4.1 Representation of Intermediate Codes

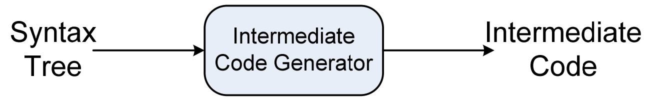


Fig. Input and Output of Intermediate Code Generator

It is practically possible to translate the source program directly into the target language that a typical programming language compiler does for us. On the other hand, there are significant benefits to have an intermediate i.e. machine-independent representation of a source language.

(a) A clear distinction between the machine-independent and machine-dependent elements of the compiler

(b) It facilitates to re-target for the new machine i.e. the implementation of language processors for a new machine requires to replace only the back-end of the compiler.

(c) It allows us to apply the machine independent code optimization techniques on newly generated intermediate codes.

(d) It covers the gap between the source language and target language.

⇒ **High Level Representation**: is closer to the source language, undemanding to generate from the source language, helps to implement the code optimization algorithm and may not be straight forward from the source language.

⇒ **Low Level Representation**: is closer to the target language, suitable for register allocation and instruction selection, easier for target language generation and optimization.

Finally, the intermediate code is a linearization of the syntax tree produced during syntax and semantic analysis. It is formed by breaking down the tree structure into sequential instructions, each of which is equivalent to a single, or small number of machine instructions. The equivalent machine code can then be generated (it might be required to use symbol tables etc) using such intermediate representation.

4.2 Types of Intermediate Code

(a) Postfix Notation- Operators must appears after operands and operator can either be unary, binary or ternary.

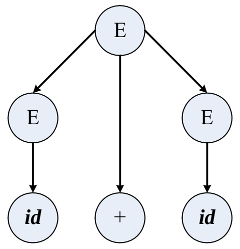
Example:

Expressions	*Postfix Notations*
1. a + b	`a, b, +`
2. a < b	`a, b, <`
3. goto 20	`20, goto`
4. if a < b goto 80	`a, b, <, 80`
5. a = b	`a, b, =`

The Grammar: (that parses *a* + *b*)

$E \rightarrow E + E$	[Action: print '+']
$E \rightarrow E * E$	[Action: print '*']
$E \rightarrow (E)$	
$E \rightarrow$ ***id***	[Action: print '***id***']

Output: *a, b,* +



(b) Abstract Syntax Tree- contains only terminal symbols i.e. abstract view of parse tree.

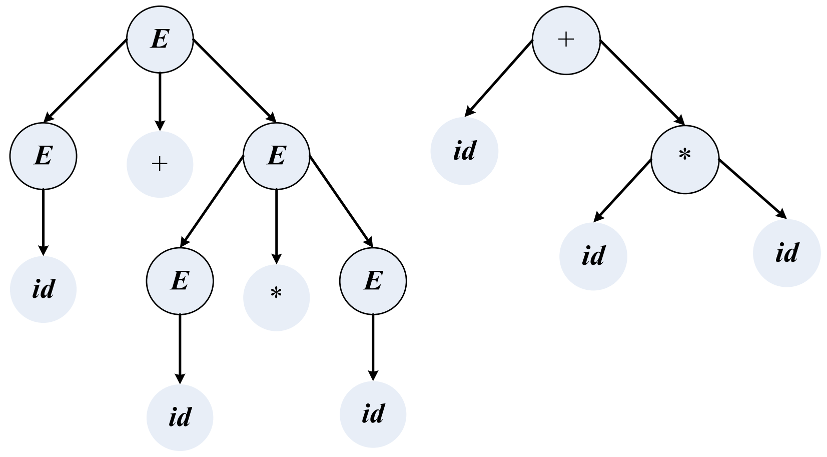


Fig. Abstract syntax tree of *id + id * id*.

(c) Quadruple Representation- Four address fields are allocated for each instruction. The number of operands accessed depends on the opcode used.

Address representation:

Op-Code	Op - 1	Op - 2	Result

Example:

Expressions	*Quadruple*
1. c = a + b	`+, a, b, c`
2. c = -b	`-b, --, --, c`
3. goto 20	`goto, --, --, 20`
4. if a < b goto 80	`< , a, b, 80`

5. Call P1(a1,a2,...,an)	`Param, a1, --, --`
	`Param, a2, --, --`
	`...`
	`Param, an, --, --`
	`Call, P1, n, --`

6. x = a + b * C	`*, b, c, T1`
	`+, a, T1, T2`
	`=, T2, --, x`

7. if a < b goto 20 else goto 30	`<, a, b, 20`
	`goto, --, --, 30`

(d) Triple Representation- has three address fields of each instruction and the result of the operation is referred by its position itself.

Address representation:

Op-Code	Op - 1	Op - 2

Example:

Expressions	*Triple*
1. t1 = a + b	(1) +, a, b
t2 = c + d	(2) +, c, d
t3 = t1 + t2	(3) +, (1), (2)
2. a = b * -c + b * -c	(1) -, c, --
	(2) *, b, (1)
	(3) -, c, --
	(4) *, b, (3)
	(5) +, (2), (4)
	(6) =, a, (5)

It is space efficient than quadruple and many instructions may not require the result. If we change the sequence number during optimization, the whole intermediate code need to be checked if any reference to be changed for modified triple that requires substantial computational overhead.

(e) Indirect Triple- It consist of a list of pointers to the triples, rather than a list of the triples themselves. During optimization, the compiler can move an instruction by reordering the instruction list without affecting the triples themselves i.e. we can change the triple pointer sequence without changing the triple physically.

Example:

Triple	*Execution instruction sequence*
(1) -, c, --	100 (1)
(2) *, b, (1)	101 (2)
(3) -, c, --	102 (3)
(4) *, b, (3)	103 (4)
(5) +, (2), (4)	104 (5)
(6) =, a, (5)	105 (6)
...	...

So, it avoids the major disadvantage of triple only. But it also looks like quadruple as we consider the space requirements. If we consider the computation time, quadruple demands more time for code movement during optimization as compared to the indirect triple. But, the execution itself will be faster in quadruple.

4.3 Syntax Directed Translation Scheme (STDS)

STDS uses modified production rule for generating intermediate code.

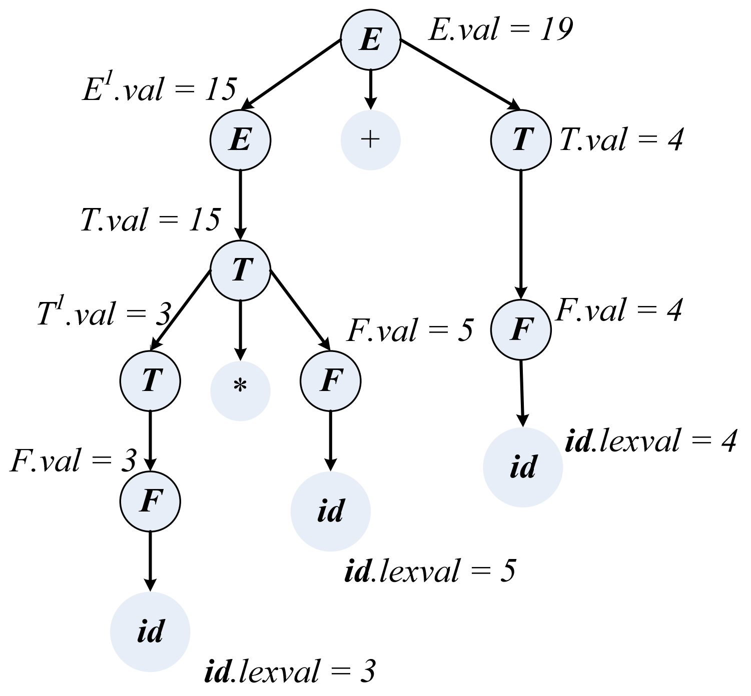
$$A \to \alpha, a_i$$

Where a_i (semantic action) is a function (set of statements) executed when the production rule $A \to \alpha$ is applied during parsing. ICG phase deals with the association of appropriate semantic actions such that during parsing the intermediate code is also generated in terms of syntactic structure of the source language.

We augment a grammar i.e. *attribute grammar* by associating attributes with each grammar symbol that describes its properties. With each production in a grammar, we give semantic rules/actions, which describe how to compute the attribute values associated with each grammar symbol in a production. The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree. An example of attribute grammar:

Production Rules	*Semantic Actions*
(1) E → E¹ + T	`E.val = E¹.val + T.val`
(2) E → T	`E.val = T.val`
(3) T → T¹ * F	`T.val = T¹.val * F.val`
(4) T → F	`T.val = F.val`
(5) F → (E)	`F.val = E.val`
(6) F → id	`F.val = id.lexval`

The grammar is based on the arithmetic expressions for + and * operators. The non-terminal have attribute *val* and the terminal id have the attribute *lexval* and is returned by lexical analyzer. An example of annotated parse tree for ***3 * 5 + 4***.



4.4 Quadruple Generation

Example: `x = a + b * (c * d)`

Let,

E.code ⇒ The intermediate code associated with *E*.

E.place ⇒ The identifier associated with *E*.

Now, the derived rules are:

1. Assignment statement → ***id*** = *E*
2. $E \rightarrow E^1 + E^2$
3. $E \rightarrow E^1 * E^2$
4. $E \rightarrow (E^1)$
5. $E \rightarrow$ ***id***
6. $E \rightarrow -E^1$

Rule1: Assignment statement → ***id*** = *E*

```
As.place = id.place
As.code = As.code||As.place=E.place
```

Rule 2: $E \rightarrow E^1 + E^2$

```
T = new Temp()
E.place = T
E.code = E¹.code || E².code || '+'
       = E¹.place + E².place
```

Rule 3: $E \rightarrow E^1 * E^2$

```
T = new Temp()
E.place = T
E.code = E¹.code || E².code || '*'
       = E¹.place * E².place
```

Rule 4: $E \rightarrow (E^1)$

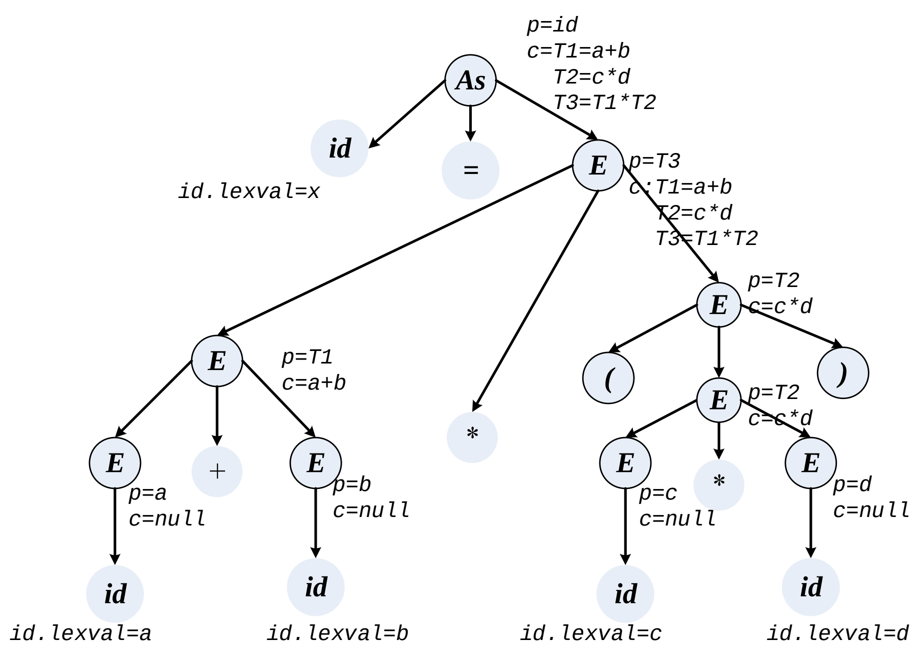
```
E.code  = id.place
E.place = E¹.place
```

Rule 5: $E \rightarrow$ ***id***

```
E.place  = id.place
E.code   = null
```

Rule 6: $E \rightarrow -E^1$

```
T = new Temp()
E.place = T
E.code = E.code||E.place=-E¹.place
```



Result: Intermediate code (Quadruple)

```
T1 = a + b
T2 = c * d
T3 = T1 * T2
x  = T3
```

4.5 Handling Conditional Expression

Grammar:

$E \rightarrow id\ rel\_op\ id$

$E \rightarrow id$

A conditional expression can be represented as:

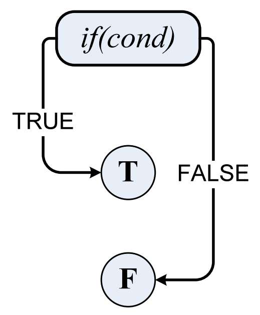
(a) Numerical Value (condition is evaluated as numerical value)

if(cond)

i.e. cond = 1 if cond = true

cond = 0 if cond = false.

(b) Control Flow



Example: `x = (a < b) + y`

Quadruple: `T1 = a < b`

`T2 = T1 + y`

Conversion:

```
99      ...
100     if a < b goto 103
101     T1 = 0
102     goto 104
103     T1 = 1
104     T2 = T1 + y
105     ...
```

Quadruple generation:

```
T = new Temp()
E.code = if id¹.place rel_op id².place
         goto next_quad + 3
T = 0
Goto next_quad + 2
T = 1
```

Here, `next_quad` is a relative variable.

4.6 Construction of Abstract Syntax Tree

Syntax tree are useful to represents the constructs like statements and expressions of a typical programming language where each node represent a construct and the children represent the meaningful component of the construct. For example: a node for an expression E_1+E_2 has label '+' i.e. *op-field* and two children representing the sub expressions E_1 and E_2 which are additional fields. There could be two cases:

(a) If the node is a *leaf*, an additional field holds the lexical value for the leaf. This is created by function `Leaf(op, val)`.

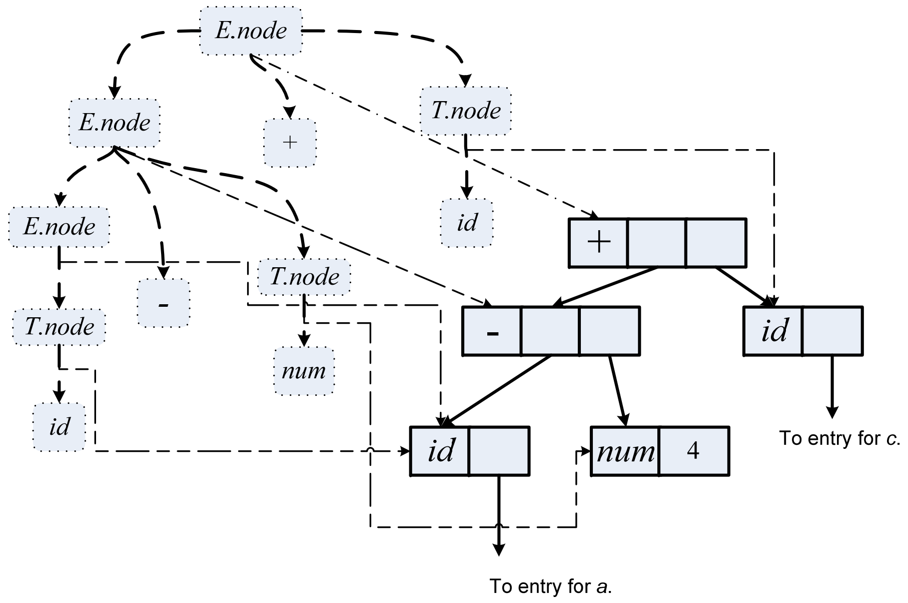
(b) If the node is *not leaf*, there are as many fields as the node has children in the syntax tree. This is created by function `Node(op, c₁, c₂,...,cₖ)`.

Example: Given the set of production rule below:

Production	*Semantic Rules*
E → E¹ + T	`E.node = new Node('+',E¹.node,T.node)`
E → E¹ - T	`E.node = new Node('-',E¹.node,T.node)`
E → T	`E.node = T.node`
T → (E)	`T.node = E.node`

T* → *id	`T.node = new Leaf(id, id.entry)`
T* → *num	`T.node = new Leaf(num, num.val)`

The Syntax tree for *a-4+c* using the above SDD is shown below.



<u>Steps to construct above Syntax Tree: (with bottom-up parsing)</u>

```
1. p1 = new Leaf(id, entry(a))
2. p2 = new Leaf(num, 4)
3. p3 = new Node('-' p1, p2)
5. p4 = new Leaf(id, entry(c))
5. p5 = new Node('+', p3, p4)
```