COMPILER DESIGN

TEAM - 10

CS18B037 Swetha R, CS18B043 Shruti Priya, CS18B044 Sai Datta, CS18B045 Pranathi W

1. Introduction

The team has built a compiler for a custom language that resembles C, sans a few features. When given as input, a file containing code written in this language (refer language manual), it is lexed or tokenized and then parsed according to a Context-Free Grammar (discussed later). Semantic checks are done along with the parsing stage itself, and a custom intermediate code is generated. This is then converted to the MIPS assembly code, and the register allocation methods involved are discussed later in this report. This MIPS code, which is written to a file named result.asm, is capable of running on the tool QtSPIM. The compiler is capable of handling all basic features like variable declaration and initialisations, various kinds of expressions, if and else statements, while loops (including nested ones) along with input and print statements, non-recursive functions and multi-dimensional arrays.

This report describes the programming language and tools used by the team and the target code in Section 2. Section 3 delves into the detailed explanation of each component of the compiler, its implementation and the teams' intuition behind the choices made followed by a Flow Diagram in Section 4. Section 5 gives brief insights about how the source code is organised and how one can use the compiler and test with input code. The team's results of testing are displayed in Section 6 and the compiler's limitations are stated in the 7th Section. Section 8 and 9 contain concluding remarks and contributions by each team member.

2. Language and Tool Choices:

The entire source code is written in Python. Python was chosen because of the convenient data structures and code constructs it offers. Additionally, the PLY library has been used for its components ply.lex and ply.yacc, which are very similar in usage to the C-based lex and yacc tools.

The target language chosen is the MIPS assembly code that can be run using the simulator QtSPIM tool.

3. Major components of the Project:

a. Scanner/Lexer

Scanner is the first phase of the compiler which works as a lexer. This phase lexes the source code as a stream of characters, compares it with the list of provided tokens and reserved words and converts each token as a meaningful lexeme (i.e <token-name, attribute-value>) and this is passed to the parser.

b. Parser including Grammar

Parser takes tokens as an input produced by the scanner and generates a symbol table and the parsed output. This phase compares the token arrangement with the provided grammar and checks the syntax and semantics.

Parser Grammar

Grammar regarding statements and structure in a program:

S : prgm
prgm : prgm stmt
prgm : ϵ
stmt : funcdef
stmt : funccall SEMICOLON
stmt : declare
stmt : assign
stmt : ifstmt
stmt : whilestmt
stmt : printstmt
stmt2 : stmt2 stmtelt
stmt2 : ϵ
stmtelt : funccall SEMICOLON
stmtelt : declare
stmtelt : assign
stmtelt : ifstmt
stmtelt : whilestmt
stmtelt : printstmt
stmtelt : returnstmt
stmtelt : continuestmt
stmtelt : breakstmt

Grammar for function definition and arguments:

funcdef : FUNCTION type2 IDENTIFIER funcdefy LCB nulltypeargsx RCB LFB stmt2 RFB fundefexit
type2 : type
type2 : ϵ
funcdefy : ϵ
fundefexit : ϵ
nulltypeargsx : nulltypeargs
nulltypeargs : typeargs
nulltypeargs : ϵ
typeargs : typeargs SEPARATORS typearg
typeargs : typearg
typearg : type typeargval
typeargval : IDENTIFIER

Grammar for expressions and terms:

expr : expr OR andterm
expr : andterm
andterm : andterm AND equalterm
andterm : equalterm
equalterm : equalterm LOG relopterm
equalterm : relopterm

relopterm : arithterm
arithterm : arithterm ARITHOP multerm
arithterm : multerm
multerm : multerm MULTOP singleterm
multerm : singleterm
singleterm : IDENTIFIER
singleterm : prefix INTNUM
singleterm : prefix FLOATNUM
singleterm : CHARACTER
singleterm : LCB expr RCB
singleterm : arrayid
prefix : ARITHOP
prefix : ϵ

Grammar for assignment statements:
assign : lhs ASSIGN rhs SEMICOLON
lhs : IDENTIFIER
lhs : arrayid
rhs : inputstmt
rhs : expr
rhs : funccall
inputstmt : INPUT LCB type RCB
funccall : IDENTIFIER LCB nullargs RCB
nullargs : args
nullargs : ϵ
args : args SEPARATORS arg
args : arg
arg : IDENTIFIER
arg : prefix INTNUM
arg : prefix FLOATNUM
arg : CHARACTER
arg : arrayid

Grammar for if statement:
ifstmt : IF LCB expr RCB LFB ifbegin stmt2 RFB ifend elsepart
ifbegin : ϵ
ifend : ϵ
elsepart : ELSE LFB elsebegin stmt2 RFB elseend
elsebegin : ϵ
elseend : ϵ
elsepart : ϵ

Grammar for while statement:
whilestmt : WHILE LCB expr RCB LFB whilebegin stmt2 RFB whileend
whilebegin : ϵ

whileend : ϵ

Grammar for print statement:
printstmt : PRINT LCB printables RCB SEMICOLON
printables : printables SEPARATORS printable
printables : printable
printable : STRING
printable : IDENTIFIER
printable : arrayid

Grammar for return statement:
returnstmt : RETURN returnelt SEMICOLON
returnelt : expr
returnelt : ϵ

Grammar for break and continue statements:
breakstmt : BREAK SEMICOLON
continuestmt : CONTINUE SEMICOLON

Grammar for declarations:
declare : decbegin type vars SEMICOLON
decbegin : ϵ
type : FLOAT | INT
vars : var SEPARATORS vars
vars : var
var : IDENTIFIER val
var : arrayvar
arrayvar : arrayvar LSB INTNUM RSB
arrayvar : IDENTIFIER LSB INTNUM RSB
arrayid : arrayid1
arrayid1 : arrayid1 LSB index RSB
arrayid1 : IDENTIFIER LSB index RSB
index : INTNUM
index : IDENTIFIER
val : ASSIGN expr
val : ASSIGN inputstmt
val : ASSIGN funccall
val : ϵ

c. Symbol table

Symbol Table is a dictionary where we store the information about each of the parsed identifiers along with its scope, to use in later stages of the compiler.

The format to store information about each entry in the table:

{identifier: *Name* dimension: [],
size: 4,
type: *identifiertype*,
lineno: *linenum*,
start_addr: start_addr}

where,

- identifier : the Identifier name.
- dimension : in case of an array, it contains the dimensions of the array as a list else it is empty.
- size : Identifier size- in our case it is always 4 bytes (for both int and float)
- type : Type of the identifier, in our case can be either int or float.
- lineno : Declaration line number of the identifier.
- start_addr : start address of the identifier in memory.

The scopes have been maintained by creating a subtable inside the table. Whenever we enter a new scope, we create a subtable inside the previous table entry and store the new scope declarations in this subtable.

The format for a subtable entry in the table is:

{lineno: *linenum*,
subtable: [{ identifier: *Name,*
dimension: [],
size: 4,
type: *identifiertype*,
lineno: *linenum*,
start_addr: start_addr}]
}

where,

- lineno : stores the starting line number of the new scope (all entries inside will have a line number greater than this).
- subtable : it is a list that stores the entities declared inside the new scope.

* The start and end of the scope has been determined by seeing the position of **{** and **}** respectively.

d. <u>Semantic analyser</u>

The Semantic Analyser checks whether the parsed output follows the rules of language and is meaningful or not.

- Whenever there is a variable declaration in the code, the compiler traverses through the symbol table and checks if the same identifier name already exists in the current scope and whether the identifier name is "main". If any of these cases is true, the declaration cannot happen and the compiler throws a semantic error "*Multiple Declaration*".

- Whenever an identifier is accessed, the compiler traverses through the symbol table and checks whether the same identifier name exists in the current scope (and outer scopes where is appropriate) hierarchy. If not, the compiler throws a semantic error "*Identifier Not found*".
- Whenever there is an access to an array element, along with scope and identifier name checks as mentioned above, the compiler also checks for the 'out of bounds' condition for the indices. If the index is a constant, the symbol table entry is referred to and checked if the index exceeds the dimension mentioned in the symbol table. In the case it is found to be exceeding, the compiler returns "*Array out of bound*" error. But in case the index is a variable, its actual value is unknown and is only known during run-time. Hence it is handled dynamically in assembly code by inserting lines to compare the index with the corresponding dimension so that if the index is greater than or equal to dimension, a jump to the error label is made and an error message "ERROR!!" is thrown in the console (during run-time).
- Whenever there is a declaration of a function, the compiler traverses the symbol table and checks, whether another function with the same name exists or if the function name is "main". If any of these conditions is true, the compiler throws an error "*Function already Exists*"
- Whenever there is a function call, the compiler checks whether a function with the same name and number of arguments exists. If there is no such function in the symbol table, the compiler returns "*Function is not defined*".

e. Typechecker

- Whenever there is an assignment of a value to a variable, the compiler checks whether the destination type is the same as the value type or not. If not then the value is type cast as per the destination type and then assigned to it.
- Whenever there is an operation to be performed between two operands, the compiler checks for the type. If both of them are not of the same type and one of them is a float, then the corresponding type conversion to float occurs.
- In case of logical and relational operations, the destination type is made int and it returns either 1 or 0 only.

f. Intermediate code generator

After semantic analysis, the compiler converts each instruction into intermediate code, which will be used later for code generation. Each instruction in the code has been converted into intermediate code in the below defined format and kept in the list, which will be passed for code generation.
The intermediate code format for an instruction is:

<inst_type, src1, src2, dest>

where,

- inst_type : Instruction type
- src1 : Source 1
- src2 : Source 2
- dest : Destination

Different instruction types and the intermediate code format.

S. No.	Instruction Type	Source 1	Source 2	Destination	Description
1.	LABEL			L	L:
2.	DECLARE			a	int a;
3.	IF1	a		L	If a==1 goto L
4.	IFEQL	a	b	L	If a==b goto L
5.	IF0	a		L	If a==0 goto L
6.	SGT	a	b	c	c=(a>b)? 1:0
7.	SLT	a	b	c	c=(a<b)? 1:0
8.	ARRAYVAL	a	b	c	c=a[b]
9.	INPUT			a	a=input()
10.	FUNCALL	Start Address	List of arguments	Function Label	goto Function Label
11.	RETURN			a	return a
12.	ARGS			[a,b,c..]	function(int a, int b,..)
13.	EOF				End of Function
14.	ADD	a	b	c	c = a+b
15.	SUB	a	b	c	c = a-b
16.	MUL	a	b	c	c = a*b
17.	DIV	a	b	c	c = a/b
18.	AND	a	b	c	c = a&&b
19.	OR	a	b	c	c = a || b
20.	NOT	a		b	b = !a
21.	ASGN	constant		b	b = constant
22.	STORE	a		b	b = a
23.	GOTO			L	goto L
24.	PRINT			a	print(a)

25.	BREAK			L (label at end of loop and outside it)	break;
26.	CONTINUE			L (label at start of loop, pointing to loop condition)	continue;
27.	ERROR				Error

For different program statements, equivalent intermediate instructions are generated and appended to the code list. The Intermediate Code for different statements are as follows:

Code	Intermediate Code:

while(i >= j){
j = i + j
i = i - 1
}
	goto L2
	L1:
	j = i + j
	i = i - 1
	L2:
	r1 = i >= j
	if r1 goto L1
	L3:

if (a > b){
i = i+ 2;
}else
{
i = i+ 1;
}
	r1 = a>b
	if r1 goto L1
	goto L2
	L1:
	i = i+ 2
	goto L3
	L2:
	i = i + 1
	L3:

function int func(int a, int b){
a = b+1;
return a;
}
	func:
	a = b + 1
	v0 = a // fixed register to store return
	//value
	return a
	eof // end of function

c = func(i, j);	k1 = str_addr // start address of the function to be added to the relative addresses in symbol table entries of local variables
	a1 = 2 // number of arguments
	goto func
	c = v0 // functions will always store the return value to v0. In case there is no return statement, v0 is set to 0

g. Code generator

In this phase the compiler takes the generated Intermediate Code as input and maps it to the assembly code.

Steps involved in the code generation:

- Division of the Intermediate Code into basic blocks based on the following three rules
    - The First Intermediate Code instruction should be the start of a basic block
    - Any instruction that is the target of a conditional or unconditional jump is the start of a basic block..
    - Any instruction that immediately succeeds a conditional or unconditional jump statement is the start of a basic block
- Reverse Traverse each basic block and generate the status of each variable in the following format:
  <variable, live/not live, Nextuse>
  where,
    - ➔ *Not Live*, when next mention of the variable is an assignment and hence this instance of the variable need not live or be stored back to the memory.
    - ➔ *Live*, all the cases when the next mention of the variable is not an assignment, which means, if the register holding the variable needs to be spilled, a store-back to memory should happen before allocating the register to some other variable
    - ➔ *Nextuse*, contains the line number where the variable is going to be in use next.
- Now, to map the variable to assembly code we need to assign registers to them. The available (unreserved) registers are: 17 int registers and 27 float registers. In each basic block, the registers are allocated to the variable based on the following rules:
    - Choose an empty register.
    - If not available, choose the register occupied by a temporary variable with no next use.
    - If not available, choose the register occupied by a non-temporary variable with status not-live.
    - If not available, choose the register occupied by a non-temporary variable with the farthest nextuse.
    - In case of a register with the farthest nextuse its value is stored back to memory before reassignment.
- At the end of each basic block, all the non-temporary variables that have been modified will be stored back with the appropriate assembly code for storing to the memory. Additionally, all the variables (irrespective of whether altered or not) are removed from the register maps. This is to ensure that no matter which basic block the program control flows to next, there are no mistakes due to previously assuming that the variable had been in some different register. This implies that, in every basic block, the first instance of a non-temporary variable will require a load from memory and a new register is allocated.
- As for the temporary variables, their registers are **not** lost, but they continue to occupy the same register across basic blocks too until they can be spilled. This is required because temporary variables simply do not have any memory space allotted to them, so they cannot be stored back and hence need to live in their allocated registers even across basic blocks until spilled.
- To handle function calls and passing arguments, the current memory address from where allocation can begin for the function variables (arguments and local variables) is specified in the

intermediate code for the function call. So starting at this address, all the actual arguments are stored in memory just before the jump to the function instructions. This starting address is stored in $k1 and the number of arguments in $a1 (registers specially used for these purposes).

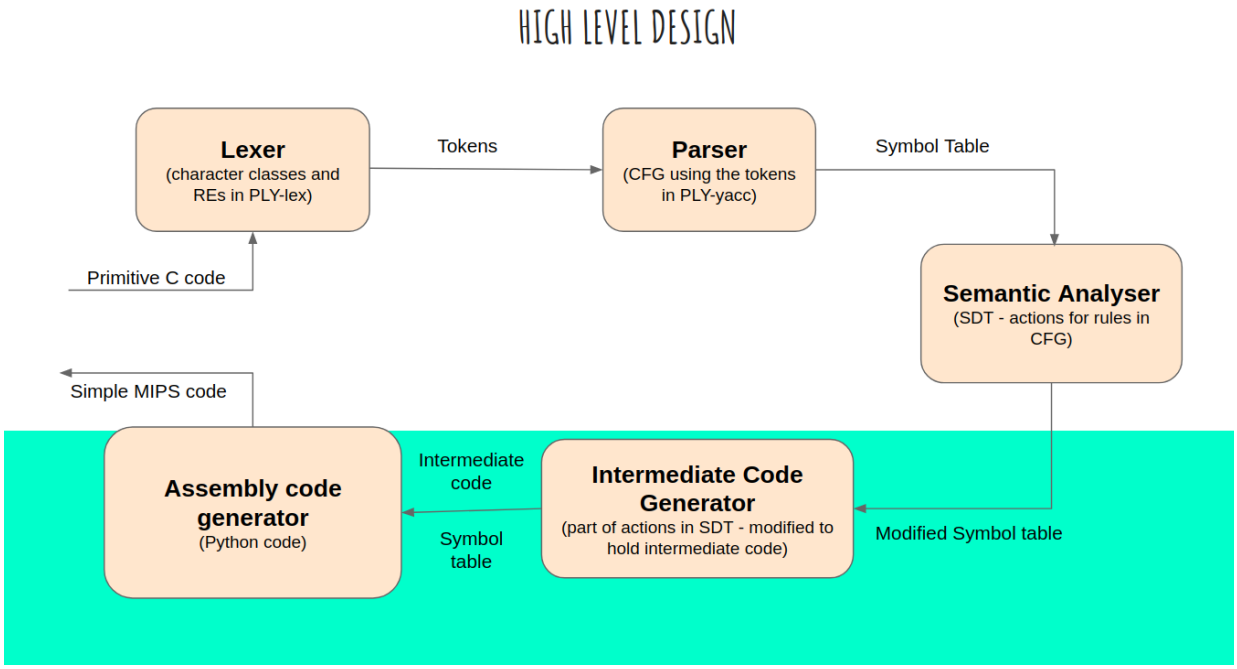
- At the start of every function definition, using the starting address in $k1, all the arguments are allocated registers and are loaded into them for further use inside the function. Local variables that are declared inside the function are taken care to load from the memory with $k1 offset. Initially in Parser.py, symbol table entries of such local variables are provided with a relative address and a boolean value specifying whether it was declared inside or outside a function. Hence when load or store statements in Assem.py are written, this boolean value is used to decide whether the $k1 offset should be added to get the absolute memory address or not. So this enables both the use of local **and** global variables inside functions.
- Handling of **floating point operations and conversions** in MIPS assembly code:
  To convert from float to int register:
  - cvt.w.s $f31, <source float register>
    // converting bit storage format from float to int format
  - mfc1 <destination int register>, $f31
    // moving value from float to int register (from one co-processor to the other)

  To convert from int to float register:
  - mtc1 <source int register>, $f31
    // moving value from int to float register (from one co-processor to the other)
  - cvt.s.w <destination float register>, $f31
    // converting bit storage format from int to float format

Reserved Registers

Register	Description
$k0	Stores the starting address of the entire code block
$v0	int return value of the function
$k1	Starting address of a function
$a1	Number of arguments of the function
$a0	syscall
$v0	syscall
$f0	for float return value
$f12	syscall
$f29	Register with value 1.0
$f30	Register with value 0.0
$a2	Reserved for other purposes
$a3	Reserved for other purposes
$f31	Reserved for other purposes

4. Flow/interactions between different components (diagram)



5. Source Code Organization

The libraries imported are ply.lex, ply.yacc, json and sys. The source code consists of 3 files: Lexer.py contains code that lexes the input code and generates tokens. Parser.py gets these tokens as input and parses the code according to the grammar specified in the ply.yacc format. Parser.py also contains code for semantic checks and intermediate code generation, including traversing the intermediate code in the reverse direction to get each variables NextUse and Live status. This intermediate code is the input for Assem.py which contains code for register allocation, register spills and generating corresponding final assembly code for each instruction in the intermediate code.
To compile the input code, the following command should be run in the terminal:
python3 Assem.py <input_code_file>
The assembly code will be generated in a file called Result.asm (in the current directory) and this can be fed to QtSPIM and run to see the output on the console provided by the tool.

6. Final testing and evaluation with screenshots

The following images show the code for Selection Sort of an array in the custom language created

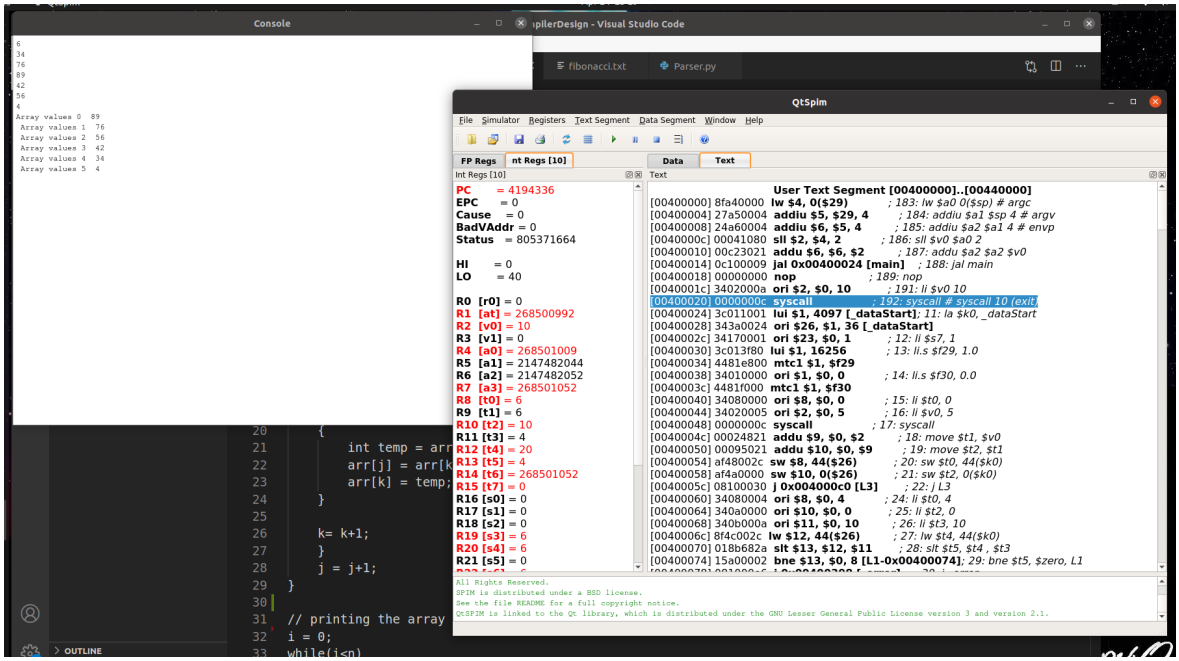
selectionSort.txt

```
// Sample Program of selection sort

int n, arr[10];
int i =0;
n =  input(int);
while(i<n)
{
    arr[i] = input(int);
    i=i+1;
}
int j=0,k=0;

// sorting the array
while(j<n)
{
    k = j+ 1;
    while(k<n)
    {
    if(arr[j]<arr[k])
    {
        int temp = arr[j];
        arr[j] = arr[k];
        arr[k] = temp;
    }

    k= k+1;
    }
    j = j+1;
}
// printing the array
i = 0;
while(i<n)
```

selectionSort.txt

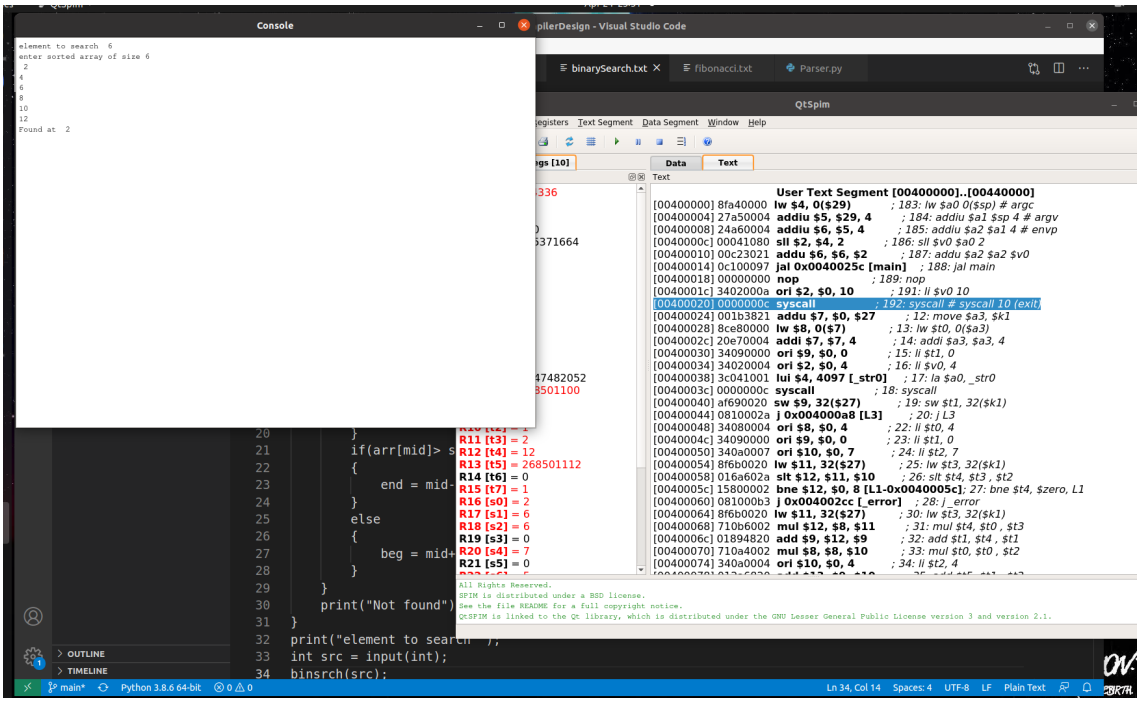
```
    arr[i] = input(int);
    i=i+1;
}
int j=0,k=0;

// sorting the array
while(j<n)
{
    k = j+ 1;
    while(k<n)
    {
    if(arr[j]<arr[k])
    {
        int temp = arr[j];
        arr[j] = arr[k];
        arr[k] = temp;
    }

    k= k+1;
    }
    j = j+1;
}

// printing the array
i = 0;
while(i<n)
{
    print("Array values", i , " ",arr[i], "\n");
    i=i+1;
}
```

In the image below we can see the Result.asm after compiling above code run on QtSPIM(on the right) and the result i.e. the sorted array that was input has been sorted and printed on the console (on the left):



Below are the images of the code for binary search of a number in the array input and the result on console (Found/Not found) on the console after running on QtSPIM:

```
int n = 6;
function binsrch(int src)
{
    int arr[7];
    int i = 0;
    print("enter sorted array of size 6 \n");
    while(i<n)
    {
        arr[i] = input(int);
        i = i+1;
    }
    int beg=0, end=n-1, mid;
    while(beg<=end)
    {
        mid= (beg+end)/2;
        if(arr[mid] == src)
        {
            print("Found at ", mid);
            return;
        }
        if(arr[mid]> src)
        {
            end = mid-1;
        }
        else
        {
            beg = mid+1;
        }
    }
    print("Not found");
}
print("element to search ");
int src = input(int);
```



7. Limitations of the compiler

- The compiler regards each basic block as an independent entity, which implies that the non-temporary variables do not live in the registers beyond each block, and in each block, every first instance of a non-temporary variable is loaded from memory and assigned a register anew. This leads to more loads and stores from memory as compared to a more optimised compiler that keeps track of flow of program control among the basic blocks.
- There is the possibility of some unnecessary labels being generated, that could have been otherwise avoided if the corresponding optimisation was incorporated.

- The compiler does not support dynamic memory allocations and recursive functions. So the custom language (as specified in the language manual) also does not allow constructs for the same.
- Functions also cannot have full arrays as formal arguments, although single array elements are allowed as actual arguments.
- Writing break/continue statements outside loops does not trigger an error, but they will not affect the flow of the program either. They are just ignored outside loops.
- Only 2 data types are supported: int and float, and hence character and string declaration and manipulation are not supported.

8. Conclusion

The team has gained clarity and intuition behind every component of the compiler previously described. The project gave the team an opportunity to dive deeper into the concepts and algorithms learnt previously in the course and also made for an enriching and enjoyable experience overall.

9. Contributions

The work division was made such that each subtask was done in teams of two. In this process, each team member got to work with every other team member.

S. No.	Task	Contributors
1.	Lexer and Parser - Grammar design	all four
2.	Code for Lexer	Swetha, Shruti
3.	Validation of Parser Grammar	Sai Datta, Pranathi
4.	Base code for Parser	Swetha, Pranathi
5.	Symbol Table - Design	Shruti, Sai Datta
6.	Populating Symbol Table (declare statement and function definition)	Swetha, Sai Datta
7.	Checking the validity of variables used (in other statements)	Pranathi, Shruti
8.	Memory allocation - address updation in Symbol Table	Swetha, Pranathi
9.	Intermediate Code design	Shruti, Sai Datta
10.	Intermediate Code Generation (expressions, assign statement, input statement)	Sai Datta, Swetha
11.	Intermediate Code Generation (other statements)	Shruthi, Pranathi

12.	Compute the next use and live / not-live information of the symbol table variables, Record the start of the basic blocks in the code	Swetha, Shruti
13.	Register allocation, spilling (considering type conversion in assembly code level and basic block level dependencies)	Sai Datta, Pranathi
14.	Assembly code generation (first 14 types of instructions)	Swetha, Shruti
15.	Assembly code generation (other 13 types of instructions)	Sai Datta, Pranathi