ExpL: An Instructional Programming Language

A THESIS
Submitted by

ASHWATHY T REVI (B110208CS)
SUBISHA V (B110475CS)

In partial fulfilment for the award of the degree of

BACHELOR OF TECHNOLOGY
IN
COMPUTER SCIENCE AND ENGINEERING

Under the guidance of
Dr. Muralikrishnan K

DEPARTMENT OF COMPUTER ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY CALICUT
NIT CAMPUS PO, CALICUT
KERALA, INDIA 673601

May 15, 2015

ACKNOWLEDGEMENTS

We would like to express our deepest gratitude , first and foremost, to our guide, Dr Muralikrishnan K for taking so much time and effort to guide us through this project. We also thank the other teams working in this project for their support and cooperation. We thank our juniors, Salakha, Mahesh, Anjana, Vishnu Priya, Peeyush and Disni for their feedback and assistance. Finally we thank the Computer Science Department of NITC for giving us this platform.

DECLARATION

"I hereby declare that this submission is my own work and that, to the best of my 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 acknowledgment has been made in the text".

Place:
Date:

Signature :
Name :
Reg.No:

CERTIFICATE

*This is to certify that the thesis entitled: "***ExpL: An Instructional Programming Language***" submitted by Sri/Smt/Ms* **ASHWATHY T REVI (B110208CS) SUBISHA V (B110475CS) to National Institute of Technology Calicut towards partial fulfillment of the requirements for the award of Degree of Bachelor of Technology in Computer Science Engineering is a bonafide record of the work carried out by him/her under my/our supervision and guidance.**

Signed by Thesis Supervisor(s) with name(s) and date

Place:
Date:

Signature of Head of the Department

Office Seal

Contents

Chapter

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 under graduate 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.

Chapter 1

Introduction

In this project, we aim to develop a learning platform that aids students in understanding the key concepts of compiler design such as Lexical Analysis, Syntax Analysis, Semantic Analysis, Interpretation and management of static and dynamic memory. A variety of such learning platforms are already available. SIL (Simple Integer Language) is one such language, developed with the intention of being non trivial enough to cintain the key aspects of a programming language but simple enough that an interpreter/compiler may be developed easily enough and attempts to avoid features that provided no insight to compiler design. Language specification and other details may be found at silcnitc.github.io. In ExpL, we have added the following features to the Simple Integer Language (SIL) that was used in compiler lab: Support user defined types Support dynamic memory allocation Hence students can gain insight into these features of a programming language as well.

We have organised the report in the following manner. An informal language speification is contained in Chapter . Its formal syntax is given as CFG in Chapter 3. Chapter 4 discusses the various Data Structures we have used for the implentation of the interpreter. Semantic Analysis is discussed in Chapter 5. Dynamic Memory Allocation strategies and algorithms are outlined in Chapter 6. The implementation of the interpreter is explained in Chapter 7. Finally a conclusion of the work done and future work is offered in Chapter 8.

Chapter 2

Source Language Description

2.1 Informal specification

2.1.1 Experimental Language (eXpL)

The language specification given here is informal and gives a lot of flexibility for the designer to write the grammatical specifications to his/her own taste. The following features are the minimal requirements for the language.

2.1.2 Data Types supported by eXpL

The types supported by eXpL are given below.

Primitive data types

Integer An integer value can range from -32767 to +32768. An integer type variable is declared using the keyword int.

String A string type represents the set of string values. A string value can be atmost 16 characters long. String type variables is declared using the keyword str.

Boolean eXpL does not allow Boolean variables but Boolean expressions (like a¡b , a==b & a¡5) are supported.

Composite Data types

Arrays Arrays are sequence of elements of a single type. Arrays can be of integer or string types. eXpL allows the use of single-dimensional arrays only, i.e. linear arrays. Array elements are accessed by the array name followed by an index value within square brackets (e.g. arr [10]).

User-defined types eXpL allows user defined data types .The member fields of a newly defined type may be of type integer, string, a previously defined type or the type that is currently being defined. Arrays are not allowed. There may be a maximum of eight fields.

Note Compiler does not statically allocate memory for user defined types. It has to be dynamically allocated at run time. Hence the programmer has to call the library function allocate() before a variable of user defined type can be used. User defined types take default value null unless allocated or assigned otherwise.

Example 2.2.1

A user defined type mytype is declared as:

typedef mytype

int a;

str b;

a variable of type mytype is declared as:

mytype var;

and initialised as:

var=alloc();

Its fields may be accessed as:

var-¿a=10;

var-¿b=”string”; // -¿ is used to address the fields.

The memory allocate may be freed as:

var=dealloc();

Note var is internally represented like a pointer variable. The library function alloc is used to allocate memory and return pointer to allocated memory. (Library functions are explained in detail later on.) The returned pointer is stored in var. Compiler should report error if the declaration of mytype (which is the type of var) does not contain fields named a and b of the appropriate type.

2.1.3 General Program Structure

Type Declarations (optional , only required if user defined types are used)

Global Declarations

Function Definitions

Main Function Definition

Type Declarations User-defined types are declared in the type declaration section of the program. Type Declaration section starts with keyword typedecl and ends with endtypedecl. Declaration of all user defined types that are to be used in the program are to be made within this section. So continuing the earlier example, mytype has to be declared in the type declaration section along with other user defined types (say mytype2 and mytype3).

Example 3.1.1 typedecl

typedef mytype

int a;

str b;

typedef mytype2

```
str a;
mytype b;
typedef mytype3
int c;
mytype3 m;
endtypedecl
. . .
//Rest of the program
```

Global Declarations The global declaration part of a eXpL program begins with the keyword vardecl and ends with the keyword endvardecl. Declarations should be made for global variables and functions defined in the eXpL program. Global variables may be of type Integer, String, a User defined type, Integer array or String array. The variables declared globally must be allocated statically by the compiler. Global variables are visible throughout the program unless suppressed by a redeclaration within the scope of some function. Array type variables can be declared only globally. Only single dimensional arrays are allowed. Variables cannot be assigned values during the declaration phase. For every function except the main function defined in a eXpL program, there must be a declaration. A function declaration should specify the name of the function, the name and type of each of its arguments and the return type of the function. A function can have integer/string/user defined type arguments. Parameters may be passed by value or reference.Arrays cannot be passed as arguments. (Note that by default, User defined types are passed by reference.) If a global variable name appears as an argument, then within the scope of the function, the new declaration will be valid and global variable declaration is suppressed. Different functions may have arguments of the same name. However, the same name cannot be given to different

arguments in a function. The return type of a function must be either integer or string or user defined type. The general form of declarations is as follows: Type VarName/FunctionName [ArraySize]/(ParameterList); //Third part needed only for arrays/functions

Example 3.2.1

vardecl

int x,y,a[10],b[20]; // x,y are integers, a,b are integer arrays

int f1(int a1,a2; str b1; int &c1), f2(); // c1 is passed by reference, rest by value

str t, q[10], f3(str x); // variable, array and a functions declared together

int swap(int &x, &y); // x, y are passed by reference

mytype m, f4(int a); //myptype is a user defined type that is declared earlier in type declaration section. m is a pointer to an object of type mytype and f4 is a function that takes an integer argument and returns a pointer to an object of type mytype.

endvardecl // Please note the use of "," and ";"

Declaring functions at the beginning avoids the "forward reference" problem and facilitates simpler single pass compilation. Note that the declaration syntax of functions is structurally same as that for variables. Finally, inside swap, the global variables x and y are no more visible because of the re-declaration and global declaration for x is suppressed in f3. If a variable/function is declared twice at the same point, a compilation error should result. **Function Structure and Local Variables**

All globally declared variables are visible inside a function, unless suppressed by a re-declaration. Variables declared inside a function are invisible outside. The

general form of a function definition is given below:

¡Type¿ FunctionName(ArgumentList)

Local Declarations

Function Body

The arguments and return type of each function definition should match exactly with the corresponding declaration. Argument names must be type checked for name equivalence against the declaration. Every declared function must have a definition. The compiler should report error otherwise.

The syntax of local declarations and definitions are similar to those of global declarations except that arrays and functions cannot be declared inside a function. Local variables are visible only within the scope of the function where they are declared. Scope rules for parameters are identical to those for variables.

The main() function, by specification, must be a zero argument function of type integer. Program execution begins from the body of the main function. The main function need not be declared. The definition part of main should be given in the same format as any other function. The Body of a function is a collection of statements embedded within the keywords begin and end.

Example 3.3.1 The definition of swap declared above may look like the following:

int swap (int &x, &y)

vardecl

int q ;// q is re-declared causing suppression of global declaration

endvardecl

begin

q = x;

```
x = y;
y = q; // Note the syntax for using variables passed by reference.
return 1; // swap must return an integer.
end // Note that eXpL doesnt support void functions.
```

Local Variables and parameters should be allocated space in the run-time stack. The language supports recursion and static scope rules apply.

Main and Function Body A Body is a collection of statements embedded within the keywords begin and end. Each statement should end with a ; which is called the terminator. There are five types of statements in eXpL. They are:

a) Assignment Statement

b) Conditional Statement

c) Iterative statement

d) Return statement

e) Input/Output

Before taking up statements, we should look at the different kinds of expressions supported by eXpL.

2.1.4 Expressions

eXpL has two kinds of expressions,a) Arithmetic and b) Logical

Arithmetic Expressions Any constant, integer variable or an indexed array variable is a eXpL expression provided the expression is within the scope of the concerned variable declarations. eXpL treats a function as an expression and the value of a function is its return value. eXpL supports recursion. eXpL provides five arithmetic operators, viz., +, -, *, / (Integer Division) and I% (Modulo oper-

ator) 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 paranthesization hold. eXpL is strongly typed and any type mismatch must be reported at compile time.

Examples: 5, a[a[5+x]]+x , (f2() + b[x] + 5) etc. are arithmetic expressions.

Logical Expressions Logical expressions can take values TRUE or FALSE.

Logical expressions may be formed by combining arithmetic expressions using relational operators. The relational operators supported by eXpL are ¡, ¿, ¡=, ¿=, ==, and !=. Again standard syntax and semantics conventions apply. TRUE and FALSE are constant logical expressions. Every boolean variable is a logical expression and its value is the value stored in its location. Logical expressions themselves may be combined using logical operators AND, OR and NOT. Logical expressions cannot be assigned to a variable since Boolean operators are not supported. Note that a relational operator can compare only two arithmetic expressions and not two logical expressions. Similarly, a logical operator can connect only two logical expressions (except for NOT which is a unary logical operator). Example:((x==y)==a[3]) is not valid eXpL expression because (x==y) is a logical expression, while a[3] is an arithmetic expression and "==" operates only between two arithmetic expressions.

2.1.5 Assignment Statement

The eXpL assignment statement assigns the value of an expression to a variable, or an indexed array of the same type. Type errors must be reported at compile time. The general syntax is as the following:

¡Variable¿ = ¡Expression¿;

Example: q[3]= "hello" ; t= "world" ; are both valid assignments to string vari-

ables.

* Note Assignments among variables of user defined types take a slightly differ-

ently interpretation. In an assignment x=y where x and y are of a primitive type (integer or string), the value inside the location indicated by y is copied into the location indicated by x. On the other hand, if x and y are of a user defined type, the assignment only makes both x and y indicate the same location in memory. This is because since user defined types are allocated dynamically, a variable of user defined type can store only a reference to such a memory location (that will be allocated at run time).

2.1.6 Conditional Statement

The eXpL conditional statement has the following syntax:

if ⟨*LogicalExpression*⟩ then

Statements

else

Statements

endif;

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

2.1.7 Iterative Statement

The eXpL iterative statement has the following syntax:

while ⟨*LogicalExpression*⟩ do

Statements

endwhile;

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

2.1.8 Return Statement

The main body as well as each function body should have exactly one return statement and it should be the last statement in the body. The syntax is:

return ⟨*Expression*⟩ ;

The return value of the function is the value of the expression. The return type should match the type of the expression. Otherwise, a compilation error should occur.

The return type of main is integer by specification.

2.1.9

{*Input/Output*} eXpL has two I/O statements read and write. The syntax is as the following:

read (¡Variable¿);

write (¡Arithmetic Expression/String¿);

The read statement reads an integer value or string from the standard input device into corresponding variable or an indexed array variable. The write statement outputs the value of the arithmetic expression or string into the standard output.

Example: read (a [x]);

write (7*(5+a[9]);

2.1.10 Library Functions alloc() and dealloc() for Dynamic Memory Management

alloc() usage: t=alloc();

A call to alloc() allocates 8 contiguous memory locations in the heap memory (memory reserved for dynamic memory allocation) and returns the address to the starting location. This address may be stored in a variable of user defined type . (Example 2.2.1). dealloc() usage: t=dealloc();

A call to dealloc() deallocates 8 contiguous memory locations in the heap memory that is indicated by user defined type variable t.

Note: alloc and dealloc may be implemented by declaring a large array and using it as a heap.

An Example eXpL Program without user defined type The following eXpL program calculates and prints out the factorial of the first n numbers, value of n read from standard input.

```
vardecl
integer factorial(integer n);
endvardecl
integer factorial (integer n)
vardecl
integer rvalue;
endvardecl
begin
if (n==1) then
rvalue = 1;
else
```

```
rvalue = n * factorial (n-1);
endif;
return rvalue; // Note only one RETURN statement is allowed.
end

integer main( ) // Main definition should always begin like this
vardecl
integer n,i ;
endvardecl
begin
read (n);
i = 1;
while ( i ¡= n) do
write ( factorial(i));
i = i + 1;
endwhile;
return 1; // Any integer value may be returned
end
```

An Example eXpL program using User Defined types Read elements into a linked list and print them.

```
typedecl
typedef linkedList
int data;
linkedList next;
```

```
endtypedecl
vardecl
linkedList head;
endvardecl
int main()
vardecl
int length, x;
linkedList p, q;
endvardecl
begin
head=null;
read(length);
q=head;
while(length!=0) //reading elements to a linked list
do
read(x);
p=alloc();
p-¿data=x;
p-¿next=null;
if(head==null)
then
head=p;
q=p;
else
q-¿next=p;
q=q-¿next;
endif;
```

```
length=length-1;
endwhile;
p=head;
while(!p) //printing elements of the linked list to stdout
do
write(p-¿data);
p=p-¿next;
endwhile;
end
return 1;
```

2.1.11 APPENDIX

Keywords The following are the reserved keywords in eXpL and it cannot be used as identifiers.

read print if then else endif while do endwhile break continue integer string main return decl enddecl Create Open Write Seek Read Close Delete Fork Exec Exit breakpoint typedecl endtypedecl vardecl endvardecl null

Operators and Delimiters The following are the operators and delimiters in eXpL

( ) {} ⟨ ⟨= ⟩= ⟩ [!=] == / ; * = + && - % ! &

Identifiers Identifiers are names of variables and user-defined functions. Identifiers should start with an alphabet, and may contain both alphabets and

digits. Special characters are not allowed in identifiers.

identifier :: (alphabet)(alphabet — digit)*

Literals There are integer literals and string literals in eXpL. An integer literal is a sequence of digits representing an integer. Negative integers are represented with a negative sign preceding the sequence of digits. Any sequence of characters enclosed within double quotes are considered as string literals. However eXpL restricts string literals to size of atmost 16 characters including the ' 0' character which is implicitly appended at the end of a string value. Examples of literals are :

19, -35, "Hello World"

Chapter 3

Formal Context Free Grammar

```
Prog:           TypeDeclBlock GDecblock Fdefblock Mainblock
                ;

TypeDeclBlock: TYPE TypeDefList ENDTYPE
                |
                ;

TypeDefList:    TypeDefList TypeDef     //Appends the newly created TypeTable to the existing.
                |TypeDef                //The globally maintained TypeTable TTable is set to $1.
                ;

TypeDef:        ID '{' TypeDeclList '}'   //Creates a 'TypeTable' out of the intermediate list.
                                          //Verifies for multiple declaration of variables.
                                          //Verifies if the type assigned to the used defined variables are
                                          declared before or is the current one under defintion
                ;

TypeDeclList:   TypeDeclList TypeDecl
                |TypeDecl
                ;

TypeDecl:       INTEGER IDList ';'        //Fills the Type pointer in the intermediate list with integer
                |STRING IDList ';'        //Fills the Type pointer in the intermediate list with string
                |ID IDList ';'            //Fills the Type pointer in the intermediate list(IntermList) with the
                                  name of the given identifier($1)
                ;

IDList : IDList ',' ID   //Creates an intermediate list(IntermList) containing the name of the given identifier.
                |ID   //Creates an intermediate list(IntermList) containing the name of the given identifier.
                ;

GDecblock :      DECL GDecList ENDDECL
                ;

GDecList :       GDecList GDecl      //Appends the newly created entries to the GST.
                |GDecl
                ;

GDecl :         INTEGER GIdList ';' //The Type field of the global symbol table entry is set to integer.
                |STRING GIdList ';'   //The type field of the global symbol table entry is set to string

                |ID GIdList ';   //Type field of the global symbol table entry is set to the specified type.
                                 //The specified type for used defined type is obtained from a call to
                                 TypeTableLookUp function.
                ;
```

```
GIdList :       GIdList ',' GId   //Binds together the global symbol tabe entries.
               |GId
               ;

GId :           ID '[' NUM ']'    //Creats a global symbol table entry
               |ID                //Creates a global symbol table entry
               |ID '(' ArgList ')' //Creates a global symbol table entry
               ;

FArgList :      ArgList           //A Local Symbol Table is created out the entries made.
               ;

ArgList :       ArgList ArgType   //Appends newly created entries to the existing
               |ArgType
               ;

ArgType :       INTEGER Args ';'  //The Type field in the ArgStruct entry is set to the specified type.
               |STRING Args ';'   //The Type field in the ArgStruct entry is set to the specified type.
               |ID Args ';'       //The Type field in the ArgStruct entry is set to the specified type.
               |INTEGER Args      //The Type field in the ArgStruct entry is set to the specified type.
               |STRING Args       //The Type field in the ArgStruct entry is set to the specified type.
               |ID Args           //The Type field in the ArgStruct entry is set to the specified type.
               ;

Args :          Args ',' Arg      //Appends newly created entries to the existing.
               |Arg
               ;

Arg :           ID                //Creates an ArgStruct entry containing name of the identifier.
               ;

Fdefblock :     Fdefblock Fdef
               |
               ;

Fdef :          INTEGER ID '(' FArgList ')' '{' Ldecblock Body '}'
        //Function definition is compared with their declarartion earlier for compatibility
        //Lentry is set to the LST of the function
        //LST is set to NULL

               |STRING ID '(' FArgList ')' '{' Ldecblock Body '}'
        //Function definition is compared with their declarartion earlier for compatibility
        //Lentry is set to the LST of the function
        //LST is set to NULL

               |ID ID '(' FArgList ')' '{' Ldecblock Body '}'
        //Function definition is compared with their declarartion earlier for compatibility
        //Lentry is set to the LST of the function
        //LST is set to NULL
               ;
```

```
Ldecblock :     DECL LdecList ENDDECL  //Appends the local symbol table to the LST(with parameters)
                ;

LdecList :      LdecList Ldec1          //Appends newly created local symbol table entries to the existing.
                |Ldec1                  //$$=$1;
                ;

Ldec1 :         INTEGER LIdList ';'     //Fills the Type field of the local symbol table entry with integer.
                |STRING LIdList ';'     //Fills the Type field of the local symbol table entry with string.
                |ID LIdList ';'         //Fills the Type field of the Local symbol table with the specified
        user defined type.
                ;

LIdList :       LIdList ',' LId    //Appends newly created local symbol table entries to the existing.
                |LId               //$$=$1;
                ;

LId :           ID                 //Creates a Local Symbol Table entry containing the name of the identifier
                ;

Mainblock :     INTEGER MAIN '(' ')' '{' Ldecblock Body '}'
        //Verifies if the type of the return statemnt is of type integer
        //calls interpreter;
                ;

Body :          BEG slist retstmt END   //Return statement is set as Ptr2 of slist
                ;

slist:          slist stmt   //Appends newly created statement node to the existing.
                |stmt       //$$=$1;
                ;

stmt:           ID ASG E ';'    //Verifies if the LHS and RHS of the assignment node  is of the same type.

                |ID '[' E ']' ASG E '; //Verifies if the LHS and RHS of the Assignment node  is of the same
                type.
                                //Also type checks for array

                |READ '(' ID ')' ';'   //Verifies if the identifier is of type integer or string

                |READ '(' FIELD ')' ';'  //Verifies if the FIELD is of type integer or string

                |READ '(' ID '[' E ']' ')' ';'     //Verifies if the identifier is of type integer or string.
                                                  //Being an array, Expression node has to be of type integer.

                |WRITE '(' E ')' ';'    //Verifies if the Expression node is of type integer or string

                |IF '(' E ')' THEN slist ENDIF ';'
                        //Verifies if the Conditional Expression node is of boolean type.

                |IF '(' E ')' THEN slist ELSE slist ENDIF ';'
                        //Verifies if the Conditional Expression node is of boolean type.
```

|WHILE '(' E ')' DO slist ENDWHILE ';'
//Verifies if the Conditional Expression node is of boolean type.

|ID ASG ALLOC '(' ')' ';' //Verifies if the identifier is of user defined type.

|FIELD ASG ALLOC '(' ')' ';' //Verifies if the FIELD is of user defined type.

|FIELD ASG E ';'
//Verifies if the left hand side and right hand side of the Assignment statement is of same type.

|DEALLOC '(' ID ')' ';' //Verifies if the field is of user defined type.

|DEALLOC '(' FIELD ')' ';' //Verifies if the FIELD is of user defined type.
;

FIELD :ID DOT ID //The Type field for the identifiers are set.
//Example: In the reduction of a.b, the Type field of a is set based on the symbol table entry.
//The Type field of b is set to that specified in the fieldlist of the Typetable entry for a

|FIELD DOT ID
//The Type field of the identifier is from the fieldlist entry of the TypeTable.
;

E: E AROP1 E //Verifies if both the expression is of type integer.

|E AROP2 E //Verifies if both the expression is of type integer.

|'(' E ')' //$$=$2;

|E RELOP E //Both the expression has to be of either integer or string.
//User defined variables are allowed only for the relational operators '==' and '!='

|E LOGOP E //Verifies if both the expression is of type boolean

|NOTOP E //Verifies if the expression is of type boolean or null

|NUM //$$=$1;

|STRCONST //$$=$1;

|ID //type field of the identifier is set to that specified in the symbol table.

|ID '[' E ']' //Type field in the Tnode is set to that specified in the symbol table.
//Verifies if the expression node is an integer

|FIELD //$$=$1

|null //$$=$1

|ID '(' param ')'

```
                    //Verifies for the parameter compatibility in function declaration and calling.
                    //Type of the identifier is set to that specified in the global symbol table during declaration.
                    //The Argument list created before is set to the Arglist field.

                    ;

param :             param ',' E     //Creates a statement node and its Ptr1 field is set to the expression node.
                                    //Appends the newly created parameter list to the existing.
                    |E              //Creates a statement node and its Ptr1 field is set to the expression node.

                    ;

retstmt : RETURN E ';'              //Appends expression to the return statement.
                    ;
```

Chapter 4

Data Structures used for Implementation

4.1 Type Table

The Type Table stores all necessary information regarding the various user defined types in the source program. The compiler creates an entry in the Type Table for each user defined type. In addition to this, there are default entries created for primitive types (int, str) and types internal to the interpreter (null, Boolean and void). Entries for all internal types and primitive types are made beforehand whereas entries for user defined types are made as the Type Declaration Section of the source code is parsed.

4.1.1 Structure

The following structure can implement a Type Table:

```
typedef struct TypeTable
{
char *name; // type name
filedList *fields; //fieldlist  permits only 8 member fields
struct TypeTable *next;
}TypeTable;
```

- The name field stores the type name such as integer or a user-defined mytype.
- The fieldList stores information regarding the different fields in case of a user

defined type. We will use a linked list to store the the name and type of all the fields. The pointer to the head of this list is stored in fieldlist.

```
typedef struct fieldList
{
char *Name;
struct TypeTable *type;
struct fieldList *next;
}fieldList;
```

The name field is set to the name of the field (as a string) and type stores a pointer to entry for its type in the Type Table. The next field stores pointer to the node carrying information about the next field.

• The next field stores pointer to the next type table entry.

4.1.2 Associated methods:

• void TTableCreate(): Function to initialize the type table with entries for primitive and internal data types.

• TypeTable* TLookUp(char *name): Returns pointer to type table entry of type name.

• TypeTable* TInstall(char *name, fieldList *fields): Creates a type table entry for user defined type of given name with given fieldlist.

• TypeTable* TAppend(TypeTable *t1): Updates the Type Table by appending the entry for the type whose information is stored in t1 into it.

4.1.3 Illustration:

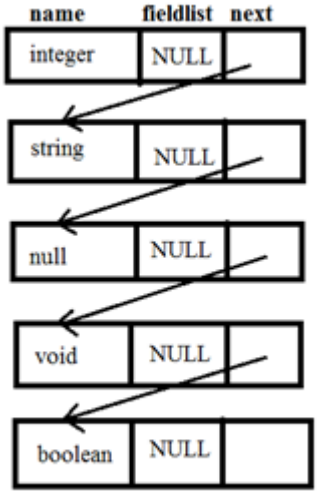
Let us consider the following sample code:

```
decl
```

```
linkedlist {
int data;
linkedlist *next; }
marklist{
str name;
linkedlist *marks; }
enddecl
// Global Declarations
//Functions.
```

1. The type table is first created and initialized to contain the default entries for each of the primitive data types and internal types. This is done through a call to function, void TTableCreate() from the main function even before yyparse() is called to start parsing the code.

On the completion of Step 1, the created type table would look like this:

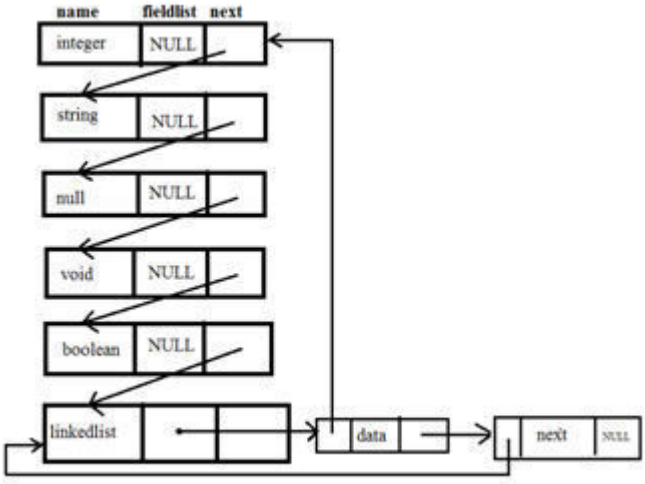


2. Now yyparse() is invoked. It starts parsing the Type Declaration Section

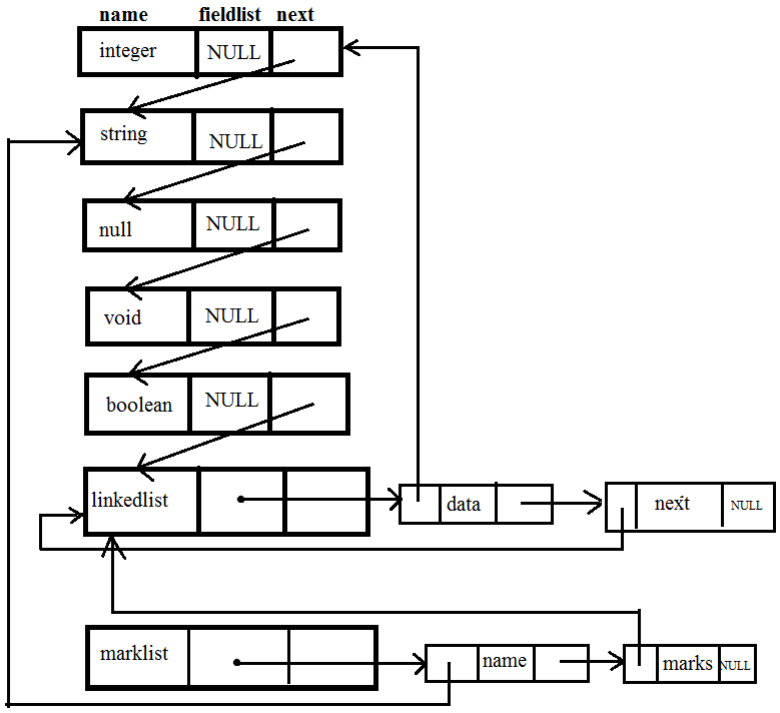
of the source code. As the definitions of user defined types are encountered, they are entered in the type table. Consider the first type declaration:

linkedlist {
int data;
linkedlist *next; }

When the corresponding reductions take place, an intermediate linked list,(intermList{ char *type; char *name }) is created containing the names and types of all the fields in the field list of the user defined type. Note that the type is stored as a string and not as the pointer to its type table entry as required in the type table. This is because when these reductions take place we dont yet have an entry for linkedlist itself. So fields of the user defined type that is being currently declared cannot store a pointer to type table entry to indicate type. Hence the type name is stored as a string and a call is made to function, TypeTable* TInstall(char *name, intermList *fields)when the final reduction for ID { fieldlist } happens to create the required type table entry. It is then passed to the function, TypeTable* TAppend(TypeTable *t1)to be appended to the Type Table.



3. Similar actions are carried out for the second user defined type declaration. The Type table would now look like this:



4. Once the type declaration section is completely parsed, the Type Table is fully created and will not be further modified or updated.

We will see when and how this table is used for interpretation later on.

4.2 Symbol Tables

Symbol Tables are maintained by the compiler to store all necessary information regarding the variables used in the source program. Information regarding global variables is stored in the Global Symbol Table and that of local variables are stored in the Local Symbol Table.

4.2.1 Global Symbol Table

Structure: typedef struct GSymbol {

char *name;

TypeTable *type;

int size;

int BINDING;

struct ArgStruct *ARGLIST;

struct AST *FBINDING;

struct GSymbol *next;

} GSymbol;

- The name field stores the name of the variable.
- The type field stores the pointer to the Type Table entry of its type.
- The size field stores the size in case of array. (The sizes of all other variables are 1.)
- BINDING stores the memory address allocated to the variable.
- ARGLIST stores the argument list in case of functions.
- FBINDING stores pointer to the root of AST of the function (in case of functions.)
- The next field stores the pointer to the next entry in the Global Symbol Table.

Associated functions:

- GInstall(char*name, TypeTable *type, int size) : Creates as Global Symbol Table entry of given name, type and size.

• Glookup(char *name) : Returns pointer to symbol table entry of variable of given name.

• GAppend(GSymbol *g1,GSymbol *g2) : Updates the Global Symbol Table by appending the entries for the variables whose information is stored in list pointed to by g2into the end of the list pointed to by g1.

• AddGType(TypeTable *gtype,GSymbol *g) : Updates the Global Symbol Table entry g by setting its type field to the Type table entry pointed to by the parameter, gtype.

4.2.2 Local Symbol Table

Structure typedef struct LSymbol {

char *NAME;

TypeTable *Type;

int BINDING;

struct LSymbol *NEXT;

} LSymbol;

•]The name field sores the name of the variable.

•]The type field stores the pointer to the Type Table entry of its type.

•]BINDING stores the memory address allocated to the variable.

•]The next field stores the pointer to the next entry in the Global Symbol Table.

Associated functions

• LInstall(char*name, TypeTable *type, int size) : Creates as Local Symbol Table entry of given name, type and size.

• Llookup(char *name) : Returns pointer to symbol table entry of variable of given name.

• LAppend(LSymbol *l1) : Updates the Local Symbol Table by appending the entry for the variable whose information is stored in l1into the it.

• AddLType(TypeTable *ltype,LSymbol *l) : Update the Local Symbol Table entry l by setting its type field to the Type table entry pointed to by the parameter, ltype.

Chapter 5

Semantic Analysis

We use the following function to type check the source code:

void CheckType(int type, AST *t1, AST *t2, AST *t3)

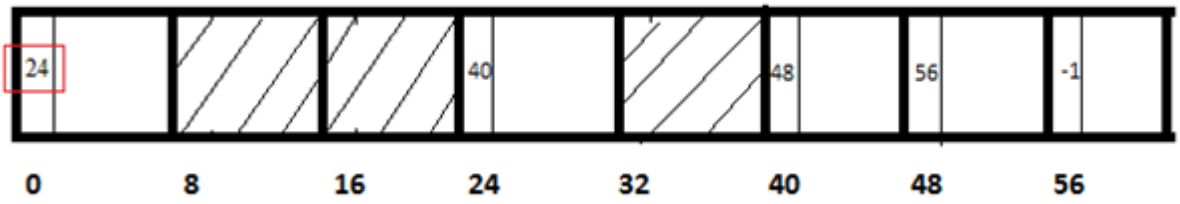
The type parameter indicates the kind of type check that is to be carried out on the three nodes under consideration, t1, t and t3. This type checker is called each time certain reductions are made with the appropriate arguments. For example, when ID '[' E ']' is being reduced to E, we call CheckType(ARRAY,$3,$1,NULL) where ARRAY stands for an integer constant that indicates that t1 and t should indicate the name and valid index of a valid array and that array index is not out of bounds. Similarly, Arithmetic expressions, Logical expressions, read and write statements, Assignments, etc also need to be checked for consistency.

Chapter 6

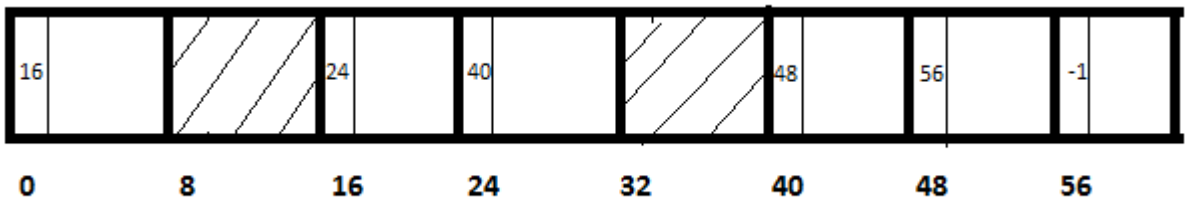
Dynamic Memory Allocation

Dynamic memory allocation and deallocation (in chunks of size 8) is supported by the language. Memory is allocated from a separate area called the heap. The algorithms for allocation and deallocation are given below.

The Free List: All the blocks that are not allocated are arranged in a list called the free list. Lets say blocks 2 (indices 8-15) and 5(32-39) are not free. The rest of the blocks constitute the free list which is maintained as shown:

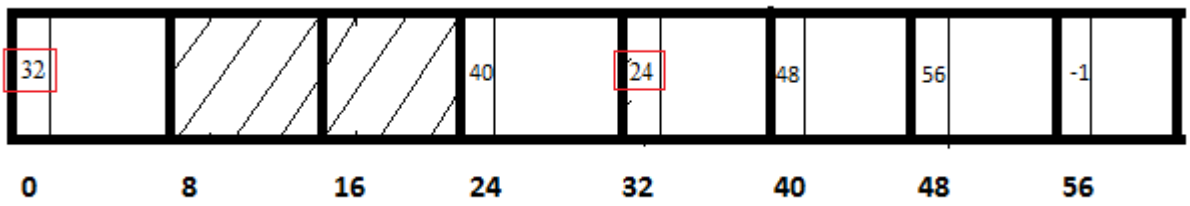


The first block is reserved and its first index always stores the index of the first free block. The first index of every free block stores the index of the next free block. When a block (say block 3) is allocated, we remove it from the free list as shown:



Note that since the free list now starts with 24, the value at 0 has now changed. Now if one block were to be deallocated, (say block 5), the list would have to be

updated accordingly:



6.1 Algorithms

Algorithm 2 int Alloc()

bind1 ← *heap*[0].*Bind*
*bind*2 ← *heap*[*bind*1].*Bind*

heap[0].Bind ← *bind*2
*returnbind*1

Algorithm 4 void DeAlloc(int bind)

bind1 ← *heap*[0].*Bind*
bind[0].*Bind* ← *bind*

heap[bind].Bind ← *bnd*1

Chapter 7

Interpreter

When the whole source code is parsed, the Symbol Tables, Type Table and the Abstract Syntax Tree will have been created. We will now see how these structures are used to evaluate the program. The memory is simulated using the following global arrays:

Reg[1000] for static memory.

Heap[1000]- for dynamic memory.

Stack[1000]- for the stack.

Evaluation starts from the root of the AST of the main function and proceeds by evaluating the left and right children (and in case of function call, a whole other AST) as required. The following summarizes the actions of the interpreter in case of each kind of node it encounters:

- Statement connector node - interpret() is called on its left child and right child.
- Read/Write node The left child contains the identifier indicating which variable value is to be read to/written from. The binding of this variable found from the Global Symbol Table with the help of GLookUp() and the read value is stored to reg[binding] / retrieved from reg[binding] and printed.

NOTE: In case of local variables, binding may be found from the Local Symbol Table. Since this binding refers to index in the stack, stack[binding] should be

used. In case of user defined types, binding refers to index in the heap since they are allocated dynamically. Hence heap[binding] should be used.

• If node Ptr1(condition) is first evaluated. If the condition evaluated to true, Ptr2 is evaluated. Otherwise, Ptr3 (else part) is evaluated.

• While node- Ptr1 (condition) is first evaluated. If the condition evaluated to true, Ptr2 is evaluated. This is repeated till condition evaluates to false.

• Assignment node- The left child contains the identifier indicating the variable to which assignment is to take place. The binding of this variable is found from Global/Local Symbol Table using GlookUp()/LlookUp(). The right child is then evaluated and the result is stored in reg/heap/stack [binding] correspondingly.

• Constant node- its value is simply returned.

• Variable node- The binding of the variable is found from Global/Local Symbol Table using GlookUp()/LlookUp(), its value is retrieved from reg/heap/stack [binding], and returned.

• Operator node- The operands are found as its children. Hence the left and right children are evaluated and the required operation is performed on the results. The final result is returned.

• Alloc node- Memory is allocated as per the strategy mentioned in the previous section and index of newly allocated memory in heap is returned.

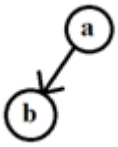
• DeAlloc node - Memory is deallocated as per the strategy mentioned in the previous section.

7.1 User defined type and their field

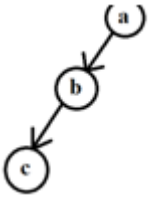
:

The field b of a variable, a of a user defined type is represented in the AST like this:

a.b.c would be represented like this:



Hence each time we find that the current node to be evaluated is of user defined type, we go the left most descendant to get its value.



7.2 Function calls

Function calls are managed with the help of stack. We globally maintain a base pointer(BP) and stack pointer(SP) to indicate the Activation Record. BP points to the base of the active region f the stack and SP increments as the stack grows to indicate the top of the stack. The part of the stack in between BP and SP is the activation record or the region that is currently active. When a function call node is encountered, its current BP value is pushed to the stack followed by its arguments. (SP is incremented accordingly and BP is set to SP-number of arguments to indicate the active region for the new function.) Hence, the parameters of the function will have bindings starting from 0 and the rest of the local variables will be allocated bindings below them. The pointer to the old Local Symbol Table is backed up in a variable l and the LST is set to the pointer to the Local Symbol Table of the new function (found in LEnrty field of the function call node). Interpret() is called on the function body. (The pointer to the root of AST for the function can be found from the Global Symbol Table in the field FBinding of the entry for the function.)The return value of the function

is returned by interpret(). Now, the local variables and arguments are popped out (SP decremented accordingly) and BP is set to the old BP popped from the stack and rest of the AST continues to be evaluated.

Chapter 8

Conclusion

Our intention was to build an educational platform for compiler design. The specification of ExpL was developed with this in mind.It touches upon the major aspects of a programming language like functions, types (primitive and user defined), global and local scope, dynamic memory allocation and deallocation, etc. Advanced concepts like object oriented programming are not coevered by ExpL. Besides fixing the language specification, we have implemented an interpreter for it. Though this project will not be fully complete withput writing a compiler for it as well, we have gained some insight into the various complications that might arise when students attempt to write the interpreter/compiler for this language, such as deciding what Data Structures to use. Hence we have also developed supporting materials that will aid them with the necessary information as well as a tutorial for Lex and Yacc. This work is a continuation of the work done by Nachiappan V P last year. The documentations may be found at silcnitc.github.io/Documentation. This project has given us insight into various aspects of compiler design. Future directions that this project can take is introduce features such as System calls and design the compiler to generate assembly code that is consistent with the ABI provided by ExpOS. This will help the students get the complete picture of how the high level language is converted to an assembly language which then executes on the operating system. Further, libraries can be written for dynamic memory management and the linking and loading procedure may also be explored.

Bibliography

[1] Muralikrishnan, K., Nachiappan, V., and Rajan, A. Simple integer language : Nitc.

[2] V.Aho, A., S.Lam, M., Sethi, R., and D.Ullman, J. Compilers : Principles, Techniques and Tools. 1986.