# Chapter 5. IC Generation

1. Introduction
2. IC Representations
   a) Abstract Syntax Tree (AST)
   b) Postfix Notation
   c) Three Address Code
3. Bottom-up Translations
4. Top-Down Translations

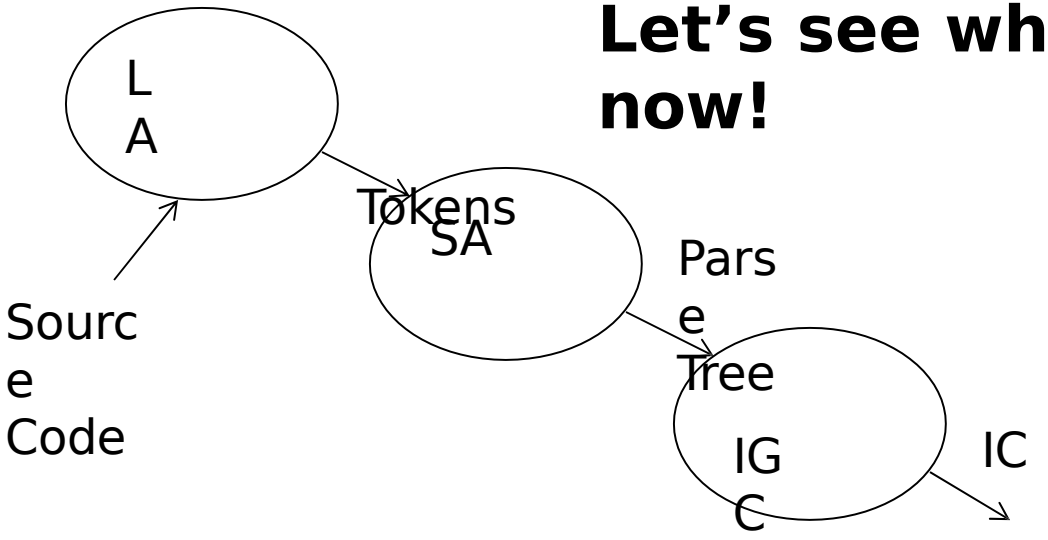
Mr. Param Venkat Vivek Kesireddy

Teaching Associate

Mail: pkesireddy@fullerton.edu

Inputs from Prof Doina Bein & Prof James Choi

# 5.1 Introduction




**Let's see where we are now!**

**kinds of IC:**

- Abstract Syntax Tree (AST)
- Three Address Code (3AC)
- Postfix Notation etc.

**Which one?**

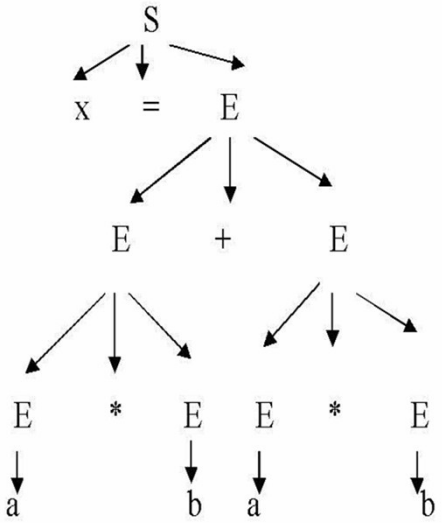
Depends on Optimization, Re-targetable
(with different back-ends)

# 5.2 IC representations

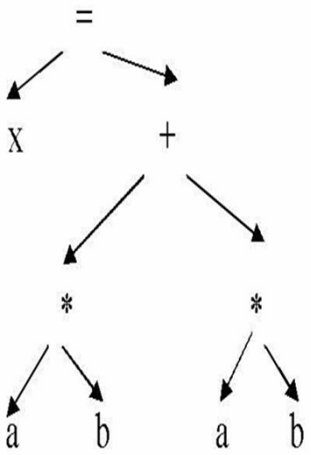
a) API

A form of parse tree, where all "unnecessary" parsing info is removed e.g., **x = a*b + a*b**

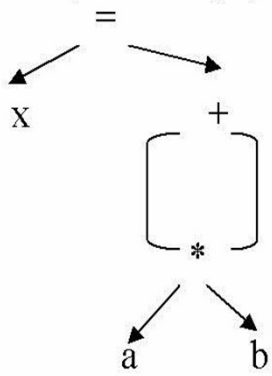
Parser Tree:




Remove all unnecessary NT (AST)




Another form (Directed Acyclic Graph)



**b) Postfix notation (Polish Reverse Notation)**
**Post order Traversal of the parse Tree**

**ab* ab* + x =**

**c) 3AC**

**This program breaks the Parse Tree down into elementary statements, where each statement has no more than 3 addresses (variables) and one operator =>**
**(will be our focus, since it is close to assembly)**

**T1**

**= a *b T2**

**= a*b**
**T3**

**Q: How do we generate the ICs?**

***One way:*** **from the parse tree**
***Another way:*** **During the SA, combine the SA and ICG which is called *Syntax Directed Translation* (SDT)**

**How?**
**Attach the " meaning" to each production, so called *Semantic action***
**E.g.**

| | | |
|---|---|---|
| **E** | **-> E +** | **{V (E1) = V(E2) +** |
| **E** | | **V(E3) }** |
| **E** | **-> id** | **{V (E) = V(id) }** |

**And whenever we recognize a production, we execute the attached action**

**5.3 Bottom-up Translation (SDT with Bottom-up Parsers)**

**a) Translation into AST**

**E.g. for Assignment Statement**

| | |
|---|---|
| **R1: S -> id = E(1)** | **{E = makeleaf (id);** |
| | **S = maketree (=, E, E(1)) }** |
| **R2: E (1) -> E (2) + E (3)** | **{E(1) = maketree (+, E (2),** |
| **R3: E (1) -> E (2) * E (3)** | **E(3) ) }** |
| **R4: E(1) -> ( E(2) )** | **{E(1) = maketree (*, E(2), E(3) ) }** |
| **R5: E -> id** | **{ E(1) = E (2) }** |
| | **{E = makeleaf (id) }** |

**Type:**

```
nodeptr =  treenode->;
treenode = record
                    token: tokentype;
                    left, right: nodeptr;
                endrecord
```

```
function makeleaf (t : tokentype)
   : nodeptr;
{
    new (leaf);
    with leaf-> do
        token = t;
        left = right = nil;
    endwith
        makeleaf = leaf;
    }
```

```
function  maketree  (op: tokentype;
   leftson, rightson: nodeptr): nodeptr;
{
  new (root); with
     root-> do
       token = op; left
           = leftson;
           right = rightson;
      endwith
           maketree =
      root;
   }
```

**Example of SDT using Bottom-up Parser**

**for x = (a+b) * c**

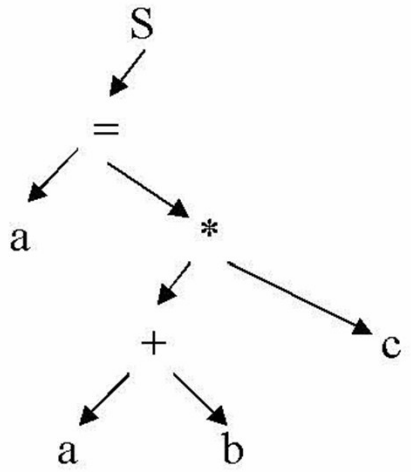
- **We need additional stack for SDT = Semantic Stack keeps information regarding SDT**

- **Idea: Use of a generic Bottom-up parser. We will reduce whenever we see a handle**

R1: S -> id = E(1) {E = makeleaf (id);
S = maketree (=, E, E(1)) }
R2: E (1) -> E (2) + E (3) {E(1) = maketree (+, E (2), E(3) ) }
R3: E (1) -> E (2) * E (3) {E(1) = maketree (*, E(2), E(3) ) }
R4: E(1) -> ( E(2) ) { E(1) = E (2) }
R5: E -> id {E = makeleaf (id) }

| Parsing Stack | Input | Prod. Used | Semantic action | Semantic Stack |
|---|---|---|---|---|
| $ | x= (a+b)*c $ | | | |
| $ x | = (a+b)*c $ | | | |
| $x= | (a+b)*c $ | | | |
| $x = ( | a+b)*c $ | | | |
| $x = (a | +b)*c $ | E -> id | E(1)= makeleaf(a); | E (1) |
| $x = (E | + b)*c$ | | | E(1) |
| $x =(E + | b)*c$ | | | E(1) |
| $x= (E+ b | )*c$ | E -> id | E(2) = makeleaf(b); | E(1) |
| $x =(E+E | )*c$ | E -> E + E | E(3) = maketree(+, E(1), E(2)) | E(2) E(1) |
| $ x = (E | )*c$ | | | E(3) |
| $x = (E) | *c$ | E -> (E) | E(4) = E(3) | E(4) |
| $x = E | *c$ | | | E(4) |
| $x = E* | c$ | | | E(4) |
| $x = E * c | $ | E -> id | E(5) = makeleaf(c) | E(5) |
| $x = E * E | $ | E = E*E | E(6) = maketree(*, E(4),E(5)) | E(5) E(4) |
| $x = E | $ | S -> id = E | E(7) = makeleaf (x); | E (6) |
| | | | S = maketree (=, E(7), E(6)) | E(7) E(6) |
| $ S | | | | S |

Results in a AST:




**Observation:**
**Semantic Stack and Parsing Stack work in tandem with the same symbol**

**_ => we can put all info in one stack**

## b) Translation into 3 AC

**Representation of 3 AC in a = b + c**

| OP | Oprnd1 | Oprnd2 | result |
|---|---|---|---|
| +<br>or<br>+ | b<br>(addresses<br>) 201 | c<br><br>300 | a<br><br>309 |

R1: S -> id = E { S.loc = makequad (=, E.loc, id.loc);
R2: E(1) -> E(2) + E(3) {E(1).loc = makequad (+, E(2).loc,
R3: E(1) -> E(2) * E(3) E(3).loc) }
R4: E(1) -> ( E(2) ) {E(1).loc = makequad (*, E(2).loc, E(3).loc) }
R5: E -> id {E(1).loc = E(2).loc}
{E.loc = get_address(id) }

**function makequad (op: char, addr1, addr2 :integer) : integer**

```
{
    If op = "=" then
        Temp = addr2;
        addr2 = 0;
else temp = gettemp();              /* get next a temporary
    variable */ iquad = iquad ++; /* index to QUAD table */
Quad [iquad].op = op;
Quad [iquad].oprnd1 = addr1;
    Quad [iquad].oprnd2 = addr2;
    Quad [iquad].result  = temp;
    makequad = temp;
}
```

**Ex. Generate 3Acs for x = (a+b) * c, assume that address location are a=20, b = 30, c = 40 , x = 10 and temporary starts from 300**

| | |
|---|---|
| R1: S -> id = E | { S.loc = makequad (=, E.loc, id.loc); |
| R2: E(1) -> E(2) + E(3) | {E(1).loc = makequad (+, E(2).loc, |
| R3: E(1) -> E(2) * E(3) | E(3).loc) } |
| R4: E(1) -> ( E(2) ) | {E(1).loc = makequad (*, E(2).loc, E(3).loc) } |
| R5: E -> id | {E(1).loc = E(2).loc} |
| | {E.loc = get_address(id) } |

| Parsing Stack | Input | Prod. Used | Seman.action |
|---|---|---|---|
| $ | x= (a+b)*c $ | | |
| $ x | =(a+b)*c $ | | |
| $x= | (a+b)*c $ | | |
| $x = ( | a+b)*c $ | | |
| $x = (a | +b)*c $ | E -> id | E.loc = id.loc = 20 |
| $x = (E[20] | + b)*c$ | | |
| $x =(E + | b)*c$ | | |
| $x= (E+ b | )*c$ | | E.loc = id.loc = 30 |
| $x =(E[20]+ E[30] | | | E(3) =makequad (+, 20,30) = 300 |
| $ x = (E[300] | E -> | | |
| $x = (E) | id | E -> (E) | E(1).loc=E(2).loc = 300 |
| $x = E[300] | )*c$ | | |
| $x = E* | | | |
| $x = E * c | E -> | E -> id | E.loc = id.loc = 40 |
| $x = E [300]* E[40] | E + | E -> E*E | E.loc= makequad(*,300,40)=301 |
| $x = E [301] | E | S -> id = E | S.loc = makequad (=, 301, 10) = 10 |
| $S [10] | )*c$ | | |
| | *c$ | | |
| | *c$ | | |
| | c$ | | |

| Op | Oprnd1 | | Oprnd2 | | Result | |
|---|---|---|---|---|---|---|
| + | 20 | (a) | 30 | (b) | 300 | (t1) |
| * | 300 | (t1) | 40 | (c) | 301 | (t2) |
| = | 301 | (t2) | 0 | | 10 | (x) |

```
t1 = a
+b t2 =
t1 * c x =
t2
```

**c)**

**Translation into the postfix notation**

**S -> id = E { output (id); output (=)}**
**E -> E + E { output (+) }**
**E -> E * E {output (*) }**
**E -> ( E ) { - }**
**E -> id {output (id) }**

| Parsing Stack | Input | Prod. Used | Seman. action | Seman. Stack |
|---|---|---|---|---|
| $ | x = (a+b)*c $ | | | |
| $ x | =(a+b)*c $ | | | |
| $x = | (a+b)*c $ | | | |
| $x = ( | a+b)*c $ | | | |
| $x = (a | +b)*c $ | E -> id | output(a) | |
| $x = (E | + b)*c$ | | | |
| $x =(E + | b)*c$ | | | |
| $x= (E+ b | )*c$ | E -> id | output(b) | |
| $x =(E+E | )*c$ | E -> E + E | output (+) | |
| $ x = (E | )*c$ | | | |
| $x = (E) | *c$ | E -> (E) | - | |
| $x = E | *c$ | | | |
| $x = E* | c$ | | | |
| $x = E * c | $ | E -> id | output (c ) | |
| $x = E * E | $ | | output(*) | |
| | E = E*E | | | |
| $x = E | $ | s | output (x); output (=) | |
| | -> id = E | | | |

**Output:**
**ab+c*x=**

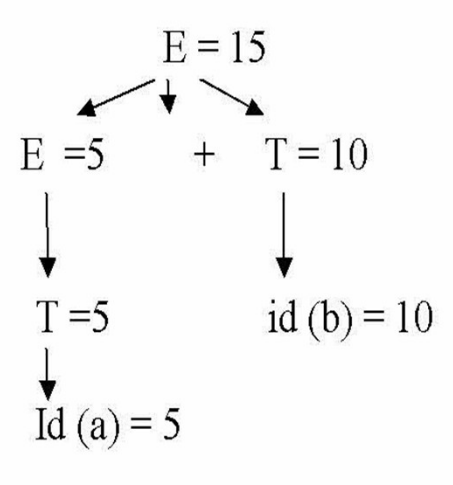
## 5.4 Top-down translation

**Top-down is not easy (as straight forward) due to the change of the original grammar to remove the left-recursion.**

**Consider the bottom-up translation:**

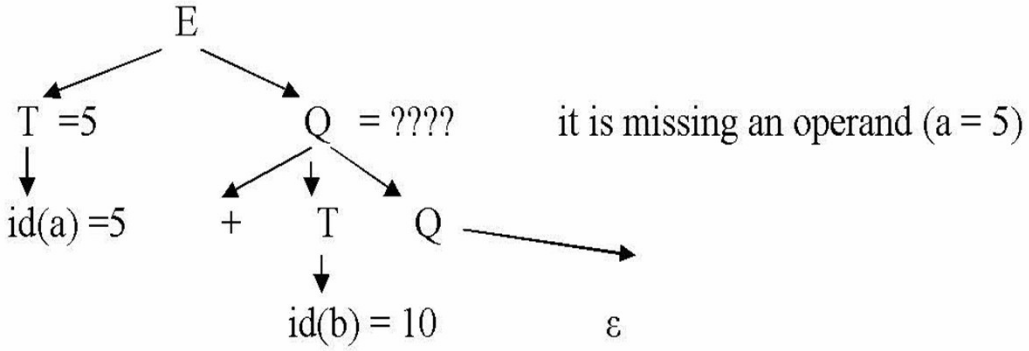
**E -> E + T {V(E1) = V(E2) + V(T) }**
**E -> T {V (E) = V (T)}**
**T -> id {V(T) = V(id) }**

**And we have a string a +b with a = 5 and b = 10**



**However, in top-down we have to change the grammar:**

**E -> T Q**
**Q -> +**
**TQ Q ->**
**ϵ**
**T -> id**








**t h**

**So, in order to make both operands available for the operation, we have to transfer some values from sibling nodes**

This leads that each node has two attributes/values
Synthesized and Inherited

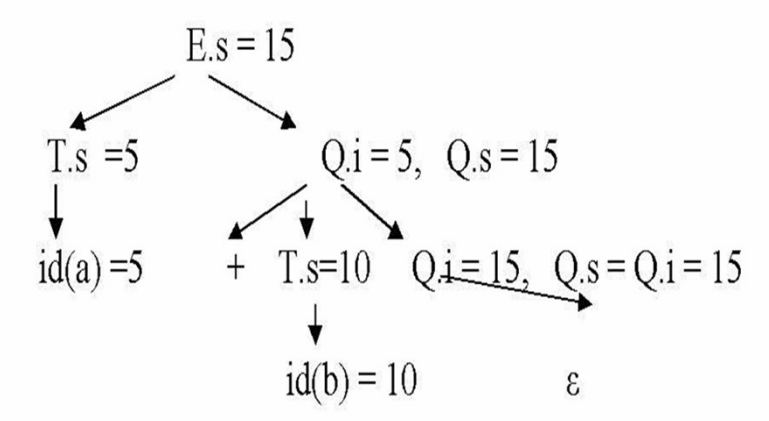
**Def:**
**Synthesized attributes(values):** attributes that are transferred from children nodes

**Inherited attributes:** attributes that are transferred from parent or sibling notes.

**Now the Semantic Actions for Top-Down parsing:**

**R1: E -> T {A1: Q.i = T.s} Q {A2: E.s = Q.s}**
**R2: Q1 -> + T {A3: Q2.i = Q1.i + T.s } Q2 { A4: Q1.s = Q2.s} R3: Q -> ϵ { A5: Q.s = Q.i}**
**R4: T -> id {A6: T.s = V(id)}**

Now let's consider: a +b




**So, the botto m-line is:**

**1.We may have to do more than one semantic actions per production**

**a) Top-down Translation into AST**

**Original Grammar**
**S -> id = E**
**E -> E + T**
**E -> T**
**T -> id**

**After removing the left recursion on**
**R1: S -> id = E1 { A1: E.s = makeleaf (id); S.s = maketree(=, E.s, E1.s ) } R2: E -> T {A2: Q.i = T.s} Q {A3: E.s = Q.s}**
**R3: Q1 -> + T {A4: Q2.i = maketree (+, Q1.i, T.s)} Q2 {A5: Q1.s =**

## Let's remember how Top-down parser worked from chapter 3

| | id | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | TE' | | | TE' | | |
| E' | | +TE' | | | ε | ε |
| T | FT' | | | FT' | | |
| T' | | ε | *FT' | | ε | ε |
| F | id | | | (E) | | |

**Ex: String b +**

| ***Stack*** | ***Input*** | ***Action*** |
|---|---|---|
| **$ E** | **b+c$** | **pop(E), Push (E', T)** |
| **$E'T** | **b+c$** | **pop(T), push(T', F)** |
| **$E'T'F** | **b+c$** | **pop(F), push (id);** |
| **$E'T' id** | **b+c$** | **pop(id), lexer();** |
| **$E'T'** | **+c$** | **pop(T'); push (ε)** |
| **$E'** | **+c$** | **pop(E'); push (E', T, +)** |
| **$E'T+** | **+c$** | **pop(+), lexer()** |
| **$E'T** | **c$** | **pop(T), push (T',F)** |
| **$E'T'F** | **c$** | **pop(F); push (id);** |
| **$E'T'id** | **c$** | **pop(id), lexer()** |
| **$E'T'** | **$** | **pop(T'), push (ε)** |
| **$E'** | **$** | **pop(E'), push (ε)** |
| **$** | **$** | **Stack empty** |

## Let's use a generic top-down parser

## (uses two stacks: parsing and Semantic Stack)

1. Push $ at the EOS and also push S on TOS
2. Push the starting symbol on TOS
3. While (Parsing stack not empty) and (no error) do

    Let t = TOS symbol and i incoming token

    If t is terminal then

        If t = i /*incoming token */ then pop(t) and call lexer()

        else error

    else if t = NT then

        pop(t);

        push RHS symbols backwards including the Semantic Actions

    else perform the semantic action and push the result on Semantic Stack

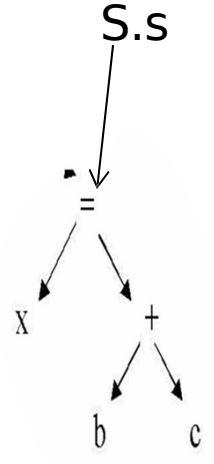
    endwhile

R1: S -> id = E1 { A1: E.s = makeleaf (id); S.s = maketree(=, E.s, E1.s ) } R2: E -> T {A2: Q.i = T.s} Q {A3: E.s = Q.s}
R3: Q1 -> + T {A4: Q2.i = maketree (+, Q1.i, T.s)} Q2 {A5: Q1.s = Q2.s} R4: Q -> ε { A6: Q.s = Q.i}
R5: T -> id {A7: T.s = makeleaf (id)}

Ex.

| P. Stack | input | S. Actions | S. stack |
|---|---|---|---|
| $ S | x= b+c$ | | |
| $ A1, E=id | x = b+c$ | | x |
| $A1, E = | = b+c$ | | |
| $A1, E | b+c$ | | |
| $A1,A3,Q,A2,T | b+c$ | | |
| $A1,A3,Q,A2,A7,id | b+c$ | | b, x |
| $A1,A3,Q,A2,A7 | +c$ | A7: T.s = makeleaf(b) | T.s, x |
| $A1, A3,Q, A2 | +c$ | A2: Q.i = T.s | Q.i, x |
| $A1,A3,Q | +c$ | | |
| $A1,A3,A5,Q,A4,T,+ | +c$ | | |
| $A1,A3,A5,Q,A4,T | c$ | | |
| $A1,A3,A5,Q,A4,A7,id | c$ | | c,Q.i,x |
| $A1,A3,A5,Q,A4,A7 | $ | A7: T.s =makeleaf(c) Q.i, x | T.s, |
| $A1,A3,A5,Q,A4 | $ | A4: Q2.i = maketree(+,Q1.i,T.s) x | Q2.i, |
| $A1,A3,A5,Q | $ | | |
| $A1,A3,A5,A6 | $ | A6: Q.s = Q.i | Q2.s, x |
| $A1,A3,A5 | $ | A5: Q1.s = Q2.s | Q1.s, x |
| $A1, A3 | $ | A3: E.s = Q.s | E1.s, |
| $A1 | $ | A1: E.s = makeleaf (x); S.s = maketree(=,E.s, E1.s) | x |

**Example: x = b + c**



**b) Translation into 3AC**

**S -> id = E {A1: S.s = makequad (=, E.s, id.loc)}**
**E -> T {A2: Q.i = T.s}**
**Q {A3: E.s = Q.s}**
**Q -> + T {A4: Q2.i = makequad (+, Q1.i, T.s)}**
**Q {A5: Q1.s = Q2.s}**
**- T {A6: Q2.i = makequad (-, Q1.i, T.s)}**
**Q {A7: Q1.s = Q2.s}**
**ε {A8: Q.s = Q.i}**
**T -> id {A9: T.s = id.loc}**

**R1: S -> id = E1 { A1: E.s = makeleaf (id); S.s = maketree (=, E.s, E1.s ) }**
**R2: E -> T {A2: Q.i = T.s} Q {A3: E.s = Q.s}**
**R3: Q1 -> + T {A4: Q2.i = maketree (+, Q1.i, T.s)} Q2 {A5: Q1.s = Q2.s}**
**R4: Q -> ε { A6: Q.s = Q.i}**
**R5: T -> id {A7: T.s = makeleaf (id)}**

| P. Stack | input | S. Actions | S. stack |
|---|---|---|---|
| $S | x= b+c$ | | |
| $ A1, E=id | x = b+c$ | | x |
| $A1, E = | = b+c$ | | |
| $A1, E | b+c$ | | |
| $A1,A3,Q,A2,T | b+c$ | | |
| $A1,A3,Q,A2,A7,id | b+c$ | | b, x |
| $A1,A3,Q,A2,A7 | +c$ | A7: T.s = id.loc | T.s (b), x |
| $A1, A3,Q, A2 | +c$ | A2: Q.i = T.s | Q.i (b), x |
| $A1,A3,Q | +c$ | | |
| $A1,A3,A5,Q,A4,T,+ | +c$ | | |
| $A1,A3,A5,Q,A4,T | c$ | | |
| $A1,A3,A5,Q,A4,A7,id | c$ | | c,Q.i,x |
| $A1,A3,A5,Q,A4,A7 | $ | A7: T.s = id.loc x | T.s ( c) ,Q.i, |
| $A1,A3,A5,Q,A4 | $ | A4: Q2.i = makequad(+, b,c) | Q2.i (t1), x |
| $A1,A3,A5,Q | $ | | |
| $A1,A3,A5,A6 | $ | A6: Q.s = Q.i x | Q2.s (t1), |
| $A1,A3,A5 | $ | A5: Q1.s =Q2.s x | Q1.s (t1), |
| $A1, A3 | $ | A3: E.s = Q.s x | E1.s (t1), |

| Op | Oprnd1 | Oprnd2 | Result |
|---|---|---|---|
| + | b | c | t1 |
| = | t1 | | x |

**c) Translation into postfix**

S -> id = E {A1:output (id, =)}

E -> T {A2: - } Q {A3: - }

Q -> + T { A4: output (+)} Q{A5: - }

Q -> ϵ {A6: - }

T -> id { A7: output (id) }


**x = b+c**

| P. | P.Stack | input | S. Actions | S. stack |
|---|---|---|---|---|
| | $S | x= b+c$ | | |
| | $ A1, E=id | x = b+c$ | | |
| | $A1, E = | = b+c$ | | x |
| | $A1, E | b+c$ | | |
| | $A1,A3,Q,A2,T | b+c$ | | |
| | $A1,A3,Q,A2,A7,id | b+c$ | | |
| | $A1,A3,Q,A2,A7 | +c$ | A7: output (b) | |
| | $A1, A3,Q, A2 | +c$ | | |
| | $A1,A3,Q | +c$ | | |
| | $A1,A3,A5,Q,A4,T,+ | +c$ | | |
| | $A1,A3,A5,Q,A4,T | c$ | | |
| | $A1,A3,A5,Q,A4,A7,id | c$ | | |
| | $A1,A3,A5,Q,A4,A7 | $ | A7: output (c) | x |
| | $A1,A3,A5,Q,A4 | $ | A4: output(+) | |
| | $A1,A3,A5,Q | $ | | |
| | $A1,A3,A5,A6 | $ | | x |
| | $A1,A3,A5 | $ | | x |
| | $A1, A3 | $ | | x |
| | $A1 | $ | A1: output | x= |

## Ex. Implementation of Expression Calculator:

E -> T {Q.i = T.s} Q {E.s = Q.s}
Q1 -> + T {Q2.i = Q1.i + T.s } Q2 { Q1.s = Q2.s} Q -> ϵ { Q.s = Q.i}
T -> id {T.s = V(id)}

```
procedure E (E.s)
{
  T (T.s);
  Q.i = T.s;
  Q (Q.i, Q.s);
  E.s = Q.s
}
```

```
procedure  T (T.s)
{
    If token = id then
       Begin
       T.s = value(id);
       Lexer()
       End
}
```

```
procedure Q (Q1.i, Q1.s)
{
   If token = +  then
      {
      Lexer();
      T (T.s);
      Q2.i = Q1.i + T.s;
      Q (Q2.i, Q2.s);
      Q1.s = Q2.s;
      }
   else  if token in Follow(Q)
         then Q1.s = Q1.i
else  error (….)
}
```

**Ex: a+b with a =5 and b = 7**

**E N D**