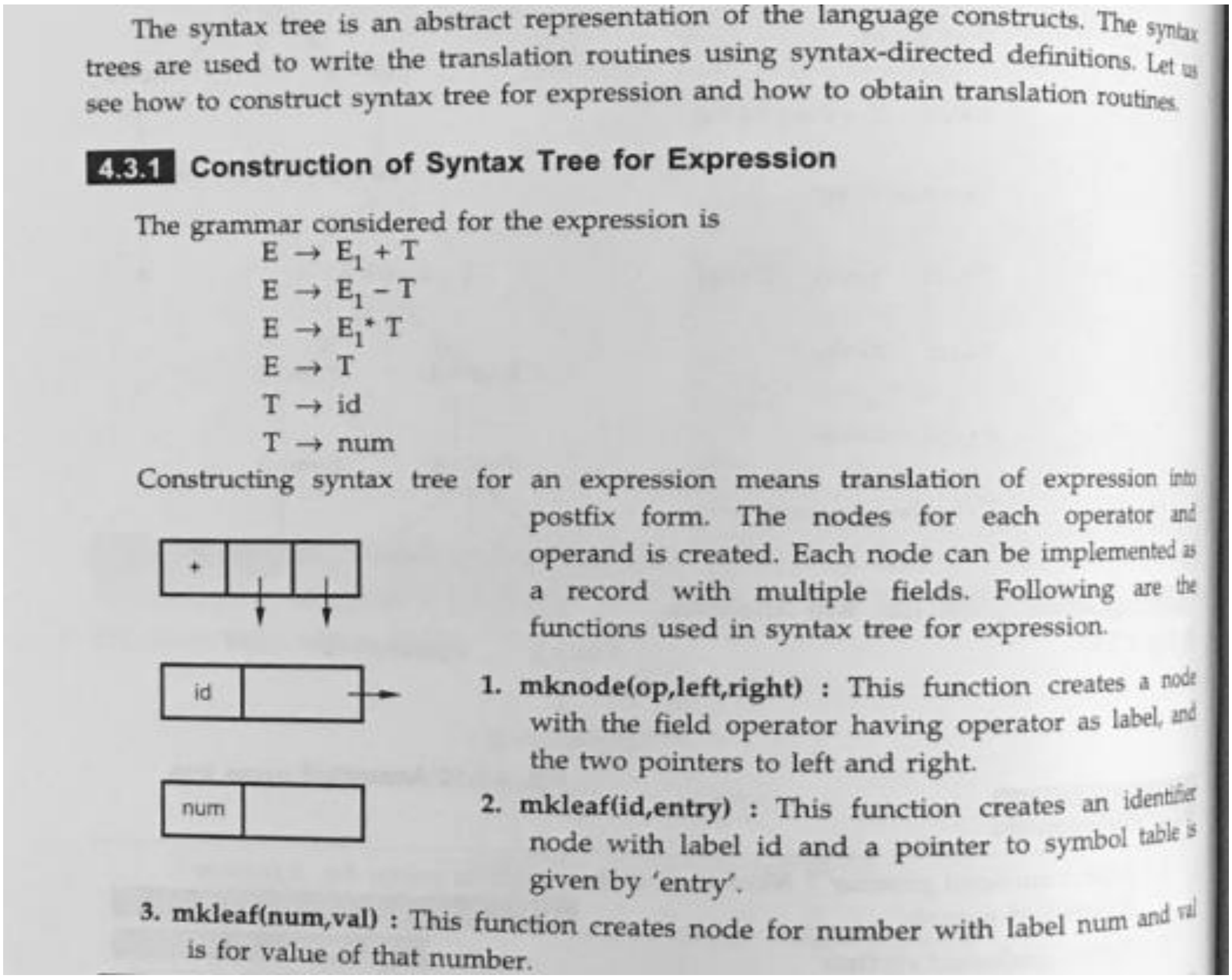


Parse Stack	Input	Semantic Stack
E $	a + b * c $	
T E' $	a + b * c $	
F E' $	a + b * c $	
a <a> E' $	a + b * c $	
E' $	+ b * c $	
E' $	+ b * c $	<a>
+ T E' <+> $	+ b * c $	<a>
T E' <+> $	b * c $	<a>
F T' E' <+> $	b * c $	<a>
b <b> T' E' <+> $	b * c $	<a>
T' E' <+> $	* c $	<b> <a>
* F T' <*> E' <+> $	* c $	<b> <a>
F T' <*> E' <+> $	c $	<b> <a>
c <c> T' <*> E' <+> $	c $	<b> <a>
T' <*> E' <+> $	$	<c> <b> <a>
ε <*> E' <+> $	$	<c> <b> <a>
<*> E' <+> $	$	<c> <b> <a>
E' <+> $	$	<*> <c> <b> <a>
ε <+> $	$	<*> <c> <b> <a>
<+> $	$	<*> <c> <b> <a>
$	$	<+> <*> <c> <b> <a>

When the semantic stack is popped, the translated string is:

a b c * +

CONSTRUCTION OF SYNTAX TREE:



The syntax tree is an abstract representation of the language constructs. The syntax trees are used to write the translation routines using syntax-directed definitions. Let us see how to construct syntax tree for expression and how to obtain translation routines.

4.3.1 Construction of Syntax Tree for Expression

The grammar considered for the expression is

$E \rightarrow E_1 + T$

$E \rightarrow E_1 - T$

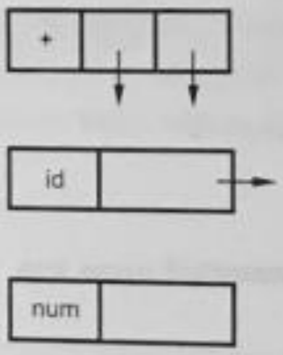
$E \rightarrow E_1 * T$

$E \rightarrow T$

$T \rightarrow id$

$T \rightarrow num$

Constructing syntax tree for an expression means translation of expression into postfix form. The nodes for each operator and operand is created. Each node can be implemented as a record with multiple fields. Following are the functions used in syntax tree for expression.



1. **mknode(op,left,right)** : This function creates a node with the field operator having operator as label, and the two pointers to left and right.
2. **mkleaf(id,entry)** : This function creates an identifier node with label id and a pointer to symbol table is given by 'entry'.
3. **mkleaf(num,val)** : This function creates node for number with label num and val is for value of that number.

Solution :

Step 1 : Convert the expression from infix to postfix xy*5–z+.

Step 2 : Make use of the functions mknode(),mkleaf(id,ptr) and mkleaf(num,val).

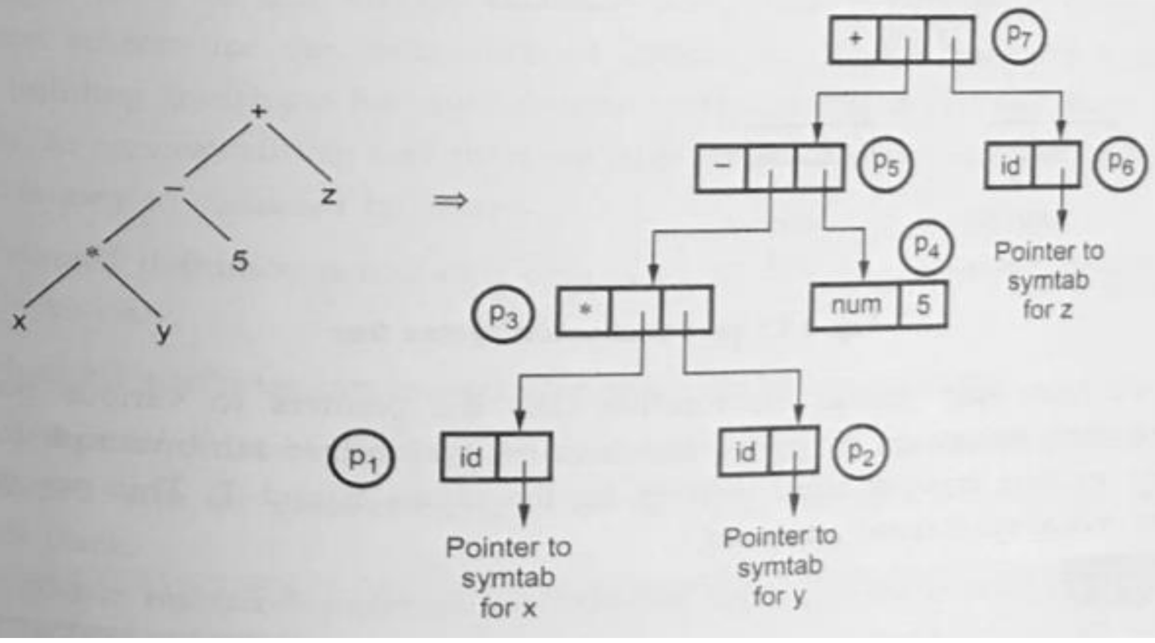
Step 3 : The sequence of function calls is given.

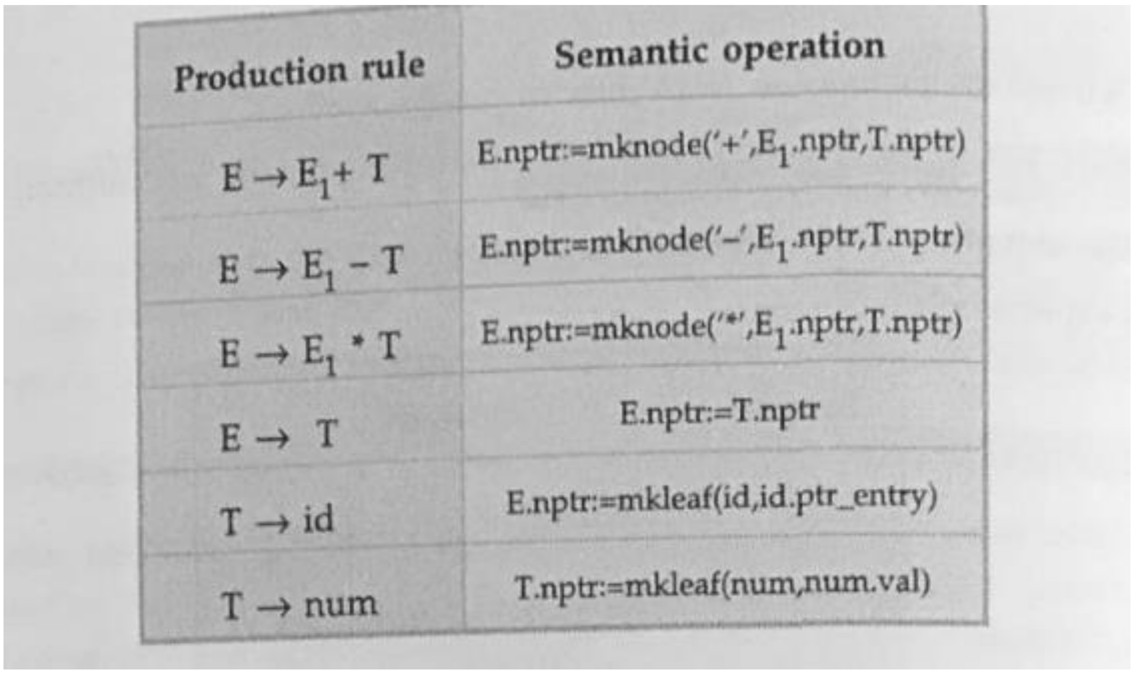
Postfix expression xy*5 – z+

Symbol	Operation
x	p_1=mkleaf(id, ptr to entry x)
y	p_2= mkleaf(id, ptr to entry y)
*	p_3=mknode(*,p_1,p_2)
5	p_4= mkleaf(num,5)
–	p_5=mknode(–, p_3,p_4)
z	p_6= mkleaf(id, ptr to entry z)
+	p_7= mknode(+, p_5,p_6)

Consider the string x*y–5+z and let us draw the syntax tree.

The syntax-directed definition for the above grammar is as given below.





Production rule	Semantic operation
$E \rightarrow E_1 + T$	E.nptr:=mknode('+',E_1.nptr,T.nptr)
$E \rightarrow E_1 - T$	E.nptr:=mknode('–',E_1.nptr,T.nptr)
$E \rightarrow E_1 * T$	E.nptr:=mknode('*',E_1.nptr,T.nptr)
$E \rightarrow T$	E.nptr:=T.nptr
$T \rightarrow id$	E.nptr:=mkleaf(id,id.ptr_entry)
$T \rightarrow num$	T.nptr:=mkleaf(num,num.val)

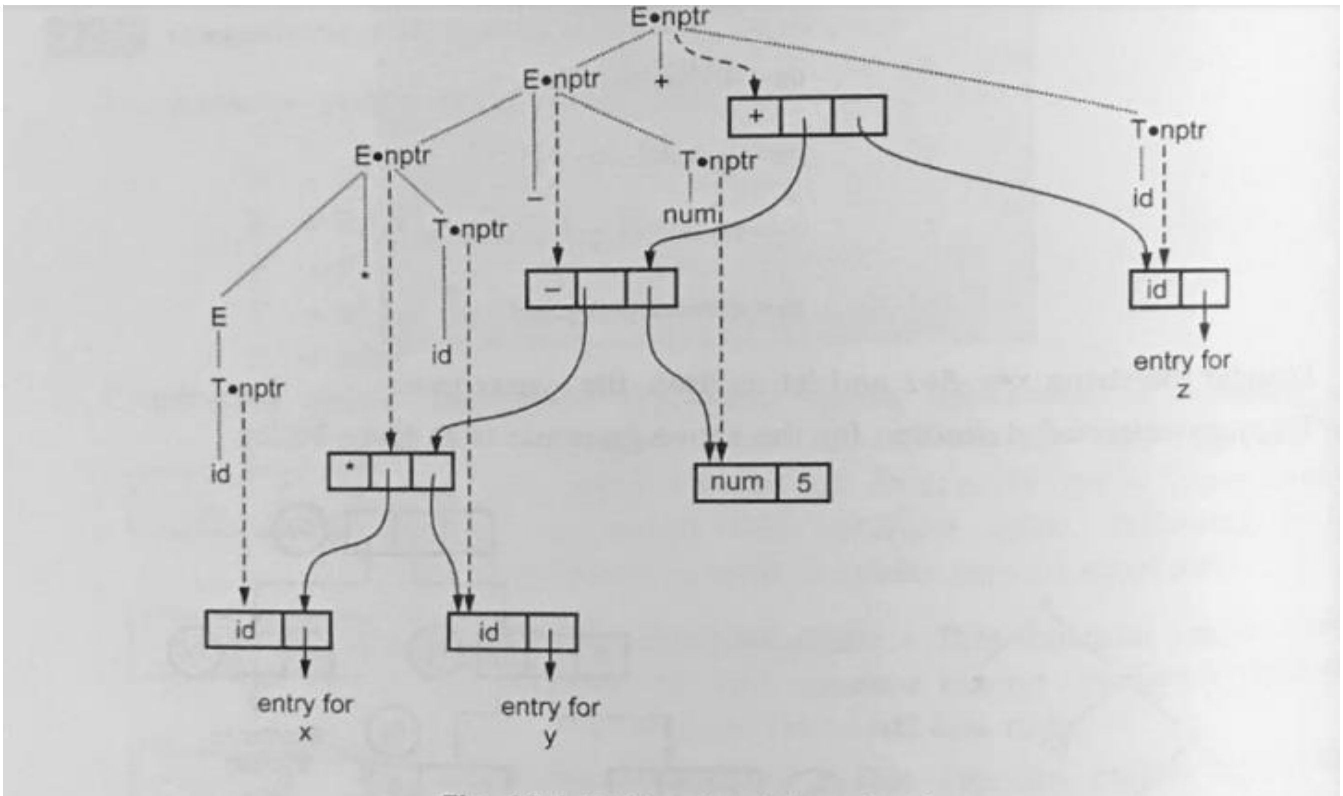


Fig. 4.3.1 (a) Constructed syntax tree

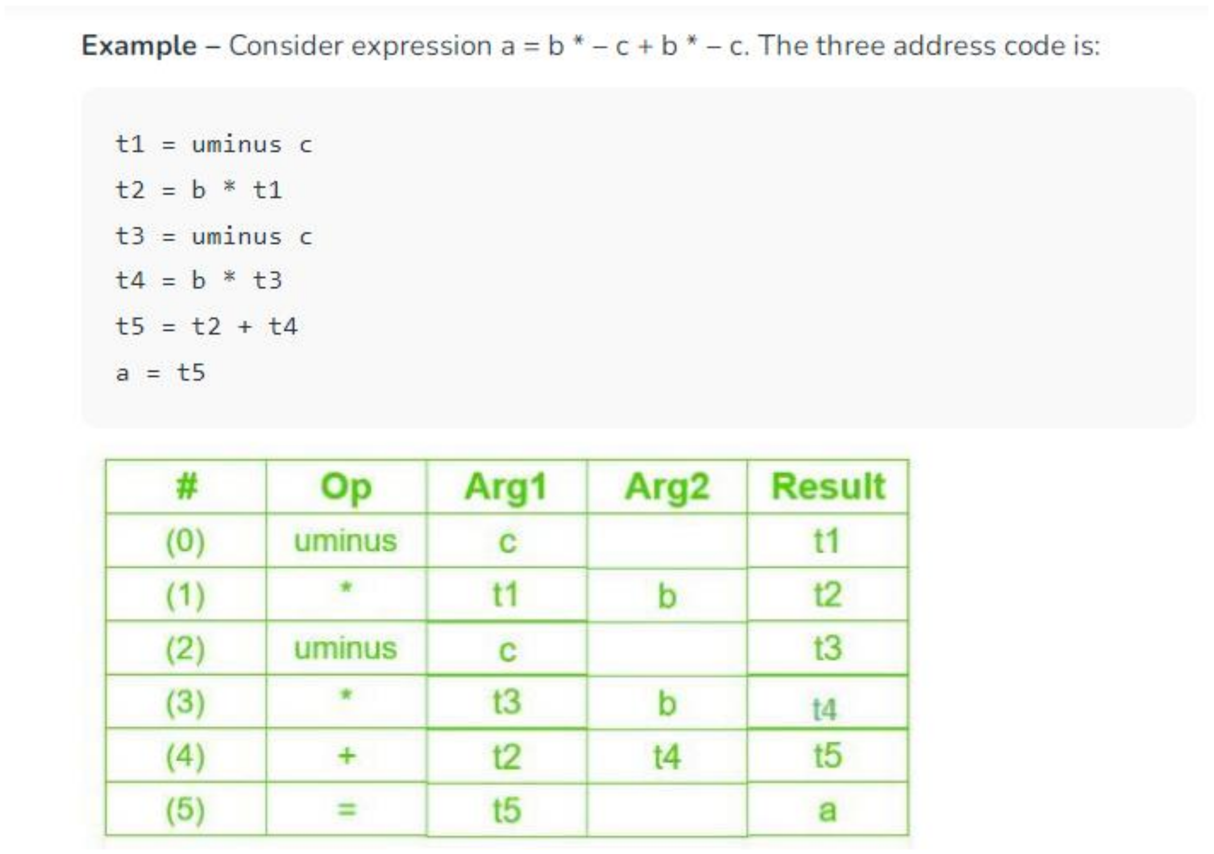
THREE ADDRESS CODE

Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code.

(Types of Three Address Code)Implementation of Three Address Code
There are 3 representations of three address code namely
1. Quadruple

2. Triples
3. Indirect Triples

Quadruple – It is a structure which consists of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.



Example – Consider expression a = b * – c + b * – c. The three address code is:

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

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

2. Triples – This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

Example – Consider expression a = b * – c + b * – c

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

3. Indirect Triples – This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Example – Consider expression a = b * – c + b * – c

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

List of pointers to table

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)