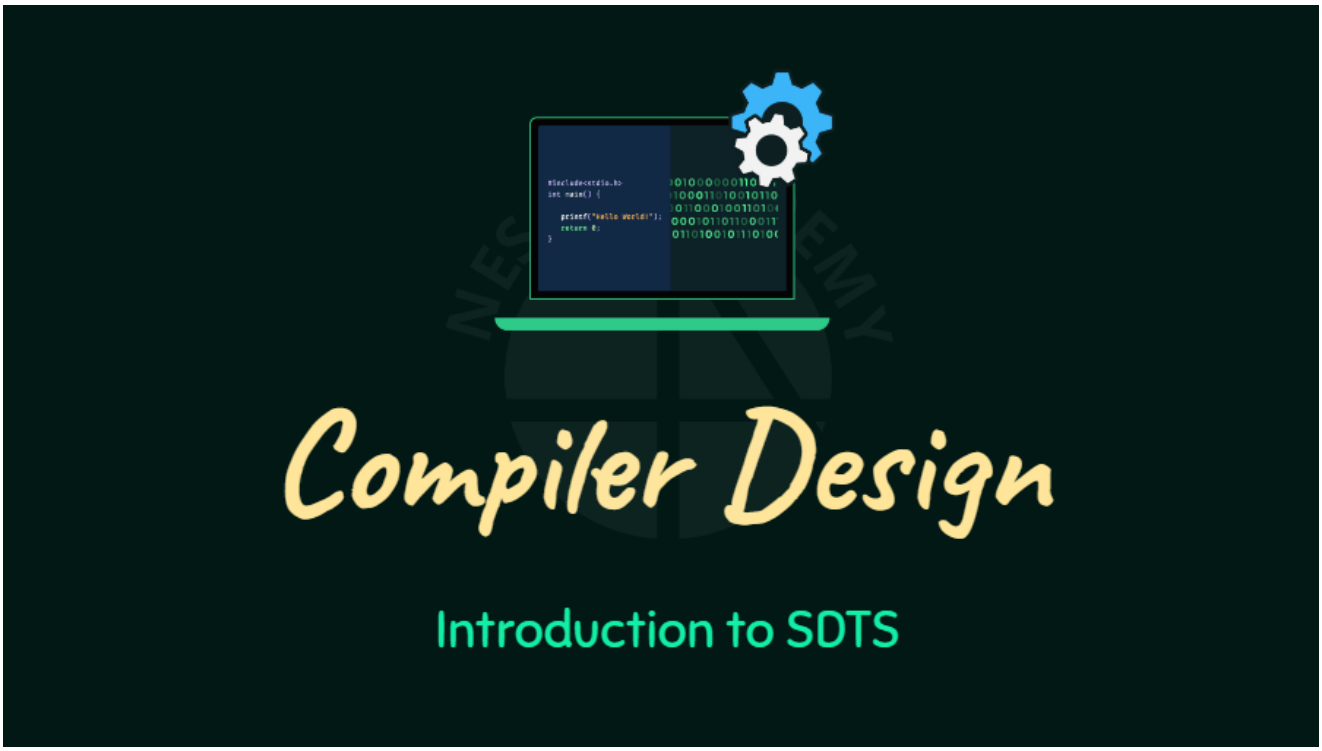
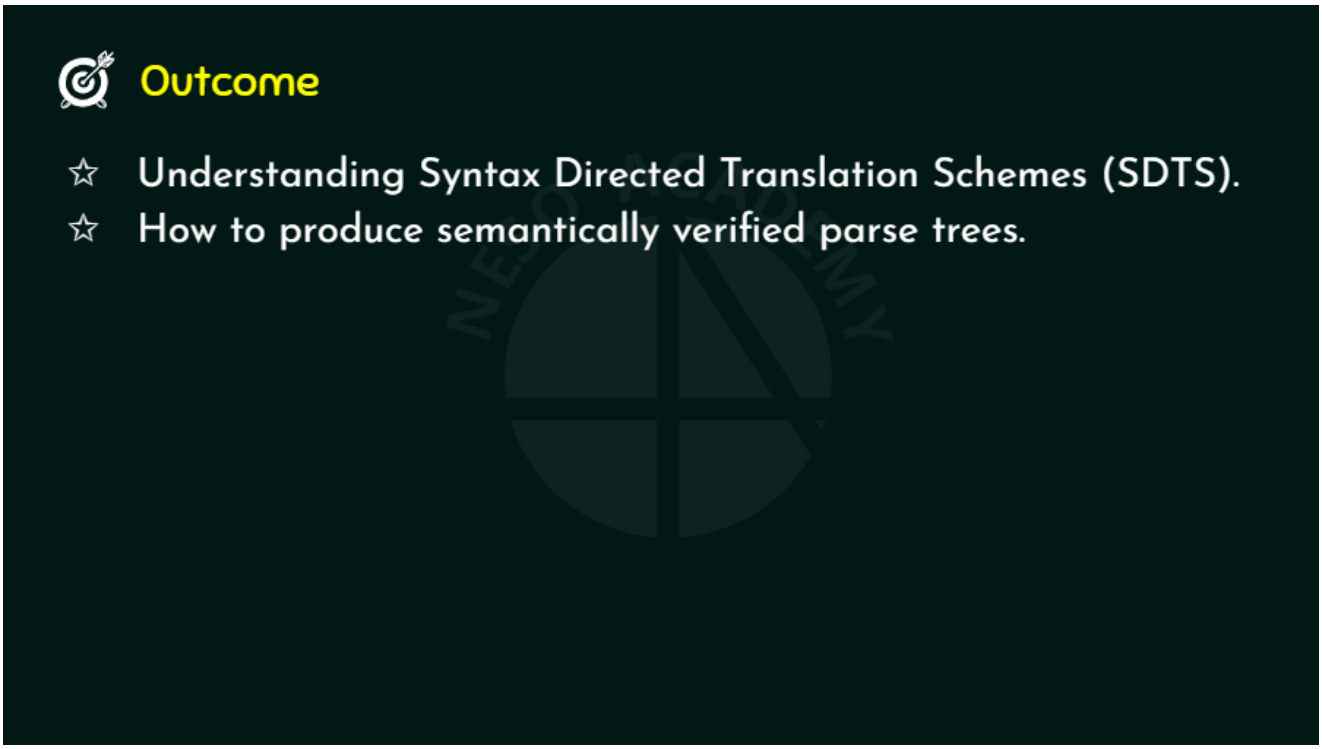
Semantic Analysis | Neso Academy

nesoacademy.org/cs/12-compiler-design/ppts/05-semanticanalysis

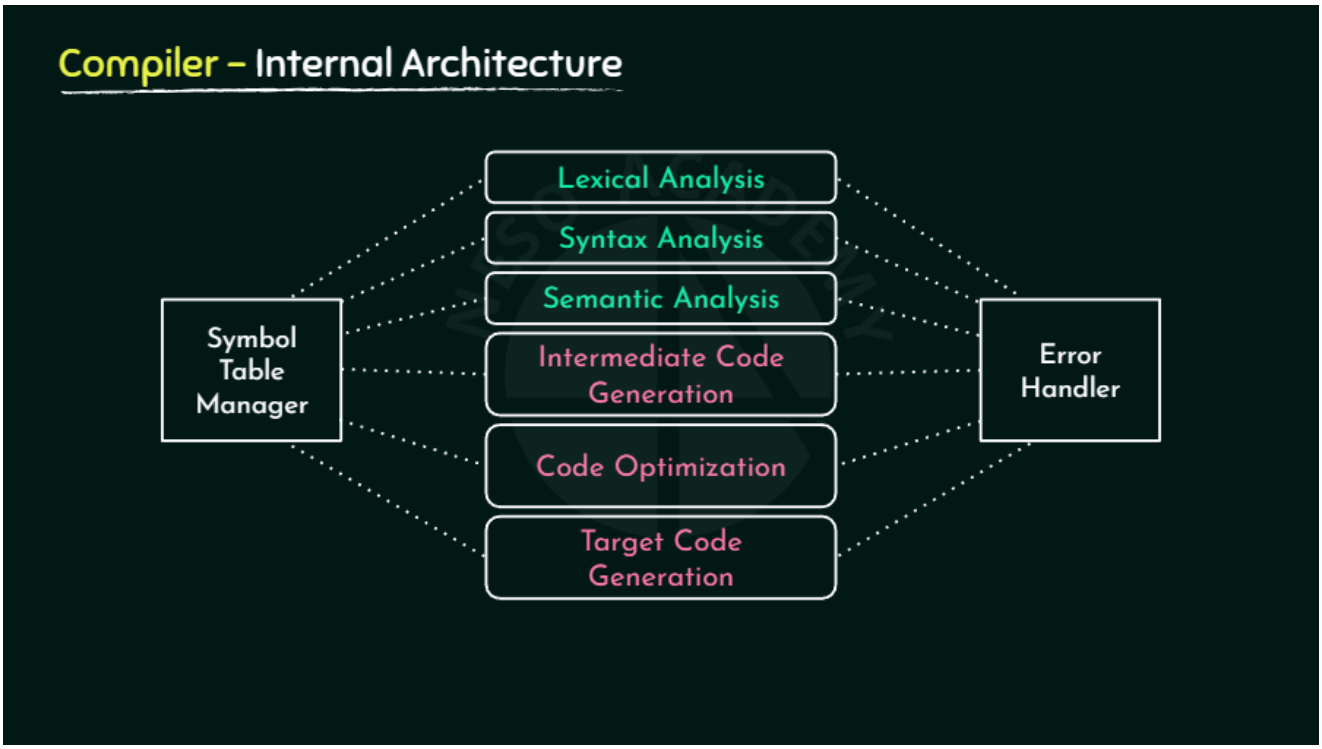
Semantic AnalysisNeso AcademyCHAPTER-5



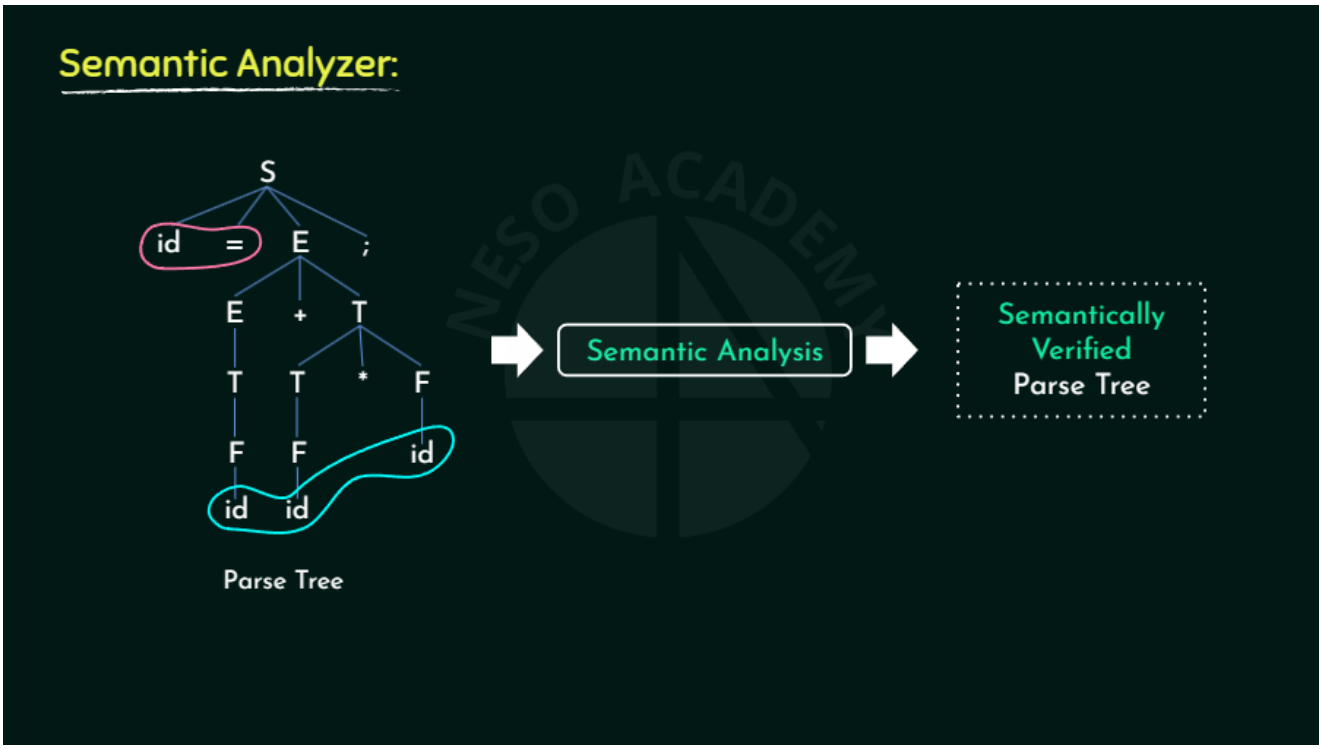
Compiler Design Introduction to SDTS



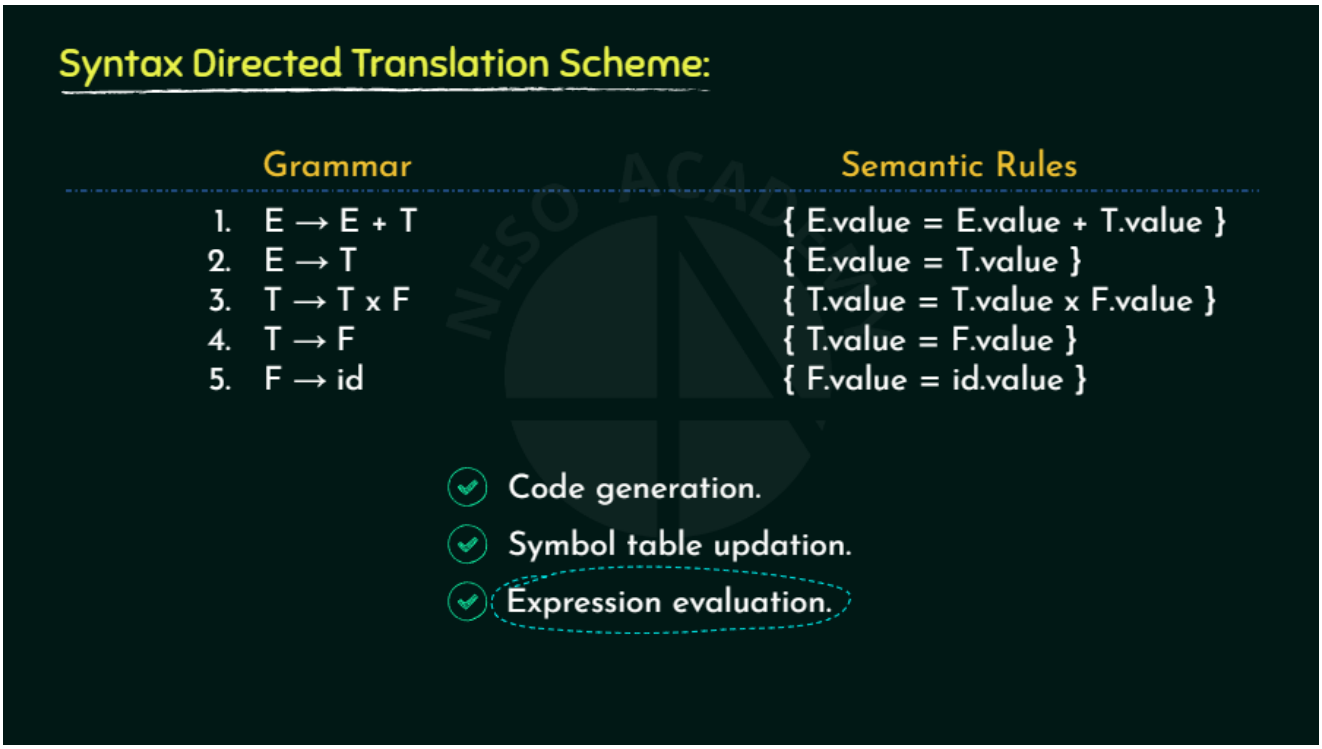
Outcome☆Understanding Syntax Directed Translation Schemes (SDTS).☆How to produce semantically verified parse trees.



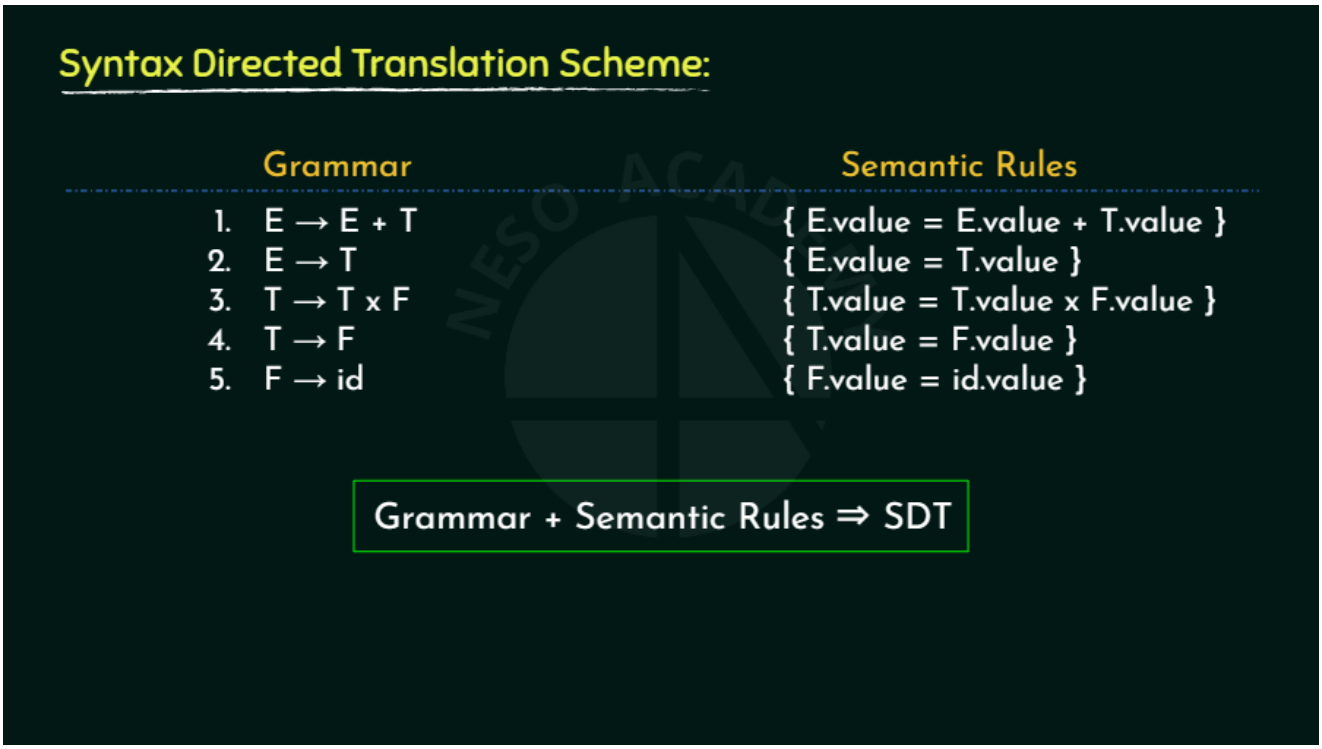
Lexical AnalysisSyntax AnalysisSemantic AnalysisIntermediate Code GenerationCode OptimizationTarget Code GenerationSymbol Table ManagerError HandlerCompiler - Internal Architecture



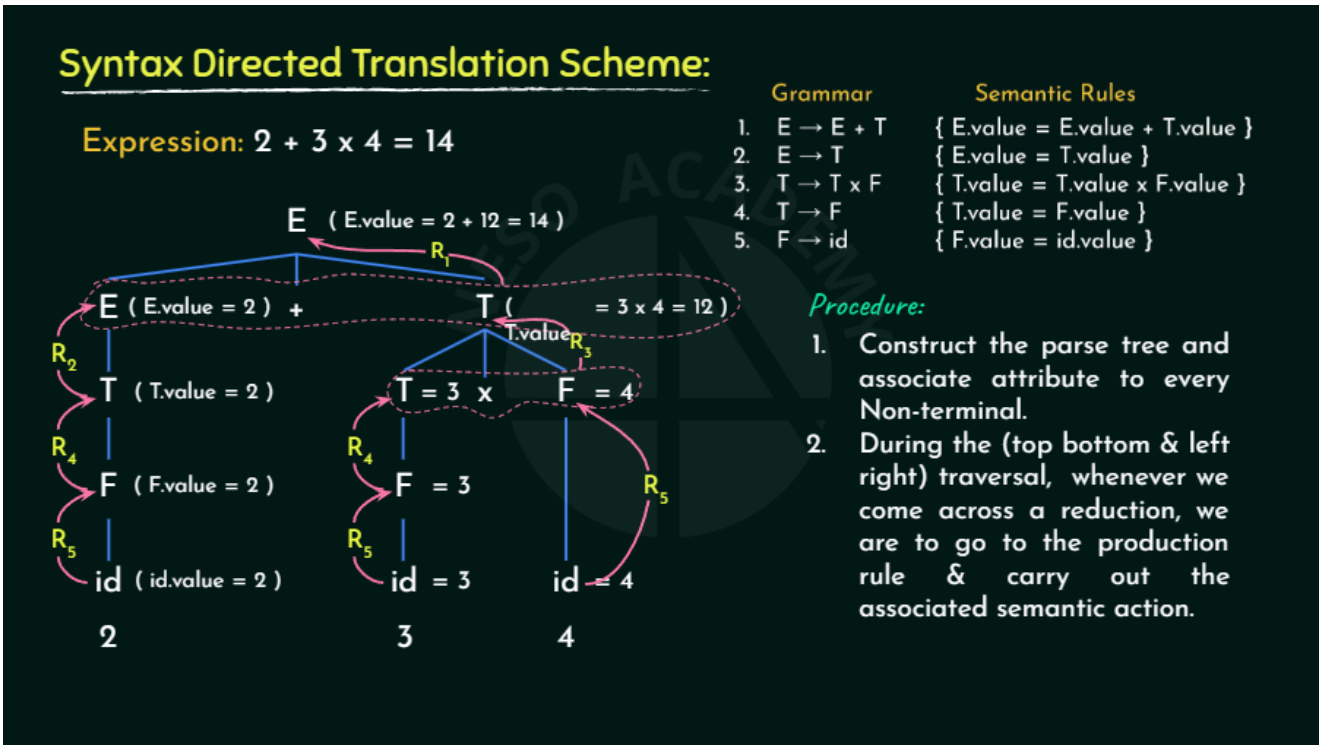
Semantic AnalysisSid=E;E+TT*FTFFidididParse Tree Semantically VerifiedParse Tree Semantic Analyzer:



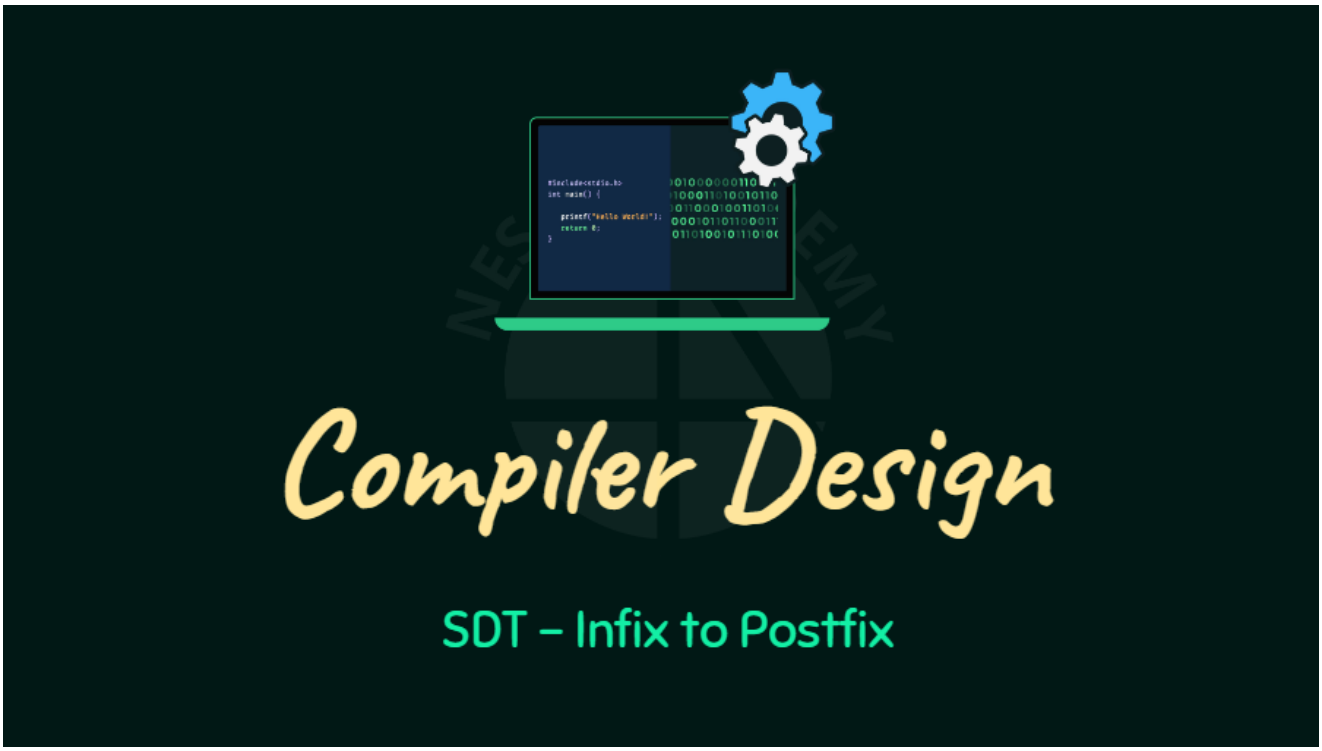
Syntax Directed Translation Scheme:1.E → E + T 2.E → T 3.T → T x F 4.T → F5.F → idGrammar E → E + TE → T T → T x F T → FF → id{ E.value = E.value + T.value } { E.value = T.value }{ T.value = T.value x F.value }{ T.value = F.value }{ F.value = id.value }Semantic Rules Code generation.Symbol table updation.Expression evaluation.



⇒ SDT Grammar + Semantic Rules Grammar Syntax Directed Translation Scheme:Semantic Rules 1.E → E + T 2.E → T 3.T → T x F 4.T → F5.F → idE → E + TE → T T → T x F T → FF → id{ E.value = E.value + T.value } { E.value = T.value }{ T.value = T.value x F.value }{ T.value = F.value }{ F.value = id.value }

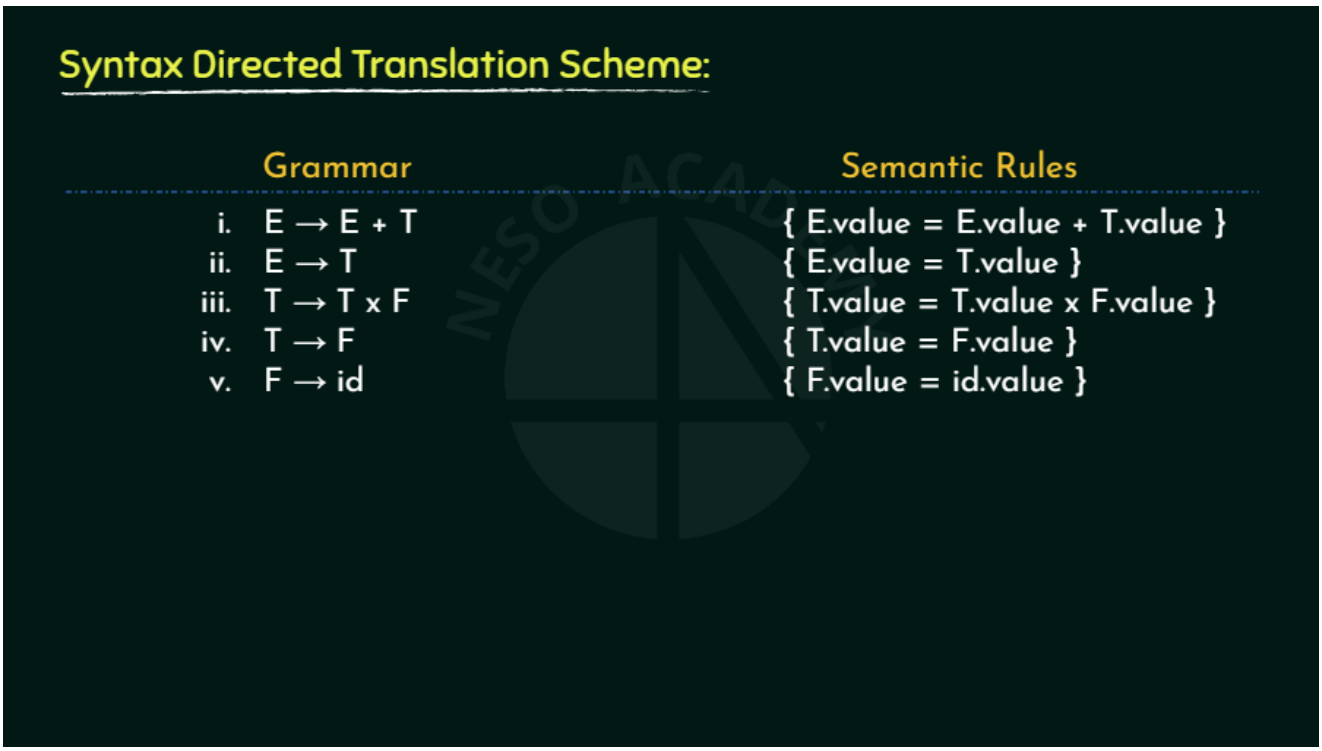


= 12)= 14)Syntax Directed Translation Scheme:Expression:2 + 3 x 4 = 14EE+TTxFTFidFidid(id.value = 2)(F.value = 2)(T.value = 2)(E.value = 2)= 3 = 3 = 3 = 4= 4R5R4R2R5R4R5R3R1Procedure:1.Construct the parse tree and associate attribute to every Non-terminal.2.During the (top bottom & left right) traversal, whenever we come across a reduction, we are to go to the production rule & carry out the associated semantic action. (E.value = 2 + 12(T.value = 3x 4E.valueT.valueF.valueid.value4 2 3

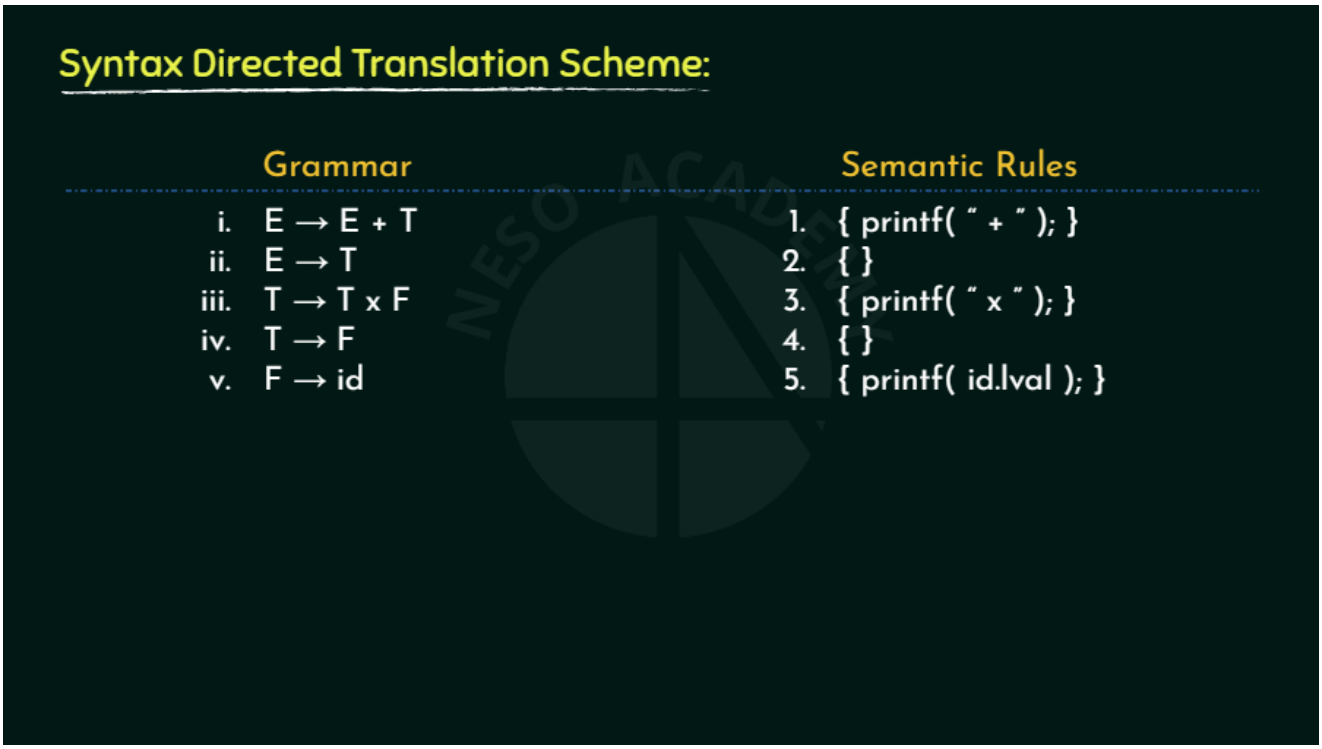


Compiler Design SDT - Infix to Postfix

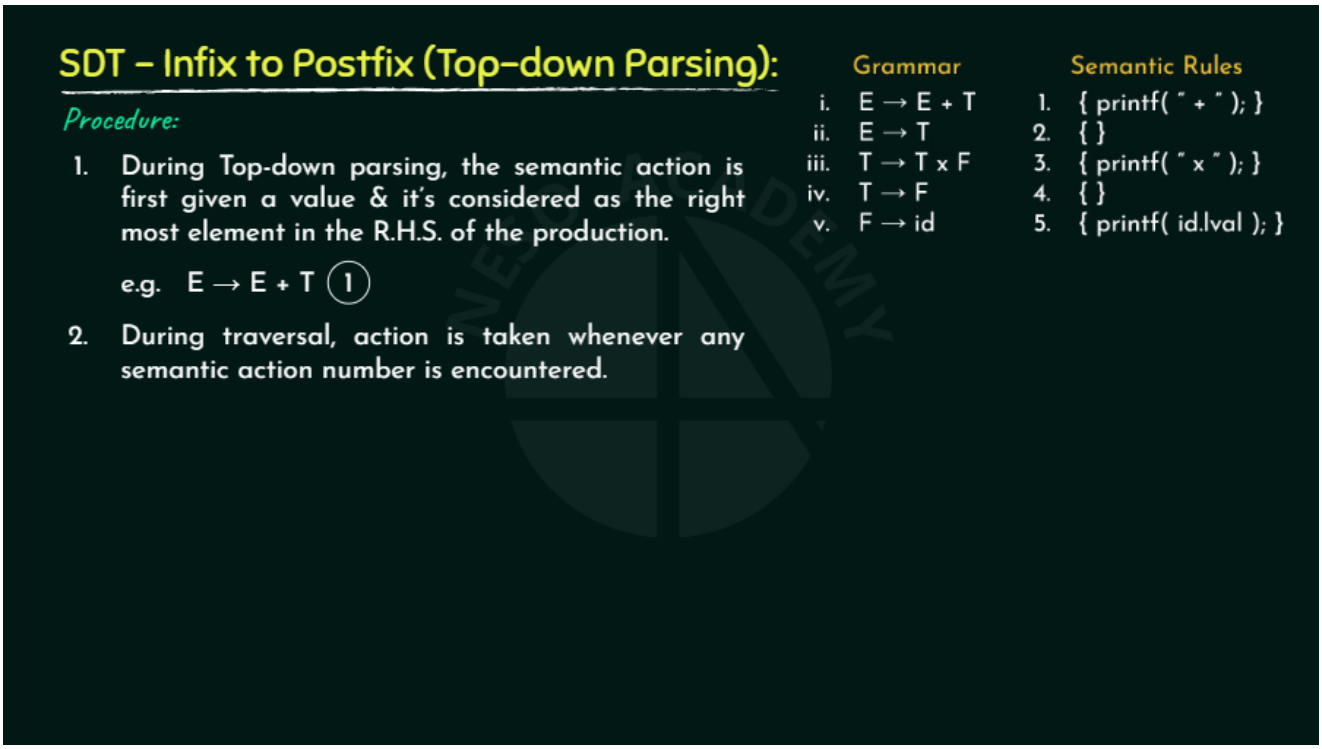
Outcome☆Top-down parsing of SDT.☆Bottom-up parsing of SDT.



Syntax Directed Translation Scheme:i.E → E + T ii.E → T iii.T → T x F iv.T → Fv.F → idGrammar E → E + TE → T T → T x F T → FF → id{ E.value = E.value + T.value } { E.value = T.value }{ T.value = T.value x F.value }{ T.value = F.value }{ F.value = id.value }Semantic Rules

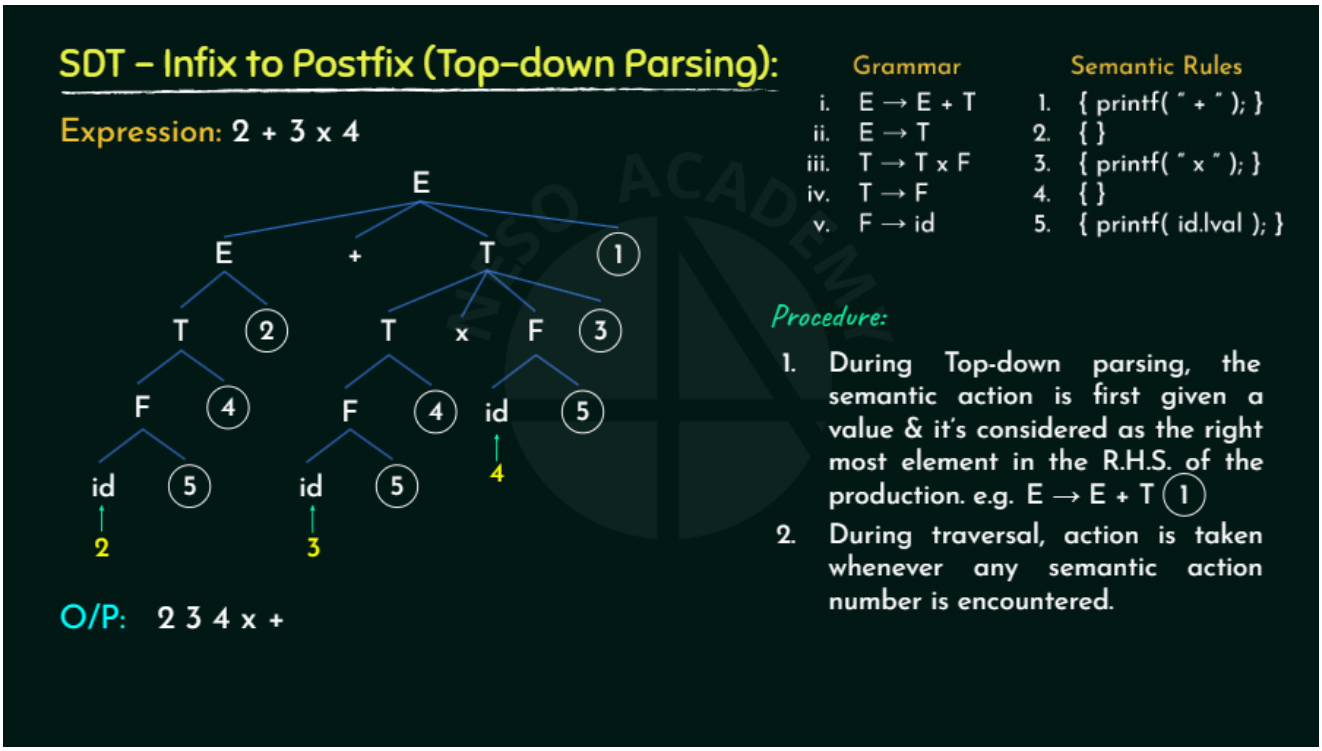


1.{ printf(" + "); } 2.{ }3.{ printf(" x "); }4.{ }5.{ printf(id.lval); }Syntax Directed Translation Scheme:i.E → E + T ii.E → T iii.T → T x F iv.T → Fv.F → idGrammar E → E + TE → T T → T x F T → FF → id{ printf(" + "); } { }{ printf(" x "); }{ }{ printf(id.lval); }Semantic Rules

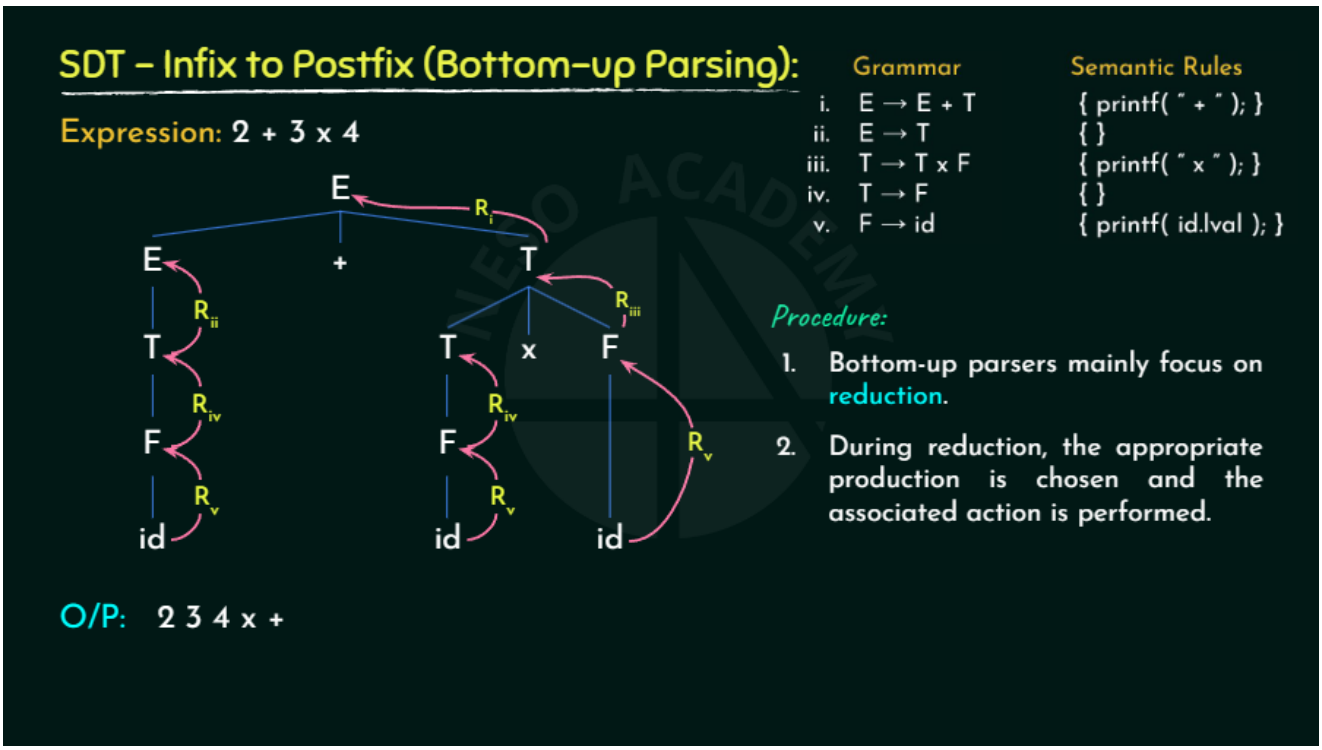


SDT - Infix to Postfix (Top-down Parsing):Procedure:1.During Top-down parsing, the semantic action is first given a value & it's considered as the right most element in the R.H.S. of the production.e.g.E → E + T12.During traversal, action is taken whenever any semantic

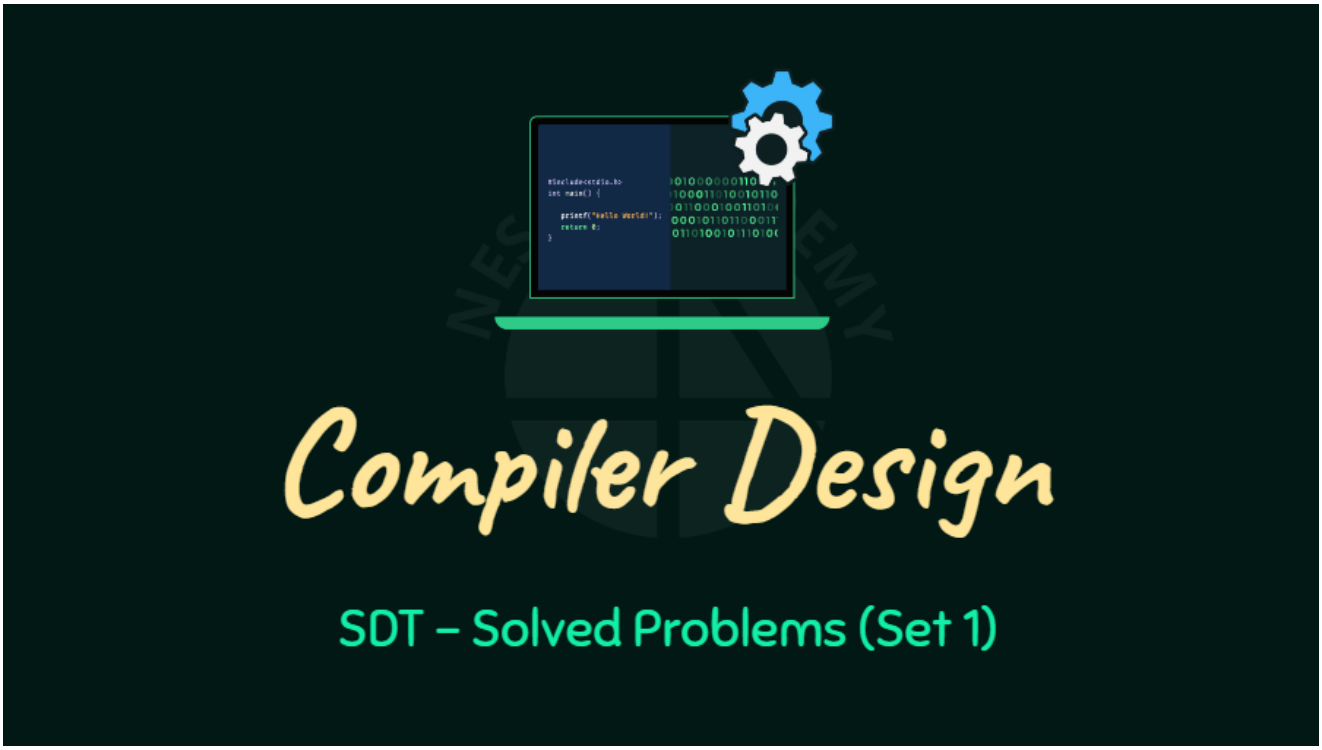
action number is encountered.



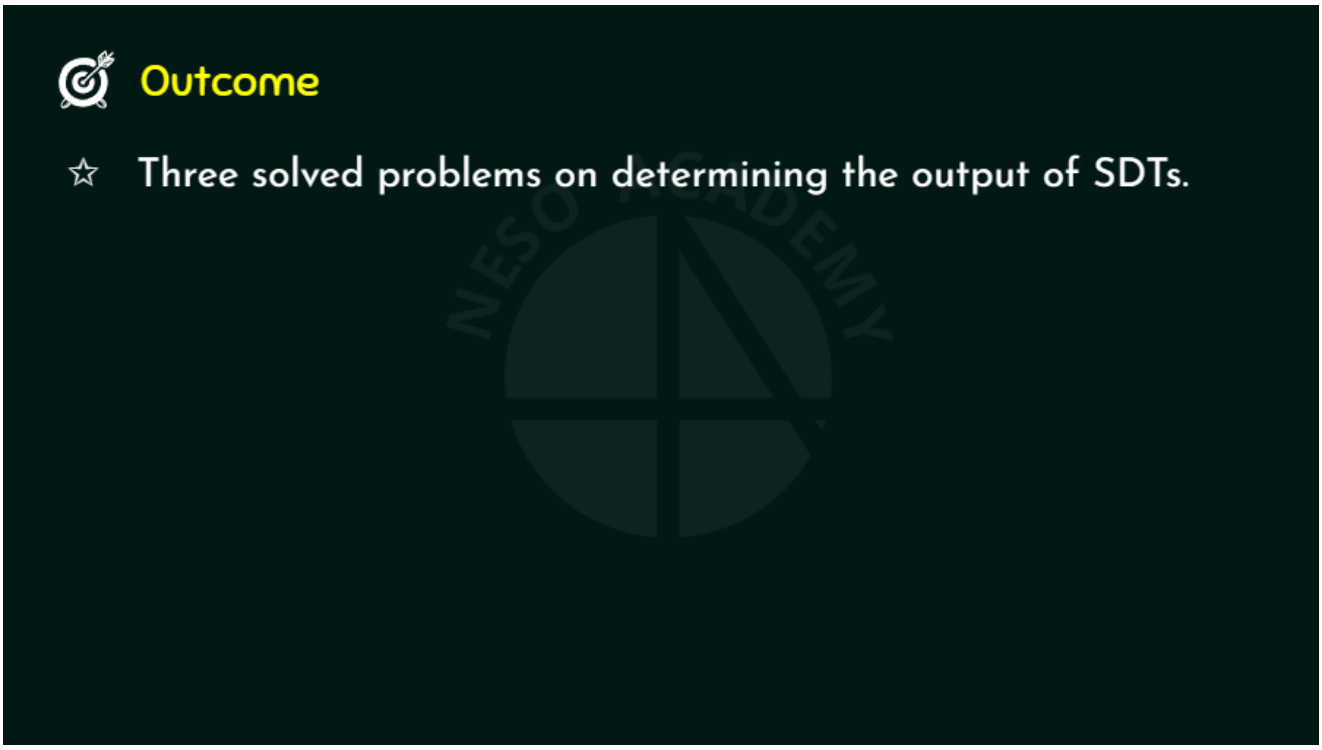
SDT - Infix to Postfix (Top-down Parsing):Procedure:1.During Top-down parsing, the semantic action is first given a value & it's considered as the right most element in the R.H.S. of the production. e.g.E → E + T12.During traversal, action is taken whenever any semantic action number is encountered.EE+T1Expression:2 + 3 x 4 T24F5id4F5id5idTxF3243O/P:2 34x+



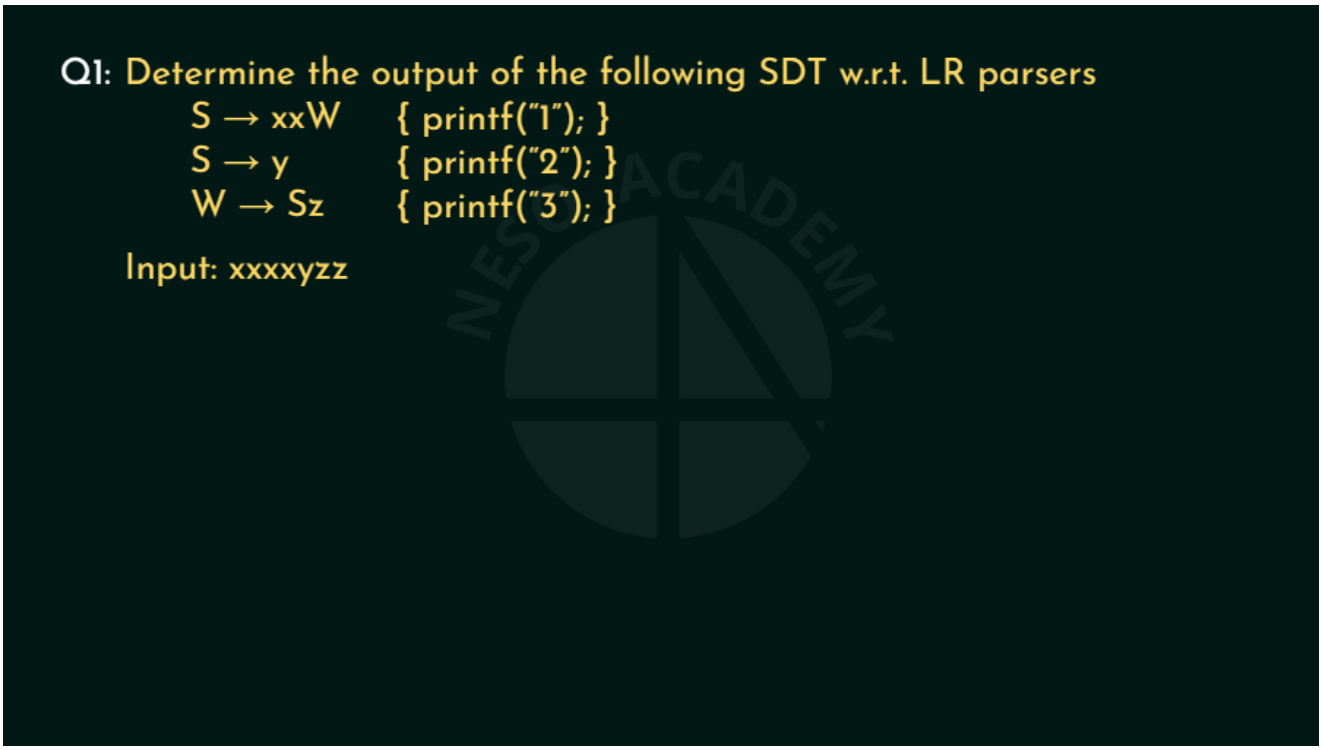
SDT - Infix to Postfix (Bottom-up Parsing):Procedure:1.Bottom-up parsers mainly focus on reduction.2.During reduction, the appropriate production is chosen and the associated action is performed.Expression:2 + 3 x 4 O/P:2 34x+ETFidFididTxFRiiRvRiiiRiE+TRivRvRivRv



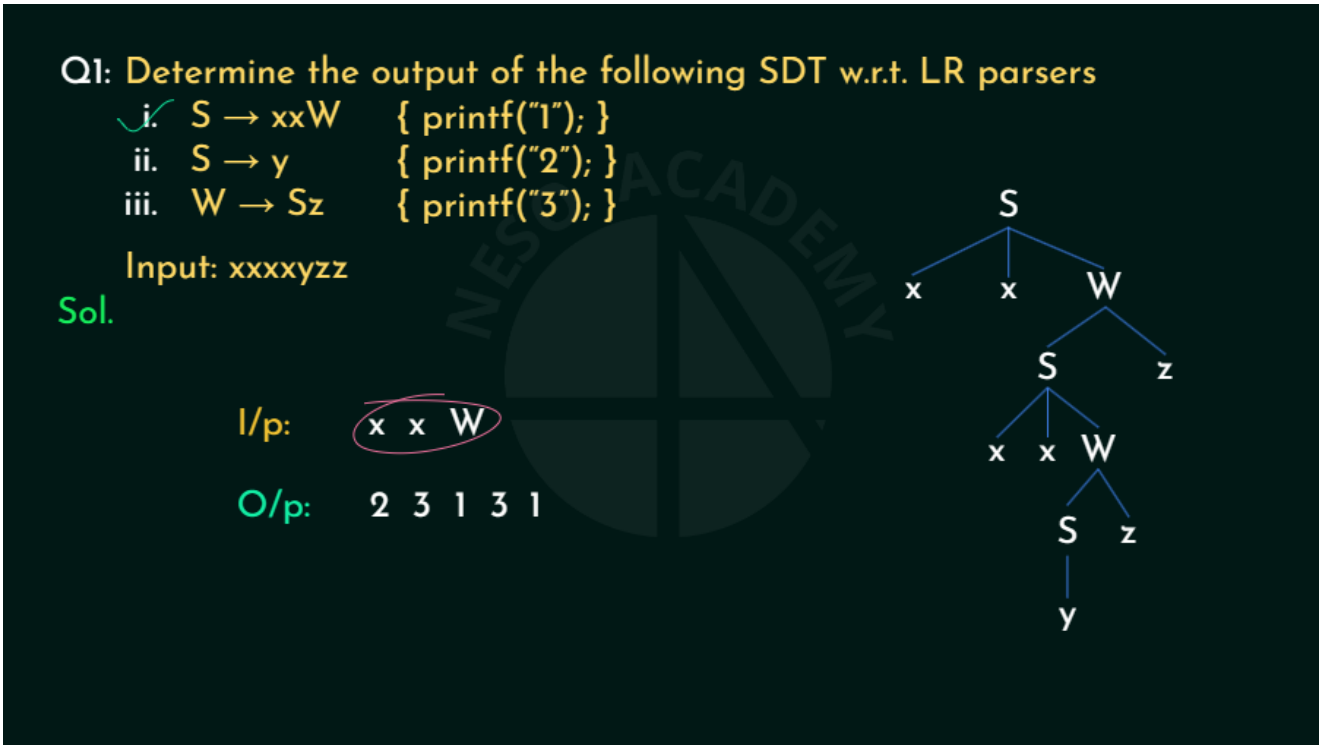
Compiler Design SDT - Solved Problems (Set 1)



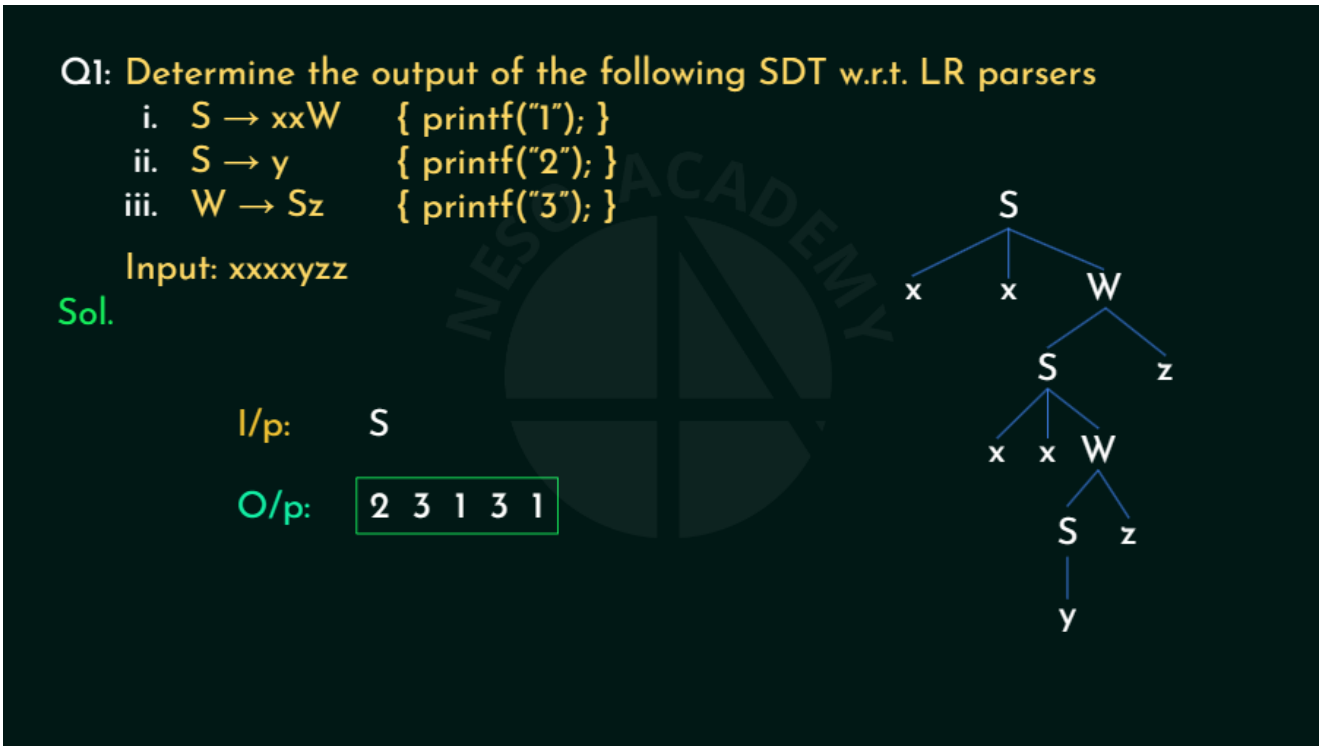
Outcome☆Three solved problems on determining the output of SDTs.



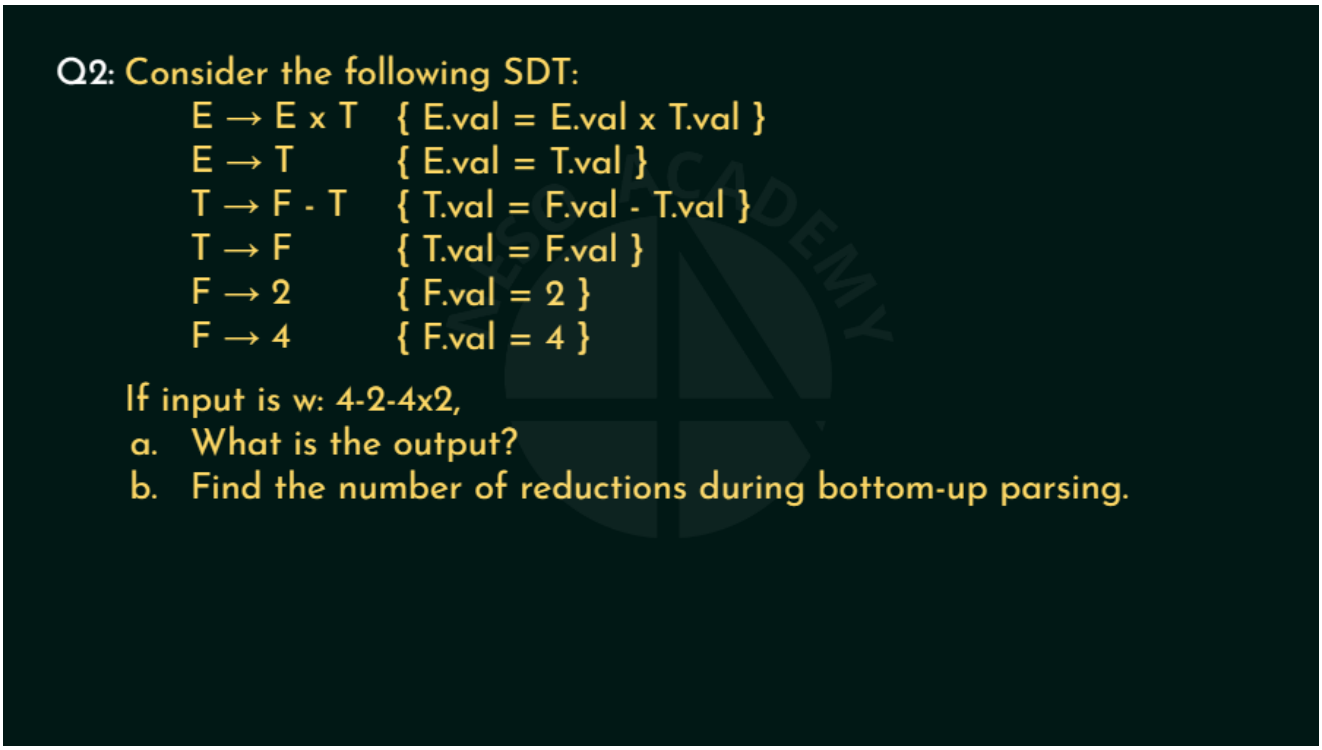
Q1:Determine the output of the following SDT w.r.t. LR parsersi.S → xxWii.S → yiii.W → SzInput: xxxxyzz{ printf("1"); }{ printf("2"); }{ printf("3"); }



Determine the output of the following SDT w.r.t. LR parsersi.S → xxWii.S → yiii.W → SzInput: xxxxyzzQ1:{ printf("1"); }{ printf("2"); }{ printf("3"); }Sol.xxWSzxWxSzySI/p: x x WO/p: 2 31 31

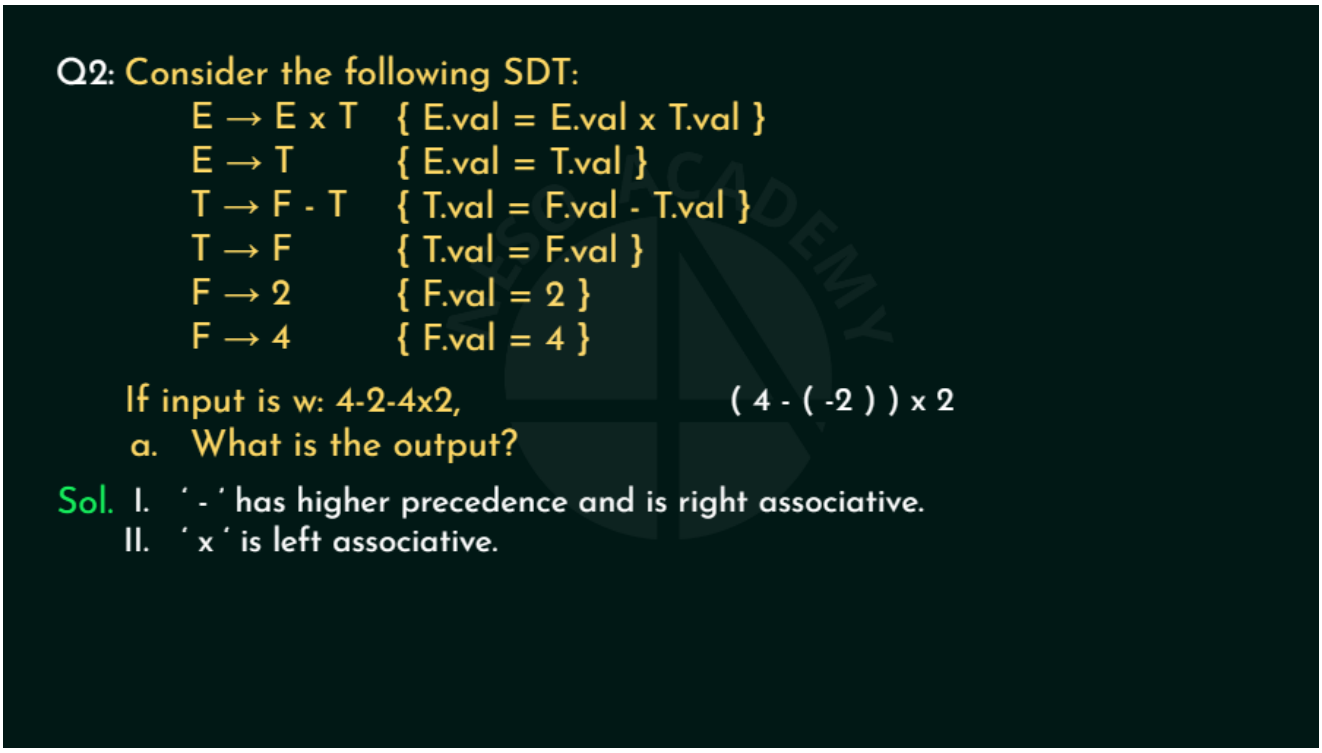


Determine the output of the following SDT w.r.t. LR parsersi.S → xxWii.S → yiii.W → SzInput: xxxxyzzQ1:{ printf("1"); }{ printf("2"); }{ printf("3"); }Sol.xxWSzxWxSzySI/p: SO/p: 2 31 31

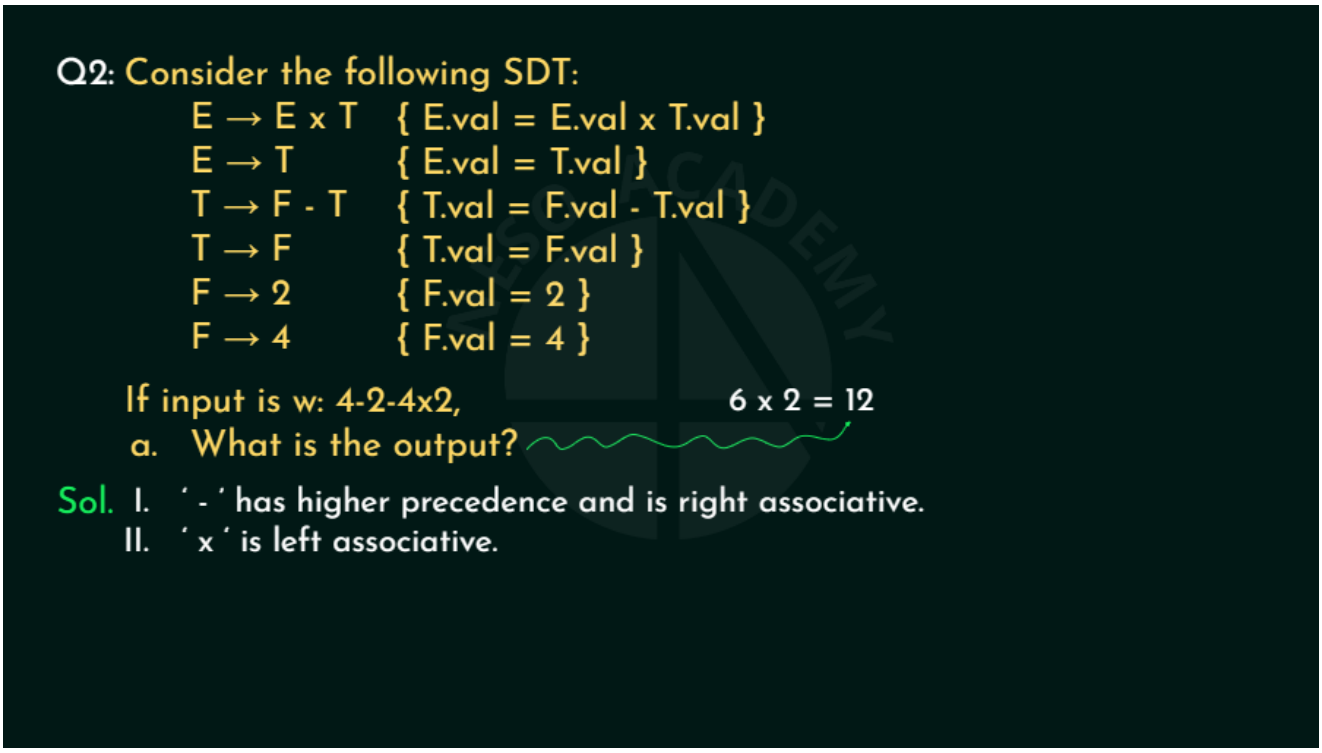


Consider the following SDT:i.E → E x Tii.E → Tiii.T → F - Tiv.T → Fv.F → 2vi.F → 4If input is w: 4-2-4x2,a.What is the output?b.Find the number of reductions during bottom-up parsing.Q2:{ E.val = E.val x T.val }{ E.val = T.val }{ T.val = F.val - T.val }{ T.val = F.val }{ F.val =

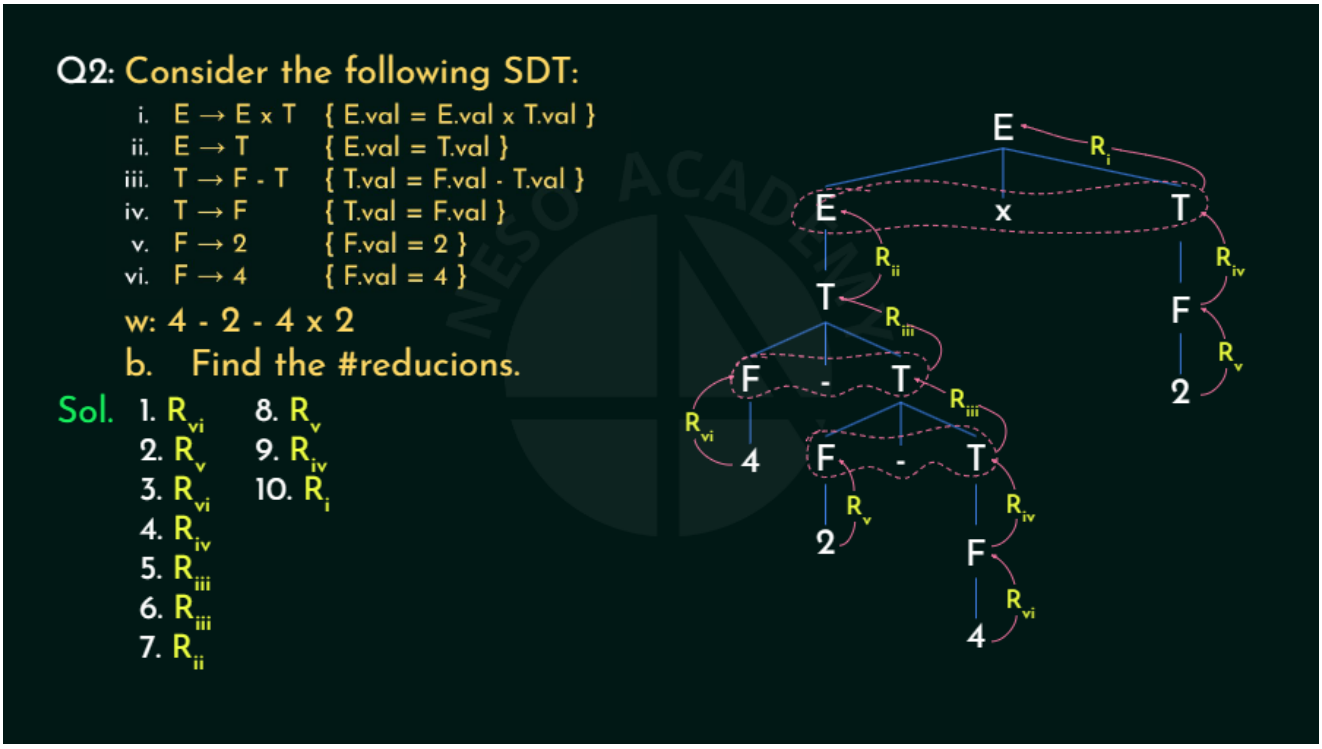
2 }{ F.val = 4 }



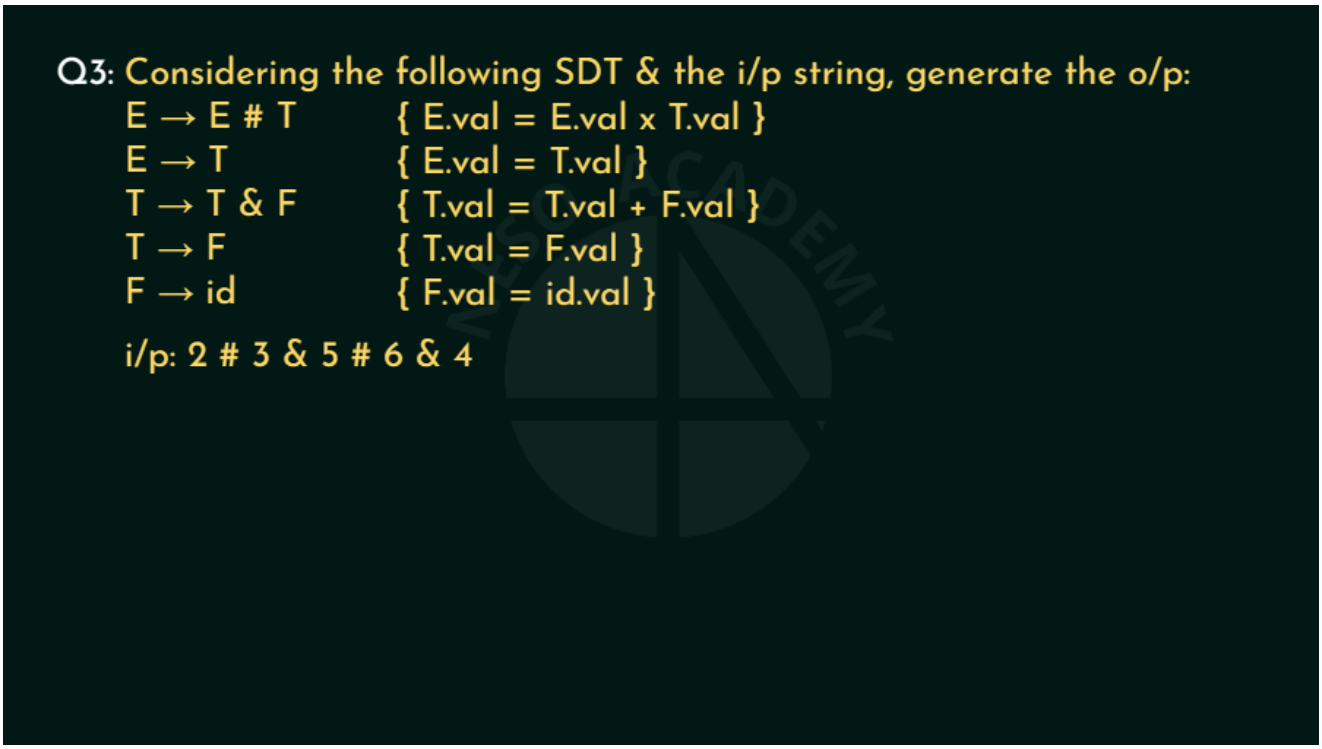
Consider the following SDT:i.E → E x Tii.E → Tiii.T → F - Tiv.T → Fv.F → 2vi.F → 4If input is w: 4-2-4x2,a.What is the output?Q2:{ E.val = E.val x T.val }{ E.val = T.val }{ T.val = F.val - T.val }{ T.val = F.val }{ F.val = 2 }{ F.val = 4 }Sol.I.' - ' has higher precedence and is right associative.II.' x ' is left associative.(4 - (-2)) x 2



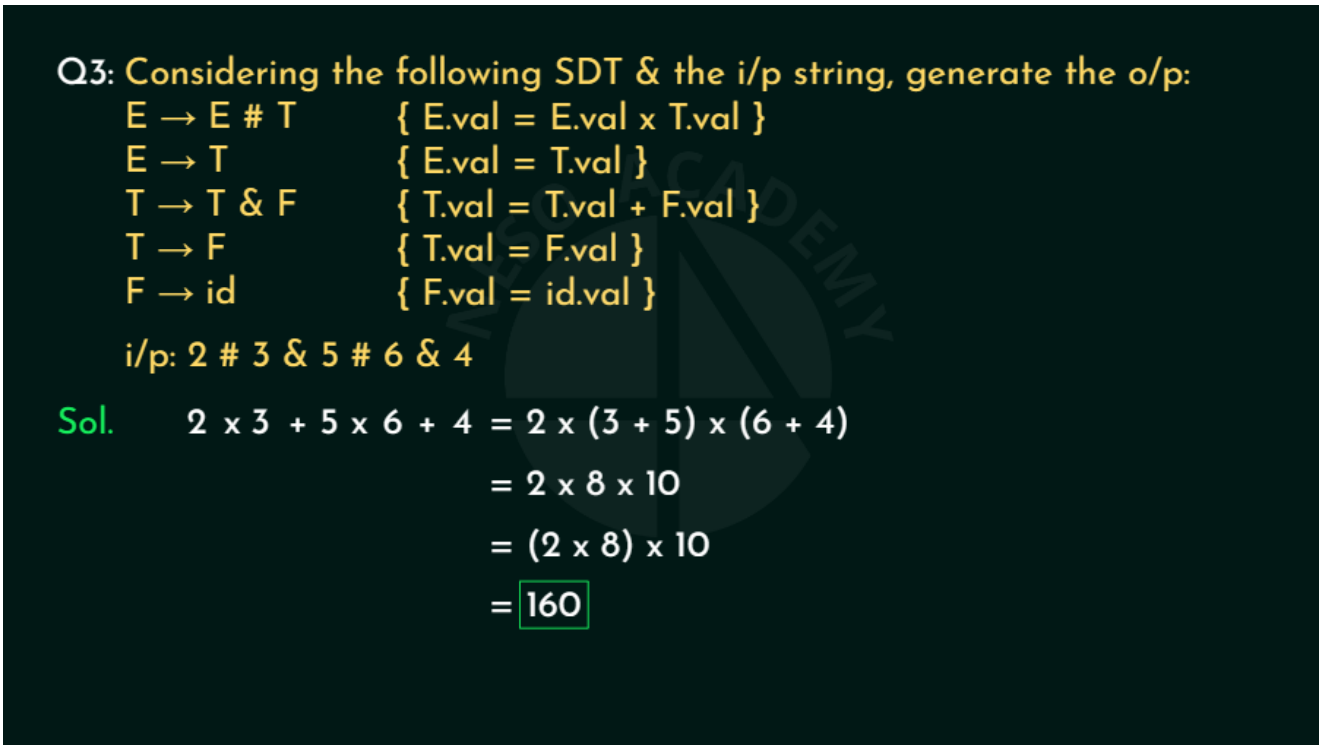
Consider the following SDT:i.E → E x Tii.E → Tiii.T → F - Tiv.T → Fv.F → 2vi.F → 4If input is w: 4-2-4x2,a.What is the output?Q2:{ E.val = E.val x T.val }{ E.val = T.val }{ T.val = F.val - T.val }{ T.val = F.val }{ F.val = 2 }{ F.val = 4 }Sol.I.‘ - ’ has higher precedence and is right associative.II.‘ x ‘ is left associative.6 x 2= 12



w: 4 - 2 - 4 x 2b.Find the #reducions.7. Rii8. Rv9. RivConsider the following SDT:Q2:Sol.EExTTFT-FT-FF4242RviRvRviRivRiiiRiiiRiiRvRivRi1. Rvi2. Rv3. Rvi4. Riv5. Riii6. Riii10. Ri

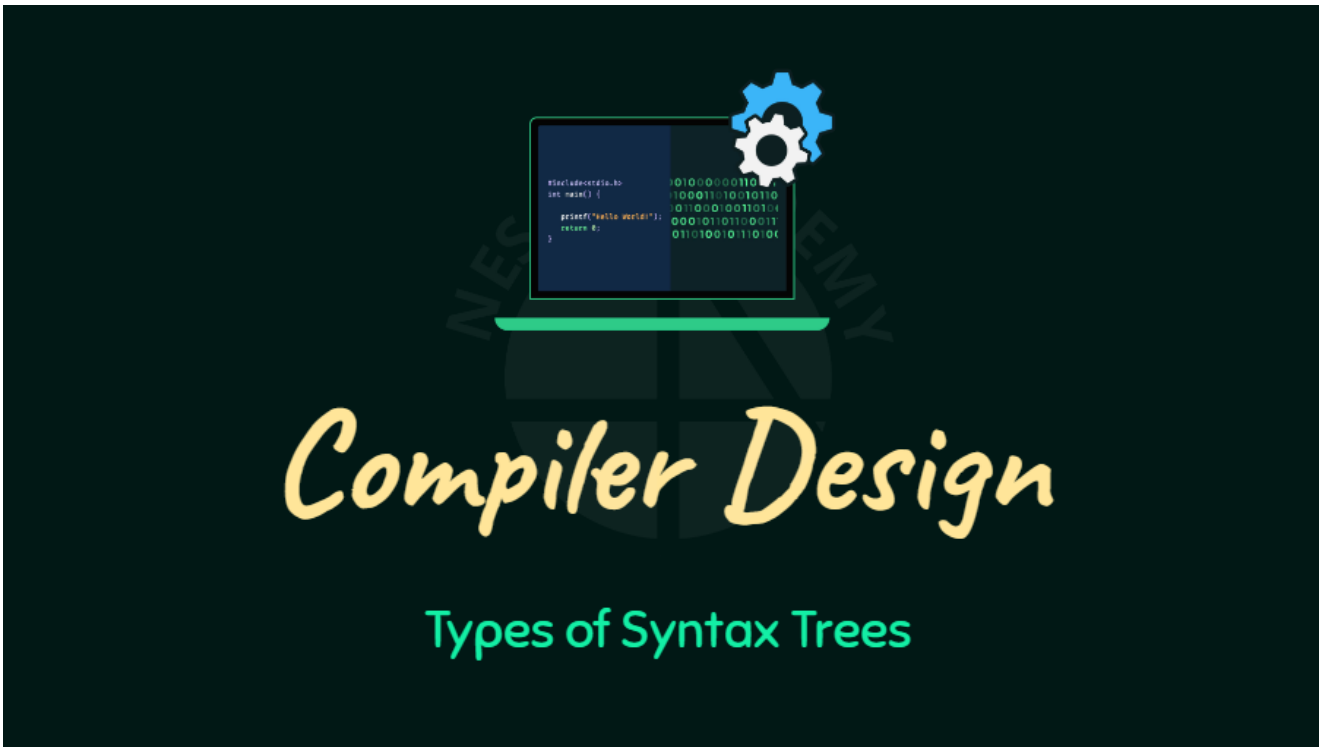


Considering the following SDT & the i/p string, generate the o/p:E → E # TE → TT → T & FT → FF → idi/p: 2 # 3 & 5 # 6 & 4Q3:{ E.val = E.val x T.val }{ E.val = T.val }{ T.val = T.val + F.val }{ T.val = F.val }{ F.val = id.val }

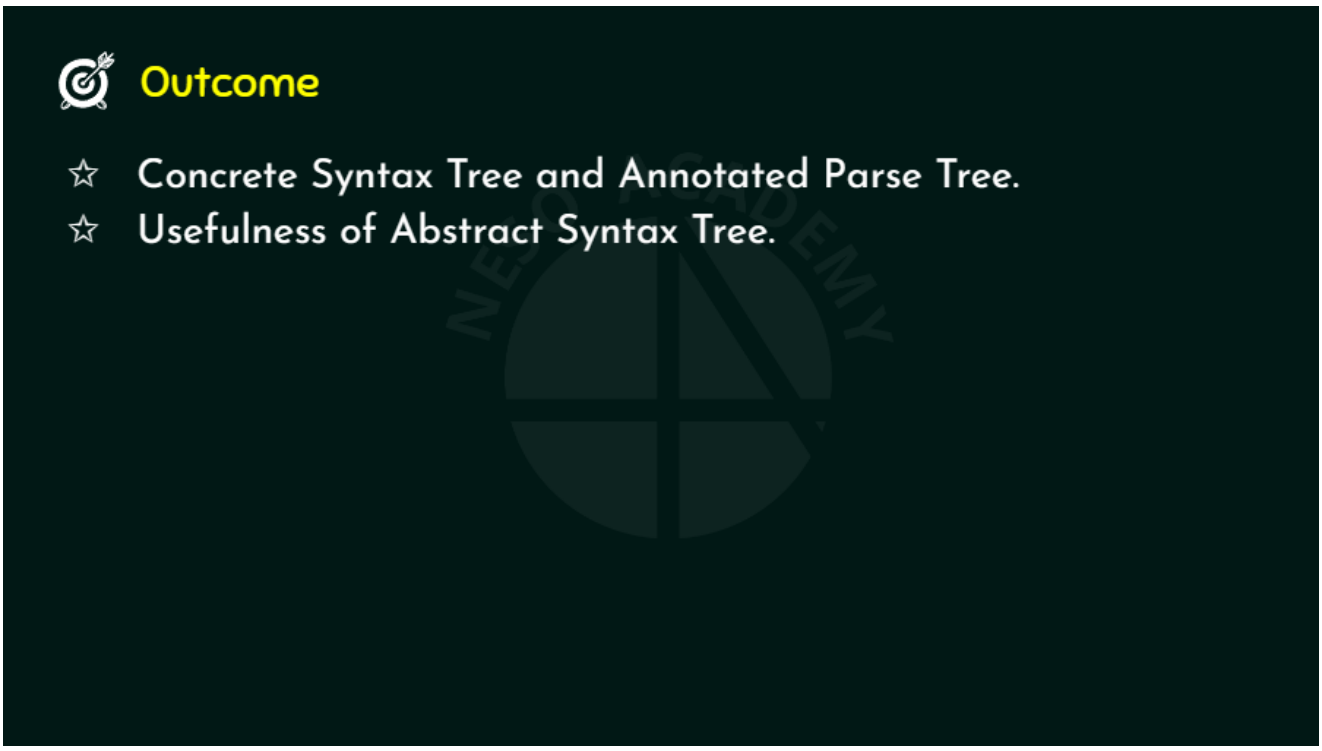


Considering the following SDT & the i/p string, generate the o/p:E → E # TE → TT → T & FT → FF → idi/p: 2 # 3 & 5 # 6 & 4Q3:Sol.{ E.val = E.val x T.val }{ E.val = T.val }{ T.val = T.val + F.val }{ T.val = F.val }{ F.val = id.val }2 # 3 & 5 # 6 & 4xx++= 2 x (3 + 5) x (6 + 4)= 2 x 8 x 10=

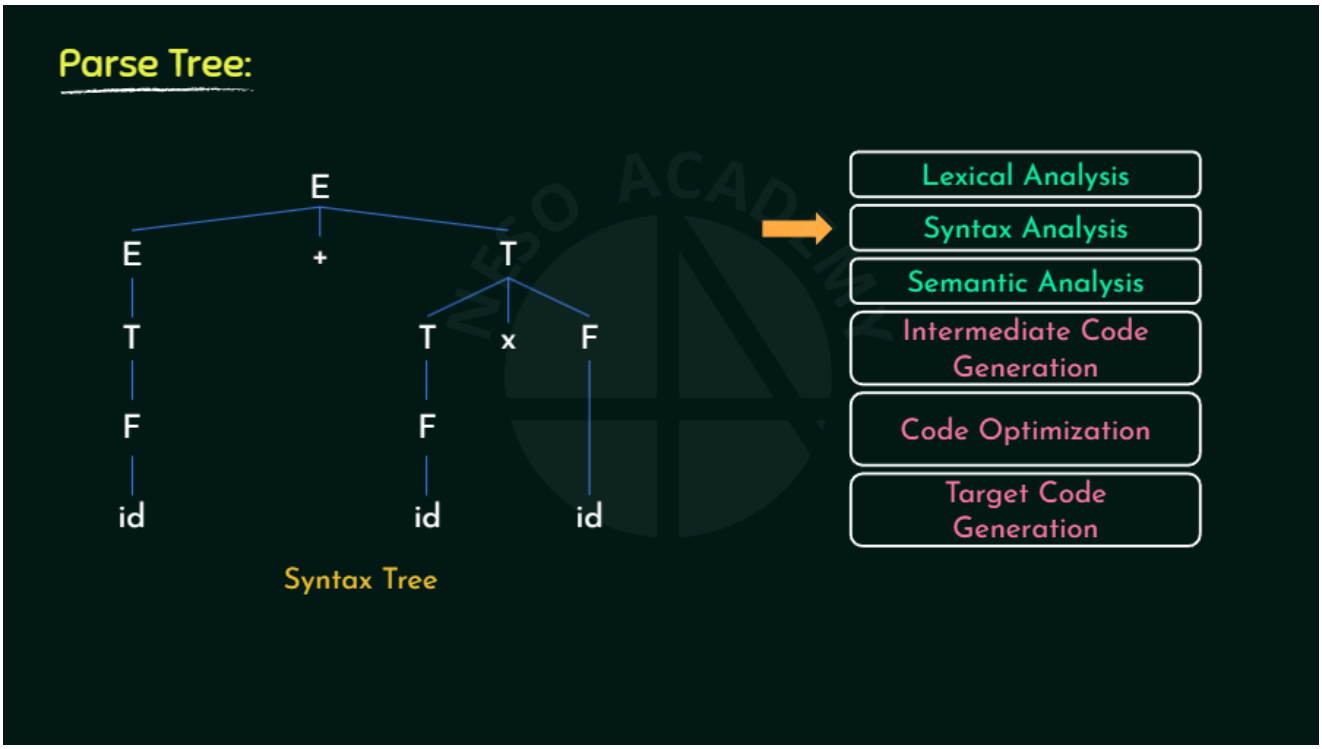
(2 x 8) x 10= 160



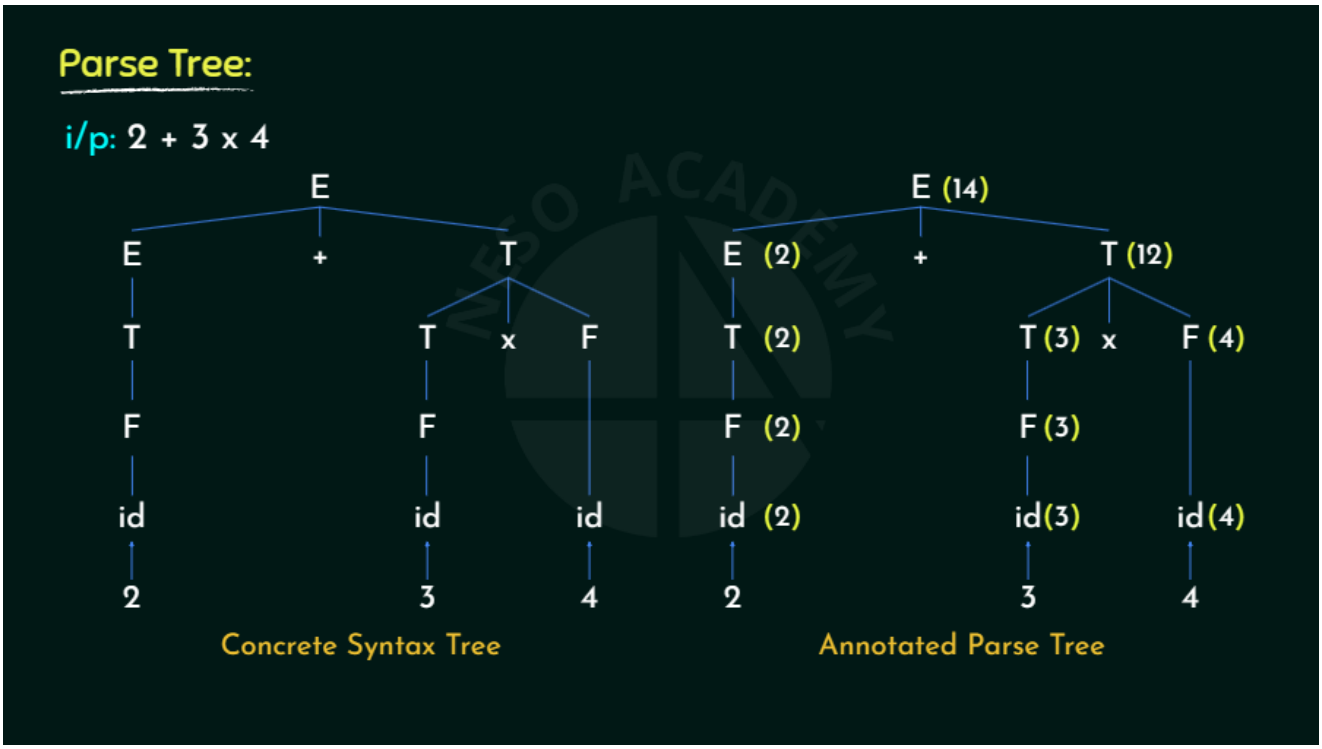
Compiler Design Types of Syntax Trees



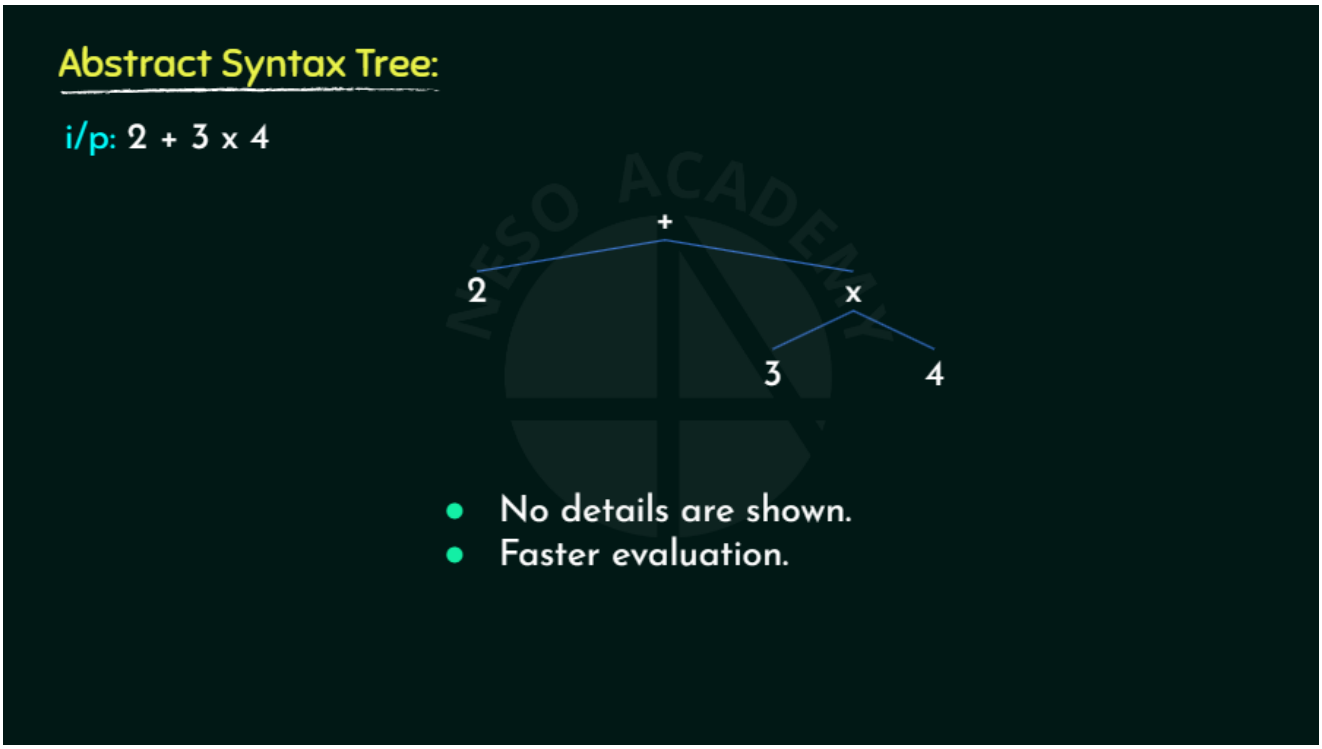
Outcome☆Concrete Syntax Tree and Annotated Parse Tree.☆Usefulness of Abstract Syntax Tree.



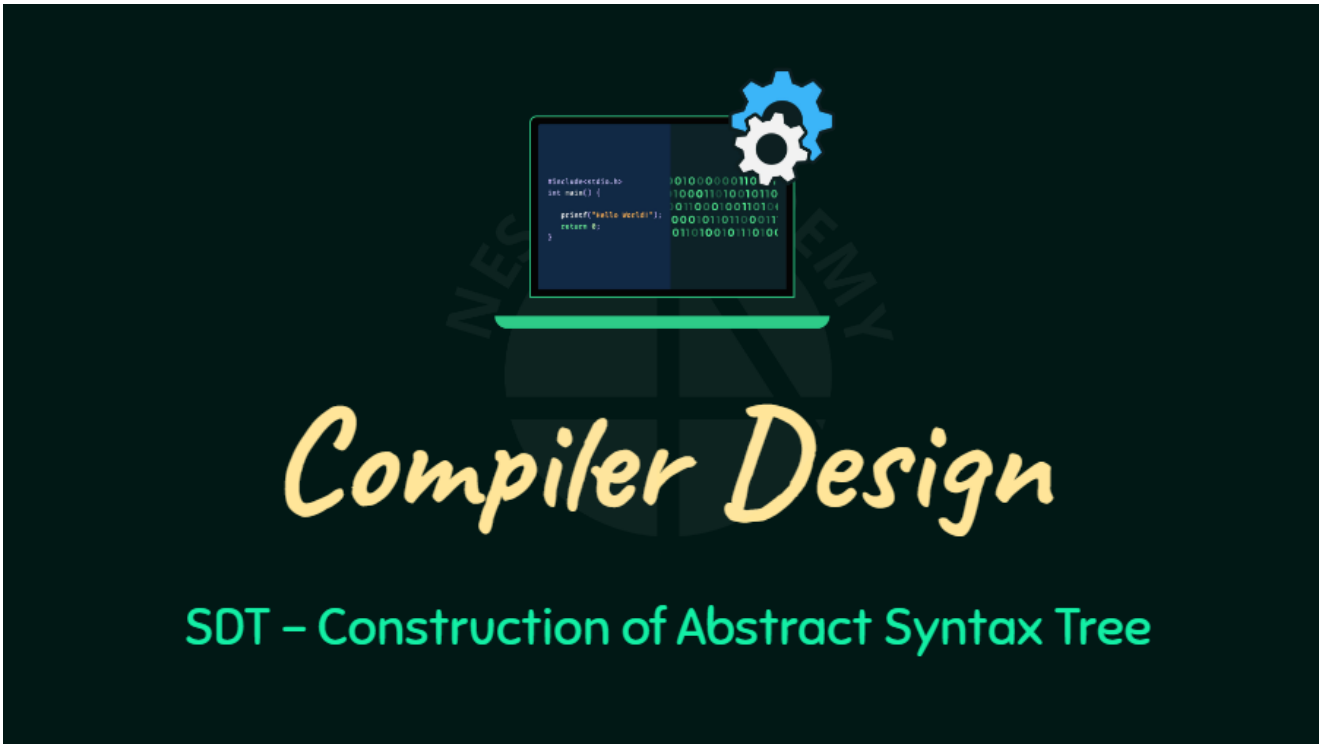
Lexical AnalysisSyntax AnalysisSemantic AnalysisIntermediate Code GenerationCode OptimizationTarget Code GenerationParse Tree:ETFidFididTxFE+TSyntax Tree



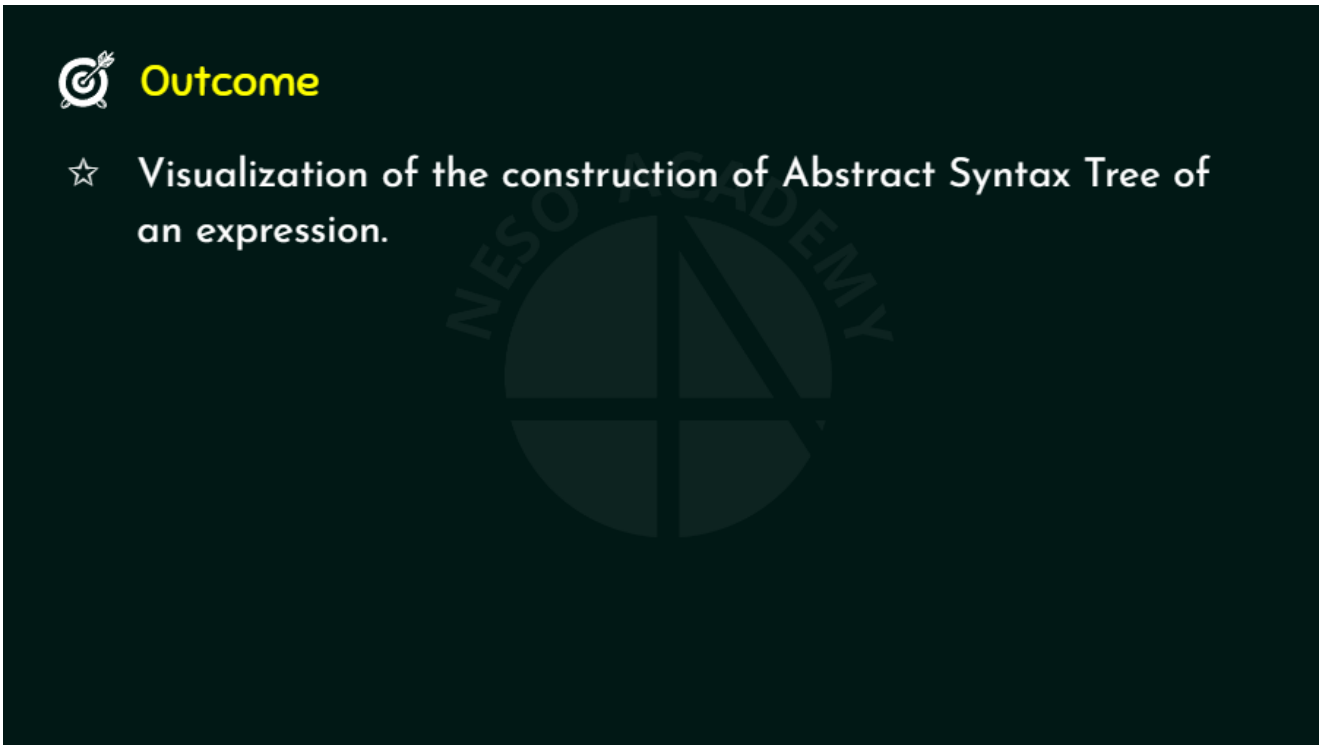
Parse Tree:ETFidFididTxFE+TConcrete Syntax TreeAnnotated Parse Tree i/p: 2 + 3 x 4234ETFidFididTxFE+T234(2)(3)(4)(2)(3)(4)(3)(12)(2)(2)(14)



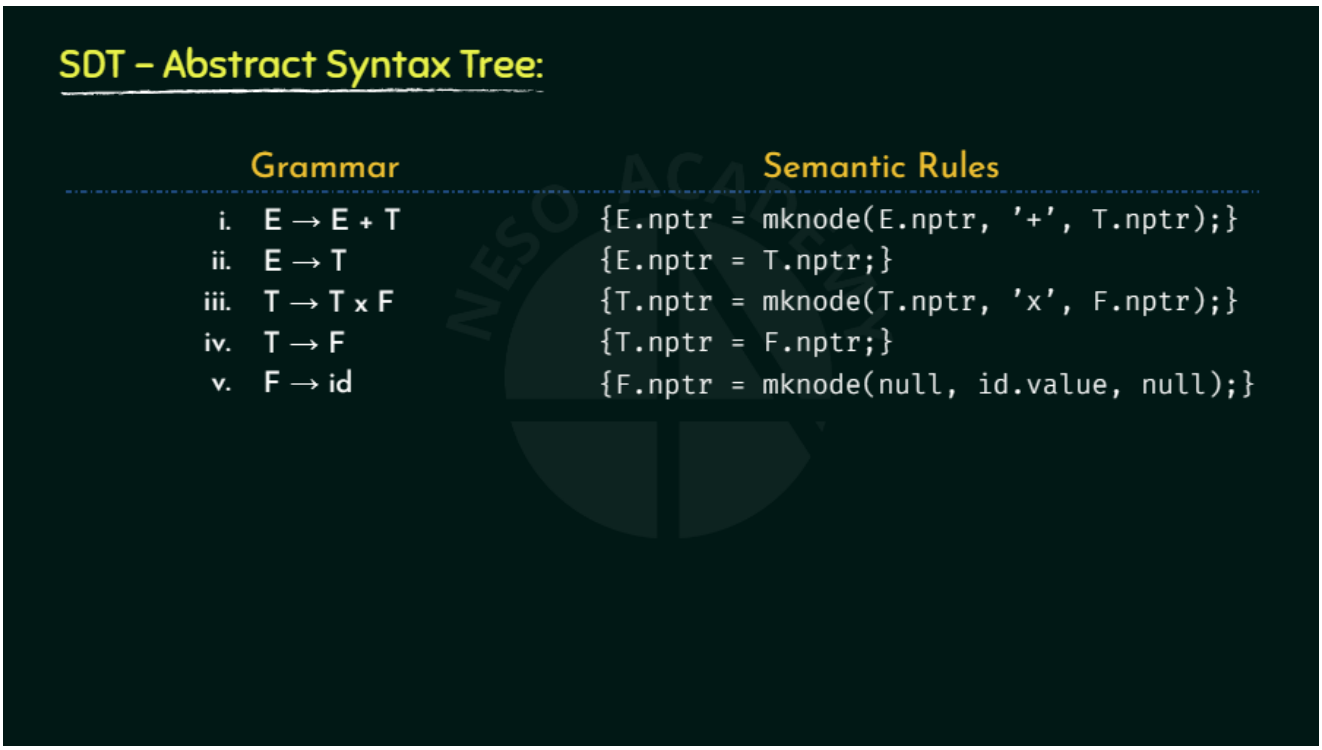
Abstract Syntax Tree: i/p: 2 + 3 x 434x+2●No details are shown.●Faster evaluation.



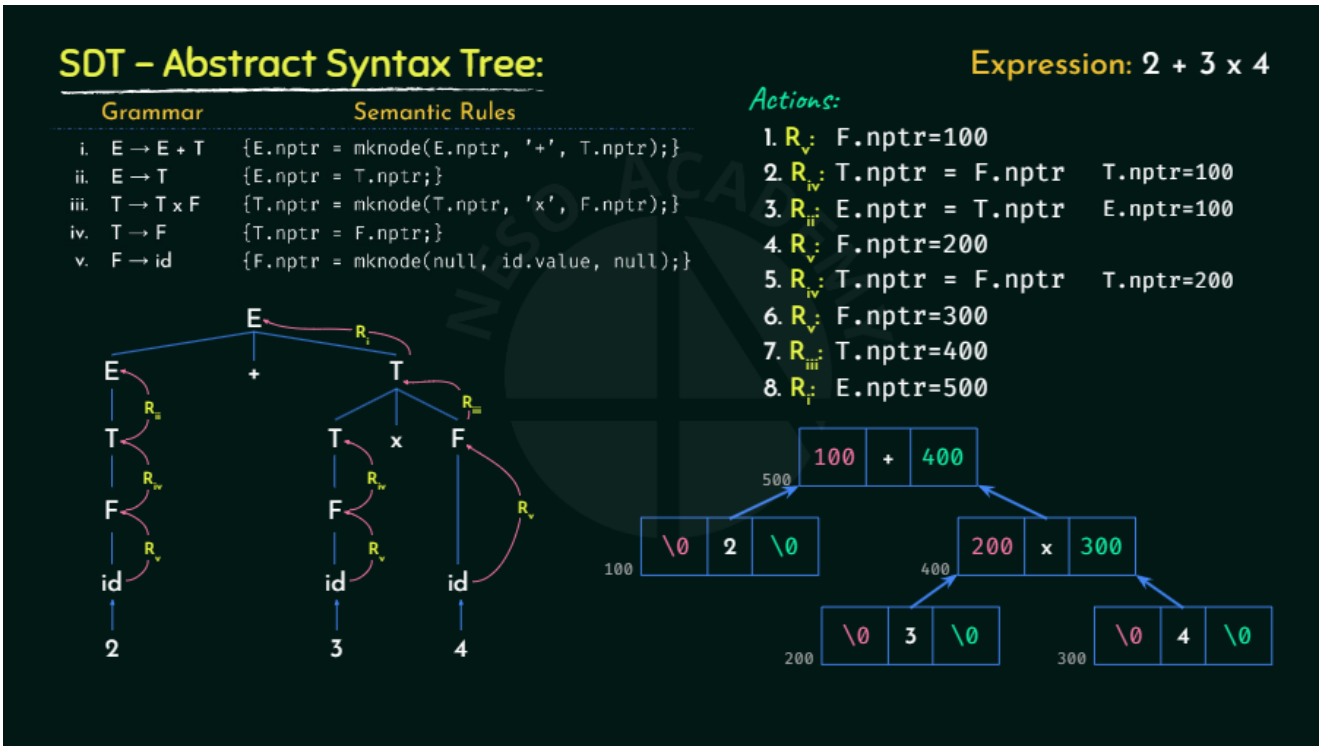
Compiler Design SDT - Construction of Abstract Syntax Tree



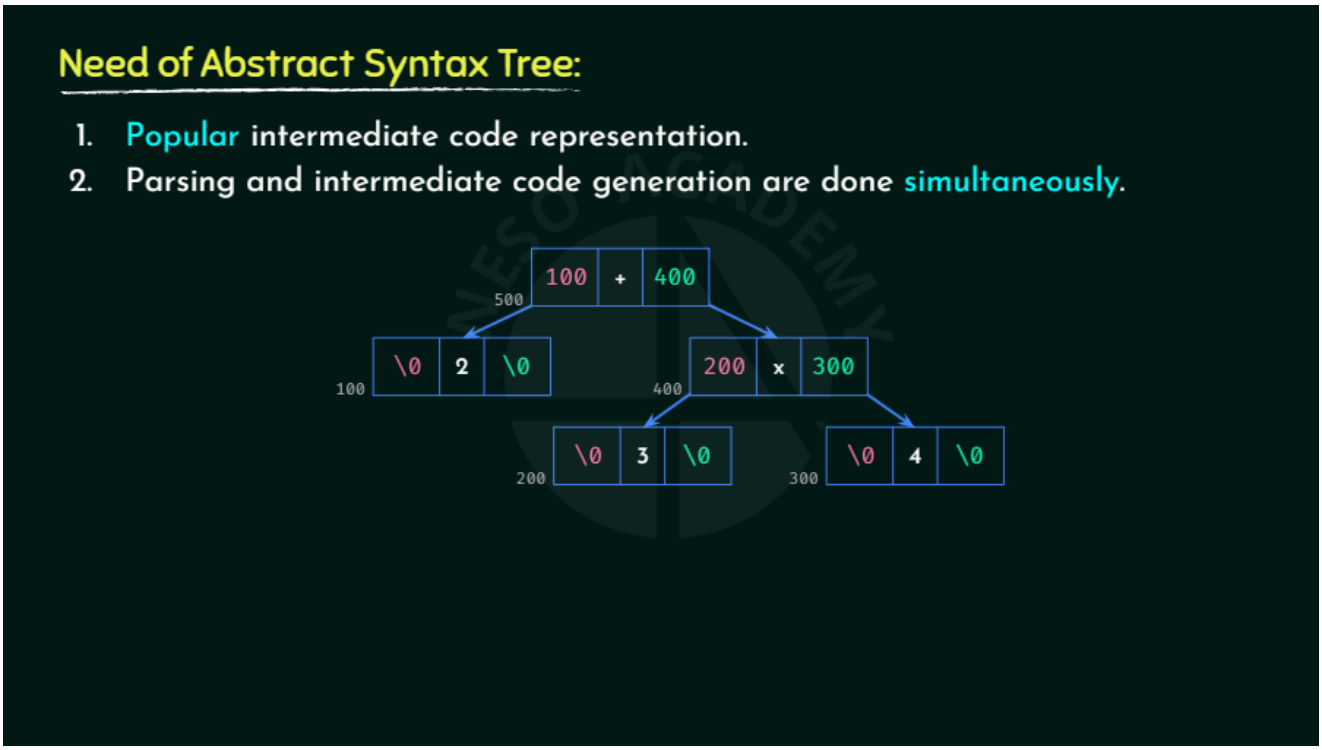
Outcome☆Visualization of the construction of Abstract Syntax Tree of an expression.



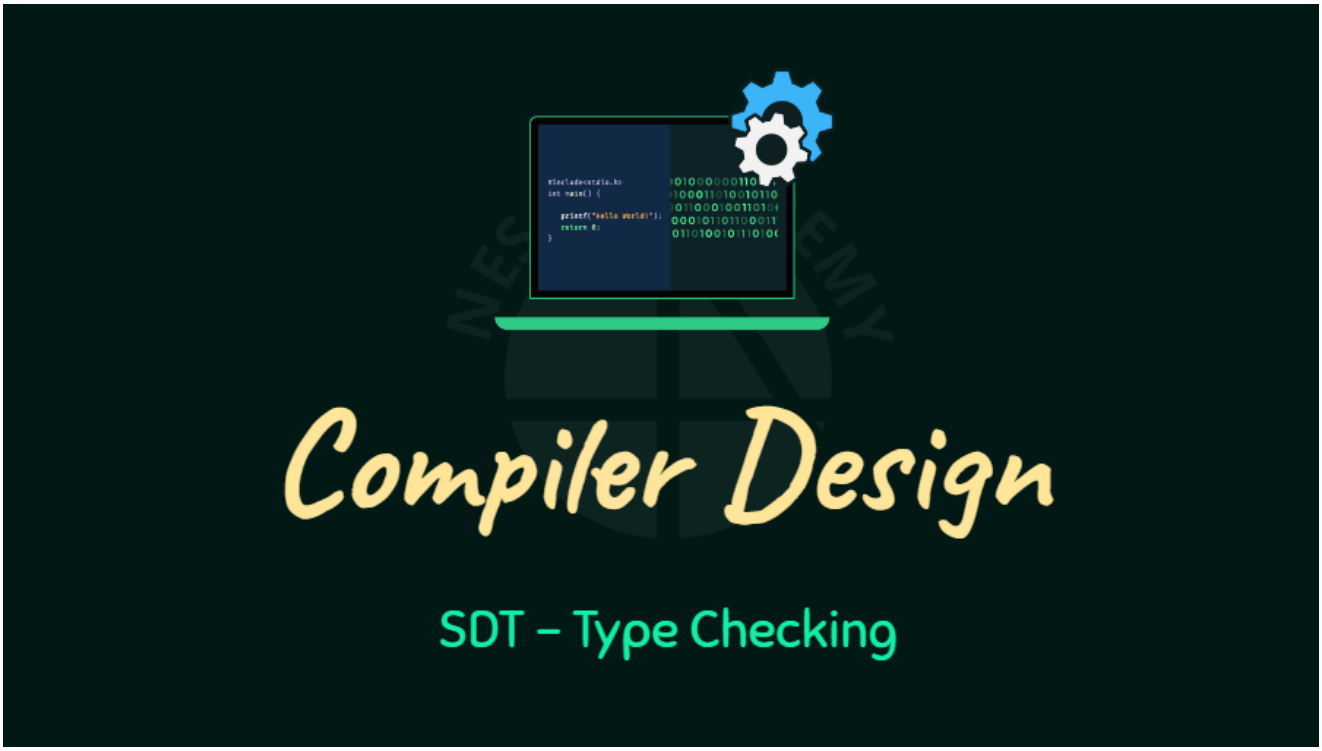
SDT - Abstract Syntax Tree:i.E → E + T ii.E → T iii.T → T x F iv.T → Fv.F → idGrammar E → E + TE → T T → T x F T → FF → id{E.nptr = mknode(E.nptr, '+', T.nptr);} Semantic Rules {F.nptr = mknode(, id.value, null);} {T.nptr = mknode(T.nptr, 'x', F.nptr);} {E.nptr = T.nptr;} {T.nptr = F.nptr;}



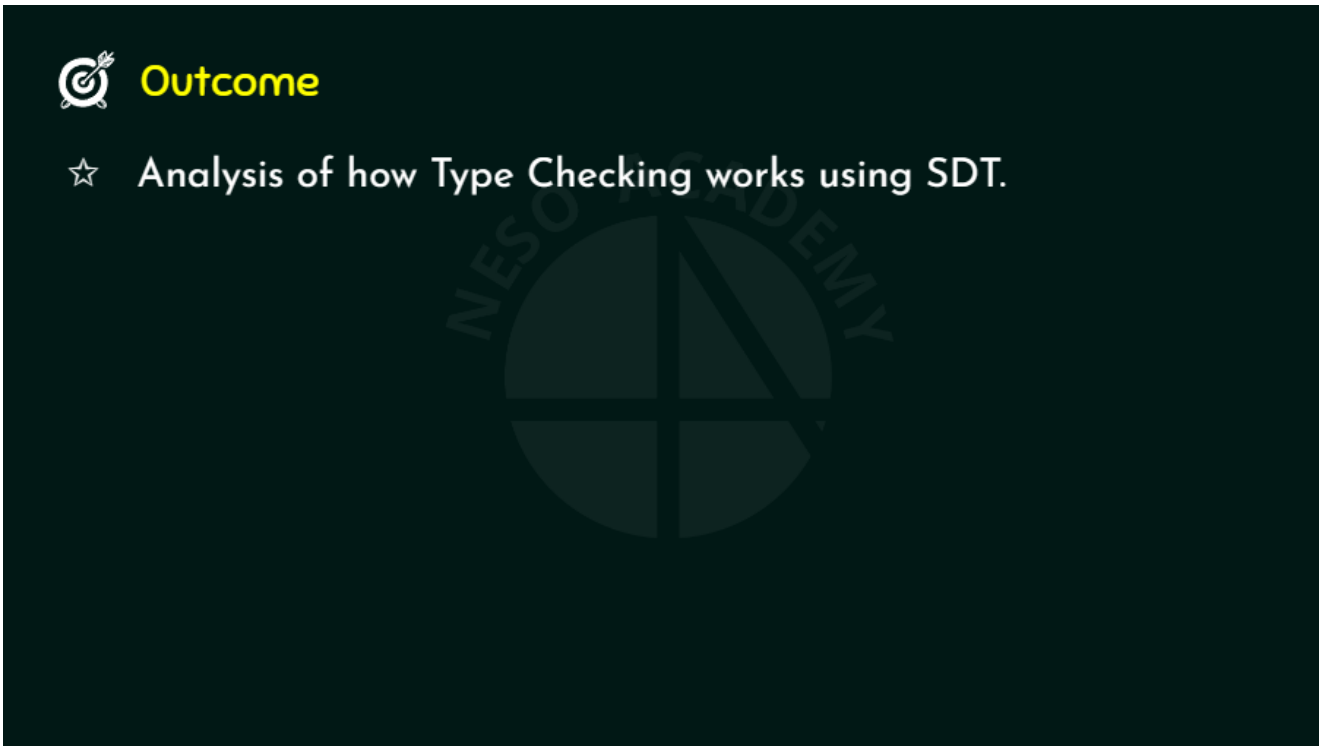
8. Ri:E.nptr=5002003. Rii:E.nptr = T.nptrSDT - Abstract Syntax Tree:Expression:2 + 3 x 4 ETFidFididTxFRiiRvRiiiRiE+TRivRvRivRv234Actions:1. Rv:F.nptr=1007. Riii:T.nptr=4004. Rv:F.nptr=2006. Rv:F.nptr=3002. Riv:T.nptr = F.nptrT.nptr=100E.nptr=1005. Riv:T.nptr = F.nptrT.nptr=200\02\0100\04\0300100+400500\03\0200x300400



Need of Abstract Syntax Tree:100+400500\02\0100200x300400\03\0200\04\03001.Popular intermediate code representation.2.Parsing and intermediate code generation are done simultaneously.



Compiler Design SDT - Type Checking



Outcome☆Analysis of how Type Checking works using SDT.

SDT – Type Checking:

	Grammar
i.	$E \rightarrow E_1 + E_2$
ii.	$E \rightarrow E_1 == E_2$
iii.	$E \rightarrow (E_1)$
iv.	$E \rightarrow num$
v.	$E \rightarrow True$
vi.	$E \rightarrow False$

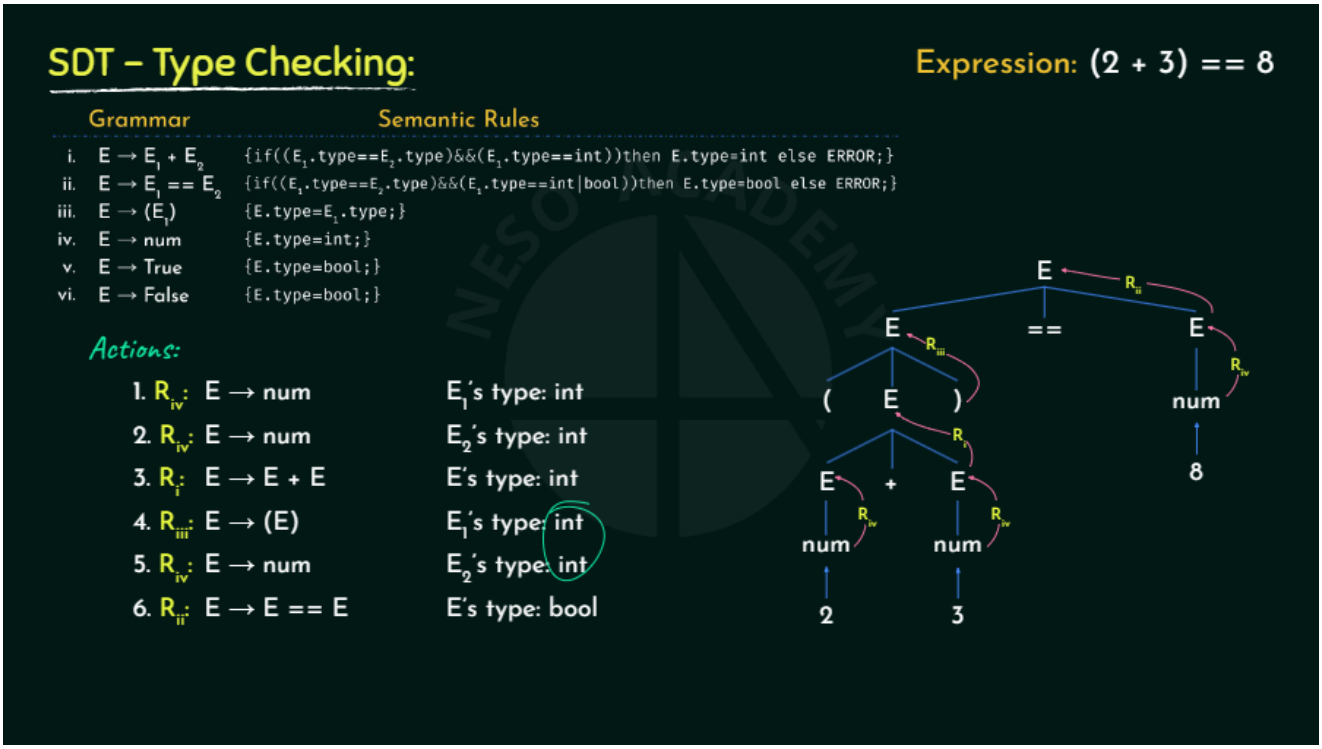
SDT - Type Checking:i.E → E + T ii.E → T iii.T → T x F iv.T → Fv.F → idvi.Grammar E → E1 + E2E → E1 == E2E → (E1) E → numE → TrueE → False

SDT – Type Checking:

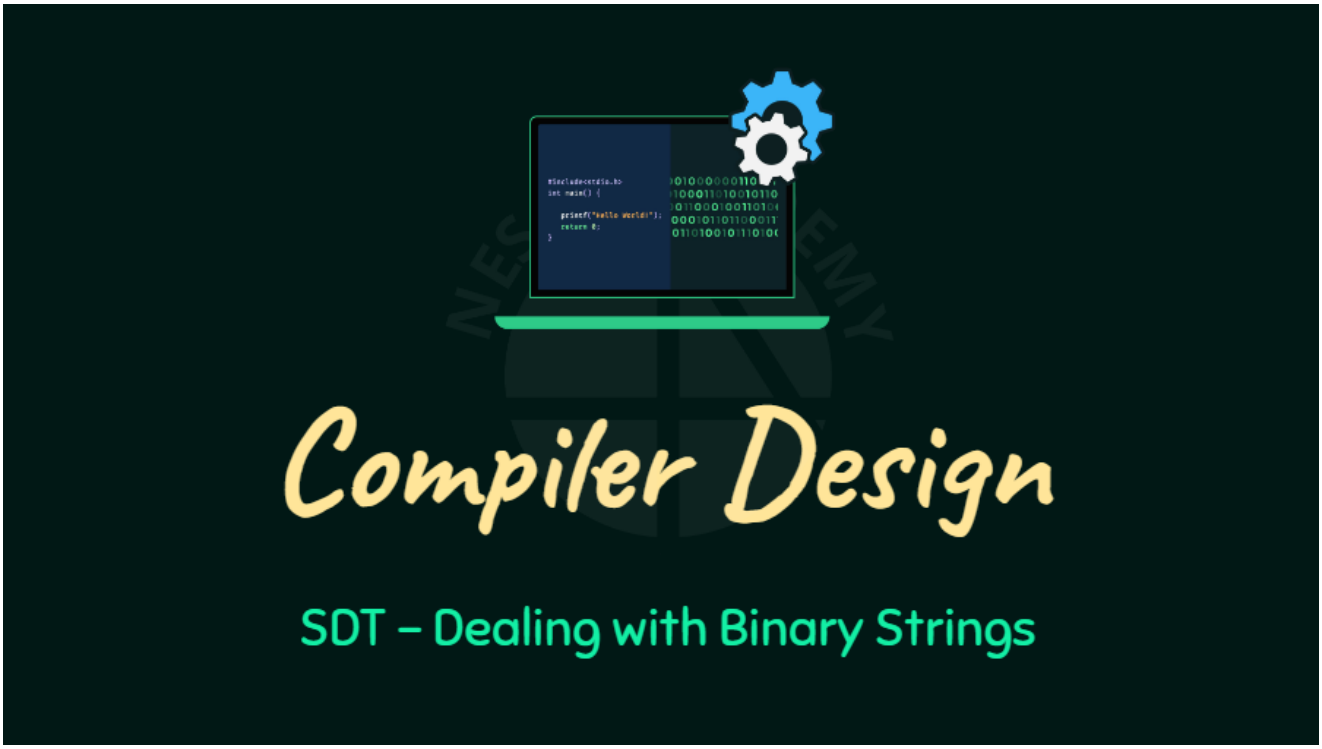
	Grammar	Semantic Rules
i.	$E \rightarrow E_1 + E_2$	{if((E_1.type==E_2.type)&&(E_1.type==int))then E.type=int else ERROR;}
ii.	$E \rightarrow E_1 == E_2$	{if((E_1.type==E_2.type)&&(E_1.type==int|bool))then E.type=bool else ERROR;}
iii.	$E \rightarrow (E_1)$	{E.type=E_1.type;}
iv.	$E \rightarrow num$	{E.type=int;}
v.	$E \rightarrow True$	{E.type=bool;}
vi.	$E \rightarrow False$	{E.type=bool;}

SDT - Type Checking:i.E → E + T ii.E → T iii.T → T x F iv.T → Fv.F → idvi.Grammar E → E1 + E2E → E1 == E2E → (E1) E → numE → TrueSemantic Rules E → False{if((E1.type==E2.type)&&(E1.type==int))then E.type=int else ERROR;}

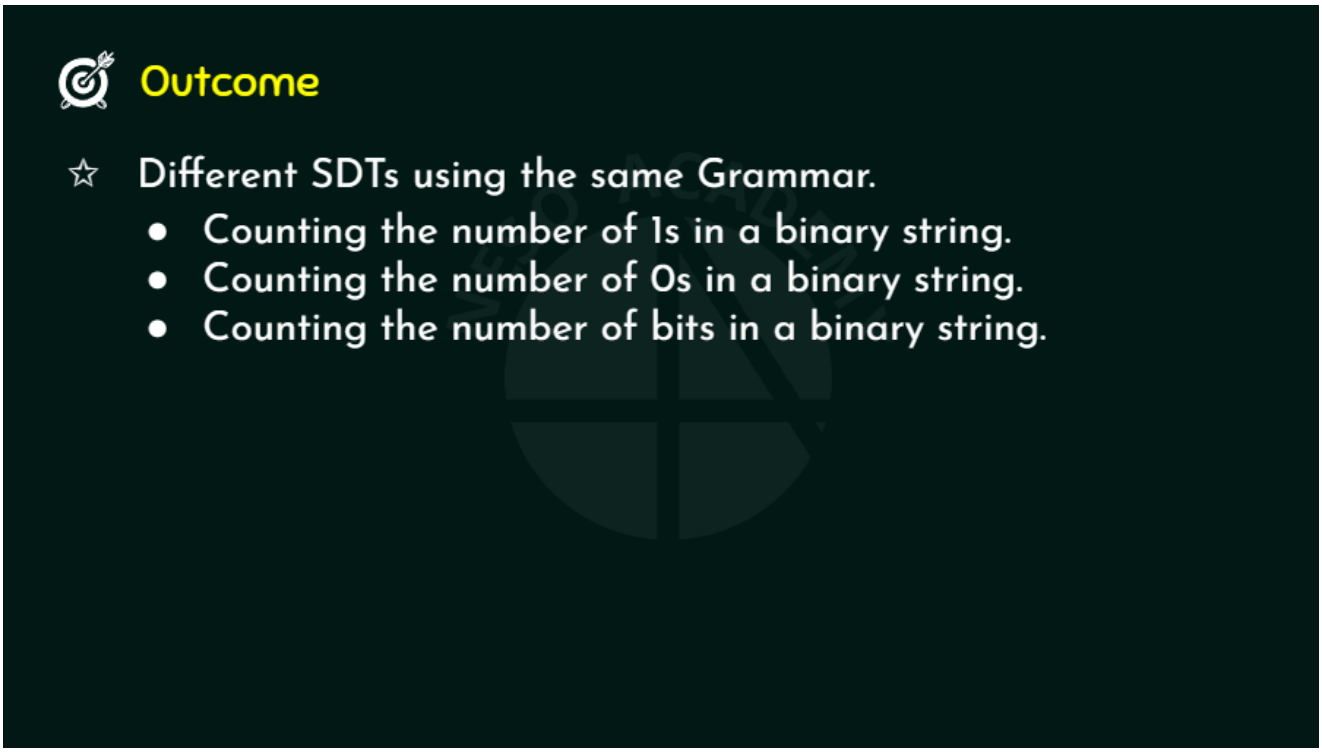
{if((E1.type==E2.type)&&(E1.type==int|bool))then E.type=bool else ERROR;} {E.type=E1.type;} {E.type=int;} {E.type=bool;} {E.type=bool;}



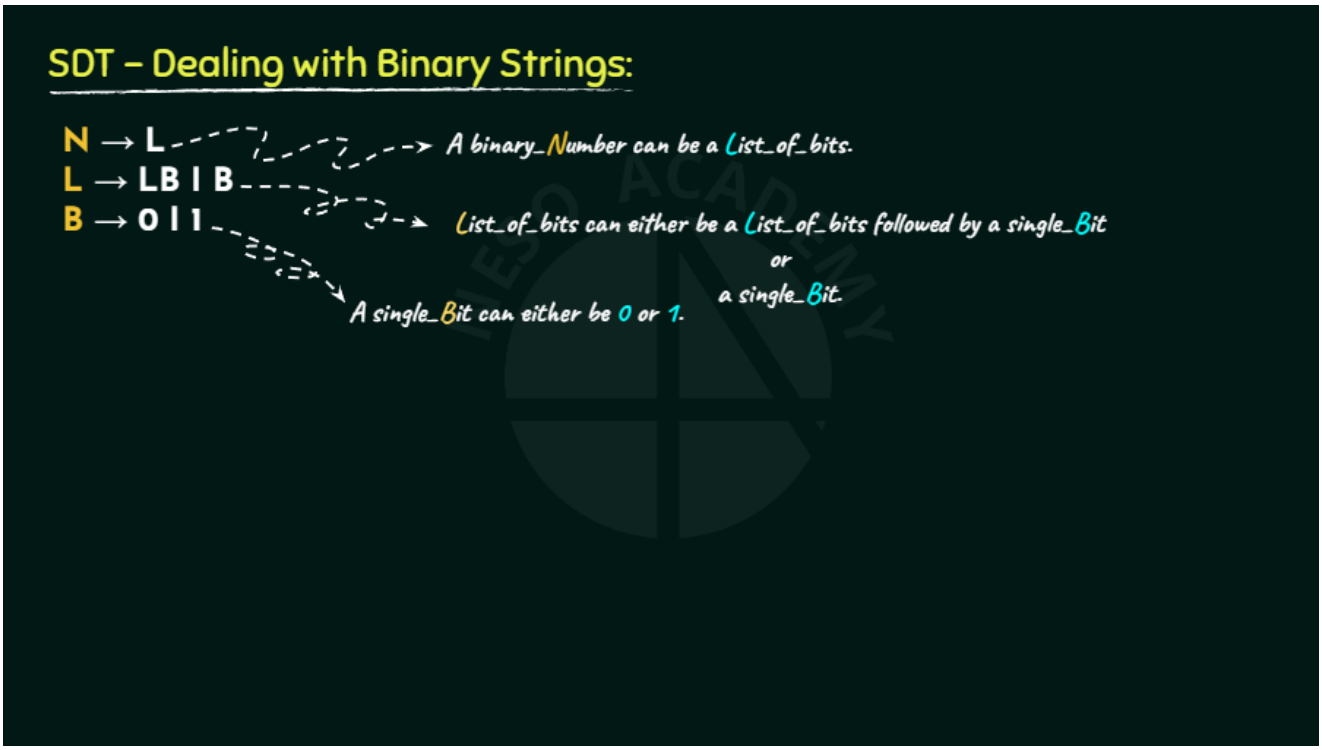
SDT - Type Checking:Expression:(2 + 3) == 8 E(E)E==E238E+Enumnumnum3. Ri:E → E + EActions:1. Riv:E → num4. Riii:E → (E)6. Rii:E → E == E2. Riv:E → num5. Riv:E → numRivRivRiRiiiRivRiiE1's type: int E2's type: int E's type: int E1's type: int E2's type: int E's type: bool



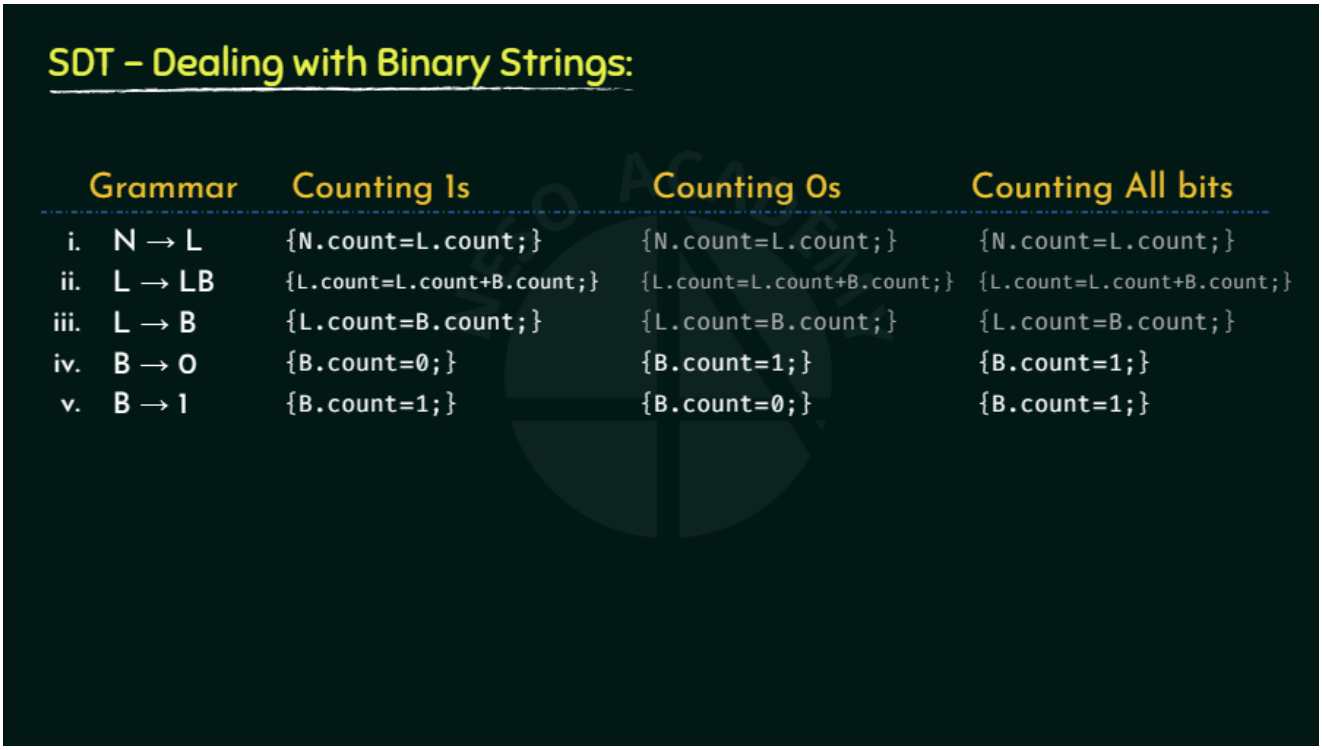
Compiler Design SDT - Dealing with Binary Strings



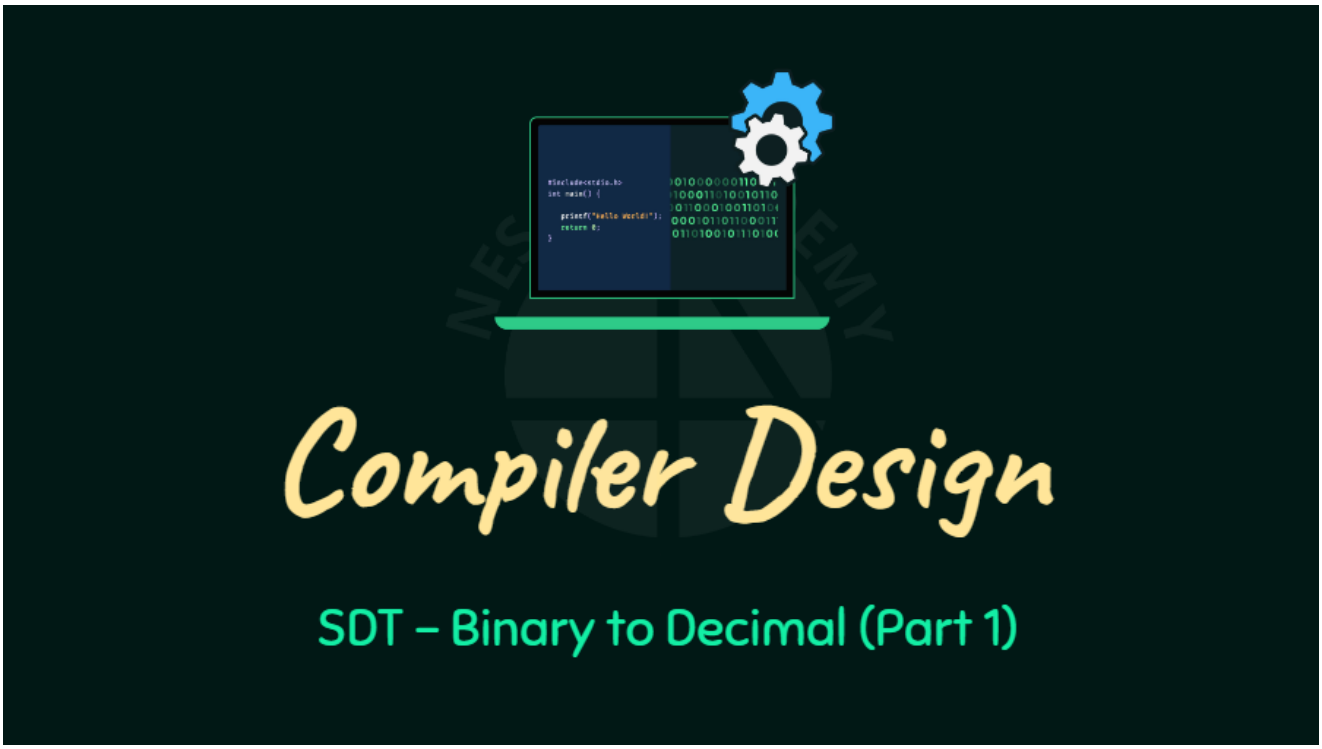
Outcome☆Different SDTs using the same Grammar.●Counting the number of 1s in a binary string.●Counting the number of 0s in a binary string.●Counting the number of bits in a binary string.



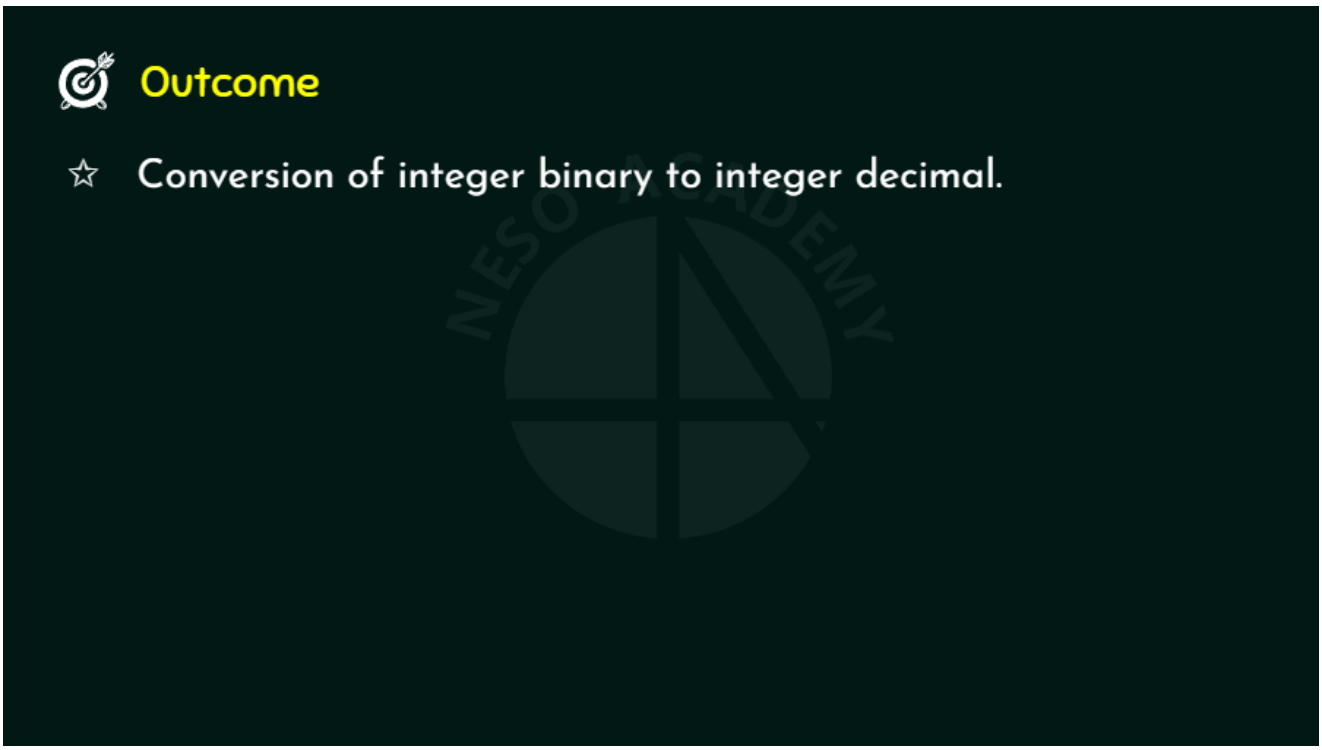
SDT - Dealing with Binary Strings:N → LL → LB I BB → 0 I 1A binary_Number can be a List_of_bits.List_of_bits can either be a List_of_bits followed by a single_Bitora single_Bit.A single_Bit can either be 0 or 1.



SDT - Dealing with Binary Strings:i.ii.iii.iv.v.Grammar N → LL → LBL → BB → 0B → 1Counting 1s{B.count=0;} {B.count=1;} {N.count=L.count;} {L.count=B.count;} {L.count=L.count+B.count;} Counting 0s{B.count=1;} {B.count=0;} {N.count=L.count;} {L.count=B.count;} {L.count=L.count+B.count;} Counting All bits{B.count=1;} {B.count=1;} {N.count=L.count;} {L.count=B.count;} {L.count=L.count+B.count;}



Compiler Design SDT - Binary to Decimal (Part 1)



Outcome☆Conversion of integer binary to integer decimal.

SDT – Dealing with Binary Strings:

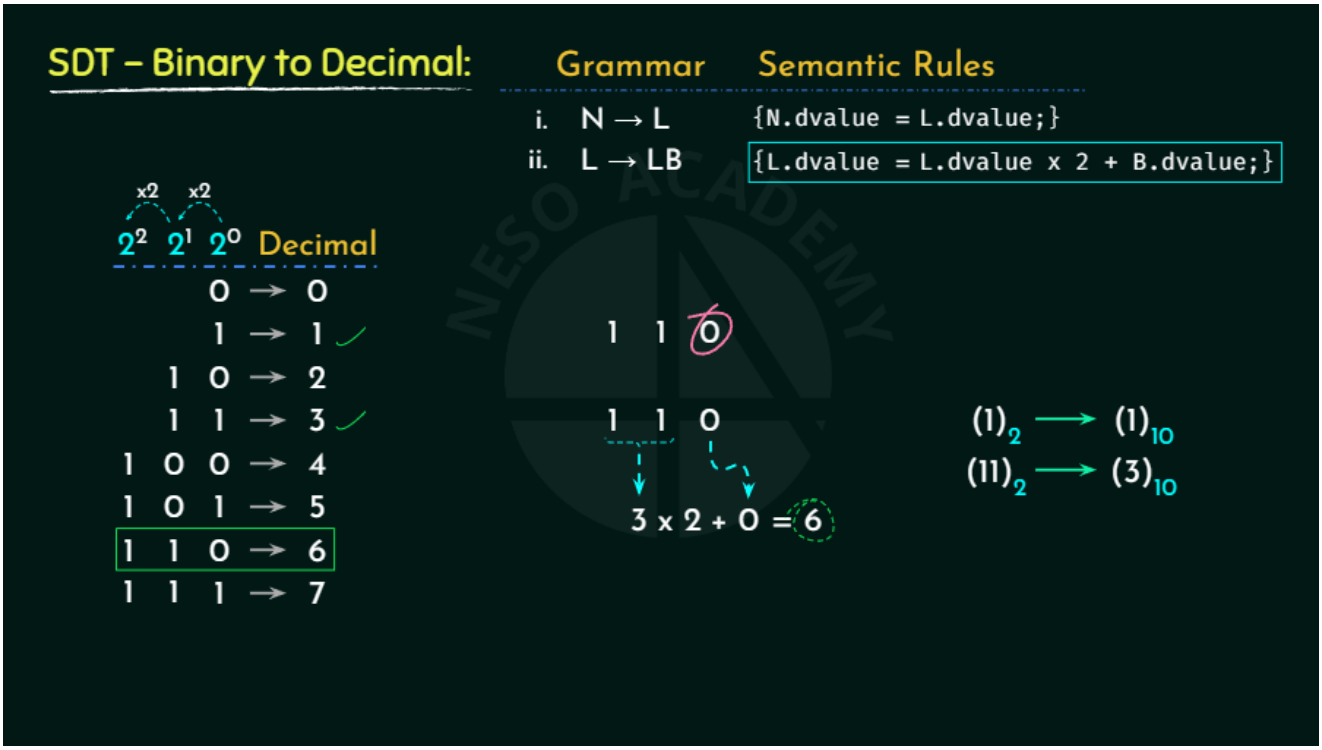
	Grammar	Counting 1s	Counting 0s	Counting All bits
i.	N → L	{N.count=L.count;}	{N.count=L.count;}	{N.count=L.count;}
ii.	L → LB	{L.count=L.count+B.count;}	{L.count=L.count+B.count;}	{L.count=L.count+B.count;}
iii.	L → B	{L.count=B.count;}	{L.count=B.count;}	{L.count=B.count;}
iv.	B → 0	{B.count=0;}	{B.count=1;}	{B.count=1;}
v.	B → 1	{B.count=1;}	{B.count=0;}	{B.count=1;}

SDT - Dealing with Binary Strings:i.ii.iii.iv.v.Grammar N → LL → LBL → BB → 0B → 1Counting 1s{B.count=0;} {B.count=1;} {N.count=L.count;} {L.count=B.count;} {L.count=L.count+B.count;} Counting 0s{B.count=1;} {B.count=0;} {N.count=L.count;} {L.count=B.count;} {L.count=L.count+B.count;} Counting All bits{B.count=1;} {B.count=1;} {N.count=L.count;} {L.count=B.count;} {L.count=L.count+B.count;}

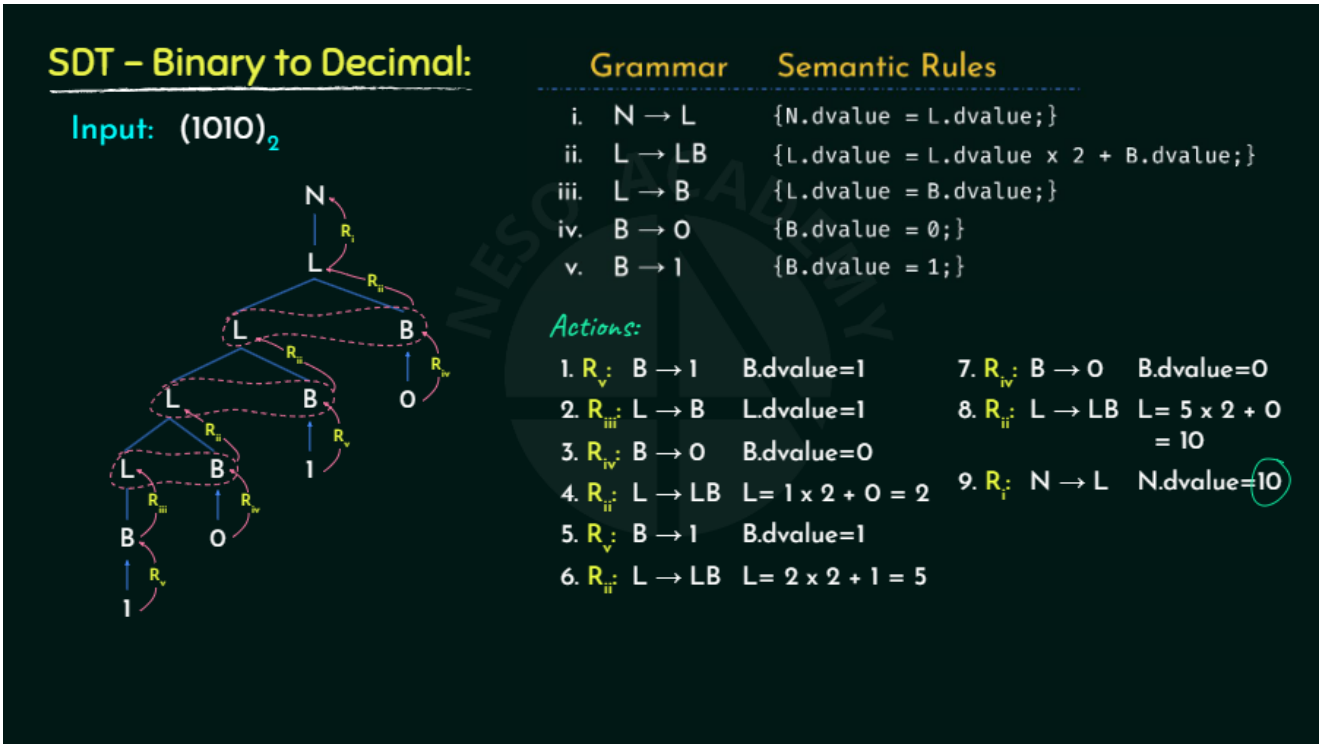
SDT – Binary to Decimal:

	Grammar	Semantic Rules
i.	N → L	{N.dvalue = L.dvalue;}
ii.	L → LB	{L.dvalue = L.dvalue x 2 + B.dvalue;}
iii.	L → B	{L.dvalue = B.dvalue;}
iv.	B → 0	{B.dvalue = 0;}
v.	B → 1	{B.dvalue = 1;}

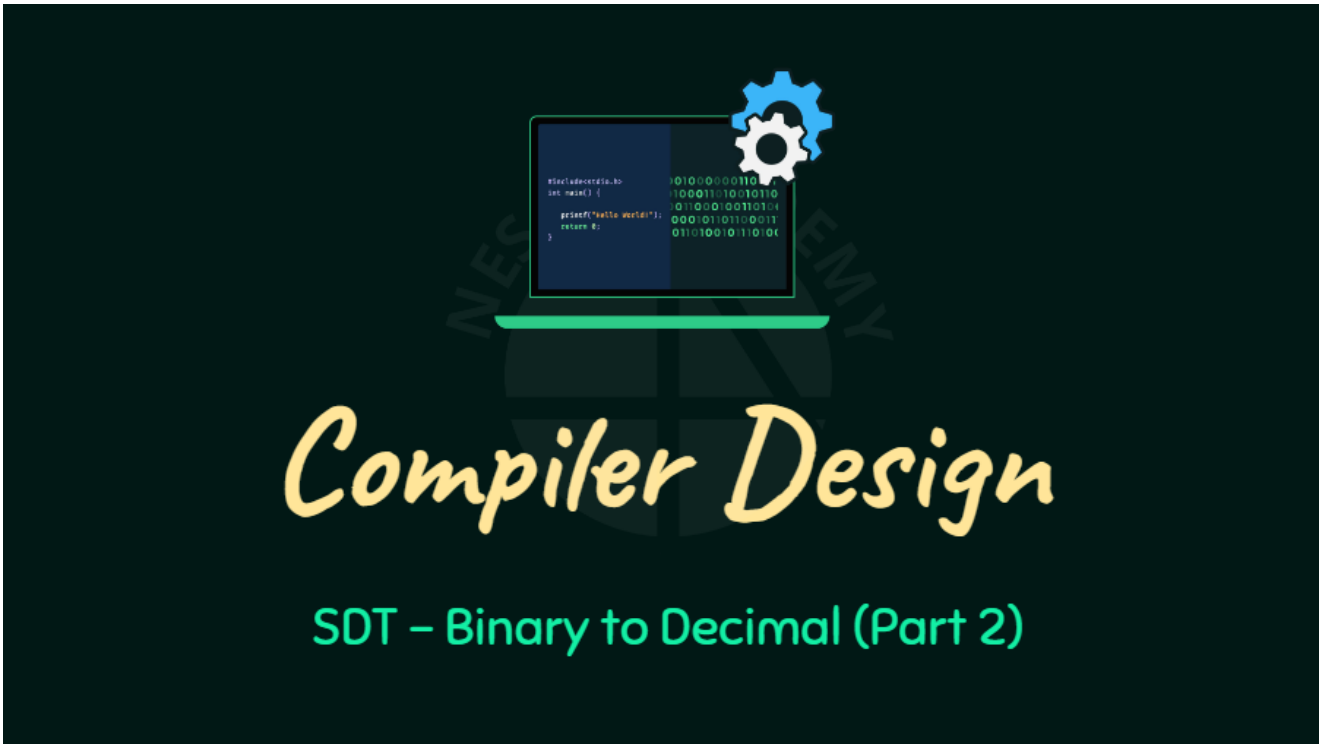
SDT - Binary to Decimal:i.ii.iii.iv.v.Grammar N → LL → LBL → BB → 0B → 1Semantic Rules{B.dvalue = 0;} {B.dvalue = 1;} {N.dvalue = L.dvalue;} {L.dvalue = B.dvalue;} {L.dvalue = L.dvalue x 2 + B.dvalue;}



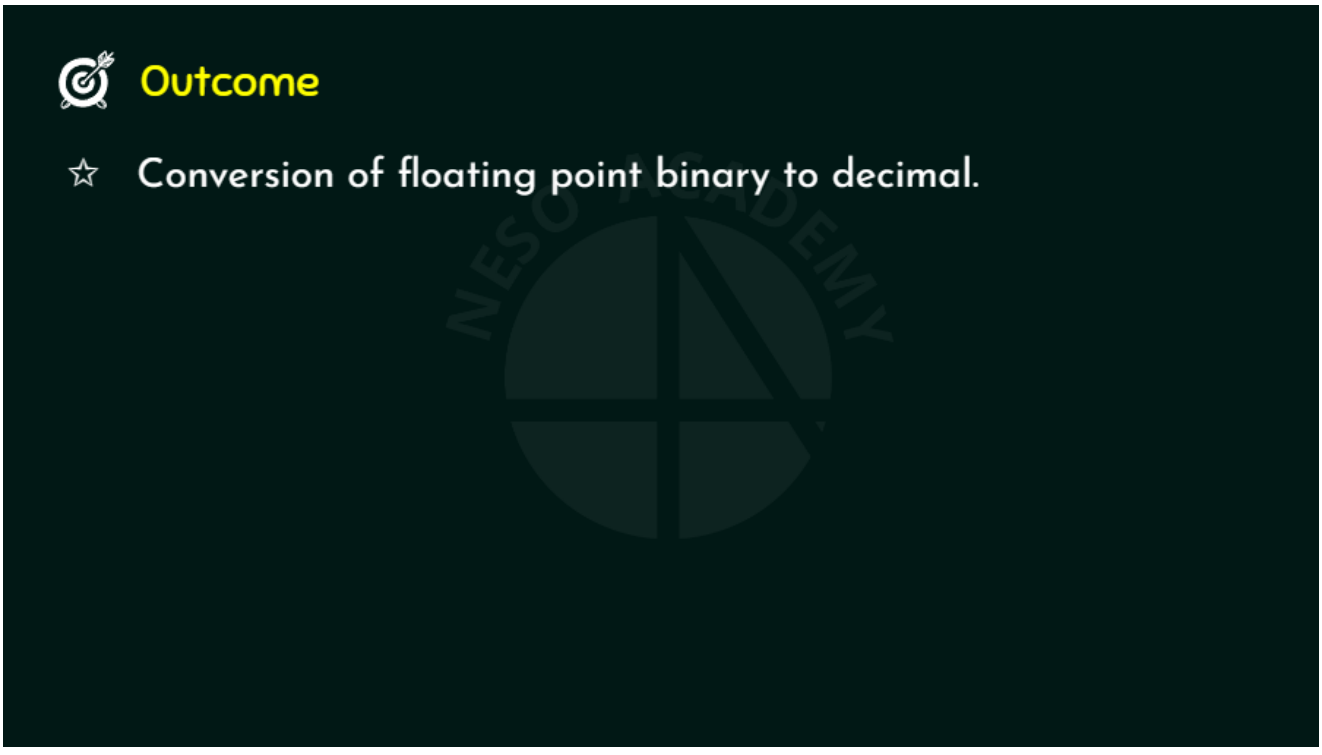
SDT - Binary to Decimal:i.ii.Grammar N → LL → LBSemantic Rules{N.dvalue = L.dvalue;} {L.dvalue = L.dvalue x 2 + B.dvalue;} 0101010111001111110123456720212 2Decimal0110(1)2(1)10113 x 2+ 0= 6(11)2(3)10x2x2



+ 0L== 2SDT - Binary to Decimal:NLLBLB10LBB01Input:(1010)2RvRiiiRivRiiRvRiiRivRiiRi3. Riv:B → 0Actions:1. Rv:B → 14. Rii:L → LB6. Rii:L → LB2. Riii:L → B5. Rv:B → 1B.dvalue=1 L.dvalue=1 B.dvalue=0B.dvalue=17. Riv:B → 08. Rii:L → LB9. Ri:N → LB.dvalue=0 N.dvalue=10 L=1 x 2 + 02x 2 + 1= 5L=5x 2 = 10



Compiler Design SDT - Binary to Decimal (Part 2)



Outcome☆Conversion of floating point binary to decimal.

SDT – Binary to Decimal (Integers):

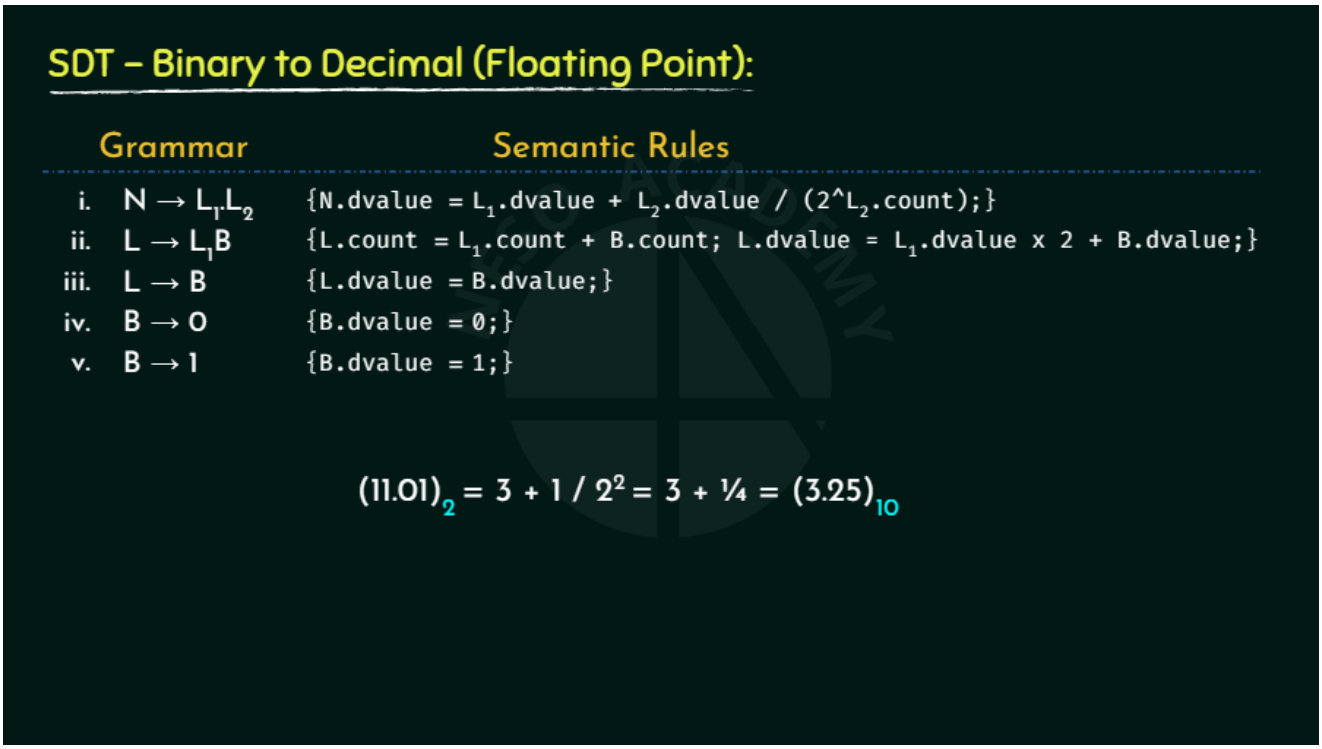
	Grammar	Semantic Rules
i.	N → L	{N.dvalue = L.dvalue;}
ii.	L → LB	{L.dvalue = L.dvalue x 2 + B.dvalue;}
iii.	L → B	{L.dvalue = B.dvalue;}
iv.	B → 0	{B.dvalue = 0;}
v.	B → 1	{B.dvalue = 1;}

SDT - Binary to Decimal (Integers):i.ii.iii.iv.v.Grammar N → LL → LBL → BB → 0B → 1Semantic Rules{B.dvalue = 0;} {B.dvalue = 1;} {N.dvalue = L.dvalue;} {L.dvalue = B.dvalue;} {L.dvalue = L.dvalue x 2 + B.dvalue;}

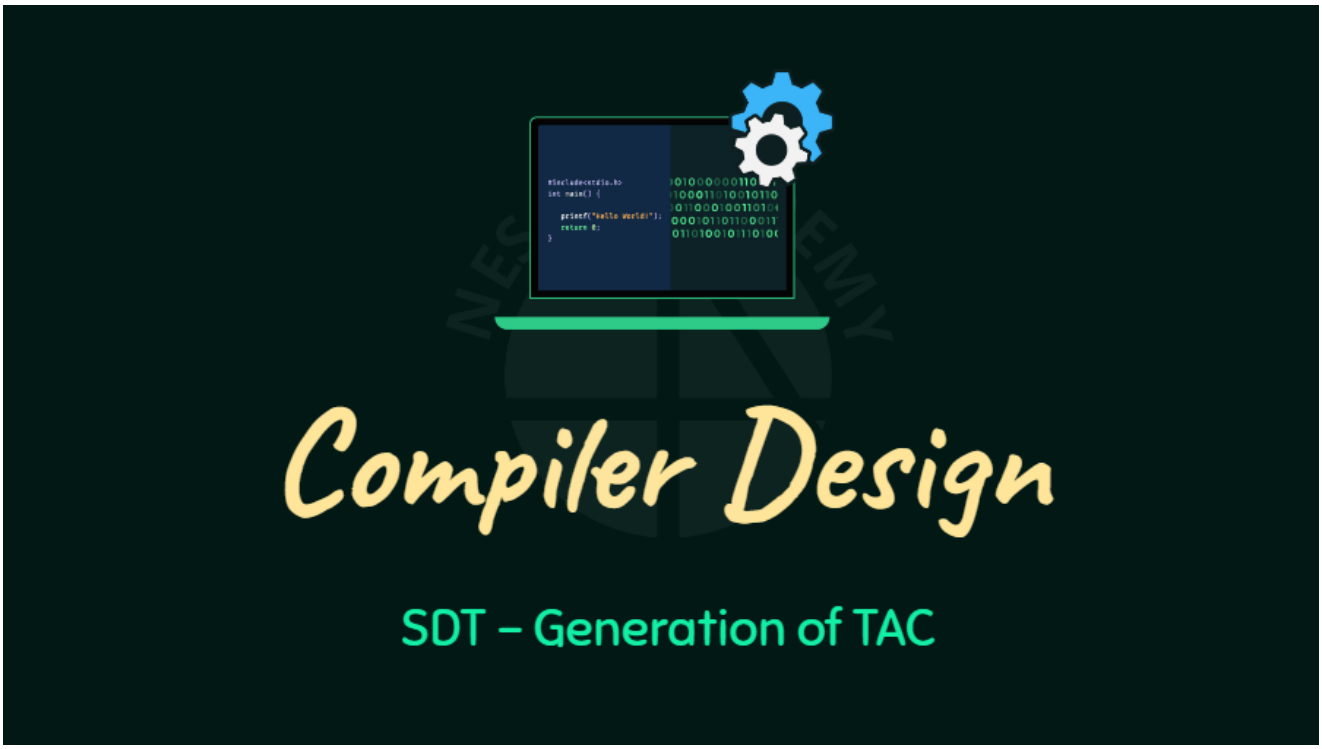
SDT – Binary to Decimal (Floating Point):

Procedure: $(0.011)_2$

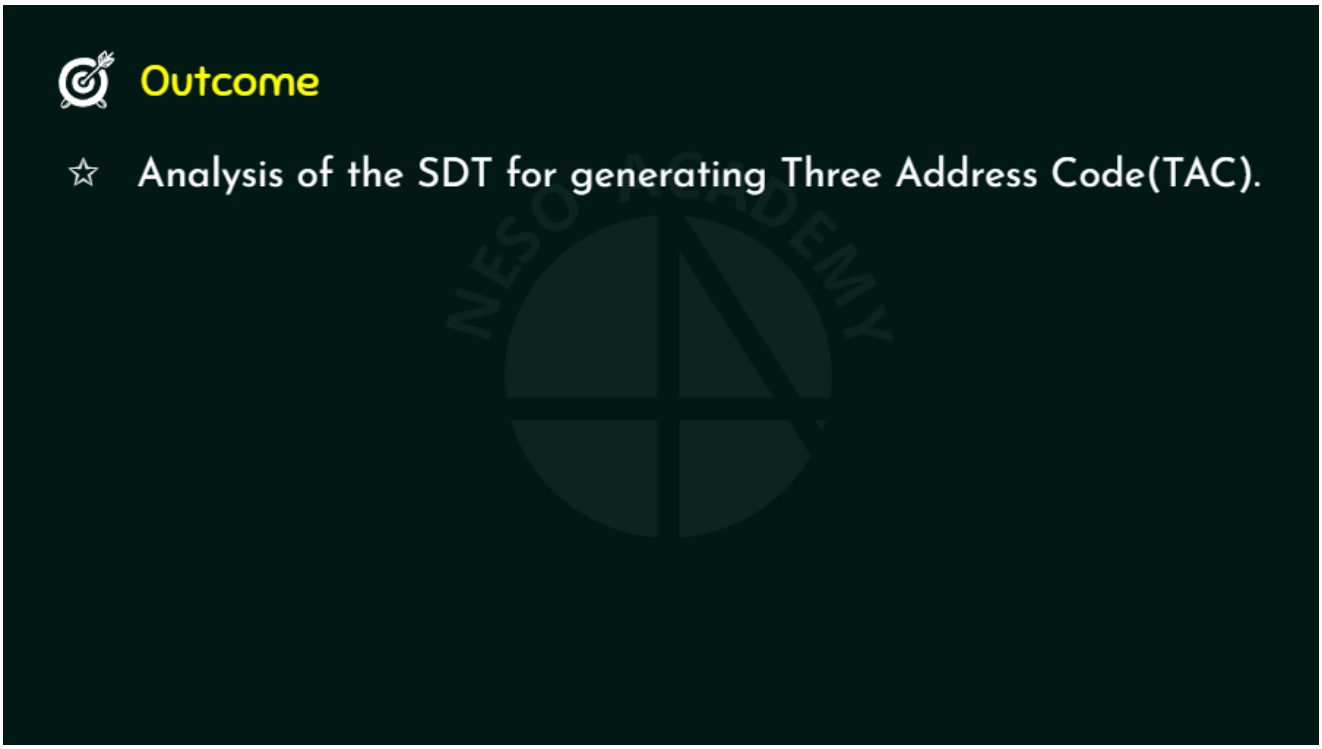
1. Evaluate the binary bit stream without considering the radix point.
   i.e. $(0.011)_2 \rightarrow (0011)_2 \rightarrow (3)_{10}$
2. Count the number of digits after the radix point (say, n) and divide the previously determined value by 2^n.
   i.e. $(0.011)_2$
   3 $\quad 3 / 2^3 = 3/8 = (0.375)_{10}$



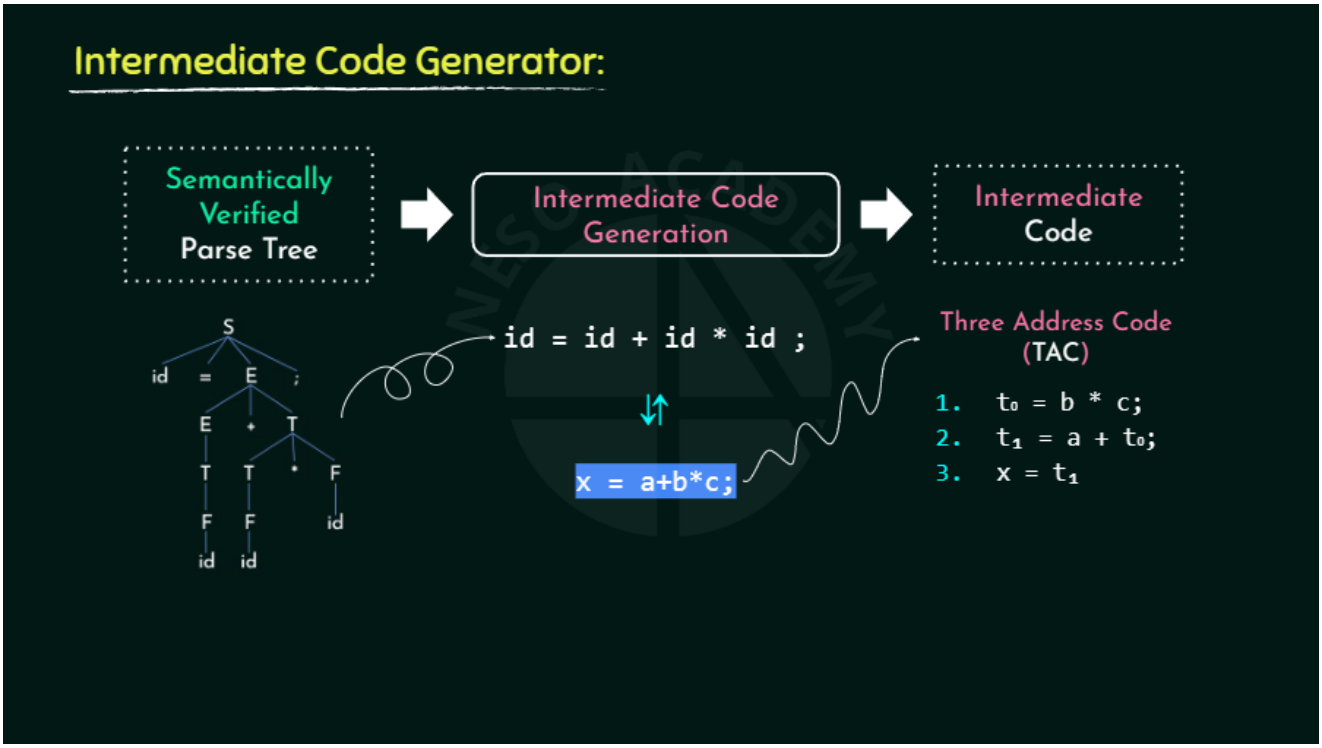
= (3.25)10 {N.dvalue = L1.dvalueSDT - Binary to Decimal (Floating Point):i.ii.iii.iv.v.Grammar N → L1.L2L → L1BL → BB → 0B → 1Semantic Rules{B.dvalue = 1;} {B.dvalue = 0;} {L.dvalue = B.dvalue;} {L.count = L1.count + B.count;L.dvalue = L1.dvalue x 2 + B.dvalue;} / (2^L2.count);} + L2.dvalue(11.01)2= 3 + 1 / 22= 3 + ¼



Compiler Design SDT - Generation of TAC



Outcome☆Analysis of the SDT for generating Three Address Code(TAC).



Intermediate Code Generator:Semantically VerifiedParse Tree Intermediate Code GenerationIntermediate Code x = a+b*c;id = id + id * id ;⇆Three Address Code (TAC)1.t_0 = b * c;2.t_1 = a + t_0;3.x = t_1

SDT – Generation of Three Address Code:

	Grammar	Semantic Rules
i.	$S \rightarrow id = E$	{gen(id.name = E.place);}
ii.	$E \rightarrow E_1 + T$	{E.place = newTemp(); gen(E.place = E_1.place + T.place);}
iii.	$E \rightarrow T$	{E.place = T.place;}
iv.	$T \rightarrow T_1 \times F$	{T.place = newTemp(); gen(T.place = T_1.place x F.place);}
v.	$T \rightarrow F$	{T.place = F.place;}
vi.	$F \rightarrow id$	{F.place = id.name;}

X.place:
It is a placeholder variable, i.e. the variable remembers the place of the R.H.S.

newTemp():
It is a function that creates temporary variables. E.g. t_1, t_2.

gen():
It is a function that generates an expression.

SDT - Generation of Three Address Code:{gen(id.name = E.place);} i.ii.iii.iv.v.Kvi.Grammar S → id = EE → E1 + TE → TT → T1 x FT → FSemantic Rules{T.place = newTemp(); gen(T.place = T1.place x F.place);}{T.place = F.place;} {E.place = T.place;} {E.place = newTemp(); gen(E.place = E1.place + T.place);} F → id{F.place = id.name;} X.place: It is a placeholder variable, i.e. the variable remembers the place of the R.H.S.newTemp(): It is a function that creates temporary variables. E.g. t1, t2.gen(): It is a function that generates an expression.

SDT – Generation of Three Address Code:

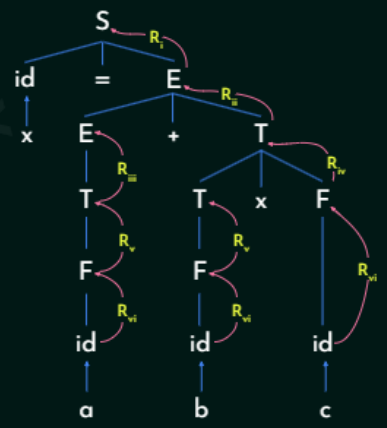
Expression: x = a + b x c

	Grammar	Semantic Rules
i.	$S \rightarrow id = E$	{gen(id.name = E.place);}
ii.	$E \rightarrow E_1 + T$	{E.place = newTemp(); gen(E.place = E_1.place + T.place);}
iii.	$E \rightarrow T$	{E.place = T.place;}
iv.	$T \rightarrow T_1 \times F$	{T.place = newTemp(); gen(T.place = T_1.place x F.place);}
v.	$T \rightarrow F$	{T.place = F.place;}
vi.	$F \rightarrow id$	{F.place = id.name;}

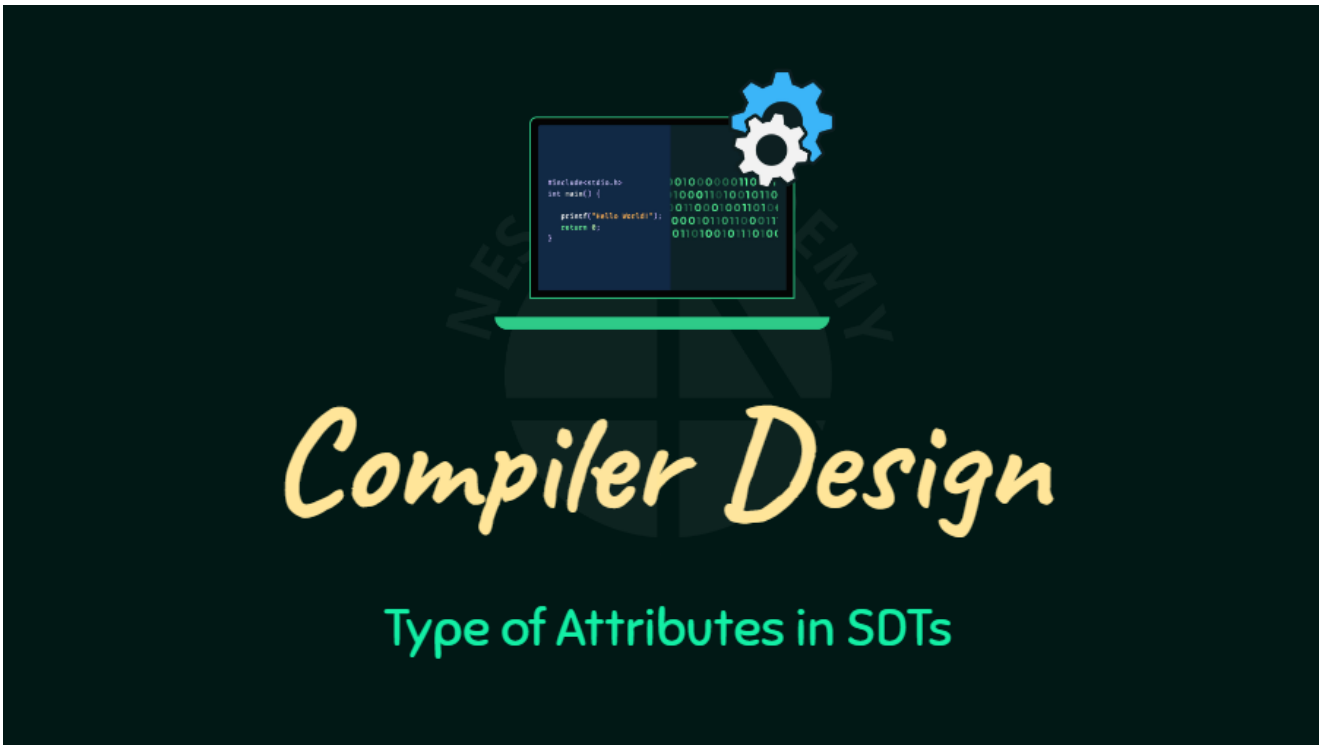
Actions:

1. R_{vi}: $F \rightarrow id$ F.place = a
2. R_{v}: $T \rightarrow F$ T.place = a
3. R_{iii}: $E \rightarrow T$ E.place = a
4. R_{vi}: $F \rightarrow id$ F.place = b
5. R_{v}: $T \rightarrow F$ T.place = b
6. R_{vi}: $F \rightarrow id$ F.place = c
7. R_{iv}: $T \rightarrow T_1 \times F$ T.place = t_1
8. R_{ii}: $E \rightarrow E_1 + T$ E.place = t_2
9. R_{i}: $S \rightarrow id = E$

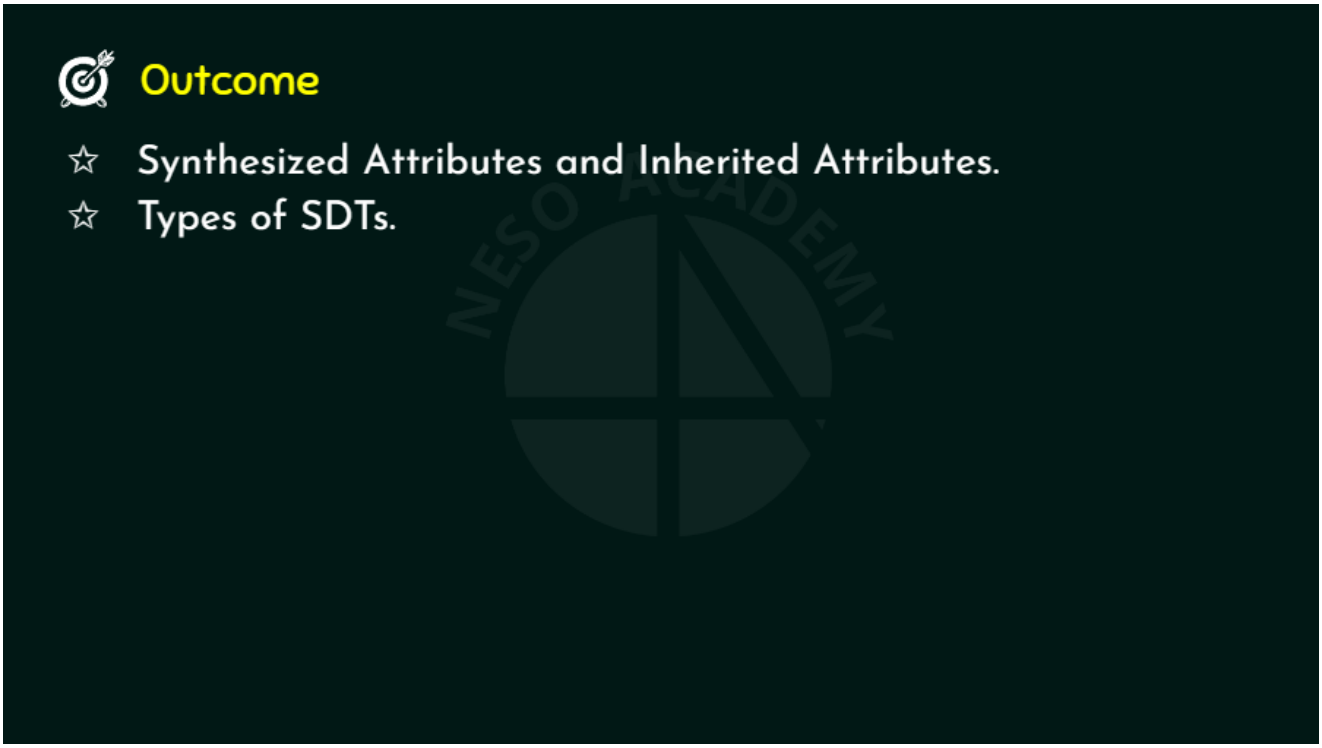
$t_1 = b \times c$
$t_2 = a + t_1$
$x = t_2$



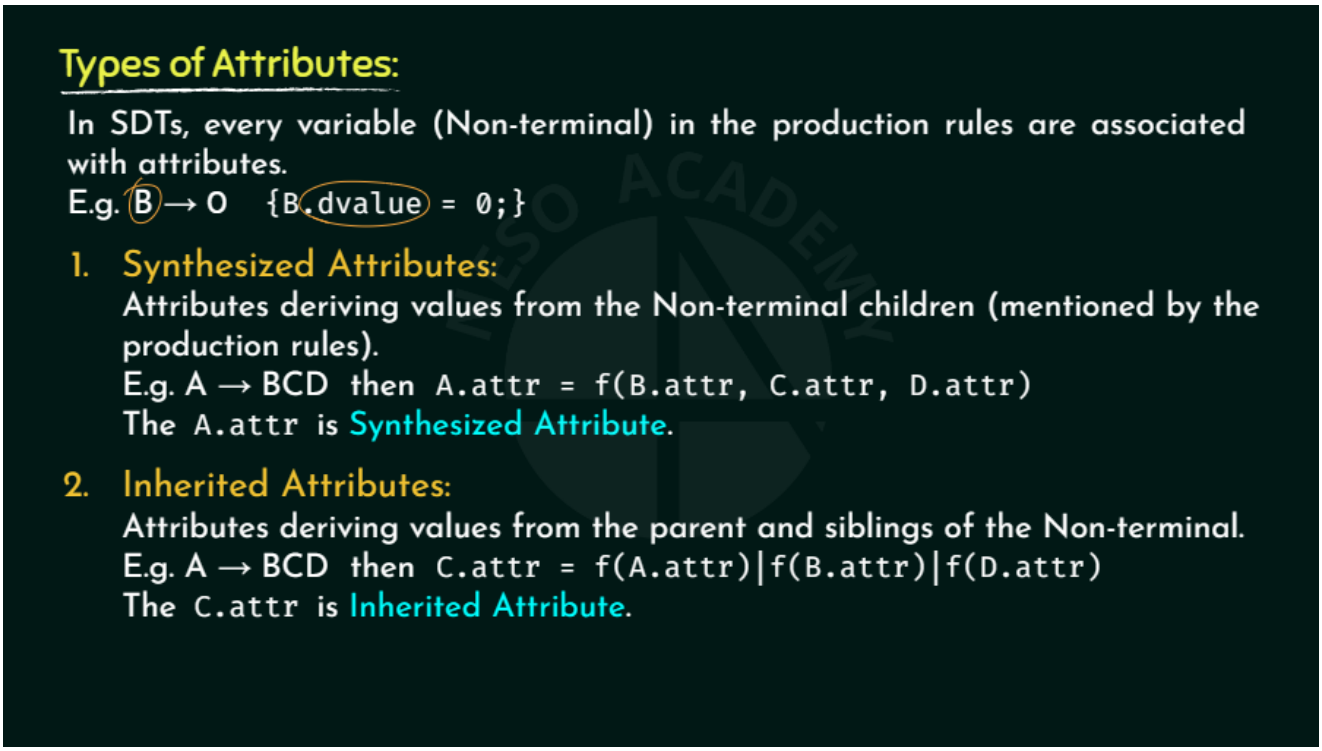
t1 SDT - Generation of Three Address Code:TFidFididTxFRiiiRviRivRiiE+TRvRviRvRviabcSE=idxRiF.place = c3. Riii:E → TActions:1. Rvi:F → id4. Rvi:F → id6. Rvi:F → id2. Rv:T → F5. Rv:T → FF.place = a T.place = a E.place = aT.place = b7. Riv:T → T1 x F8. Rii:E → E1 + T9. Ri:S → id = EF.place = bE.place = t2T.place = t1 = b x c+ t1t2 = ax= t2Expression:x = a + b x c



Compiler Design Type of Attributes in SDTs

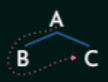


Outcome☆Synthesized Attributes and Inherited Attributes.☆Types of SDTs.

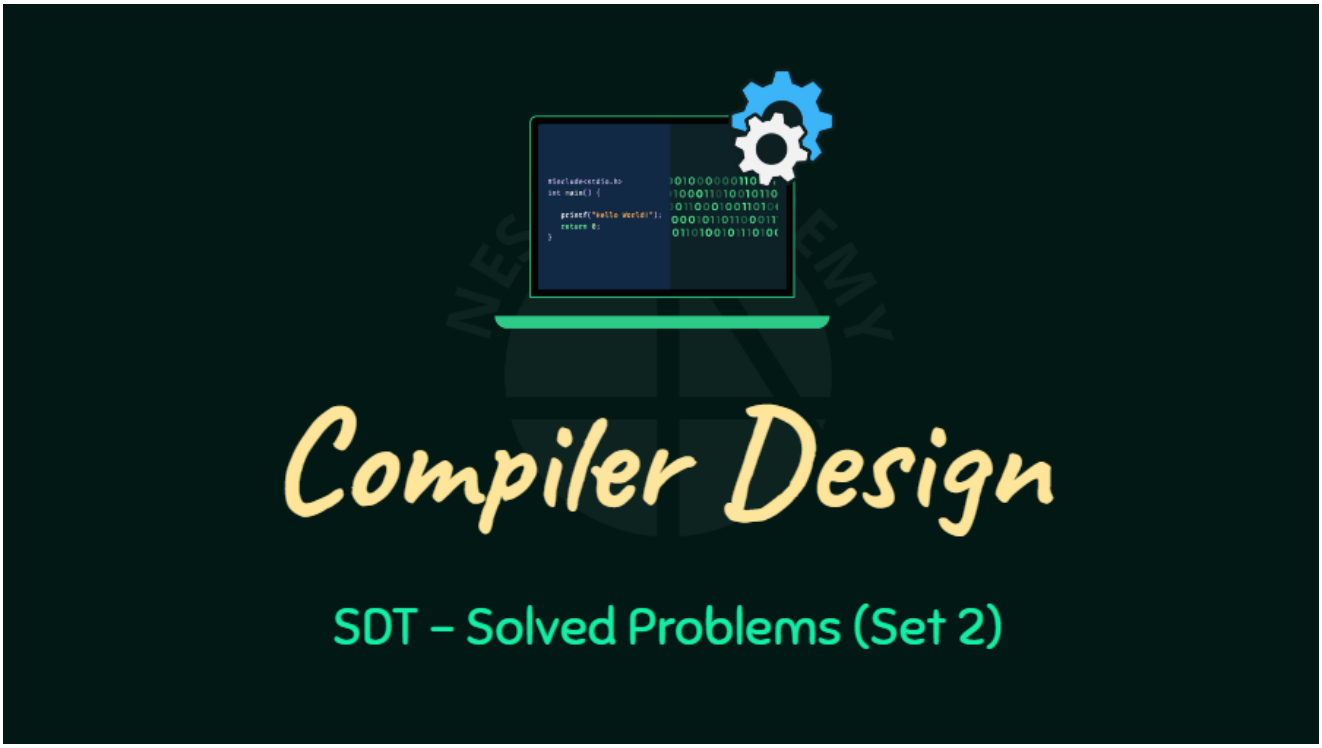


Types of Attributes:In SDTs, every variable (Non-terminal) in the production rules are associated with attributes. E.g.B → 0 {B.dvalue = 0;} 1.Synthesized Attributes:Attributes deriving values from the Non-terminal children (mentioned by the production rules).E.g. A → BCD then A.attr = f(B.attr, C.attr, D.attr)The A.attr is Synthesized Attribute.2.Inherited Attributes:Attributes deriving values from the parent and siblings of the Non-terminal.E.g. A → BCD then C.attr = f(A.attr)|f(B.attr)|f(D.attr)The C.attr is Inherited Attribute.

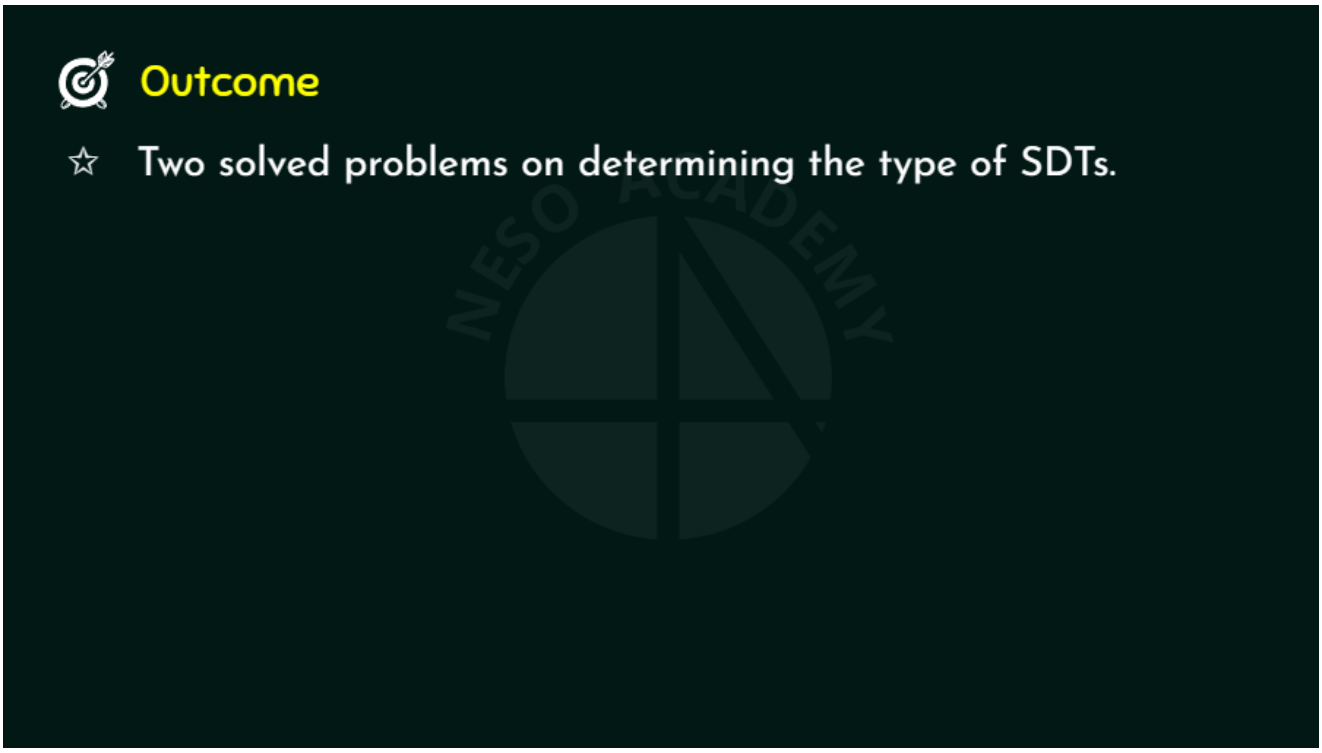
Types SDTs:

S-attributed SDT	L-attributed SDT
i. Uses only synthesized attributes.	i. Uses both synthesized and inherited attributes. Each inherited attribute is restricted to inherit either from parent or left sibling(s) only. e.g. A → XYZ {Y.attr = A.attr, Y.attr = X.attr, Y.attr ≠ Z.attr}
ii. Semantic actions are placed at the right most end of the productions. e.g. A → BC { }	ii. Semantic actions can be placed anywhere in the R.H.S. e.g. A → { }BC | D{ }E | FG{ }
iii. Attributes are evaluated during Bottom-up parsing.	iii. Attributes are evaluated by traversing the parse tree depth first, left to right. 

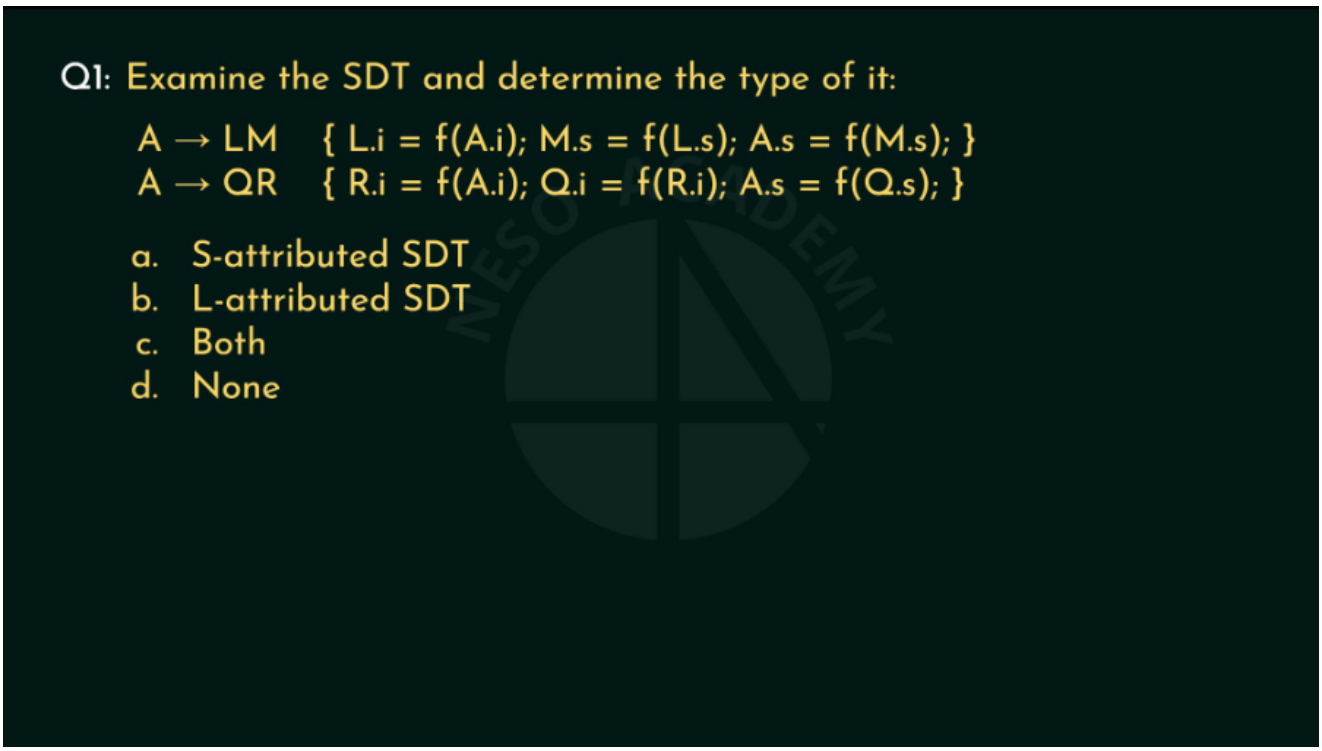
L-attributed SDTS-attributed SDTTypes SDTs:iii. Attributes are evaluated by traversing the parse tree depth first, left to right.e.g. A → XYZ{Y.attr = A.attr, Y.attr = X.attr, Y.attr ≠ Z.attr}i. Uses both synthesized and inherited attributes. Each inherited attribute is restricted to inherit either from parent or left sibling(s) only.i. Uses only synthesized attributes.ii. Semantic actions can be placed anywhere in the R.H.S.e.g. A → { }BC | D{ }E | FG{ }ii. Semantic actions are placed at the right most end of the productions.e.g. A → BC { }iii. Attributes are evaluated during Bottom-up parsing.ABC

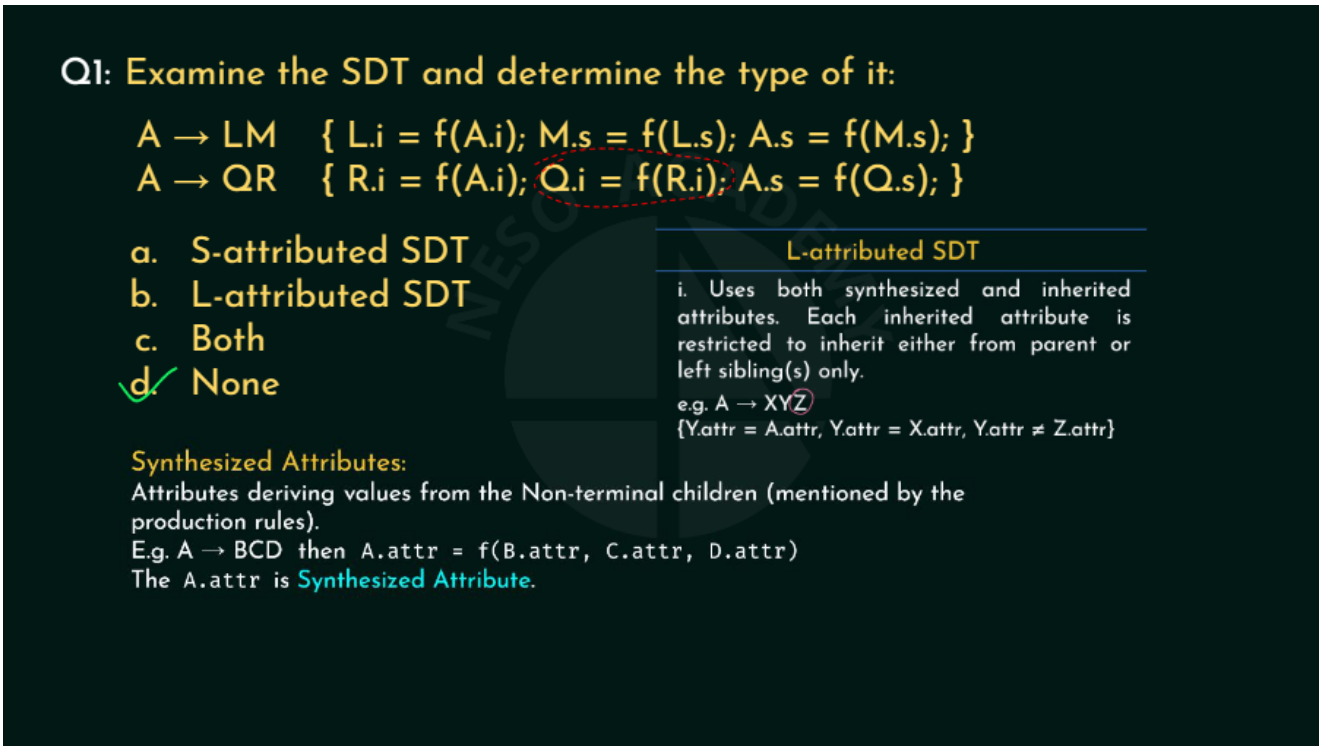


Compiler Design SDT - Solved Problems (Set 2)

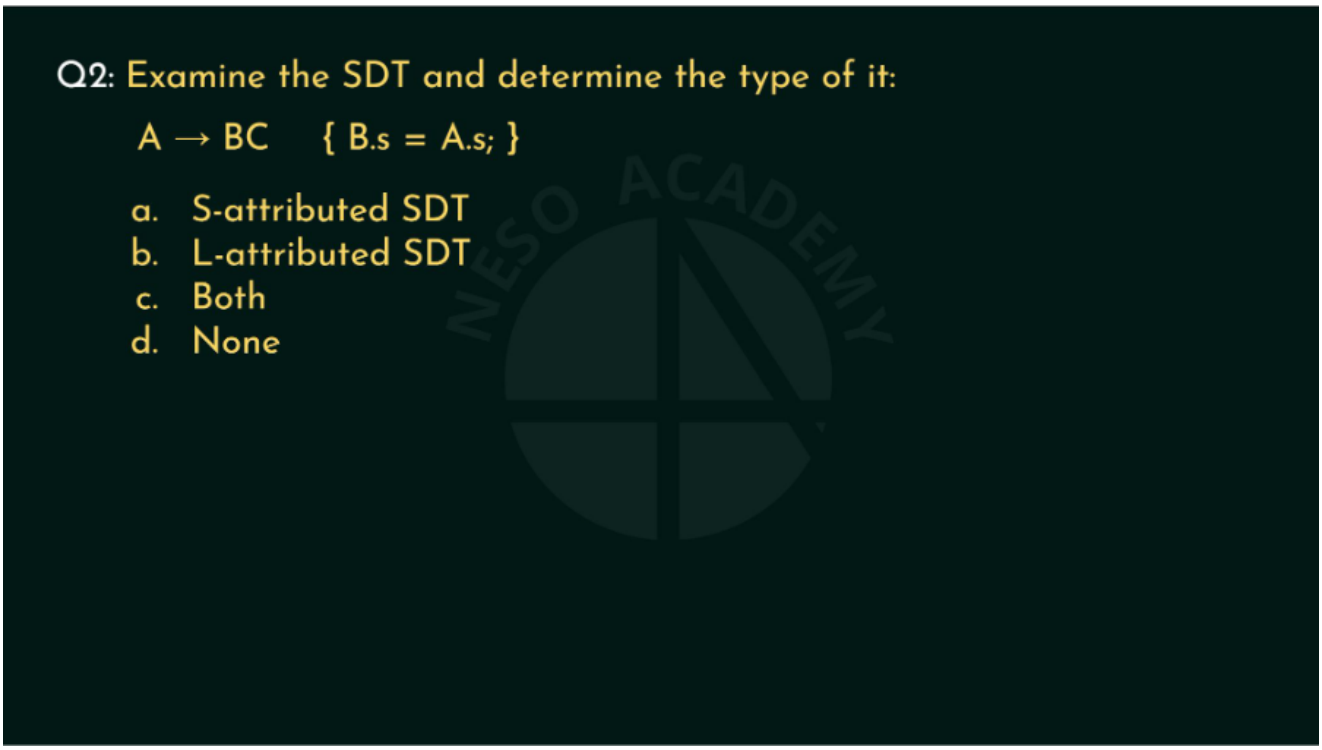


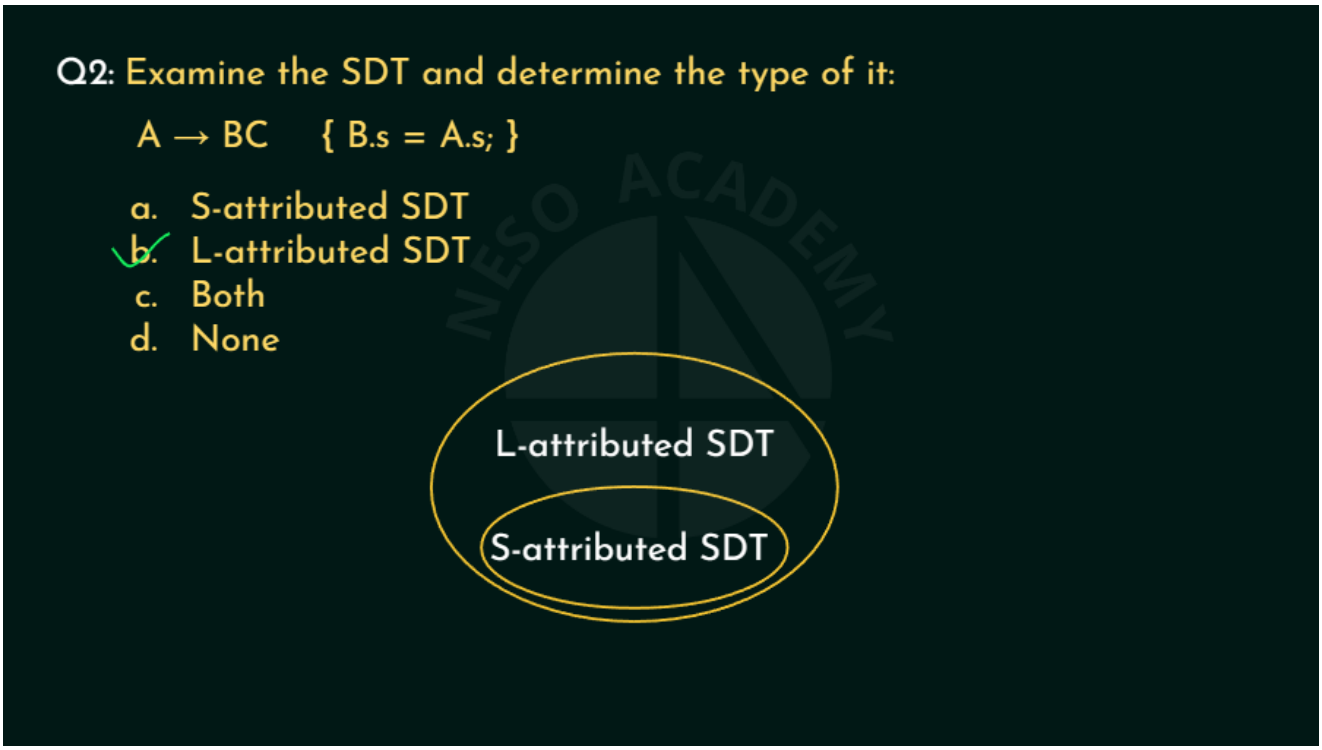
Outcome☆Two solved problems on determining the type of SDTs.



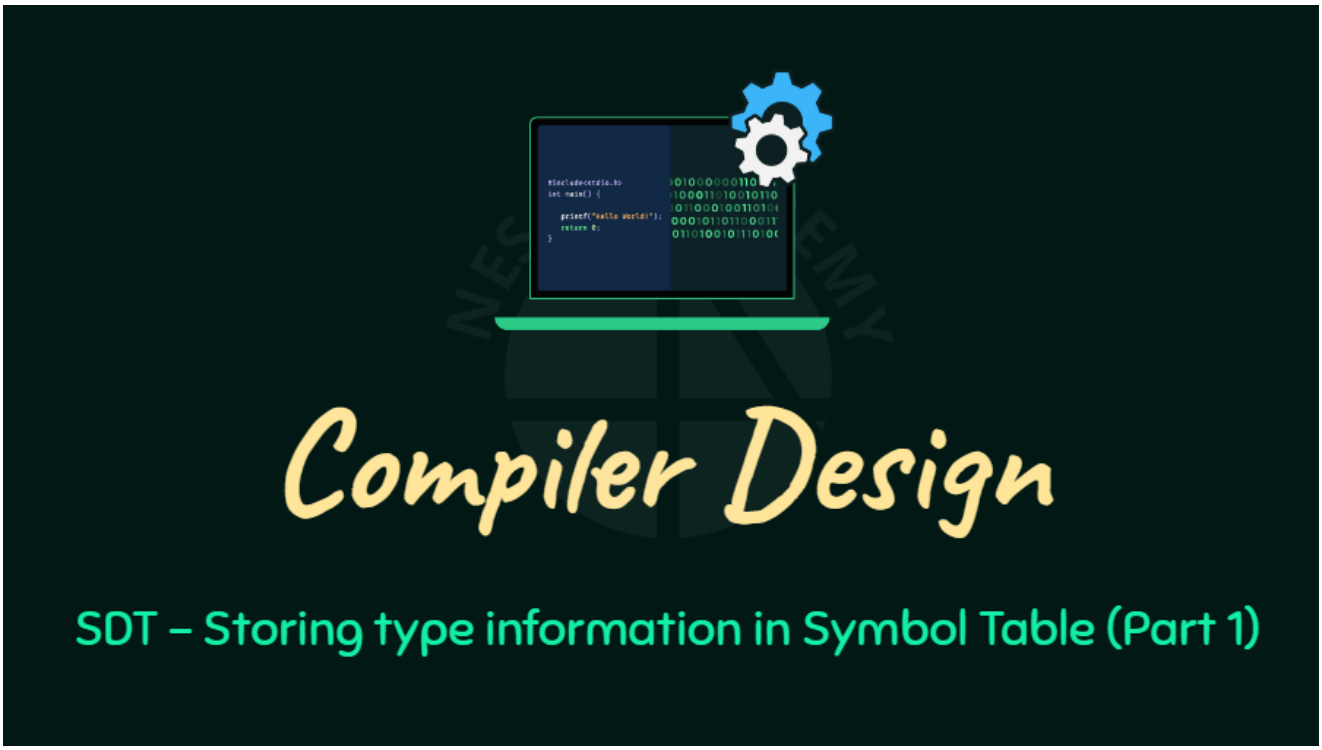


Q1:Examine the SDT and determine the type of it: A → LM{ L.i = f(A.i); M.s = f(L.s); A.s = f(M.s); } A → QR { R.i = f(A.i); Q.i = f(R.i); A.s = f(Q.s); }a.S-attributed SDTb.L-attributed SDTc.Bothd.None

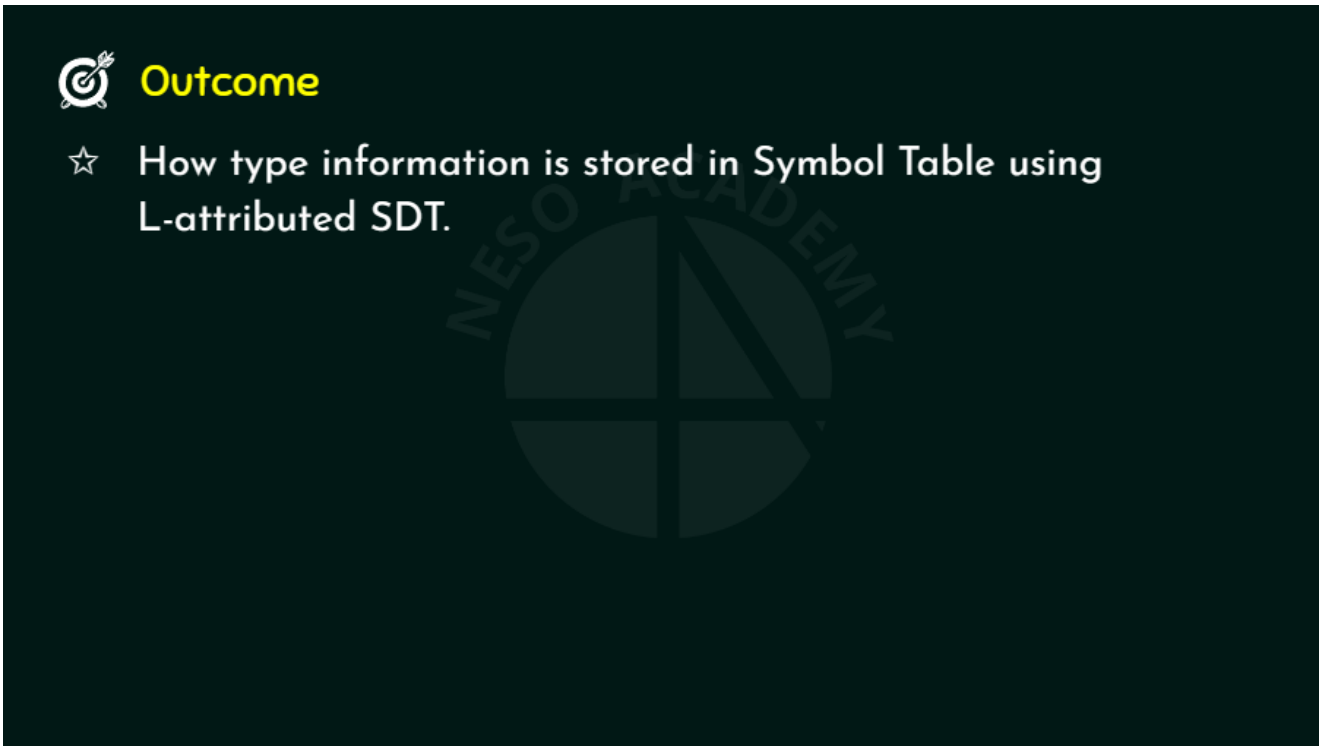




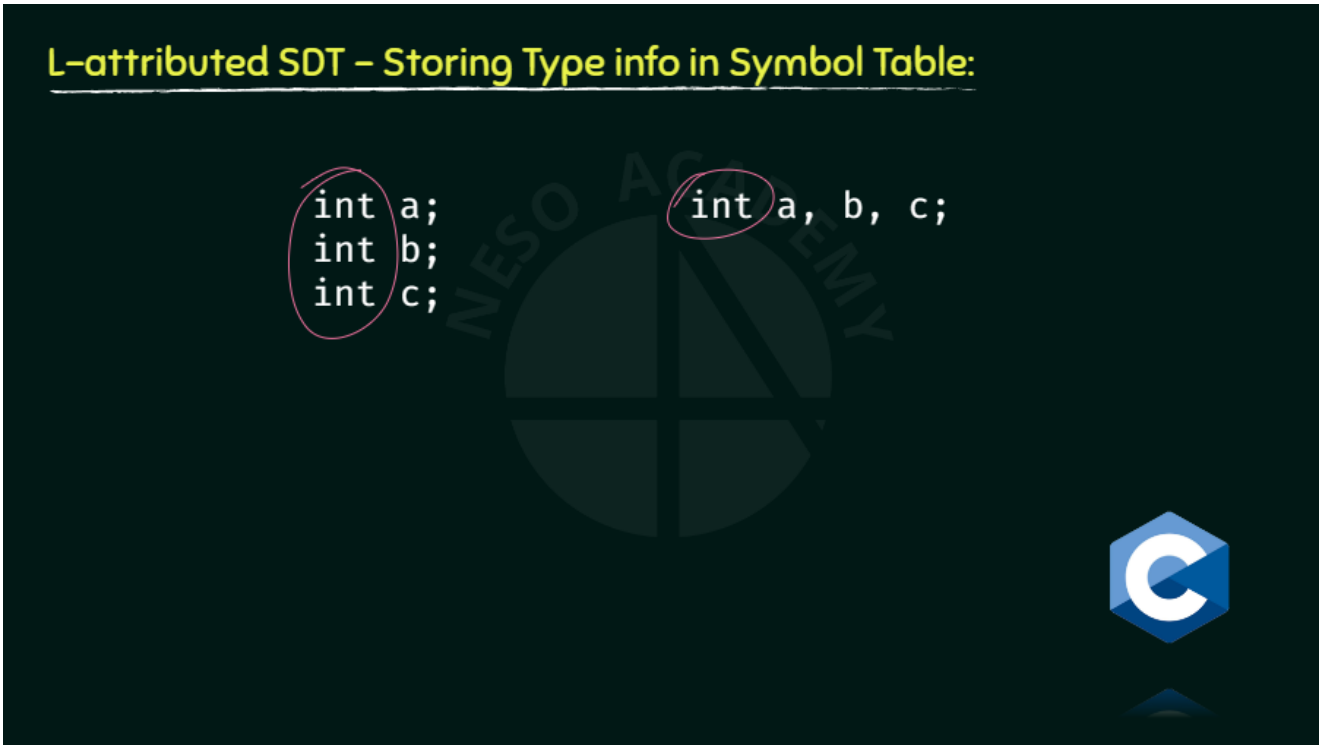
L-attributed SDTQ2:Examine the SDT and determine the type of it: A → BC{ B.s = A.s; }a.S-attributed SDTb.L-attributed SDTc.Bothd.NoneS-attributed SDT



Compiler Design SDT - Storing type information in Symbol Table (Part 1)



Outcome☆How type information is stored in Symbol Table using L-attributed SDT.



L-attributed SDT - Storing Type info in Symbol Table:int a;int b;int c;int a, b, c;

L-attributed SDT – Storing Type info in Symbol Table:

	Grammar	Semantic Rules
i.	$D \rightarrow TL$	{L.in = T.type;}
ii.	$T \rightarrow int$	{T.type = int;}
iii.	$T \rightarrow char$	{T.type = char;}
iv.	$L \rightarrow L_1$, id	{L_1.in = L.in, addtype(id.name, L.in);}
v.	$L \rightarrow id$	{addtype(id.name, L.in);}

addtype():
It is a function that adds the identifier and its type to the Symbol Table.

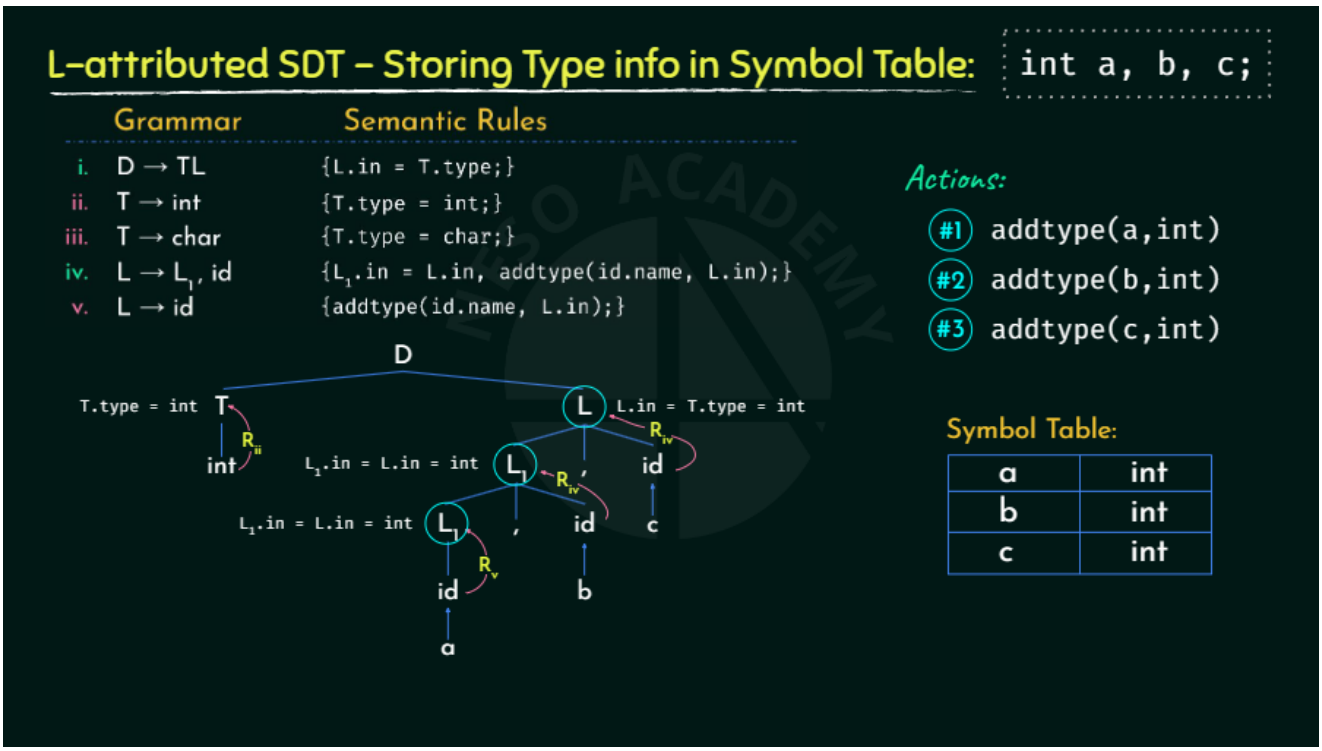
L-attributed SDT - Storing Type info in Symbol Table:{L.in = T.type;} i.ii.iii.iv.v.KGrammar D → TLT → intT → charL → L1 , idL → idSemantic Rules{L1.in = L.in, addtype(id.name, L.in);} {addtype(id.name, L.in);} {T.type = char;} {T.type = int;} addtype(): It is a function that adds the identifier and its type to the Symbol Table.

L-attributed SDT – Storing Type info in Symbol Table:

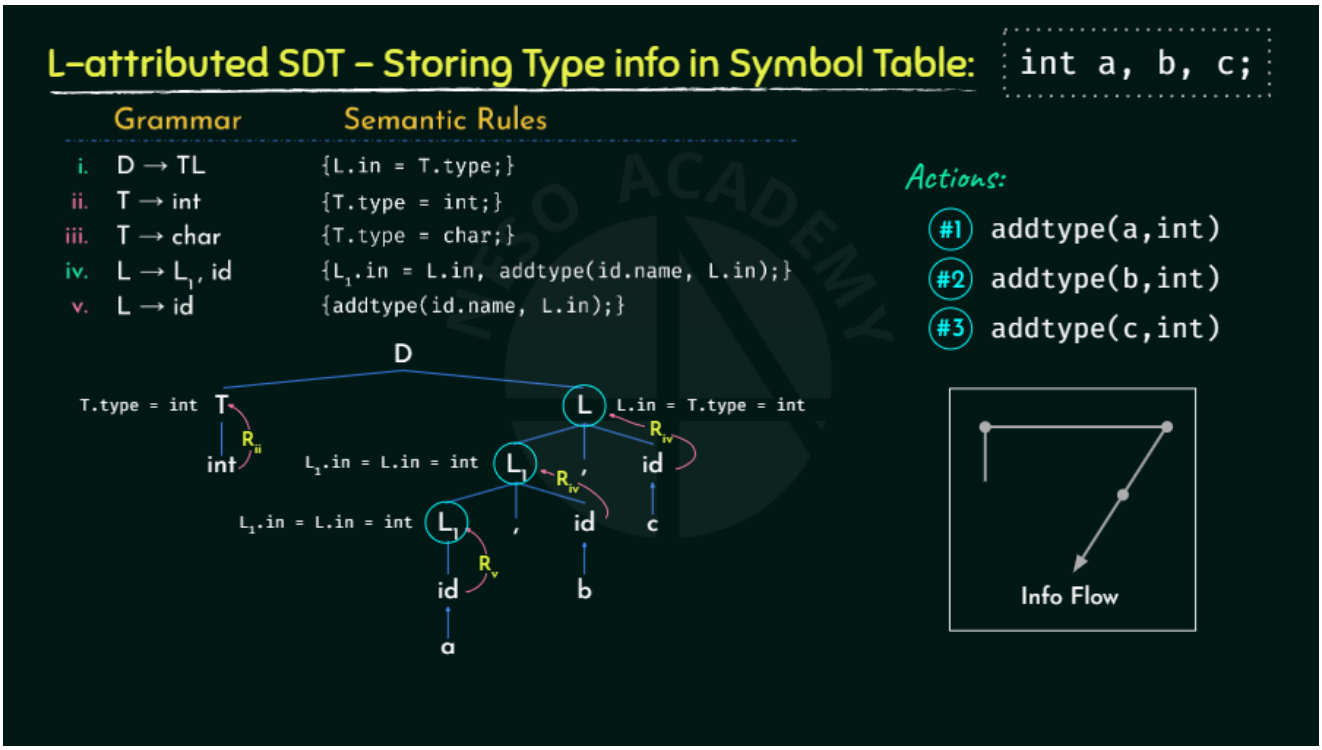
	Grammar	Semantic Rules
i.	$D \rightarrow TL$	{L.in = T.type;}
ii.	$T \rightarrow int$	{T.type = int;}
iii.	$T \rightarrow char$	{T.type = char;}
iv.	$L \rightarrow L_1$, id	{L_1.in = L.in, addtype(id.name, L.in);}
v.	$L \rightarrow id$	{addtype(id.name, L.in);}

- Synthesized Attributes (info. moves up)
- Inherited Attributes (info. moves parent to children)

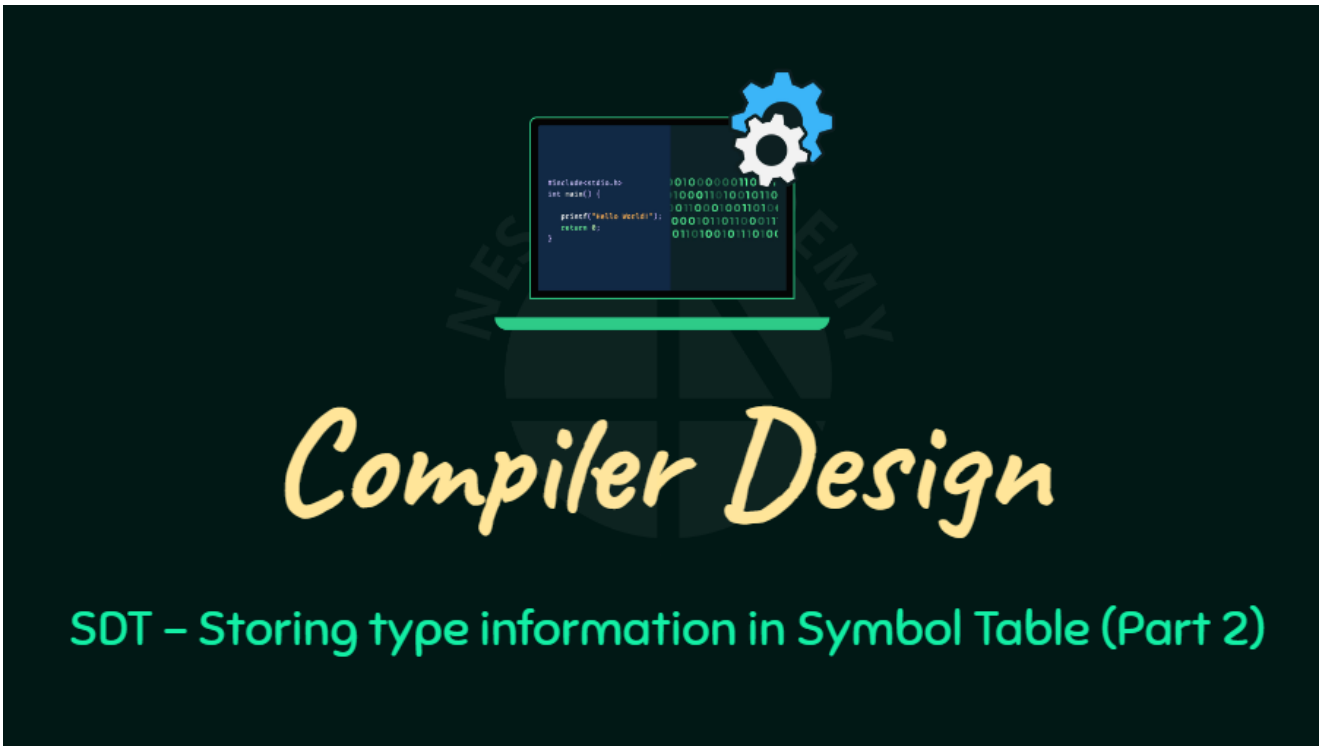
L-attributed SDT - Storing Type info in Symbol Table:{L.in = T.type;} i.ii.iii.iv.v.KGrammar D → TLT → intT → charL → L1 , idL → idSemantic Rules{L1.in = L.in, addtype(id.name, L.in);} {addtype(id.name, L.in);} {T.type = char;} {T.type = int;} ●Synthesized Attributes (info. moves up)●Inherited Attributes (info. moves parent to children)



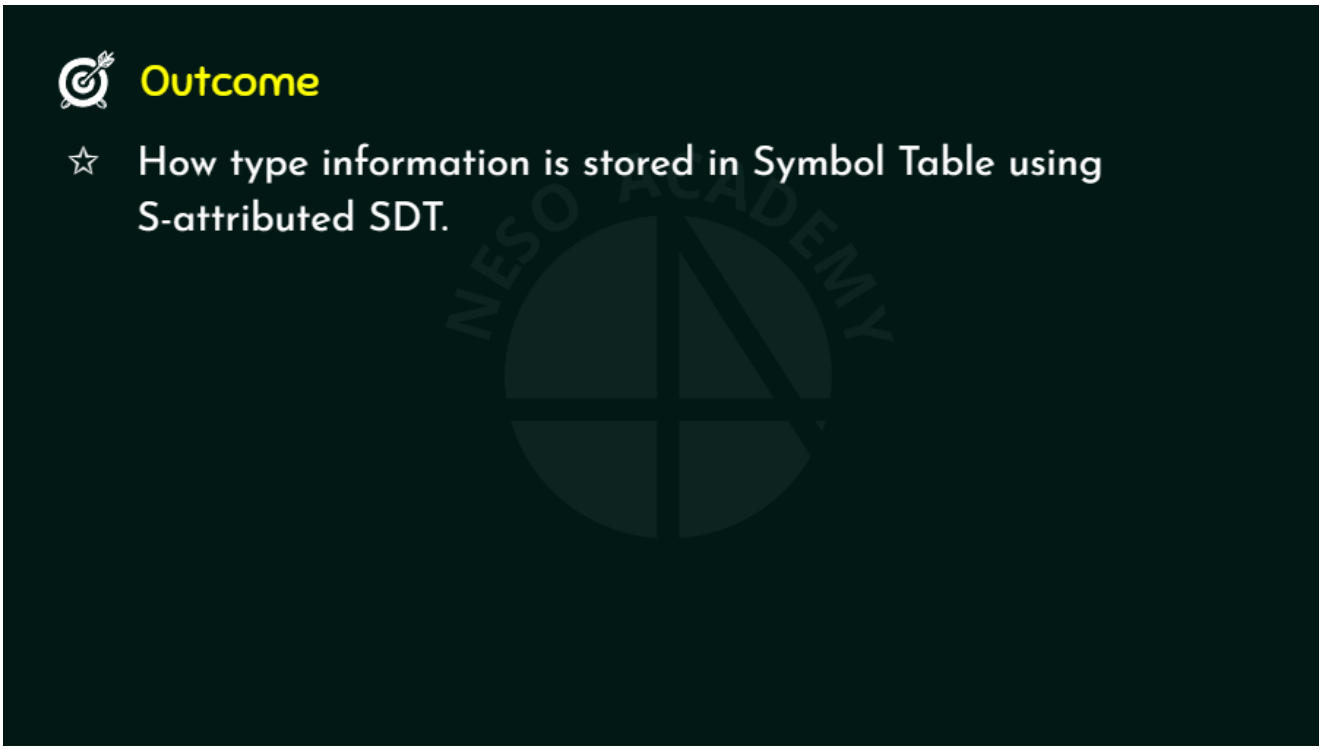
L.in = intL-attributed SDT - Storing Type info in Symbol Table:int a, b, c; Actions:#1addtype(a,int)#2addtype(b,int)#3addtype(c,int)aintbintc intSymbol Table:DTLidintabcL1id,L1id,RiiT.type = intL.in =L1.in = L.in = intRvT.type = intL1.in =RivRiv



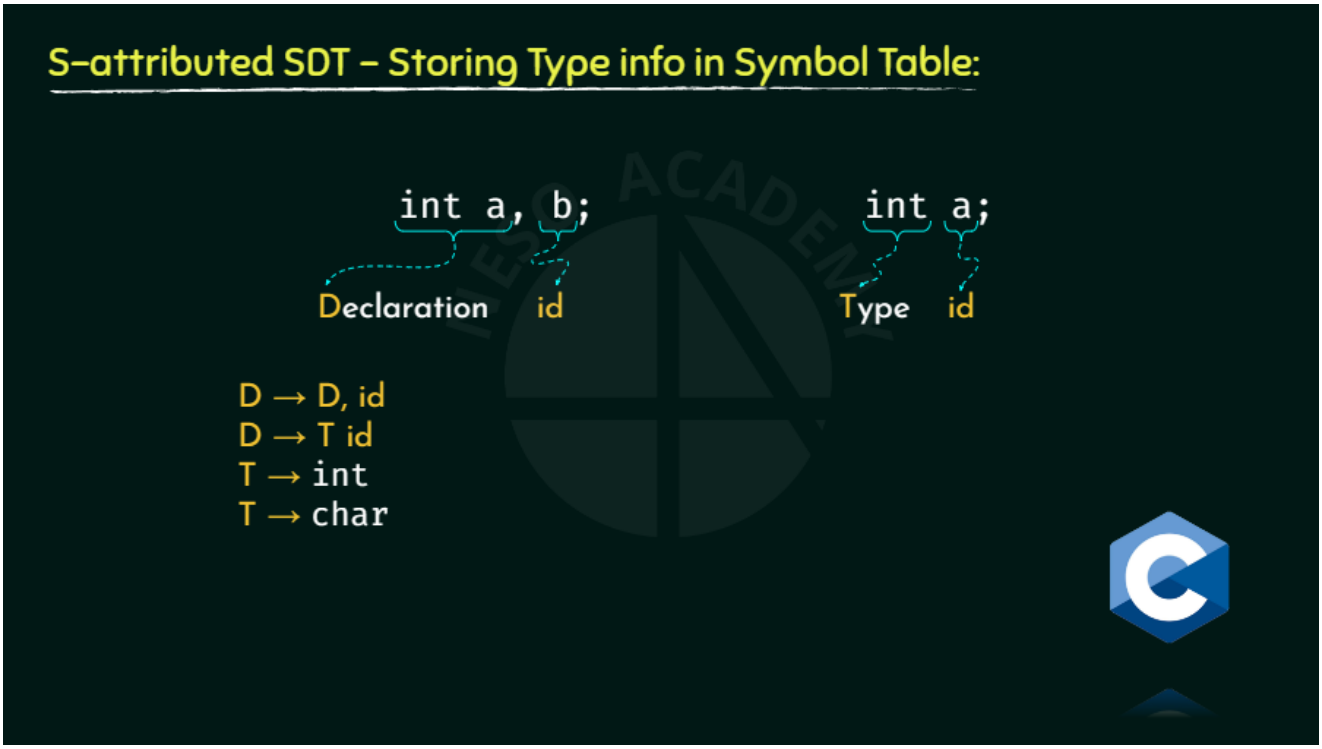
Info FlowL-attributed SDT - Storing Type info in Symbol Table:int a, b, c; DTLidintabcL1id,L1id,RiiT.type = intL1.in = L.in = intL1.in = L.in = intRvActions:#1addtype(a,int)#2addtype(b,int)#3addtype(c,int)RivRivL.in =T.type = int



Compiler Design SDT - Storing type information in Symbol Table (Part 2)



Outcome☆How type information is stored in Symbol Table using S-attributed SDT.



S-attributed SDT - Storing Type info in Symbol Table:int a, b;int a;DeclarationididTypeD → D, idD → T idT → intT → char

S-attributed SDT – Storing Type info in Symbol Table:

	Grammar	Semantic Rules
i.	D → D, id	{addtype(id.name, D.type), D.type = D.type;}
ii.	D → T id	{addtype(id.name, T.type), D.type = T.type;}
iii.	T → int	{T.type = int;}
iv.	T → char	{T.type = char;}

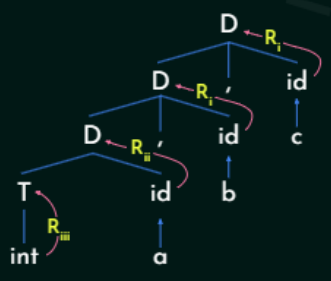
addtype():
It is a function that adds the identifier and its type in the Symbol Table.

{addtype(id.name, D.type), D.type = D.type;}S-attributed SDT - Storing Type info in Symbol Table:i.ii.iii.iv.Grammar D → D, idD → T idT → intT → charSemantic Rules{T.type = int;} addtype(): It is a function that adds the identifier and its type in the Symbol Table. {addtype(id.name, T.type){T.type = char;} , D.type = T.type;}

S-attributed SDT – Storing Type info in Symbol Table:

`int a, b, c;`

	Grammar	Semantic Rules
i.	D → D, id	{addtype(id.name, D.type), D.type = D.type;}
ii.	D → T id	{addtype(id.name, T.type), D.type = T.type;}
iii.	T → int	{T.type = int;}
iv.	T → char	{T.type = char;}



Actions:

1. R_{iii}: T → int — T.type = int
2. R_{ii}: D → T id — addtype(a, int)
3. R_{i}: D → D, id — addtype(b, int)
4. R_{i}: D → D, id — addtype(c, int)

Symbol Table:

a	int
b	int
c	int

Info FlowS-attributed SDT - Storing Type info in Symbol Table:int a, b, c; D,idD,idDidTintcbaRiiiRiRiRiiActions:aintbintc intSymbol Table:3. Ri:D → D, id1. Riii:T → int2. Rii:D → T idT.type = intaddtype(a, int) addtype(b, int) addtype(c, int) 4. Ri:D → D, id