EXPI

Home About Help Roadmap Documentation

EXPL SPECIFICATION

Download as PDF

➤ Introduction
➤ Supported data types
➤ General Program Structure
➤ Statements and Expressions
➤ Dynamic Memory Allocation
➤ Parameter passing
➤ Sample Programs
➤ Appendix

Introduction

ExpL (Experimental language) is the language for which we will build a compiler in this course. This document provides an informal specification for the language.

Supported Data Types

Primitive data types

Integer : An integer type variable is declared using the keyword int. *Example* :

```
int a, b, c; /* Declares variables a, b, c of type integer */
```

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. */
```

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.

Composite data types

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 */
```

User-defined types

ExpL allows user defined data types. The (member) fields of a user defined type may be of type a) integer, b) string, c) a previously defined user defined type or d) the type that is currently being defined.

Note : ExpL specifies that the store for variables of user defined types shall be dynamically allocated. Hence the programmer has to call the library function *alloc()* to allocate memory for variables of user defined types. User defined types take default value NULL unless allocated or assigned otherwise. Store allocated for a variable of a user defined type is de-allocated using the *free()* library function.

Example : A user defined type, mytype is defined as:

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

a variable of type mytype is declared as:

```
mytype var1, var2;
```

Storage for the variable is allocated as:

```
var = alloc(); /* Note: Access without allocation can lead to run time
errors */
```

Its fields may be accessed as:

```
var.a = 10; /* the "." symbol is used to access member fields */
```

The memory allocated may be freed as:

```
retval = free(var);
```

Note : The ExpL compiler may internally represent var like a pointer variable. The library function alloc may be designed so as to allocate memory and return a pointer to the allocated memory. The returned pointer is stored in var. (Library functions are explained in detail later on.)

General Program Structure

An ExpL program consists of the following sections:

- Type Definitions - Optional (for user defined types)
- Global Declarations - for global variables, arrays and functions
- Function Definitions and the main Function Definition

.

The following subsections explain each program section.

Type Definitions

All user-defined types in a program must be defined in the type definition section. The Type Definition section starts with the keyword type and ends with the keyword endtype.

Example

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

illustration_1.1 hosted with ❤ by GitHub view raw

Global Declarations

The global declaration part of an ExpL program begins with the keyword decl and ends with the keyword enddecl. All global variables, arrays and functions in a program must be declared in this section.

Global variables may be of type integer, string, a user defined type, an integer array, a 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 special main function defined in an 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. The return type of a function also can be integer/string/user-defined type. ExpL enforces call-by-value semantics for integer and string parameters and call-by-reference for user-defined types. (A variable of a user defined type typically stores a reference to its store.) Arrays cannot be passed as arguments. If a global variable name appears as an argument of a function, then within the scope of the function, the new declaration will be valid and the global declaration is suppressed. Different functions may have arguments of the same name. However, the same name cannot be given to two or more arguments in a function. The general form of declarations is as follows:

```
type VarName1, VarName2 ; /* variable declarations */
rettype FunctionName (ParameterList); /* A function declaration */
type VarName[ArraySize]; /* An array declaration */
```

Note : Declarations for variables/functions of the same type can be combined as shown in the following example.

Example :

```
decl /* Please note the use of "," and ";" */
  int x,y,a[10],b[20]; /* x,y are integers, a,b are integer arrays */
  str t, q[10], f3(str x); /*variable, array and a functions declared
together*/
  mytype m, fun(mytype t); /* myptype must be a user defined type */
  /* The argument and the return value of fun are references to mytype */
enddecl
```

Declaring functions at the beginning avoids the forward reference problem and facilitates single pass compilation. If a variable/function is declared multiple times, a compilation error should result.

Function Definitions and the Main Function

All globally declared variables are visible inside a function, unless suppressed by a re-declaration within the function. 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 names and types of the arguments and return value of each function definition should match exactly (*name equivalence*) with the corresponding declaration. Every declared function must have exactly one 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. The scope of a parameter is limited to the function. Static scope rules apply. A function can have a user defined type as its return type. Similarly, parameters to a function can be of user defined types.

The main() function, by specification, must be a zero argument function of return type integer. It must be defined immediately after declaration section, before all other functions are defined. Program execution begins from the body of the main function. The main function must 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 : The following is an example for a simple function definition.

```
int fun(int a,int b)
{
  decl
    int c,d;
  enddecl
  begin
    c = a + b;
    d = a - b;
    write(c);
    write(d);
    return c;
  end
}
```

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

The next section discusses statements and expressions in ExpL.

Statements and Expressions

Before taking up statements, we should look at the different kinds of constants and expressions in the language. ExpL has four kinds of expressions, a) Arithmetic expressions, b) String expressions, c) Logical expressions and d) Expressions that evaluate to user defined types.

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).

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 paranthesization apply. ExpL is strongly typed and any type mismatch or scope violation must be reported at compile time.

Examples : _5, a[a[5+x]]+x , (f2() + b[x] + 5), sum + listObject.data , a[listObject.data] + f2(listObject) * 8_ etc. are arithmetic expressions, provided type and scope rules are not violated. Note that the function f2() must have return type int for the expression to be valid.

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.

Variables of user defined types can only be checked for equality/inequality using the ==/!= operator.

Example : *((x==y)==a[3])* is not valid ExpL expression because (x==y) is a logical expression, while a[3] must be a variable of type integer/string. The "==" operator can be applied only between expressions of the same type.

(“hello”==a and "hello" < a) are a valid logical expressions provided, a is a variable of string type. (p==q) is a valid logical expression provided p and q are variables of the same type.

SetOne.name == "a", SetOne.total >= SetTwo.total are valid only if SetOne is a user defined type with a field 'name' of type string and another field 'total' of type integer.

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.

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" ;* are both valid assignments to string variables provided q is declared as an array of type string and t is declared as a variable of type string. *x=y*; is valid if x and y are of the same type and scope rules are not violated.

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 variables of a user defined type, the assignment only makes both x and y refer to the same memory object. This is because a variable of a user defined type stores a reference to its store allocated using alloc().

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.

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.

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 syntax is:

```
return < Expression* > ; /* The type of the expression should match with
the return type of the function*/
Note* : As an exception to the rule above, the expression returned by a
function whose return type is a user defined type can be the constant
NULL.
```

If the return type of the function does not match the type of the expression/variable returned, a compilation error should occur. The return type of a function can be of type int , str or user-defined type. The return type of main is integer by specification

.

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 > );
```

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.

The next section briefly discusses the library functions for dynamic memory allocation.

Breakpoint Statement

This statement results in the ExpL compiler setting a break point in the program. This feature is useful for debugging.

```
breakpoint;
```

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.

Parameter passing

Functions can take any number of arguments and must return a single value. The arguments and return values may be of integer, string or user defined types.

Arguments to functions are passed by value. This means that changes to made to the values of an argument inside a function will not be visible outside the function.

However, if a variable is of a user defined type, after allocation using the Alloc() function, the variable stores the memory address of the actual data in heap memory where the data associated with the variable is stored. ExpL specification allows the data to be modified using the variable. If the data stored variable is changed with in a function, the change will be visible globally. This is because heap memory is globally visible.

The scope rules are illustrated by the following example.

```
type
node {
int x;
}
endtype
decl
node fun(node t);
enddecl
node fun(node t) { // t and r points to the same heap memory location.
begin
t.x=3; // modifies heap, change is visible in main.
t = null; // r does not become NULL in main() by this assignment
return t;
end
}
int main()
{
decl
int val;
node r, s;
enddecl
begin
val = initialize();
```

```
r = alloc(); // r stores address of a node in the heap memory
r.x = 5;
s = fun(r);
write(r.x); // Now 3 will be output.
return 0;
end
}
```

functions and parameter passing hosted with ❤ by GitHub view raw

---

Appendix

Keywords

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

read	write	if	then	else	begin	initialize
endif	do	endwhile	break	while	end	
int	str	return	decl	enddecl	alloc	
type	endtype	NULL	continue	main	free	

Operators and Delimiters

The following are the operators and delimiters in ExpL

>	<	>=	<=	!=	==	(	)
{	}	[	]	/	;	*	=
+	-	%	AND	NOT	OR	.	

Identifiers

Identifiers are names of variables and user-defined functions. Identifiers should start with an letter, and may contain both letters and digits. Special characters are not allowed in identifiers.

letter -> [a-z]|[A-Z]

digit -> [0-9]

identifier -> (letter)(letter | digit)*

Github

Contributed By : Ashwathy T Revi, Subisha V, Nunnaguppala Surya Harsha, Vishnu Priya Matha

Home | About