|  |  |  |
| --- | --- | --- |
| UNIT-IV | | |
| 1 | Write short notes on  a)s attributed definitions  b)l attributed definitions  c)inherited attributes  d)synthesized attributes | 5 |
| 2 | Check whether the given SDD is L-attributed or not   |  |  | | --- | --- | | A-->PQ | P.in=p(A.in) | | Q.in=q(p.sy) | | A.sy=f(q.sy) | | A-->XY | Y.in=y(A.in) | | x.in=x(y.sy) | | A.sy=f(X.sy) | |  |
| 3 | Write the quadruples,triples, and indirect triples for the expresstion  A=b\*-c+b\*-c |  |
| 4 | Write the type expressions for the for the following types  a)An Array of pointers to reals,where the array index ranges from 1 to 100  b)A two dimentional array of integers (i.e an array of array)whose rows are indexed from 0-9 and whose columns are indexed from -10 to10  c)functions whose domains are functions from integers to pinters of integers and whose ranges are records consisting of an integer and a character |  |
| 5 | Explain the equivalence of type expressions |  |
| 6 | Draw the syntax tree for the expression (a\*b)+(c-d)+\*(a\*b)+b |  |
| 7 | For the following grammar construct the Syntax-directed definition and generate the code fragment (translator)using S-attributed definitions  S🡪EN  E🡪E+T/ E-T/T  T🡪T\*F/F  F🡪(E)  F🡪digit  N🡪;  Also evaluate the input string 2\*3+4 with parser stack using LR parsing method |  |
| UNIT-V | | |
|  | Explain the concept of Function-Preserving Transformations?  Structure-Preserving Transformations:  The primary Structure-Preserving Transformation on basic blocks are:   Common sub-expression elimination   Dead code elimination   Renaming of temporary variables   Interchange of two independent adjacent statements.   Common sub-expression elimination:  Common sub expressions need not be computed over and over again. Instead they can be  computed once and kept in store from where it’s referenced when encountered again – of course  providing the variable values in the expression still remain constant.  Example:  a: =b+c  b: =a-d  c: =b+c  d: =a-d  The 2nd and 4th statements compute the same expression: b+c and a-d  Basic block can be transformed to  a: = b+c  b: = a-d  c: = a  d: = b   Dead code elimination:  It’s possible that a large amount of dead (useless) code may exist in the program. This  might be especially caused when introducing variables and procedures as part of construction or  error-correction of a program – once declared and defined, one forgets to remove them in case  they serve no purpose. Eliminating these will definitely optimize the code.   Renaming of temporary variables:  A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is  another temporary name, and change all uses of t to u.  In this we can transform a basic block to its equivalent block called normal-form block.   Interchange of two independent adjacent statements:  Two statements  t1:=b+c  t2:=x+y  can be interchanged or reordered in its computation in the basic block when value of t1  does not affect the value of t2. Algebraic Transformations:  Algebraic identities represent another important class of optimizations on basic blocks.  This includes simplifying expressions or replacing expensive operation by cheaper ones  i.e. reduction in strength.  Another class of related optimizations is constant folding. Here we evaluate constant  expressions at compile time and replace the constant expressions by their values. Thus  the expression 2\*3.14 would be replaced by 6.28.  The relational operators <=, >=, <, >, + and = sometimes generate unexpected common  sub expressions.  Associative laws may also be applied to expose common sub expressions. For example, if  the source code has the assignments  a :=b+c  e :=c+d+b  the following intermediate code may be generated:  a :=b+c  t :=c+d  e :=t+b   Example:  x:=x+0 can be removed  x:=y\*\*2 can be replaced by a cheaper statement x:=y\*y   The compiler writer should examine the language carefully to determine what  rearrangements of computations are permitted, since computer arithmetic does not always  obey the algebraic identities of mathematics. Thus, a compiler may evaluate x\*y-x\*z as  x\*(y-z) but it may not evaluate a+(b-c) as (a+b)-c. | 10 |
|  | Explain hthe following  Copy propagation  Dead code elimination  Constant folding  Strength reduction   * **Strength reduction** : There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x \* 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result. | 5 |
|  | What are the various attributes of the symbol table. | 5 |
|  | Explain in detail the various storage allocation strategies  Static storage allocation   * In static allocation, names are bound to storage locations. * If memory is created at compile time then the memory will be created in static area and only once. * Static allocation supports the dynamic data structure that means memory is created only at compile time and deallocated after program completion. * The drawback with static storage allocation is that the size and position of data objects should be known at compile time. * Another drawback is restriction of the recursion procedure.   Stack Storage Allocation   * In static storage allocation, storage is organized as a stack. * An activation record is pushed into the stack when activation begins and it is popped when the activation end. * Activation record contains the locals so that they are bound to fresh storage in each activation record. The value of locals is deleted when the activation ends. * It works on the basis of last-in-first-out (LIFO) and this allocation supports the recursion process.   Heap Storage Allocation   * Heap allocation is the most flexible allocation scheme. * Allocation and deallocation of memory can be done at any time and at any place depending upon the user's requirement. * Heap allocation is used to allocate memory to the variables dynamically and when the variables are no more used then claim it back. * Heap storage allocation supports the recursion process.   Example:   1. fact (int n) 2. { 3. if (n**<**=1) 4. return 1; 5. else 6. return (n \* fact(n-1)); 7. } 8. fact (6) | 10 |
|  | What is an activation record ?explain how it is related with runtime storage organization. **Activation Record**  * Control stack is a run time stack which is used to keep track of the live procedure activations i.e. it is used to find out the procedures whose execution have not been completed. * When it is called (activation begins) then the procedure name will push on to the stack and when it returns (activation ends) then it will popped. * Activation record is used to manage the information needed by a single execution of a procedure. * An activation record is pushed into the stack when a procedure is called and it is popped when the control returns to the caller function.   The diagram below shows the contents of activation records:  Activation Record  **Return Value:** It is used by calling procedure to return a value to calling procedure.  **Actual Parameter:** It is used by calling procedures to supply parameters to the called procedures.  **Control Link:** It points to activation record of the caller. | 5 |
|  | Write briefly about various loop optimization techniques.  **Loop Optimization** is the process of increasing execution speed and reducing the overheads associated with loops. It plays an important role in improving cache performance and making effective use of parallel processing capabilities. Most execution time of a scientific program is spent on loops.  *Loop Optimization is a machine independent optimization.*  Decreasing the number of instructions in an inner loop improves the running time of a program even if the amount of code outside that loop is increased.  **Loop Optimization Techniques:**     1. **Frequency Reduction (Code Motion):** In frequency reduction, the amount of code in loop is decreased. A statement or expression, which can be moved outside the loop body without affecting the semantics of the program, is moved outside the loop.   **Example:**  **Initial code:**  while(i<100)  {  a = Sin(x)/Cos(x) + i;  i++;  }  **Optimized code:**  t = Sin(x)/Cos(x);  while(i<100)  {  a = t + i;  i++;  }   1. **Loop Unrolling:** Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program’s speed by eliminating loop control instruction and loop test instructions.   **Example:**  **Initial code:**  for (int i=0; i<5; i++)  printf("Pankaj\n");  **Optimized code:**  printf("Pankaj\n");  printf("Pankaj\n");  printf("Pankaj\n");  printf("Pankaj\n");  printf("Pankaj\n");   1. **Loop Jamming:** Loop jamming is the combining the two or more loops in a single loop. It reduces the time taken to compile the many number of loops.   **Example:**  **Initial Code:**  for(int i=0; i<5; i++)  a = i + 5;  for(int i=0; i<5; i++)  b = i + 10;  **Optimized code:**  for(int i=0; i<5; i++)  {  a = i + 5;  b = i + 10;  } | 5 |
| Unit-6 | | |
| 1 | Construct the DAG for the following programm code  Sum=0  For(i=0;i<=10;i++)  Sum=sum+a[i]. | 10 |
| 2 | Explain how the following expression is converted into DAG  a+b\*(a+b)+c+d. | 5 |
| 3 | Explain natural loops and inner loops of a flow graph with an example  **Natural Loops:**    One application of dominator information is in determining the loops of a flow graph suitable for improvement. There are two essential properties of loops:    Ø     A loop must have a single entrypoint, called the header. This entry point-dominates all nodes in the loop, or it would not be the sole entry to the loop.  Ø     There must be at least one way to iterate the loop(i.e.)at least one path back to the headerOne way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If a→b is an edge, b is the head and a is the tail. These types of  edges are called as back edges.    Example:    In the above graph,    7→4 4 DOM 7    10 →7 7 DOM 10  4→3  8→3  9 →1    The above edges will form loop in flow graph. Given a back edge n → d, we define the natural loop of the edge to be d plus the set of nodes that can reach n without going through d. Node d is the header of the loop.    **Algorithm: Constructing the natural loop of a back edge**.  Input: A flow graph G and a back edge n→d.    Output: The set loop consisting of all nodes in the natural loop n→d.    Method: Beginning with node n, we consider each node m\*d that we know is in loop, to make sure that m’s predecessors are also placed in loop. Each node in loop, except for d, is placed once  on stack, so its predecessors will be examined. Note that because d is put in the loop initially, we never examine its predecessors, and thus find only those nodes that reach n without going through d.    Procedure insert(m);    if m is not in loop then begin loop := loop U {m};    push m onto stack end;  stack : = empty;    loop : = {d}; insert(n);  while stack is not empty do begin  pop m, the first element of stack, off stack;    for each predecessor p of m do insert(p)  end    **Inner loops:**  If we use the natural loops as “the loops”, then we have the useful property that unless two loops have the same header, they are either disjointed or one is entirely contained in the  other. Thus, neglecting loops with the same header for the moment, we have a natural notion of inner loop: one that contains no other loop.    When two natural loops have the same header, but neither is nested within the other, they are combined and treated as a single loop. | 5 |
| 4 | Explain the different issues in the design of code generator  **Code generator** converts the intermediate representation of source code into a form that can be readily executed by the machine. A code generator is expected to generate the correct code. Designing of code generator should be done in such a way so that it can be easily implemented, tested and maintained.  **The following issue arises during the code generation phase:**   1. **Input to code generator –** The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG’s, etc. The code generation phase just proceeds on an assumption that the input are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary. 2. **Target program –** The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language.      * + Absolute machine language as output has advantages that it can be placed in a fixed memory location and can be immediately executed.   + Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader. But there is added expense of linking and loading.   + Assembly language as output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code. And we need an additional assembly step after code generation.  1. **Memory Management –** Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator. A name in the three address statements refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name. 2. **Instruction selection –** Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also plays a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straight-forward.   For example, the respective three-address statements would be translated into the latter code sequence as shown below:  P:=Q+R  S:=P+T  MOV Q, R0  ADD R, R0  MOV R0, P  MOV P, R0  ADD T, R0  MOV R0, S  Here the fourth statement is redundant as the value of the P is loaded again in that statement that just has been stored in the previous statement. It leads to an inefficient code sequence. A given intermediate representation can be translated into many code sequences, with significant cost differences between the different implementations. A prior knowledge of instruction cost is needed in order to design good sequences, but accurate cost information is difficult to predict.   1. **Register allocation issues –** Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers are subdivided into two subproblems:    * During **Register allocation –** we select only those set of variables that will reside in the registers at each point in the program.    * During a subsequent **Register assignment** phase, the specific register is picked to access the variable.   As the number of variables increases, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain machine requires register pairs consist of an even and next odd-numbered register. For example  M a, b  These types of multiplicative instruction involve register pairs where the multiplicand is an even register and b, the multiplier is the odd register of the even/odd register pair.   1. **Evaluation order –** The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results. However, picking the best order in the general case is a difficult NP-complete problem. 2. **Approaches to code generation issues:**Code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face. Some of the design goals of code generator are:    * Correct    * Easily maintainable    * Testable    * Efficient | 5 |
| 5 | Explain about peephole optimization  Peephole optimization is a type of [Code Optimization](https://www.geeksforgeeks.org/compiler-design-code-optimization/) performed on a small part of the code. It is performed on the very small set of instructions in a segment of code.  *The small set of instructions or small part of code on which peephole optimization is performed is known as****peephole****or****window****.*  It basically works on the theory of replacement in which a part of code is replaced by shorter and faster code without change in output.  Peephole is the machine dependent optimization.    **Objectives of Peephole Optimization:**  The objective of peephole optimization is:   1. To improve performance 2. To reduce memory footprint 3. To reduce code size   **Peephole Optimization Techniques:**   1. **Redundant load and store elimination:** In this technique the redundancy is eliminated. 2. **Initial code:** 3. y = x + 5; 4. i = y; 5. z = i; 6. w = z \* 3; 7. **Optimized code:** 8. y = x + 5; 9. i = y;   w = y \* 3;   1. **Constant folding:** The code that can be simplified by user itself, is simplified. 2. **Initial code:** 3. x = 2 \* 3; 4. **Optimized code:**   x = 6;   1. **Strength Reduction:** The operators that consume higher execution time are replaced by the operators consuming less execution time. 2. **Initial code:** 3. y = x \* 2; 4. **Optimized code:** 5. y = x + x; or y = x << 1; 6. **Initial code:** 7. y = x / 2; 8. **Optimized code:**   y = x >> 1;   1. **Null sequences:** Useless operations are deleted. 2. **Combine operations:** Several operations are replaced by a single equivalent operation. | 5 |
| 6 | Show the various steps in the code generation algorithm of the expression (a+b)/(c+d) Assuming two machine registers to be available | 5 |
| 7 | Consider the following code sequence  i)MOV B R0  ADD C,R0  MOV R0,A  ii)MOV B,A  ADD C,A  Calculate the machine cost for above instructions | 5 |