California State University, Fullerton
Dept of Computer Science

CPSC 323 Compilers & Languages (Spring 2024)

ASSIGNMENT 3

Please answer the questions appropriately, and if you believe a diagram is required, draw one.

If you want to solve the problems, do so while documenting your steps.

NOTE: Please avoid writing single-word responses or just the single solution to a problem. Please submit a PDF document.

---

1. Explain Three-Address Code (TAC). Convert a = (c * b) + (c * d) into three address code.

Three Address Code is a type of intermediate code which is easy to generate and can be converted to machine code, it makes use of at most three addresses and done operator to represenat an expression, and vaue computed at every instrictipn and is stored in a temporart variable, a maximum of three addresses or variables are allowed,the program breaks the parse tree down into elementary statements, where each statement has no more than 3 addresses(variable),it is also a low level intermediate representation used in compilers and interpreters, its designed to be easy for machines to work with while still being humanely readable, each instruction in TAC typically involves three operands, three or less can work, but these operands can be variables, contants, or memory locations, the three address code breaks down the original expression into similair steps that a compiler or interpreter can handle efficeintly, each step performs on a signle operation, making it easier to optimize and translate into machine code later on, three address code is used as an intermediate representation of code during code generation of the phase of the compilation process, the three address code allows the compiler to generate code that is specific to the target platform, which ensures that the generated code is correct and efficient, there are three representations of the three address code, quadruple, triples, and indirect triples, the quadrupe is a structure which consists of 4 fields and a result, the result is used to store the result of the expression, the triples doesnt make use of any extra temporary variable to represent a single operation, instead when a reference to another triple value is needed, the pointer to that triple is used, the indirect triple is a represeentation that makes use of pointer to listing to all references to comptations which is made seperately and stored, the TAC is also a sort of intermediate code that is simple to create and convert to machine code, it can only define an expression with three addresses and one operator, the three address code helps detimes the sequence in which operations are actioned by the compiler, the three address code is considered as an intermediate code and utilised by omptimising compilers, the three address code is a given expression broken down into multiple guidlines, which translates it to assembly language

convert the expression a=(cb)+(cd) into Three-Address Code:

```
t1 = (c * b)
t2 = (c * d)
a = t1 + t2
```

2. Convert the expression into Postfix notation:
   A + B * C / D – E

B * C = BC*
a *b = ab*

Step 1. A+ BC* / D - E

(BC*) / D = (BC*) D /
(a) / b = (a)b/

Step 2. A + BC* D / - E

A + (BC* D /) = A (BC* D /)+
a + (b) = a b+

Step 3. A BC* D /+ - E

(A BC* D /+) - E = A BC* D /+ E-
a - b = ab-

Step 4. A BC* D /+ E-

Final answer = A BC* D /+ E-

3. What is an Abstract Syntax Tree (AST)? How is an AST different from a parse tree?

An Abstract Syntax Tree is a form of parse tree, where all unnecessary parsing info is removed, all the non terminals in the tree are removed, it is also a tree like data structure used in compilers and languages to represent the structure of code after it has been parsed, it captures the hierarchical structure and relationship between different elements of the code, such as expressions, statements, and declarations, the ASTs are commonly used in compilers, interpreters, and static analysis tools. The AST is different from the parse tree in many ways, the parse tree represents a syntactic structure of code based on the grammar rules of the programming language, it includes all the details of the parsing process, such as tokens, grammar rules and operator precedence, whereas, the AST, focuses on essential structure of code, in which it gets rid of details or ignore details like parenthese, commas, and semicolons, it represents the logical structure of the code in a more simplified and meaningful way for analysis and interpretation, parse trees contains nodes for every syntactic element specified by the grammar including terminals(tokens) and non-terminals(grammar rules), wheras the AST typically omits nodes that arent essential for understanding the code's logic, such as the non terminals, punctuation, and grouping symbols, it focuses on nodes that represent meaningful construct like expressions, statements, and declarations, parse trees are primary used during the parsing phase of the compiler to ensure that the syntax is correct and generates an AST,wheras the AST is then used in the subsequent phase of compilation, such as semantic analysis, optimization, and code generation, it serves a the basis for undertanding the codes semantics and generating executable code. The parse tree can be larger and more complext than AST because it includes detailed syntactic information, the AST is generally smaller and simpler than parse trees because it focuses on the structure needed for anlysis and execution , the parse tree captures detailed syntactic strucure of code based on grammar rules, an AST abstracts away uneccesary details and focused on the logical structure, which makes it more suitable for the other tasks in the compilers, the AST is also a representation of the structure of a proramming langauge, it's a tree like strucrue composed of nodes, which reperesents a langauge element like a variable, operator, or keyword, its essentially a simplified version of a parse tree, the nodes of an AST are abstracted away from the syntax of the language making it easier to learn about the structure, the parse tree is a tree like structure that represents the syntactic structure of the language, it's composed of nodes, each of which corresponds to a langauge element, unlike the abstarct syntax tree, the parse tree arent abstracted away from the details of the language syntax, an AST is composed of symbols like an operator, identifiers, and keywords, wheras a parse tree is composed of tokens, such as a terminal, non terminal symbols, an AST is presented using a standard tree strcuture with each node representing a particular symbol or rule, whereas the parse tree is represented using a graph strucruee with each node representing a particular token or rule.An AST is used to analyze the source code and optimize it for compilation, whereas a parse tree is used to check the syntax correctness of the code,An AST is more expressive than a parse tree, as it provides more information about the source code., parse tree is less expressive than an AST only provides basic information about the source code. A parse tree only provides basic information about the source code, an AST is more efficient than a parse tree, whereas a parse tree is less efficient than an AST, it only provides basic information.An AST is easier to comprehend

4. How does constant folding contribute to improved code performance?

Constant folding makes the computer take less computation power, constant folding is the pre computation of the results at compile time, it improves the execution time of the program, it reduces overall memory requirement, it helps avoid redundant computations in the code, which makes it more efficient, it also reduces power consumption, it makes the hardware usage more efficient it makes the code cleaner and shorter, it is also a compiler optimization technique that evaluates constant expressions at run time rathar than compile time, it identifies expressions that involve only constants and computing their results during compilation, by computing constant expressions at compile time, constant folding reduces the amount of work that the program needs to do at runtime, this can lead to a faster execution because the program doesnt how to perform these uneccesary computations repeateadly, since constant folding replaces constant expressions with their computed values, it can reduce the amount of memory needed to store immediate results or temporary variables related to these expresions, this optimization can be particulary beneficial in memory contrained enviroments, constant folding can identiy and eliminate redudant computations of constant expressions, if a function is called multiple times with the same parameters constant folding can compute the function's result once and reuse it, reducing redundant computations. Constant folding simplifies code by replacing complex constant expressions with simpler values.This not only improves readability but also makes the code easier for other optimizations to analyze and optimize further, it helps in producing more efficent and optimized code by reducing runtime overhesad, optimizing memory usage, eliminating reduant computations and simplies the code structure. It saves exeuction time, they are calculated beforehand to save execution time. It also reduces the code sizes as well. Constant folding is used to decrease the execution time, it optimizes the code, it reduces the lines of code, it also helps in managing efficient memory, it also silently improves performance, there is also faster execution, in which it eliminates redunant calculations.Constant folding reduces the runtime computational load and can lead to more efficient code. Constant fold recognizes and evaluates constant expressions at compile time rathar than run time

5. What is dead code elimination. Give an example.

Dead code ellimination is a complier optimization technique that identifies and removes code that is never executed or is unreachable, which then improves the efficiency of the compiled program by reducing its size and eliminating unecessary computation, dead code ellimination removes dead code, it also refers to section of code within a program that is never executed during runtime and has no impact on the program, it is essential for improving program efficiency, reducing complexity, an enhancing maintanability,eliminates code that is never executed or does nothing useful, by removing dead codem unecessary computations and memory usage are eliminated resulting in faster and more efficient program execution, there is also improved maintainability, in which dead code complicates the understanding and maintnance of software systems, by eliminiating it, developers can focus on relevant code, improve code readibility, and bug fixes, it also removes code that is never used, it also removes the code that doesnt affect the program results

Examples of dead code

```
if ( 5 < 3) ..... /* never can be executed */

X =5; /* assigned but never used */
...

X =7; /* reassignment without the using previous assignment */
```

6. How can reduction in strength code optimization be applied to the following example? Justify.

```
#include <stdio.h>

void doubleArrayElements(int array[], int n) {
    for (int i = 0; i < n; i++) {
        array[i] = array[i] * 2;
    }
}
```

Strength reduction is a code optimization technique that replaces expensive and extensive operastions that take more computation power, instead it would take less computation power to improve performance, in this example instead of multiplying the array, we replace the multiplication operator with the plus operator, we replace "*" with "+", becuase the multiplication operator(*) takes more computational power, becuase you add it again and again, which causes to take more computational power and its repetitive, addition only does it once per iteration, while multiplication does it multiple times per iteration

Rewritten version in strength code optimization

```
#include <stdio.h>

void doubleArrayElements(int array[], int n) {
    for (int i = 0; i < n; i++) {
        array[i] = array[i] + 2;
    }
}
```

In this example we replaced * with +, because * takes more computation power, and memory and it repeats the same thing again and again, the plus operator does it once without repetition, and has less computation power

7. Examine this code snippet:

```
#include <stdio.h>

int sum_array(int arr[], int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    return sum;
}
```

In this example, which expression is loop invariant expression? How does recognizing loop invariant expressions contribute to reducing redundant computations?

Loop invariant expression is when the code inside the loop isn't dependent on or affected by the loop, loop invariant is a condition that is true at the beginning and end of every iteration of a loop.Recognizing loop invariant expressions is important for optimizing code and reducing redundant computations, a loop invariant is a condition that remains hroughout the execution of a loop. In this case, the loop invariant is that sum accumulates the sum of elements in the array up to the current index i. Since the code in the loop is dependent on the loop, "i" in "sum += arr[i];" is dependent on the for loop confiton, "for (int i = 0; i < n; i++)", the expression is something that doesn't change with the loop even if the loop is running, there is no expression that is loop invariant in the given code, because there is only one expression in the for loop and that expression is dependent on the for loop , the expression inside the for loop changes inside the loop within the loop, and the sum also changes, the loop is running fine and the expression or sum changes while the loop is running

8. How does peephole optimization contribute to code size reduction? Explain with an example.

Peeophole optimization reduces code redunancy, in which unnecessary code or instructions are elliminated, if there are two consecutive insturctions that have the same effet, the compiler design peeophole optimatization can remove one without affecting the program's bhehavior,Peephole optimization eliminates unreachable code by removing and recognizing instruction sequences that can never be executed, it is also a technique used by compilers to improve the efficiency of generated code by anaylyzing a small window of instructions and making local optimizations, the primary objective is code size reduction achieved by eliiminating redundant or inefficient code, peeophole optimization can identify and remove redudant code sequences that don't contribute to theprogram's functionality, if there are consecutive instructions that set a variable to the same value, one of those instructions can be eliminated.

1. Reducing Instruction Redundancy:
Before optimization:

```
mov eax, 0      ; Load 0 into register eax
add eax, 5      ; Add 5 to eax
mov ebx, eax    ; Move eax to ebx
```

After optimization:

```
mov eax, 0      ; Load 0 into register eax
add eax, 5      ; Add 5 to eax
```

In the example peephole optimization identified each of the four insturctions thaat set the register to 0,the redudant move insturction that copies the value of eax to ebx immediately after eax is modified, they can identify this patter and eliminate uncessary move insturction, after peeophole optimization, the code stil achieves the same result but with one less instruction, leading to reduced code size

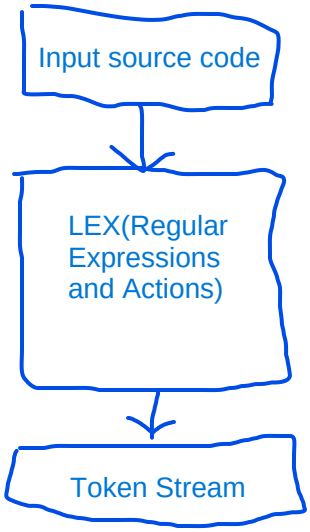
9. What is live analysis in the context of compiler optimization?

Live analysis is a crucial technique used to deterine the lifetime and usage of variables within a prgram.This analysis determines which variables are live(in use) at any point in the program.It helps the compiler decide when it's necessary to keep a variable's value in a register to avoid unnecessary memory accesses.Variables that are live are those that will be used later in the program.A variable is said to be “Live” if it is going to be used again, it also consists of a specified technique that is implemented to optimize register space allocation, for a given piece of code and facilitate the procedure for dead code elimination, as any machine has a limited number of registers to hold a variable or data which is being used or manipulated, there also exists a need to balance out efficient allocation of memory to reduce cost and enable the machine to handle complex code and a considerable amount of variables at the same time. The procedure is carried out during the compilation of an input code by a compiler itself. There is a live variable that is live at any instant of time during the process of compilation of a program if its value is used to process a computation as the ealuation of an arithmetic operationn at that instnt or ir holds a value that'll be used in the future whithout redifining it, there is also an liveness algorithm in which is used to evaluate the live variables at each step, if there are any live variables during the execution of a statement the algorithm adds the evaluation to the final result and repeats the same procedurem the out is furuthur used to analyze live range of variables, it also reffers to the process of anlyzing the lifetime of variables within a program, the analysis determines which variables are live at any given point during the execution of the program.This analysis determines which variables are live at any given point during program execution. A variable is considered live if its value will be used in the future.Live analysis is crucial for optimizing code because it helps compilers eliminate unnecessary computations and memory operations.By identifying variables that are no longer live, the compiler can free up resources such as registers and memory locations, leading to more efficient code generation.One common optimization based on live analysis is dead code elimination, where the compiler removes code that computes values that are never used. This optimization can significantly reduce the size and complexity of generated code, improving overall program performance.

10. What is the difference between LEX and YACC. Explain with the help of block diagrams.

LEX is a program (generator) that generates lexical analyzers or lexers, (widely used on Unix), Window version is called Flex, it reads the input stream (specifying the lexical analyzer) and outputs source code implementing the lexical analyzer in the C programming language, it includes token definition which includes elementary expressions. This section describes, what to do with it once a token is recognized (actions). THere's a section where a user can define own functions that can be used I previous sections. YACC is a program (generator) that generates syntax analyzers or parsers,Yacc reads the grammar and generate C code for a parser, Grammars written in Backus Naur Form (BNF), BNF grammar used to express context-free languages,This known as bottom-up or shift-reduce parsing,it define or declare variables, tokens etc, Rules are defined similar to the BNF notation, and Define own functions to be used in the Yacc source, They are typically used together: you Lex the string input, and YACC the tokenized input provided by Lex.LEX is a tool used to generate lexical analyzers or tokenizers. It takes a set of regular expressions and corresponding actions and generates code for a lexer. The lexer breaks down input into tokens based on the regular expressions provided.YACC is a tool used to generate parsers. It takes a set of grammar rules and corresponding actions and generates code for a parser. The parser processes the token stream generated by the lexer and builds a parse tree or performs semantic actions based on the grammar rules. YACC depends on LEX for the tokens, becuase LEX generates a token stream, whereas YACC generates a parse tree and semantic actions from the token stream.

LEX



YACC