****

**Final Report**

**CSE 421: Compiler Design**

Name: Shihab Muhtasim

ID: 21301610

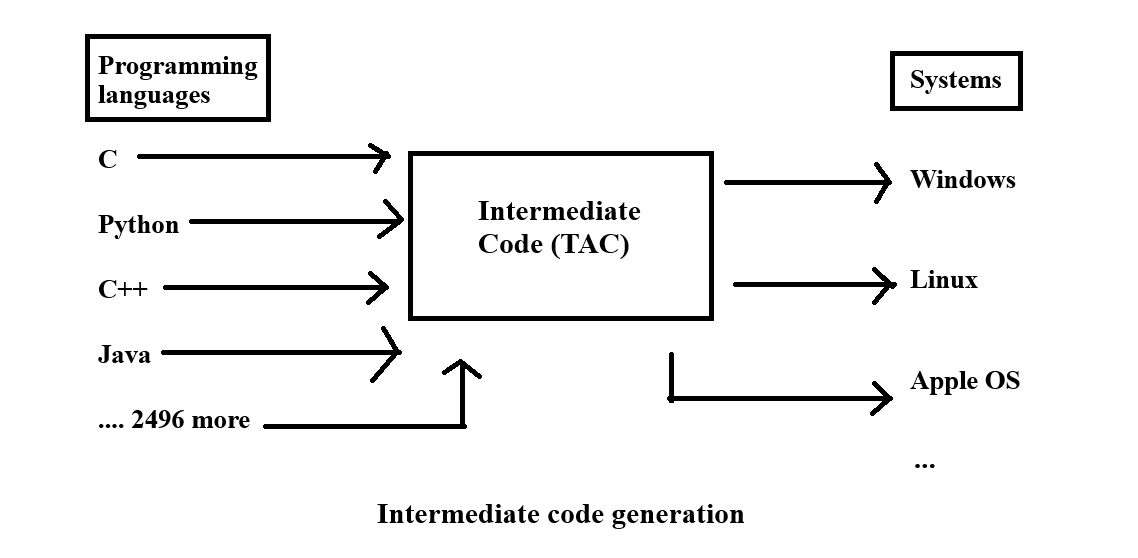
Sec: 02

Summer 24

**Intermediate code generation and Three address code**

Intermediate code generation is a significant part of the compiler that makes compiling much more efficient than it would be without this process. There are several high to low-level languages for programming in today's world. After writing the code using a programming language, the compiler converts it according to the operating system of the corresponding device the operator is using. Hence, for each language and its combination with the system, there needs to be a separate compiler to compile its corresponding program.

The issue arises when we talk about the mass number of languages and the number of compilers needed for that. This creates a serious problem. For instance, if there are 5 operating systems and approximately 2500 programming languages, for each language there would be a need for 5 compilers making a total of 2500 \* 5= 12500.



In such a situation, intermediate code generation reduces the number of compilers that we will need. Intermediate code generation is a common process between the systems and the programming language. So, each language will need a separate compiler to compile it to an Intermediate code like three-address code. Then, that common Intermediate code will be converted to the corresponding operating system’s understandable language using one separate compiler for each of the systems. Hence, now the total need for compilers to compile from 2500 programming languages to Intermediate code is 2500 and the compilers needed to compile that intermediate code for the systems are 5 for 5 systems making the total of 2505 when it would’ve been 12500 if not for Intermediate code generation.

Three address code or TAC is an intermediate code that can handle complex codes and properly compile those to the intermediate form. In TAC, variables are stored in addresses and there can be a maximum of 3 addresses in one line. Each instruction can have at most one operator on the right-hand side keeping 2 operands on that side. One benefit is that TAC can easily identify the registers used to store data.

There are many different types of notations in TAC such as i) name or variables to store data (a,b,c,d); ii) temporary variables to help keep track of the values and make sure only 3 addresses are used; iii) constants like integer or float values. Moreover, there can be conditional or unconditional jump operations to help with flow control. Additionally, assign operations can be used to copy or store information. There are if and goto conditions that act with a label address to control the flow of the code. Moreover, array operations are handled using the address of each index calculated by using the index number and base address. If the array is stored in the row-major method, then the location of an element can be found using base + (i - low) \* w where i is the index of the array.

Three address codes can be stored in 4 field vectors called Quadruples, or in 3 field ones called Triples, or doubles with 2 fields. It can be stored in indirect triples that use the physical logical addresses along with the 3 fields of triples.

**Code Generation process in compiler**

There are two parts of a compiler which are front end and back. The parser is part of the front-end process that can keep static checks as well as generate intermediate codes in the process of parsing using semantic rules and parsing grammers.

An algorithm that includes some variable declarations including an array declaration, one while loop, and one if/else block is of linear search. This can be translated into a proper three-address code sequence using the grammar and SDT definitions provided here [grammers.pdf](https://drive.google.com/file/d/1fEm3IQTZf91lNyKBseAecMtSY49amJ_E/view?usp=sharing)

| **High-level code** | **Three Address Code** |
| --- | --- |
| **Linear search:**  // Define variables  int a[4];  Int n;  Int i;  Int result;  // Declare variables and array  a[0] = 1;  a[1] = 2;  a[2] = 3;  a[3] = 4;  n = 3;  i = 0;  //While loop  while (i < 4) {  //if/else block  if (a[i] == n) {  result = i;  }  else{  i=i+1;  }  } | //Declare variables and arrays  t1 = 0 \* 4  a [t1] = 1  L1 :  t2 = 1 \* 4  a [t2] = 2  L3 :  t3 = 2 \* 4  a [t3] = 7  L2 :  t4 = 3 \* 4  a [t4] = 9  L12 :  n = 3  L11 :  i = 0  L9 :  //while loop  begin :  if i < 4 goto L7  goto L8  L7 :  t5 = i \* 4  t6 = a [t5]  //if /else block  if t6 == n goto L4  go to L5  L4 :  result = i  goto L6  L5 :  t7 = i + 1  i = t7  L6 :  go to begin  L8:  L 14: |

Using SDT definitions and grammers the linear search algorithm has been compiled into three address codes. Firstly, defining the variables have been parsed by the parser. However, in three address codes, we do not need to define variables so there is no code for that. After that, the defined values have been declared/ initialized with a value. For example, t1 calculates the index location of the first index of the array. Then it accesses the index 0 of array a and assigns it with the value 1 using instruction a [t1]=1. Similarly, TAC declares variables like n=3 as n is the element that we have to find from the array.

Parser uses a combination of grammers to compile while loop and if else block along with the conditions that need to be maintained. For example, B->E1 rel E2 grammar has been used for condition statement creation and grammar like “S-> if (B) S1 else S2” and “while (B) S1” has been used to check for condition accuracy and to maintain the flow of code using methods like goto or declaring a label name at the end of if condition.

In the TAC generated, the “begin” label is the start of while loop, and then as long as “i<4” code will execute block L7, otherwise it’ll go to label L8 which is at the end of the code, exiting the while loop. After the execution of block L7 which prepares the array’s next i th index for checking, the code will execute the if condition checking, and upon finding true it’ll execute L4. This label gives us our results. The break statement has not been used for simplicity but if it was used, after executing result=i in label L7 there would be a code to “goto L8” (exit\_label). Now, if the else condition is true then the code will execute L5 which just increments the i value by 1. After that code will go to begin and continue the while loop. This while loop will continue till the end of the array and iterate through all instances.

Parse Tree representation of TAC generation is given in the next section.

Modified [grammars](https://drive.google.com/file/d/1fEm3IQTZf91lNyKBseAecMtSY49amJ_E/view?usp=sharing) that have been used can be found in this link.

Link: [grammers.pdf](https://drive.google.com/file/d/1fEm3IQTZf91lNyKBseAecMtSY49amJ_E/view?usp=sharing)

For better quality of the parse tree, visit this link: [420 annotated tree main.pdf](https://drive.google.com/file/d/1iYxjS2hMAxGiuLyufKYxTjk1NAtEjfTG/view?usp=sharing)