Unit V : Semantic Analysis And Storage Allocation

- **Need, Syntax Directed Translation, Syntax Directed Definitions**, Translation of assignment Statements, iterative statements, Boolean expressions, conditional statements, Type Checking and Type conversion.
- Intermediate Code Formats: Postfix notation, Parse and syntax tress, Three address code,
- Quadruples and triples.

Syntax Directed Translation, Syntax Directed Definitions and SDT Schemes

SDT, SDD and SDTS

- **Syntax Directed Translations**
  - Syntax Directed Definitions
  - Implementing Syntax Directed Definitions
    - Dependency Graphs
    - S-Attributed Definitions
    - L-Attributed Definitions
  - Translation Schemes

Semantic Analysis

- Semantic Analysis **computes additional information** related to the meaning of the program once the syntactic structure is known.
- In typed languages as C, semantic analysis involves adding information to the symbol table and performing type checking.
- The information to be computed is beyond the capabilities of **standard parsing techniques**, therefore it is not regarded as syntax.
- As for Lexical and Syntax analysis, also for Semantic Analysis we need both a **Representation Formalism** and an Implementation Mechanism.
- As representation formalism - is **Syntax Directed Translations**

- **Syntax Directed Translations**
  - Syntax Directed Definitions
  - Implementing Syntax Directed Definitions
    - Dependency Graphs
    - S-Attributed Definitions
    - L-Attributed Definitions
  - Translation Schemes

Introduction to Syntax Directed Translation (SDT)

- SDT uses the principle that - meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree.

- Using Syntax Directed Translations we specify translations for programming language constructs guided by context-free grammars

Syntax Directed Translation (SDT)

- Method used in compiler design where the translation of a programming language's syntax into machine code or another representation is driven by the grammar's production rules.
- In SDT, Actions are associated with productions in the grammar. These actions are to be taken when a particular production is used during parsing.
- SDT can be used to do the following action - generate intermediate code, optimize code, or perform other tasks alongside parsing
- SDT puts **program fragments** within the production bodies themselves; is more efficient and also easy to implement

A Simple Arithmetic Grammar

E -> E + T | T

T -> T * F | F

F -> (E) | id

SDT SAMPLE Actions **for the above**

E -> E1 + T2 {E.code = E1.code || T2.code || “ADD"}

T -> T1 * F2 {T.code = T1.code || F2.code || “MULT"}

F -> (E1) {F.code = E1.code}

F -> id {F.code = "LOAD " || id.lexeme}

...contd..Introduction to Syntax Directed Translation

- There are **two notations for attaching semantic rules**:
  - 1. Syntax Directed Definitions (**SDD**). **High-level specification** hiding many implementation details (also called **Attribute Grammars**).
  - 2. Translation Schemes(**SDTS**)**. More implementation oriented**: Indicate the order in which semantic rules are to be evaluated.
  - A grammar specification embedded with actions to be performed is called a ***<u>syntax-directed translation scheme(SDTS)</u>***

SDT Schemes:

- Specific plans or strategies for implementing syntax-directed translation.
- Outlines how semantic rules are embedded within the grammar's productions and how they are executed during parsing.
- Can involve various techniques such as attribute evaluation orders, attribute propagation strategies, and methods for handling conflicts or ambiguities.
- Serve as blueprints for building syntax-directed translators or compilers.

An SDT scheme outlines how to implement the translation rules efficiently.

- For example, we can use a bottom-up parsing technique like LR parsing to construct a syntax tree and then traverse it to generate intermediate code. In this scheme, we can decide attribute evaluation orders, handling of conflicts, and error detection strategies

An SDT scheme for previous example could specify the **order of attribute evaluation** to ensure that child attributes are computed before parent attributes. It could also **define how to handle conflicts or errors** during parsing and attribute evaluation.

- Syntax Directed Translations
- **Syntax Directed Definitions (SDD)**
- Implementing Syntax Directed Definitions
  - Dependency Graphs
  - S-Attributed Definitions
  - L-Attributed Definitions

- Translation Schemes

Syntax Directed Definitions (SDD)

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.

- Such formalism generates **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**).

Attribute Grammar (SDD)

- Attribute grammar is a special form of context-free grammar (CFG) where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information
- Each attribute has well-defined domain of values, such as integer, float, character, string
- E=E+T does not convey any semantic information
- E=E+T {E.value= E.Value + T.value}
- Attribute grammar is a medium to provide semantics to the CFG and it can help specify the syntax and semantics of a programming language.

A **syntax-directed definition(SDD) is a generalization of a context-free grammar in which:**

- Each grammar symbol is associated with set of attributes.
- This **set of attributes** for a grammar symbol is partitioned into two subsets called
  - **synthesized** and
  - **inherited** attributes of that grammar symbol.
- Each production rule is associated with a set of semantic rules.
- SDDs are commonly used in compiler design to specify the semantics of programming languages and to guide the translation process.

E -> E + T | T

T -> T * F | F

F -> (E) | id

For the grammar we can specify semantic rules to define attributes associated with each non-terminal symbol. Using a synthesized attribute **val** for each non-terminal to represent the value of the expression; E.val, T.val, and F.val represent the value of the expression, computed using the attributes of their children

```
E -> E1 + T2      {E.val = E1.val + T2.val}
E -> T            {E.val = T.val}
T -> T1 * F2      {T.val = T1.val * F2.val}
T -> F            {T.val = F.val}
F -> ( E1 )       {F.val = E1.val}
F -> id           {F.val = lookup(id.lexeme)}
```

....contd... Syntax Directed Definitions (SDD)

- 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.
- We distinguish between **two kinds of attributes**:
- 1. **Synthesized Attributes**. They are computed from the values of the attributes of the children nodes.
  - **Note.** Terminal symbols are assumed to have **synthesized attributes** supplied by the lexical analyzer.
- 2. **Inherited Attributes**. They are computed from the values of the attributes of the siblings and the parent nodes.

Synthesized attribute & Inherited attribute

- values are computed from constants & other attributes
- **Synthesized attribute** – value computed from children
- **Inherited attribute** – value computed from siblings & parent

“Synthesized”, “Inherited" attributes

- **Synthesized Attributes** :
  - Attribute values at a node in the annotated parse tree, depend only on the attribute values at its children;
  - A synthesized attribute for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N.
    - Note that the production must have A as its head.
  - A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.

- **Inherited attributes :**
  - Attributes values at a parse-tree node are determined from attribute values at the node itself, its parent, and its siblings in the parse tree;
  - An inherited attribute for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N.
    - Note that the production must have B as a symbol in its body.
  - An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

- An S-attributed definition:
  - A syntax directed definition that uses synthesized attributes exclusively is said to be an S-attributed definition.
- Example:

Production	semantic rules
L ->E n	print(E.val)
E->E1 + T	E.val = E1.val + T.val
E->T	E.val = T.val
T->T1 * F	T.val = T1.val * F.val
T->F	T.val = F.val
F->(E)	F.val = E.val
F->digits	F.val = digits.lexval

- **Inherited attribute** – value computed from siblings & parent
- Example: inherited attributes

production	semantic rules
D ->T L	L.in = T.type
T->int	T.type = integer
T->real	T.type = real
L->L1, id	L1.in = L.in, addtype(id.entry, L.in)
L->id	addtype(id.entry, L.in)

real a1, a2, a3

SDD Example

Let us consider the Grammar for arithmetic expressions.
The Syntax Directed Definition associates to each non terminal, a synthesized attribute called val.

PRODUCTION	SEMANTIC RULE
$L \to E\mathrm{n}$	$print(E.val)$
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val * F.val$
$T \to F$	$T.val := F.val$
$F \to (E)$	$F.val := E.val$
$F \to \mathrm{digit}$	$F.val := \mathrm{digit}.lexval$

Syntax-Directed Definition -- Example

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

1. Symbols E, T, and F are associated with a synthesized attribute *val*.
2. The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).

Annotated Parse Tree

1. A parse tree showing the values of its attributes at each node is called an **annotated parse tree**.

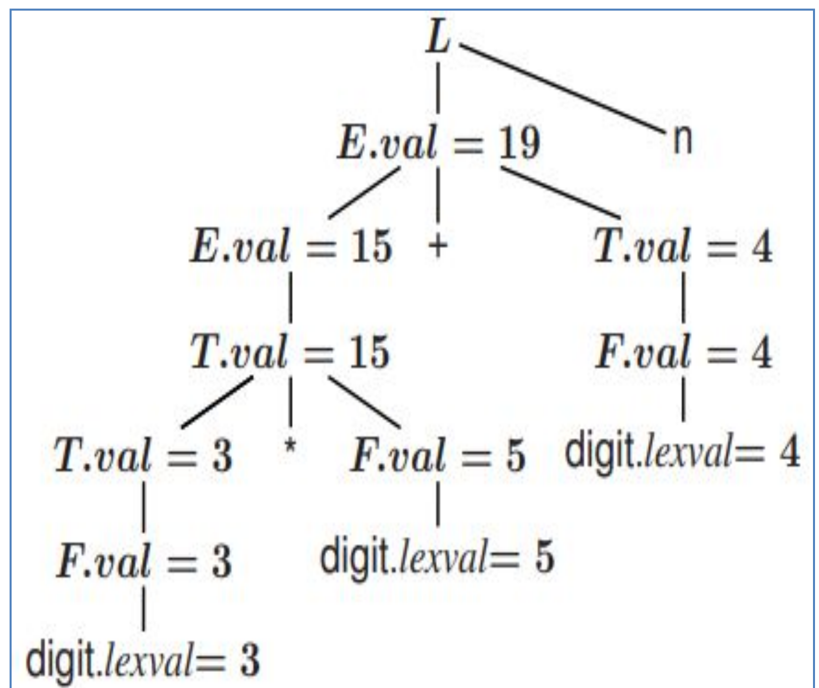
2. The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.

3. Of course, the order of these computations depends on the dependency graph induced by the semantic rules.

S-Attributed Definitions

- Definition. An S-Attributed Definition is a Syntax Directed Definition that uses only synthesized attributes.
- Evaluation Order. Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up, or PostOrder, traversal of the parse-tree.
- Example. The previous arithmetic grammar is an example of an S-Attribute d Definition. The **annotated parse-tree** for the input 3*5+4**n** is:

PRODUCTION	SEMANTIC RULE
$L \to E\mathrm{n}$	$print(E.val)$
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val * F.val$
$T \to F$	$T.val := F.val$
$F \to (E)$	$F.val := E.val$
$F \to \mathrm{digit}$	$F.val := \mathrm{digit}.lexval$

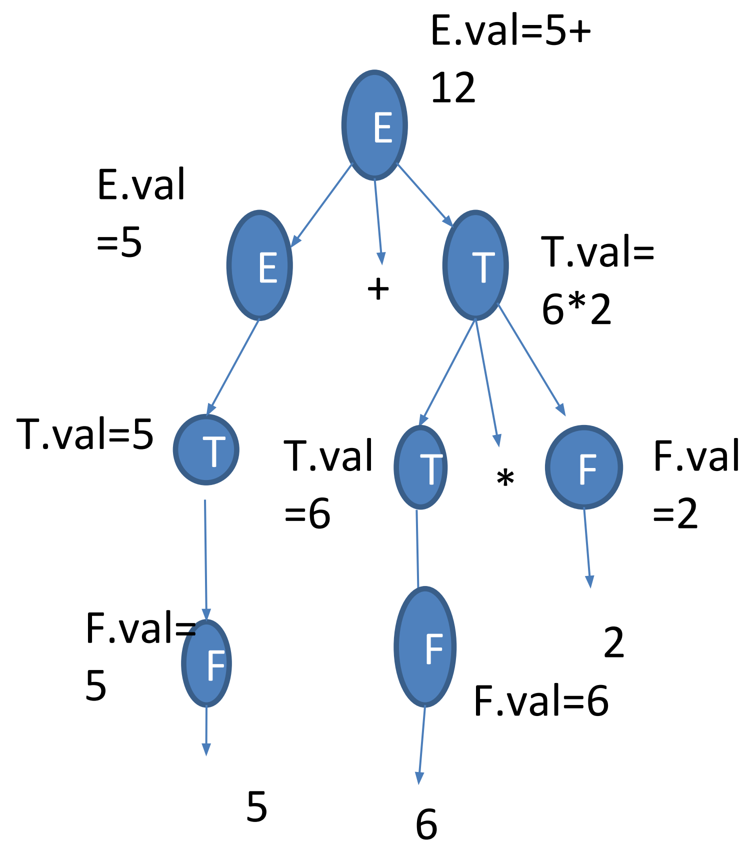


Annotated Parse Tree for S=5+6*2

E-> E+T | T
T-> T* F |
F
F-> digit

E-> E+T {E.val=E.val + T.val}
E-> T {E.val= T.val}
T-> T* F {T.val= T.val* F.val}
T-> F {T.val=F.val}
F-> digit {F.val=digit.Lexval}

- Example S=5+6*2
- Parse Tree



SDT Implementation

Convert Infix expression to postfix expression

- E-> E+T {printf("+");}
- E-> T { }
- T-> T* F {printf("*");}
- T-> F { }
- F-> digit {printf(digit.lexval);}

Input: 5+6*2

Convert Infix expression to postfix expression using SDT

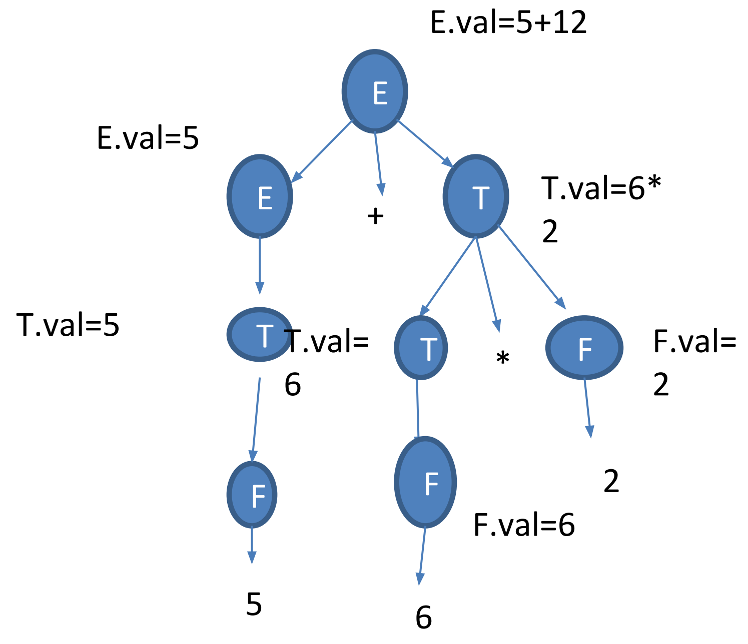
E-> E+T | T
T-> T* F | F
F-> digit

Annotated Parse Tree for 5+6*2

E-> E+T {printf("+");}
E-> T { }
T-> T* F {printf("*");}
T-> F { }
F-> digit {printf(digit.lval);}

- Example S=5+6*2
- Parse Tree

Input:S=5+6*2
Output: 562*+



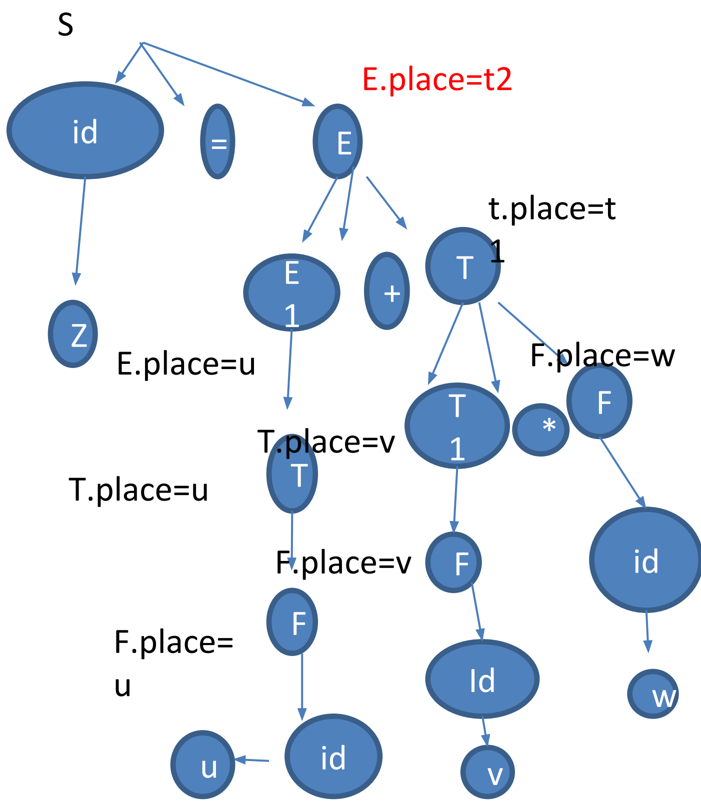
Postfix=562*+

To generate a 3 address code using SDT

S-> id = E
E-> E + T
E-> T
T-> T + F
T-> F
F-> id

Z=u+v*w

3-address code
t1=v*w
t2=u+t1
Z=t2



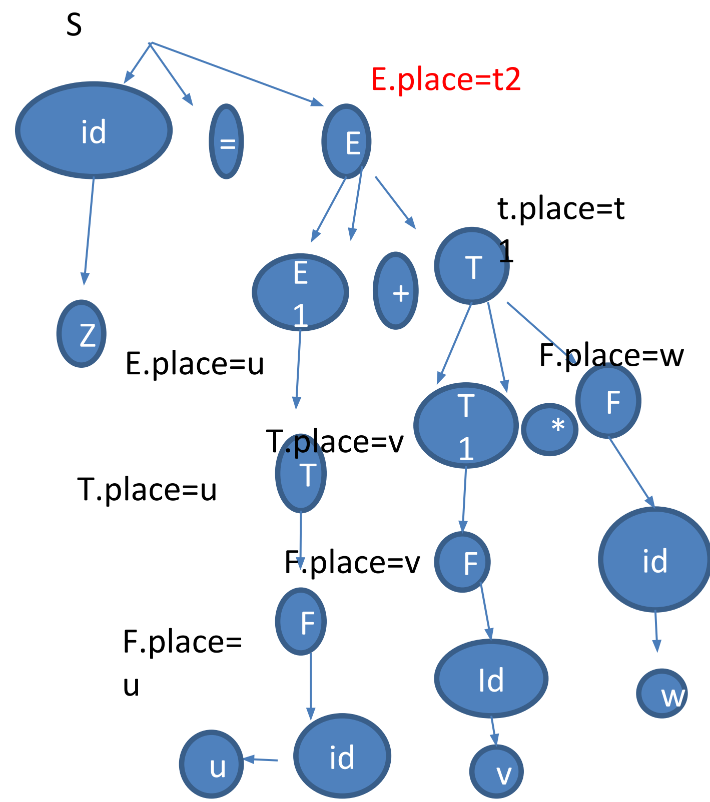
- S->id=E {gen(id.name=E.place); }
- E->E1+T {E.place=newTemp(); gen(E.place=E1.place+T.place); }
- E->T {E.place=T.place}
- T->T1* F {T.place=newTemp(); gen(T.place=T1.place*F.place); }
- T->F {T.place=F.place}
- F->id {F.place=id.name}

To generate a 3 address code using SDT

- S->id=E {gen(id.name=E.place); }
- E->E1+T {E.place=newTemp(); gen(E.place=E1.place+T.place); }
- E->T {E.place=T.place}
- T->T1* F {T.place=newTemp(); gen(T.place=T1.place*F.place); }
- T->F {T.place=F.place}
- F->id {F.place=id.name}

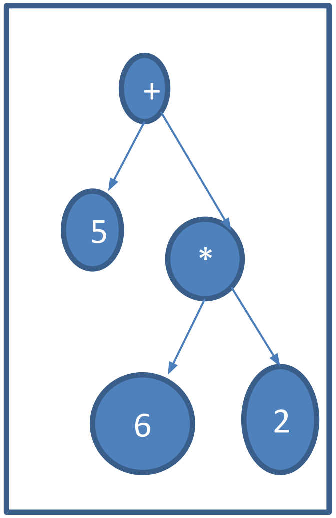
Z=u+v*w

3-address code
t1=v*w
t2=u+t1
Z=t2



SDT to build a syntax tree

- E->E1+T {E.nptr=mknode(E1.nptr ,'+' ,T.nptr);}
- E->T {E.nptr=T.nptr}
- T->T1*F {T.nptr=mknode(T1.nptr ,'*' ,F.nptr);}
- T->F {T.nptr=F.nptr}
- F->id {F.nptr=mknode(null,id.name,null);}



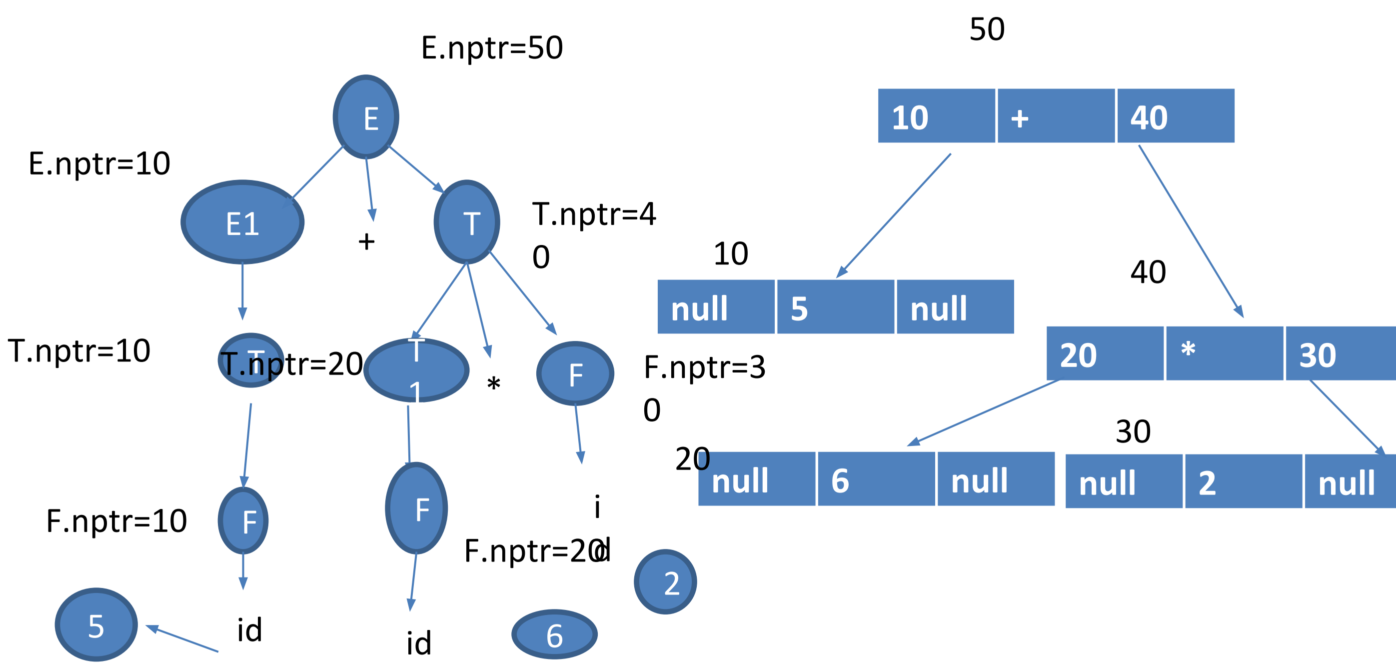
5+6*2

SDT to build a syntax tree

E->E1+T {E.nptr=mknode(E1.nptr '+' T.nptr);}
E->T {E.nptr=T.nptr}
T->T1*F {T.nptr=mknode(T1.nptr '*' F.nptr);}
T->F {T.nptr=F.nptr}
F->id {F.nptr=mknode(null,id.name,null);}
5+6*2

Annotated Parse Tree for 5+6*2

- Example 5+6*2
- Parse Tree



Type Checking using Syntax Directed Translation(SDT)

- E->E1+E2 {if(E1.type==E2.type)&&(E1.type=int))then E.type=int else error;}
- E->E1==E2{if ((E1.type==E2.type) && (E1.type=int/boolean)) then E.type=boolean else error;}
- E->(E1) {E.type=E1.type}
- E->num {E.type=int}
- E->true {E.type=Boolean}
- E->false {E.type=Boolean}

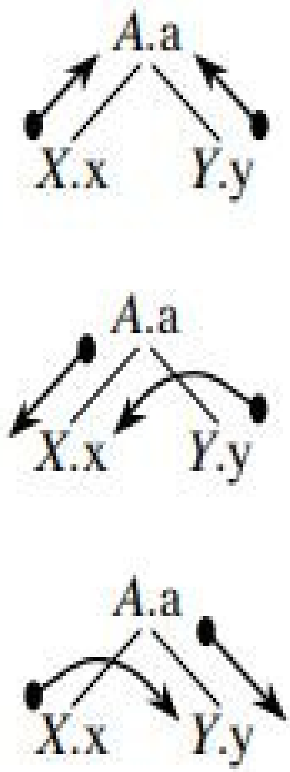
- Syntax Directed Translations
  - Syntax Directed Definitions
  - **Implementing Syntax Directed Definitions**
    - **Dependency Graphs**
    - S-Attributed Definitions
    - L-Attributed Definitions
  - Translation Schemes

Dependency Graphs

- Implementing a Syntax Directed Definition consists primarily in finding an **order for the evaluation of attributes** – Each attribute value must be available when a computation is performed.
- • Dependency Graphs are the most general technique used to evaluate syntax directed definitions with both synthesized and inherited attributes.
- • A Dependency Graph shows the interdependencies among the attributes of the various nodes of a parse-tree.
- – There is a node for each attribute;
- – If attribute b depends on an attribute c there is a link from the node for c to the node for b (b ← c).
- Dependency Rule: If an attribute b depends from an attribute c, then we need to fire the semantic rule for c first and then the semantic rule for b.

Acyclic Dependency Graphs for Parse Trees

$A \rightarrow X\ Y$



$A.\mathrm{a} := f(X.\mathrm{x}, Y.\mathrm{y})$

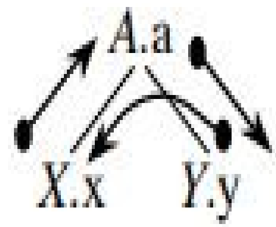
$X.\mathrm{x} := f(A.\mathrm{a}, Y.\mathrm{y})$

$Y.\mathrm{y} := f(A.\mathrm{a}, X.\mathrm{x})$

Direction of $\bullet\!\!\longrightarrow$ value dependence

Dependency Graphs with Cycles?

- Edges in the dependence graph show the evaluation order for attribute values
- Dependency graphs cannot be cyclic

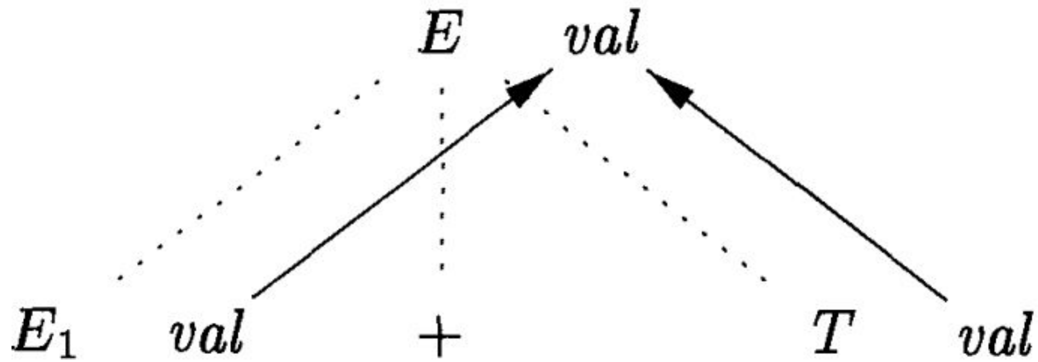


$A.a := f(X.x)$
$X.x := f(Y.y)$
$Y.y := f(A.a)$

Error: cyclic dependence

Dependency Graph

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$



- The directed graph that represents the interdependencies between synthesized and inherited at nodes in the parse tree is called dependency graph

- The graph has a node for each attribute and an edge to the node for b from the node for c if attribute b depends on attribute c.

- *Semantic rules* set up dependencies between attributes which can be represented by a *dependency graph*

- This *dependency graph* determines the evaluation order of these semantic rules.

- Dotted line represent the parse tree and is not part of dependency graph.
- This dependency graph represents the synthesized attribute.

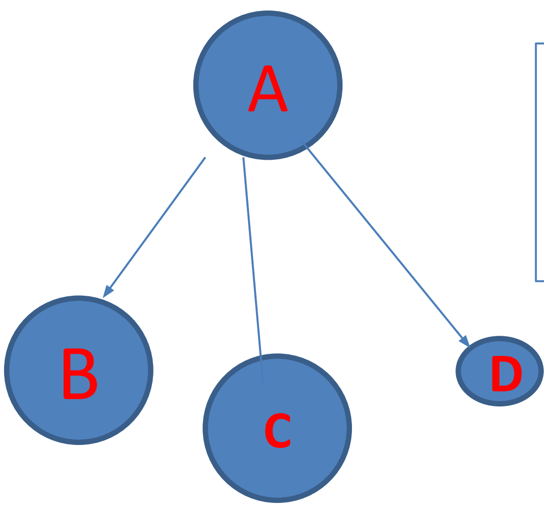
Synthesized and Inherited Attributes

Synthesized Attributes	**Inherited Attributes**
Attribute can take value only from its children	Attribute can take value either from its parent or from its siblings
A->XYZ A.S-> X.S A.S->Y.S A.S->Z.S	A-> BCD C.i->A.i Ci->B.i C.i->D.i

S-Attributed Grammar and L-Attributed Grammar

- S-Attributed Grammar - grammar containing only synthesised attributes
- L-Attributed Grammar – grammar for which attributes can always be evaluated by a depth first, L to R traversal of the parse tree

S attributed SDT	L Inherited attributed SDT
1. It uses only synthesized attribute	1. It uses both synthesized and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only
2. It is evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes	2. It is evaluated by depth-first and left-to-right parsing manner.
3. Semantic actions are placed in rightmost place of RHS. e. g. A-> b { }	3. Semantic actions are placed anywhere in RHS. e. g. A-> B { } A-> B { } C A-> { } BC



A-> BCD
C.i->A.i
Ci->B.i
C.i->D.i wrong

A->BCD

B.i->A.i

C.i->A.i
C.i->B.i

D.i->A.i
D.I ->B.i
D.i->C.i

Q) Is the given grammar S attributed or L attributed?

Example 1: A->UVW {A.S=U.S, A.S=V.S, A.S= W.S}

- Ans: S attributed

Example 2 A->UVW {U.i=A.i, V.i=A.i., V.i= U.i}

- L Inherited attributed SDT: as the node is taking a value from its parent 'A' and its left siblings

Ex 3: A->XYZ {X.i=A.i, Y.i=A.i., Y.i= U.i, Y.i= Z.i}

- Its neither S attributed nor L inherited attributed SDT as the node is taking a value from its right siblings

Example 4: Check if the given grammar is L-attributed or not?

```
A -> XYZ {Y.S = A.S, Y.S = X.S, Y.S = Z.S}
```

It is not L- attributed SDT because of Y.S=Z.S

Example: 5

```
P1: S -> MN  {S.val= M.val + N.val}
P2: M -> PQ  {M.val = P.val * Q.val  and P.val =Q.val}
```

A. Both P1 and P2 are S attributed.

B. P1 is S attributed and P2 is L-attributed.

C. P1 is L attributed but P2 is not L-attributed.

D. None of the above

Inherited Attributes: An Example

- Example. Consider the syntax directed definition with both inherited and synthesized attributes for the grammar for “type declarations”
- The non terminal T has a synthesized attribute, type, determined by the keyword in the declaration.

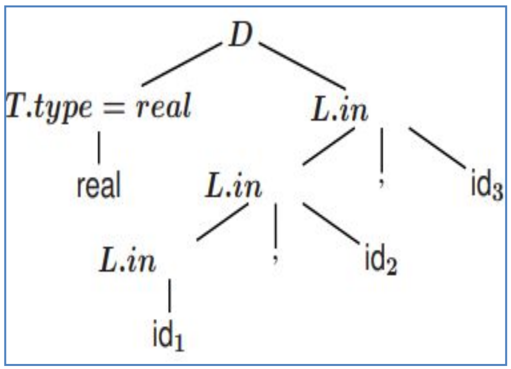
PRODUCTION	SEMANTIC RULE
$D \to TL$	$L.in := T.type$
$T \to$ int	$T.type := integer$
$T \to$ real	$T.type := real$
$L \to L_1$, id	$L_1.in := L.in$; $addtype($id$.entry, L.in)$
$L \to$ id	$addtype($id$.entry, L.in)$

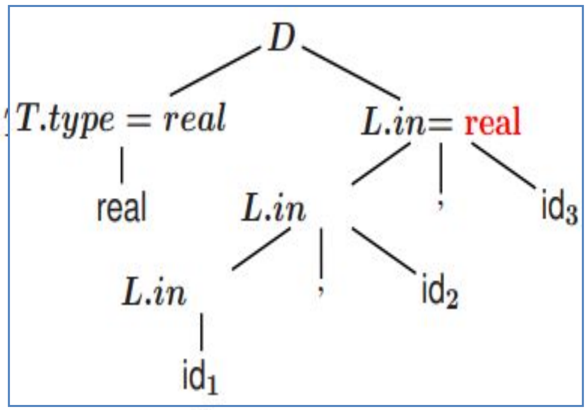
- The production **D → T L** is associated with the semantic **rule L.in := T .type** which set the inherited attribute L.in.
- Note: The **production L → L_1, id** distinguishes the two occurrences of L.

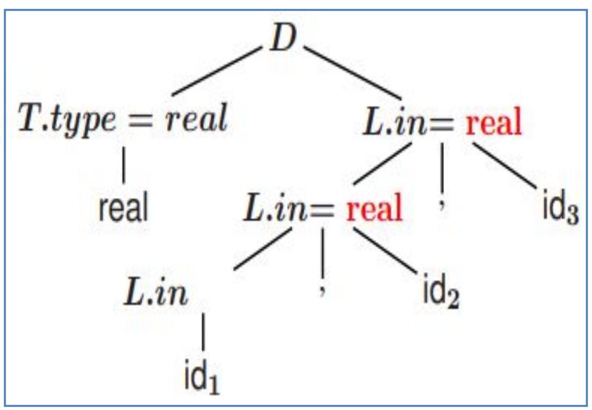
…contd..Inherited Attributes: An Example

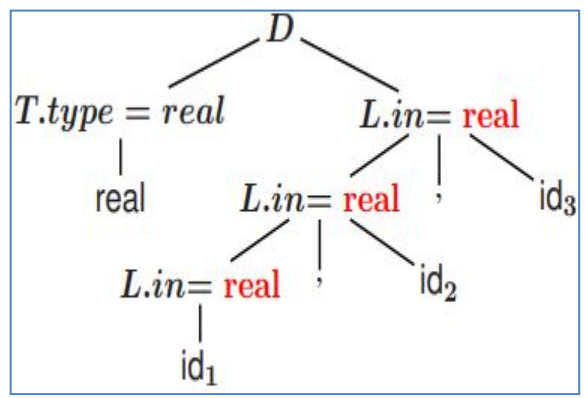
- **Synthesized attributes** can be evaluated by a **PostOrder** traversal.
- **Inherited attributes** that do not depend from right children can be evaluated by a classical **PreOrder** traversal.
- The annotated parse-tree for the input **real id1, id2, id3** is:

PRODUCTION	SEMANTIC RULE
$D \to TL$	$L.in := T.type$
$T \to$ int	$T.type :=$ *integer*
$T \to$ real	$T.type :=$ *real*
$L \to L_1$, id	$L_1.in := L.in$; *addtype*(id.*entry*, *L.in*)
$L \to$ id	*addtype*(id.*entry*, *L.in*)









L.in is then inherited top-down the tree by the other L-nodes.
At each L-node the procedure addtype inserts into the symbol table the type of the identifier.

Unit V : Semantic Analysis And Storage Allocation

- **Need, Syntax Directed Translation, Syntax Directed Definitions**, Translation of assignment Statements, iterative statements, Boolean expressions, conditional statements, Type Checking and Type conversion.
- Intermediate Code Formats: Postfix notation, Parse and syntax tress, Three address code,
- Quadruples and triples.

- Syntax Directed Translations
- Syntax Directed Definitions
- Implementing Syntax Directed Definitions
  - Dependency Graphs
  - S-Attributed Definitions
  - L-Attributed Definitions
- **Translation Schemes**

Translation Schemes

Translation Schemes

- Translation Schemes are more implementation oriented than syntax directed definitions since they indicate the order in which semantic rules and attributes are to be evaluated.
- Definition. A Translation Scheme is a context-free grammar in which 1. **Attributes** are associated with grammar symbols;

  2. **Semantic Actions** are enclosed between braces {} and are inserted within the right-hand side of productions.

Yacc uses Translation Schemes.

- ...contd... Translation Schemes

the semantic actions are enclosed between {} and are inserted within the right side of productions **to indicate the order in which translation takes place -- must be careful with the order.**

  - Example:

    E->T R

    R->+ T {print('+')} R | - T {print('-')} R | e

    T->num {print(num.val)}

...contd... Translation Schemes

- Translation Schemes deal with both synthesized and inherited attributes.
- Semantic Actions are treated as terminal symbols: Annotated parse-trees contain semantic actions as children of the node standing for the corresponding production.
- Translation Schemes are useful to evaluate L-Attributed definitions at parsing time (even if they are a general mechanism).
  – An L-Attributed Syntax-Directed Definition can be turned into a Translation Scheme.

-L-Attributed Definitions contain both synthesized and inherited attributes but do not need to build a dependency graph to evaluate them.

- S-attributed definitions can directly translated into a translation scheme by placing the semantic actions at the end of each productions.
  - Perfect for bottom up parsing (LR parsing)

- Actions in the middle of productions can be removed to be put at the end of productions by changing the grammar (adding markers).

Translation Schemes: An Example

- Consider the Translation Scheme for the L-Attributed Definition for “type declarations”:

D → T {L.in := T .type } L

T → int {T .type :=integer }

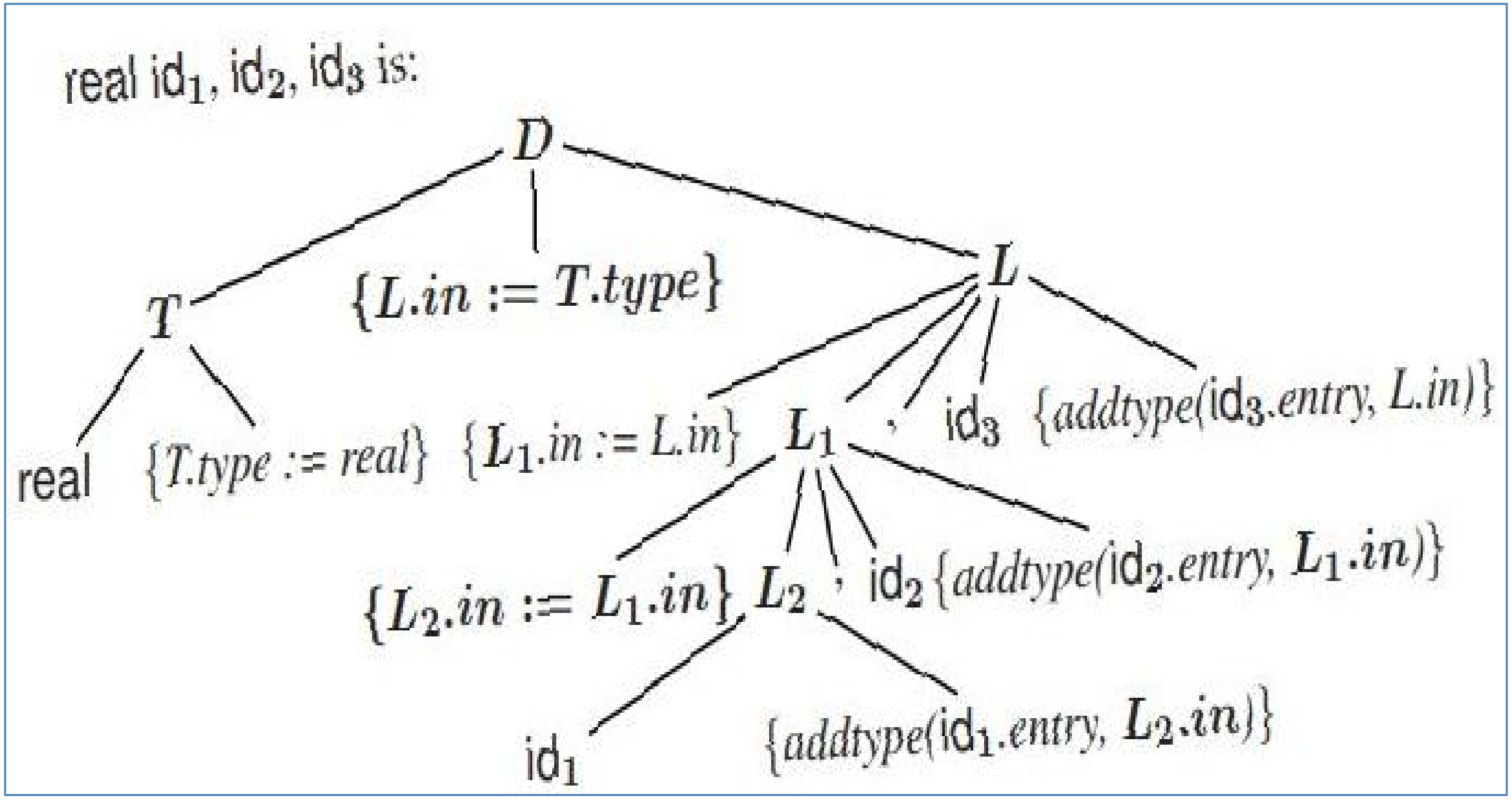
T → real {T .type :=real }

L → { L 1.in := L.in } L 1, id {addtype(id.entry, L.in) }

L → id {addtype(id.entry, L.in) }

Example :

- The parse-tree with semantic actions for the input real id 1, id 2, id 3 is as below
- Traversing the Parse-Tree in depth-first order (PostOrder) we can evaluate the attributes.



Design of Translation Schemes

- When designing a Translation Scheme we must be sure that an attribute value is available when a semantic action is executed.
- When the semantic action involves only synthesized attributes: The action can be put at the end of the production.
- – Example. The following Production and Semantic Rule:

  $T \rightarrow T1 * F \quad T.val := T1.val * F.val$

  yield the translation scheme:

  $T \rightarrow T1 * F \qquad \{T.val := T1.val * F.val\}$

Rules for Implementing L-Attributed SDD's.

If we have an L-Attibuted Syntax-Directed Definition we must enforce the following restrictions:

- 1. An inherited attribute for a symbol in the right-hand side of a production must be computed in an action before the symbol;
- 2. A synthesized attribute for the non terminal on the left-hand side can only be computed when all the attributes it references have been computed: The action is usually put at the end of the production.

Applying SDT (Infix to postfix, 3 address code generation, Build syntax tree)

Convert Infix expression to postfix expression using SDT

Annotated Parse Tree for 5+6*2

- Example S=5+6*2
- Parse Tree

Input:S=5+6*2
Output: 562*+

E-> E+T | T
T-> T* F |
F
F-> digit

E-> E+T {printf("+");}
E-> T { }
T-> T* F {printf("*");}
T-> F { }
F-> digit
{printf(digit.lval);}

Postfix=562*+

To generate a 3 address code using SDT

id=id+id*id
Z=u+v*w

SDT to generate 3 address code

S -> id=E
{gen(id.lexname=E.place); }

E-> E_1+T {E.place=newTemp();
gen(E.place=E1.place+T.place); }

E->T {E.place=T.place}

T->T_1* F {T.place=newTemp();
gen(T.place=T1.place*F.place); }

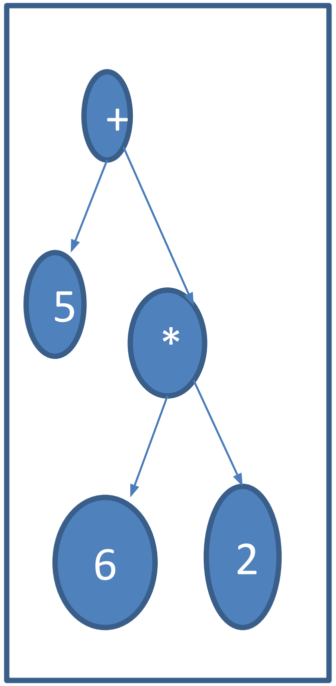
T->F {T.place=F.place}

F->id {F.place=id.lexname}

3-address code
t1=v*w
t2=u+t1
Z=t2

SDT to build a syntax tree

- E->E1+T {E.nptr=mknode(E1.nptr '+' T.nptr);}
- E->T {E.nptr=T.nptr}
- T->T1*F {T.nptr=mknode(T1.nptr '*' F.nptr);}
- T->F {T.nptr=F.nptr}
- F->id {F.nptr=mknode(null,id.name,null);}
- 5+6*2

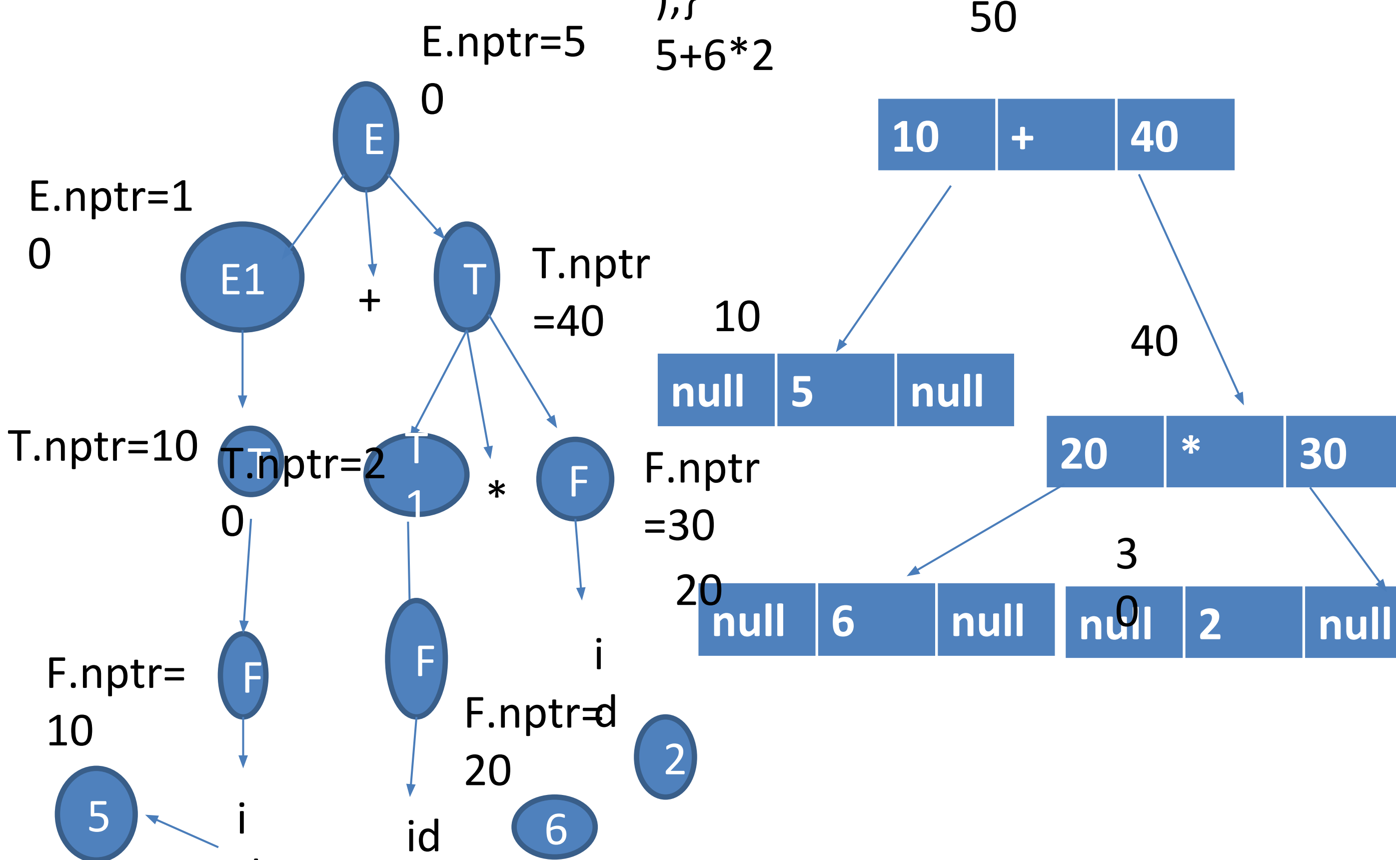


SDT to build a syntax tree

Annotated Parse Tree for 5+6*2

- Example 5+6*2
- Parse Tree

E->E1+T {E.nptr=mknode(E1.nptr '+' T.nptr);}
E->T {E.nptr=T.nptr}
T->T1*F {T.nptr=mknode(T1.nptr '*' F.nptr);}
T->F {T.nptr=F.nptr}
F->id {F.nptr=mknode(null,id.name,null);}
5+6*2



SYLLABUS

Unit V : Semantic Analysis And Storage Allocation

- Need, Syntax Directed Translation, Syntax Directed Definitions, **Translation of assignment Statements, iterative statements, Boolean expressions, conditional statements, Type Checking and Type conversion.**
- Intermediate Code Formats: Postfix notation, Parse and syntax tress, Three address code,
- Quadruples and triples.

Intermediate Code Generation:

Translation of assignment Statements, iterative statements, Boolean expressions, conditional statements, Type Checking and Type conversion.

Intro to Intermediate Code Generation

Intermediate Code Generation

- Translating source program into an “intermediate language.”
  - Simple
  - CPU Independent,
  - …yet, close in spirit to machine language.

- Or, depending on the application other intermediate languages may be used, but in general, we opt for simple, well structured intermediate forms.

- (and this completes the “Front-End” of Compilation).

Benefits

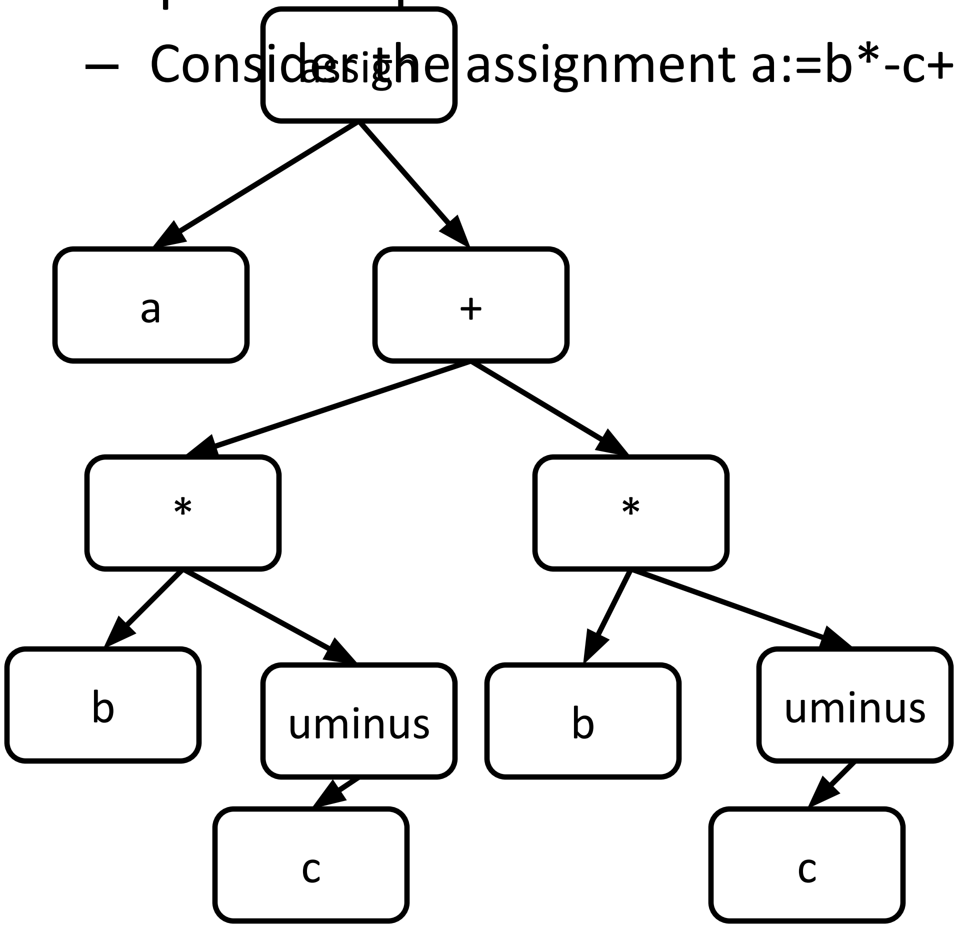
1. **Retargeting is facilitated**
2. **Machine independent Code Optimization can be applied.**

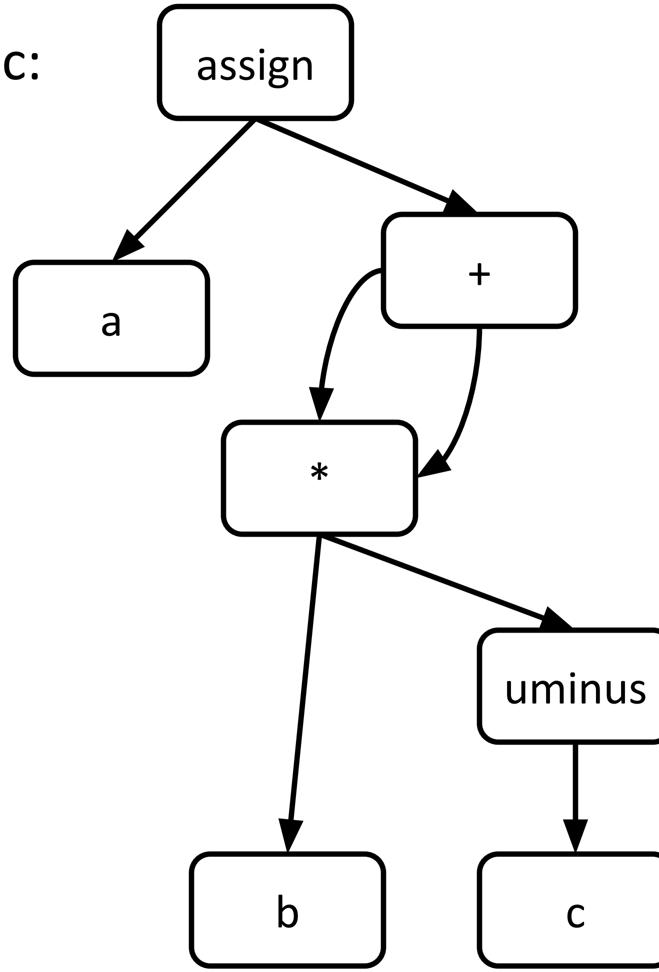
…contd..Intermediate Code Generation

- *Intermediate codes* are machine independent codes, but they are close to machine instructions.
- The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.
- Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language.
  - syntax trees can be used as an intermediate language.
  - postfix notation can be used as an intermediate language.
  - three-address code (Quadraples) can be used as an intermediate language
    - we will use quadraples to discuss intermediate code generation
    - quadraples are close to machine instructions, but they are not actual machine instructions.
  - some programming languages have well defined intermediate languages.
    - java – java virtual machine
    - prolog – warren abstract machine
    - In fact, there are byte-code emulators to execute instructions in these intermediate languages.

Types of Intermediate Languages

- Graphical Representations.
  - Consider the assignment a:=b*-c+b*-c:





Syntax Dir. Definition for Assignment Statements

PRODUCTION	Semantic Rule
$S \rightarrow id := E$	{ $S.nptr = mknode$ ('assign', $mkleaf$(id, id.$entry$), E.$nptr$) }
$E \rightarrow E_1 + E_2$	{$E.nptr = mknode$('+', $E_1.nptr$,$E_2.nptr$) }
$E \rightarrow E_1 * E_2$	{$E.nptr = mknode$('*', $E_1.nptr$,$E_2.nptr$) }
$E \rightarrow - E_1$	{$E.nptr = mknode$('uminus',$E_1.nptr$) }
$E \rightarrow (E_1)$	{$E.nptr = E_1.nptr$ }
$E \rightarrow id$	{$E.nptr = mkleaf$(id, id.$entry$) }

Three Address Code

- Statements of general form $x := y \text{ op } z$
- No built-up arithmetic expressions are allowed.
- As a result, $x := y + z * w$
  should be represented as
  $t_1 := z * w$
  $t_2 := y + t_1$
  $x := t_2$
- Observe that given the syntax-tree or the dag of the graphical representation we can easily derive a three address code for assignments as above.
- In fact three-address code is a linearization of the tree.
- Three-address code is useful: related to machine-language/ simple/ optimizable.

Example of 3-address code

Consider the assignment a:=b*-c+b*-c:

$t_1 := -\ c$
$t_2 := b\ *\ t_1$
$t_3 := -\ c$
$t_4 := b\ *\ t_3$
$t_5 := t_2\ +\ t_4$
$a := t_5$

$t_1 := -\ c$
$t_2 := b\ *\ t_1$
$t_5 := t_2\ +\ t_2$
$a := t_5$

Types of Three-Address Statements.

Assignment Statement: x:=y op z
Assignment Statement: x:=op z
Copy Statement: x:=z
Unconditional Jump: goto L
Conditional Jump: if x relop y goto L
Stack Operations: Push/pop

More Advanced:

Procedure:

param x_1
param x_2
…
param x_n
call p,n

Index Assignments:

x:=y[i]
x[i]:=y

Address and Pointer Assignments:

x:=&y
x:=*y
*x:=y

Recap : **3 address code** for **a = b * – c + b * – c**

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

Quadruple representation

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

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)

Indirect Triples representation

SYLLABUS

Unit V : Semantic Analysis And Storage Allocation

- Need, Syntax Directed Translation, Syntax Directed Definitions, **Translation of assignment Statements, iterative statements, Boolean expressions, conditional statements, Type Checking and Type conversion.**
- Intermediate Code Formats: Postfix notation, Parse and syntax tress, Three address code,
- Quadruples and triples.

Translation of assignment Statements, iterative statements, Boolean expressions, conditional statements, Type Checking and Type conversion.

Syntax-Directed Translation into 3-address code.

- First deal with assignments.
- Use attributes
  - E.*place:* the name that will hold the value of E
    - Identifier will be assumed to already have the place attribute defined.
  - E.*code:*hold the three address code statements that evaluate E (this is the `translation’ attribute).
- Use function ***newtemp*** that returns a new temporary variable that we can use.
- Use function ***gen*** to generate a single three address statement given the necessary information (variable names and operations).

Translation of Assignment Statements

You need to apply this understanding for Topics -

Iterative statements, (while, for, do-while)

Boolean expressions,

Conditional statements,(if, if-else, if-then-else, switch-case)

Type Checking and Type conversion

(1) Translation of Assignment Statement

x=a+b*c

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

Sample Grammar: Assignment Statements with Integer Types

A stands for assignment statement, **E** for expression, id for identifier(variable) token

A -> id=E

E -> E+E | E*E | -E | (E) | id

TASK :=

3 address code for Assignment Statement

a = b * – c + b * – c

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

In general,
3 address code for Assignment Statement **id= E** , where E is an expression, consists of code to evaluated E into **t,**
and then id=**t**

To evaluate E into, say **t3**, we may need to evaluate say E+E, which is to evaluate first E into **t1**, and second E into **t2,** and then adding **t1** and **t2** into **t3**

(i) Abstract Translation Scheme

Abstract Translation Scheme(for Sample Grammar)

A-> id=E
E->E+E |E*E|-E|(E)|id

a = b * – c + b * – c

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

Abstractly,

- For translation of A there is one field, **A.CODE**, which is **3-address code** to do the assignment operation
- We use **id.PLACE** to denote the **name corresponding** to the particular instance of token **id**
- **E** as structure can have **2 fields**

(I)E.PLACE, name to hold **value of expression**

(II)E.CODE, **sequence of 3-address statements** evaluating the expression

To create new temporary name, **NEWTEMP()** is used to return appropriate name

Abstract Translation Scheme

A-> id:=E
E->E+E |E*E|-E|(E)|id

Production	Semantic Action
1. $A \rightarrow id := E$	{ A.CODE := E.CODE || id.PLACE || ':=' || E.PLACE }
2. $E \rightarrow E^{(1)} + E^{(2)}$	{ T := NEWTEMP( ) ; E.PLACE := T ; E.CODE := $E^{(1)}$.CODE || $E^{(2)}$.CODE || E.PLACE || ':=' || $E^{(1)}$.PLACE || '+' || $E^{(2)}$.PLACE }
3. $E \rightarrow E^{(1)} * E^{(2)}$	{ T := NEWTEMP( ) ; E.PLACE := T ; E.CODE := $E^{(1)}$.CODE || $E^{(2)}$.CODE || E.PLACE || ':=' || $E^{(1)}$.PLACE || '*' || $E^{(2)}$.PLACE }
4. $E \rightarrow -E^{(1)}$	{ T := NEWTEMP( ) ; E.PLACE := T ; E.CODE := $E^{(1)}$.CODE || E.PLACE || ':= –' || $E^{(1)}$.PLACE }
. $E \rightarrow (E^{(1)})$	{ E.PLACE := $E^{(1)}$.PLACE ; E.CODE := $E^{(1)}$.CODE }
$E \rightarrow id$	{ E.PLACE := id.PLACE ; E.CODE := null }

Sample code (Concatenated together) for
a = b * – c + b * – c

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

Modified Scheme (with Notational Convenience)

- This uses - function GEN(A:=B+C) to emit the 3-address statement A:=B+C, with actual values substituted for A, B and C and actual operator, in the quadruple array

Production	Semantic Rules
1. $A \rightarrow id := E$	A.code := E.code || GEN(id.Place ':=' E.PLace)
2. $E \rightarrow E^{(1)} + E^{(2)}$	E.Place := NEWTEMP ; E.code := $E^{(1)}$.code || $E^{(2)}$.code || Gen(E.Place ':=' $E^{(1)}$.Place '+' $E^{(2)}$.Place)
3. $E \rightarrow E^{(1)} * E^{(2)}$	E.Place := NEWTEMP ; E.code := $E^{(1)}$.code || $E^{(2)}$.code || GEN(E.Place ':=' $E^{(1)}$.Place '*' $E^{(2)}$.Place)
4. $E \rightarrow -E^{(1)}$	E.Place := NEWTEMP ; E.code := $E^{(1)}$.code || GEN (E.Place ':=' 'Uminus' $E^{(1)}$.Place)
5. $E \rightarrow (E^{(1)})$	E.Place := $E^{(1)}$.Place ; E.code := $E^{(1)}$.code
6. $E \rightarrow id$	E.Place := id.Place ; E.code := Null

Syntax-Dir. Definition for 3-address code (Assignment statement)

PRODUCTION	Semantic Rule
S → id := E	{ S.*code* = E.*code* || *gen*(id.*place* '=' E.*place* ';') }
E → E_1 + E_2	{E.*place*= *newtemp* ; E.*code* = E_1.*code* || E_2.*code* || || *gen*(E.*place*':='E_1.*place*'+'E_2.*place*) }
E → E_1 * E_2	{E.*place*= *newtemp* ; E.*code* = E_1.*code* || E_2.*code* || || *gen*(E.*place*'='E_1.*place*'*'E_2.*place*) }
E → - E_1	{E.*place*= *newtemp* ; E.*code* = E_1.*code* || || *gen*(E.*place* '=' 'uminus' E_1.*place*) }
E → (E_1)	{E.*place*= E_1.*place* ; E.*code* = E_1.*code*}
E → id	{E.*place* = id.*entry* ; E.*code* = '' }

e.g. a := b * - (c+d)

Example 2: Translation of Assignment Statement

Sample example2

S-> id = E

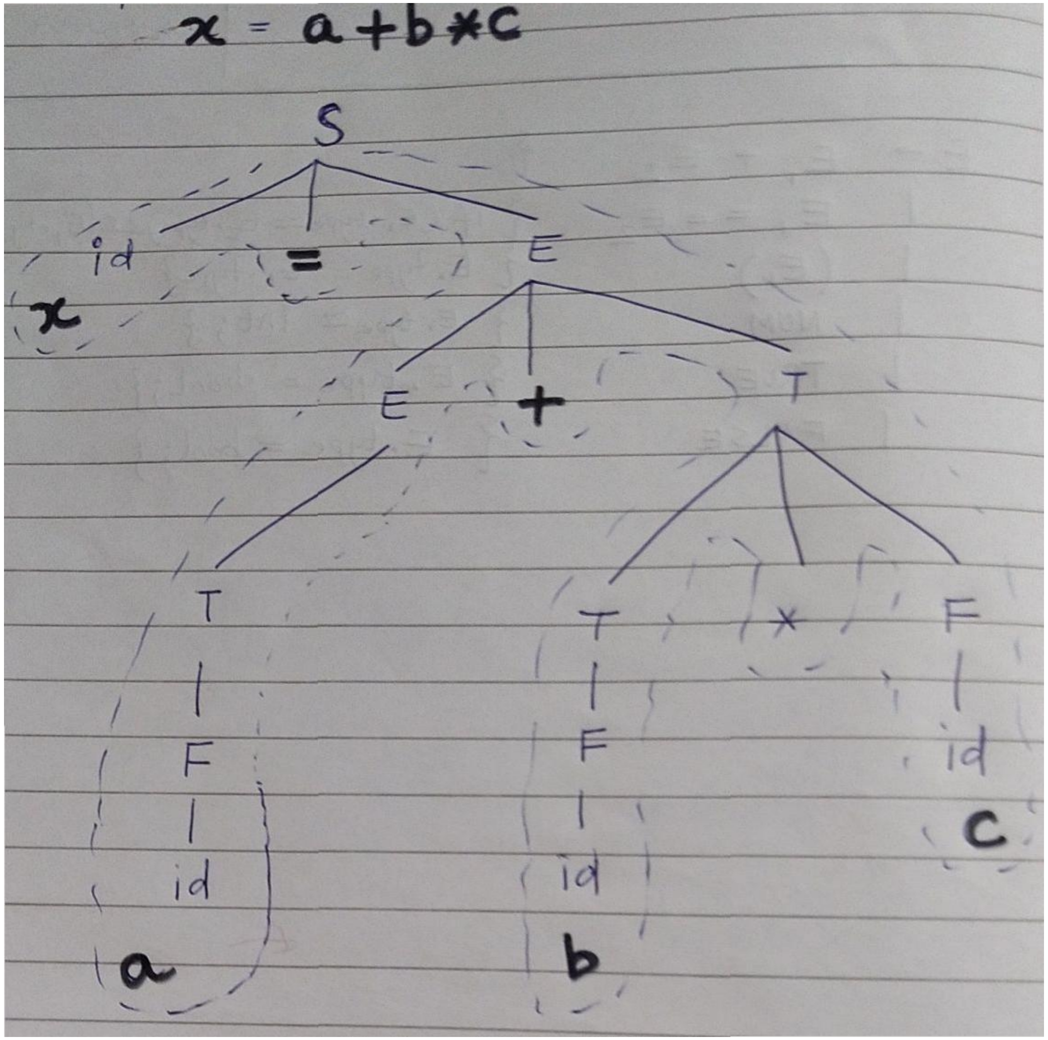
E->E+T

|T

T ->T*F

| F

F->id



..contd ...Sample example2

S -> id = E

E -> E+T

 |T

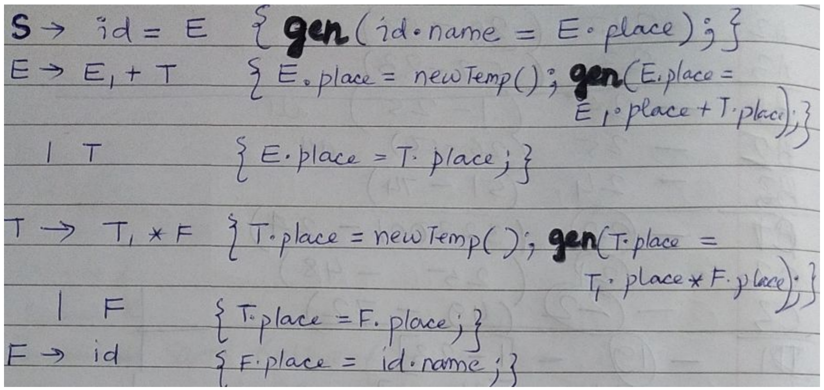
T ->T*F

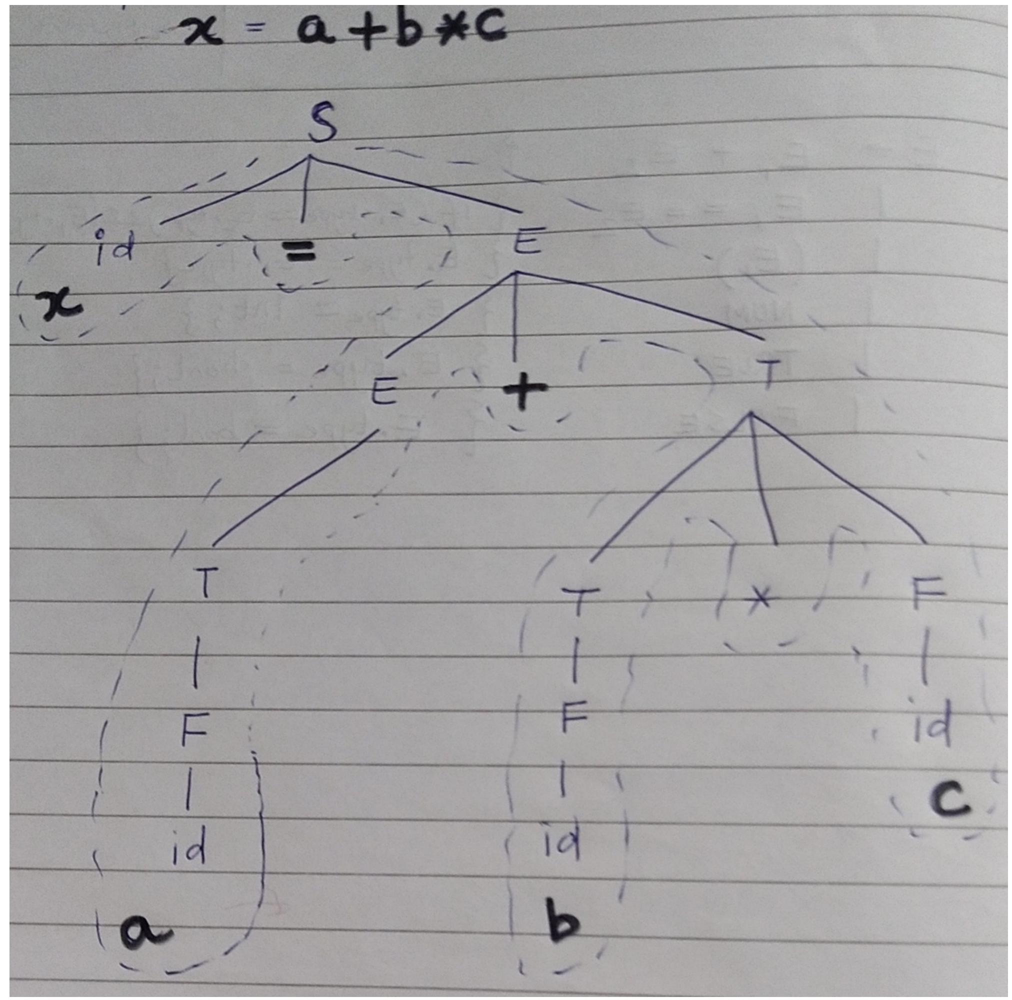
 | F

F -> id

S $\rightarrow$	id = E	{ **gen**(id.name = E.place); }
E $\rightarrow$	E_1 + T	{ E.place = newTemp(); **gen**(E.place = E_1.place + T.place); }
|	T	{ E.place = T.place; }
T $\rightarrow$	T_1 * F	{ T.place = newTemp(); **gen**(T.place = T_1.place * F.place); }
|	F	{ T.place = F.place; }
F $\rightarrow$	id	{ F.place = id.name; }

..contd ...Sample example2


S → id = E { gen(id.name = E.place); }
E → E1 + T { E.place = newTemp(); gen(E.place = E1.place + T.place); }
| T { E.place = T.place; }
T → T1 * F { T.place = newTemp(); gen(T.place = T1.place * F.place); }
| F { T.place = F.place; }
F → id { F.place = id.name; }



- x=a+b*c

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

..contd ...Sample example2

x=a+b*c

t1=b*c

t2=a+t1

x=t2

S → id = E { **gen**(id.name = E.place); }

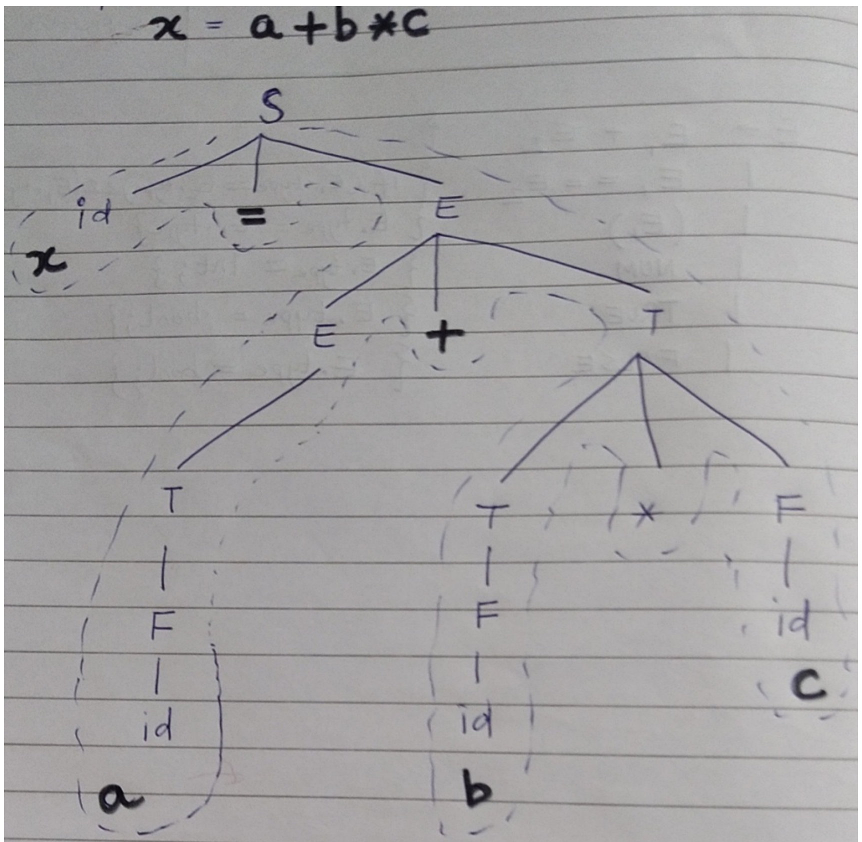
E → E_1 + T { E.place = newTemp(); **gen**(E.place = E_1.place + T.place); }

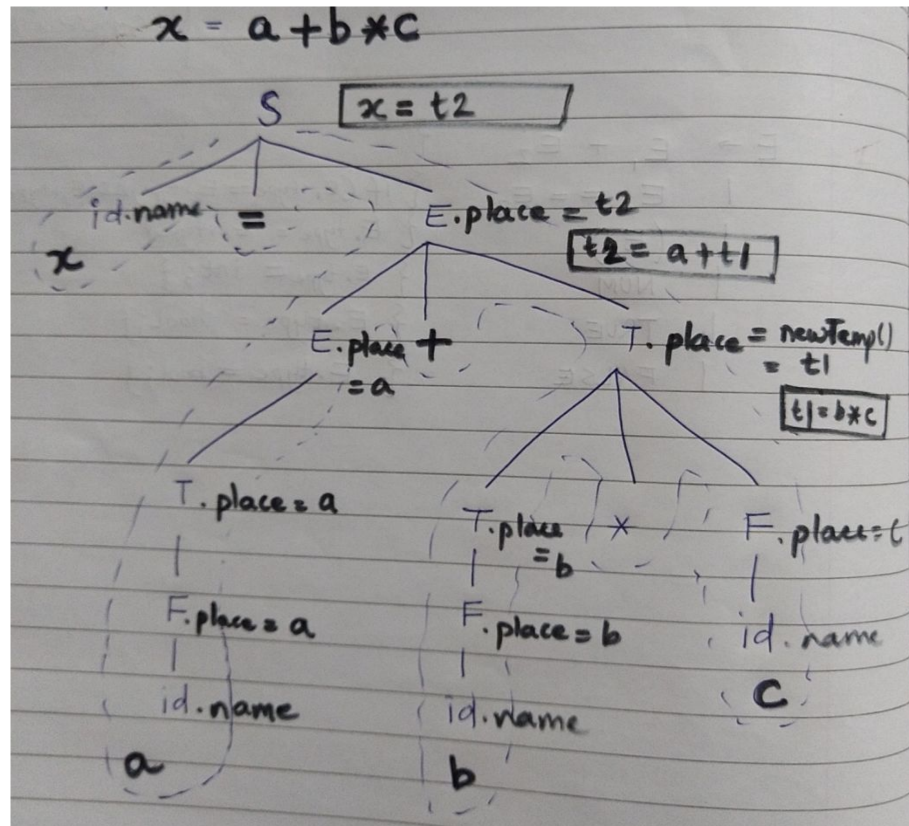
| T { E.place = T.place; }

T → T_1 * F { T.place = newTemp(); **gen**(T.place = T_1.place * F.place); }

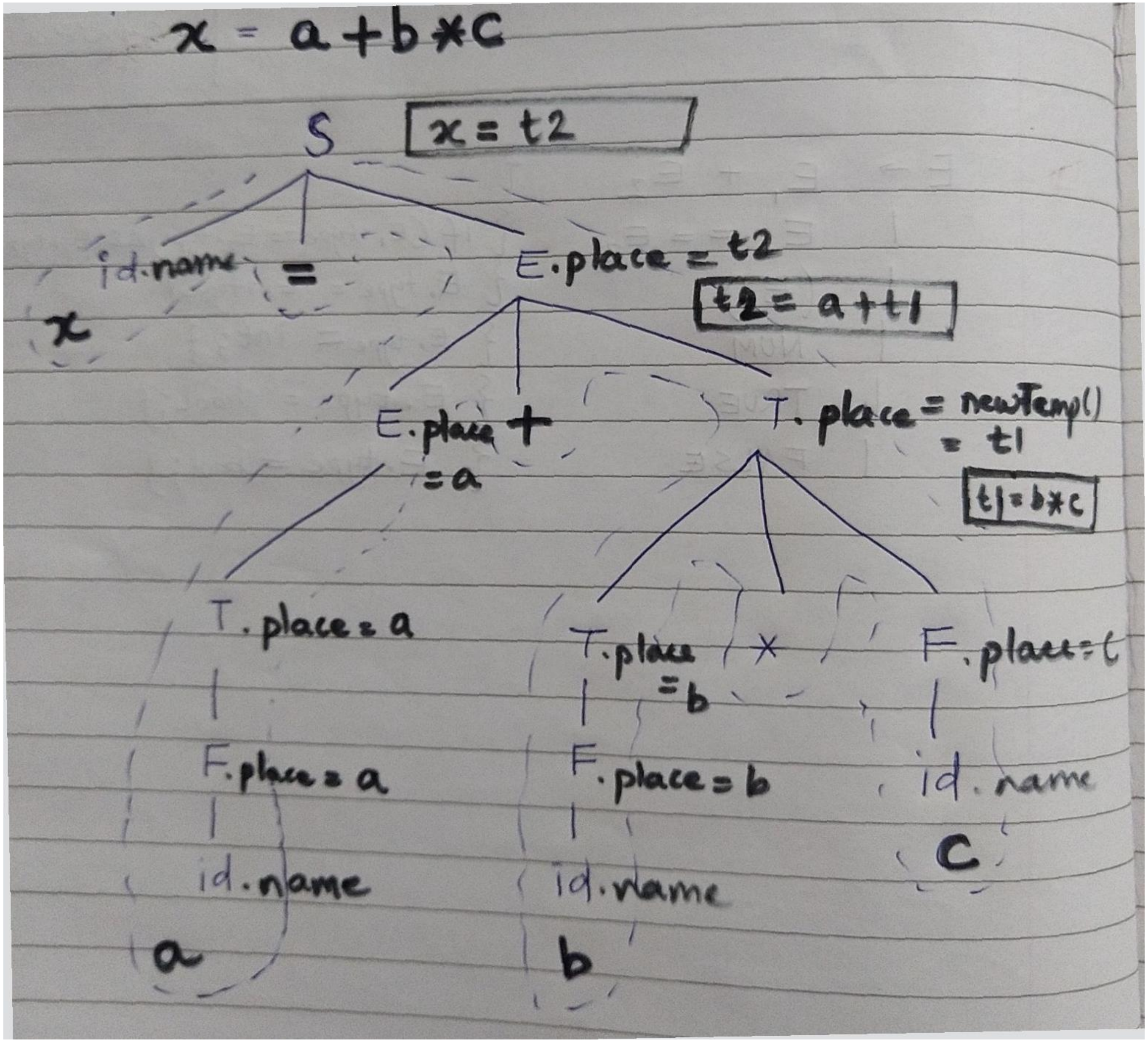
| F { T.place = F.place; }

F → id { F.place = id.name; }





..contd ...Sample example2



(2) Translation of Boolean Expressions

Boolean Expressions

- Composed of the Boolean operators (*and* , *or*, and *not*) applied to the elements that are Boolean variables or relational expressions.

- **E -> E *or* E | E *and* E | *not* E | id_1 *relop* id_2 | true | false | (E)**

Boolean Expression (3 address code output)

$(a<b<c)$

$t1= a < b$

$t2=t1<c$

Method of implementing Boolean expressions

- Encode true and false numerically; Evaluate a Boolean expression (like Arithmetic Expression)
- *Note: Here true, false are terminal values (not to be confused with true, false in semantic actions*

- **E -> E *or* E | E *and* E | *not* E | $\mathbf{id_1}$ *relop* $\mathbf{id_2}$ | true | false | (E)**

Meanings of words in next slide

- E.place stands for the variable name.
- *gen()* places three Address statements into an output file in the right format
- *nextstat* gives the index of the three address statement in the output sequence
- gen() increments *nextstat* after producing each three address statement
- *newtemp* is the function creating a new temporary variable for 3 address code

Syntax Directed Translation for **Boolean statement**

- E->E1 OR E2 {E.place= newtemp();
  gen (E.place=E1.place ‘OR’ E2.place);}
- E->E1 AND E2 {E.place= newtemp();
  gen (E.place=E1.place ‘AND’
E2.place);}
- E-> NOT E1 {E.place= newtemp();
  gen(E.place=‘NOT’ E1.place);}

- E->E1 OR ME2 {BACKPATCH(E1.FALSE,M.QUAD);
- E.TRUE=MERGE(E1.TRUE, E2.TRUE);
- E.FALSE=E2.FALSE }
- E->E1 AND ME2 {BACKPATCH(E1.TRUE,M.QUAD);
- E.TRUE=E2.TRUE;
- E.FALSE=MERGE(E1.FALSE,E2.FALSE); }
- E-> NOT E1 {E.TRUE=E1.FALSE;
- E.FALSE=E1.TRUE); }
- E-> (E1) {E.TRUE=E1.TRUE;
- E.FALSE=E1.FALSE); }
- M->€ {M.QUAD=NEXTQUAD}

Boolean Expressions

$E \rightarrow E_1$ **or** E_2

{E_1.true := E.true; E_1.false := newlabel;
 E_2.true := E.true; E_2.false := E.false;
 E.code := E_1.code || gen(E_1.false ':') || E_2.code; }

$E \rightarrow E_1$ **and** E_2

{E_1.true := newlabel; E_1.false := E.false;
 E_2.true := E.true; E_2.false := E.false;
 E.code := E_1.code || gen(E_1.true ':') || E_2.code; }

$E \rightarrow$ **not** E_1

{E_1.true := E.false; E_1.false := E.true;
 E.code := E_1.code; }

Boolean Expressions

$E \rightarrow$ "(" E_1 ")"

{ E_1.true := E.true; E_1.false := E.false;

E.code := E_1.code; }

$E \rightarrow$ **id**$_1$ **relop id**$_2$

{ E.code :=

gen('if' **id**$_1$.place **relop**.op **id**$_2$.place 'goto' E.true)

|| gen('goto' E.false); }

$E \rightarrow$ **true**

{ E.code := gen('goto' E.true); }

$E \rightarrow$ **false**

{ E.code := gen('goto' E.false); }

(3) Translation of Conditional Statements

Generate 3 address code for the following if else statement

If((a < b) and ((c > d) or (a > d)))

then

z= x + y * z

Else

z= z + 1

3 address code

100 If a< b goto 102___

101 goto_113_

102 if c>d goto_106_

103 goto_104_

104 If a>d then goto _106__

105 Goto__110_

106 t1= y*z

107t2=x+t1

108 z= t2

109 goto__113

110 t3= z+1

111 z= t3

112 goto__113

113 ____

Backpatching

Backpatching

- Need for Backpatching: During the generation of 3 address code in a single pass we may not know the address(label) where the program control should go

- Backpatching: is the process of filling up unspecified information of labels using appropriate semantic actions during the code generation process.

Backpatching

- Previous codes for Boolean expressions insert **symbolic labels** for jumps
- It therefore needs a separate pass to set them to appropriate addresses
- We can use a technique named backpatching to avoid this
- We assume we save instructions into an array and labels will be indices in the array
- For nonterminal B we use two attributes B.truelist and B.falselist together with following functions:
  - makelist(i): create a new list containing only I, an index into the array of instructions
  - Merge(p1,p2): concatenates the lists pointed 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 instruction on the list pointed to by p

Generate 3 address code for the following if else statement

If((a < b) and ((c > d) or (a > d)))

then

z= x + y * z

Else

z= z + 1

3 address code

100 if a< b then goto _102__

101 goto_110____

102 if c>d then goto__106

103 goto_104__

104 a >d then goto_106__

105 goto_110__

106 t1= x+y

107 t2= t1*z

108 z=t2

109 goto_113__

110 t3= z+1

111 z=t3

112 goto_113__

113_Next:_

translation of a simple if-statement

-
```
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:
```

“If” statement (3 address code output)

if (x < 10 || x > 20 && x != y) x = 0;

```
    T1 = x<10
    if T1 goto L2
    goto L3
L3: T2=x>20
    if T2 goto L4
    goto L1
    T3=x!=y
L4: T3  goto L2
    goto L1
L2: x = 0
L1:
```

“If” statement (3 address code output)

If A<B then x=y else y=x

T1=A<B

If T1 Goto L1

y1=x1

Goto L2

L1: x=y

L2:

“If” statement (3 address code output)

if A<B then x=y else y1=x1

T1=A<B

If T1 Goto L1

y1=x1

Goto L2

L1: x=y

L2:

“If” statement (3 address code output)

if (x < 10 || x > 20 && x != y) x = 0;

```
    T1 = x<10
    if T1 goto L2
    goto L3
L3: T2=x>20
    if T2 goto L4
    goto L1
    T3=x!=y
L4: T3  goto L2
    goto L1
L2: x = 0
L1:
```

BASIC CONCEPT INVOLVED

- The basic idea of converting any flow of control statement to a three address code is to simulate the “branching” of the flow of control using the **goto statement**.

- This is done by skipping to different parts of the code (**label**) to mimic the different flow of control branches.

Example

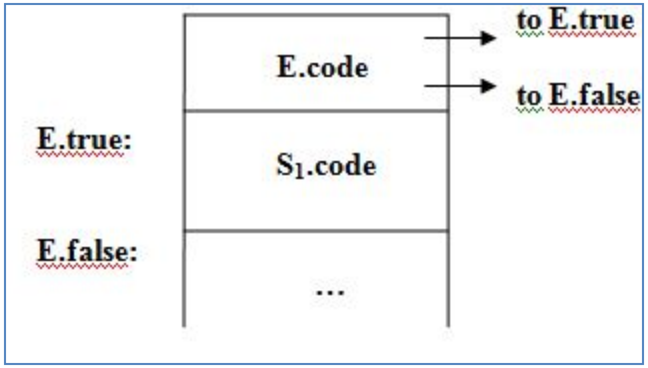
- Suppose we have the following grammar:-

$S \rightarrow \text{if } E \text{ then } S_1$

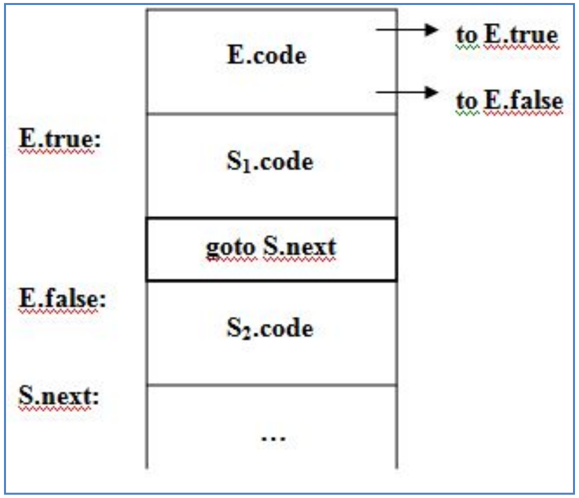
$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$

The simulation of the flow of control branching for each statement is depicted pictorially as follows:-

$S \rightarrow$ if E then S_1



$S \rightarrow$ if E then S_1 else S_2



SEMANTIC RULES:-

$S \dashrightarrow$ if E then S_1

{

E.true := newlabel ;

E.false := S.next ;

S_1.next := S.next ;

S.code := E.code || gen(E.true ':') || S_1.code

}

$S \dashrightarrow$ if E then S_1 else S_2

{

E.true := newlabel ;

E.false := newlabel ;

S_1.next := S.next ;

S_2.next := S.next ;

S.code := E.code || gen(E.true ':') || S_1.code || gen('goto' S.next) || gen(E.false ':') || S_2.code

}

(4) Translation of Iterative Statements

While-do, do-while, for, switch -case

“do - while” statement (3 address code output)

DO WHILE Loop

```
a=3;
b=4;
i=0;
do{
        a=b+1;
        a=a*a;
        i++;
}while(i<n);
c=a;
```

```
        a=3;
        b=4;
        i=0;
L1:
        VAR2=b+1;
        a=VAR2;
        VAR3=a*a;
        a=VAR3;
        i++;

        VAR1=i<n;
        if(VAR1) goto L1;
        goto L2;

L2:     c=a;
```

Translation of “for” into 3 address code

```
for(i = 0; i<5; i++)
{
y = x * 2;
}
```

Convert to ”if”

```
i=0;
L1:if (i<5) goto L2
   goto L3
L2:y=x*2;
   i=i+1;
   goto L1
L3:
```

Convert to 3 address code

```
i=0
T1=i<5
L1:   if T1 goto L2
      goto L3
L2:   T2=x*2
      y=T2
      T3=i+1
      i=T3
      goto L1
L3:
```

"for" statement (3 address code output)

FOR LOOP

```
a=3;
b=4;
for(i=0;i<n;i++){
        a=b+1;
        a=a*a;
}
c=a;
```

```
        a=3;
        b=4;
        i=0;
L1:
        VAR1=i<n;
        if(VAR1) goto L2;
        goto L3;
L4:     i++;
        goto L1;
L2:     VAR2=b+1;
        a=VAR2;
        VAR3=a*a;
        a=VAR3;
        goto L4

L3:     c=a;
```

“Switch-case” (3 Address Codes output)

```
switch (ch)
{
case 1 : c = a + b;
break;
case 2 : c = a – b;
break;
}
```

```
if ch = 1 goto L1
if ch = 2 goto L2
if ch = 3 goto L3
L1:
T1 = a + b
c = T1
goto Last
L2:
T2 = a – b
c = T2
goto Last
Last:
```

Generate 3 address code for the following switch case statement

- Switch(ch)

  {

  - Case 1: x=a + b;

    break;

  case 2: x=a - b;

    break;

  }

3 address code	3 address code
100 If ch=1 goto _102__	100 If ch=1 goto L1__
101 if ch=2 goto_105_	101 if ch=2 goto_L2
102 T1=a+b	102 L1 :T1=a+b
103 X=t1	103 X=t1
104 goto 108___	104 goto NEXT__
105 T2 = a-b	105 L2: T2 = a-b
106 x= T2	106 x= T2
107 goto_108_	107 goto_NEXT_
108 NEXT	108 NEXT:

“while” statement (3 address code output)

while (A < C and B > D) do
if A = 1 then C = C + 1
else
while A <= D
do A = A + B

Three address code for the given code is-

1. if (A < C) goto (3)
2. goto (15)
3. if (B > D) goto (5)
4. goto (15)
5. if (A = 1) goto (7)
6. goto (10)
7. T1 = c + 1
8. c = T1
9. goto (1)
10. if (A <= D) goto (12)
11. goto (1)
12. T2 = A + B
13. A = T2
14. goto (10)
15.

“while” statement (3 address code output)

while (A < C and B > D) do
if A = 1 then C = C + 1
else
while A <= D
do A = A + B
	Three address code for the given code is-
	if (A < C) goto (3)
	goto (15)
	if (B > D) goto (5)
	goto (15)
	if (A = 1) goto (7)
	goto (10)
	T1 = c + 1
	c = T1
	goto (1)
	if (A <= D) goto (12)
	goto (1)
	T2 = A + B
	A = T2
	goto (10)

Note: (A<C) etc. is kept as is

Generate 3 address code for the following **'do while statement'**

- c=0

do

{

If(a < b) then

x++;

else

x--;

c++;

} while (c<5)

100 c=0
101 if a<b goto _103__
102 goto_106__
103 t1= x+1
104 x= t1
105 goto__109__
106 t2= x -1
107 x= t2
108 goto 109__
109 t3= c+1
110 c= t3
111 if c< 5 then goto_101__
112 goto___113
113______

- while statements of the form “while E do S” (intepreted as while the value of E is not 0 do S)

<u>PRODUCTION</u>

S → while E do S_1

<u>Semantic Rule</u>

***S.begin = newlabel*;**
***S.after = newlabel* ;**
S.*code* = *gen*(S.*begin* ‘:’)
|| E.*code*
|| *gen*(‘if’ *E.*place ‘=’ ‘0’ ‘goto’ S.*after*)
|| S_1.*code*
|| gen(‘goto’ S.*begin*)
|| gen(S.*after* ‘:’)

Functions and Attributes used in the translation of iterative Statements

Flow of control statements may be converted to three address code by use of the following **functions:-**

newlabel– returns a new symbolic label each time it is called.
gen( ) – “generates” the code (string) passed as a parameter to it.

The following **<u>Attributes</u>** are associated with the non-terminals for the code generation:-

code – contains the generated three address code.
true – contains the label to which a jump takes place if the Boolean expression associated (if any) evaluates to “true”.
false – contains the label to which a jump takes place if the Boolean expression (if any) associated evaluates to “false”.
begin – contains the label / address pointing to the beginning of the code chunk for the statement “generated” (if any) by the non-terminal.
next - contains the label / address pointing to the end of the code chunk for the statement “generated” (if any) by the non-terminal

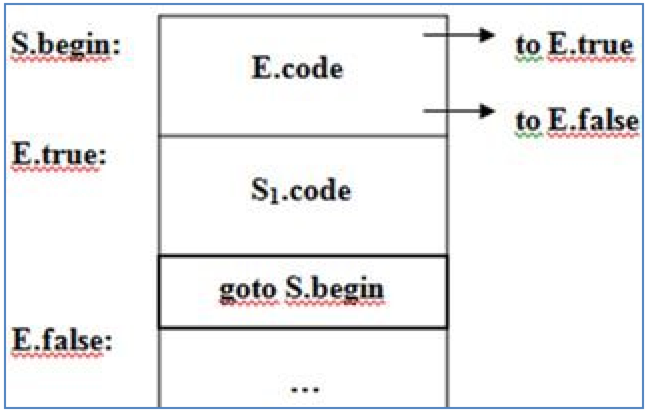
BASIC CONCEPT INVOLVED

- The basic idea of converting any flow of control statement to a three address code is to simulate the “branching” of the flow of control using the **goto statement**.
- This is done by skipping to different parts of the code (**label**) to mimic the different flow of control branches.

Example (while-do)

Flow of control branching for statement is depicted pictorially

For Grammar S → while E do S_1



SEMATIC RULES

{

S.begin := newlabel ;

E.true := newlabel ;

E.false := S.next ;

S_1.next := S.begin ;

S.code := gen(S.begin ':') || E.code || gen(E.true ':') || S_1.code || gen('goto' S.begin)

}

“do - while” statement (3 address code output)

	Three address code for the given code is:-
c = 0	1. c = 0
do	2. if (a < b) goto (4)
{	3. goto (7)
if (a < b) then	4. T1 = x + 1
x++;	5. x = T1
else	6. goto (9)
x–;	7. T2 = x – 1
c++;	8. x = T2
} while (c < 5)	9. T3 = c + 1
	10. c = T3
	11. if (c < 5) goto (2)

Note: (a<b) etc. is kept as is

Flow-of-Control Statements

$S \rightarrow$ **if** E **then** S_1
  | **if** E **then** S_1 **else** S_2
  | **while** E **do** S_1
  | **switch** E **begin**
    case V_1: S_1
    case V_2: S_2
      …
    case V_{n-1}: S_{n-1}
    default: S_n
  end

Type Conversion

$E \rightarrow E_1 + E_2$

```
{E.place := newtemp;
 if E1.type = int and E2.type = int then begin
    emit(E.place ':=' E1.place 'int+' E2.place); E.type := int;
 end else if E1.type = real and E2.type = real then begin
    emit(E.place ':=' E1.place 'real+' E2.place);
    E.type := real;
 end else if E1.type = int and E2.type = real then begin
    u := newtemp; emit(u ':=' 'inttoreal' E1.place);
    emit(E.place ':=' u 'real+' E2.place); E.type := real;
 end else if ...
}
```

Unit V : Semantic Analysis And Storage Allocation

- Need, Syntax Directed Translation, Syntax Directed Definitions, Translation of assignment Statements, iterative statements, Boolean expressions, conditional statements, Type Checking and Type conversion.
- **Intermediate Code Formats**: Postfix notation, Parse and syntax tress, **Three address code,**
- **Quadruples and triples.**

Intermediate Code Generation

1. *Intermediate codes* are machine independent codes, but they are close to machine instructions.

2. The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.

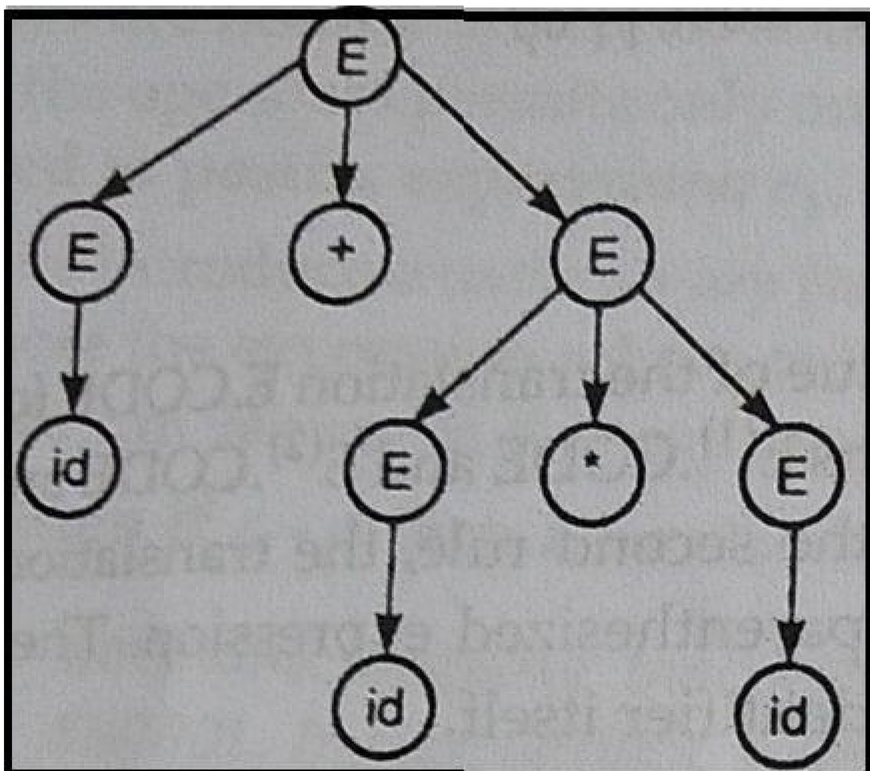
3. Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language.
   - **syntax trees** can be used as an intermediate language.
   - **postfix notation** can be used as an intermediate language.
   - **three-address code** (Quadraples) can be used as an intermediate language
   - we will use quadraples to discuss intermediate code generation
     - quadraples are close to machine instructions, but they are not actual machine instructions.

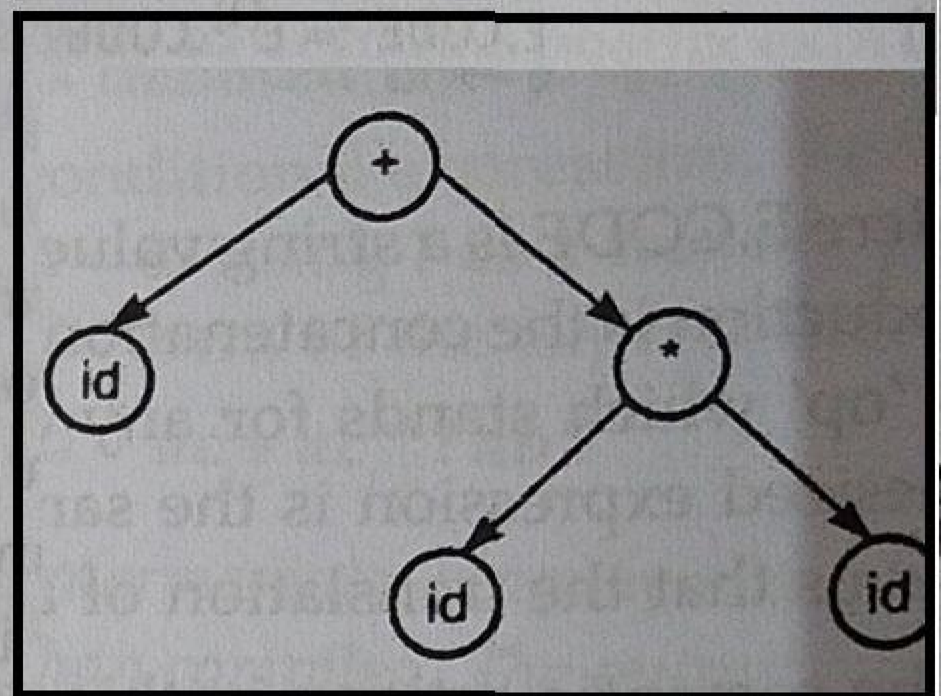
(1) Postfix

(2) Syntax Trees(AST) vs Parse Trees

Syntax Tree=AST=Abstract Syntax Tree

LHS (Parse tree for id+id*id)
VS
RHS (its corresponding Syntax tree)





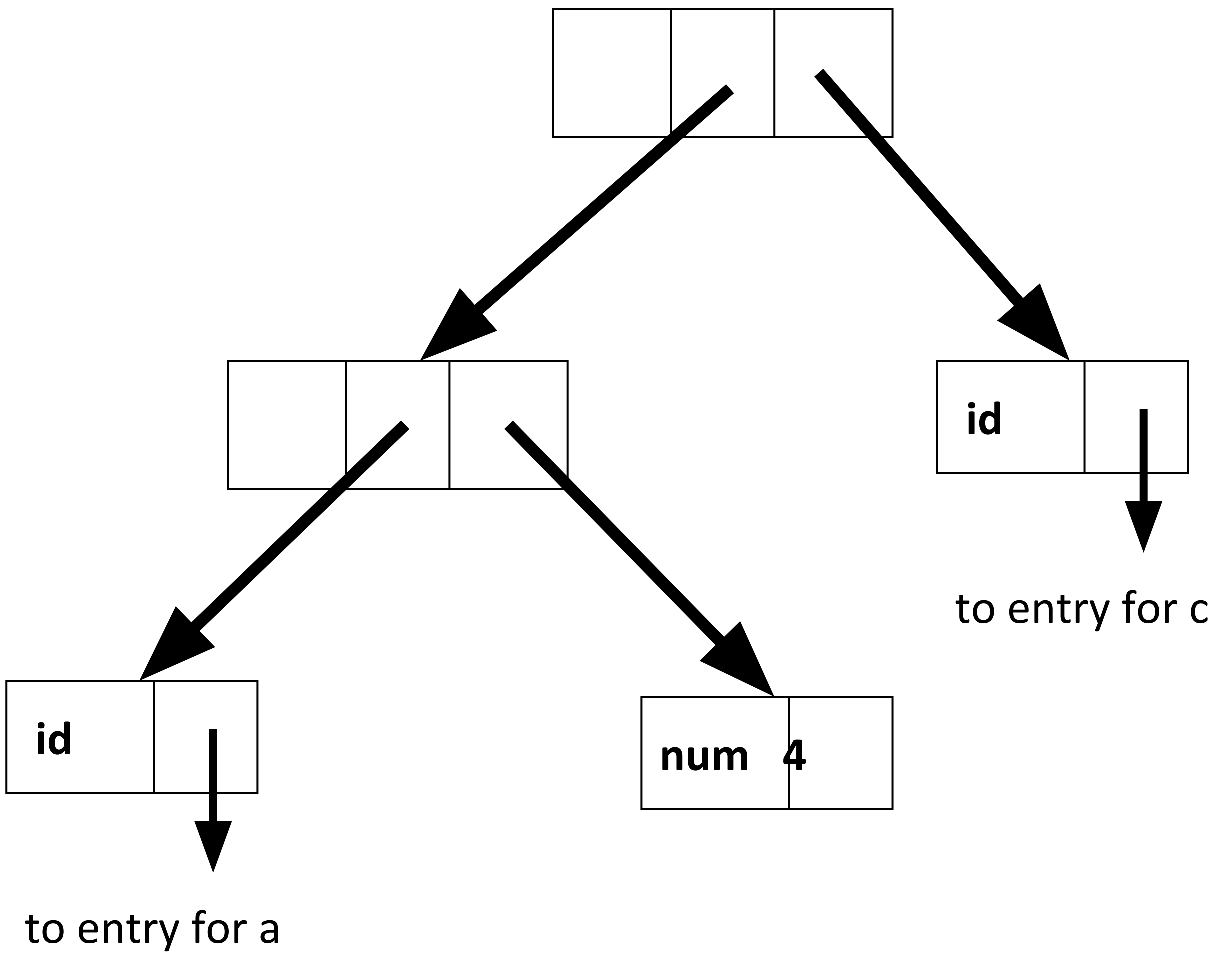
Generating Syntax Trees

1. Syntax-Tree: an intermediate representation of the compiler's input.
2. Example Procedures:
   mknode, *mkleaf*
3. Employment of the synthesized attribute *nptr* (pointer)

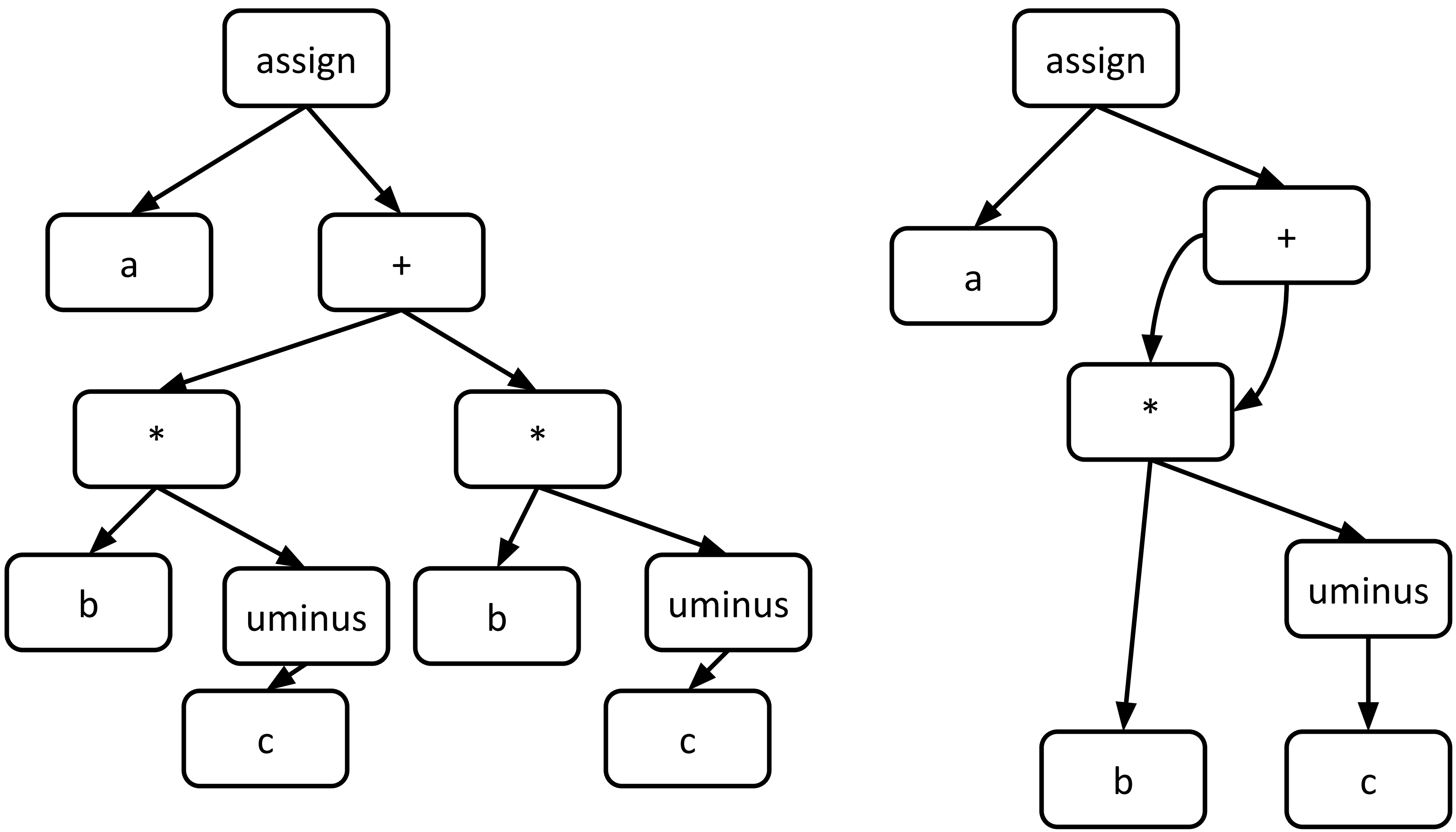
PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	**E.*nptr* = *mknode*("+",E_1.*nptr* ,T.*nptr*)**
$E \rightarrow E_1 - T$	**E.*nptr* = *mknode*("-",E_1.*nptr* ,T.*nptr*)**
$E \rightarrow T$	**E.*nptr* = T.*nptr***
$T \rightarrow (E)$	**T.*nptr* = E.*nptr***
$T \rightarrow$ id	**T.*nptr* = *mkleaf*(id, id.*lexval*)**
$T \rightarrow$ num	**T.*nptr* = *mkleaf*(num, num.*val*)**

Draw the Syntax Tree

a-4+c

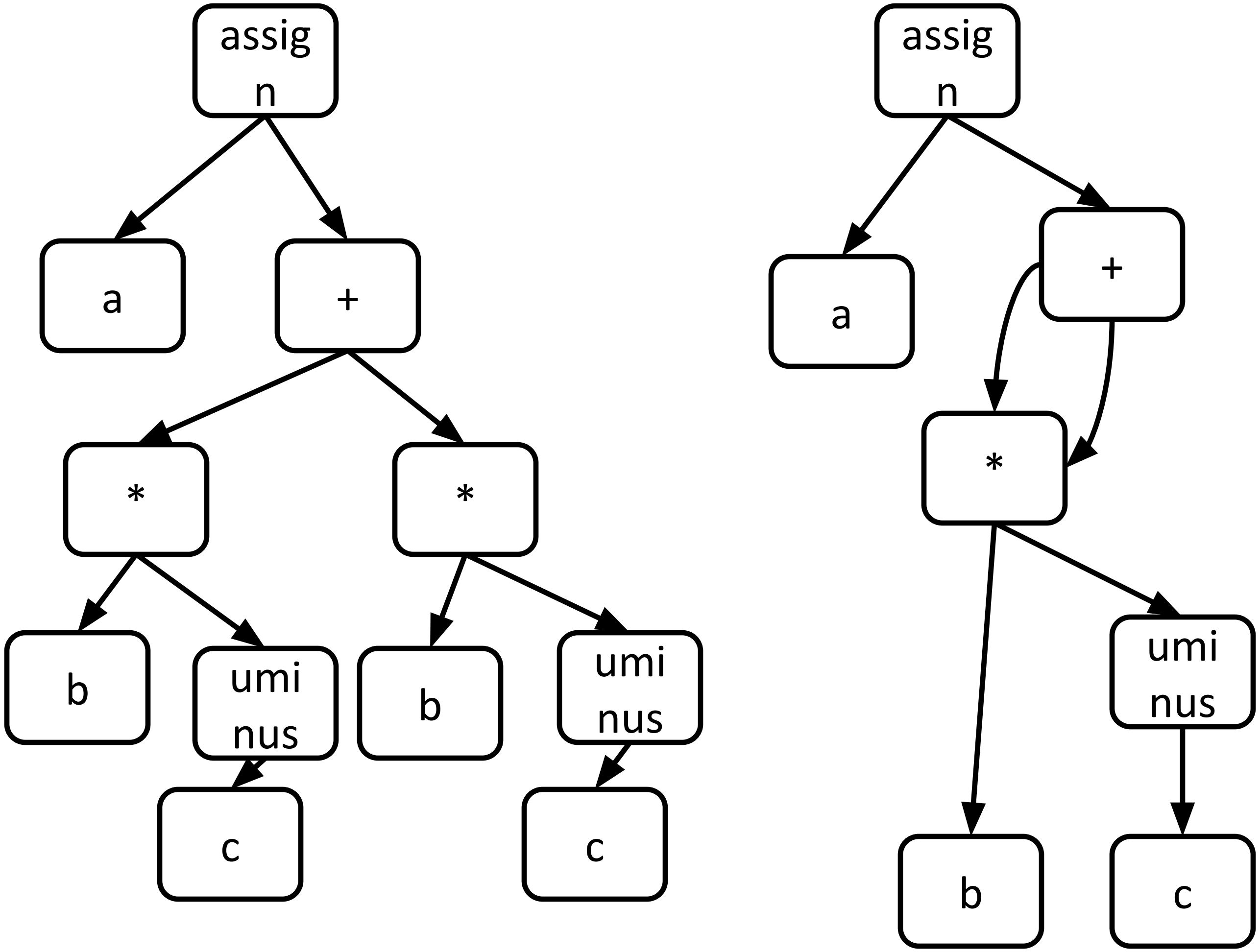


Graphical Representation
assignment a:=b*-c+b*-c:



Types of **Intermediate Languages**

- Graphical Representations.
  - Consider the assignment a:=b*-c+b*-c:



3 Address Code

Example of 3-address code

Consider the assignment a:=b*-c+b*-c:

T1=-c
T2=b*T1
T3=-c
T4=b*T3
T5=T2+T4
a =T5

Example of 3-address code

Consider the assignment a:=b*-c+b*-c:

t_1:=- c
t_2:=b * t_1
t_3:=- c
t_4:=b * t_3
t_5:=t_2 + t_4
a:=t_5

t_1:=- c
t_2:=b * t_1
t_5:=t_2 + t_2
a:=t_5

Types of Three-Address Statements.

Assignment Statement:	x:=y op z
Copy Statement:	x:=z
Unconditional Jump:	goto L
Conditional Jump:	if x relop y goto L
Stack Operations:	Push/pop

More Advanced:

Index Assignments:

x:=y[i]
x[i]:=y

Address and Pointer Assignments:

x:=&y
x:=*y
*x:=y

- Write the 3 address code for the following expressions

1. a=b + c + d
2. –(a*b)+(c+d)-(a+b+c+d)
3. If A<B then 1 else 0
4. If A<B and C<D then t=1 else t=0

- **If A<B and C<D then t=1 else t=0**

1 If A<B then go to __3_

2 Goto __4__

3 If C< D then go to __6_

4 T=0

5 go to _7__

6 T=1

7 ..

Ex: Generate three address code for following code fragment. (8 Mks)

```
sum=0
for (j=1; j<=10; j++)
    sum= sum + a[j] + b[j]
```

Ex: Generate quadruples and indirect triples for following statement. (8 Mks)

a= b ^ (c + d) *f/g

Generate 3 address code for the following switch case statement

- Switch(ch)

  {

  – Case 1: x=a + b;

  break;

  case 2: x=a - b;

  break;

  }

3 address code
100 If ch=1 go
to_102__
101 if ch =2
goto_105_
102 t1=a + b
103 x = t1
104 goto__108_
105 t2 = a-b
106 x = t2
107 goto_ 108__
108NEXT:

3 address code
100 If ch=1 go
to_L1__
101 if ch =2
goto_L2_
102 L1: t1=a + b
103 x = t1
104 goto_NEXT_
105 L2: t2 = a-b
106 x = t2
107 goto_ NEXT_
108 NEXT:

Generate 3 address code for the following switch case statement

- Switch(ch)

  {

  – Case 1: x=a + b;

    break;

    case 2: x=a - b;

    break;

  }

3 address code

100 If ch=1 goto
_102__
101 if ch=2
goto_105_
102 T1=a+b
103 X=t1
104 goto 108___
105 T2 = a-b
106 x= T2
107 goto_108_
108 NEXT

3 address code

100 If ch=1 goto
L1__
101 if ch=2 goto_L2
102 L1 :T1=a+b
103 X=t1
104 goto NEXT__
105 L2: T2 = a-b
106 x= T2
107 goto_NEXT_
108 NEXT:

Generate 3 address code for the following do while statement

- c=0

```
do
{
    If(a < b) then
        x++;
    else
        x--;
    c++;
} while (c<5)
```

```
3 address code
100 C=0
101 if a< b then
goto_103__
102 goto_106_
103     t1=x+1
104     x = t1
105    goto__109_
106     t2 =x-1
107     x= t2
108    Goto_109_
109     t3= c+1
110     c=t3
111     if c< 5 goto_101__
112     goto__113
113     NEXT:
```

```
3 address code
100 C=0
101  LX: if a< b then
goto_L1__
102 goto_L2_
103     L1: t1=x+1
104         x = t1
105         goto__L3_
106      L2:   t2 =x-1
107          x= t2
108         Goto_109_
109     L3: t3= c+1
110         c=t3
111      if c< 5 goto_LX__
112     goto__NEXT
113     NEXT:
```

Generate 3 address code for the following do while statement

```
• c=0
do
{
    If(a < b) then
        x++;
    else
        x--;
    c++;
} while (c<5)
```

```
100 c=0
101 if a<b goto _103__
102 goto_106__
103 t1= x+1
104 x= t1
105 goto__109__
106 t2= x -1
107 x= t2
108 goto 109__
109 t3= c+1
110 c= t3
111 if c< 5 then
goto_101__
112 goto___113
113______
```

Generate 3 address code for the following if else statement

If((a < b) and ((c > d) or (a > d)))

then

z= x + y * z

Else

z= z + 1

Backpatching

3 address code

100 If a< b goto 102___

101 goto_113_

102 if c>d goto_106_

103 goto_104_

104 If a>d then goto _106__

105 Goto__110_

106 t1= y*z

107t2=x+t1

108 z= t2

109 goto__113

110 t3= z+1

111 z= t3

112 goto__113

113 ____

Generate 3 address code for the following if else statement

If((a < b) and ((c > d) or (a > d)))

then

z= x + y * z

Else

z= z + 1

3 address code

100 if a< b then goto _102__

101 goto_110____

102 if c>d then goto__106

103 goto_104__

104 a >d then goto_106__

105 goto_110__

106 t1= x+y

107 t2= t1*z

108 z=t2

109 goto_113__

110 t3= z+1

111 z=t3

112 goto_113__

113_Next:_

Generate the 3 address code for the following program segment

- While (A < C and B > D)

  do if A ==1 then c= c+1

  else

  while A<=D

  do A=A+B

3 address code

100 if A< C goto_102__
101 goto_114__
102 if B>D goto 104___
103 goto_114__ __
104 if A==1 then goto_106__
105 goto_109_
106 t1 = c+1
107 c=t1
108 goto_100_
109if A<=D goto_111_
110 goto_100_
111 t2= A+ B
112 A=t2
113 goto_109_
114 NEXT:____

Representation of 3 address code

Representation of 3 address code using **Quadruple**

- T1=-c
- T2=b*T1
- T3=-c
- T4=b*T3
- T5=T2+T4
- a =T5

Location	Operator	Op1	Op2	Result	
(0)	uminus	c		T1	
(1)	*	b	T1	T2	
(2)	uminus	c		T3	
(3)	*	b	T3	T4	
(4)	+	T2	T4	T5	
(5)	=		T5	a	

Representation of 3 address code using **triples**

3 address code

- T1=-c
- T2=b*T1
- T3=-c
- T4=b*T3
- T5=T2+T4
- a =T5

location	Operator	Op1	Op2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

Implementations of 3-address statements

- Quadruples

$t_1 := -\ c$
$t_2 := b * t_1$
$t_3 := -\ c$
$t_4 := b * t_3$
$t_5 := t_2 + t_4$
$a := t_5$

	op	*arg1*	*arg2*	*result*
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	assign	t5		a

Temporary names must be entered into the symbol table as they are created.

Implementations of 3-address statements

- Quadruples

t_1:=- c
t_2:=b * t_1
t_3:=- c
t_4:=b * t_3
t_5:=t_2 + t_4
a:=t_5

	op	*arg1*	*arg2*	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

Temporary names must be entered into the symbol table as they are created.

Implementations of 3-address statements(**Triples**)

- Triples

$t_1 := - c$

$t_2 := b * t_1$

$t_3 := - c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$a := t_5$

	op	*arg1*	*arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	(4)	

Temporary names are <u>not entered</u> into the symbol table.

Implementations of 3-address statements (**Triples**)

- Triples

$t_1 := - c$

$t_2 := b * t_1$

$t_3 := - c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$a := t_5$

	op	*arg1*	*arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

Temporary names are not entered into the symbol table.

Other types of 3-address statements

- e.g. ternary operations like
  x[i]:=y   x:=y[i]
- require two or more entries. e.g.

	op	*arg1*	*arg2*
(0)	[]=	X	i
(1)	assign	(0)	y

	op	*arg1*	*arg2*
(0)	[]=	y	i
(1)	assign	X	(0)

Other types of 3-address statements

- e.g. ternary operations like
  x[i]:=y   x:=y[i]
- require two or more entries. e.g.

	op	*arg1*	*arg2*
(0)	[ ] =	x	i
(1)	assign	(0)	y

	op	*arg1*	*arg2*
(0)	[ ] =	y	i
(1)	assign	x	(0)

Implementations of 3-address statements, III

- Indirect Triples

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

	op	*arg1*	*arg2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Ex. : Generate intermediate codes for the given assignment stmt (8 Mks)

cost = rate *(start-finish) + 2 * (start-finish-100)

Solution :- 3 address code

- Three address code
  - Quadruples
  - Triples
  - Indirect triplets

Generate intermediate codes for the given assignment stmt (8 Mks)

cost = rate *(start-finish) + 2 * (start-finish-100)

- Three address code
  - Quadruples
  - Triples
  - Indirect triplets

Quadrapl
e

3- Address
code
T1=Start-
finish
T2=rate* T1
T3=T1-100
T4=2*T3
T5=T2+T4
Cost=T5

Location	Operator	Op1	Op2	Result
(0)	-	start	finish	T1
(1)	*	rate	T1	T2
(2)	-	T1	100	T3
(3)	*	2	T3	T4
(4)	+	T2	T4	T5
(5)	=	T5		Cost

Triple

3- Address code
T1=Start-finish
T2=rate* T1
T3=T1-100
T4=2*T3
T5=T2+T4
Cost=T5

Location	Operator	Op1	Op2
(0)	-	Start	Finish
(1)	*	rate	(0)
(2)	-	(0)	100
(3)	*	2	(2)
(4)	+	(1)	(3)
(5)	=	Cost	(4)

Indirect Triple

3- Address code
T1=Start-finish
T2=rate* T1
T3=T1-100
T4=2*T3
T5=T2+T4
Cost=T5

Location	Operator	Op1	Op2
(11)	-	Start	Finish
(12)	*	rate	(0)
(13)	-	(0)	100
(14)	*	2	(2)
(15)	+	(1)	(3)
(16)	=	Cost	(4)

(0)	(11)
(1)	(12)
(2)	(13)
(3)	(14)
(4)	(15)
(5)	(16)