1. **What is SDD?**
2. **Differenciate between Synthesized and inherited attributes.**

|  |  |  |
| --- | --- | --- |
| Definition | Synthesized attributes are attributes whose value at a node in the syntax tree is determined solely from attributes of its children. | Inherited attributes are attributes whose value at a node in the syntax tree is determined from attributes of its parent and possibly its siblings. |
| Direction of Flow | Synthesized attributes have a bottom-up flow of information, where values are propagated from children to parent nodes. | Inherited attributes have a top-down flow of information, where values are passed from parent nodes to their children. |
| Dependency | Synthesized attributes depend only on the attributes of child nodes. | Inherited attributes may depend on attributes of both parent and child nodes. |
| Semantic Properties | Synthesized attributes are typically used to represent properties that are computed based on the properties of the node's children. | Inherited attributes are often used to represent properties that are derived from the context of the node within the syntax tree. |
| Value Computation | Synthesized attribute values are computed using attribute equations associated with production rules. | Inherited attribute values are typically computed using semantic rules associated with production rules. |
| Usage | Synthesized attributes are commonly used for representing properties like type, size, or value. | Inherited attributes are often used for representing context-dependent properties like scope or visibility. |
| Complexity | Synthesized attributes tend to be simpler to understand and implement compared to inherited attributes. | Inherited attributes may introduce additional complexity due to their dependency on parent nodes. |
| Cycles | Synthesized attributes do not lead to circular dependencies, as they propagate bottom-up in the syntax tree. | Inherited attributes can potentially lead to circular dependencies if not carefully handled. |
| Order of Evaluation | Synthesized attributes can be evaluated in any order, as long as all their dependencies are resolved. | Inherited attributes often require a specific order of evaluation, typically top-down, to ensure correct propagation of values. |
| Inheritance | Synthesized attributes are not inherited by child nodes. | Inherited attributes are explicitly inherited by child nodes. |
| Propagation Mechanism | Synthesized attributes use a mechanism similar to a post-order traversal of the syntax tree for propagation. | Inherited attributes use a mechanism similar to a pre-order traversal of the syntax tree for propagation. |
| Example | Synthesized attribute example: In a syntax tree representing arithmetic expressions, the type of an expression node might be synthesized from the types of its children (e.g., integer, float). | Inherited attribute example: In a syntax tree representing a programming language, the scope of a variable might be inherited from its parent node (e.g., block or function). |
| Flexibility | Synthesized attributes are more flexible in terms of expressing node properties based solely on children. | Inherited attributes allow for expressing properties that depend on context or parent nodes. |
| Implementation | Synthesized attributes are often easier to implement efficiently in compilers and interpreters. | Inherited attributes may require more complex data structures or algorithms for correct propagation and evaluation. |
|  |  |  |

1. **What is S attributed definition and L attributed definition? Explain with example.**

In an S-attributed grammar, all attributes are synthesized. This means that the values of attributes associated with a node in the syntax tree are determined solely by attributes associated with its children. There are no dependencies on attributes associated with siblings or ancestors.

Example:

Consider a simple grammar for arithmetic expressions:

E → E + T | T

T → T \* F | F

F → (E) | num

Let's define synthesized attributes:

E.val: The value of the expression.

T.val: The value of the term.

F.val: The value of the factor.

The attribute equations for this grammar might be:

E → E1 + T2 { E.val = E1.val + T2.val }

| T1 { E.val = T1.val }

T → T1 \* F2 { T.val = T1.val \* F2.val }

| F1 { T.val = F1.val }

F → (E) { F.val = E.val }

| num { F.val = num.value }

In this example, each attribute's value is determined only by the values of its children.

**L-attributed Definition:**

In an L-attributed grammar, attributes can be either synthesized or inherited, but the dependencies must respect the left-to-right order of traversal of the syntax tree. This means that the values of attributes associated with a node in the syntax tree may depend on attributes associated with its children as well as its left siblings.

Example:

Extending the previous grammar to make it L-attributed:

E → E1 + T { E.val = E1.val + T.val }

| T { E.val = T.val }

T → T1 \* F { T.val = T1.val \* F.val }

| F { T.val = F.val }

F → (E) { F.val = E.val }

| num { F.val = num.value }

Here, the value of E.val depends on T.val, but also on E1.val, and the value of T.val depends on F.val as well as T1.val. These dependencies follow the left-to-right traversal order.

1. **Construct SDD syntax tree for expressions.**
2. **What is SDT? Explain top down translation for L attributed definition.**
3. **What is intermediate code generation?***It’s there in hand written notes*
4. **Explain following:  
   a. Syntax tree**

**b. Postfix notation**

Postfix Notation

* Postfix notation is the useful form of intermediate code if the given language is expressions.
* Postfix notation is also called as 'suffix notation' and 'reverse polish'.
* Postfix notation is a linear representation of a syntax tree.
* In the postfix notation, any expression can be written unambiguously without parentheses.
* The ordinary (infix) way of writing the sum of x and y is with operator in the middle: x \* y. But in the postfix notation, we place the operator at the right end as xy \*.
* In postfix notation, the operator follows the operand.

Example

**Production**

1. E  →  E1 op E2
2. E  →  (E1)
3. E   →  id

|  |  |
| --- | --- |
| **Semantic Rule** | **Program fragment** |
| E.code = E1.code || E2.code || op | print op |
| E.code = E1.code |  |
| E.code = id | print id |

c. Three address code

Three address code

* Three-address code is an intermediate code. It is used by the optimizing compilers.
* In three-address code, the given expression is broken down into several separate instructions. These instructions can easily translate into assembly language.
* Each Three address code instruction has at most three operands. It is a combination of assignment and a binary operator.

Example

GivenExpression:

1. a := (-c \* b) + (-c \* d)

Three-address code is as follows:

t1 := -c

t2 := b\*t1

t3 := -c

t4 := d \* t3

t5 := t2 + t4

a := t5

**t** is used as registers in the target program.

The three address code can be represented in two forms: **quadruples** and **triples**.

1. **Differentiate between parse tree & syntax tree.**

|  |  |  |
| --- | --- | --- |
|  | **Parse tree** | **Syntax tree.** |
| Purpose | Represents the syntactic structure of the input according to the grammar rules and the parsing process. | Represents the syntactic structure of the input abstracted from specific grammar rules, focusing more on the underlying language structure. |
| Granularity | Captures all the details of the parsing process, including non-terminal expansions and terminal symbols. | Abstracts away some parsing details, focusing on the essential structure of the program. |
| Construction | Directly reflects the parsing process, with nodes corresponding to grammar rules and terminals. | Constructed after parsing, by transforming the parse tree, collapsing nodes and removing unnecessary details. |
| Node Representation | Nodes correspond directly to grammar rules or terminal symbols. | Nodes represent language constructs or syntactic elements, abstracted from specific grammar rules. |
| Clarity | Provides a detailed view of how the input conforms to the grammar rules, making it useful for debugging parsers. | Offers a clearer view of the program's structure, focusing on the relevant syntactic elements for subsequent compiler phases. |
| Ambiguity Handling | May contain multiple parse trees for an ambiguous input, reflecting different interpretations of the input according to the grammar rules. | Typically represents the disambiguated structure of the input, resolving ambiguities during parsing or through subsequent analysis. |
| Repetition Handling | Includes nodes for repetitions, such as loops and recursion, as dictated by the grammar rules. | May abstract away repetitions, representing them in a more compact form, such as loops or recursion constructs. |
| Size | Can be larger and more detailed, especially for complex or ambiguous inputs. | Generally smaller and more concise, focusing on essential language constructs. |
| Context Sensitivity | Captures context-sensitive information directly from the parsing process, such as symbol tables and type information. | Typically abstracts away context-sensitive details, focusing on the syntactic structure. |
| Transformation | Often transformed into a syntax tree through a process of collapsing and abstraction. | May undergo further transformations during semantic analysis and optimization phases of compilation. |
| Tool Usage | Used primarily in parser generators and debugging tools to visualize the parsing process. | Essential for subsequent compiler phases, such as semantic analysis, optimization, and code generation. |
| Tree Depth | Tends to have greater depth, reflecting the nesting of grammar rules during parsing. | Often shallower, abstracting away unnecessary nesting and focusing on the essential structure. |
| Terminology | Also known as Concrete Syntax Tree (CST) or Derivation Tree. | Also known as Abstract Syntax Tree (AST). |
| Usability | Less suitable for semantic analysis and code generation due to its focus on parsing details. | Optimized for semantic analysis, optimization, and code generation, providing a more suitable representation of the program's structure. |

1. **What is backpatching? https://www.ques10.com/p/9481/explain-back-patching-with-an-example/**

* The problem in generating three address codes in a single pass is that we may not know the labels that control must go to at the time jump statements are generated.
* So to get around this problem a series of branching statements with the targets of the jumps temporarily left unspecified is generated.
* Back Patching is putting the address instead of labels when the proper label is determined.

Back patching Algorithms perform three types of operations

1) makelist (i) – creates a new list containing only i, an index into the array of quadruples and returns pointer to the list it has made.

2) Merge (i, j) – concatenates the lists pointed to by i and j, and returns a pointer to the concatenated list.

3) Backpatch (p, i) – inserts i as the target label for each of the statements on the list pointed to by p.

**Back patching for Boolean Expressions**

* We now construct a translation scheme suitable for generating code for boolean expressions during bottom-up parsing.
* A marker nonterminal M in the grammar causes a semantic action to pick up, at appropriate times, the index of the next instruction to be generated. The grammar is as follows:

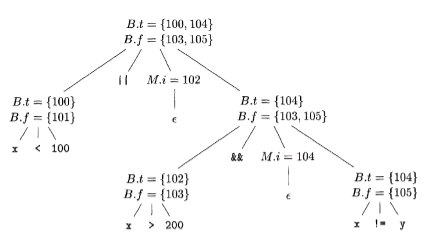
B -+ B1 || MB2 | B1 && M B2 | ! B1 | (B1) | E1 rel E2 | true | false M+€

| **1** | **B -> B1**||||**M B2** | **{backpatch(B1.falselist, M.instr); B. truelist = merge(B1. truelist, B2. truelist); B. falselist = B2. falselist; )** |
| --- | --- | --- |
| 2 | B -> B1 && M B2 | {backpatch(B1 . truelist, M. instr); B. truelist = B2 . truelist;,B. falselist = merge(Bl. falselist, B2 . falselist); } |
| 3 | B -> ! B1 | { B. truelist = B1 . falselist; B. falselist = B1 . truelist; ) |
| 4 | B->(B1) | { B, truelist = B1. truelist; B. falselist = B1 .falselist; ) |
| 5 | B -> E1 rel E2 | { B. truelist = makelist(nextinstr) ; B. falselist = makelist(nextinstr + 1);,emit(‘if’ E1 .addrrel.op E2.addr ’goto –‘); emit(‘goto –‘); } |
| 6 | B -> true | { B . truelist = makelist(nextinstr) ; emit(‘goto –‘); ) |
| 7 | B -> false | { B .falselist = makelist(nextinstr) ; emit(‘goto –‘); ) |
| 8 | M ->€ | M. instr = nextinstr; } |

**Translation scheme for Boolean expressions**

Consider semantic action (1) for the production B ->B1|| M B2. If B1 is true, then B is also true, so the jumps on B1. *Truelist* become part of B. *truelist*.

* If B1 is false, however, we must next test B2, so the target for the jumps B1.falselist must be the beginning of the code generated for B2.
* This target isobtained using the marker nonterminal M. That nonterminal produces, as a Synthesized attribute M.instr, the index of the next instruction, just before B2 code starts being generated.
* To obtain that instruction index, we associate with the production M ->€ the semantic action { M. instr= nextinstr; }
* The variable nextinstr holds the index of the next instruction to follow.
* Thisvalue will be backpatched onto the Bl .falselist (i.e., each instruction on the list Bl. falselist will receive M.instr as its target label) when we have seen the remainder of the production B ->B1 I I M B2.
* Semantic action (2) for B ->B1 &&M BZ is similar to (1). Action (3) for B ->!B swaps the true and false lists. Action (4) ignores parentheses. For simplicity, semantic action (5) generates two instructions, a conditional goto and an unconditional one. Neither has its target filled in. These instructions are put on new lists, pointed to by B.truelist and B.falselist, respectively.



Consider again the expression

x<100 | | x> 200 && x!=y

* An annotated parse tree is shown for readability, attributes truelist, falselist, and instr are represented by their initial letters.
* The actions are performed during a depth-first-traversal of the tree.
* Since all actions appear at the ends of right sides, they can be performed in conjunction with reductions during a bottom-up parse.
* In response to the reduction of x <100 to B by production (5), the two instructions

100: if x< 100 go to

101: goto -

are generated. (We arbitrarily start instruction numbers at 100.) The markernonterminalM in the production.

B→B1||MB2𝐵→𝐵1||𝑀𝐵2

records the value of next instr, which at this time is 102. The reduction of x >200 to B by production (5) generates the instructions

102: if x\gt200 go to103:goto−102: if x\gt200 go to103:𝑔𝑜𝑡𝑜−

The sub expression x >200 corresponds to B1 in the production

B→B1  &&M  B2𝐵→𝐵1  &&𝑀  𝐵2

The marker nonterminal M records the current value of next instr, which is now 104. Reducing x ! = y into B by production (5) generates

104:if  x!=ygoto105:goto−104:𝑖𝑓  𝑥!=𝑦𝑔𝑜𝑡𝑜105:𝑔𝑜𝑡𝑜−

* We now reduce by B -+ B1 &&M B2. The corresponding semantic action calls backpatch (B1 .truelist, M.instr) to bind the true exit of Blto the first instruction of B2.
* Since B1. truelist is (102) and M. instr is 104, this call to backpatch fills in 104 in instruction 102. The six instructions generated so far are thus as shown.
* The semantic action associated with the final reduction by B -+ B1 I IM B2 calls backpatch ({101},102) which leaves the instructions.
* The entire expression is true if and only if the gotos of instructions 100 or 104 are reached, and is false if and only if the gotos of instructions 103 or 105 are reached.
* These instructions will have their targets filled in later in the compilation, when it is seen what must be done depending on the truth or falsehood of the expression.

**Flow-of-Control Statements**

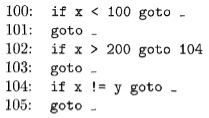
* We now use backpatching to translate flow-of-control statements in one pass.
* Consider statements generated by the following grammar:

S →→ if (B)S | if (B) S else S | while (B) S| {L}|A;

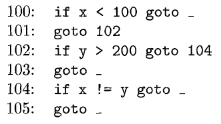
L → LS |S

Here S denotes a statement , L a statement , L a statement list, A an assignment-statement, and B a boolean expression.

Note that there must be other productions, such as



(a) After back patching 104 into instruction 102.



(b) After backpatching 102 into instruction 101.

those for assignment-statements.

* The productions given, however, are sufficient to illustrate the techniques used to translate flow-of-control statements.
* We make the tacit assumption that the code sequence in the instruction array reflects the natural flow of control from one instruction to the next.
* If not, then explicit jumps must be inserted to implement the natural sequential flow of control.

**Applications of Backpatching:**

* Backpatching is used to translate flow-of-control statements in one pass itself.
* Backpatching is used for producing quadruples for boolean expressions during bottom-up parsing.
* It is the activity of filling up unspecified information of labels during the code generation process.
* It helps to resolve forward branches that have been planted in the code.

1. **Write is Annoted Parse Tree? Construct annotated parse tree for 3\*5+4 using S attribute definition.**

An Annotated Parse Tree, also known as an Annotated Syntax Tree (AST), is a data structure used in compiler construction and programming language processing. It extends the concept of a traditional parse tree by associating additional information, called annotations or attributes, with each node of the tree. These annotations provide semantic information about the corresponding syntax elements of the source code. Let's delve into this concept in detail:

**1. Structure:**

* Node Structure: Each node in the annotated parse tree represents a syntactic construct from the source code. Nodes correspond to terminals (e.g., tokens, identifiers, literals) or non-terminals (e.g., expressions, statements, declarations) of the language grammar.
* Hierarchical Representation: The tree structure reflects the hierarchical relationships between language constructs as dictated by the grammar rules.

**2. Annotations:**

* Semantic Information: Annotations associated with each node convey additional semantic details about the corresponding language construct.
* Types of Annotations: Annotations can include type information, symbol table entries, intermediate representation (IR) nodes, evaluation results, and any other relevant semantic attributes.
* Computed or Derived: Annotations are typically computed or derived during the parsing process or subsequent semantic analysis phases. They provide a means to capture and propagate semantic information throughout the compilation process.

**3. Integration with Parsing:**

* Semantic Actions: During parsing, semantic actions associated with grammar rules compute and propagate annotations as the parse tree is constructed.
* Bottom-up or Top-down: Annotations can be computed bottom-up, starting from the leaf nodes and moving towards the root, or top-down, starting from the root and moving towards the leaves.
* Augmentation of Parsing Algorithms: Parsing algorithms such as LL parsing or LR parsing can be augmented to support the computation and propagation of annotations alongside the construction of the parse tree.

**4. Semantic Analysis:**

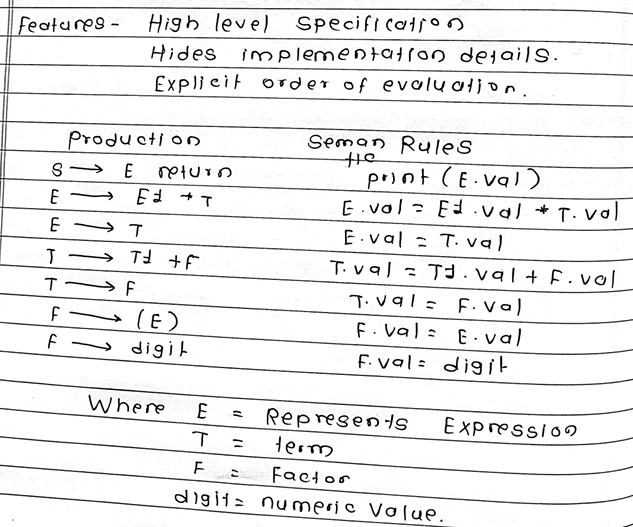
* Foundation for Analysis: Annotated parse trees serve as the foundation for subsequent semantic analysis phases in the compilation process.
* Semantic Checks: Semantic analysis algorithms traverse the annotated parse tree, performing checks and validations based on the semantic information captured in the annotations.
* Error Reporting: Semantic errors detected during analysis may result in additional annotations being added to the parse tree to record diagnostic information.

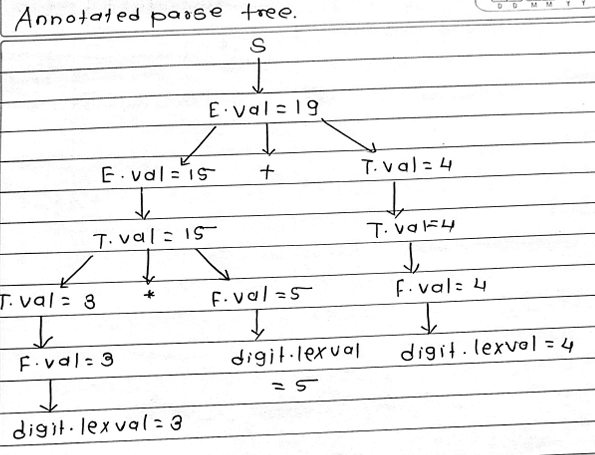
**5. Optimization and Code Generation:**

* Input for Optimization: Annotated parse trees are used as input for optimization phases of the compiler.
* Semantic Information Utilization: Optimization techniques analyze the annotations to identify opportunities for improving the program's efficiency or reducing code size.
* Code Generation: Code generation algorithms utilize the annotated parse tree to produce target code, incorporating the semantic information captured in the annotations.

**6. Representation:**

* Data Structures: Annotated parse trees can be represented using various data structures, such as ASTs augmented with annotation fields at each node.
* Storage of Annotations: Annotations may be stored directly within the parse tree nodes or maintained separately in symbol tables or other data structures, depending on the requirements of the compiler.





**Advantages:**

1. Unified Representation: Annotated parse trees provide a unified representation that combines both syntactic and semantic information. This integrated view simplifies subsequent compiler phases, as semantic information is readily available.
2. Facilitates Semantic Analysis: By capturing semantic information directly within the parse tree, annotated parse trees streamline the semantic analysis process. Semantic checks and validations can be performed efficiently using the annotations associated with each node.
3. Supports Optimization: Annotated parse trees serve as a basis for optimization techniques. The semantic information contained in the annotations can be leveraged to identify and apply various optimization strategies, leading to improved program efficiency and performance.
4. Enables Targeted Error Reporting: Semantic errors detected during analysis can be associated with specific nodes in the annotated parse tree, facilitating targeted error reporting. This makes it easier for developers to understand and address issues in their code.
5. Simplifies Code Generation: During code generation, annotated parse trees provide valuable semantic context that guides the generation of target code. This simplifies the code generation process and ensures that the generated code adheres to the intended semantics of the source program.

**Disadvantages:**

1. Increased Memory Overhead: Annotated parse trees require additional storage space to store annotations alongside the tree nodes. This can lead to increased memory overhead, especially for large source files or complex programs.
2. Complexity in Construction: Augmenting parsing algorithms to compute and propagate annotations adds complexity to the parsing process. Implementing semantic actions and ensuring correct annotation propagation requires careful design and implementation.
3. Potential Performance Overhead: The computation and propagation of annotations during parsing may introduce performance overhead, particularly in cases where extensive semantic analysis is performed during parsing. This can impact the overall compilation time, especially for large codebases.
4. Maintenance Overhead: Managing and updating annotations as the parse tree is manipulated and transformed throughout the compilation process can introduce maintenance overhead. Ensuring consistency and correctness of annotations across different compiler phases requires careful attention.
5. Limited Expressiveness: Despite capturing semantic information, annotated parse trees may not fully represent all aspects of program semantics. Certain complex semantic properties or dynamic behaviors may be challenging to capture and annotate effectively within the parse tree structure.
6. **Write Syntax Directed Translation Scheme for Assignment Statements.**
7. **Write Syntax Directed Translation Scheme for boolean expression.**