Boolean expression in compiler design pdf

BOOLEAN EXPRESSIONS Boolean expressions have two main goals. They are used to calculate logical values, but they are most often used as conditional expressions in statements that change the flow of management, for example, if something-else, or while making statements. The boolean expressions are composed of boolean operators (and, or, and not) chained to elements that boolean variable or relational expressions. Relational expressions are in the form of E1 relop E2, where E1 and E2 are arithmetic expressions. Here we look at the boolean expressions generated by the following grammar : E-'gt;EorE EandE (e) (e) y id relop id true false methods of translating the expressions of Bulea: There are two main methods of presenting the value of the expressions of bulean. They are : - To encode the true and false numerical and appreciate the expression of the boulean by analogy with the arithmetic expression. Often one is used to refer to the true and 0 to refer to false. Implement the expressions of the boolean by flow management, that is, representing the value of the expressions of the boolean position achieved in the program. This method is particularly useful when implementing boolean expressions in control flow statements, such as if-then and while-do statements. The numerical representation Here, 1 denotes the true and 0 denotes false. Expressions will be judged entirely from left to right, thus as arithmetic expressions. For example: translation to a or b, not c, is a sequence of three t1 addresses: not c t2: b and t1 t3: a or t2 - relational expression, such as a b, equivalent to a conditional application, if it is a 1st 0, which can be translated into a three-address code sequence (aga statement numbers at 100) : 1000: if a qlt; b goto 103 101 : t : t : 0 102 : goto 104 103 : t : 1 104 : Translation scheme using numerical view for booleans Short-Circuit Code: We can also translate the boolean expression into three addressable code without creating code for any of the boolean operators and without the obligatory evaluation of the code of the entire expression. This style of assessment is sometimes referred to as short circuit or jumping. You can evaluate boolean expressions without generating code for boolean operators and, or, or, or not if we represent the meaning of the expression by a provision in the code sequence. Translation of the lt; b or c and e qlt; f 100: if a qlt; b goto 103 101 : t1 : 0 102 : goto 104 103 : t1 : 1 104 : if c zlt; d goto 107 105 : t2 : No 0 106 : goto 108 107 : t2 : No 1 Flow-of-Control Statements We now consider translating boolean expressions into three address codes in the context of if-then-else, and while making statements such as those generated by the following grammar: S--zgt;ifEthenS1, if E, then S1 still S2 while E do S1 In each of these productions, E is expression that needs to be translated. In translation, we assume that a statement with three addresses can be symbolically marked, and that the newlabel function returns a new symbolic label every time it is called. E.true is the mark that controls if E is true, and E.false is the mark that controls if E is false. Semantic S flow application translation rules allow the control to flow from S.code to the instruction from three addresses immediately after S.code. S.next is a label that is attached to the first instruction of the three addresses that will be executed after the code for S. Start Lecture #12 6.6.5: Avoiding Excess Gotos If there are Galileo variables (or variables in which osmelin value can be placed), we may have a boolean appointment statement. That is, we could evaluate the expressions of the boolean outside of the control flow operators. Recall that the code we created for boolean expressions (internal control flow operators) used inherited attributes to push the B.true and B.false output tag tree. How can we cope with boolean appointment statements? Two methods for Booleans Appointments Statements: Method 1 So far we have used the so-called code jumping method for Boolean quantities. We evaluated Boolean expressions (in the context of control thread operators) using inherited attributes for true and false outputs (i.e. target locations to move to if the expression is rated as true and false). With this method, if we have a statement about the appointment of Boolean, we simply allow true and false outlets to lead accordingly to statements of LHS - true LHS - false two methods for booleans Assignment Statements: Method 2 In the second method we simply treat the expression of boulean as expressions. That is, we just mimic the actions we did for integer/real ratings. Thus, Boolean's appointment statements as b OR (c AND d AND (x qlt; y)) just work. To manage the flow of statements, such as while boolean-expression make a statement-list end ; If boolean-expression, the statement-list is still a statement-list end; we just appreciate the boolean expression as if it were part of the destination statement and then two jump to where we should go if the result is true or false. However, as mentioned earlier, this is wrong. In C and other languages, if (a'0 1/a zgt; f(a) is guaranteed not to be divided into zero and the aforementioned implementation does not provide this guarantee. So even if we use Method 2 to implement Boolean expressions in destination statements, we need to implement a Boolean short-circuit assessment to manage the flow. That is, we have to use the jumping code to control the flow. The easiest way to use Method 1, i.e. jump code for all BOOLEAN expressions. Our intermediate code uses symbolic labels. At some point, they should be transferred to the instructions. If we use quad bikes, all instructions are the same thus, the address is only the number of the instruction. Sometimes we generate a jump before we generate a target, so we can't put in the no-fly instructions. Indeed, that is why we used symbolic labels. The easiest way to fix this is to make an extra pass (or two) over the quads to determine the correct instruction number and use it to replace the symbolic tag. This is additional work; a more effective method that is independent of compilation is called backpatching. 6.8: Switch operators evaluate an expression, compare it to a constant vector that is treated as switch hand tags, and perform appropriate hands (or by default). Language C is unusual in that various cases are just labels for the giant computer goto at the beginning. The more traditional idea is that you only carry one of the weapons, as in the series, if more if more ... End if 6.8.1: Translation switch-Statements Simple implementation to understand, it's just to convert the switch into a series, if still, if higher. This performs approximately k jumps (worst case) for k cases. Instead you can start by jumping in each case. It again performs about k jumps. Create a jump table. If the constant values lie in a small range and are dense, then make a list of jumps one for each number in the range and use the value calculated to determine which of these jumps to jump. This performs 2 jumps to get to the code to perform and another to go to the end of the switch. 6.8.2: Syntax-Directed Translation switch-Statement Grammar Class does not have a switch operator, so we will not do a detailed SDD. SDD for the second method above can be organized as follows. When handling the switch (E) ... production, call newlabel () to create tags for the next and testing that contribute to inherited and synthesized attributes respectively. The expression is then evaluated using code and address. The code for the switch has after the code for the E goto test. Each case begins with a new marking. The code for the case starts with this label and then the hand translation itself and ends with goto next. A generated label paired with the value for this case is added to the inherited attribute that represents the turn of these pairs - in fact, it is done by some production, as cases → cases ε as usual, the queue is sent back up the tree of Epsilone production. When we get to the end of the cases we are back on the production switch, which now adds the code to the end. Specifically, the test label is gen'ed, followed by a series if E.addr and Vi goto Li statements where each Li,Vi pair from the generated queue. 6.9 Intermediate Code of Procedures (and Functions) Part of the work on procedures is related to data storage problems and storage time conditions; this is discussed in the next chapter. Teh Teh The language we use has commands for settings, calls and returns. Improving the SDD to create this code won't be difficult, but we won't do it. The first requirement to call the type-checking procedure and call functions the first requirement is to write down the signature definition of the procedure/function in the character table (or related). The signature of the procedure is a type vector, one for each setting. The signature function also includes the type of returned value. Implementation will strengthen the SDD so that the production of parameters (s) is treated in the same way as our attitude to declarations. Procedures/functions defined at the same nesting level first consider the procedures (or functions) P and q determined at the top level, which is the only case supported by class grammar. Suppose the definition of P precedes the definition of q. If the body q (part between START and END) contains a P call, the compiler can check the types because the P signature is already saved. If P is triggered instead, additional mechanisms are needed because the q signature is not available on the call site. (Requiring the so-called procedure to always precede the subscriber would eliminate the case of reciprocal recursion, P-calls and P-calls.) The compiler can make a preliminary pass through the disassembly/syntax tree, during which it simply fills the tables. This wording could be expanded to allow both procedural/functional declarations without organs and all the definitions we have seen so far. Declarations will then be placed in the early stages of input, assuring that even one pass compiler will face the announcement of the so-called procedure/function before the call site (the definition may well be later in the file). The same considerations apply if both P and q are embedded in a different procedure/function R. The difference is that the P and q signatures are placed in the R table rather than in the top-level character table. The nesting procedures and grammar class function do not support the nesting areas at all. However, the grammatical changes are small. Another form of declaration is the definition of procedures/functions. DECLARE statements START STATEMENTS END ; becomes another form of statement. As we mentioned earlier, the nesting/nesting procedure/block has a significant impact on the management of the character table. In the next chapter, we'll see what the compiler-generated code has to do to access non-local names that are the result of nested areas. Homework: Read Chapter 7. 7.1: Storage Organization We discuss storage organization from the point of view of the compiler, which should make room for the launch of programs. In particular, we care only about virtual addresses and treat them evenly. This should be compared to operating systems where we worry about effectively match these virtual addresses with real memory. For example, see a discussion of these diagrams in my OS class notes that illustrate the OS's difficulty

with the distribution method used in this course, a method that uses a very large virtual address range. Perhaps the simplest solution uses tiered page tables. Some systems require different alignment restrictions. For example, 4-row integers may need to start at byte, which is a multiple of four. Uncoordinated data can be illegal or reduce performance. Upholstery is often used to achieve proper alignment. Areas (segments) of memory As mentioned above, there are various OS problems we ignore, such as displaying from virtual to physical addresses, and the effects of demand paging. In this class, we simply highlight memory segments in virtual memory, allowing the operating system to worry about managing real memory. Specifically, we'll look at the following four areas of virtual memory. The code (often called OS-speak text) is fixed size and unchanged (self-changing code has long gone out of fashion). If there is OS support, the text can only be tagged (or perhaps read and perform, but not write). All other areas will be labeled as unfeasible. Fixed-size data that can be determined by the compiler is likely by studying only the structure of the program without determining the program's execution pattern. One example is global data. Storage of this data will be highlighted in the next area immediately after the code. The key point is that because the size of both the code and this so-called static area does not change during execution, these areas, unlike the next two so-called dynamic areas, do not need a expansion region. The stack is used for memory that looks like a stack. It is organized in a recording of activation that is created as the procedure is called and is destroyed when the procedure comes out. It adjoins the area of unused memory, so it can grow easily. Typically, the stack is stored at the highest virtual addresses and grows downwards (towards smaller addresses). However, it is sometimes easier in describing activation records and using them to pretend that addresses are increasing (so the increments are positive). We'll discuss the stack in some detail below. A pile is used for data whose lifespan is not easy to describe. This data is distributed by the program itself, usually either through a language design, such as a new one, or by calling a library function, such as malloc. It is located either by the other you have performed the application, such as calling for free, or automatically by the system through garbage collection. We're going to have a little to say about the heap. 7.1.1: Static vs. dynamic storage distribution A lot (often most) data cannot be statically distributed. Either its size is not known at the time of compilation, or its lifespan is only a subset of subsms Perform. Early versions of Fortran used only static data. This required each array to have a permanent size specified in the program. Another consequence of supporting only static distribution was that recursion was banned (otherwise the compiler could not say how many versions of the variable would be needed). Modern languages, including new versions of Fortran, support both static and dynamic memory distribution. The advantage of maintaining a dynamic storage distribution is the increased flexibility and efficiency of storage (instead of declaring an array sufficient for the largest data set in size; just highlight what you need). The advantage of static storage distribution is that it avoids the cost of the distribution/deal time and can allow faster code sequences to link to the data. An error (unfortunately too common) is a so-called memory leak in which a long-term re-programme highlights memory that it doesn't delete, even after it can no longer be referenced. To avoid memory leaks and facilitate programming, several language programming systems use automatic garbage collection. This means that the run time system itself determines when the data can no longer be referenced and automatically deallocates it. 7.2: The Distribution of Space Stacks Scheme, to be presented, achieves the following goals. Memory is shared by procedural calls that have disparate durations. Note that we are not able to identify disunity simply by studying the program itself (due to data-dependent branches among other issues). The relative address of each (visible) non-local variable is constant throughout each execution of the procedure. Please note that during this exercise, the procedure may trigger other procedures. 7.2.1: Trees Activation Recall Fibonacci sequence 1.1,2,3,5,8, ... f(1) f(2) and, for n'gt;2, f(n)f (n-1) f (n-2). Let's look at feature calls that are the result of a major program called f/5. Surrounding a more general pseudocode that calculates (very inefficiently) the first 10 fibonacci numbers, we show calls and returns that are the result of the main call f(5). On the left they are shown linearly, and on the right we show them in the form of a tree. The latter is sometimes referred to as an activation tree or called a tree. The system starts from the main int a.10; Enter f(5) int main ()) enter f(4) int i; Enter f(3) for (i'0; i'lt;10; i') enter f(2) a'i'i f(i); Exit f(2) - enter f(1) - exit f(1) int f (int n) - exit f(3), if (n'lt;3) return 1; Enter f(2) return f (n-1)) ; Exit f(2) - exit f(4) enter f(3) enter f(2) exit f(2) enter f(1) f(1) Exit f (3) main ends We can make the following comments about these call procedures. If activation p triggers q, the activation p is not terminated before the activation q.The order of activations (procedural calls) corresponds to the pre-order bypass of the call call the order of deactivagy (return of procedure) corresponds to the postal bypass of the call tree. The Euler-tour order records both calls and returns. When performing the N tree activation node, the current live activations are those corresponding to N and its ancestors in the tree. These live activations were triggered in the manner given from root to N on the tree, and returns will occur in reverse order. Homework: 1, 2. 7.2.2: Activation Records (ARs) The information required for each procedure call is stored in a run time data structure called activation record (AR) or frame. The frames are stored in a stack called a control stack. Please note that this is a memory used by a program, not a compiler. The compiler's task is to create code that first receives the memory it needs, and secondly, refers to the data stored in the ARs. At any given time, the number of frames in the stack is the current depth of procedural calls. For example, in the fibonacci performed above, when f(4) is active, there are three activation entries in the control stack. ARs vary depending on the language and implementation of the compiler. Typical components are described below and are shown on the right. On the diagrams, the stack grows down the page. Arguments (sometimes called actual parameters). The first few arguments are often placed in registers. Returned value. It is also often instead placed in the register (if it is scalar). Link management. These pointers connect ARs, pointing from AR-called routing to the caller's AR. Link to access. This link is used to refer to non-local variables in languages with nested procedures, an interesting task that we discuss in some detail below. Saved status from the caller, which usually includes return address and machine registers. Register values are restored when management is returned to the caller. Local data to activate the procedure. Temporary. For example, think of the time-pace created during the expression evaluation. Often they can be held in machine registers. When this is not possible, for example, when there are more time than registers, a temporary zone is used. In fact we will see in Chapter 8 that only living temporary have an attitude. The diagram on the right shows (part) of the control stack for the example of fibonacci at three points during the run. Solid lines separate ARs; dotted line of individual components in AR. In the upper left, we have an initial state. We show a global variable, although it is not in the activation record. Instead, it statically stands out before the program starts (remember that the stack and heap are distributed dynamically). Also shown is the activation record for the main, which contains storage for the local i variable. towards the end Below the original state, we see the following state, where the main is called f(1) and there are two activation records, one for the main and one for f. Record activation for f contains space for the n argument as well as for the returned value. Recall that the arguments and the value of the return stand out closer to the beginning of the AP. There are no local variables in f. In the far right is a later state in execution, when f(4) was caused by the main and, in turn, called f(2). There are three activation records, one for the main and two for f. It is these multiple activations for f allow recursive execution. There are two places for n and two for return value. 7.2.3: Call a sequence of calls, performed when one procedure (caller) calls another (call), highlights the activation record (AR) in the stack and fills the fields. Some of this work is done by the subscriber; the rest of the conscript. Although the work is common, AR is called an AR conscript. Because the procedure, called, is defined in one place, but is usually called from many places, we expect to find more instances of the caller activation code than the call activation code. Thus, it is reasonable, for all that to be equal, to appoint as much work as possible to the conscript. Although details vary between implementations, the following principle often follows: the values calculated by the subscriber are placed in front of any unknown size elements. Thus, they can be referenced by the subscriber using fixed offsets. One possible mechanism is the following. Place the values calculated by the caller at the beginning of the activation record, i.e. next to the caller's AR. The number of arguments may not be the same for different calls of the same function (so-called varargs, such as printf() in C). However (compiler) subscriber knows how many arguments there are on this call site so where pink calls are blue, compilers know how far the return value is from the beginning of the blue AR. Since this beginning of blue AR is the end of the pink AR (or is one place further depending on how you feel), the subscriber knows (but only during time time time time time) the location of the return value from their own stack pointer (sp, see below). Fixed-length items are placed next to each other. Their size is known to the caller and caller during the compilation. Examples of fixed-length items include links and saved status. Finally, there are elements highlighted by a conscript whose size is known only during the time of time, such as arrays, the size of which depends on the parameters. The sp stack fits comfortably, as shown in the diagram. Notice the three implications of this choice. Temporary and local data is actually above the stack. It would seem even more surprising if used the terminology of the book, which top_sp. Fixed length data may refer to fixed biases (known generator) from sp. The caller knows the distance from the start of the caller's AR to sp and therefore can set the sp when calling. The image above illustrates the situation when the pink procedure (caller) causes a blue procedure (call). AR Blue is also shown. It's called AR Blue because its lifespan coincides with Blue's, even though the responsibility for this single AR is shared by both procedures. The picture is just an approximation: For example, the returned value is actually Blue's responsibility, although space may well be highlighted by Pink. Naturally, the returned value is only related to functions, not procedures. In addition, some of the saved status, such as the old sp, is retained by Pink. The image on the right shows what happens when Blue, callee, calls the green procedure itself and thus the blue is also the caller. You can see that Blue's responsibility includes part of his AR as well as part of Green. Actions during the call Next action takes place during the call. The caller begins the process of creating an AR call by evaluating arguments and placing them in the AR caller. (I use the arguments for the caller, the parameters for the caller.) The caller stores a return address and (soon-to-be-updated) sp into the conscript's AR. The SP increments subscriber so that instead of pointing to his AR, he points to the appropriate point in the AR conscript. The caller retains registers and other (system-dependent) information. The conscript allocates and initiates his local data. The conscript begins to perform. Actions during the return, when the procedure returns, the following actions are performed by the inductee, essentially cancelling the consequences of the call sequence. The inductee keeps the return value (if the caller is a function). The caller restores sp and registers. The conscript jumps to the return address. Note that varargs are supported. Also note that the values written during the call sequence are not erased and the space is clearly not returning. Instead, sp is restored, and if and when the caller makes another call, the space will be reused. 7.2.4: Variable stack data there are two tastes of variable length data. Data obtained by malloc/new, it is difficult to determine life expectancy and are stored in a pile instead of a stack. Data, such as arrays with boundaries defined by parameters, is still stacked as in their lives (if A causes B, these A variables are distributed before and released after the corresponding B variables). This is the second flavor that we want to highlight on the stack. The goal is for the inemen to be able to access these arrays through addresses identified during the compilation, even if the size of the arrays is not known until the program is called, and often differs from one call to the next (even if two calls match correspond Source statement). The solution is to leave room for pointers on arrays in AR. These pointers are fixed sizes and can thus be accessed by offsets known during compilation. When the procedure is called and the dimensions are known, the pointers are filled and the space is allocated. The difficulty of storing these variable-sized elements in the stack is that it is no longer obvious where the real top of the stack is in relation to sp. Consequently, another pointer (we could call it a real top-in-the-stack) is also saved. This pointer tells the participant where to start the new AR if the caller makes the call himself. Alternative is probably a more common name for (stack-pointer, real top-of-the-stack-pointer) pair (stack-pointer, Frame Pointer) Homework: 4. 7.3: Access to non-local stack data As we'll see the possibility of a P procedure to access data announced outside of P (or declared globally outside of all procedures or, especially, those that are declared within another procedure. 7.3.1: Access to data without nested procedures in languages such as standard C without nested procedures. , visible names are either local for this procedure or are announced globally. For global names, the address is known statically during compilation, provided that there is only one source file. If there are several original files, the binder knows. In any case, no link to the activation record is required; addresses are known before execution. For names local for the current procedure, the address required in AR at known on compilation time is constantly compensated from sp. In the case of variable arrays, the constant shift refers to the actual storage pointer. 7.3.2: There are problems with the inserted procedures with the procedures invested. Say g nested inside f. So g can refer to the names announced in f. These names refer to objects in AR for f. The difficulty is to find that AR when the g is performed. We cannot say during the compilation where (the last) AR for f will be relative to the current AR for g, since the dynamically defined (i.e. unknown compilation time) number of procedures could have been caused in the middle. There's an example in the next section. in which g refers to x, which is announced directly in the outer area (main), but the AR is on 2 away because f was called between them. (In this example, you can tell during the compilation what was caused in what order, but with a more complex data-dependent program, it's not possible.) 7.3.3: Language with nested procedural declarations The book states (correctly) that C has no nested procedures, so introduces ML, which makes (and is quite smooth). However, many of you don't know ML and I haven't used it. Fortunately, the general C is to resolve the attachments. In particular, gcc supports the investment procedures. Who to who my memory I made up and ran the next program. #include <stdio.h> int main (int argc, char s argv)) - int x 10; int g (int y) - int z y x'y; return z; - int f (int y) - return g(2'y); The program compiles without errors and printed the correct answer 30. In this way, we can use C (actually GCC C) as a language to illustrate the enclosed procedural declarations. Note: Many consider this expansion of gcc (or its implementation) evil. For example, look here, here or here. 7.3.4: The nesting procedures of the depth of the outer surface have a nesting depth of 1. Other treatments have a nesting depth of 1 more than the depth of the nesting directly external procedure. In the example above, the main one has a nesting depth of 1; both f and g have a nesting depth of 2. 7.3.5: Accessing the AR link for the nested procedure contains an access link that indicates ar's latest activation immediately external procedure. Thus, in the above example, the access link for f and the access link for g will be each point to the AR activation main. Then, when g links x, defined basically, the activation record for the main can be found by following the access link to AR for f. Since the f is embedded in the main, they are compiled together, so that once AR is determined, the same methods can be used for both local f variables. Everything can be determined during the compilation, as there are no branches that depend on the data. This is only one AR for basic during the entire execution, since the main is not (directly or indirectly) recursive and there is only one AR for each of the f and g. However the technique is pretty common. For the P procedure to access a name defined in a three-part area, i.e. a unique outer area whose nesting depth is 3 times less than P, you follow the access links three times. Make sure you understand why n-y external volume, but not n-y internal volume is unique) The remaining question is: How are access links supported?. 7.3.6: Manipulating access links Assume that there are no procedure parameters. We also assume that the entire program is designed at once. For a few files, the main problems are related to the link, which is not covered in this course. I cover it with it a little bit in the OS know. Without the parameters of the procedure, the compiler knows the name of the procedure called and, therefore, its depth of attachment. The compiler always knows the depth of the caller's attachment. Let the caller be the F procedure and let it be called the G procedure, so that we have the F call G. Let N (proc) be the depth of the nesting procedure proc. We distinguish between two cases. N(G)>N(F). The only way G can be seen in F and have a greater nesting depth for G, which will be announced directly inside the F. Then when compiling a call from F to G we just install G access to point to AR F. In this case, the access link is the same as the zlt;/stdio.h'gt; control link. N(G) ≤N(F). This includes the F'G case, i.e. a direct recursive call. For G to be visible in F, there must be a different P procedure, applied as F and G, with G directly within P. That is, we have the following situation where k'N(G)-N (F)≥0 P()) P1 () - P2 () ..... Pk () - F ()... G(); ...} } ... } Our goal when creating AR for G when calling from F is to set an access link to point to AR for P. Please note that the entire structure in the skeleton code is visible to the compiler. The current (at the time of call) AR is one for F, and if we follow the access links k once we get a pointer to AR for P, which we can put in an access link for the created AR for G. The above works perfectly when the F is nested (perhaps deep) inside the G. This is the picture above, but P1 is G. When we get the gcc code that I've shown before, as well as a direct case of repetition where G-F. I don't know why the book separates the K'0 case, especially since the previous edition didn't. The problem is that if f causes g with parameter h (or pointer to h in C-speak) and then g causes this option (i.e. causes h), g may not know the context h. Solution for f pass g pair (h, access connection h) instead of just passing h. Naturally, this makes the compiler, the programmer is not aware of the access links. 7.3.8: Displays the basic idea is to replace a linked list of access links with an array of direct pointers. In theory, access links can form long chains (in practice, the nesting depth rarely exceeds a dozen or so). The display represents one array in which the i record indicates the most recent (highest on the stack) AR depth i. 7.4: Heap control Almost the entire section is covered in the OS class. 7.4.1: Memory manager covered by OS. 7.4.2: A hierarchical memory of a computer covered with architecture. 7.4.3: Localization in PROGRAMS covered by the OS. 7.4.4: Reducing (external) fragmentation covered by the OS. 7.4.5: Data from the manual stack of distribution requests are automatically determined when the defining procedure is returned. What to do with the pile of data highlighted by new/malloc? The manual method is to require the programmer to explicitly look at the data. There are two problems. Memory leaks. The programmer forgets the deallocate. Cycle highlight X use X forget the deallocate X end of the cycle As this program continues to work it will require more and more memory even if the actual usage does not increase significantly. Hanging links. The programmer forgets that they made a deallocate. Highlight X to use X deallocate X 100,000 lines of code without using X use X Both can be disastrous and motivate the next topic that is covered in programming language courses. 7.5: Introduction to Garbage Collection System Detects Data may not be available (there are no direct or indirect references) and data automatically. Covered in programming languages. 7.5.1: Design Goals for Garbage Collectors 7.5.2: Accessibility 7.5.3: Background Counting of Garbage Collectors 7.6: Introduction to the Tracing 7 collection. 6.1: Basic Mark-and-Collector Sweep 7.6.2:Basic Abstraction 7.6.3: Mark-and-Sweep Optimization 7.6.4.4: Mark-and-Compact Garbage Collectors 7.6.5: Collectors Copy 7.6.6: Cost Comparison 7.7: Short Pause Garbage Collection 7.7.1: Incremental Garbage Collection 7.7.2: Additional Availability Analysis 7.7.3: Partial Collection Of Basics 7.7.7.4: Generation Garbage Collection 7.7.7.7. : Train Algorithm 7.8: Advanced Topics in Garbage Collection 7.8.1: Parallel and Parallel Garbage Collection 7.8.2: Partial Object Move 7.8.3: Conservative Collection for Unsafe Languages 7.8.4: Weak Links Links boolean expression in compiler design slideshare. boolean expression in compiler design notes. boolean expression in compiler design tutorialspoint. boolean expression in compiler design ppt. boolean expression in compiler design geeksforgeeks. boolean expression in compiler design pdf. translation of boolean expression in compiler design