PRINCIPLES OF PROGRAMMING LANGUAGE

MID SEMESTER REPORT

By

Kevin Roy 2015CSE064

Krishnaraj – 2015CSE067

Renjith E Chacko - 2015CSE111

Date of submission: 07th November 2017

GAIN MORE KNOWLEDGE
REACH GREATER HEIGHTS

PRESIDENCY UNIVERITY,

BENGALURU

1. ABSTRACT

The name of the new programming language is BCG.

Comparative study of 3 programming languages from different domains – Swift , Rust , Julia, which includes comparision between identifiers, datatypes, statements, operators, iterative and conditional statements , enumeration, arrays, sub programs and so on.

Requirement analysis of the new programming language such as overall design of the language, implementation and execution of the programs.

Runtime environment analysis of the language such as memory management, recursion, lifetime, scoping and support of function

Designing of the imperative programming language which inclues design of syntax, datatypes, operators, statements and sub programs or functions.

Case study on the language with derivations to prove that the language is grammatically correct.

Keywords: data types, procedure, itereative, conditional, syntax, structures.

2. INTRODUCTION

We three (Kevin Roy, Krishnaraj P.B. and Renjith E Chacko) have studied three different languages from three different domains i.e, Mobile application, systems programming, technical computing and tried to to design a new programming language with a purpose and goal to attain.

The name of the language we are working on is BCG, a procedural language, easy to understand and implement. Our language covers most of the topic a simple programming should have i.e, Syntax – the syntax we used in our language is quite simple and straight forward and very easy to implement.

3. LITERATURE SURVEY:

The three programming languages chosen to be compared are :

1) Swift
   Domain: Mobile Application
   Swift is a new programming language developed by Apple Inc for iOS and OS X development. Swift adopts the best of C and Objective-C, without the constraints of C compatibility. Swift designers took ideas from various other popular languages such as Objective-C, Rust, Haskell, Ruby, Python, C#, and CLU.

Swift uses the same runtime as the existing Obj-C system on Mac OS and iOS, which enables Swift programs to run on many existing iOS 6 and OS X 10.8 platforms.

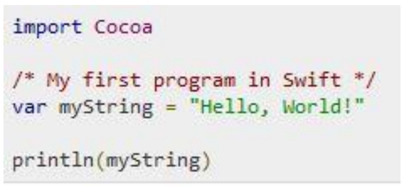
2) Rust
Domain: Systems Programming
Rust is a systems programming language sponsored by Mozilla Research. It is designed to be a "safe, concurrent, practical language", supporting functional and imperative-procedural paradigms. Rust is syntactically similar to C++, but is designed for better memory safety while maintaining performance.
Rust is an open source programming language. The design of the language has been refined through the experiences of writing the Servo web browser layout engine and the Rust compiler. A large portion of current commits to the project are from community members.

3) Julia
Domain: Technical Computing
Julia is a high level dynamic programming language designed to address the needs of high-performance numerical analysis and computational science, without the typical need of separate compilation to be fast, while also being effective for general-purpose programming, web user as a specification language.
Julia aims to create an unprecedented combination of ease-of-use, power, and efficiency in a single language.

I. Import

Swift

The import statement is used to import any Objective-C framework directly into a Swift program.



```
import Cocoa

/* My first program in Swift */
var myString = "Hello, World!"

println(myString)
```

Justification: We don’t need to import a separate library for functionality like input/output or string handling. Code written at global scope is used as the entry point for the program, so you don’t need a main function. Semicolons are not necessary at the end of every statement.
Grammar of an import declarationimport-declarationattributesimportimport-kindimport-pathimport-kindtypealiasstructclassenumprotocolvarfuncimport-pathimport-path-identifierimport-path-identifier.import-pathimport-path-identifieridentifieroperator

Rust
In rust there are no import or header files. Instead they have a module system which enables a

user to not having to write function signatures twice, keep them in sync, etc. Justification: Unlike many other languages a user doesn't have to import various inbuilt functions according to the necessity by using header files. In rust all the inbuilt functions are readily available to the user by default.

Julia

julia code is organized into files, modules, and packages. Files containing Julia code use the .jl file extension. There are two ways in which a function can use data from another module in julia. Import and Using

Justification-the using statement, which accepts the names of one or more installed modules for the user and can be accessed outside the module And the Import function could be accessed only within the module. This is to help the program to perform different modules in in different ways required in the function of module.

II. Comments

Swift

Comments are like helping texts in your Swift program. They are ignored by the compiler. Multi-line comments start with /* and terminate with the characters */. For example: /* My first Program */

Justification: Comments are ignored by the Swift compiler when your code is compiled. Comments in Swift are very similar to comments in C. Single-line comments begin with two forward-slashes (//). Unlike multiline comments in C, multiline comments in Swift can be nested inside other multiline comments. Nested multiline comments enable you to comment out large blocks of code quickly and easily, even if the code already contains multiline comments.

Rust

Comments are notes that you leave to other programmers to help explain things about your code. The compiler mostly ignores them. Rust has two kinds of comments that you should care about: line comments and doc comments. '//' is used for line comments and '///' '//!' is used for doc comments

Justification: In Rust, comments must start with two slashes and continue until the end of the line. For comments that extend beyond a single line, you'll need to include // in each line. When the compiler encounters // it ignores the whole line from the point it read //. need to include // on each line,

Julia

They are added with the purpose of making the source code easier for person to understand, are generally ignored by compilers and interpreters. And provides information to the person who reads the program to understand the purpose of the line or function. There are two ways to comment in julia ie single line comment and multi-line comment.

Justification:- begin single line comment (#) is used for one line comment in the program .Begin multiline comment(#=) more than then one line and for nested comments for the programmer end multi line comment(=#) This are used for commenting

III. Identifiers

Swift

A Swift identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with an alphabet A to Z or a to z or an underscore _ followed by zero or more letters, underscores, and digits (0 to 9).Swift does not allow special characters such as @, $, and % within identifiers. Swift is a case sensitive programming language.

Rust

In rust an identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with an alphabet A to Z or a to z or an underscore _ followed by zero or more letters, underscores, and digits (0 to 9).Rust does not allow special characters such as @, $, and % within identifiers.

Justification: Rust identifiers are same as most of the languages because they are easy to identify by the users.

Julia

In julia programming, identifiers are name associated given to julia entities, such as variables, functions, structures etc.

Justification Identifier are created to give unique name to julia entities to identify it during the execution of program.It provides user to extremely flexible environment for naming variable ,function, structure. This help the language to not treat variable on different name and the name can be case sensitive or no meaning.

```
# You can assign values of other types, like strings of text

julia> x = "Hello World!"

"Hello World!"
```

IV. Data types

Swift

Int or UInt − This is used for whole numbers. More specifically, you can use Int32, Int64 to define 32 or 64 bit signed integer, whereas UInt32 or UInt64 to define 32 or 64 bit unsigned integer variables. For example, 42 and -23.
Justification : Swift provides signed and unsigned integers in 8,16,32 and bit forms. Like all

types in swift, these integer types have capitalized names. Int is used for signed integer and UInt is used for unsigned integer type.

Float − This is used to represent a 32-bit floating-point number and numbers with smaller decimal points. For example, 3.14159, 0.1, and -273.158.

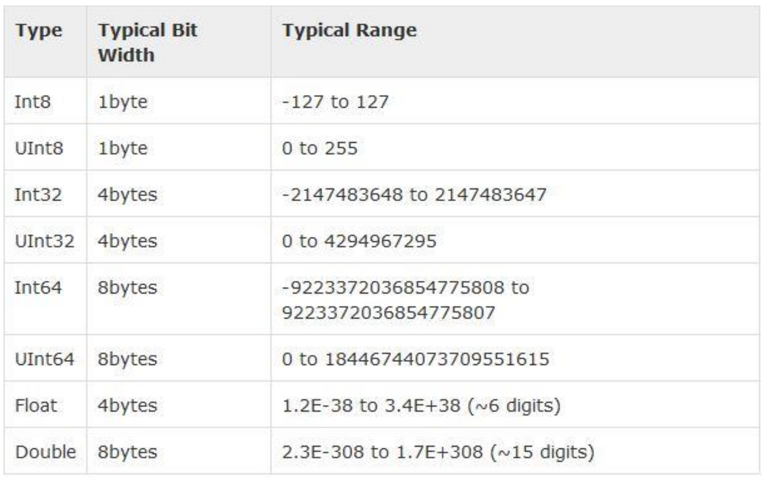
Justification: Double − This is used to represent a 64-bit floating-point number and used when floating-point values must be very large. For example, 3.14159, 0.1, and -273.158.

Bool − This represents a Boolean value which is either true or false.

String − This is an ordered collection of characters. For example, "Hello, World!" Justification: Swifts String and character types provide a fast, Unicode-compliant way to work with text in code. Swift’s String type is a fast, modem string implementation. Every string is composed of encoding-independent Unicode characters, and provides support for accessing those characters in various Unicode representations.

Character − This is a single-character string literal. For example, "C"

Optional − This represents a variable that can hold either a value or no value.



Type	Typical Bit Width	Typical Range
Int8	1byte	-127 to 127
UInt8	1byte	0 to 255
Int32	4bytes	-2147483648 to 2147483647
UInt32	4bytes	0 to 4294967295
Int64	8bytes	-9223372036854775808 to 9223372036854775807
UInt64	8bytes	0 to 18446744073709551615
Float	4bytes	1.2E-38 to 3.4E+38 (~6 digits)
Double	8bytes	2.3E-308 to 1.7E+308 (~15 digits)

Rust

Booleans : Rust has a built-in boolean type, named bool. It has two values, true and false:

let x = true;
let y: bool = false;

Character: The char type represents a single Unicode scalar value. You can create chars with a single tick: (')

let x = 'x';

Unlike some other languages, this means that Rust’s char is not a single byte, but four.

Numeric types:

- i8
- i16
- i32
- i64
- u8
- u16
- u32
- u64
- isize
- usize
- f32
- f64

i- Integer, u- Unsigned integer, f- Float and the number indicates the number of bytes assigned.

Justification: If you're unsure, Rust's defaults are generally good choices, and integer types default to i32: it's generally the fastest, even on 64-bit systems. The primary situation in which you'd use isize or usize is when indexing some sort of collection.

Floating-point numbers are represented according to the IEEE-754 standard. The f32 type is a single-precision float, and f64 has double precision.

Rust's char type represents a Unicode Scalar Value, which means it can represent a lot more than just ASCII. Accented letters, Chinese/ Japanese/Korean ideographs, emoji, and zero width spaces are all valid char types in Rust. Unicode Scalar Values range from U+0000
to U+D7FF and U+E000 to U+10FFFF inclusive. However, a "character" isn't really a concept in Unicode, so your human intuition for what a "character" is may not match up with what a char is in Rust.

Julia

Like most languages, Julia language defines and provides functions for operating on standard data types such as

- integers
- floats
- strings

Integers and Floating-Point Numbers

Integers and floating-point values are the basic building blocks of arithmetic and computation.

Integer types:

Type		Number of bits	Smallest value	Largest value
Int8		8	-2^7	2^7 - 1
UInt8		8	0	2^8 - 1
Int16		16	-2^15	2^15 - 1
UInt16		16	0	2^16 - 1
Int32		32	-2^31	2^31 - 1
UInt32		32	0	2^32 - 1
Int64		64	-2^63	2^63 - 1
UInt64		64	0	2^64 - 1
Int128		128	-2^127	2^127 - 1
UInt128		128	0	2^128 - 1
Bool		8	false (0)	true (1)

Floating-point types:

Type	Precision	Number of bits
Float16	half	16
Float32	single	32
Float64	double	64

Integers

```
julia> 1
```

```
1
julia> 1234
1234
```

Floating-Point Numbers

```
julia> 1.0
1.0
julia> 1.
1.0
```

Justification:-In julia 1 is an integer , while 1.0 is a floating-point.The default type for an data depends on whether the system has a 32-bit architecture or a 64-bit architecture. Integer and float-point numbers are represented in the standard library to make the user more easy to compute the values

Strings

Strings are finite sequences of characters used in programming language.

Justification:- Like C and Java, but unlike most dynamic languages, Julia has a first-class type representing a single character, called Char.

Julia supports the full range of Unicode characters via the UTF-8 encoding. So the is eliminates data corruption and other problems due to incompatible codes ,for different language conversion

V. Arrays

Swift

Syntax:

```
var someArray = [SomeType](count: NumbeOfElements, repeatedValue: InitialValue)
```

Example:

```
import Cocoa

var someInts = [Int](count: 3, repeatedValue: 10)

var someVar = someInts[0]

println( "Value of first element is \(someVar)" )
println( "Value of second element is \(someInts[1])" )
println( "Value of third element is \(someInts[2])" )
```

Justification: The type of a Swift array is written in full as Array<Element>, where Element is the type of values the array is allowed to store. You can also write the type of an array in shorthand form as [Element]. Although the two forms are functionally identical, the shorthand form is preferred and is used throughout this guide when referring to the type of an array. The first item in the array has an index of 0, not 1. Arrays in Swift are always zero-indexed.

Rust

Arrays have type [T; N].T is the data type. The N is a compile-time constant, for the length of the array.

```
let a = [1, 2, 3]; // a: [i32; 3]
let mut m = [1, 2, 3]; // m: [i32; 3]
```

You can get the number of elements in an array a with a.len():

```
let a = [1, 2, 3];

println!("a has {} elements", a.len());
```

Justification: Arrays in Rust are different than arrays in some other languages because arrays in Rust have a fixed length: once declared, they cannot grow or shrink in size. When you attempt to access an element using indexing, Rust will check that the index you've specified is less than the array length. If the index is greater than the length, Rust will panic, which is the term Rust uses when a program exits with an error.

VI. Declarations

Swift

Syntax: var variableName = <initial value>

For example:

```
import Cocoa

var varA = 42

println(varA)
```

Justification: the value of a constant doesn't need to be known at compile time, but must be assign it a value exactly once. Providing a value when a constant or variable is created lets the compiler infer its type. Compiler infers that Variable name is an integer because its initial value is a integer.

Rust

They bind some value to a name, so it can be used later. 'let' is used to introduce a binding, like this:

```
fn main() {
    let x = 5;
```

}

The left-hand side of a ‘let’ statement is a pattern, not a variable name. This means we can do things like: let (x, y) = (1, 2);

Justification: In rust while declaring a variable, specifying the data type is not necessary because the compiler by default recognises the value that is stored in a variable and assigns the data type itself. By default, bindings are immutable. let mut x = 5; x = 10;Adding ‘mut’ will let you change values of a variable at any point in the program.

Julia

A julia declaration determines the name and data type of a variable or other element. Programmers declare variables by writing the name of the variable into code, along with any data type indicators and other required syntax.

Justification : the value of a constant doesn’t need to be known at compile time, but must be assign it a value exactly once. Providing a value when a constant or variable is created lets the compiler infer its type. Compiler infers that Variable name is an integer because its initial value is a integer

VII. Printing

Swift

To print the current value of a constant or variable println function can be used.

    Println("Value of \(varA) is more than \(varB) millions");

Justification: Printing variables is similar to C program. Println is a global function that prints a value, followed by a line break, to an appropriate output. In Xcode, println prints its output in Xcode’s “console” pane.

Rust

‘println!()’ . This is calling a Rust macro, which is how metaprogramming is done in Rust. If it were calling a function instead, it would look like this: println() (without the !).

println!("The value of x is: {}", x);

Justification: If you include two curly braces ({} in your string to print, Rust will interpret this as a request to interpolate some sort of value. We add a comma, and then x, to indicate that we want x to be the value we’re interpolating. The comma is used to separate arguments we pass to functions and macros, if you’re passing more than one.

Julia

Printing

julia uses similar printing code like java

```
println("hello world")
```

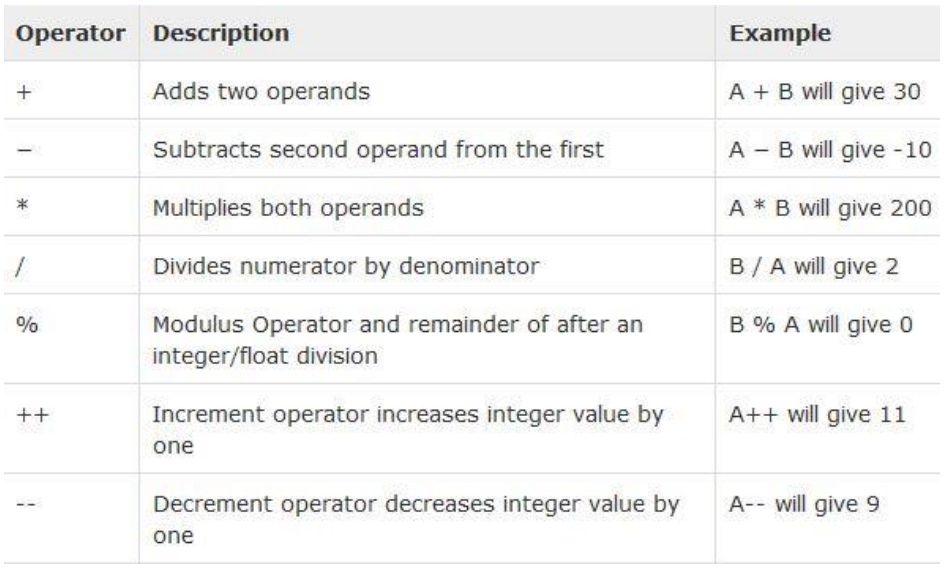
Justification: Printing variables is similar to C program. Println is a global function that prints a value, followed by a line break, to an appropriate output.

VIII. Operators

Swift

- Arithmetic Operators
- Comparison Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators

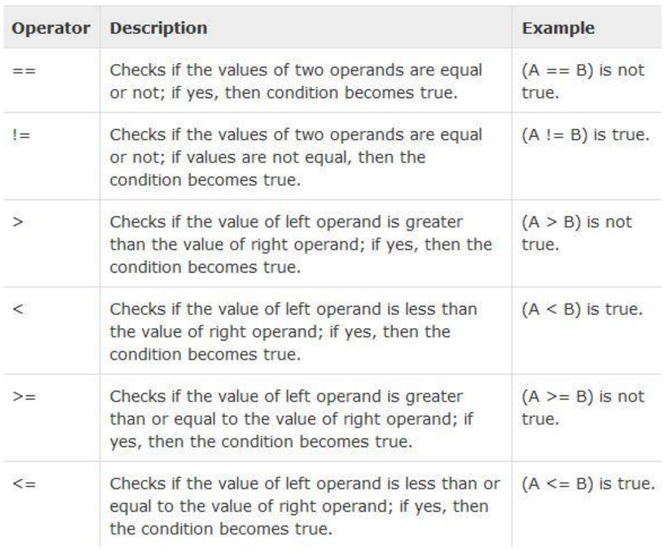
<u>Arithmetic Operators</u>



Operator	Description	Example
+	Adds two operands	A + B will give 30
−	Subtracts second operand from the first	A − B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by denominator	B / A will give 2
%	Modulus Operator and remainder of after an integer/float division	B % A will give 0
++	Increment operator increases integer value by one	A++ will give 11
--	Decrement operator decreases integer value by one	A-- will give 9

Justification: Arithmetic Operator in Swift is similar to Objective-C. But in Swift arithmetic operator don't allow values to overflow. To have Overflow in swift ampersand (&) is used. If a number is inserted into an integer constant or variable that cannot hold that value, complier reports an error rather than allowing an invalid value to be created. This behaviour gives extra safety with numbers that are too large or too small.

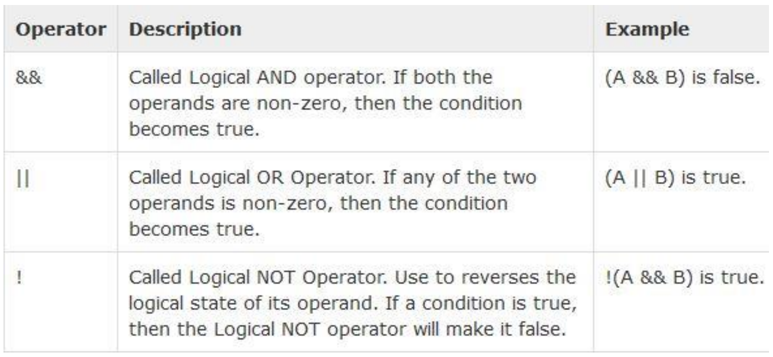
<u>Comparison Operators</u>



Operator	Description	Example
==	Checks if the values of two operands are equal or not; if yes, then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not; if values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	(A <= B) is true.

Justification: When the complier encounters these operators, it will compare two values and check of the condition is true or false. These operators is similar to objective-C and C. Two tuples are compared if they have the same type and the same number of values. Tuples are compared from left to right, one value at a time, until the comparison finds two values that aren’t equal.

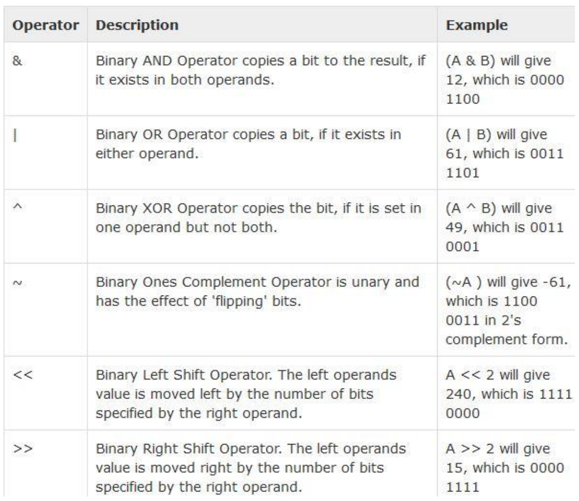
<u>Logical Operators</u>



Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then the Logical NOT operator will make it false.	!(A && B) is true.

Justification: When the complier encounters these operators it should give output as true or false. The Swift logical operators && and || are left-associative, meaning that compound expressions with multiple logical operators evaluate the leftmost subexpression first.

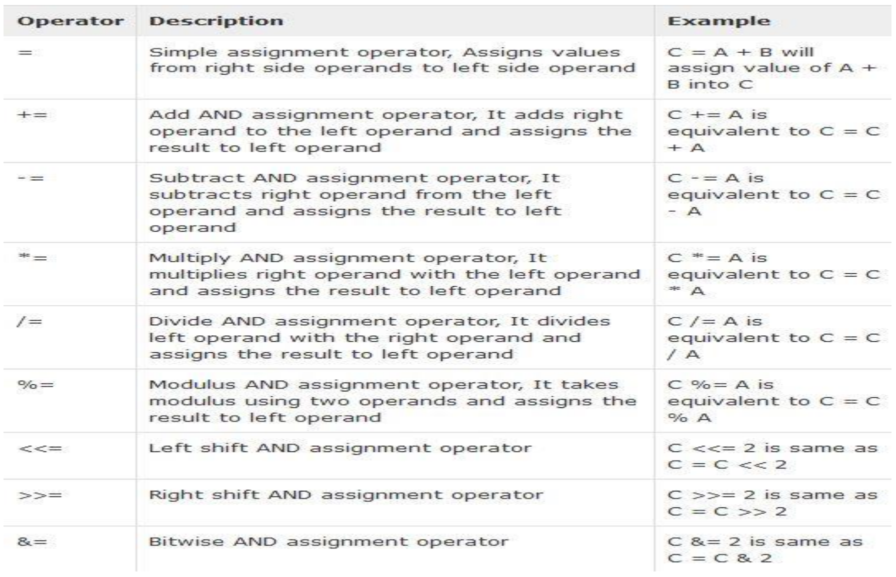
<u>Bitwise Operators</u>



Operator	Description	Example
&	Binary AND Operator copies a bit to the result, if it exists in both operands.	(A & B) will give 12, which is 0000 1100
|	Binary OR Operator copies a bit, if it exists in either operand.	(A | B) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit, if it is set in one operand but not both.	(A ^ B) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61, which is 1100 0011 in 2's complement form.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15, which is 0000 1111

Bitwise operators enable to manipulate the individual raw data bits within a data structure. They are often used in low-level programming, such as graphics programming and device driver creation. Bitwise operators can also be useful when worked with raw data from external sources, such as encoding and decoding data for communication over a custom protocol.

Assignment Operators



Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assigns the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assigns the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assigns the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assigns the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assigns the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2

Rust

Unary operator expressions

- - : Negation. Signed integer types and floating-point types support negation. It is an error to apply negation to unsigned types; for example, the compiler rejects -1u32.
- *: Dereference. When applied to a pointer, it denotes the pointed-to location. For pointers to mutable locations, the resulting value can be assigned to. On non-pointer types, it calls the deref method of the std::ops::Deref trait, or the deref_mut method of the std::ops::DerefMut trait (if implemented by the type and required for an outer

expression that will or could mutate the dereference), and produces the result of dereferencing the & or &mut borrowed pointer returned from the overload method.

- ! : Logical negation. On the boolean type, this flips between true and false. On integer types, this inverts the individual bits in the two's complement representation of the value.
- & and &mut : Borrowing. When applied to a value, these operators produce a reference (pointer) to that value. The value is also placed into a borrowed state for the duration of the reference. For a shared borrow (&), this implies that the value may not be mutated, but it may be read or shared again. For a mutable borrow (&mut), the value may not be accessed in any way until the borrow expires.

Arithmetic operators

- + : Addition and array/string concatenation. Calls the add method on the std::ops::Add trait.
- - : Subtraction. Calls the sub method on the std::ops::Sub trait.
- *: Multiplication. Calls the mul method on the std::ops::Mul trait.
- / : Quotient. Calls the div method on the std::ops::Div trait.
- % : Remainder. Calls the rem method on the std::ops::Rem trait.

Comparison operators

- == : Equal to. Calls the eq method on the std::cmp::PartialEq trait.
- != : Unequal to. Calls the ne method on the std::cmp::PartialEq trait.
- < : Less than. Calls the lt method on the std::cmp::PartialOrd trait.
- > : Greater than. Calls the gt method on the std::cmp::PartialOrd trait.
- <= : Less than or equal. Calls the le method on the std::cmp::PartialOrd trait.
- >= : Greater than or equal. Calls the ge method on the std::cmp::PartialOrd trait.

Julia

An operator in julia is a character that represents an action

lexicographically compare strings using the standard comparison operators:

```
julia> "abracadabra" < "xylophone"
true
```

You can search for the index of a particular character using the search() function:

```
julia> search("xylophone", 'x')
1
```

You can start the search for a character at a given offset by providing a third argument:

```
julia> search("xylophone", 'o')
4
```

You can use the contains() function to check if a substring is contained in a string:

```
julia> contains("Hello, world.", "world")
true
```

Justification: In julia the operators can perform basic arthematic operation and string to string comparision etc this helps the programer to check compare string pass operation in the program

IX. Functions

Swift

A function is a set of statements organized together to perform a specific task. A Swift function can be as simple as a simple C function to as complex as an Objective C language function. It allows us to pass local and global parameter values inside the function calls.

- Function Declaration − It tells the compiler about a function's name, return type, and parameters.
- Function Definition − It provides the actual body of the function.

Swift functions contain parameter type and its return types.

Function Definition

Syntax

```
func funcname(Parameters) -> returntype {
  Statement1
  Statement2
  ---
  Statement N
  return parameters
}
```

Functions without Parameters

Syntax

```
func funcname() -> datatype {
  return datatype
}
```

Justification: In Swift, a function is defined by the "func" keyword. When a function is newly defined, it may take one or several values as input 'parameters' to the function and it will process the functions in the main body and pass back the values to the functions as output 'return types'.

Grammar of a function

deoacrilntufnocintd-eoacrilntufnocinth-eadufnocintn-amegencpe-irarmeaecut-lrseufnocigntsin-auteurfnocintb-odyufnocinth-eadbrtiauetsdeoacrilntspeu-sfrienfcuicfnocintn-amerifdeiotenpeaortufrnocigntsin-auterparmeaecut-lrseusfnocinteurs-futnolcinter>s-b-utrtialuetsytpeufnocintb-odycodebo-lckparmeaecut-lrsesparmeaecut-lrseparmeaecut-lrsespa)arm...steaiecut-lpeprl(-sre)a(rmpstiaeaerl-tmreetprarmpee,atarstmipel-raeartmenetioreu#tltparmeent-ramoelpc-aalarmeent-rameytpea-nnoatintdeagf-urtulmeancu-ltseparmenetioruvta#rparmeent-ramoelpc-aalarmeent-rameytpea-nnoatintdeagf-urtulmeancu-ltseparmbertetitauretsytpeparmeent-ramedreifioeiltpnc-aalarmeent-ramedreifieidtneagf-urtulmeancu-ltse=expeorsin

Rust

```
fn foo(x: i32) -> i32 {
    return x;

        x + 1
}
```

You separate arguments with a comma, both when you call the function, as well as when you declare it. Unlike ‘let’, you must declare the types of function arguments. Rust functions return exactly one value, and you declare the type after an ‘arrow’, which is a dash (-) followed by a greater-than sign (>). The last line of a function determines what it returns. You’ll note the lack of a semicolon here. In this case the ‘return’ keyword is used for early returns

Justification: Rust code uses snake case as the conventional style for function and variable names. In snake case, all letters are lowercase and underscores separate words. The curly braces tell the compiler where the function body begins and ends. Rust doesn’t care where you define your functions, only that they’re defined somewhere. Rust is an expression-based language, this is an important distinction to understand, other languages don’t have the same distinctions

Julia

In Julia, a function is an object that maps a tuple of argument values to a return value. Julia functions are not pure mathematical functions, in the sense that functions can alter and be affected by the global state of the program. The basic syntax for defining functions in Julia is:

```
julia> function f(x,y)
           x + y
       End
f (generic function with 1 method)
```

Justification = Julia function arguments follow a convention sometimes called "pass-by-sharing", which means that values are not copied when they are passed to functions. Function arguments themselves act as new variable *bindings* (new locations that can refer to values), but the values they refer to are identical to the passed values. Modifications to mutable values (such as Arrays) made within a function will be visible to the caller. This is the same behavior found in Scheme, most Lisps, Python, Ruby and Perl, among other dynamic languages.

X. Structures

Swift

Syntax
Structures are defined with a 'Struct' Keyword.

```
struct nameStruct {
  Definition 1
  Definition 2
   ---
  Definition N
}
```

Justification: Swift provides a flexible building block of making use of constructs as Structures. By making use of these structures once can define constructs methods and properties. Unlike C and Objective C, Structure need not require implementation files and interface. Structure allows us to create a single file and to extend its interface automatically to other blocks. In Structure the variable values are copied and passed in subsequent codes by returning a copy of the old values so that the values cannot be altered.

Rust

structs are a way of creating more complex data types. For example, if we were doing calculations involving coordinates in 2D space, we would need both an x and a y value:

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let origin = Point { x: 0, y: 0 }; // origin: Point

    println!("The origin is at ({}, {})", origin.x, origin.y);
}
```

Justification: We create an instance by stating the name of the struct, and then add curly braces containing key: value pairs where the keys are the names of the fields and the values are the data we want to store in those fields. We don’t have to specify the fields in the same order in which we declared them in the struct. In other words, the struct definition is like a general template for the type, and instances fill in that template with particular data to create values of the type.

Julia

Julia does not have the feature of using structures in the programs.

XI. Condition statements

Swift

- If statement
- If..else statement
- If..else.. if statement
- Nested statement
- Switch statement

If statement

Syntax :

```
if boolean_expression
{
  /* statement(s) will execute if the boolean expression is true */
}
```

Justification: If is keyword used in iteration. Condition is declared after If keyword. In swift braces '(' &')' are not used unlike in C. Body is written inside the curly braces so that the compiler understands that the statements are declared here after the brackets. If there are no curly braces then the compiler will not pop an error.

If..else statement

Syntax :

```
if boolean_expression {
  /* statement(s) will execute if the boolean expression is true */
} else {
  /* statement(s) will execute if the boolean expression is false */
}
```

Justification: Two keywords are used in this statement, If and else. If the condition is false in 'if' and then the 'else' statement starts executing. Body is written inside the curly braces so that the compiler understands that the statements are declared here after the brackets. If there are no curly braces then the compiler will not pop an error, but making it difficult for the compiler to understand the statements.

Grammar of an if

statementif-statementifif-conditioncode-blockelse-clauseif-conditionexpressiondeclarationelse-clauseelsecode-blockelseif-statement

If..else.. if statement

Syntax :

```
if boolean_expression_1 {
  /* Executes when the boolean expression 1 is true */
} else if boolean_expression_2 {
  /* Executes when the boolean expression 2 is true */
} else if boolean_expression_3 {
  /* Executes when the boolean expression 3 is true */
} else {
  /* Executes when the none of the above condition is true */
}
```

Justification: This statement is similar objective-C and C. Multiple If and else statements can be written in program. If a programmer has to use 'if' again after declaring first 'If' then it should be declared as 'else if'. The complier detects 'else-if' as a 'if' statement. This helps readability for the complier.

Nested statement

Syntax :

```
if boolean_expression_1 {
  /* Executes when the boolean expression 1 is true */
  if boolean_expression_2 {
    /* Executes when the boolean expression 2 is true */
  }
}
if boolean_expression_1 {
  /* Executes when the boolean expression 1 is true */
  if boolean_expression_2 {
    /* Executes when the boolean expression 2 is true */
  }
}
```

Justification: Nested statement is declared as multiple block of 'if' statements. Curly braces are used inside each block and every block has different Boolean expressions.s

Switch statement

Syntax :

```
switch(expression) {
  case constant-expression  :
    statement(s);
```

```
      break; /* optional */
   case constant-expression  :
      statement(s);
      break; /* optional */

   /* you can have any number of case statements */
   default : /* Optional */
      statement(s);
}
```

Justification: Switch statement consists of multiple possible *cases*, each of which begins with the case keyword. In addition to comparing against specific values, Swift provides several ways for each case to specify more complex matching patterns. Like the body of an if statement, each case is a separate branch of code execution. The switch statement determines which branch should be selected. Swift complier doesn't require break, break statement used to match and ignore a particular case or to break out of a matched case before that case has completed its execution.

Grammar of a switch statementswitch-statementswitchexpression{switch-cases}switch-casesswitch-caseswitch-casesswitch-casecasea-lbesltatementsdefauta-lbesltatementsswtcih-casecasea-lbed;lefauta-lbec;lasea-lbeclasecasete-imsic-tl:asete-imsi-tlpaternguard-calusepaternguard-calusec,asete-imsi-tldefauta-lbedlefaugt:luard-calusewhereguard-expressoinguard-expressoinexpressoin

Rust

If statement

In the case of if, there is one choice that leads down two paths:

```
let x = 5;

if x == 5 {
    println!("x is five!");
}
```

Justification: The block of code we want to execute if the condition is true is placed immediately after the condition inside curly braces. Blocks of code associated with the conditions in if expression.

If you want something to happen in the false case, use an else:

If – else statement

```
let x = 5;

if x == 5 {
    println!("x is five!");
} else {
    println!("x is not five :(");
}
```

If there is more than one case, use an else if:

Nested if statement

```
let x = 5;

if x == 5 {
    println!("x is five!");
} else if x == 6 {
    println!("x is six!");
} else {
    println!("x is not five or six :(");
}
```

If can also be used as an expression

```
let x = 5;

let y = if x == 5 { 10 } else { 15 }; // y: i32
```

Justification: When this program executes, it checks each if expression in turn and executes the first body for which the condition holds true. This works because if is an expression. The value of the expression is the value of the last expression in whichever branch was chosen. An if without an else always results in () as the value. Rust will only execute the block for the first true condition, and once it finds one, it won’t even check the rest.

Julia

Compound Expressions: begin and (;).

Conditional Evaluation: if-elseif-else and ?: (ternary operator).

Short-Circuit Evaluation: &&, || and chained comparisons.

Repeated Evaluation: Loops: while and for.

Exception Handling: try-catch, error() and throw().

Tasks (aka Coroutines): yieldto().

- LOOPS

Swift

- For-in
- For
- While

- Do..while

For-in

Syntax :

```
for index in var {
  statement(s)
}
```

Justification: Swift also provides a for-in loop that makes it easy to iterate over arrays, dictionaries, ranges, strings, and other sequences. The value of index is set to the first number in the range (1), and the statements inside the loop are executed. The loop contains only one statement, which prints an entry from the variable for the current value of index. If “in” is not used in for loop the complier will display error message. Range of iteration can be defined with (….).
For example,

```
{
for index in 1…5
}
```

Grammar of a for-in

statementfor-in-statementforpatterninexpressioncode-block

While

Syntax :

```
while condition {
  statement(s)
}
```

Justification: The while keyword used in the loop is a special keyword which calls the constructor pre-defined in the Swift library that supports it to repeat itself again and again until the condition is false. The while loop is used this way so that the compiler can differentiate between the constructor, the condition and the body of the loop, the condition is written without braces ‘(‘ and ‘)’. Body is written inside the curly braces so that the compiler understands that the statements are declared here after the brackets. If there are no curly braces then the compiler will not pop an error, but making it difficult for the compiler to understand the statements and the you might get an unexpected o/p as the compiler might not understand which statement to execute and which statement not to execute, this may reduce the readability of the compiler and then increase the time if many statements exist.
Grammar of a while
statementwhile-statementwhilewhile-conditioncode-blockwhile-conditionexpressiondeclaration

Do..while

Syntax :

```
do {
  statement(s);
}
while( condition );
```

Justification: The do and while keywords used in the loop are special keyword which calls the constructor pre- defined in the swift library that supports it to repeat itself again and again until the condition is false. The execution starts with statements to execute under the do statement which tells us what to do when the condition is true which is under the while statement. Here we pass only the condition in the while statement and the statements to be executed in an iterative has been declared in the do constructor.

Grammar of a do-while
statementdo-while-statementdocode-blockwhilewhile-condition

Rust

Rust currently provides three approaches to performing some kind of iterative activity. They are: loop, while and for. Each approach has its own set of uses.

<u>loop</u>: The infinite loop is the simplest form of loop available in Rust. Using the keyword loop, Rust provides a way to loop indefinitely until some terminating statement is reached. Rust's infinite loops look like this:

```
loop {
    println!("Loop forever!");
}
```

Justification: The loop keyword tells Rust to execute a block of code over and over again forever or until you explicitly tell it to stop. Most terminals support a keyboard shortcut, ctrl-C, to halt a program that is stuck in a continual loop.

<u>While</u>: while loops are the correct choice when you’re not sure how many times you need to loop.

```
let mut x = 5; // mut x: i32
let mut done = false; // mut done: bool

while !done {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 {
        done = true;
    }
}
```

Justification: When the compiler reads a while it recognises it as a combination of loop, if, else and break. This construct eliminates a lot of nesting that would be necessary if you used loop, if, else, and break, and it's clearer. While a condition holds true, the code runs; otherwise, it exits the loop.

For: The for loop is used to loop a particular number of times. Rust's for loops work a bit differently than in other systems languages.

```
for x in 0..10 {
    println!("{}", x); // x: i32
}
```

When you need to keep track of how many times you have already looped, you can use the .enumerate() function.

```
for (index, value) in (5..10).enumerate() {
    println!("index = {} and value = {}", index, value);
}
```

Outputs:

```
index = 0 and value = 5
index = 1 and value = 6
index = 2 and value = 7
index = 3 and value = 8
index = 4 and value = 9
```

Justification: The for keyword used in the for loop is a special keyword which calls the constructor pre-defined in the rust library that supports it to repeat itself again and again until the condition is false. The for loop is used this way so that the compiler can differentiate between the constructor and the body of the loop, the body is written inside the curly braces so that the compiler understands that the statements are declared here after the brackets.

Julia

There are two constructs for repeated evaluation of expressions: the while loop and the for loop. An example of a while loop:

```
julia> i = 1;julia> while i <= 5
           println(i)
           i += 1
       end1
```

ju

An example of a for loop:

```
julia> for i = 1:5
           println(i)
       end1
```

Justification = The for *increment*defines how the loop control variable changes each time the loop is repeated. The body of loop can either be empty or a single statement or a block of statements.

XII. Control statements

Swift

- Continue
- Break
- Fallthrough

Continue

Syntax : Continue

Justification: The continue statement tells a loop to stop what it is doing and start again at the beginning of the next iteration through the loop.

Grammar of a continue statementcontinue-statementcontinuelabel-name

Break

Syntax : Break

Justification: The break statement ends execution of an entire control flow statement immediately. The break statement can be used inside a switch or loop statement when you want to terminate the execution of the switch or loop statement earlier than would otherwise be the case. When used inside a loop statement, break ends the loop's execution immediately and transfers control to the code after the loop's closing brace (}). No further code from the current iteration of the loop is executed, and no further iterations of the loop are started. Grammar of a break statementbreak-statementbreaklabel-name

Fallthrough

Syntax :

```
 switch expression {
   case expression1  :
      statement(s)
      fallthrough /* optional */
   case expression2, expression3  :
      statement(s)
      fallthrough /* optional */

   default : /* Optional */
      statement(s);
}
```

Justification: In Swift, switch statements don’t fall through the bottom of each case and into the next one. That is, the entire switch statement completes its execution as soon as the first matching case is completed. C requires you to insert an explicit break statement at the end of every switch case to prevent fallthrough. Avoiding default fallthrough means that Swift switch statements are much more concise and predictable than their counterparts in C, and thus they avoid executing multiple switch cases by mistake.

Grammar of a continue
statementcontinue-statementcontinuelabel-name

Rust

Rust has two keywords to help us with modifying iteration: break and continue.

Break:

```
let mut x = 5;

loop {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 { break; }
}
```

We now loop forever with loop and use break to break out early. Issuing an explicit return statement will also serve to terminate the loop early.

Continue:

continue is similar, but instead of ending the loop, it goes to the next iteration. This will only print the odd numbers:

```
for x in 0..10 {
    if x % 2 == 0 { continue; }

    println!("{}", x);
}
```

Justification: We had to keep a dedicated mut boolean variable binding, done, to know when we should exit out of the loop. Rust has two keywords to help us with modifying iteration: break and continue.

Julia

Compound Expressions: begin and (;).

Conditional Evaluation: if-elseif-else and ?: (ternary operator).

Short-Circuit Evaluation: &&, || and chained comparisons.

Repeated Evaluation: Loops: while and for.

Exception Handling: try-catch, error() and throw().

Tasks (aka Coroutines): yieldto().

XIII. Enumeration

Swift

Syntax

```
enum enumname {
  // enumeration values are described here
}
```

Example

```
enum DaysofaWeek {
  case Sunday
  case Monday
   ---
  case Saturday
}
```

Justification: An enumeration is a user-defined data type which consists of set of related values. Keyword enum is used to defined enumerated data type.

Enumeration Functionality

Enumeration in swift also resembles the structure of C and Objective C.

- It is declared in a class and its values are accessed through the instance of that class.
- Initial member value is defined using enum intializers.
- Its functionality is also extended by ensuring standard protocol functionality.

  Grammar of an enumeration
  odniaetreclnumbrotdutin-ieaetutsracloen-yniltsu-mbrtutirasewtaeuvn-aytsel--umuoen-yniltsu-menumn-amegpec-naairemraucluese-oe{etrn-yniltsu-mm-embue}sroen-yniltsu-mm-embeusroen-yniltsu-mm-embeuroen-yniltsu-mm-embeusroen-yniltsu-mm-emboednriaetrucloen-yniltsu-macau-cesles-uoen-yniltsu-mbractau-cteisles-ctsaesuoentns-yniltsui-uamlesoen-n-yniltscu-t-sonmnieitsn-ay-lelscu--mca-esuoen-yniltsu-mucaoe-esn,tns-yniltsui-uamlesoen-n-yniltscu-mca-esenumca-nes-

ameuytptep-elenumerni-afmietndeueinmecar-nesi-afmariwteudevn-aytseli--unmenumn-amre{ergipfec-naiairatemawrutceulesey-vn-aydtspe:rl--ueietnm-m-emarbwe}sreuvn-aytsel--umm-embaerswreuvn-aytsel--umm-embarwereuvn-aytsel--umm-embarwesreuvn-aytsel--umm-emboedniraterwcleuvn-aytsel--umacau-cesleasr-weuvn-aytsel--umbractau-cteisles-ctsaeaswreutvns-aytsel--uiarwamrleseteus-vn-aytseli-c-ua-lwesm-euvn-aytsel-c-u-mca-easwreuvn-aytsel--umra,wca-eustvns-aytsel--uiarwamleseu-vn-aytsel-c-u-mca-esenumca-nes-ameawrugva-aenlis-mearnwtugva-aenlis-mta=ileterln

Rust

An enum in Rust is a type that represents data that is one of several possible variants. Each variant in the enum can optionally have data associated with it:

```
enum Message {
   Quit,
   ChangeColor(i32, i32, i32),
   Move { x: i32, y: i32 },
   Write(String),
}
```

We use the :: syntax to use the name of each variant: they’re scoped by the name of the enum itself. This allows both of these to work:

```
 let x: Message = Message::Move { x: 3, y: 4 };

}
```

Justification: Rust’s enums are most similar to algebraic data types in functional languages like F#, OCaml, and Haskell. Enums allow you to define a type by enumerating its possible values.

Julia

An enumeration is a complete, ordered listing of all the items in a collection,in julia enumeration is used as”@enum” .To create a object of a give type in julia .

Justification:= this creates a competley new data type which is converted to numbers by calling which provides easy accessiblity for the list.

4. INTRODUCTION TO THE DOMAIN TO WHICH PROGRAMMING LANGUAGE IS BEING DEVELOPED.

Domain: Systems Programming

A system programming language usually refers to a programming language used for system programming; such languages are designed for writing system software, which usually requires different development approaches when compared with application software. System software is computer software designed to operate and control the computer hardware, and to provide a platform for running application software. System software includes software categories such as operating systems, utility software, device drivers, compilers, and linkers.

System programming leads to the development of computer system software that manages and controls the computer operations. The low-level codes are very close to the hardware level and deal with things such as registers and memory allocations. The system programs or system software coordinates data transfer across the various components and deals with the compiling, linking, starting and stopping of programs, reading from files as well as writing to files. The system programming enhances or extends the functions of an operating system and may comprise components such as drivers, utilities and updates. They enable efficient management of hardware resources such as memory, file access, I/O operations, device management and process management such as process administration and multi-tasking.

5. REQUIREMENT SPECIFICATION

Identifiers:

Identifier must be unique. Identifier refers to name given to entities such as variables, functions, structures etc.
Rules for writing an identifier:

- A valid identifier can have letters (both uppercase and lowercase letters), digits and underscores.
- The first letter of an identifier should always be an alphabet.
- Alphabets, numbers and underscores can be used after the first letter
- The maximum length of an identifier is 20.
- Keywords cannot be used as an identifier

Example: num

A programmer uses an identifier to have a specific name to variables or functions so that it is easier to call them whenever required.

Data types:

The data types required are integer, floating point number, char, double and Boolean.

- The keyword for Integer - it
- The keyword for floating point – ft
- The keyword for double - db
- The keyword for character- ch
- The keyword for boolean- bl
- The keyword for String - st

Variable declaration : <data type> <variable name> ;

Example : it num;

Two new data types introduced in this language are currency and time

- The keyword for currency – cu
  The keyword for time – ti

  Basically currency is an array of size 2 and if a[1] is greater than 100 a[1] = a[1] – 100 and a[0] = a[0]+1
  Similarly time is an array of size 3 and here a[1] and a[2] checks if it is greater than 60 and then increments the previous location in the array.

Programmer must use the data type while declaring a variable so that the compiler knows what type of data can be assigned to the variable.

Compiler designer responsibility is to assign the type of data to assigned to the variable.

During Runtime with respect to the datatype, memory is allocated.

- it- 2 bytes
- ft – 4 bytes
- db – 8 bytes
- ch – 4 bytes
- bl – 1 byte

Arrays:

Collection of variables of same data type.

ex: it num\10\;

    ft a\50,50\;

Operators:

Arithmetic operators

At user and programming level the following symbols will be used for the respective functions or purposes.

- + : Addition and array/string concatenation.
- - : Subtraction.
- *: Multiplication
- / : Quotient.
- % : Remainder

Comparison operators

- == : Equal to.
- != : Not equal to.
- < : Less than
- > : Greater than
- <= : Less than or equal.
- >= : Greater than or equal

Boolean operators

- & : And
- | : Or

At runtime the following operations will be done .

Expressions:

Every expression consists of at least one operand and can have one or more operators.

Example : num+8 , num1 – num2, num1<num2

Programmer or a user uses an expression to do various arithmetic or logical operations

During runtime the operations will be done on the memory locations.

Assignment statements

An assignment statement gives a value to a variable. For example #num=10. Also the value of a variable can be changed using assignment statements or the value of an entire expression can be assigned to a variable. <variable> = <expression>

For example : num = num+1; num = num1 * num2;

During runtime data is copied or moved or assigned into a memory location when an assignment statement is executed.

Printing variability

The keyword for printing a statement or variables is disp.

For example: disp[“ Hello world”];

disp[“Value =”,num];

Reading input

The keyword for reading input of a variables is inp. Syntax: inp[<variable>];

For example: inp[num];

Iteration statements

Iteration is the process where a set of instructions or statements is executed repeatedly for a specified number of time or until a condition is satisfied.

Three types of iterations:

- While
- Do-while
- For

While loop:

```
 while condition expression {
        <body >
        }
```

Do-while loop:

```
Do {

        <body >

}while condition expression
```

For loop:

```
for <initialization expression>;<condition expression>;<update expression> {

        <body>

}
```

A programmer or a user uses an iteration statement when a set of expressions has be executed repeatedly;

Control statements:

- If
- If else
- If else if
- Switch
- Break
- Continue

If statement

```
if <condition expression> {

        <statements>

}
```

if is used only when a set of instructions are to be executed only if a condition is true.

If -else statement

```
if <condition expression> {

        <statements>

}

else {

        <statements>

}
```

if –else is used when a set of instructions are to be executed only if a condition is true else another set of instructions are to be executed.

If –else- if statement

```
if <condition expression> {
        <statements>
}
else if <condition expression> {
        <statements>
}
```

if –else-if is used when a set of instructions are to be executed only if a condition is true else another condition is checked if true or not and another set of instructions are to be executed.

Switch statement

```
Switch <condition expression > {
     case constant-expression1:     statements1;
     case constant-expression2:     statements2;
     case constant-expression3:     statements3;
     default : statements4;
   }
```

The switch statement is much like a nested if .. else statement. Its mostly a matter of preference which you use, switch statement can be slightly more efficient and easier to read.

Break and continue statement

break - exit form loop or switch.

continue - skip 1 iteration of loop.

Sub –program

Function calls and defining a function is called sub – program.

Function call:

```
<function name>[ <arg1>,<arg2>….<argn>]
```

A programmer uses this statement to call a function so a set of instructions will be executed .

Defining a function:

```
fun <return datatype> <Function name> <arg1>,<arg2>….<argn> {
        <body>
        ret <variable/expression>
```

}

Parameter passing:

Implementation of parameter passing in this language will be done using three models :- Pass-by-value, Pass-by-value–result and Pass-by-reference. In pass-by- reference the parameters with inout before datatypes will return the values at the end of the function execution.

6. RUNTIME ENVIRONMENT

Memory management:

Allocate space for a software stack and initialize the stack pointer

On 8-bit devices that have a hardware based return address stack, the software stack is mostly used for parameter passing to and from functions. On 16-bit or higher devices the software stack also stores the return address for each function call and interrupt.

Allocate space for a heap (if used)

A heap is a block of RAM that has been set aside as a sort of scratchpad for your application. It has the ability to dynamically create variables at runtime. This is done in the heap.

Copy values from Flash into variables declared with initial values

Variables declared with initial values must have those initial values loaded into memory before the program can use them. The initial values are stored in flash program memory (so they will be available after the device is power cycled) and are copied into each RAM location allocated to an initialized variable for its storage.

Clear uninitialized RAM

Any RAM (file register) not allocated to a specific purpose (variable storage, stack, heap, etc.) is cleared so that it will be in a known state.

Lifetime:

The lifetime of a variable is the time during which it is bound to a particular memory cell. This language will be using two types of binding, Stack-dynamic and Explicit heap-dynamic.

Stack-dynamic-

Storage bindings are created for variables when their declaration statements are elaborated.

• A declaration is elaborated when the executable code associated with it is executed.

• Local variables

Explicit heap-dynamic-

Allocated and de-allocated by explicit directives, specified by the programmer, which take effect during execution. Referenced only through pointers or references. It also provides for dynamic storage management.

Recursion:

Since it follows Stack-dynamic and Explicit heap-dynamic binding recursion is possible in this language. But tail recursion will not be followed or allowed in this language because improper implementation of tail recursion or without proper tail recurdion calls, one could easily run out of stack space.

Scoping:

The language will be implemented using "Dynamic Scoping". The binding of variables to declarations is done at run time. It is based on the calling sequence of subprograms. The implementation of scoping will be done using deep access method where non-local references are found by searching the activation record instances on the dynamic chain.

7. DESIGNING AN IMPERATIVE PROGRAMMING LANGUAGE

Designing Name:

Identifier must be unique. Identifier refers to name given to entities such as variables, functions, structures etc.
Rules for writing an identifier:

• A valid identifier can have letters (both uppercase and lowercase letters), digits and underscores.

• The first letter of an identifier should always be an alphabet.

• Alphabets, numbers and underscores can be used after the first letter

• The maximum length of an identifier is 20.

• Keywords cannot be used as an identifier

Example: num

<id> -> [a-z]+ [A-Z] [0-9]*[a-z]*[A-Z]*[_]*

Designing Operators:

Arithmetic operators

- + : Addition and array/string concatenation.
- - : Subtraction.
- *: Multiplication
- / : Quotient.
- % : Remainder

Comparison operators

- == : Equal to.
- != : Not equal to.
- < : Less than
- > : Greater than
- <= : Less than or equal.
- >= : Greater than or equal

Boolean operators

- & : And
- | : Or

Assignment operators

- = : assign value
- *= : multiply and assign
- /= : divide and assign
- += : add and assign
- -= : subtract and assign

<assop> -> = | *= | /= | += | -=
<compop> -> == / != / < / > / <= / >=
<boolop> -> & / |

Designing Statements:

Iteration statements:

Three types of iterations:

- While
- Do-while
- For

While loop:

```
 while condition expression {
           <body >


             }
```

Do-while loop:

```
Do                                                          {
<body                                                       >
}while condition expression
```

For loop:

```
for <initialization expression>;<condition expression>;<update expression> {
<body>
}
```

Control statements:

- If
- If else
- If else if
- Break
- Continue

If statement

```
if <condition expression> {
<statements>}
```

If -else statement

```
if <condition expression> {
<statements>
}
else {
<statements>
}
```

If –else- if statement

```
if <condition expression> {
<statements>
}else if <condition expression> {
<statements>
}
```

Break and continue statement:

break - exit form loop or switch.

continue - skip 1 iteration of loop.

```
<statement> -> <expstat>;<statement>/<boolstat>;<statement>  / <iterstat>;<statement> /
<condstat>;<statement> / <outstat>;<statement> / <inpstat>;<statement>
/<jumpstat>;<statement>/ e
<jumpstat> -> goto <id> / continue / break
<expstat> ->  <id><assop><E> / <id> = <id><more> / <condass> /e
<condass> -> <boolstat>?<E>:<E><assop><E>
```

<boolstat> -> <boolstat> <boolop> <rel> / <rel>
<rel> -> <E> <compop> <E> / <rE>
<iterstat> -> for <statement> { <statement> } /
for <statement> <statement> /
while <statement> { <statement> } /
while <statement> <statement> /
do {<statement>}while<statement> /
do <statement>while<statement> / e
<condstat> -> if<statement>{<statement>}else{ <statement>}/
if<statement>{<statement>} /
if<statement><statement>else{ <statement>}/
if<statement><statement>/
if<statement>{<statement>}else <statement>/
if<statement><statement>else<statement>
<outstat> -> disp[<print>]
<inpstat> -> inp[<varlist>]
<E> -> <E> + <mE> / <E> – <mE> /<E><unop> / <unop><E>/ +<E><unop> / -<E><unop> /<mE>
<mE> -> <mE> * <pE> | <mE> / <pE> | <mE> % <pE> | <pE>
<pE> -> <rE> ^ <pE>/<rE>
<rE> -> (<E>) / <id> / <integerliteral> / <floatliteral> / <doubleliteral> / <charliteral> / <stringliteral> / <booleanval>

Designing Data types

The data types are integer, floating point number, char, double and Boolean.

- The keyword for Integer - it
  The keyword for floating point – ft
  The keyword for double – db
  The keyword for character- ch
  The keyword for boolean- bl
  The keyword for String – St

Two new data types introduced in this language are currency and time

- The keyword for currency – cu
  The keyword for time – ti

Basically currency is an array of size 2 and if a[1] is greater than 100 a[1] = a[1] – 100 and a[0] = a[0]+1
Similarly time is an array of size 3 and here a[1] and a[2] checks if it is greater than 60 and then increments the previous location in the array.

<type> -> it/ft/db/ch/St/bl/cu/ti/void

Designing of Sub program or procedure or function

Function call:

```
<function name>[ <arg1>,<arg2>….<argn>]
```

Defining a function:

```
fun <return datatype> <Function name> <arg1>,<arg2>….<argn> {
<body>
ret <variable/expression>
}
```

Syntax:

Grammar

1) <program> -> ~BCG~<functions>~BCG~
2) <functions> -> <function> <functions>/ e
3) <function> -> <funsig> <funbody>
4) <funsig> -> fun <type> <id> <params>
5) <type> -> it/ft/db/ch/St/bl/void
6) <params> -> <type> <id> <comma> <params>/ e
7) <funbody> -> { <decl> <statement> <return> }
8) <decl> -> <type> <varlist>;<decl> / e
9) <varlist> -> <id> <comma> <varlist>; / <id>\dimen\ <comma> <varlist>; e
10) <dimen> -> <E > <comma> <dimen> / e
11) <statement> -> <expstat>;<statement>/<boolstat>;<statement> / <iterstat>;<statement> / <condstat>;<statement> / <outstat>;<statement> / <inpstat>;<statement> /<jumpstat>;<statement>/ e
12) <jumpstat> -> goto <id> / continue / break
13) <expstat> -> <id><assop><E> / <id> = <id><more> / <condass> /<E> /<boolstat> / e
14) <condass> -> <boolstat>?<E>:<E><assop><E>
15) <boolstat> -> <boolstat> <boolop> <rel> / <rel>
16) <rel> -> <E> <compop> <E> / <rE>
17)
```
<iterstat> -> for <statement> { <statement> } /
              for <statement>  <statement>  /
              while <statement> { <statement> } /
              while <statement>  <statement> /
              do {<statement>}while<statement> /
              do <statement>while<statement> / e
```
18)
```
<condstat> -> if<statement>{<statement>}else{ <statement>}/
               if<statement>{<statement>} /
               if<statement><statement>else{ <statement>}/
               if<statement><statement>/
               if<statement>{<statement>}else <statement>/
               if<statement><statement>else<statement>
```
19) <outstat> -> disp[<print>]
20) <print> -> "<stringliteral>" <comma> <print> / <id> <comma> <print> /<id>\dimen\ <comma> <print> / e
21) <inpstat> -> inp[<varlist>]
22) <return> -> ret <id>; / ret <E>; / e
23) <more> -> [<args>]

24) <args> -> <id> <comma> <args> / e
25) <E> -> <E> + <mE> / <E> – <mE> /<E><unop> / <unop><E>/ +<E><unop> / -<E><unop> /<mE>
26) <mE> -> <mE> * <pE> | <mE> / <pE> | <mE> % <pE> | <pE>
27) <pE> -> <rE> ^ <pE>/<rE>
28) <rE> -> (<E>) / <id> / <integerliteral> / <floatliteral> / <doubleliteral> / <charliteral> / <stringliteral> / <booleanval>
29) <assop> -> = | *= | /= | += | -=
30) <unop> -> ++ / --
31) <compop> -> == / != / < / > / <= / >=
32) <boolop> -> & / |
33) <id> -> [a-z]+ [A-Z] [0-9]*[a-z]*[A-Z]*[_]*
34) <comma> -> , / e

Removing left recursion

1) <E> -> <E> + <mE> / <E> – <mE> /<E><unop> / <unop><E>/ +<E><unop> / -<E><unop> /<mE>

After removing left recursion

<E> -> <unop><E><t1> / +<E><unop> <t1> / -<E><unop><t1> /<mE><t1>
<t1> -> + <mE><t1>/ – <mE><t1>/ <unop><t1>/e

2) <mE> -> <mE> * <pE> | <mE> / <pE> | <mE> % <pE> | <pE>

After removing left recursion

<mE> -> <pE><t2>
<t2> -> * <pE><t2> | / <pE><t2> | % <pE><t2> | e

Removing left factoring

1) <return> -> ret <id>; / ret <E>; / e
   After removing factoring

<return> -> ret <t3> / e
<t3> -> <id>; / <E>;

2) <expstat> -> <id><assop><E> / <id> = <id><more> / <condass> /e

After removing factoring
<expstat> -> <id><t4> / <condass> / e
<t4> -> <assop><E> / =<id><more>

3) <iterstat> -> for <statement> { <statement> } /
   for <statement> <statement> /
   while <statement> { <statement> } /
   while <statement> <statement> /
   do {<statement>}while<statement> /
   do <statement>while<statement> / e

After removing factoring

<iterstat> -> for <statement><t5> / while <statement><t5> / do<t6> / e
<t5> -> { <statement> } / <statement>
<t6> -> {<statement>}while<statement> / <statement>while<statement>

```
4)  <condstat> -> if<statement>{<statement>}else{ <statement>}/
                  if<statement>{<statement>} /
                  if<statement><statement>else{ <statement>}/
                  if<statement><statement>/
                  if<statement>{<statement>}else <statement>/
                  if<statement><statement>else<statement>
```

After removing factoring

<condstat> -> if<statement><t7>
<t7> -> {<statement>}else{ <statement>} / {<statement>} / <statement>else{ <statement>} / <statement> / {<statement>}else <statement> / <statement>else<statement>

8. CASE STUDIES

1)~BCG~

```
fun it main{
it a;
disp[“Enter value of a”];
inp[a];
disp[“value of a =”,a];
ret 0;
}
~BCG~
```

```
<program> -> ~BCG~<functions>~BCG~
        -> ~BCG~<function> <functions>~BCG~
        -> ~BCG~<funsig> <funbody><functions>~BCG~
        -> ~BCG~ fun <type> <id> <params> <funbody><functions>~BCG~
        -> ~BCG~ fun it <id> <params> <funbody><functions>~BCG~
        -> ~BCG~ fun it main <params> <funbody><functions>~BCG~
        -> ~BCG~ fun it main  <funbody><functions>~BCG~
        -> ~BCG~ fun it main  { <decl> <statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { <type> <varlist>;<decl>  <statement> <return>
}<functions>~BCG~
        -> ~BCG~ fun it main  { it<varlist>;<decl>  <statement> <return>
}<functions>~BCG~
        -> ~BCG~ fun it main  { it<id> <comma> <varlist>;<decl>  <statement> <return>
        }<functions>~BCG~
```

-> ~BCG~ fun it main { it a<comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a; <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a; <outstat>;<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a; disp[<print>];<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a; disp["<stringliteral>" <comma> <print>];<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a; disp[“Enter value of a”];<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a; disp[“Enter value of a”]; <inpstat>;<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a; disp[“Enter value of a”];inp [<varlist>] ;<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a; disp[“Enter value of a”];inp [<id> <comma> <varlist>] ;<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a <comma> <varlist>] ;<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a <varlist>] ;<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a];<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; <outstat>;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; disp["<stringliteral>" <comma> <print>];<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; disp["“value of a =”, <print>];<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; disp["“value of a =”,<id> <comma> <print>];<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; disp["“value of a =”,a <comma> <print>];<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a];disp["“value of a =”,a];<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; disp["“value of a =”,a];<return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a];disp["“value of a =”,a]; ret <id>; }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; disp["“value of a =”,a]; ret a ; }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; disp["“value of a =”,a]; ret a ; ~BCG~

2) ~BCG~
fun it main{

```
it a;
disp[“Enter value of a”];
inp[a];
if a=0
a=1;
disp[“value of a =”,a];
ret 0;
}
~BCG~
```

```
<program> -> ~BCG~<functions>~BCG~
          -> ~BCG~<function> <functions>~BCG~
          -> ~BCG~<funsig> <funbody><functions>~BCG~
          -> ~BCG~ fun <type> <id> <params> <funbody><functions>~BCG~
          -> ~BCG~ fun it <id> <params> <funbody><functions>~BCG~
          -> ~BCG~ fun it main <params> <funbody><functions>~BCG~
          -> ~BCG~ fun it main  <funbody><functions>~BCG~
          -> ~BCG~ fun it main  { <decl> <statement> <return> }<functions>~BCG~
          -> ~BCG~ fun it main  { <type> <varlist>;<decl>  <statement> <return>
}<functions>~BCG~
          -> ~BCG~ fun it main  { it<varlist>;<decl>  <statement> <return>
}<functions>~BCG~
          -> ~BCG~ fun it main  { it<id> <comma> <varlist>;<decl>  <statement> <return>
          }<functions>~BCG~
          -> ~BCG~ fun it main  { it a<comma> <varlist>;<decl>  <statement> <return>
          }<functions>~BCG~
          -> ~BCG~ fun it main  { it a <varlist>;<decl>  <statement> <return>
          }<functions>~BCG~
          -> ~BCG~ fun it main  { it a;<decl>  <statement> <return> }<functions>~BCG~
          -> ~BCG~ fun it main  { it a; <statement> <return> }<functions>~BCG~
          -> ~BCG~ fun it main  { it a; <outstat>;<statement>  <return> }<functions>~BCG~
          -> ~BCG~ fun it main  { it a; disp[<print>];<statement>  <return>
          }<functions>~BCG~
          -> ~BCG~ fun it main  { it a; disp["<stringliteral>" <comma> <print>];<statement>
          <return> }<functions>~BCG~
          -> ~BCG~ fun it main  { it a; disp[“Enter value of a”];<statement>  <return>
          }<functions>~BCG~
          -> ~BCG~ fun it main  { it a; disp[“Enter value of a”]; <inpstat>;<statement>
          <return> }<functions>~BCG~
          -> ~BCG~ fun it main  { it a; disp[“Enter value of a”];inp [<varlist>]
          ;<statement> <return> }<functions>~BCG~
          -> ~BCG~ fun it main  { it a; disp[“Enter value of a”];inp [<id> <comma> <varlist>]
          ;<statement> <return> }<functions>~BCG~
          -> ~BCG~ fun it main  { it a; disp[“Enter value of a”];inp [a <comma> <varlist>]
          ;<statement> <return> }<functions>~BCG~
          -> ~BCG~ fun it main  { it a; disp[“Enter value of a”];inp [a <varlist>]
          ;<statement> <return> }<functions>~BCG~
          -> ~BCG~ fun it main  { it a; disp[“Enter value of a”];inp [a]
```

;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; <condstat>;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if<statement><statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if <expstat>;<statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if <id><assop><E><statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if a = <E><statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if a = <mE ><statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if a = <pE><statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if a = <rE><statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if a = <integerliteral><statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if a = 0 <statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if a = 0 <expstat>;<statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if a = 0; <id><assop><E><statement> <statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if a = 0; a=1 ;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if a = 0; a=1 ; <outstat>;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if a = 0; a=1 ; disp["<stringliteral>" <comma> <print>];<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if a = 0; a=1 ; disp["“value of a =”, <print>];<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if a = 0; a=1 ; disp["“value of a =”,<id> <comma> <print>];<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if a = 0; a=1 ; disp["“value of a =”,a <comma> <print>];<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if a = 0; a=1 ; disp["“value of a =”,a];<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if a = 0; a=1 ; disp["“value of a =”,a];<return> }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if a = 0; a=1 ; disp["“value of a =”,a]; ret <id>; }<functions>~BCG~
->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if a = 0; a=1 ; disp["“value of a =”,a]; ret a ; }<functions>~BCG~

->~BCG~ fun it main { it a; disp[“Enter value of a”];inp [a]; if a = 0; a=1 ; disp["“value of a =”,a]; ret a ; ~BCG~

3) ~BCG~

```
fun it main
{
it a,b,c;
c=add[a,b];
ret 0;
}
fun it add it a, it b {
it c;
c=a+b;
ret c;
}
~BCG~
```

```
-> ~BCG~<functions>~BCG~
        -> ~BCG~<function> <functions>~BCG~
        -> ~BCG~<funsig> <funbody><functions>~BCG~
        -> ~BCG~ fun <type> <id> <params> <funbody><functions>~BCG~
        -> ~BCG~ fun it <id> <params> <funbody><functions>~BCG~
        -> ~BCG~ fun it main <params> <funbody><functions>~BCG~
        -> ~BCG~ fun it main  <funbody><functions>~BCG~
        -> ~BCG~ fun it main  { <decl> <statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { <type> <varlist>;<decl>  <statement> <return>
}<functions>~BCG~
        -> ~BCG~ fun it main  { it<varlist>;<decl>  <statement> <return>
}<functions>~BCG~
        -> ~BCG~ fun it main  { it<id> <comma> <varlist>;<decl>  <statement> <return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a<comma> <varlist>;<decl>  <statement> <return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a ,<varlist>;<decl>  <statement> <return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, <id> <comma> <varlist>;<decl>  <statement> <return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c <comma> <varlist>;<decl>  <statement> <return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;<decl>  <statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;  <expstat>;<statement><return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   <id> = <id><more>;<statement><return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add <more>;<statement><return>
        }<functions>~BCG~
```

-> ~BCG~ fun it main { it a, b, c; c=add [<args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=add [<id> <comma> <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=add [a, <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=add [a, <id> <comma> <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=add [a, b <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=add [a, b];<return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=add [a, b]; ret <id>; }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; }<function>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun <type> <id> <params> <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it <id> <params> <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it add <params> <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it add < type> <id> <comma> <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it add it a, <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it add it a, < type> <id> <comma> <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it add it a, it b , <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it add it a, it b <{ <decl> <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it add it a, it b <{ <type> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it add it a, it b <{ it<varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it add it a, it b <{ it<id> <comma> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it add it a, it b <{ it c <comma> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it add it a, it b <{ it c;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it add it a, it b <{ it c; <id><assop><E>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it add it a, it b <{ it c; c = <E>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it add it a, it b <{ it c; c = <E> + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it add it a, it b <{ it c; c = <mE> + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it add it a, it b <{ it c; c = <pE> + <mE>;<statement> <return> } >~BCG~

->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it add it a, it b <{ it c; c = <rE> + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it add it a, it b <{ it c; c = <id > + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it add it a, it b <{ it c; c = a+ <pE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it add it a, it b <{ it c; c = a+ <rE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it add it a, it b <{ it c; c = a+ <id >;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it add it a, it b <{ it c; c = a+b ;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it add it a, it b <{ it c; c = a+b ;<return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it add it a, it b <{ it c; c = a+b ;ret <id>; } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it add it a, it b <{ it c; c = a+b ;ret c; } >~BCG~

```
4) ~BCG~
fun it main
{
it a,b,c;
c=func[a,b];
ret 0;
}
fun it func it a, it b {
it c;
c=a+b;
a=c*b;
b++;
c=a+b;
ret c;
}
```

-> ~BCG~<functions>~BCG~
-> ~BCG~<function> <functions>~BCG~
-> ~BCG~<funsig> <funbody><functions>~BCG~
-> ~BCG~ fun <type> <id> <params> <funbody><functions>~BCG~
-> ~BCG~ fun it <id> <params> <funbody><functions>~BCG~
-> ~BCG~ fun it main <params> <funbody><functions>~BCG~
-> ~BCG~ fun it main <funbody><functions>~BCG~
-> ~BCG~ fun it main { <decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { <type> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it<varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it<id> <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~

-> ~BCG~ fun it main { it a<comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a ,<varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, <id> <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; <expstat>;<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   <id> = <id><more>;<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=add <more>;<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=add [<args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=add [<id> <comma> <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=add [a, <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=add [a, <id> <comma> <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=add [a, b <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=add [a, b];<return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret <id>; }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; }<function>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun <type> <id> <params> <funbody >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it <id> <params> <funbody >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func <params> <funbody >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func < type> <id> <comma> <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, < type> <id> <comma> <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b , <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ <decl> <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ <type> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it<varlist>;<decl > <statement> <return> } >~BCG~

->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it<id> <comma> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c <comma> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; <id><assop><E>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <E>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <E> + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <mE> + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <pE> + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <rE> + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <id > + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+ <pE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+ <rE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+ <id >;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; <id><assop><E>; <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <E>; <statement><return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <mE>*<pE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <pE>*<rE>; <statement><return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <rE>*<id>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <id>*<b>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<expstat >;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<E >;<statement> <return> } >~BCG~

->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<E><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<mE><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<pE><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<rE><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<id><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;<expstat>; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;<id><assop><E >; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;c = <E >; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;c = <E >+<mE>; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;c = <mE >+<pE>; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;c = <pE >+<rE>; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;c = <rE >+<id>; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;c = <id >+<id>; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;c = a+b; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;c = a+b; ret <id>;} >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;c = a+b; ret <c >;} >~BCG~

5)~BCG~
```
fun it main
{
it a\10\;
for i=0;i<10;i++
inp[a\i\];
ret 0;
}
~BCG~
```

-> ~BCG~<functions>~BCG~
-> ~BCG~<function> <functions>~BCG~

-> ~BCG~<funsig> <funbody><functions>~BCG~
-> ~BCG~ fun <type> <id> <params> <funbody><functions>~BCG~
-> ~BCG~ fun it <id> <params> <funbody><functions>~BCG~
-> ~BCG~ fun it main <params> <funbody><functions>~BCG~
-> ~BCG~ fun it main <funbody><functions>~BCG~
-> ~BCG~ fun it main { <decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { <type> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it<varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it<id>\<dimen>\ <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<dimen>\ <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\< integerliteral> <comma> <dimen >\ <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\< 10> <comma> <dimen >\ <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < iterstat>;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for <statement> <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for <expstat> <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for < id><assop><E > <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = <E > <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = <mE > <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = <pE > <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = <rE > <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = <integerliteral > <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0; <expstat >;<statement><statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; <E >;<statement ><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ;<E><unop>;<statement > <statement><return> }<functions>~BCG~

```
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ;<mE><unop>;<statement
><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ;<pE><unop>;<statement >
<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ;<rE><unop>;<statement
><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ;<id ><unop>;<statement >
<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i<unop>;<statement >
<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ;<statement ><statement>
<return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ;<expstat
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ;<boolstat
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ;<rel >;<statement><statement>
<return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; <E> <compop> <E>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; <mE> <compop> <E>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; <rE> <compop> <E>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; <id> < <E>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < <mE>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < <pE>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < <rE>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < <integerliteral
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10;<statement><statement>
<return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10;<inpstat
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[<varlist>]
;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[<id>\dimen\
<comma> <varlist>] ;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\<E> \ <comma>
<varlist>] ;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\<mE> \ <comma>
<varlist>] ;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\<rE> \ <comma>
<varlist>] ;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\<id> \ <comma>
<varlist>] ;<statement><statement> <return> }<functions>~BCG~
```

```
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \ <comma>
<varlist>] ;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \]
;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] ;<return>
}<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] ;ret <id>
}<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] ;ret 0
}<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] ;ret 0 }~BCG~
```

6) ~BCG~

```
fun it main{
it a,b;
disp["Enter value of a"];
inp[a];
a=1
while a<10
{
b=a;
a++;
}
disp["value of b =",b];
ret 0;
}
~BCG~
```

```
<program> -> ~BCG~<functions>~BCG~
    -> ~BCG~<function> <functions>~BCG~
    -> ~BCG~<funsig> <funbody><functions>~BCG~
    -> ~BCG~ fun <type> <id> <params> <funbody><functions>~BCG~
    -> ~BCG~ fun it <id> <params> <funbody><functions>~BCG~
    -> ~BCG~ fun it main <params> <funbody><functions>~BCG~
    -> ~BCG~ fun it main  <funbody><functions>~BCG~
    -> ~BCG~ fun it main  { <decl> <statement> <return> }<functions>~BCG~
    -> ~BCG~ fun it main  { <type> <varlist>;<decl>  <statement> <return>
}<functions>~BCG~
    -> ~BCG~ fun it main  { it<varlist>;<decl>  <statement> <return>
}<functions>~BCG~
    -> ~BCG~ fun it main  { it<id> <comma> <varlist>;<decl>  <statement> <return>
    }<functions>~BCG~
    -> ~BCG~ fun it main  { it a<comma> <varlist>;<decl>  <statement> <return>
    }<functions>~BCG~
    -> ~BCG~ fun it main  { it a, <varlist>;<decl>  <statement> <return>
    }<functions>~BCG~
    -> ~BCG~ fun it main  { it a, < id> <comma> <varlist >;<decl>  <statement>
    <return> }<functions>~BCG~
```

~BCG~ fun it main { it a, b<comma> <varlist >;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a,b;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a,b; <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a,b; <outstat>;<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a,b; disp[<print>];<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a,b; disp["<stringliteral>" <comma> <print>];<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a,b; disp[“Enter value of a”];<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a,b; disp[“Enter value of a”]; <inpstat>;<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [<varlist>] ;<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [<id> <comma> <varlist>]
;<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a <comma> <varlist>] ;<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a <varlist>] ;<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a] ;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; <iterstat>;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while <statement> { <statement> }/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while<expstat>;<statement> {<statement>}<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while <boolstat><statement> {<statement>};<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while <rel><statement>{<statement>};<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while < E> <compop> <E ><statement> {<statement>}/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while < mE> <compop> <E ><statement> {<statement>];<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while < rE> <compop> <E ><statement> {<statement>};<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <E ><statement> {<statement>};<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <rE ><statement> {<statement>};<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <integerliteral ><statement> {<statement>};<statement> <return> }<functions>~BCG~

->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10
><statement> {<statement>};<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{<statement>};<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{<expstat><statement>};<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{<E><assop><E ><statement>};<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{<mE>==<mE ><statement>};<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{<pE>==<pE ><statement>};<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{<id>==<id ><statement>};<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{b==a; <statement>};<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{b==a; <expstat><statemet>};<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{b==a; <E ><statemet>};<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{b==a; <E><unop<statemet>};<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{b==a; <id ><unop<statemet>};<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{b==a; a <unop<statemet>};<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{b==a; a++; <statemet>};<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{b==a; a++; };<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{b==a; a++;} <outstat>;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{b==a; a++;} disp["<stringliteral>" <comma> <print>];<statement> <return>
}<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{b==a; a++;} disp[“value of b =”, <print>];<statement> <return>
}<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{b==a; a++;} disp["“value of b =”,<id> <comma> <print>];<statement> <return>
}<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{b==a; a++;} disp["“value of b =”,b];<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{b==a; a++;} disp[value of a =”,a];<return> }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{b==a; a++;} disp[value of a =”,a]; ret <id>; }<functions>~BCG~
->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{b==a; a++;} disp[value of a =”,a]; ret 0; }<functions>~BCG~

->~BCG~ fun it main { it a,b; disp[“Enter value of a”];inp [a]; while a < <10 >
{b==a; a++;} disp[value of a =”,a]; ret 0;}~BCG~

```
7) ~BCG~
fun it main
{
it a,b,c;
c=sub [a,b];
ret 0;
}
fun it sub it a, it b {
it c;
c=a-b;
ret c;
}
~BCG~
```

-> ~BCG~<functions>~BCG~
-> ~BCG~<function> <functions>~BCG~
-> ~BCG~<funsig> <funbody><functions>~BCG~
-> ~BCG~ fun <type> <id> <params> <funbody><functions>~BCG~
-> ~BCG~ fun it <id> <params> <funbody><functions>~BCG~
-> ~BCG~ fun it main <params> <funbody><functions>~BCG~
-> ~BCG~ fun it main <funbody><functions>~BCG~
-> ~BCG~ fun it main { <decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { <type> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it<varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it<id> <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a<comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a ,<varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, <id> <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; <expstat>;<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; <id> = <id><more>;<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=sub <more>;<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=sub [<args>];<statement><return> }<functions>~BCG~

-> ~BCG~ fun it main { it a, b, c; c=sub [<id> <comma> <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c= sub [a, <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c= sub [a, <id> <comma> <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c= sub [a, b <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c= sub [a, b];<return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c= sub [a, b]; ret <id>; }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; }<function>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; } fun <type> <id> <params> <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; } fun it <id> <params> <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; } fun it sub <params> <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; } fun it sub < type> <id> <comma> <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; } fun it sub it a, <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; } fun it sub it a, < type> <id> <comma> <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; } fun it sub it a, it b , <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; } fun it sub it a, it b <{ <decl> <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; } fun it sub it a, it b <{ <type> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; } fun it sub it a, it b <{ it<varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; } fun it sub it a, it b <{ it<id> <comma> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; } fun it sub it a, it b <{ it c <comma> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; } fun it sub it a, it b <{ it c;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; } fun it sub it a, it b <{ it c; <id><assop><E>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; } fun it sub it a, it b <{ it c; c = <E>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; } fun it sub it a, it b <{ it c; c = <E> - <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; } fun it sub it a, it b <{ it c; c = <mE> -<mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; } fun it sub it a, it b <{ it c; c = <pE> - <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=sub [a, b]; ret 0; } fun it sub it a, it b <{ it c; c = <rE> - <mE>;<statement> <return> } >~BCG~

```
        ->~BCG~ fun it main  { it a, b, c;   c=sub [a, b ]; ret 0; } fun it sub it a, it b <{ it c; c =
        <id > - <mE>;<statement> <return> } >~BCG~
        ->~BCG~ fun it main  { it a, b, c;   c=sub [a, b ]; ret 0; } fun it sub it a, it b <{ it c; c =
        a- <pE>;<statement> <return> } >~BCG~
        ->~BCG~ fun it main  { it a, b, c;   c=sub [a, b ]; ret 0; } fun it sub it a, it b <{ it c; c =
        a- <rE>;<statement> <return> } >~BCG~
        ->~BCG~ fun it main  { it a, b, c;   c=sub [a, b ]; ret 0; } fun it sub it a, it b <{ it c; c =
        a- <id >;<statement> <return> } >~BCG~
        ->~BCG~ fun it main  { it a, b, c;   c=sub [a, b ]; ret 0; } fun it sub it a, it b <{ it c; c =
        a-b ;<statement> <return> } >~BCG~
        ->~BCG~ fun it main  { it a, b, c;   c=sub [a, b ]; ret 0; } fun it sub it a, it b <{ it c; c =
        a-b ;<return> } >~BCG~
        ->~BCG~ fun it main  { it a, b, c;   c=sub [a, b ]; ret 0; } fun it sub it a, it b <{ it c; c =
        a-b ;ret <id>; } >~BCG~
        ->~BCG~ fun it main  { it a, b, c;   c=sub [a, b ]; ret 0; } fun it sub it a, it b <{ it c; c =
        a-b ;ret c; } >~BCG~
```

8) ~BCG~

```
fun it main
{
it a,b,c;
c=func[a,b];
ret 0;
}
fun it func it a, it b {
it c;
c=c+b;
a=a*b;
c--;
c=a/b;
ret c;
}
```

```
-> ~BCG~<functions>~BCG~
        -> ~BCG~<function> <functions>~BCG~
        -> ~BCG~<funsig> <funbody><functions>~BCG~
        -> ~BCG~ fun <type> <id> <params> <funbody><functions>~BCG~
        -> ~BCG~ fun it <id> <params> <funbody><functions>~BCG~
        -> ~BCG~ fun it main <params> <funbody><functions>~BCG~
        -> ~BCG~ fun it main  <funbody><functions>~BCG~
        -> ~BCG~ fun it main  { <decl> <statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { <type> <varlist>;<decl>  <statement> <return>
}<functions>~BCG~
        -> ~BCG~ fun it main  { it<varlist>;<decl>  <statement> <return>
}<functions>~BCG~
        -> ~BCG~ fun it main  { it<id> <comma> <varlist>;<decl>  <statement> <return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a<comma> <varlist>;<decl>  <statement> <return>
        }<functions>~BCG~
```

```
-> ~BCG~ fun it main  { it a ,<varlist>;<decl>  <statement> <return>
}<functions>~BCG~
-> ~BCG~ fun it main  { it a, <id> <comma> <varlist>;<decl>  <statement> <return>
}<functions>~BCG~
-> ~BCG~ fun it main  { it a, b, c <comma> <varlist>;<decl>  <statement> <return>
}<functions>~BCG~
-> ~BCG~ fun it main  { it a, b, c;<decl>  <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a, b, c;  <expstat>;<statement><return>
}<functions>~BCG~
-> ~BCG~ fun it main  { it a, b, c;   <id> = <id><more>;<statement><return>
}<functions>~BCG~
-> ~BCG~ fun it main  { it a, b, c;   c=add <more>;<statement><return>
}<functions>~BCG~
-> ~BCG~ fun it main  { it a, b, c;   c=add [<args>];<statement><return>
}<functions>~BCG~
-> ~BCG~ fun it main  { it a, b, c;   c=add [<id> <comma>
<args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a, b, c;   c=add [a, <args>];<statement><return>
}<functions>~BCG~
-> ~BCG~ fun it main  { it a, b, c;   c=add [a, <id> <comma>
<args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a, b, c;   c=add [a, b <args>];<statement><return>
}<functions>~BCG~
-> ~BCG~ fun it main  { it a, b, c;   c=add [a, b ];<return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret <id>; }<functions>~BCG~
-> ~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; }<functions>~BCG~
-> ~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; }<function>~BCG~
-> ~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun <type> <id> <params>
<funbody >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it <id> <params>
<funbody >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func <params>
<funbody >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func < type> <id>
<comma> <params > <funbody >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, <params >
<funbody >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, < type> <id>
<comma> <params > <funbody >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b , <params
> <funbody >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ <decl>
<statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{
<type> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{
it<varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it<id>
<comma> <varlist>;<decl > <statement> <return> } >~BCG~
```

->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c <comma> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; <id><assop><E>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <E>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <E> + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <mE> + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <pE> + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <rE> + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <id > + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+ <pE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+ <rE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+ <id >;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; <id><assop><E>; <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <E>; <statement><return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <mE>*<pE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <pE>*<rE>; <statement><return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <rE>*<id>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <id>*<b>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <a >*<b>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <a>*<b>;<expstat >;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <a >*<b>;<E >;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = < a >*<b>;<E><unop>;<statement> <return> } >~BCG~

->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <a >*<b>;<mE><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <a >*<b>;<pE><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <a >*<b>;<rE><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <a >*<b>;<id><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <a >*<b>;<c><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <a >*<b>;<b>--;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <a >*<b>;<b> --;<expstat>; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <a >*<b>;<b> --;<id><assop><E >; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <a >*<b>;<b> --;c = <E >; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <a >*<b>;<b> --;c = <E >/<mE>; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <a >*<b>;<b> --;c = <mE >/<pE>; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <a >*<b>;<b> --;c = <pE >/<rE>; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c c=c+b ; a = <a >*<b>;<b> --;c = <rE >/<id>; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <a >*<b>;<b> --;c = <id >/<id>; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <a >*<b>;<b> --;c = a/b; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <a >*<b>;<b> --; c = a/b; ret <id>;} >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = c+b ; a = <a >*<b>;<c> --;c = a/b; ret c ;} >~BCG~

```
9)~BCG~
fun it main
{
it a\10\;
for i=0;i<10;i++
inp[a\i\];
for i=0;i<10;i++
disp[a\i\];
ret 0;
}
~BCG~
```

-> ~BCG~<functions>~BCG~
-> ~BCG~<function> <functions>~BCG~
-> ~BCG~<funsig> <funbody><functions>~BCG~

-> ~BCG~ fun <type> <id> <params> <funbody><functions>~BCG~
-> ~BCG~ fun it <id> <params> <funbody><functions>~BCG~
-> ~BCG~ fun it main <params> <funbody><functions>~BCG~
-> ~BCG~ fun it main <funbody><functions>~BCG~
-> ~BCG~ fun it main { <decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { <type> <varlist>;<decl> <statement> <return>
}<functions>~BCG~
-> ~BCG~ fun it main { it<varlist>;<decl> <statement> <return>
}<functions>~BCG~
-> ~BCG~ fun it main { it<id>\<dimen>\ <comma> <varlist>;<decl> <statement>
<return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<dimen>\ <comma> <varlist>;<decl> <statement>
<return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\< integerliteral> <comma> <dimen >\ <comma>
<varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\< 10> <comma> <dimen >\ <comma> <varlist>;<decl>
<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ <comma> <varlist>;<decl> <statement> <return>
}<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ;<decl> <statement> <return>
}<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < iterstat>;<statement > <return>
}<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for <statement> <statement >;<statement >
<return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for <expstat> <statement >;<statement >
<return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for < id><assop><E > <statement
>;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = <E > <statement >;<statement >
<return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = <mE > <statement >;<statement >
<return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = <pE > <statement >;<statement >
<return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = <rE > <statement >;<statement >
<return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = <integerliteral > <statement
>;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 <statement >;<statement > <return>
}<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0; <expstat >;<statement><statement >
<return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; <E >;<statement ><statement>
<return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ;<E><unop>;<statement >
<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ;<mE><unop>;<statement
><statement> <return> }<functions>~BCG~

```
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ;<pE><unop>;<statement >
<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ;<rE><unop>;<statement
><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ;<id ><unop>;<statement >
<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i<unop>;<statement >
<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ;<statement ><statement>
<return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ;<expstat
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ;<boolstat
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ;<rel >;<statement><statement>
<return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; <E> <compop> <E>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; <mE> <compop> <E>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; <rE> <compop> <E>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; <id> < <E>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < <mE>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < <pE>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < <rE>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < <integerliteral
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10;<statement><statement>
<return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10;<inpstat
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[<varlist>]
;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[<id>\dimen\
<comma> <varlist>] ;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\<E> \ <comma>
<varlist>] ;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\<mE> \ <comma>
<varlist>] ;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\<rE> \ <comma>
<varlist>] ;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\<id> \ <comma>
<varlist>] ;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \ <comma>
<varlist>] ;<statement><statement> <return> }<functions>~BCG~
```

```
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \]
;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] ;<
iterstat>;<statement ><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] ;< for
<statement>  <statement>;<statement ><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] ;< for
<expstat>  <statement>;<statement ><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] ;< for <
id><assop><E >  <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] ;< for i = <E >
<statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] ;< for i = <mE
>  <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] ;< for i = <pE >
<statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] ;< for i = < for i
= 0  <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] ;<  for i = 0;
<expstat >;<statement><statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] ;<  for i = 0
;<E><unop>;<statement > <statement><return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] ;<  for i = 0
;<mE><unop>;<statement ><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] ;<   for i = 0
;<pE><unop>;<statement > <statement><return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] ;<   for i = 0
;<id ><unop>;<statement > <statement><return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ;
i<unop>;<statement > <statement><return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++
;<expstat >;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++
;<rel >;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
<mE> <compop> <E> >;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
<rE> <compop> <E> >;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \]
 -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
<rE> <compop> <E> >;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
<rE> <compop> <E> >;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
i < <rE> >;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
i < <integerliteral >;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
i < 10;<statement><statement> <return> }<functions>~BCG~
```

```
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10;<statement><statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10;<outstat><statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; disp[<print>]<statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; disp[<id> <comma> <print >]<statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; disp[<id> <comma> <print >]<statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; disp[a\dimen\<comma> <print >]<statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; disp[a \id\<comma> <print >]<statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; disp[a \i \<comma> <print >]<statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; disp[a \i \];<statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; disp[a \i \]; <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; disp[a \i \]; ret <id > }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; disp[a \i \]; ret 0 }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; disp[a \i \]; ret 0 }~BCG~
```

```
10) ~BCG~
fun it main
{
it a,b,c;
c=mul [a,b];
ret 0;
}
fun it mul it a, it b {
it c;
c=a*b;
ret c;
}
~BCG~
```

```
-> ~BCG~<functions>~BCG~
        -> ~BCG~<function> <functions>~BCG~
        -> ~BCG~<funsig> <funbody><functions>~BCG~
        -> ~BCG~ fun <type> <id> <params> <funbody><functions>~BCG~
        -> ~BCG~ fun it <id> <params> <funbody><functions>~BCG~
        -> ~BCG~ fun it main <params> <funbody><functions>~BCG~
        -> ~BCG~ fun it main  <funbody><functions>~BCG~
```

-> ~BCG~ fun it main { <decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { <type> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it<varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it<id> <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a<comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a ,<varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, <id> <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; <expstat>;<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; <id> = <id><more>;<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=mul <more>;<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=mul [<args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=mul [<id> <comma> <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=mul [a, <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=mul [a, <id> <comma> <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=mul [a, b <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=mul [a, b];<return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret <id>; }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; }<function>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun <type> <id> <params> <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it <id> <params> <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul <params> <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul < type> <id> <comma> <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, < type> <id> <comma> <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b , <params > <funbody >~BCG~

->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ <decl> <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ <type> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ it<varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ it<id> <comma> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ it c <comma> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ it c;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ it c; <id><assop><E>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ it c; c = <E>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ it c; c = <E> * <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ it c; c = <mE> * <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ it c; c = <pE> * <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ it c; c = <rE> * <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ it c; c = <id > * <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ it c; c = a* <pE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ it c; c = a* <rE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ it c; c = a* <id >;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ it c; c = a*b ;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ it c; c = a*b ;<return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ it c; c = a*b ;ret <id>; } >~BCG~
->~BCG~ fun it main { it a, b, c; c=mul [a, b]; ret 0; } fun it mul it a, it b <{ it c; c = a*b ;ret c; } >~BCG~

11) ~BCG~
```
fun it main
{
it a,b,c;
c=func[a,b];
ret 0;
}
fun it func it a, it b {
```

```
it c;
c=a+b;
a=c*b;
b++;
c=a+b;
ret c;
}
```

```
-> ~BCG~<functions>~BCG~
        -> ~BCG~<function> <functions>~BCG~
        -> ~BCG~<funsig> <funbody><functions>~BCG~
        -> ~BCG~ fun <type> <id> <params> <funbody><functions>~BCG~
        -> ~BCG~ fun it <id> <params> <funbody><functions>~BCG~
        -> ~BCG~ fun it main <params> <funbody><functions>~BCG~
        -> ~BCG~ fun it main  <funbody><functions>~BCG~
        -> ~BCG~ fun it main  { <decl> <statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { <type> <varlist>;<decl>  <statement> <return>
}<functions>~BCG~
        -> ~BCG~ fun it main  { it<varlist>;<decl>  <statement> <return>
}<functions>~BCG~
        -> ~BCG~ fun it main  { it<id> <comma> <varlist>;<decl>  <statement> <return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a<comma> <varlist>;<decl>  <statement> <return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a ,<varlist>;<decl>  <statement> <return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, <id> <comma> <varlist>;<decl>  <statement> <return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c <comma> <varlist>;<decl>  <statement> <return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;<decl>  <statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;  <expstat>;<statement><return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   <id> = <id><more>;<statement><return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add <more>;<statement><return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add [<args>];<statement><return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add [<id> <comma>
        <args>];<statement><return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add [a, <args>];<statement><return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add [a, <id> <comma>
        <args>];<statement><return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add [a, b <args>];<statement><return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add [a, b ];<return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret <id>; }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; }<functions>~BCG~
```

-> ~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; }<function>~BCG~
-> ~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun <type> <id> <params> <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it <id> <params> <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func <params> <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func < type> <id> <comma> <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, < type> <id> <comma> <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b , <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ <decl> <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ <type> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it<varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it<id> <comma> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c <comma> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; <id><assop><E>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <E>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <E> + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <mE> + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <pE> + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <rE> + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <id > + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+ <pE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+ <rE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+ <id >;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ;<statement> <return> } >~BCG~

->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; <id><assop><E>; <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <E>; <statement><return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <mE>*<pE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <pE>*<rE>; <statement><return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <rE>*<id>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <id>*<b>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<expstat >;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<E >;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<E><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<mE><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<pE><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<rE><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<id><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;<expstat>; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;<id><assop><E >; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;c = <E >; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;c = <E >+<mE>; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;c = <mE >+<pE>; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;c = <pE >+<rE>; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;c = <rE >+<id>; <statement> <return> } >~BCG~

->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;c = <id >+<id>; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;c = a+b; <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;c = a+b; ret <id>;} >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <c >*<b>;<b>++;c = a+b; ret <c >;} >~BCG~

12) ~BCG~

```
fun it main{
it i;
disp[“Enter value of i”];
inp[i];
if i<10
i++;
disp[“value of i=”,i];
ret 0;
}
~BCG~
```

<program> -> ~BCG~<functions>~BCG~
-> ~BCG~<function> <functions>~BCG~
-> ~BCG~<funsig> <funbody><functions>~BCG~
-> ~BCG~ fun <type> <id> <params> <funbody><functions>~BCG~
-> ~BCG~ fun it <id> <params> <funbody><functions>~BCG~
-> ~BCG~ fun it main <params> <funbody><functions>~BCG~
-> ~BCG~ fun it main <funbody><functions>~BCG~
-> ~BCG~ fun it main { <decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { <type> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it<varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it<id> <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it i<comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it i<varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it i;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it i; <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it i; <outstat>;<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it i; disp[<print>];<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it i; disp["<stringliteral>" <comma> <print>];<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it i; disp[“Enter value of i”];<statement> <return> }<functions>~BCG~

-> ~BCG~ fun it main { it i; disp[“Enter value of i”]; <inpstat>;<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it i; disp[“Enter value of i”];inp [<varlist>] ;<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it i; disp[“Enter value of i”];inp [<id> <comma> <varlist>] ;<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it i; disp[“Enter value of i”];inp [i<comma> <varlist>] ;<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it i; disp[“Enter value of i”];inp [i<varlist>] ;<statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it i; disp[“Enter value of i”];inp [i ;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of i”];inp [i]; <condstat>;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value ii”];inp [i]; if<statement><statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of i”];inp [i]; if <expstat>;<statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of i”];inp [iif <id><assop><E><statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it idisp[“Enter value of i”];inp [i]; if I < <E><statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of i”];inp [i]; if i< <mE ><statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of i”];inp [i]; if i< <pE><statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of iinp [i]; if i< <rE><statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of i”];inp [i]; if i< <integerliteral><statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of i”];inp [i];if i< 10 <statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of i”];inp [i]; if i< 10 <expstat>;<statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of aiinp [i]; if i < 10; <E><statement> <statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of aiinp [i]; if i <10; <E><unop><statement> <statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of aiinp [i]; if i < 10; <id><unop><statement> <statement> <statement>/;<statement> <return> }<functions>~BCG~

->~BCG~ fun it main { it i; disp[“Enter value of aiinp [i]; if i < 10; i++;<statement> <statement> <statement>/;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of i”];inp [i];if a < 10; i++;<;<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it aidisp[“Enter value of i”];inp [i]; if i< 10; i++;<; <outstat>;<statement> <return> }<functions>~BCG~

->~BCG~ fun it main { it i; disp[“Enter value of i”];inp [i]; if i< 10; i++;< ;
disp["<stringliteral>" <comma> <print>];<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of i”];inp [i] if i< 10; i++;< ;
disp["“value of i=”, <print>];<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of i”];inp [i] if i< 10; i++;<;
disp["“value of i=”,<id> <comma> <print>];<statement> <return>
}<functions>~BCG~
->~BCG~ fun it main { ii; disp[“Enter value of i”];inp [i]; if i< 10; i++;< ;
disp["“value of i=”,i<comma> <print>];<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of i”];inp [i]; if i< 10; i++;< ;
disp["“value of i=”,i];<statement> <return> }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of i”];inp [i]; if i< 10; i++;< ;
disp["“value of i=”,i];<return> }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of i”];inp [i]; if i< 10; ai1 ;
disp["“value of i=”,i]; ret <id>; }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of i”];inp [i]; if i< 10; i++;< ;
disp["“value of i=”,i]; ret 0 ; }<functions>~BCG~
->~BCG~ fun it main { it i; disp[“Enter value of i”];inp [i]; if i< 10; i++;< ;
disp["“value of i=”,i]; ret 0 ; ~BCG~

13) ~BCG~
```
fun it main
{
it a,b,c;
c=div[a,b];
ret 0;
}
fun it div it a, it b {
it c;
c=a/b;
ret c;
}
~BCG~
```

-> ~BCG~<functions>~BCG~
-> ~BCG~<function> <functions>~BCG~
-> ~BCG~<funsig> <funbody><functions>~BCG~
-> ~BCG~ fun <type> <id> <params> <funbody><functions>~BCG~
-> ~BCG~ fun it <id> <params> <funbody><functions>~BCG~
-> ~BCG~ fun it main <params> <funbody><functions>~BCG~
-> ~BCG~ fun it main <funbody><functions>~BCG~
-> ~BCG~ fun it main { <decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { <type> <varlist>;<decl> <statement> <return>
}<functions>~BCG~
-> ~BCG~ fun it main { it<varlist>;<decl> <statement> <return>
}<functions>~BCG~
-> ~BCG~ fun it main { it<id> <comma> <varlist>;<decl> <statement> <return>
}<functions>~BCG~

-> ~BCG~ fun it main { it a<comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a ,<varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, <id> <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c; <expstat>;<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   <id> = <id><more>;<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=div<more>;<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=div[<args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=div[<id> <comma> <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=div[a, <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=div[a, <id> <comma> <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=div[a, b <args>];<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=div[a, b];<return> }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=div[a, b]; ret <id>; }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=div[a, b]; ret 0; }<functions>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=div[a, b]; ret 0; }<function>~BCG~
-> ~BCG~ fun it main { it a, b, c;   c=div[a, b]; ret 0; } fun <type> <id> <params> <funbody >~BCG~
->~BCG~ fun it main { it a, b, c;   c=div[a, b]; ret 0; } fun it <id> <params> <funbody >~BCG~
->~BCG~ fun it main { it a, b, c;   c=div[a, b]; ret 0; } fun it div<params> <funbody >~BCG~
->~BCG~ fun it main { it a, b, c;   c=div[a, b]; ret 0; } fun it div< type> <id> <comma> <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c;   c=div[a, b]; ret 0; } fun it divit a, <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c;   c=div[a, b]; ret 0; } fun it divit a, < type> <id> <comma> <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c;   c=div[a, b]; ret 0; } fun it divit a, it b , <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c;   c=div[a, b]; ret 0; } fun it divit a, it b <{ <decl> <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=div[a, b]; ret 0; } fun it divit a, it b <{ <type> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=div[a, b]; ret 0; } fun it divit a, it b <{ it<varlist>;<decl > <statement> <return> } >~BCG~

->~BCG~ fun it main { it a, b, c; c=div[a, b]; ret 0; } fun it divit a, it b <{ it<id> <comma> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=div[a, b]; ret 0; } fun it divit a, it b <{ it c <comma> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=div[a, b]; ret 0; } fun it divit a, it b <{ it c;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=div[a, b]; ret 0; } fun it divit a, it b <{ it c; <id><assop><E>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=div[a, b]; ret 0; } fun it divit a, it b <{ it c; c = <E>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=div[a, b]; ret 0; } fun it divit a, it b <{ it c; c = <E> / <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=div[a, b]; ret 0; } fun it divit a, it b <{ it c; c = <mE> / <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=div[a, b]; ret 0; } fun it divit a, it b <{ it c; c = <pE> / <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=div[a, b]; ret 0; } fun it divit a, it b <{ it c; c = <rE>/ <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=div[a, b]; ret 0; } fun it divit a, it b <{ it c; c = <id > / <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=div[a, b]; ret 0; } fun it divit a, it b <{ it c; c = a/ <pE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=div[a, b]; ret 0; } fun it divit a, it b <{ it c; c = a/ <rE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=div[a, b]; ret 0; } fun it divit a, it b <{ it c; c = a/ <id >;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=div[a, b]; ret 0; } fun it divit a, it b <{ it c; c = a/b ;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=div[a, b]; ret 0; } fun it divit a, it b <{ it c; c = a/b ;<return> } >~BCG~
->~BCG~ fun it main { it a, b, c; c=div[a, b]; ret 0; } fun it divit a, it b <{ it c; c = a/b ;ret <id>; } >~BCG~
->~BCG~ fun it main { it a, b, c; c=div[a, b]; ret 0; } fun it divit a, it b <{ it c; c = a/b ;ret c; } >~BCG~

```
14)~BCG~
fun it main
{
it a\10\;
for i=0;i<10;i++
inp[a\i\];
for i=0;i<10;i++
a\i\++;
ret 0;
}
~BCG~
```

-> ~BCG~<functions>~BCG~
-> ~BCG~<function> <functions>~BCG~

-> ~BCG~<funsig> <funbody><functions>~BCG~
-> ~BCG~ fun <type> <id> <params> <funbody><functions>~BCG~
-> ~BCG~ fun it <id> <params> <funbody><functions>~BCG~
-> ~BCG~ fun it main <params> <funbody><functions>~BCG~
-> ~BCG~ fun it main <funbody><functions>~BCG~
-> ~BCG~ fun it main { <decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { <type> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it<varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it<id>\<dimen>\ <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<dimen>\ <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\< integerliteral> <comma> <dimen >\ <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\< 10> <comma> <dimen >\ <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ <comma> <varlist>;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ;<decl> <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; <statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < iterstat>;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for <statement> <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for <expstat> <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for < id><assop><E > <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = <E > <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = <mE > <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = <pE > <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = <rE > <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = <integerliteral > <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0; <expstat >;<statement><statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; <E >;<statement ><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ;<E><unop>;<statement > <statement><return> }<functions>~BCG~

```
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ;<mE><unop>;<statement
><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ;<pE><unop>;<statement >
<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ;<rE><unop>;<statement
><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ;<id ><unop>;<statement >
<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i<unop>;<statement >
<statement><return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ;<statement ><statement>
<return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ;<expstat
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ;<boolstat
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ;<rel >;<statement><statement>
<return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; <E> <compop> <E>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; <mE> <compop> <E>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; <rE> <compop> <E>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; <id> < <E>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < <mE>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < <pE>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < <rE>
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < <integerliteral
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10;<statement><statement>
<return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10;<inpstat
>;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[<varlist>]
;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[<id>\dimen\
<comma> <varlist>] ;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\<E> \ <comma>
<varlist>] ;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\<mE> \ <comma>
<varlist>] ;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\<rE> \ <comma>
<varlist>] ;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\<id> \ <comma>
<varlist>] ;<statement><statement> <return> }<functions>~BCG~
```

-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \ <comma>
<varlist>] ;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \]
;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] ;<
iterstat>;<statement ><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] ;< for
<statement> <statement>;<statement ><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] ;< for
<expstat> <statement>;<statement ><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] ;< for <
id><assop><E > <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] ;< for i = <E >
<statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] ;< for i = <mE
> <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] ;< for i = <pE >
<statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] ;< for i = < for i
= 0 <statement >;<statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] ;< for i = 0;
<expstat >;<statement><statement > <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] ;< for i = 0
;<E><unop>;<statement > <statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] ;< for i = 0
;<mE><unop>;<statement ><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] ;<   for i = 0
;<pE><unop>;<statement > <statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] ;<   for i = 0
;<id ><unop>;<statement > <statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ;
i<unop>;<statement > <statement><return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++
;<expstat >;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++
;<rel >;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
<mE> <compop> <E> >;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
<rE> <compop> <E> >;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \]
 -> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
<rE> <compop> <E> >;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
<rE> <compop> <E> >;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
i < <rE> >;<statement><statement> <return> }<functions>~BCG~
-> ~BCG~ fun it main { it a\<10\ ; < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
i < <integerliteral >;<statement><statement> <return> }<functions>~BCG~

```
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10;<statement><statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10;<statement><statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10;<outstat><statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; disp[<print>]<statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; <E><unop><statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; <mE><unop><statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; <pE><unop><statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; <rE><unop><statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; <id>++;<statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; a\i\++;<statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; a\i\++; <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; a\i\++;ret <id > }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; a\i\++; ret 0 }<functions>~BCG~
        -> ~BCG~ fun it main  { it a\<10\ ;  < for i = 0 ; i++ ; i < 10; inp[a\i \] for i = 0 ; i++ ;
        i < 10; a\i\++;ret 0 }~BCG~
```

```
15) ~BCG~
fun it main
{
it a,b,c;
c=func[a,b];
ret 0;
}
fun it func it a, it b {
it c;
c=a+b;
a=c*b;
b++;
c=a+b;
ret c;
}
```

```
-> ~BCG~<functions>~BCG~
        -> ~BCG~<function> <functions>~BCG~
        -> ~BCG~<funsig> <funbody><functions>~BCG~
        -> ~BCG~ fun <type> <id> <params> <funbody><functions>~BCG~
        -> ~BCG~ fun it <id> <params> <funbody><functions>~BCG~
        -> ~BCG~ fun it main <params> <funbody><functions>~BCG~
        -> ~BCG~ fun it main  <funbody><functions>~BCG~
        -> ~BCG~ fun it main  { <decl> <statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { <type> <varlist>;<decl>  <statement> <return>
}<functions>~BCG~
        -> ~BCG~ fun it main  { it<varlist>;<decl>  <statement> <return>
}<functions>~BCG~
        -> ~BCG~ fun it main  { it<id> <comma> <varlist>;<decl>  <statement> <return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a<comma> <varlist>;<decl>  <statement> <return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a ,<varlist>;<decl>  <statement> <return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, <id> <comma> <varlist>;<decl>  <statement> <return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c <comma> <varlist>;<decl>  <statement> <return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;<decl>  <statement> <return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;  <expstat>;<statement><return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   <id> = <id><more>;<statement><return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add <more>;<statement><return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add [<args>];<statement><return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add [<id> <comma>
        <args>];<statement><return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add [a, <args>];<statement><return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add [a, <id> <comma>
        <args>];<statement><return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add [a, b <args>];<statement><return>
        }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add [a, b ];<return> }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret <id>; }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; }<functions>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; }<function>~BCG~
        -> ~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun <type> <id> <params>
        <funbody >~BCG~
        ->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it <id> <params>
        <funbody >~BCG~
        ->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func <params>
        <funbody >~BCG~
        ->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func < type> <id>
        <comma> <params > <funbody >~BCG~
        ->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, <params >
        <funbody >~BCG~
        ->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, < type> <id>
        <comma> <params > <funbody >~BCG~
```

->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b , <params > <funbody >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ <decl> <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ <type> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it<varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it<id> <comma> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c <comma> <varlist>;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c;<decl > <statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; <id><assop><E>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <E>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <E> + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <mE> + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <pE> + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <rE> + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = <id > + <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+ <pE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+ <rE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+ <id >;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; <id><assop><E>; <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <E>; <statement><return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <mE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <mE>*<pE>;<statement> <return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <pE>*<rE>; <statement><return> } >~BCG~
->~BCG~ fun it main { it a, b, c;   c=add [a, b]; ret 0; } fun it func it a, it b <{ it c; c = a+b ; a = <rE>*<id>;<statement> <return> } >~BCG~

```
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <id>*<b>;<statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<expstat >;<statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<E >;<statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<E><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<mE><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<pE><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<rE><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<id><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<b><unop>;<statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<b>++;<statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<b>++;<expstat>; <statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<b>++;<id><assop><E >; <statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<b>++;c = <E >; <statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<b>++;c = <E >+<mE>; <statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<b>++;c = <mE >+<pE>; <statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<b>++;c = <pE >+<rE>; <statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<b>++;c = <rE >+<id>; <statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<b>++;c = <id >+<id>; <statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<b>++;c = a+b; <statement> <return> } >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<b>++;c = a+b; ret <id>;} >~BCG~
->~BCG~ fun it main  { it a, b, c;   c=add [a, b ]; ret 0; } fun it func it a, it b <{ it c; c
= a+b ; a = <c >*<b>;<b>++;c = a+b; ret <c >;} >~BCG~
```

Comparative study of Dart & Dataflex programming language

Objects and Classes

- **Dart**

Classes

Dart is an object-oriented language. Dart is an object-oriented language which supports classes and interfaces. A class in terms of OOP is a blueprint for creating objects. Dart gives built-in support for this concept called class.

Declaration:

Declaration in dart starts with the keyword class followed by class-name.

Syntax:

```
class class_name {
  <fields>
  <getters/setters>
  <constructors>
  <functions>
}
```

Fields –Variables are declared in a class.

Setters and Getters − Program initialization and retrieval of values declared fields of a class.

Constructors – Allocation of memory in objects of the class.

Functions − Functions represent actions an object can take.

Objects

In OOPs an object as any entity that has a defined boundary. An object has the following −

State −The fields of a class represent the object’s state.

Behaviour − Describes what an object can do.

Identity − Two or more objects can share the state and behavior but not the identity.

Example :

```
Void main(){
employee e=new employee();
e.test();
e.test1();
}
```

In the above example, the dot operator (.) is used to combine objects with clsses.

- **Dataflex**
  Programs must be broken down into small, manageable units. In DataFlex, one of these units is the package file. Typically, a package file contains a class definition. Classes are the reusable tools of object oriented programming. When a class package is used by a program (through a use command naming the file), the class definition is included in the program at compile time. That program's source code may thereafter create objects based on this class. A full set of high-level, sophisticated class packages contain classes that comprise your development toolkit and are called the data-entry classes. They are the building blocks that allow you to create complex data-entry views. The classes allow you to present your data in a variety of formats such as forms, tables, pop-up lookup lists, text editors, checkboxes and radio lists.
  We need to introduce another type of package called an object package. Unlike a class package, which just includes class code, an object package includes the code to create one or more objects. Because a package file is only included once in a program, this means that an object package will be used to create its object(s) one time in the program. Object packages are most often used to create global "utility" objects. A global object is an object that all other objects may access.

Encapsulation (information hiding)

- **Dataflex**
  A view is a specialized object package. It is a stand-alone data-entry unit. Encapsulation was realized by placing separate views in separate programs. While we can still use this technique, with object technology, we now can place multiple views in a single program. If an object cannot be accessed using this access syntax, you should not be accessing it from the object you are trying to access it from. For example, no object should directly communicate with any of its grandchild objects. This is a violation of the principle of encapsulation, and the approved access method discourages this.
  One consequence of breaking encapsulation is to make it more complicated to reuse the class in other programs (a key benefit of object-oriented programming). The reason this occurs is that the class definition now contains an outside dependency upon any global functions that it uses. Its use is therefore dependant upon the existence of these functions. However, it should be remembered that any new global procedure or function you write that is used. The use of relative object addressing

makes it possible to maintain proper encapsulation while at the same time providing an object access scheme that is easy to code and easy to maintain. Relative object addressing is an important part of the DataFlex object model.

Inheritance

- **Dart**

Dart supports Inheritances which is the ability of a program to create new classes from an existing class. We have a parent class/super class and newly created classes are called the child/sub classes.

A class inherits from another class using the keyword extends. Child class inherits all properties and methods except constructors from the parent class.

Syntax

class child_class_name. extends parent class_name

Example,

```
Class parts{

Printf("each part name");

}class car extends parts{}
```

Dart supports different types of inheritance:

1) Single inheritance.
2) Multiple inheritance.
3) Multi-level inheritance.

Inheritance is similar to java.

Example for multi-level inheritance

```
Void main()
{   var c = new door();
  c.str = "hello";
  print(c.str);
}
Class car {
  String str;
}
class Child extends car {}
class door extends Child {}
```

In multilevel inheritance, the class door inherits the attribute from car and child classes.

- **Dataflex**

Each class that you declare automatically inherits all of the components of its superclass. A class declaration imports the methods of a class (other than its superclass) by executing an Import Protocol statement in its class definition. The syntax of the import protocol statement is shown below:
Import_Class_Protocol {imported-class}
where {imported-class} is the name of the class whose attributes are being imported.

An example of a class definition with multiple inheritance is shown below.

```
Class cMessage_Mixin is a Mixin
Procedure Define_cMessage_Mixin
Property String psMessage ""
End_Procedure
Procedure DoShowMessage
String sMessage
Get psMessage To sMessage
Send Info_Box sMessage
End_Procedure
End_Class
Class cMessageButton is a Button
Import_Class_Protocol cMessage_Mixin
Procedure Construct_Object
Forward Send Construct_Object
Send Define_cMessage_Mixin // important!!!
End_Procedure
Procedure OnClick
Send DoShowMessage
End_Procedure
End_Class
```

The above example declares two new classes. The first one, cMessage_Mixin, is designed specifically to be mixed into the class definition of some other class using the multiple-inheritance model. Classes designed for this purpose are called Mixin classes (they are based on the DataFlex Mixin class). You would never instantiate objects of any Mixin class; they are designed only as building blocks for other classes. The second class in the example, cMessageButton, is based on the button class but it also imports all methods from the cMessage_Mixin class. Notice that the constructor method of this class calls the inherited Define_cMessage_Mixin method. This is important because it demonstrates how properties can be defined in a Mixin and then imported. Properties defined in the constructor of a Mixin class would not get imported because the Mixin's constructor is never called.

<u>Overriding</u>

- **Dart**

Method Overriding is a mechanism by which the child class redefines a method in its parent class.

For example,

```
import ‘dart:io’;
void main() {
child c=new child();
c.m(5);
}
Class parent{
Void m(int a){ print(“value of a ${a}”);}
}
Class child extends parent{
@override
Void m(String b)
{ print(“value of b ${b}”);}
}
```

- ❖ **Dataflex**

  As of VDF15.0 dataflexing recommends that you not use message overloading. The technique is now considered to be obsolete.. There are two simple alternatives to message overloading that have always been available. Consider the following:

```
Class cMyClass is a cObject

Procedure Foo Overloaded
Showln "Value is Default Name"
End_Procedure

Procedure Foo Overloaded string sVal
Showln "Value is " sVal
End_Procedure

Procedure CallFoo
Send Foo // will show "Default Name"
Send Foo "My Name" // will show "My Name"
End_Procedure
End_Class
```

The simplest alternative is to create different names for your methods:

```
Class cMyClass is a cObject

Procedure Foo
```

```
Showln "Value is Default Name"
End_Procedure

Procedure Foo1 string sVal
Showln "Value is " sVal
End_Procedure

Procedure CallFoo
Send Foo // will show "Default Name"
Send Foo1 "My Name" // will show "My Name"
End_Procedure
End_Class
```

The other alternative is to use num_arguments

```
Class cMyClass is a cObject

Procedure Foo string sVal
If (num_arguments=0) Begin
Showln "Value is Default Name"
End
Else Begin
Showln "Value is " sVal
End
End_Procedure

Procedure CallFoo
Send Foo // will show "Default Name"
Send Foo "My Name" // will show "My Name"
End_Procedure
End_Class
```

Dynamic binding versus static binding

- **Dart**

Let d be the declaration of a static or instance variable v. If d is an instance variable, then the invocation of the implicit getter of v evaluates to the value stored in v. If d is a static or library variable then the implicit getter method of v executes as follows:

- Non-constant variable declaration with initializer. If d is of one of the forms var v = e; , T v = e; , final v = e; , final T v = e;, static v = e; , static T v = e; , static final v = e; or static final T v = e; and no value has yet been stored into v then the initializer expression e is evaluated. If, during the evaluation of e, the getter for v is invoked, a CyclicInitializationError is thrown. If the evaluation succeeded yielding an object o, let r = o, otherwise let r = null. In any case, r is stored into v. The result of executing the getter is r.

- Constant variable declaration. If d is of one of the forms const v = e; , const T v = e; , static const v = e; or static const T v = e; the result of the getter is the value of the compile time constant e. Note that a compile time constant cannot depend on itself, so no cyclic references can occur. Otherwise
- Variable declaration without initializer. The result of executing the getter method is the value stored in v.

Abstract classes

- **Dart**

The keyword abstract is used to define abstract class. Abstract classes are useful for defining interfaces, often with some implementation. Abstract classes have abstract methods. Instance, getter, and setter methods can be abstract. To make a method abstract, semicolon (;) is used instead of a method body:

Syntax :

```
Abstract class class_name{
{ void abstract_method();
}}
```

For example,

```
Abstract class car{
{ void drive();
}}
```

Here void drive is an abstract method and calling an abstract method results in a runtime error.

- **Dataflex**

  The key words you need to know are forward, delegate, and mixin. They are the basic reasons for using abstracts.

  Forward get myFunction to rVal
  This will look down into the class structure, abstracts and base classes to resolve the "get myFunction" before checking parent objects for a resolution.

  Delegate get myFunction to rVal
  This will check parent objects for a message resolution without going into the class. This would be used when you do not want a method of the class to be used, but want the message to be resolved by a parent.

  Mixins allow common properties and code to be moved into classes having different base classes. (this is a VDF thing because we can not have more than one base class). I mostly use mixins for getting common code into multiple types of classes, such as forms (dbform,dbspinform, dbcomboform, etc) and containers. If you look at the VDF pkg files, you will find mixins for common navigation and db/dd awareness.

```
function myFunction returns integer
integer rval

get myFunction to rval
//recursion is ok if you control it, else endless loop

forward get myFunction to rval
//if defined in the class structure, that value will be returned else parent objects until
resolution.

delegate get myFunction to rval
//will check parent objects to get value. Will not return anything
from within this class structure.

function_return rval
end_function
```

There will be times when you want to include a message so you will be sure it is resolved. An abstract can be used to make sure a forward will be found at a lower level in a class

Interface

- **Dart**

Dart does not have a syntax for declaring interfaces. Class declarations are themselves interfaces in Dart. Classes should use the keyword implements to define interface.

Syntax

```
Class identifier implements interface_name
```

Multiple interfaces can be implemented for a class separated by comma.

Syntax

```
Class identifier implements interfaceA,interfaceB,interfaceC
```

Example for interface

```
Class door{
Void print()
{ print(“size”);
}}
Class car implements door{
Void print()
{ print(“color”);
```

```
            }}
Example for multiple interface
Class door{
            Void print()
            { print("size");
            }}
Class body{
            Void print()
            { print("body type");
            }}

            Class car implements door,body{
            Void print()
            { print("final car");
            }}
```

- **Dataflex**

  All of the utilities themselves use the DataFlex UIMS, and their interfaces are written entirely in DataFlex. Some have their own executable files, however, and do not use the DataFlex runtime.

  The User-Interface Management System (UIMS) is a DataFlex programming environment in which programmers can create and maintain attractive, intuitive, and powerful user interfaces for their database applications. UIMS applications are event-driven, meaning that users and not the program determine the order of events as the application runs. In an event-driven system, if users can see something on the screen, they should be able to move interaction to it. This is referred to as modeless or non-modal navigation. This is contrasted 2 Prrogrrammiing iin DattaFlleex 21 with modal navigation where users are required to finish one task before moving on to the next. Modal navigation is sometimes referred to as being procedural. UIMS applications support both modal and non-modal navigation. A real-world application will be primarily nonmodal (user-driven) mixed with occasional presentation of modal tasks (sometimes the program must take control).
  From the users' point of view, event-driven navigation is both powerful and intuitive (e.g., they simply "point and select" with their mouse). From an implementation point of view, an event-driven system can be quite complex. There are two primary reasons for this: the mechanisms required to provide low-level support are complicated, and the high-level tools required to create event-driven applications must be quite sophisticated. In addition, these high-level tools must allow the developer to seamlessly integrate event-driven user interfaces with multi-file, multi-record, multi-user databases. The goal of the UIMS is to provide both the low-level support and a highlevel toolkit, thus allowing the developer to bypass most of the complexities of programming event-driven applications.

Parallel program

- ❖ **Dart**

Dart programs run in a single isolate by default. Dart provides several asynchronous programming techniques like Futures and Streams that Dart does not use capacities of modern multicore processors by default.
To process programs in parallel this package provides a parallel map function <code>pmap</code> for easy parallelization.

Syntax:
parallel(variable_name).pmap(function)

For example,

```
        Final s= new iterable.generate(15,(i)=>i+1);
        parallel(s).pmap(new fib())
                .reduce((a,b)=>a+b)
                .then((r) {
                Print(r);
                });
```

- ❖ **Dataflex**
  No threads or multitasking options are available in DataFlex. Hence it does not support parallel programming.

Network programming

- ❖ **Dart**

Network socket programming with dart SDK 1.0 which is used for full web development environment that competes with javascript. It is similar to javascript in the browser and node.js on the server. A socket is an endpoint to an interprocess communications connection across a network. They are usually implemented in the transport layer of the OSI model. Dart socket objects are implemented on TCP/IP. UDP is now supported in dart. There are two classes from the dart:io API. The first is Socket which is used to establish a connection to a server as a clien and the second is ServerSocket which is used to create a server and accept client connections.

Client connections
In the below example, the Socket class has a static method called Connect(host,int port). The host parameter can be either a String with a host name or IP address, or an InternetAddress object. Connect will return a Future<Socket> object that will make the connection asynchronously. To know when a connection is actually made, we will register a Future.then(void onValue(T value)).

```
void main() {
  Socket.connect("google.com", 80).then((socket) {
    print('Connected to: '
      '${socket.remoteAddress.address}:${socket.remotePort}');
    socket.destroy();
  });
}
```

Here 80 represent port 80. Port 80 is the port that serves webpages. After the socket is connected to the server, the IP and port that it is connected to are printed to the screen and the socket is shutdown. By shutting down the socket using Socket.destory() we are telling dart that we don’t want to send or receive any more data on that connection.

Server Sockets

Making remote connections to a server is easy to use in Dart Socket object. To connect remote clients with others we can use the ServerSocket object. In order to create a server that can handle client connections, we must first bind to a specific TCP port. To do this we use the static ServerSocket.bind(address,int port) method. This will return a Future<ServerSocket>. Again we will use Future.then(void onValue(T value)) method to register a callback so that we can know when the socket has been bound to the port. Make sure to choose a port higher than 1024. Ports lower than that are in the reserved range and may require root or administrator permissions to bind.

For example

```
import 'dart:io';

void main() {

  ServerSocket.bind(InternetAddress.ANY_IP_V4, 4567).then(

    (ServerSocket server) {

      server.listen(handleClient);

    } );

}

void handleClient(Socket client){

  print('Connection from '

    '${client.remoteAddress.address}:${client.remotePort}');

  client.write("Hello from simple server!\n");

  client.close();
```

}

Security

- ❖ **Dataflex**

The network connection between the SPLF server and the application servers attached to it can be configured using non-internet, Network Address Translation (NAT) addresses and other firewall mechanisms. In such a configuration, the SPLF server is internet connected – it has a public IP address. The application server(s) connected to the SPLF server will have internal, non-internet-exposed, NAT addresses that preclude direct access by internet users. With this configuration, applications and, just as importantly, application data, are never accessible from the internet thereby providing a highly secure environment.

Internet tool

- ❖ **Dart**

HTTP (Hypertext Transfer Protocol) is a communication protocol used to send data from one program to another over the internet. At one end of the data transfer is a server and at the other end is a client. The client is often browser-based (either a user typing in a browser or a script running in a browser), but might also be a standalone program.

The server binds to a host and port (it makes an exclusive connection to an IP address and a port number). Then the server listens for requests. Because of Dart's asynchronous nature, the server can handle many requests at a single time, as follows:

- Server listens
- Client connects
- Server accepts and receives request (and continues to listen)
- Server can continue to accept other requests
- Server writes response of request or several, possibly interleaved, requests
- Server finally ends (closes) the response(s).

In Dart, the dart:io library contains the classes and functions you need to write HTTP clients and servers. In addition, the http_server package contains some higher-level classes that make it easier to write clients and servers.

The client is a basic HTML web-page. No Dart code is involved in the client. The client request is made from the browser to the Dart server through an HTML form within make_a_guess.html, which provides an automatic way to formulate and send client HTTP requests. The form contains the pull-down list and the button. The form also specifies the URL, which includes the port number, and the kind of request (the request method). It might also include elements that build a query string.

- ❖ **Dataflex**

The DataFlex WebApp Server is an integrated component of DataFlex, for deploying browser-based applications, Web Services and service oriented architecture (SOA) solutions. Where static web pages are no longer adequate, building, publishing and consuming web services is part of the DataFlex Studio. Developers can focus on writing the application logic from which DataFlex will automate the process of creating a publishable web service. Automatic support for multi-user database operations. The applications that you can create within DataFlex manage multi-user concurrency without the need for any special coding.

Mobile application

- **Dart**

  Mobile application can be developed using Dart. Frame work for mobile application is called Flutter. Apps built with Flutter are largely indistinguishable from those built using the Android SDK, both in terms of looks and performance. Running at 60 fps, user interfaces created with Flutter perform far better than those created with other cross-platform development frameworks such as React Native and Ionic. Features of Flutter are:

1. Flutter uses Dart, a fast, object-oriented language with several useful features such as mixins, generics, isolates, and optional static types.
2. Flutter has its own UI components, along with an engine to render them on both the Android and iOS platforms. Most of those UI components, right out of the box, conform to the guidelines of Material Design.
3. Flutter apps can be developed using IntelliJ IDEA, an IDE that is very similar to Android Studio.

Flutter has almost everything a developer might look for in a cross-platform mobile app development framework. Flutter apps run on Android 4.1 or newer, and iOS 8 or newer.

- **Dataflex**

  DataFlex currently offers the developer the ability to create Desktop applications or web applications. There are two different frameworks for web applications. One is more geared towards web application that look and feel like a desktop application and the other more geared towards mobile devices such as tablets and phones. Both of them create web applications. Sooner or later you will be faced with the need to offer a true native mobile application.

  Example of a mobile application for customer information.

  The data model in Visual Studio looks as follows

```
public class Customer
{
        public string Number { get; set; }
        public string Name { get; set; }
}
```

The next step is to call the web service and retrieve the data in App.cs (the application object) we added the following

```
public List<Customer> customerData;
```

```
private async void LoadData()
{
    string uri =
String.Format("http://localhost/WebOrderMobile_18_1/OrderService.wso/GetCustomers
/");
    HttpClient cl = new HttpClient();
    var content = await cl.GetStringAsync(uri);
    customerData = content.Deserialize<Customer[]>().ToList();
}
```

the method has to be marked async as it will make and await an asynchronous call to the web service
it then simply calls the web service and deserializes the returned JSON data.
The customer page now needs to show that data in the ListView object

```
public sealed partial class customersPage : Page
{
    public customersPage()
    {
        this.InitializeComponent();

        FillData();
    }

    private async void FillData()
    {
        custListView.ItemsSource = ((App)App.Current).customerData;
    }

}
```

we simply set the ItemsSource property of the ListView object to our customerData property in the application object.

Database support

- **Dart**

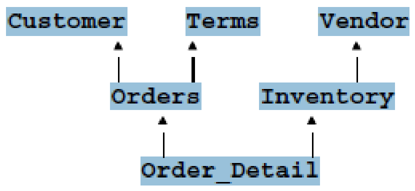
Dart does not have database like java. It can be combined with drivers MYSQL, MongoDB etc. MYSQL is a Relational database and MongoDB is NoSQL. Driver for MYSQL is SQLJockey and for MongoDB the driver is called mongo_dart.

- **Dataflex**

Visual DataFlex’s architecture is designed for building Windows and web applications, web services and service oriented architecture (SOA) using SQL-based DBMS servers such as Microsoft SQL Server, IBM DB2, Pervasive.SQL, Oracle and ODBC data sources. The FLEXquarters ODBC driver, is a fully functional driver that can read and write DataFlex version 2.3 and 3.x format files. It requires an ODBC compliant front-end application such as Microsoft Access, Visual Basic, PowerBuilder, Delphi, Microsoft Word or Microsoft Excel. Once installed, it will allow these applications to read and write DataFlex files in the same fashion as other SQL data.

DataFlex’s DBMS has the capacity to manage very large data files within a single application. It supports file relationships in multi-level hierarchies (many records in a "child" file relate to one record in a "parent" file). It supports the finding of records in data files through B+ ISAM (Indexed Sequential Access Method) indexes, and can find one among several million records in a fraction of a second under typical data and hardware conditions. The DBMS is inherently multi-user, both for multi-user environments such as UNIX, and for single-user environments such as DOS adapted for multiple users through Local Area Network (LAN) software. Multiuser locking facilities are provided that do not limit data access at all for reading, and confine access restrictions to brief periods of only a fraction of a second for writing.

Example:
A typical database file structure:



The code for the data-set structure:
Object View_Object is an Entry_View_Client no_image
Object Customer_DS is a Customer_Data_Set no_image
End_Object
Object Terms_DS is a Terms_Data_Set no_image

```
End_Object
Object Vendor_DS is a Vendor_Data_Set no_image
End_Object
Object Orders_DS is an Orders_Data_Set no_image ;
updating (Customer_DS(current_object)) ;
(Terms_DS(current_object))
End_Object
Object Inventory_DS is an Inventory_Data_Set no_image ;
updating (Vendor_DS(current_object))
End_Object
Object OrderDtl_DS is an OrderDtl_Data_Set no_image ;
updating (Orders_DS(current_object)) ;
(Inventory_DS(current_object))
Begin_Constraints
constrain OrderDtl relates to Orders
End_Constraints
```

Refinement

Access specifier

Dart

Dart doesn’t have the keywords public, protected, and private. If an identifier starts with an underscore (_), it’s private to its library.

Syntax
_identifier

Example

```
void _log(msg) {
  print("Log method called in loggerlib msg:$msg");
}
```

Features of Object

Dart

Dart represents data in the form of objects. Every class in Dart extends the Object class. The period operator **(.)** is used in conjunction with the object to access a class’ data members. every time a function is called, a reference to the object is required. The **cascade operator** can be used as shorthand in cases where there is a sequence of invocations. The cascade (..) operator can be used to issue a sequence of calls via an object.

Example:

```
class Student {
  void test_method() {
    print("This is a  test method");
  }
  void test_method1() {
  print("This is a  test method1"); }}
void main() {
  new Student()
```

```
   ..test_method()

   ..test_method1();

 }
```

Dataflex

Object declarations in Visual DataFlex can be nested, allowing you to declare one object inside another. The DataFlex Send, Set and Get statements instruct an object to execute a method. If the object does not have a definition for the method then the message will be automatically passed to the object's parent. The process of passing an unresolved message on to the parent object is called delegation.
The DataFlex object accessing method allows you to directly access all objects that you should be able to access. When an object handle is requested, through either a move command (move oMyObj to hoMyObj) or as part of message (Send MyMgs of oMyObj) the following steps are taken to
find the named object and return its handle ID.
1. The object sending the message looks for a child object with the target name. If an object is found, the object's handle is returned.
2. The object sending the message checks to see if there is a properly named sibling object. If found, a handle is returned. At this point it is actually looking at the parent's child-objects.
3. The object sending the message will now look at all child objects belonging to its grandparent object (siblings of the parent). If found, the handle is returned.
4. This process of moving up to an ancestor object and checking all children continues until an object is found, or the outer-most parent is tested and the target is still not found, in which case a handle value of zero is returned.

Mobile application

- **Dart**
  Mobile application can be developed using Dart. Frame work for mobile application is called Flutter. Apps built with Flutter are largely indistinguishable from those built using the Android SDK, both in terms of looks and performance. Running at 60 fps, user interfaces created with Flutter perform far better than those created with other cross-platform development frameworks such as React Native and Ionic. Features of Flutter are:
  1. Flutter uses Dart, a fast, object-oriented language with several useful features such as mixins, generics, isolates, and optional static types.
  2. Flutter has its own UI components, along with an engine to render them on both the Android and iOS platforms. Most of those UI components, right out of the box, conform to the guidelines of Material Design.
  3. Flutter apps can be developed using IntelliJ IDEA, an IDE that is very similar to Android Studio.

Flutter has almost everything a developer might look for in a cross-platform mobile app development framework. Flutter apps run on Android 4.1 or newer, and iOS 8 or newer.

- **Dataflex**

  DataFlex currently offers the developer the ability to create Desktop applications or web applications. There are two different frameworks for web applications. One is more geared towards web application that look and feel like a desktop application and the other more geared towards mobile devices such as tablets and phones. Both of them create web applications. Sooner or later you will be faced with the need to offer a true native mobile application.

Database support

- **Dart**

Dart does not have database like java. It can be combined with drivers MYSQL, MongoDB etc. MYSQL is a Relational database and MongoDB is NoSQL. Driver for MYSQL is SQLJockey and for MongoDB the driver is called mongo_dart.

- **Dataflex**

DataFlex’s DBMS has the capacity to manage very large data files within a single application. An important part of DataFlex is its database manager, so DataFlex is sometimes called a "DBMS". The database manager performs its basic work with data on disk, retrieving data from disk, manipulating the data, importing data, outputting data, and placing new or changed data back onto disk. Like all data on disk, DataFlex data is in files.