# Object-Oriented Extension to ExpL Language

**A THESIS**

*submitted by*

**N RUTHVIK B130887CS**

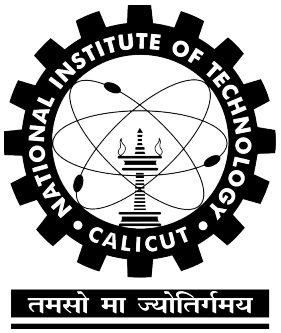
**THALLAM SAI SREE DATTA B130353CS**

in partial fulfilment for the award of the degree of

**Bachelor of Technology**
in
**Computer Science and Engineering**

under the guidance of

**DR. K MURALIKRISHNAN**




**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**
**National Institute of Technology, Calicut**
**NIT Campus P.O, Calicut**
**Kerala, India 673601**

April 24, 2017

## DECLARATION

*"We hereby declare that this submission is our own work and that, to the best of our knowledge and belief, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due reference has been made in the text".*

**Place : NIT Calicut**
**Date : 24 April 2017**

**Signature :**

**Name : N Ruthvik**
**Roll No : B130887CS**

**Signature :**

**Name : Thallam Sai Sree Datta**
**Roll No : B130353CS**

**CERTIFICATE**

*This is to certify that the thesis entitled:* ***"Object-Oriented Extension to ExpL Language"*** *submitted by Mr.* ***N Ruthvik B130887CS*** *and Mr.* ***Thallam Sai Sree Datta B130353CS*** *to National Institute of Technology, Calicut towards partial fulfilment of the requirements for the award of Degree of Bachelor of Technology in Computer Science and Engineering is a bonafide record of the work carried out by them under my supervision and guidance.*

*Signed by Guide with name and date*

**Place :**
**Date :**

*Signature of Head of the Department*

*Office Seal*

**Abstract**

ExpL (Experimental Language) is an experimental programming language with a very simple specification. The specification has been developed keeping in mind that it should be simple enough that an undergraduate student can develop a compiler for it in a few months but complex enough that the language is non-trivial. It can be used as an instructional tool to understand the working and implementation of a compiler for the students to be able to build compiler under minimal expert supervision. This project aims to develop an online self-sufficient educational platform to help undergraduate Computer Science students understand the functioning of a compiler for a simple procedural language by writing a small toy compiler themselves. An object oriented extension for the language has been proposed subsequently. The report presents design and specification of ExpL Language along with object-oriented features, ExpL Grammar outline, specification of application binary interface (ABI) of target machine and design of data structures for the new ExpL Compiler.

# Contents

# Chapter 1

# Introduction

This section gives a brief outline of the project, the existing work and also the requirement of the new system.

## 1.1 Project Objective

The main aim of the project is to propose an Object-oriented extension to the existing ExpL language specification and design the compile-time and run-time data structures required to implement the compiler for the same. This project also aims at documenting the roadmap, the description and usage of necessary tools that help in the compiler development, the specification of ABI for the target machine.

## 1.2 Existing Work

The compiler's implementation techniques and principles have been presented in a very simple and elegant manner by Aho, Lam, Sethi and Ullaman[1]. Appel[2] provides implementations with several examples in C. The documentation by Nachiappan, Ashwathy and subisha[3] provided the valuable inputs to the design of ExpL.

## 1.3 Why New System is Required?

1. Self Learning documentation under minimal expert supervision.
2. Do not wish to stop at intermediate stage but proceed to execution phase using target machine.
3. Roadmap[4] based approach is new. A step by step guidance is given in the road map. Brief theoretical explanations are provided on a need-to-do basis.
4. Simpler than existing projects (Architecture has been made simple).
5. Open source platform design.

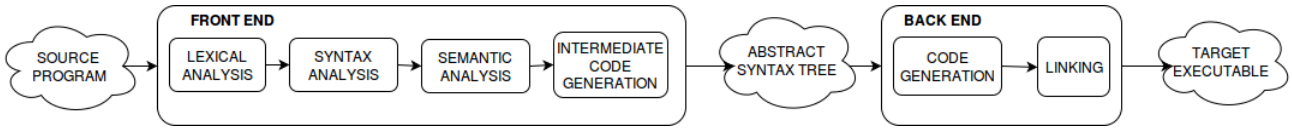
## 1.4 Overview of the project

To design a compiler, one needs the following specifications :

1. The specification of source programming language.
2. *Application Binary Interface of the target platform*: A compiler must generate a binary executable file that can be loaded and executed by an operating system running on a target machine. Normally, an OS system installation comes with an interface specification, explaining how binary executable files must be formatted to run on the system. This interface is called the Application Binary Interface (ABI). The ABI is dependent on both the OS and the machine architecture. The ABI for this project is that of the Experimental Operating System (ExpOS) running on the Experimental String Machine. [XSM]. (The simulator allows the target program to contain machine instructions in mnemonic form, avoiding translation to binary format).

A compiler takes a source language program as input and produces an executable target file formatted according to the requirements laid down by the ABI. Of course, the semantics of the program must be preserved by the translation process.

The sequence of phases starting with the lexical analysis of the source program to generation of abstract syntax tree representation is called the *front end* of the compiler.




The abstract syntax tree produced by the front end is the input to the *back end* phase which generates an assembly language program. Finally, a label translation phase is run to replace symbolic labels in the assembly language program with logical addresses (linking) and generate final target executable file.

## 1.5 Aim

1. To complete the target platform.
2. To create a documentation of the project and a roadmap of the various stages of compiler development.
3. Design a Object-oriented extension to ExpL.

# Chapter 2

# Specification

## 2.1 Informal Language Specification

Following are the minimal features that the language supports.

### 2.1.1 Primitive data types

1. **Integer** : An integer type variable is declared using the keyword int.
   Example : int a, b, c ; /* Declares variables a, b, c of type integer */

2. **String** : A string is a sequence of alphanumeric characters. A string type variable is declared using the keyword str.
   Example : str mystring ; /* Declares a variable mystring of type string. */

3. **Boolean** : ExpL does not permit boolean variables. But logical expressions like (a < b) or (a==b) and (a < 5) are supported and are considered to be of type boolean.

### 2.1.2 Composite data types

1. **Arrays** : Arrays can be of integer or string types only. Only single-dimensional arrays are allowed. Arrays are statically allocated.
   Example : int a[10] ; /* array a indexed a[0], a[1], ..., a[9], can store 10 integers */
   str stringlist[10] ; /* stringlist is an array of 10 strings */

2. **User-defined types** : ExpL allows user defined data types. The (member) fields of a user defined type may be of type integer, string, a previously defined user defined type or the type that is currently being defined.
   Example : A user defined type, mytype is defined as

```
mytype
{
  int a;
  str b;
}
```

### 2.1.3 General Program Structure

An ExpL program consists of the following sections :

(a) Type Definitions - Optional (for user defined types)

(b) Global Declarations - for global variables, arrays and functions

(c) Function Definitions and the main Function Definition

Local Variables and parameters should be allocated space in the run-time stack of the function. The language supports recursion. Each statement should end with a "**;**" which is called the *statement terminator*.

There are seven types of statements in ExpL. They are:

1. Assignment Statement
2. Conditional Statement
3. Iterative statement
4. Return statement
5. Input/Output statements
6. Break statement
7. Continue statement

### 2.1.4 Statements and Expressions

ExpL has four kinds of expressions, a) Arithmetic expressions, b) String expressions, c) Logical expressions and d) Expressions that evaluate to user defined types.

1. **Constants** : Any numerical value (Example: 234) is an integer constant. A quoted string (Example: "hello") is a string constant. ExpL allows a special constant NULL whose type is void. It is assumed that string constants contain only alphanumeric characters (and no special characters).

2. **Arithmetic and String Expressions** : Any integer(or string) constant/variable is a valid arithmetic (or string) expression, provided the scope rules are not violated. ExpL treats a function returning integer (or string) as an integer expression (or string expression) and the value of a function is its return value.

   ExpL provides five arithmetic operators, viz., +, -, *, / (Integer Division) and % (Modulo operator) through which arithmetic expressions may be combined. Expression syntax and semantics are similar to standard practice in programming languages and normal rules of precedence, associativity and parenthesization apply. ExpL is strongly typed and any type mismatch or scope violation must be reported at compile time.

3. **Logical Expressions** : Logical expressions may be formed by combining arithmetic expressions using relational operators. The relational operators supported by ExpL are <, >, <=, >=, ==, and **!**=. Again standard syntax and semantic conventions apply. Logical expressions may be combined using logical operators and, or and not.

   The relational operators <, >, <=, >=, ==, and **!**= can be used between two variables of type string, two string constants or a string constant and a variable of type string. Comparison will be performed for lexicographic ordering.

4. **Expressions of user defined types** : Any variable of a user defined type or invocation of a function whose return type is a user defined type is considered as an expression of the corresponding user defined type.

5. **Assignment Statement** : The general syntax of the assignment statement is

   *Lvalue = Rvalue;*

   The possible Lvalues are variables or indexed array variables. If the Lvalue has type integer (or string) , the Rvalue must be an arithmetic (or string) expression. If the Lvalue is a user defined variable, then the Rvalue must either be an expression of the same type, or the special constant NULL, or an invocation of the function alloc() (to be explained later).
   Example : q[3] = "hello" ; t = "world" ;

6. **Conditional Statement** : The ExpL conditional statement has the following syntax:

   *if < Logical Expression > then*
   *Statements*
   *else*
   *Statements*
   *endif;*

   The else part is optional. The statements inside an if-block may be conditional, iterative, assignment, input/output, break or continue statements, but not the return statement.

7. **Iterative Statement** : The ExpL iterative statement has the following syntax:

   *while < Logical Expression > do Statements endwhile;*

   Standard conventions apply in this case too. The statements inside a while-block may be conditional, iterative, assignment, input/output, break or continue statements, but not the return statement.

8. **Return Statement** : The body of each function (including main) should have exactly one return statement and it should be the last statement in the body. The type of the expression should match with the return type of the function. The syntax is:
   *return < Expression* > ;*

9. **Input/Output statements** : Using read statement, we can read a string or an integer into a variable of type string or integer respectively from the standard input. The syntax of the input statement is as follows

   *read( < variable > );*

   Using the write statement, we can write the value of an integer or string type variable or the value of an arithmetic expression to the standard output. The output statement is as follows

   *write( < expr > );*

10. **Break and Continue Statements** : A *break;* statement inside an iterative block tranfers control to the end of the block. A continue; inside a conditional/iterative block transfers control to the beginning of the block. These statements do nothing if not inside any conditional/iterative statement.

11. **Breakpoint Statement** : This statement results in the ExpL compiler setting a break point in the program. This feature is useful for debugging.
    *breakpoint;*

### 2.1.5 Dynamic memory allocation

The library functions initialize(), alloc() and free() are used as follows:

```
intialize();        // To Intialise the heap.
t = alloc();        // Allocates contiguous locations in the heap,
                       t must be a user defined variable.
retval = free(t);   // Free the allocated block,
                       t must be a user-defined variable.
```

Intialize() must be invoked before any allocation is made and it resets the heap to default values. A call to alloc() allocates contiguous memory locations in the heap memory (memory reserved for dynamic memory allocation) and returns the address of the starting location. The Expl compiler sets the variable (of a user defined type) on the left of the assignment to store this memory address. A call to free() deallocates contiguous memory locations in the heap memory that is referenced by the user defined type variable. The function free() returns NULL on successful deallocation. Otherwise, the value of t is unchanged by a call to free(). All unallocated user defined variables are set to the predefined constant NULL.

## 2.2 ExpL Grammar outline

```
TypeDefBlock  : TYPE TypeDefList ENDTYPE
              |
              ;

TypeDefList   : TypeDefList TypeDef
              | TypeDef
              ;

TypeDef       : ID '{' FieldDeclList '}' { TInstall(tname,size,$3); }
              ;

FieldDeclList : FieldDeclList FieldDecl
              | FieldDecl
              ;

FieldDecl     : TypeName ID ';'

TypeName      : INT
              | STR
              | ID       //TypeName for user-defined types
              ;

GDeclBlock : DECL GDeclList ENDDECL
           |
           ;

GDeclList  : GDecList GDecl
           | GDecl
           ;

GDecl      : TypeName Gidlist ';'
           ;

Gidlist    : Gidlist ',' Gid
           |   Gid
           ;

Gid        :   ID                  { GInstall(varname,ttableptr, 1, NULL); }
           |   ID '(' ParamList ')' { GInstall(varname,ttableptr, 0, $3); }
           |   ID '[' NUM ']'       { GInstall(varname,ttableptr, $3, NULL); }
           ;

ParamList     :  ParamList ',' Param  { AppendParamlist($1,$2);}
```

```
           |  Param
           |  //There can be functions with no parameters
           ;

Param      : TypeName ID { CreateParamlist($1,$2); }
           ;

FDefList   : FDefBlock
           | FDefList FDefBlock
           ;

FDefBlock : TypeName ID '(' ParamList ')' '{' LdeclBlock Body '}'
                                    { GUpdate($2->name,$1,$4,$7,$8); }
           ;

Body       : BEGIN Slist Retstmt END
           ;

Slist      : Slist Stmt
           |
           ;

Stmt       : ID ASGN Expr ';'
           | ....
           | IF '(' Expr ')' THEN Slist ELSE Slist ENDIF ';'
           | ...
           | ID ASGN ALLOC'(' ')' ';'
           | FIELD ASGN ALLOC'(' ')' ';'
           | FREE '(' ID ')' ';'
           | FREE '(' FIELD ')' ';'
           | READ '(' ID ')' ';'
           | READ '(' FIELD ')' ';'
           | WRITE '(' Expr ')' ';'
           ;

FIELD      : ID '.' ID
           | FIELD '.' ID
           ;

Expr       : Expr PLUS Expr
                   {
                      $$ = TreeCreate(TLookup("int"),NODETYPE_PLUS,NULL,
                                (union Constant){},NULL,$1,$3,NULL);
                   }
           | ....
           | '(' Expr ')'
```

```
          | NUM
          | ID
          | ID '[' Expr ']'
          | FIELD
          | ID '(' ArgList ')'
                        {
                        gtemp = GLookup($1->name);
                        if(gtemp == NULL){
                            yyerror("Yacc : Undefined function");
                            exit(1);
                        }
                        $$ = TreeCreate(gtemp->type,NODETYPE_FUNCTION,$1->name,
                                (union Constant){},$3,NULL,NULL,NULL);
                        $$->Gentry = gtemp;
                        }
           ;

MainBlock : INT MAIN '(' ')' '{' LdeclBlock Body '}'
                         {
                             type = TLookup("int");
                             gtemp = GInstall("MAIN",type,0,NULL);
                             //...Some more work to be done
                         }
          ;
```

## 2.3 Target ABI Specification

The ExpL compiler needs to translate a given source program and generate the target machine code into an executable file in a format which is recognized by the load module of the target operating system. Thus, in order to generate the executable, the following information needs to be made available to the compiler :

1. The **virtual machine model and the instruction set** of the target machine.
2. The (virtual) **address space model** available for the target program. Conventionally, this address space is logically divided into regions like code, data, stack, heap etc.
3. The format for the target file (**executable format**). The compiler typically passes information regarding the sizes and address regions allocated to the code, data, stack, text and heap regions to the loader by setting appropriate values in the header of the executable file.
4. Interfaces to OS (kernel) routines that needs to be invoked to get certain operations like input/output done. This is specified in the library interface documentation.

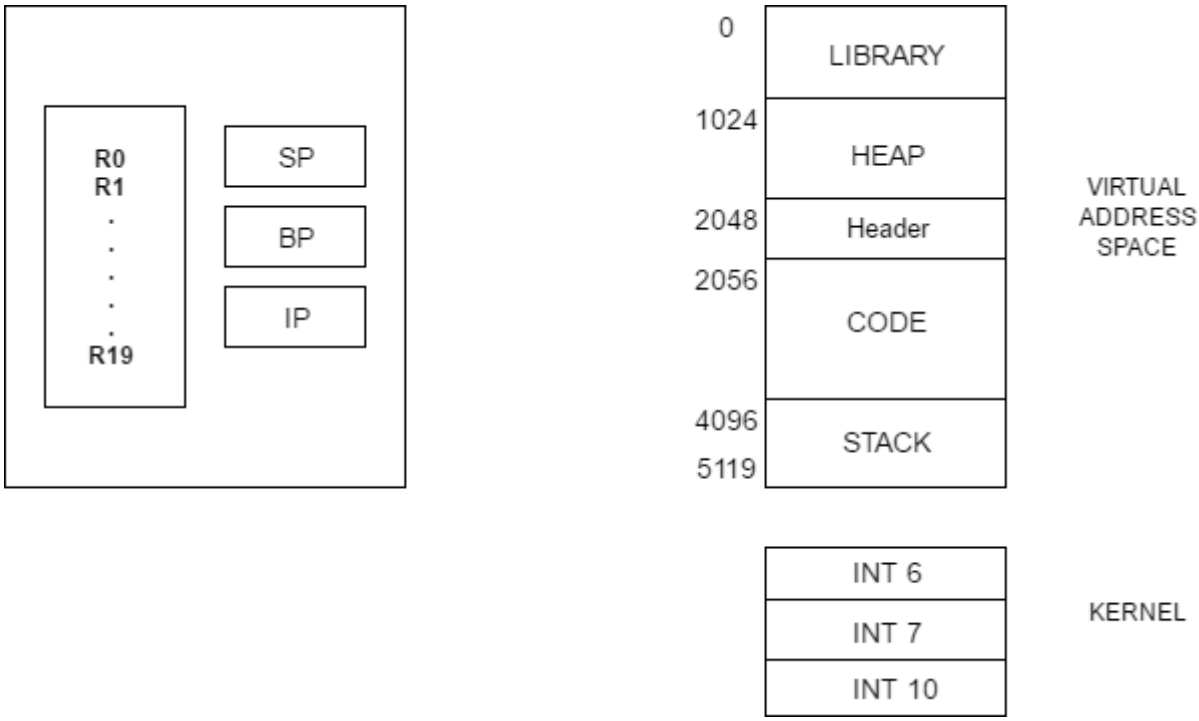
These specifications depend not only on the target machine architecture, but also on the operating system upon which the target machine code must execute. Typically these specifications are collected together in the OS specification into a document called the **Application Binary Interface** (ABI).

### 2.3.1 The XSM virtual machine model and instruction set

The virtual machine model provided to user level programs by the ExpOS on the machine is described here. The virtual machine model explains the "view" of the machine provided to the application by the operating system.

The XSM virtual machine contains 20 general purpose registers (R0-R19) and three special registers - Stack pointer (SP), Base pointer (BP) and the Instruction pointer (IP). The SP register is normally used to point to the top of the application program's stack. The BP register is used to point to the base of the activation record of the currently executing function. IP points to the next instruction in the user memory to be fetched and executed. The memory address space available to the application is of 5120 words starting at (virtual) address 0 and ending at (virtual) address 5119.

The following figure gives a high level picture of the XSM virtual machine model.




### 2.3.2 The XSM virtual machine instruction set

The XSM virtual machine instruction set specifies the set of assembly level instructions. The compiler must translate the source ExpL program to a target program containing only these instructions. The assembly instructions allowed include Data Transfer Instructions, Arithmetic Instructions, Logical Instructions, Stack Instructions, Sub-routine instructions, Debug

instructions and Software interrupts. The machine registers available to the target program are R0-R19, SP, BP and IP.

1. **Data Transfer Instructions** : This includes MOV instruction which stores the value of a register to specified memory location and vice-versa. Addressing types such as Immediate Addressing, Direct Addressing etc. can be used.

2. **Arithmetic Instructions** : Arithmetic Instructions perform arithmetic operations on registers containing integers. If the register contains a non-integer value, an exception (illegal instruction) is raised. Arithmetic instructions are ADD, SUB, MUL, DIV and MOD.

3. **Logical Instructions** : Logical instructions are used for comparing values in registers. Strings can also be compared according to the lexicographic ordering of ASCII. If one of the operands is a string, the other operand will also be considered as a string. The logical instructions are LT, GT, EQ, NE, GE and LE.

4. **Branching Instructions** : Branching is achieved by changing the value of the IP to the word address of the target instruction specified by 'target address'. They include JZ, JNZ and JUMP.

5. **Stack Instructions** : PUSH and POP.

6. **Subroutine Instructions** : The CALL instruction copies the address of the next instruction to be fetched(this value must be IP + 2 since each instruction is two memory words) on to location SP + 1. It also increments SP by one and transfers control to the instruction specified by the target address. The RET instruction restores the IP value stored at location pointed by SP, decrements SP by one and continues execution fetching the next instruction pointed to by IP.

7. **Debug Instruction** : BRKP. The machine when run in debug mode invokes the debugger when this intruction is executed. This instruction can be used for debugging system code.

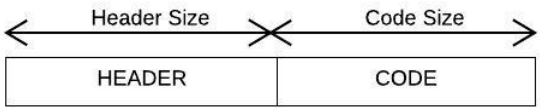
8. **Software Interrupt** : INT n
   Generates an interrupt to the kernel with the interrupt number (n between 4 to 18) as an argument.

### 2.3.3 XEXE executable file format

The compiler must generate target code into a file in the format specified below so that the eXpOS loader recognizes the format and load the program into memory for execution correctly. Each executable file contains a header in which the compiler adds information like the initial value to be given to the stack pointer in the virtual address space, initial value of the instruction pointer etc, the starting (virtual) addresses and sizes of various memory regions like text, stack, heap etc.

Executable files in eXpOS must be in the XEXE format as eXpOS executes only files of such format. An XEXE executable file in eXpOS consists of two parts:

1. Header
2. Code



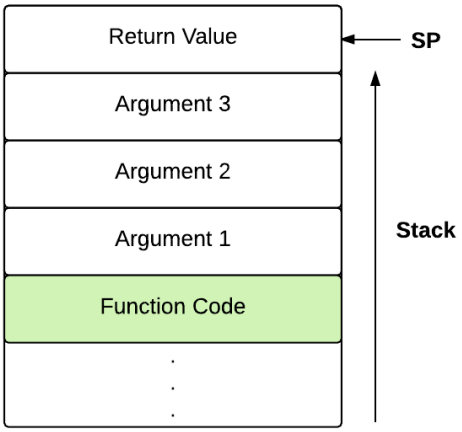

The maximum size of the file (including the header) is limited by 2048 words. The first eight words of an executable file are reserved for the header which describes the features of file. The structure of the header is :

| XMAGIC | Entry Point | Text Size | Data Size | Heap Size | Stack size | Library Flag | Unused |
|---|---|---|---|---|---|---|---|

**XMAGIC** is a number indicating the type of executable file. All XEXE files will have magic number 0. **Entry point** contains the virtual address in memory of the first instruction to be executed (entry point) of the program after the OS loader has loaded it. During loading, the instruction pointer must be initialized to this address. **Text Size**, **Data Size**, **Heap Size** and **Stack size** indicates the sizes of Text, Data, Heap and Stack regions to be allocated by the OS loader when the file is loaded for execution. If the **Runtime Library** must be included when the file is loaded, the Library Flag is set to 1 in the executable file. If this flag is not set then neither memory is allocated for the heap nor the library linked to the address space of the process at execution time.

### 2.3.4 The library interface

The library provides a uniform interface through which an application program can invoke dynamic memory allocation / de-allocation routines (Alloc(), Free() and Initialize()), input - output routines (Read() and Write()) and program exit (Exit()). Thus, while translating a high level ExpL program containing calls to the above functions, a compiler needs to be concerned only about how to translate these high level function calls to the corresponding library function calls.




The ABI stipulates that all calls to the library functions must be translated by the compiler to a CALL to virtual address 0, with an appropriate function code that identifies the service requested (Alloc(), Free(), Initalize(), Read(), Write() or Exit()). This is because the library is linked to virtual address 0 of the address space of a process by the OS loader. A call to the library requires four arguments (a function code to identify the service and three arguments) to be passed through the stack. The library will invoke the corresponding low level system call / memory management routine and returns to the user program the return value of the system call / memory management routine through the stack. The figure above shows the contents of the stack immediately before a call to this library routine.

**Invoking a library module** :

A library module invocation using the high level application programmer's interface of a programming language like ExpL must be translated by the compiler to a set of machine instructions as given below.

```
PUSH Function_Code // Push Function Code
PUSH Argument_1    // Push argument 1 to the stack
PUSH Argument_2    // Push argument 2 to the stack
PUSH Argument_3    // Push argument 3 to the stack
PUSH R0            // Push an empty space for RETURN VALUE
CALL 0             // Pass the control to virtual address 0
```

Following are the library functions and details relevant for ExpL Compilation :

| Library Function | Function Code | Argument 1 | Argument 2 | Argument 3 | Return Values |
|---|---|---|---|---|---|
| Read | "Read" | -1 | Buffer (int/str)* | - | 0 - Success |
| | | | | | -1 - File Descriptor given is invalid |
| | | | | | -2 - Read error |
| Write | "Write" | -2 | Buffer (int/str)* | - | 0 - Success |
| | | | | | -1 - File Descriptor given is invalid |
| Exit | "Exit" | - | - | - | - |
| Initialize | "Heapset" | - | - | - | 0 - Success |
| | | | | | -1 - Failure |
| Alloc | "Alloc" | Size (int) | - | - | Address in the heap allocated (int) |
| | | | | | -1 - No allocation |
| Free | "Free" | Pointer (int) | - | - | 0 - Success |
| | | | | | -1 - Failure |

*Note: The Read() and Write() library functions expect a memory address from (or to) which read or write is performed.

**After return from the library module** :

The following machine instructions are present after the CALL instruction in the ExpL compiled machine code given in the previous step.

```
POP Ri  // Pop and save the return value into some register Ri
POP Rj  // Pop and discard argument 3
POP Rj  // Pop and discard argument 2
POP Rj  // Pop and discard argument 1
POP Rj  // Pop and discard the function code
        // Now the stack is popped back to the state before call
```

The machine code shown above is executed upon return from the library call and pops out the values that were pushed before the call. The function code and arguments were inputs to the library module and hence they may be discarded now. The return value which is stored in the stack by the system call must be popped out and saved to some register. This value will be the only relevant information after return from the call.

### 2.3.5 Low Level System Call Interface

The ExpL library file library.lib contains assembly instructions to implement the library functions Alloc(), Free() and Initialize(). For the input/output functions Read()/Write() as well as the program exit system call Exit(), the library code invokes the corresponding ExpOS system calls for console read/write/exit. Thus, the library simply converts the library call to a low level call to the operating system. This is because console input/ouput functions are implemented at the OS level.

In order to implement the library, one must know the low level system call interface to the operating system so that the input/ouput calls to the library can be correctly translated to the corresponding low level OS system calls. This section specifices the low level OS interface provided by the ExpOS Operating System running on the XSM machine architecture. The interface describes the software interrupt instruction (trap) corresponding to the con solve input/output system calls and the calling conventions for passing arguments and extracting return values of the system call through the application program's stack. This part is architecture as well as operating system dependent.

For an application program, there are two stages in executing a system call:

1. **Before the system call** : The calling application must set up the arguments in the (user) stack before executing the trap instruction.
2. **After the system call** : The return value of the system call must be extracted from the stack.

**Invoking a system call** :

```
PUSH System_Call_Number // Push system call number
PUSH Argument_1         // Push argument 1 to the stack
PUSH Argument_2         // Push argument 2 to the stack
PUSH Argument_3         // Push argument 3 to the stack
PUSH R0                 // Push an empty space for RETURN VALUE
INT number              // Invoke the corresponding INT instruction.
```

**After return from the system call** :

```
POP Ri // Pop and save the return value into some register Ri
POP Rj // Pop and discard argument 3
POP Rj // Pop and discard argument 2
POP Rj // Pop and discard argument 1
POP Rj // Pop and discard the system call number
```

**System calls and their translation** :

Associated with each system call, there is a system call number and interrupt routine number. The system call number is used to identify a system call. The interrupt routine number denotes the number of the interrupt routine which handles the system call.

| System Call | System Call Number | Interrupt Routine Number | Argument 1 | Argument 2 | Argument 3 | Return Values |
|---|---|---|---|---|---|---|
| Read | 7 | 6 | -1 | Buffer (int/str)* | - | 0 - Success |
| | | | | | | -1 - File Descriptor given is invalid |
| | | | | | | -2 - Read error |
| Write | 5 | 7 | -2 | Buffer (int/str)* | - | 0 - Success |
| | | | | | | -1 - File Descriptor given is invalid |
| Exit | 10 | 10 | - | - | - | - |

*Note: The Read() and Write() library functions expect a memory address from (or to) which read or write is performed.

# Chapter 3

# Design

## 3.1 Compile Time Data Structures

The compilation of an ExpL program involves two phases. In the first phase (called the analysis phase), the source ExpL program is analyzed (lexical, syntax and semantic analysis are completed in this phase) and if the program is free of syntax and semantic errors, an intermediate representation of the source program called the abstract syntax tree is generated.

This is followed by a second phase, the second phase (called the synthesis phase) recursively traverses the abstract syntax tree and generates target code.

There are four basic data structures that are maintained during the analysis phase. These are the following:

1. The Type table is used to store information regarding the primitive and user-defined types in the program.
2. The global symbol table is used to store information about the global variables and functions in the program.
3. For each function, a separate local symbol table is maintained to store the information about local variables and arguments of the function.
4. Finally, the abstract syntax tree is constructed as the outcome of the analysis phase.

### 3.1.1 Type Table

The structure of Type Table is as follows :

```
struct Typetable{
  char *name;                 //type name
  int size;                   //size of the type
  struct Fieldlist *fields;   //pointer to the head of fields list
  struct Typetable *next;     // pointer to the next type table entry
};
```

The variable 'fields' is a pointer to the head of 'fieldlist'. Here 'fieldlist' stores the information regarding the different fields of a user-defined type.

```
struct Fieldlist{
  char *name;              //name of the field
  int fieldIndex;          //the position of the field in the field list
  struct Typetable *type;  //pointer to type table entry of the field's type
  struct Fieldlist *next;  //pointer to the next field
};
```

## Associated Methods

1. *void TypeTableCreate()* : Function to initialise the type table entries with primitive types (int,str) and special entries(boolean,null,void).
2. *struct Typetable* TLookup(char *name)* : Search through the type table and return pointer to type table entry of type 'name'. Returns NULL if entry is not found.
3. *struct Typetable* TInstall(char *name,int size, struct Fieldlist *fields)* : Creates a type table entry for the (user defined) type of 'name' with given 'fields' and returns the pointer to the type table entry. The field list must specify the field index, type and name of each field. TInstall returns NULL upon failure. This routine is invoked when the compiler encounters a type definition in the source program.
4. *struct Fieldlist* FLookup(Typetable *type, char *name)* : Searches for a field of given 'name' in the 'fieldlist' of the given user-defined type and returns a pointer to the field entry. Returns NULL if the type does not have a field of the name.
5. *int GetSize (Typetable * type)* : Returns the amount of memory words required to store a variable of the given type.

### 3.1.2 Global Symbol Table

The structure of Global Symbol Table(GST) is as follows:

```
struct Gsymbol{
  char *name;               //name of the variable or function
  struct Typetable *type;   //pointer to the Typetable entry of variable type/
                              return type of the function
  int size;                 //size of an array or a user defined type.
                                (The default types have size 1)
  int binding;              //stores the static memory address allocated to
                                 the variable
  struct Paramstruct *paramlist; //pointer to the head of the formal parameter list
                                      in the case of functions
  int flabel;               //a label for identifying the starting address
                              of the function's code in the memory
  struct Gsymbol *next;     //points to the next Global Symbol Table entry
};
```

Paramlist is used to store information regarding the types and names of the parameters. ParamStruct has the following structure.

```
struct Paramstruct{
  char *name;              //stores the name of the parameter
  struct Typetable *type;  //pointer to type table entry of parameter type
  struct Paramstruct *next;  //pointer to the next parameter
};
```

## Associated Methods

1. *struct Gsymbol* GInstall(char *name,struct Typetable *type, int size, struct ParamStruct *paramlist)* : Creates a Global Symbol entry of given 'name', 'type', 'size' and 'parameter list' and assigns a static address('binding') to the variable (or label for the function).

2. *struct Gsymbol* GLookup(char *name)* : Search for a GST entry with the given 'name', if exists, return pointer to GST entry else return NULL.

### 3.1.3 Local Symbol Table

The structure of Local Symbol Table(LST) is as follows:

```
struct Lsymbol{
  char *name;               //name of the variable
  struct Typetable *type;   //pointer to the Typetable entry of variable type
  int binding;              //stores memory address allocated to the variable
  struct Lsymbol *next;     //points to the next Local Symbol Table entry
};
```

## Associated Methods

1. *struct Lsymbol* LInstall(char *name,struct Typetable *type)* : Creates a local symbol table with given 'name' and 'type' and also sets its 'binding'.

2. *struct Lsymbol* LLookup(char *name)* : search the LST and if any entry with given 'name' is found ,return the entry,else returns NULL.

### 3.1.4 Abstract Syntax Tree

The machine independent front-end phase of a compiler constructs an intermediate representation of the source program called the Abstract Syntax Tree (AST).This is followed by a machine dependent back-end that generates a target assembly language program.

The node structure of Abstract Syntax Tree is as follows:

```
struct ASTNode{
  struct Typetable *type;          //pointer to the type table entry
  int nodetype;                    //node type information
  char *name;                      //stores the variable/function name
```

```
  union Constant value;             //stores the value of the constant
                                      if the node corresponds to a constant
  struct ASTNode *arglist;          //pointer to the expression list given
                                      as arguments to a function call
  struct ASTNode *ptr1,*ptr2,*ptr3; //Subtrees of the node
  struct Gsymbol *Gentry;           //pointer to GST entry
  struct Lsymbol *Lentry;           //pointer to the function's LST
};
```

The union Constant is used to store the value of an integer or sting constant.

```
union Constant{
  int intval;
  char* strval;
 };
```

## 3.2 Run Time Data Structures

This section explains how the compiler allocates memory for variables in a program.

### 3.2.1 Register Allocation

Register allocation is performed through two simple functions.

1. *int get_ register()* : Allocates a free register from the register pool (R0 - R19) and returns the index of the register, returns -1 if no free register is available.
2. *int free_ register()* : Frees the last register that was allocated,returns 0 if success, returns -1 if the function is called with none of the registers being allocated.

### 3.2.2 Heap Allocation

ExpL specification stipulates that variables of user defined types are allocated dynamically using the alloc() function. The alloc() function has the following syntax:

*user_ defined_ type_ var = alloc();*

We have used Fixed Memory Allocation algorithm for allocating memory in the heap. Following is the **allocation algorithm**.

1. First index of reserved block is checked, let the value be v.
2. If v is -1, return -1 indicating no free blocks are available.
3. Else,
   allocate the free block at v,
   copy the next free block index stored at v to the reserved block. Return v.

Following is the **de-allocation algorithm**.

1. The argument passed : starting address of the block(say s) to be deallocated.
2. The block s is cleared by setting all memstructs in the block to type EMPTY.
3. The value in the first index of reserved block is copied to first index of block s.
4. The first index of reserved block is set with starting address of block s.

# Chapter 4

# Implementation

A detailed step by step instructions required for implementing compiler for ExpL is documented in roadmap[4]. The road map is divided into several stages.

## 4.1 Code generation for Arithmetic Expressions

In this section we build a compiler for simple arithmetic expressions. A very simple compiler that can take an arithmetic expression as input (from some input file) and generate a target executable file containing XSM instructions to evaluate the expression and output the result.

## 4.2 Introduction to static storage allocation

In this stage, we extend the expression evaluator of the previous stage to support a set of pre-defined variables with Input/Output and assignment statements. The notion of static storage allocation enroute is introduced. We learn to differentiate between statements and expressions and also construct an abstract syntax tree representation for a program.

## 4.3 Adding Flow Control Statements

In this stage, we extend the straight-line-program compiler of Stage 2 to support control flow constructs like if-then-else, while-do, break and continue. We introduce integer and boolean expressions and the notion of type enroute. We also learn the use of labels for handling control flow constructs.

## 4.4 User Defined Variables and arrays

We extend the language of Stage 3 to permit users to declare and use variables of integer and string types. We also learn symbol table management enroute.

## 4.5 Adding Functions

We extend the language of Stage 4 by adding functions with support for recursion. Addition of functions to the language requires handling scope of variables. Support for recursion demands run-time storage allocation. Only integer and string type variables will be supported.

## 4.6 User defined types and Dynamic Memory Allocation

We extend the language of Stage 5 by adding support for user-defined types and dynamic memory allocation. Issues of Heap management will be encountered en route.

The implementation of the compiler can be found here[5].

# Chapter 5

# Object-oriented Extension

In previous chapters, we have seen the specification, Grammar outline, compile-time and run-time data structures that are needed to build the ExpL Compiler. In this chapter, we will see the extended Grammar and specification, compile-time and run-time data structures needed to include object-oriented features to this language. This extension to the ExpL language is called **OExpL** viz *Object-Oriented Extension to ExpL*.

## 5.1 Specification

### 5.1.1 General Program Structure

The language must now support Class definitions. A Class definition can have function definition or a variable declaration. As per specification, the total number of class definitions and variable declarations shouldn't exceed eight.

A class can inherit the properties of another (already defined) class using the ***extends*** keyword. A function of the parent class is overridden in the child class only if the entire function signature (name of the function, the parameter list) is matched. This is called *parametric polymorphism*. A sample definition of classes in an ExpL program is provided in Appendix I 7. Note that the classes are defined just after the user-defined datatype declarations (if any).

Every class variable must have two words of memory allocated in the stack. One word contains a pointer to the heap memory just as user-defined variable, the second word is used to store the starting address of class in the run-time data structure that is explained later.

To create an object of class of name class_name, following syntax is used:

```
<class_variable> = alloc(<class_name>);
```

So this statement invokes the **alloc()** function (in library) which returns the starting address of 8 words of memory (in heap) allocated to it. This address is stored in the binding address of class variable.

To invoke a function of a class, a Dot (.) operator is used with the class variable holding the object of the class. The syntax for invoking a member function of a class is as follows :

```
<return_value> = <class_variable> . <function_name>( <paramlist> );
```

## 5.2 ExpL Grammar outline (extended)

```
ClassDefBlock : CLASS ClassDefList ENDCLASS
              |
              ;

ClassDefList : ClassDefList ClassDef
              | ClassDef
              ;

ClassDef : ID '{' ClassDeclList '}'
               {
                  CInstall(name,CFhead,CVarhead);
               }
          | ID EXTENDS ID '{' ClassDeclList '}'
               {
                  CInstall($1 -> name,CFhead,CVhead);
                  extends($1 -> name, $3 -> name);
                  extends_var($1 -> name, $3 -> name);
               }
         ;

ClassDeclList : ClassDeclList ClassDecl
              | ClassDecl
              ;

ClassDecl : TypeName ID '(' ParamList ')' '{' LdeclBlock Body '}'
            {
                CFuncInstall(label,$2 -> name,$4);
            }
          | TypeName ID ';' { CVarInstall($2 -> name,type);}
         ;

Stmt :
       ...
     | ID ASGN ALLOC '(' ID ')' ';'
     | THIS '.' ID ASGN Expr ';'
     | ID ASGN THIS '.' ID ';'
     | THIS '.' ID ASGN THIS '.' ID ';'
     | READ '(' THIS '.' ID ')' ';'
     | WRITE '(' THIS '.' ID ')' ';'
     ;

Expr :
       ...
     | ID '.' ID '(' ')'
```

```
        | ID '.' ID '(' exprlist ')'
        | THIS '.' ID '(' ')'
        | THIS '.' ID '(' exprlist ')'
        ;
```

## 5.3 Design

### 5.3.1 Compile Time Data Structures

A Class Table is used to store information regarding classes and its associated methods and variables declared, in the program. The structure of Class Table is as follows :

```
struct Classtable {
    char *name;               //class name
    int ClassIndex;           //index of class, starts from 0
    int size;                 //size of the class
    struct CFunclist *CFuncs; //pointer to the head of class functions list
    struct CVarlist *CVars;   //pointer to the head of class variables list
    int binding;              //address of the start of class memory
                                in stack (to store label numbers)
    struct Classtable *next;  // pointer to the next class table entry
};
```

The variable *CFuncs* is a pointer to the head of *CFunclist* and *CVars* is a pointer to the head of *CVarlist*. Here *CFunclist* stores the pointer to list of functions that are defined inside a class and *CVarlist* stores the pointer to the list of variables that are declared inside a class.

```
struct CFunclist {
    char *name;             //name of the class field (function name)
    int ClassFuncIndex;     //the position of the field in the class func list
    int label;              //label number of function
    struct Paramstruct *paramlist; //pointer to the head of the formal
                                     parameter list
    struct CFunclist *next; //pointer to the next Class function field
};

struct CVarlist {
    char *name;             //name of the class variable
    int ClassVarIndex;      //the position of the variable in the list
    struct Typetable *type; //pointer to the type of the variable in class table
    struct CVarlist *next;  //pointer to the next Class variable ïňĄeld
};
```

## Associated Methods

1. *struct Classtable* CLookup(char* name)* : Search through the class table and return pointer to class table entry with class name as *name*. Returns NULL if entry is not found.

2. *void CInstall(char* name,struct CFunclist *CFuncs,struct CVarlist *CVars)* : Creates a class table entry for the class with name as *name*, given *CFuncs* and *CVars* and returns the pointer to the class table entry. CInstall returns NULL upon failure. This routine is invoked when the compiler encounters a class definition in the source program.

3. *void extends(char* name1, char* name2)* : This routine extends the CFunclist of class *name1* with that of CFunclist of class *name2*.

4. *void extends_ var(char* name1, char* name2)* : This routine extends the CVarlist of class *name1* with that of CVarlist of class *name1*.

5. *struct CFunclist* CFLookup(char* name,struct CFunclist *list)* : Search through the CFunclist of the current class that is being parsed and return pointer to the entry in the list with function name as *name*. Returns NULL if entry is not found.

6. *void CFuncInstall(int label,char* name, struct Paramstruct *paramlist)* : Creates a class function entry for the class with function name as *name* and returns the pointer to this entry. CFuncInstall returns NULL upon failure. This routine is invoked when the compiler encounters a function definition (inside classes) in the source program.

7. *struct CVarlist* CVLookup(char* name,struct CVarlist *list)* : Search through the CVarlist of the current class that is being parsed and return pointer to the entry in the list with variable name as *name*. Returns NULL if entry is not found.

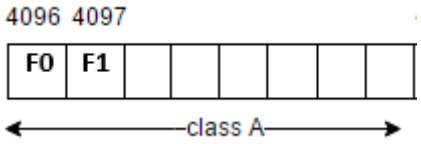
8. *void CVarInstall(char* name, struct Typetable *type)* : Creates a class variable entry for the class with variable name as *name* and returns the pointer to this entry. CVarInstall returns NULL upon failure. This routine is invoked when the compiler encounters a variable declaration (inside classes) in the source program.

### 5.3.2 Run Time Data Structures

One of the main challenge in implementing the object-oriented features is implementing *run-time polymorphism*. The challenge here is to determine the correct function that is to be invoked, depending on the class to which the object belongs to. To understand this problem more clearly we refer to the example program provided in Appendix I 7.

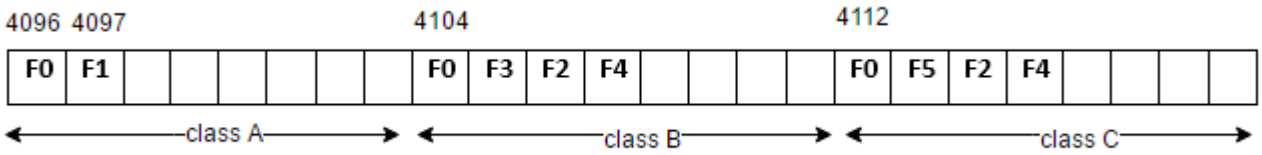
In this program, depending on the input number read into variable *val*, the corresponding *f2* function should be called. The ExpL compiler must generate code to solve this problem at compile time itself.

So, we propose a data structure to be made available in the memory at run-time to handle this issue. As per the specification, the total number of functions defined, and variables declared should not exceed eight. So we need to allocate a memory of eight words to each class where each word will store the starting address of the functions defined in that class in order.




For the purpose of understanding, we have replaced the function bindings with its labels. In case of extended classes, we need to first copy the class table of the parent class. In case the child has overridden any functions of the parent class, those labels of the parent class need to be updated with that of the child class (as *f1* of class B) and the labels of extra functions that the child class have need to be appended to this list.

For the program in Appendix I, this data-structure would look like:




The code for creating this data structure has to be generated after the parser parses all the class definitions. A class variable will be given two words of memory, one storing the address of heap memory to which it was allocated before and second storing the starting address of the class in this data structure. The variables declared inside a class are given heap memory same as in case of user-defined variables.

### 5.3.3 THIS Keyword

**this**,**self**, and **Me** are the keywords used in some computer programming languages to refer to the object, class, or other entity that the currently running code is part of. The entity referred to by these keywords thus depends on the execution context (such as which object is having its method called). The ExpL compiler also supports **THIS** keyword which has almost the same functionality as that in other programming languages. THIS holds the pointer of the current object class at run-time and so a member of the class (function or variable) in which this keyword is used can be accessed using THIS keyword.

The implementation of the compiler can be found here[5].

# Conclusion and Future Work

In this project, the existing specification of ExpL is finalised and Object-oriented features like class, single level inheritance and parametric polymorphism are included in the language. Design of compile-time and run-time data structures required for implementing the OExpL compiler are proposed and a working compiler [5] is built using the same. A step by step procedure to build the compiler is documented as roadmap[4] and some relevant documents that help in understanding the procedure are made available.

This project use library interface for calling system level functions which prevents recompiling of source program in case of underlying architecture changes. A simple data structure is proposed for implementing run-time polymorphism.

At present, the memory allocation strategy used is a fixed memory allocation. Efficient memory allocation algorithms like Buddy memory allocation can be used instead. Code optimization techniques like Copy Propagation, Dead-Code Elimination, Code Motion etc. can be done after code generation phase. Features like multiple file linking (#include) can be added. The Finite State Automata generated by the parser can be documented for better understanding of YACC tool. The step by step procedure for implementing the proposed data structures of OExpL can be appended to the existing roadmap. The OExpL can be extended further to include comments, exceptions and exception handlers like catch-throw as in C++ and Java programming languages.

# Appendix I

```
class
    A {
        int f1()
        {
            begin
                write("A f1");
                return 20;
            end
        }
        int f2()
        {
            begin
                write("A f2");
                return 0;
            end
        }
    }

    B extends A {
        int k;
        int f1(int a)
        {
            begin
                write(a);
                write("B f1");
                return 0;
            end

        }
        int f2()
        {
            begin
                write("B f2");
                return 0;
            end
        }
        int f3()
        {
            begin
                write("B f3");
                return 30;
            end
        }
    }
```

```
    C extends B {
        int f2()
        {
            begin
                write("C f2");
                return 0;
            end

        }
    }
endclass

decl
    A var;
enddecl

int main()
{
    decl
        int val;
    enddecl
    begin
        val = initialize();
        read(val);

        if(val == 1) then
            var = alloc(A);
        else if(val == 2) then
                var = alloc(B);
            else if(val == 3) then
                var = alloc(C);
                 endif;
            endif;
        endif;

        val = var.f2();
        return 0;
    end
}
```

# References


[1] Alfred V.Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D.Ulman. *Compilers : Principles,Techniques and Tools*. Pearson Education, 2007.

[2] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 2002.

[3] Nachiappan V, Ashwathy T Revi, and Subisha V. Documentation of lex, yacc, data structures involved in the implementation of expl and report on fixed memory allocator http://www.silcnitc.github.io, https://github.com/silcnitc/silcnitc.github.io/blob/master/report.pdf?raw=true.

[4] Roadmap. http://silcnitc.github.io/roadmap.html.

[5] Thallam Sai Sree Datta and N Ruthvik. Implementation of oexpl compiler https://github.com/dattathallam/object-oriented-extension-to-expl-compiler.