MIDDLE EAST TECHNICAL UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

CENG 491
SENIOR DESIGN PROJECT
FINAL DESIGN REPORT
FALL 2007

Group Name: Hellim

Group Members:

Kutay YILDIRICI	e143393@metu.edu.tr	1433937
Tayfun ÇAKICIER	e154349@metu.edu.tr	1543495
Halit Emre SAYILIR	e143385@metu.edu.tr	1433853
Anil KOYUNCU	e143380@metu.edu.tr	1433804

Group Mail: hellim-ltd@googlegroups.com

Project Title: CStar - Optimizing C Compiler

Table of Contents:

1.Introduction:

1.1.Detailed Problem Definition:

The optimization is a very important topic for compilers. Performance is increasingly being important nowadays in the market. An optimized and tuned program can run much more fast and better.

Our first aim is to implement optimizations in order to reduce run time of programs. Also a well developed implementation must not increase the compilation time too much. In order to make good implementations we must develop good and efficient algorithms for optimizations. Also our implementations must not change the functionality of the programs.

We will think in two dimensions for optimizations; time or space. Optimizations like Dead Code Elimination, Constant Folding may increase compile time but will decrease the size of the program. Optimizations like Basic Block Ordering, Strength Reduction will increase the size of program, but can dramatically decrease the run time of the program.

Our second aim is to implement as much as optimizations we can. The increased variety of simple implementations is better than having less number of complex optimizations. Optimizations will be done over intermediate representation of the framework.

In our project we will use the framework QuickC developed by CStar. QuickC is a design by contract framework. Most of the important functions (like creating expression trees, managing optimizations, etc…) are ready for us to use for implementing optimizations. For this project we will implement and add our implementations to the framework.

With the features of the framework, we can call any optimization, any times and any moment during the compilation. For example we can do the Constant Folding at the beginning of the compilation, after we can call the Common Subexpression Elimination optimization, then we can do the Constant Folding optimization again.

Optimizations are made through the “Anatrop” (analysis-transformation-optimization) and “AnatropManager” classes in the framework.

All optimizations have their options that can be set with the “Option” class of the framework. For example we can set properties for optimizations to recursively call themselves again if it had made a change on the program. We can set the number of maximum calls for an optimization.

Also the optimizations can be done in the specified scope like Basic Blocks, Program, Statements, etc…

Rather than optimizations we will also develop an Optimization Manager for the framework. The Optimization Manager will be implemented as an Anatrop in the framework. We will implement the manager in two modes; Interactive Mode and Normal Mode. In Normal mode, the manager will read the names and usages of the optimizations from an external file. In Interactive Mode, we will get the names, scopes and options of the optimizations from the user via standard input. Then we will set the options for optimizations and execute them.

We will also develop a Test Case Generator for the framework. It will get target and option files from user. It will read the options from these files and set these in the framework. Then it will dump generated C Code. At the end these will be sent to output.

1.2.Design Constraints and Limitations:

1.2.1.Constaints:

Experience: Although we have participated in many software projects and homeworks , our current project is harder than them because of new concepts. Our group is getting familiar to framework everyday , but some detailed usages must be examined correctly. Also it can be difficult to handle some unexpected problems about this new concept.

Time: The project must be finished by June and also we should provide at least two of optimizations at the end of this semester. We should use our time efficiently to not fall behind the schedule. So we should follow our works due to our Gantt Chart.It can be seen at table 1.a

Performance: It is important that our optimizations must work efficiently. So we must design and build our optimizations to be work with high performance. Then we should make some testing to confirm. Dhrystone and Whetstone benchmark tools will be used for time calculations.

1.2.2.Limitations:

Structure: As known , the compiler has got a complicated structure. Everything is handled by commands which are entered in Linux terminal. Also test case generator is handled by extra commands and the optimization manager is handled by an external file.

Platform: Our project is set on Linux machines. Also we should make our implementations by framework. There are important classes and functions in framework that should be used or inherited in order to integrate our optimizations to framework and use them.

Language: C++ programming language should be used as all framework , necessary class and functions were written in C++.

2.1Use Case Diagrams:

2.1.1 Use Case Diagram of The Test Case Generator:

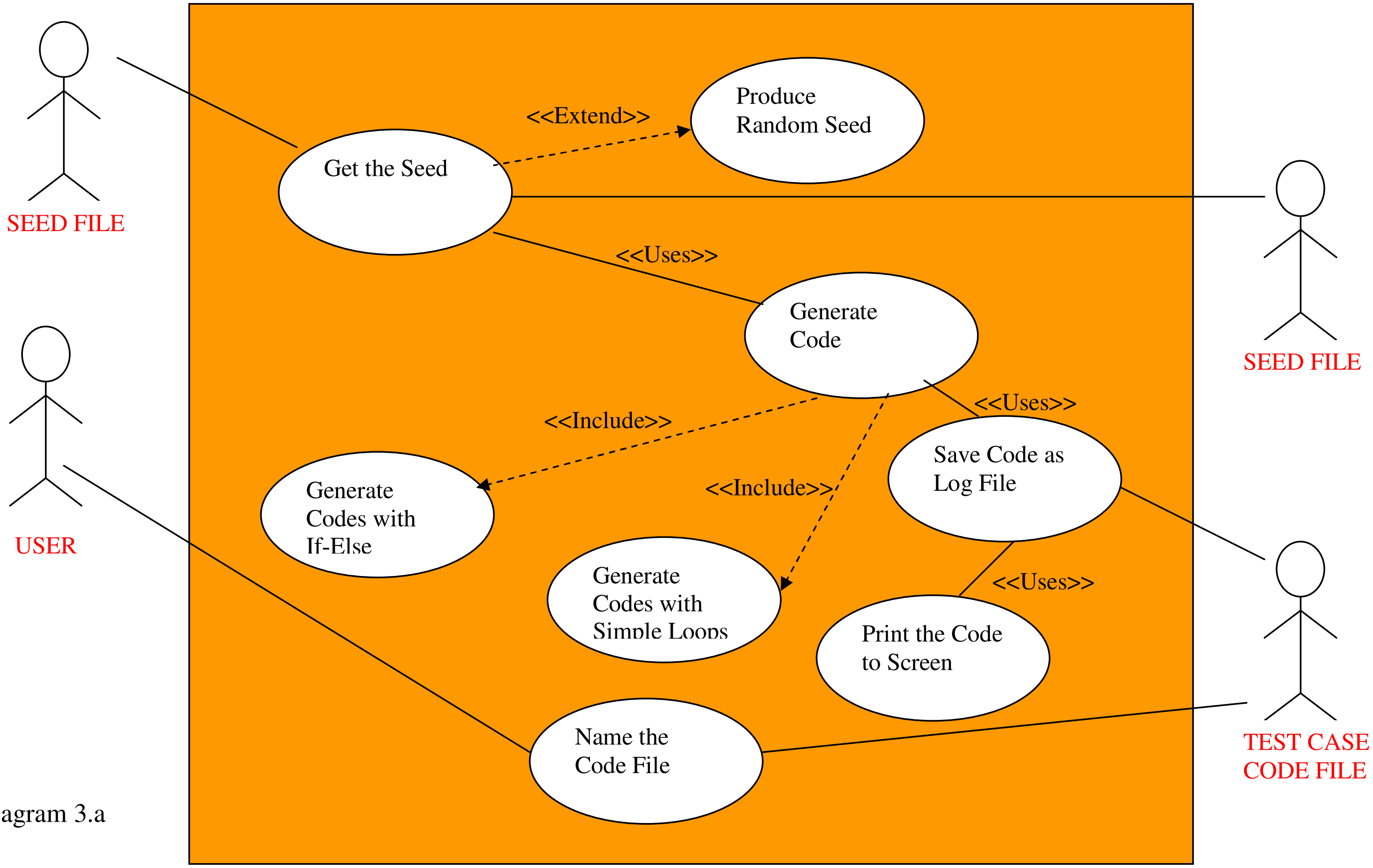


Diagram 3.a

2.1.2.Use Case Diagram of The Optimization Manager:

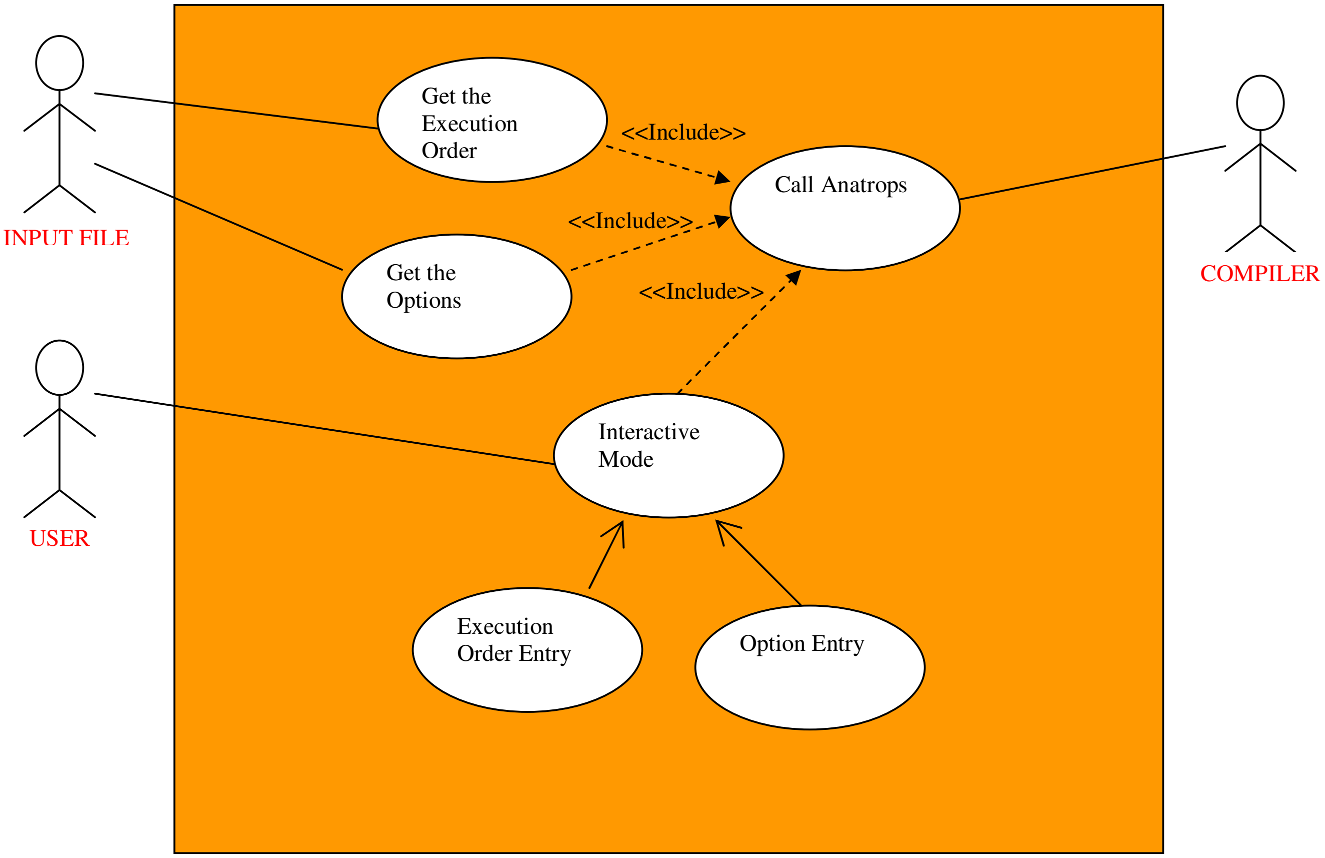


Diagram 3.b

2.1.3.TEST CASE GENERATOR EXPLANATORY TABLES:

Use Case	**Get the Seed**
Purpose	**Taking the Seed from the File or Itself**
Type	**Primary, Real**
Actors	**Seed File**
Description	**Program takes the seed from file or itself**
Cross-References	**Use Case: Produce Random Seed, Generate Code**

Use Case	Produce Random Seed
Purpose	Producing random seeds for the program
Type	Primary, Real
Actors	None
Description	When there is no need for seed file to produce seed it produces seed
Cross-References	Use Case: Get the Seed

Use Case	Generate Code
Purpose	Generating the test case codes
Type	Primary, Real
Actors	None
Description	After getting the seed, it randomly generates statements like if-else and simple loops
Cross-References	Use Case: Get the Seed, Generate Code with If-Else, Generate Code with Simple Loops, Save Code to a Log File

Use Case	Generate Code with If-Else
Purpose	Generating If-Else statements
Type	Primary, Real
Actors	None
Description	According to taken seed it generates If-Else statements
Cross-References	Use Case: Generate Code

Use Case	Generate Code with Simple Loops
Purpose	Generating Simple Loops
Type	Primary, Real
Actors	None
Description	According to taken seed generates simple Loops like for, while, etc.
Cross-References	Use Case: Generate Code

Use Case	Name the Code File
Purpose	Naming the code file as user wish
Type	Primary, Real
Actors	User, Test Case Code File
Description	Specifies the name of the produced code file
Cross-References	None

Use Case	Save Code to a Log File
Purpose	Saving the code to a log file
Type	Primary, Real
Actors	Test Case Code File
Description	Saves the code to a log file
Cross-References	Use Case: Print the Code to Screen

Use Case	**Print the Code to Screen**
Purpose	**Printing the code to screen**
Type	**Primary, Real**
Actors	**None**
Description	**Prints the generated code to screen**
Cross-References	**Use Case: Save Code to a Log File**

2.1.4.OPTIMIZATION MANAGER EXPLANATORY TABLES:

Use Case	**Get the Execution Order**
Purpose	**Specifying the Execution Order**
Type	**Primary, Real**
Actors	**Input File**
Description	**Reads the execution order of the optimizations from an external file**
Cross-References	**Use Case: Call Anatrops**

Use Case	**Get the Options**
Purpose	**Specifying the Options of anatrops**
Type	**Primary, Real**
Actors	**Input File**
Description	**Reads the options of the anatrops from an external file**
Cross-References	**Use Case: Call Anatrops**

Use Case	Interactive Mode
Purpose	Entering order and options by hand
Type	Primary, Real
Actors	User
Description	There is command for changing the mode to interactive. User can enter his own execution order and options in this mode
Cross-References	Use Case: Execution Order Entry, Option Entry

Use Case	Execution Order Entry
Purpose	Entering the execution order of optimizations by user
Type	Primary, Real
Actors	None
Description	User enters the execution orders as he wishes
Cross-References	Use Case: Interactive Mode

Use Case	Option Entry
Purpose	Entering the options of anatrops by user
Type	Primary, Real
Actors	None
Description	User enters the options of anatrops as he wishes
Cross-References	Use Case: Interactive Mode

Use Case	**Call Anatrops**
Purpose	**Calling the anatrops by specified order and options**
Type	**Primary, Real**
Actors	**Compiler**
Description	**It calls the anatrops and run it by specified order and options**
Cross-References	**Use Case: Get the Execution Order, Get the Options, Interactive Mode**

2.1.5.SCENARIOS:

a) Test Case Generator:

Scenario 1: Random seed is produced by the program and code is generated with default name after that saves it to a log file and prints it to screen

Scenario 2: A previous seed is taken from the seed file and code is generated with default name after that saves it to a log file and prints it to screen

Scenario 3: User gives a specific name to the code file

b) Optimization Manager:

Scenario 1: Optimization manager takes the execution order and options from an external input file and calls anatrops

Scenario 2: Optimization manager works at interactive mode user can enter execution order and options by hand and calls anatrops

3.Dataflows:

3.1.Test Case Generator:

Test case generator is an anatrop, so it will be implemented as an anatrop. It has some requirements to be implemented to fulfill its functionality. It will work only for the program scope. it will also be executed by a C file.

We should clear the module of the IR Program. It will be achieved by calling the reset() method of the IRProgram which removes all the modules from the program. We will use the IRBuilder class for implementation of the generator. The options set will be read and its functionalities will be added to the IR.The number of functions, statements and expressions will obey the rules which will be taken from the user with input.

If, for, while, do, assignments, expressions will be generated according to the input taken from user. The input will be specifications for the generation rules; like ranges and probabilities of statements, expressions; number of functions.

In the generator, we will implement the types as rich as possible, like integers, floats, strings, reals, bools, etc.. We will also define expressions defined in the scope of the framework like binary expressions (addition, multiplication, xor, arithmetic shifts, etc..), constants(integer, string...), locals, globals, etc...

We will implement a kind of Bayesian Networks that Mr. Karpat stated. With the use of Bayesian Network, we will implement different probabilities for statements.
For example let's consider the assignment statement; int a = 12.
Here we will look up in the Bayesian Network and create a type of integer according to the probability. Then in the given range we will assign a constant integer with its probability.

We will also define what can be the contents of the context.
At each compilation a single file is created. We will write the created random seed into a file. The seed will also be taken with user input.

In order to be able to write better looking generated code, we will extend the intermediate representation to a higher level or a HIR implemented by the framework in the following days will be used. After the generation of IR, we will dump this IR into C code. This generated C code will be saved to a log file and also it will be printed to the screen.

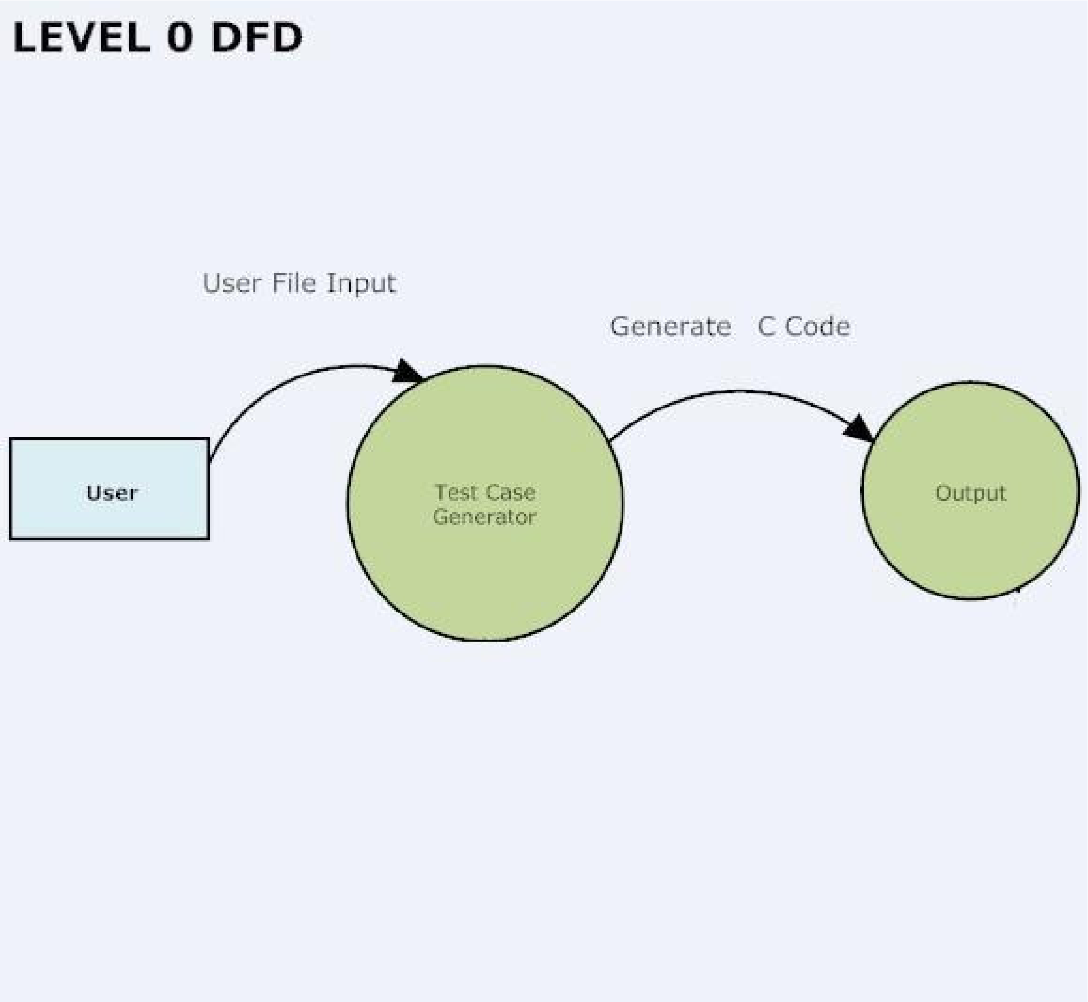


Diagram 3.a

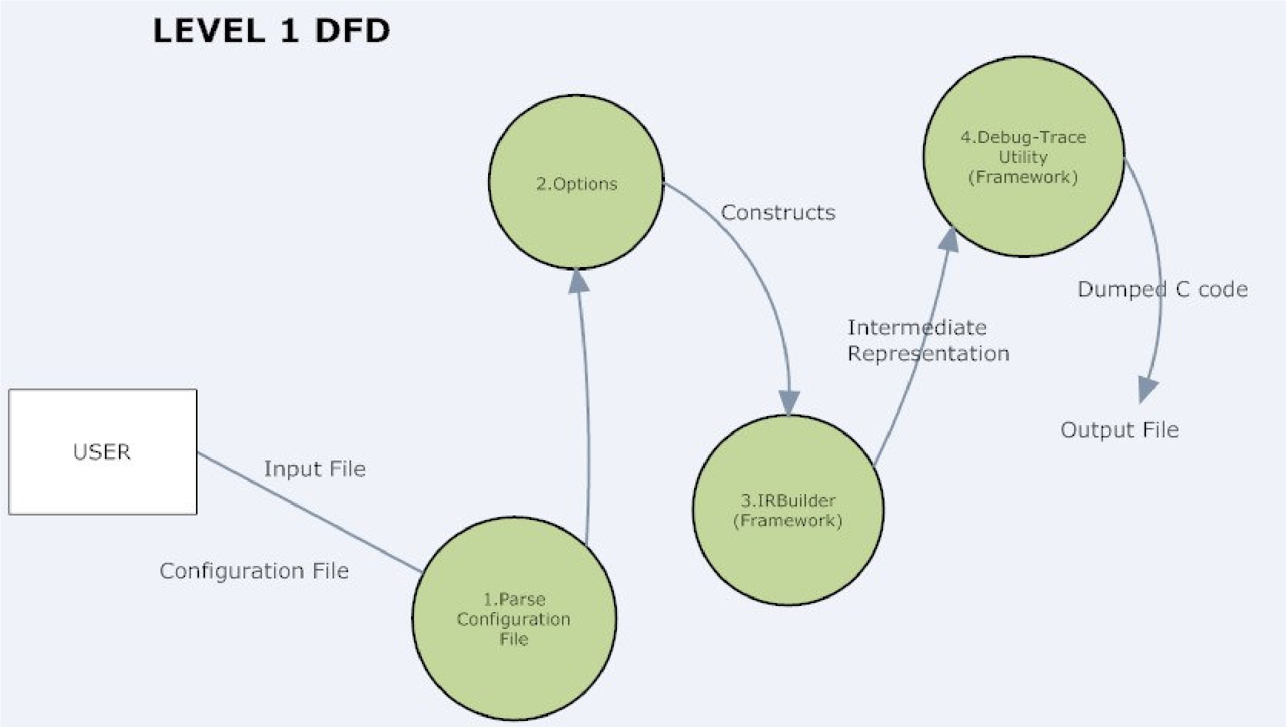


Diagram 3.b

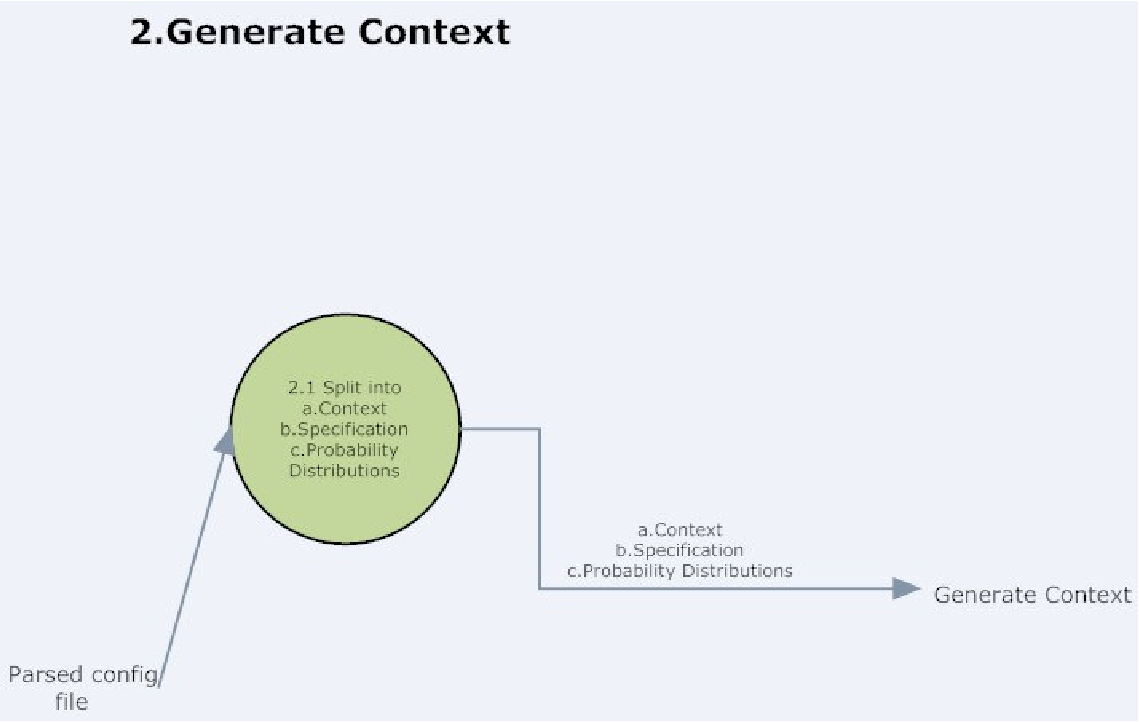


Diagram 3.c

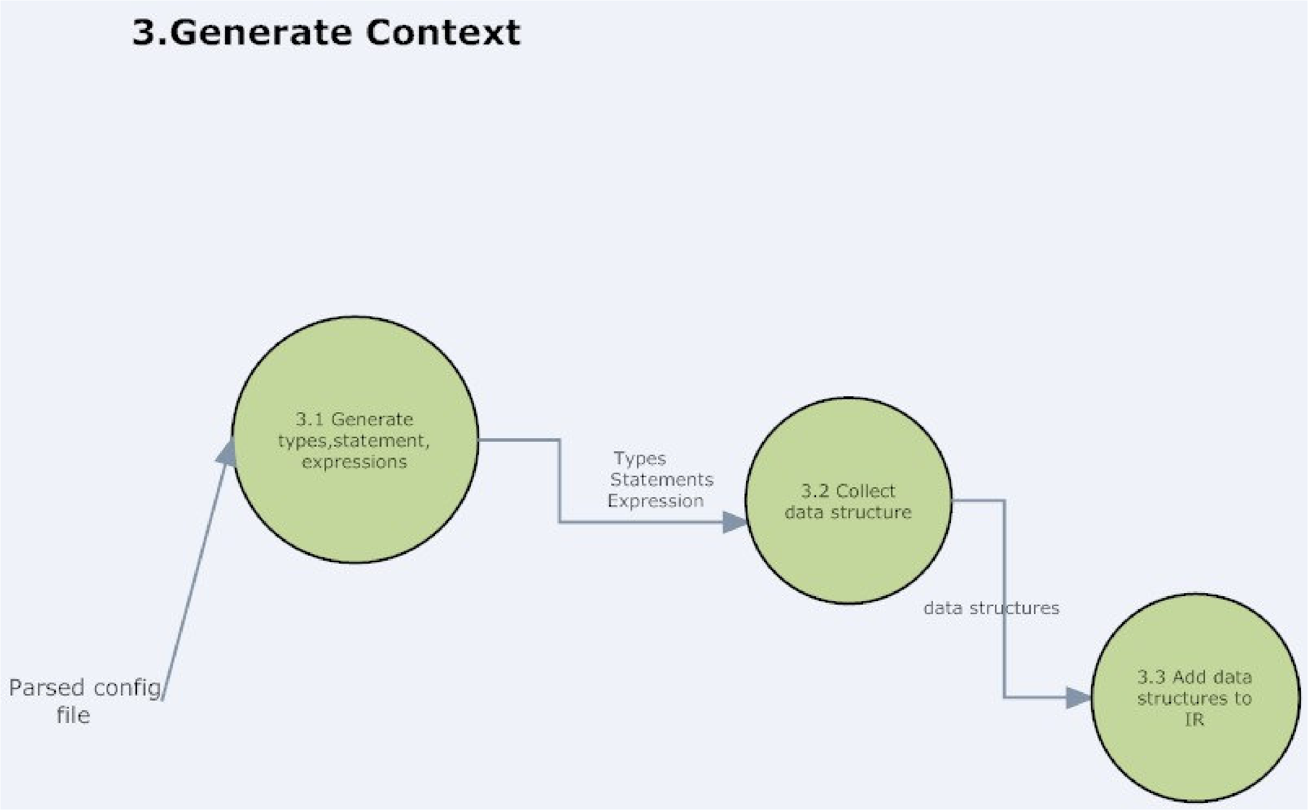


Diagram 3.d

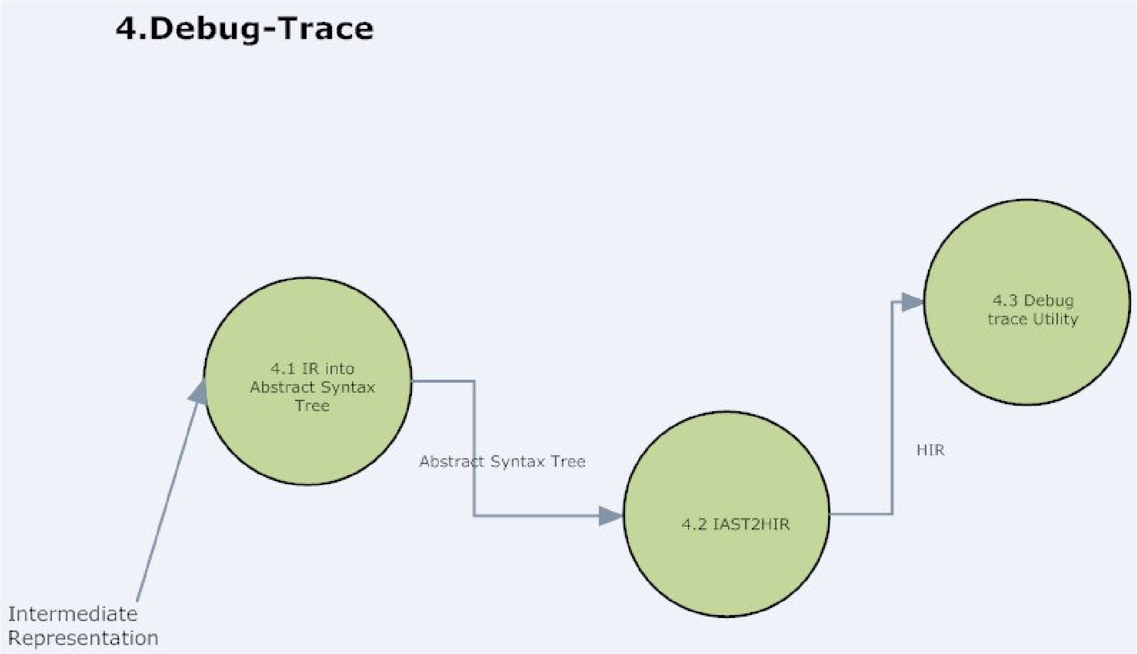


Diagram 3.e

Explanation of Vocabulary in Data Flow Diagrams for Test Case Generator

Name	Purpose	Description
Input File	İnvoke with any C file	Usually an empty files used
Configuration File	List of options of Test Case Generation	Contains options for Test Case Generation
Options	Hold configurations	Contains Context, Specifications and Probability Distirbutions
Probability Distributions	Number of constructs are created by these distributions	Provided probability of consturcts from user to Test Case Generator
Context	Provide data for Creation of Abstract Syntax Tree	List of available statements, expressions, loops, basic blocks,functions
IRBuilder	Utility of Framework	Intermediate Representation Functions
Debug/Trace Utility	Utility of Framework	Help to debug and trace the framework and also dumps ir to C Code
IAST2HIR	Convert Abstract Syntax Tree into HIR	Step for conversion from IR to HIR
HIR	Human Understandilibity for dumped Code	High Level Intermediate Representation

3.2.Optimization Manager:

Actually , the manager is an anatrop in the framework. Basicly , at compilation time an external file will be read or interactively user enters optimizations, scopes and options via standard input and so that the optimizations , which can be in some order or can be integrated to each other, will be chosen. There are two ways to do this. Firstly , user must write in a file with a specific way which optimizations can be an object .Secondly , user enters data with standard input and specify it . Then choosen optimizations will be added to be applied and then they will be dumped and executed.

The user can call the interactive mode via a code like:

```
ATOOptimizationManager.EnterInteractif();
```

Or with a command line input like :

```
-mode:Manager:"IActive”
```

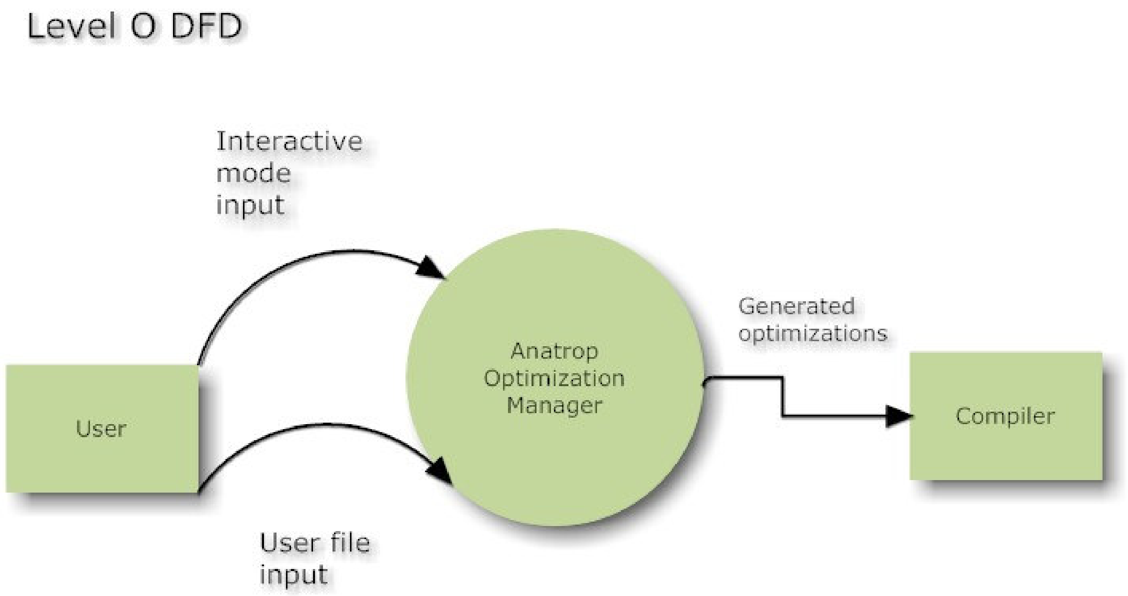


Diagram 3.f

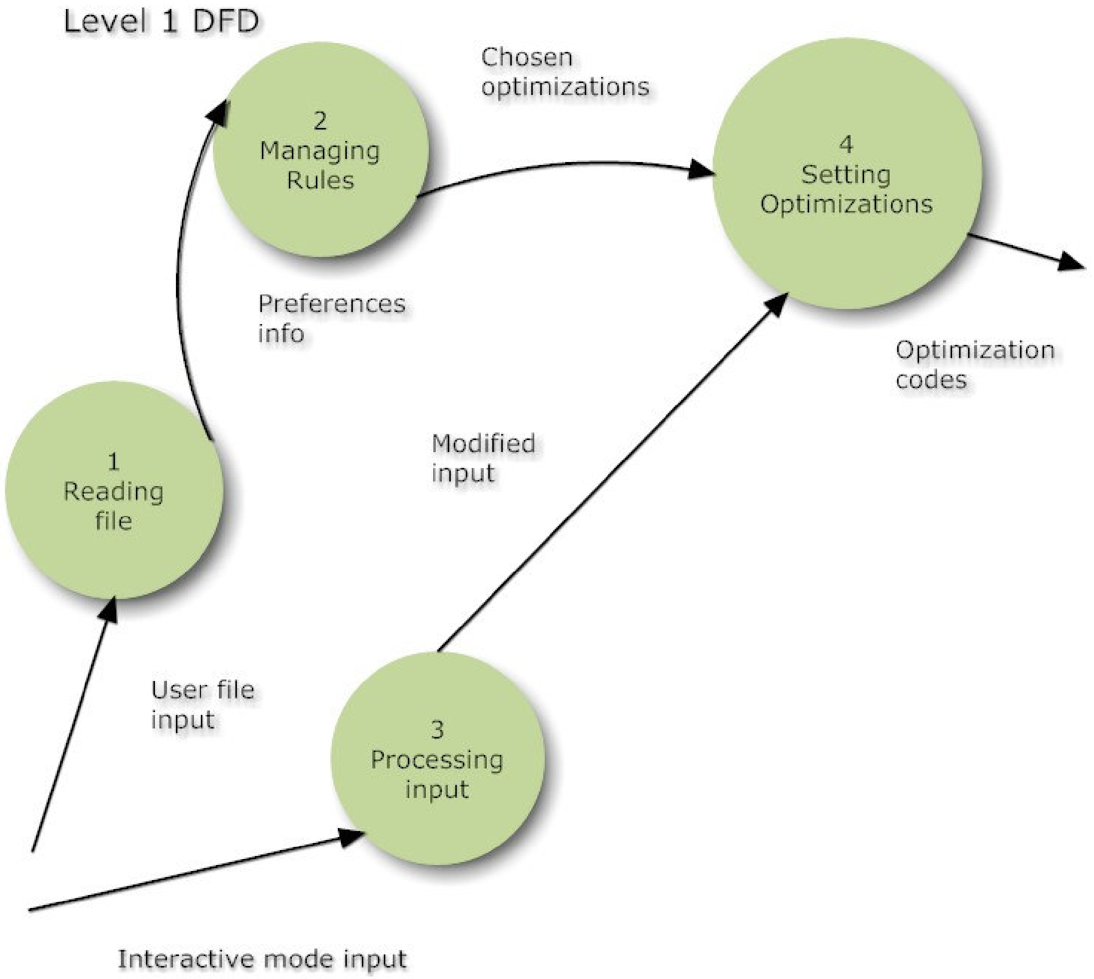


Diagram 3.g

Level 2 DFD

2 Managing Rules

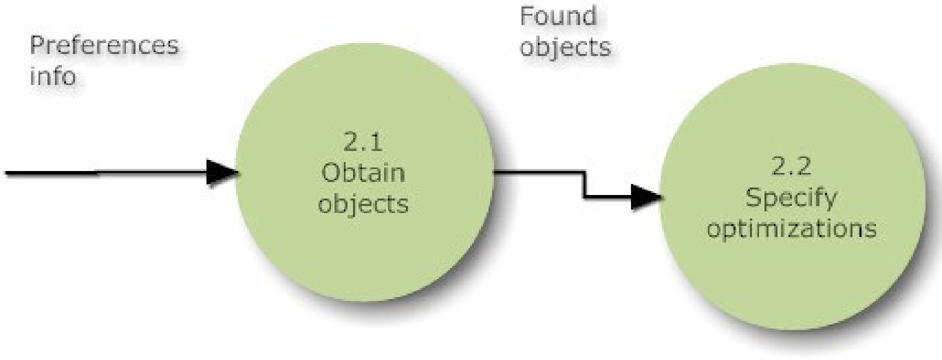


Diagram 3.h

3 Processing Input

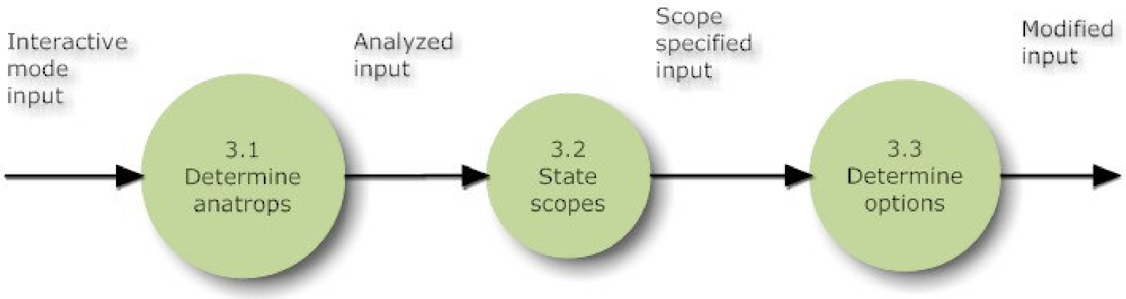


Diagram 3.i

4. Setting Optimizations

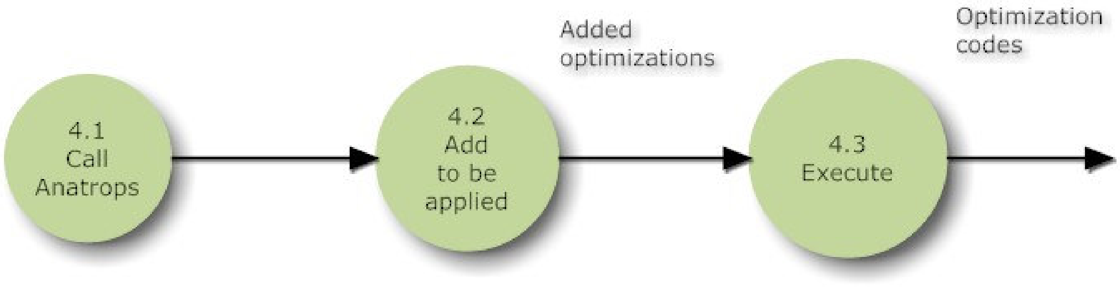


Diagram 3.j

Data Dictionary For Optimization Manager

Name	Obtain objects
Purpose	Getting objects
Type	Primary, Real
Actors	-
Description	Taking prefence objects from read file
Cross-References	Specify optimizations,reading file

Name	Specify optimizations
Purpose	Specifying optimizations
Type	Primary, Real
Actors	-
Description	Specifying the anatrops
Cross-References	Obtain objects,call anatrops

Name	Determine anatrops
Purpose	Determining the anatrops
Type	Primary, Real
Actors	user
Description	Determine each anatrop one by one
Cross-References	State scopes

Name	State scope
Purpose	Determining each scope
Type	Primary, Real
Actors	-
Description	Determining each scope one by one in an optimization
Cross-References	Determine anatrops,determine options

Name	Determine options
Purpose	Determinin the options
Type	Primary, Real
Actors	-
Description	Determining options one by one in a scope of optimization
Cross-References	State scope, call anatrop

Name	Call anatrops
Purpose	Calling anatrops from framework
Type	Primary, Real
Actors	-
Description	Calling and choosing optimizations
Cross-References	Add to be applied,determine optimizations,specify optimizations

Name	**Add to be applied**
Purpose	**Adding the optimizations to be applied**
Type	**Primary, Real**
Actors	**-**
Description	**Adding chosen optimizations with their scope and options to a list**
Cross-References	**Execute,call anatrops**

Name	**Execute**
Purpose	**Executing the optimizations**
Type	**Primary, Real**
Actors	**C code to be compiled**
Description	**Takin prefence objects from read file**
Cross-References	**Add to be applied**

Syntax Specification and Usage of Optimization Manager:

1) File mode:
There will be an external file which will be read.
Syntax will be like this:
a) For optimization calling:

Optimizationname(Scopetype(target))

There will be 3 selection:

Optimizationname(Module(modulename))
Optimizationname(Function(functionname))
Optimizationname(Program())

b)Array creating:

array arrayname

c)Array assign:

```
arrayname[number]=Scopetype
arrayname[number]=target
arrayname[number]=optimizationname
```

d)Iteration:

```
iterate(i to x)
{
//do something  i to x
}
```

Example:

```
iterate(i=0 to 10)
{
DeadCode(Function(array_a[i]
}
```

e)If :

```
if(statement)
{

}
```

e)Writing of optimization names:

constant folding == Constantfolding
basic block ordering == BlockOrdering
dead code elimination == DeadCode
local forward substitution == LForward
global forward substitution == GForward
strength reduction == SReduction
unreachable code elimination ==Unreachable
dead object elimination ==DeadObject
local common subexpression elimination == LSubexpression
global common subexpression elimination == GSubexpression
jump optimizations == Jump
if simplifications == IfSimp
tail merging == TailMerge
local copy propagation ==LCopy
global copy propagation ==GCopy
partial redundancy elimination ==Redundancy
procedure cloning and specialization ==Cloning
tail recursion ==TailRecursion

2)Interactive mode:
This mode takes the details from the user.

a)Activation:
- In filemodes’ file:
ato_interactive()

- dbccbin –intr

b)Usage:

In interactive mode, the program asks the optimization, scopes and options in a loop.
Like:

```
 Asks Optimizationname will be used? (Y/N)
-User entry: ( Y/N )
if(Y) then{ asks Scopename?):
-User entry:scopename
asks option:
-User entry:option
asks another scope for same optimization?
if (y) then go to scopename asking section
if (n) then go to optimization usage asking section}
if (N) then go to optimization usage asking section
```

Example:

```
Deadcode Elimination?
Y
Scope?
add()
Option?
optiona
Another Scope?
Y
Scope?
Min()
Option?
Nooption
Another Scope?
N
BasicBlock Ordering?
N
TAKE ALL&QUIT
```

4) Algorithms For Optimizations

1) Algebraic Simplifications

a) Addition with 0

```
Collect expressions of type addition into vector[]

foreach element in vector[]
BEGIN
    if( LeftExpression is Integer Constant )
    BEGIN
        if( LeftExpression == 0 && RightExpression is Integer Type )
        replace element with RightExpression

    END
    if( RightExpression is Integer Constant )
    BEGIN
        if( RightExpression == 0 && LeftExpression is Integer Type )
        replace element with LeftExpression

    END

END
```

Explanation
if there is an addition with 0; replace the expression with the Integer
ex:
    a+0; is replaced by a;
    0+b; is replaced by b;

b) Multiplication with 0

```
Collect expressions of type multiplication into vector[]

foreach element in vector[]
BEGIN
    if( LeftExpression is Integer Constant )
    BEGIN
        if( LeftExpression == 0 && RightExpression is Integer Type )
        replace element with 0
    END
    if( RightExpression is Integer Consant )
    BEGIN
        if( RightExpression == 0 && LeftExpression is Integer Type )
        replace element with 0
    END
```

Explanation
if there is a multiplication with 0 ; replace it directly with 0
ex:

a*0 or 0*b is replaced by 0

c) Multiplication with 1

```
Collect expressions of type multiplication into vector[]

        foreach element in vector[]
    BEGIN
        if( LeftExpression is Integer Constant )
        BEGIN
            if( LeftExpression == 1 && RightExpression is Integer Type )
            replace element with RightExpression
        END
        if( RightExpression is Integer Consant )
        BEGIN
            if( RightExpression == 1 && LeftExpression is Integer Type )
            replace element with LeftExpression
        END
```

Explanation
if there is a multiplication with 1 replace it with the Integer
ex:
a*1 can be replaced by a
1*b can be replaced by b

d) Division by 1

```
Collect expressions of type division into vector[]

foreach element in vector[]
BEGIN
    if( RightExpression is Integer Constant )
    BEGIN
        if( RightExpression == 1 && LeftExpression is Integer Type )
        replace element with Left Expression
    END
END
```

Explanation
replace division by 1's directly with the integer
ex:

a/1 is replaced by a

e) Subtraction with 0

```
          Collect expressions of type subtraction into vector[]

          foreach element in vector[]
     BEGIN
          if( RightExpression is Integer Constant )
          BEGIN
               if( RightExpression == 0 && LeftExpression is Integer Type )
               replace element with Left Expression
          END
     END
```

Explanation
replace subtraction by 0 directly with the integer
ex:
a-0 can be replaced by a

f) Multiplication with powers of 2

```
          Collect expressions of type multiplication into vector[]

          foreach element in vector[]
     BEGIN
          if( RightExpression is Integer Constant )
          BEGIN
               n = logbase(RightExpression,2) //check if 2 to the power n
               if( int(n)==float(n) )
               BEGIN
                    if(LeftExpression is Integer Type)
                    LeftExpression<<n        //leftshift rather than multiplication
               END
          END
     END
```

Explanation
Use Left shift rather than multiplication
ex:

```
a*8;

log(8,2) returns 3 //logarithm function
replaced with
a<<3; since multiplication is expensive
```

2) CONSTANT FOLDING

Scope = Statements

```
 Collect expresssions of type addition,
        multiplication,subtraction and division  into a vector[]

 For each element in vector[]
BEGIN //foreach
        IF ((LeftStatement is Integer Constant)
               && (RightStatement is Integer Constant))
        BEGIN //if
            IF ( Expression is Addition )
                      BEGIN //Add
                      a := LeftExpression + RightExpression
                      replace vector element with a
                      END   //Add
                IF ( Expression is Subtraction )
                      BEGIN //Sub
                      a := LeftExpression - RightExpression
                      replace vector element with a
                      END   //Sub
                IF ( Expression is Multiplication )
                      BEGIN //Mul
                      a := LeftExpression * RightExpression
                      replace vector element with a
                      END   //Mul
                IF ( Expression is Division )
                      BEGIN //Div
                      a := LeftExpression / RightExpression
                      replace vector element with a
                      END   //Div
        END //if
END //foreach
```

Explanation

Get Left and Right expressions of +,-,*,/ operators.
If both Left and Right expressions are integer constants
Do the operation for expressions.
replace the node with new value

ex: 3 + 8
3 and 8 are both integer constants
add them
new value 11 is replaced for 3+8

3) LOCAL COPY PROPAGATION

SCOPE = Basic Blocks

Collect list of all basic blocks into BBListVector[]

```
foreach element1 in BBListVector[]
BEGIN // foreach BBList
        Collect all statements of the element1(current  BB) into CurrStmtVector[]
        Collect all binary expressions of the element1 (current BB) into BinExprVector[]

        foreach element2 in CurrStmtVector[]
        BEGIN // foreach CurrStmt
                if ( element2 is Assign ) // if operation is an assignment operation
                        Add Statement to AvailableAssignmentsVector[]

                foreach element3 in BinExprVector[](
                BEGIN // foreach BinExpr
                        if(AvailableAssignmentsVector's LeftHand contains
                                                                element3.LeftExpression )
                        replace element3.LeftExpression with the its last occurance
                                                                        in
                                        AvailableAssignmentsVector[].RightHandside

                        if(AvailableAssignmentsVector's LeftHand contains
                                                                element3.RightExpression )
                        replace element3.RightExpression with the its last occurance
                                                                        in
                                        AvailableAssignmentsVector[].RightHandside
                END // foreach BinExpr
        END // foreach CurrStmt


        clear AvailableAssignmentsVector[]

END // foreach BBList
```

Explanation

For every basic block
look at each expression if any expressions are previously assigned
to the variables in the current basic block
if(yes) -> replace the expressions with the last occurrence of previous declaration

ex:

BB1

1) a=b;

2) c=a+4;

for 2) in BB1 ; a is previously defined so a can be replaced by b so c=b+4;

4) GLOBAL COPY PROPAGATION

Scope : Functions

Collect all basic blocks into BBListVector[]

```
  for(i=BBListVector.size() ; i>0 ; --i)
  BEGIN
       collect all statements in StmtVector[]
       for(j=StmtVector.size() ; j>0 ; --j)
       BEGIN
          if(StmtVector[j].RightHandSide is a binary expression)
          BEGIN
               Iterate backwards through Control Flow Graph
                 if(StmtVector[j].RightHandSide.leftchild
                              == Any assignment statement's lefthandside )
                       replace StmtVector[j].RightHandSide.leftchild
                        with found assingment statement's RightHandSide
                 if(StmtVector[j].RightHandSide.rightchild
                              == Any assignment statement's lefthandside )
                   replace StmtVector[j].RightHandSide.rightChild
                        with found assignment statement's RightHandSide
          END
        END
  END
```

Explanation : Through the basic blocks Iterate Backwards through Control Flow Graph in order to find if a expression is previously defined with a assignment statements.

5) TAIL RECURSION ELIMINATION

```
Collect all functions into FunctionsVector[]
foreach element1 in FunctionsVector[]
BEGIN
  collect all statements of element1 into StatementsVector[]
  foreach element2 in StatementsVector[]
  BEGIN
       if(element2 is a Procedure Call)
       BEGIN
         if( name(element2) == name(element1) )//look if it is a
         BEGIN                                            recursive call
              Insert a LABEL to the beginning of element1
              Get the arguments of element2
              Insert evaluation of statements for arguments of element2
              if( StackSize of element2 <= StackSize of element1 ) // prevent overflow
                      Equalize the stack sizes                                  of stack
                      replace element2 with a jump statement to LABEL
         END
       END
```

```
  END
END
```

Explanation
Replace the recursive function calls with a GOTO(jump) statement in order do decrease procedure calls.
ex:

```
void f(int n)                void f (int n)
{                                   {
a=b+8;                            LABEL
if(n<a)                           a=b+8;
f(n+1);         ---->>        if(n<a)
else                                 { n=n+1;
c=12;                                  JUMP LABEL
}                                    }
                                    else
                                    c=12;
                                    }
```

6) LOCAL COMMON SUBEXPRESSION ELIMINATION

```
Get Statements of Basic Block StmtVector[]

size:=stmtVector.size()

for(i:=size,i>=0;i--)
  BEGIN
    if(stmtVector[i].RightHandSide == Binary Expression)
        BEGIN
          for(j:=i;j>=0;j--)
            BEGIN
                if(stmtVector[i].RightHandSide.LeftExpression
                                    != stmtVector[j-1].RightHandSide
                                            ||
                  stmtVector[i].RightHandSide.RightExpression
                                    != stmtVector[j-1].RightHandSide)
                 BEGIN
                   if(stmtVector[i].RightHandSide == stmtVector[j-1].RightHandSide)
                        BEGIN
                                tmp element :=stmtVector[i].RightHandSide
                                Add tmp element before stmtVector[j-1]
                                stmtVector[i].RightHandSide:=tmp element
                                stmtVector[j].RightHandSide:=tmp element
                        END
                 END
            END
      END
   END
```

We get basic blocks statements(eg. x=a+b,y=3,..) into a Statement Vector. From the bottom of the basic block to the top of the basic block we implemented the algorithm. We get the last statement of the basic block. We look if the right hand-side of this statement is a binary expression(eg. a+b,a&&b,a<<2). If so then we enter in a loop that initially searchs the entire statement vector for the prior definiton statements of operands of the binary expression. If we found a matching definition,we cannot apply Local Common Subexpression to the given expression. Secondly,if there is no matching definition in the entire statement vector we look if there is a matching binary expression that we could eliminate. If found we eliminate this binary expression by setting its value to a temporary element.We insert this tmp definition before stmtVector[j-1].Then we reset the values of the binary expression that we eliminated by setting its value to tmp.

7) GLOBAL COMMON SUBEXPRESSION ELIMINATION

```
Collect all basic blocks into to BBVector[]

foreach element in BBVector[]
BEGIN
        Collect BBVector[].LeftHandSide into BBLHSVector[]
        Collect statements of Basic Block into BBStmtVector[]
        foreach statement in StmtVector[]
                  BEGIN
                        if(StmtVector[].LeftHandSide == BBLHSVector[])
                           BEGIN
                                 Collect BBVector[i] into stmtVector[]
                           END
                  END
END


Get Statements of Basic Block stmtVector[]

size:=stmtVector.size()

for(i:=size,i>=0;i--)
  BEGIN
    if(stmtVector[i].RightHandSide == Binary Expression)
        BEGIN
          for(j:=i;j>=0;j--)
            BEGIN
                if(stmtVector[i].RightHandSide.LeftExpression != stmtVector[j-
1].RightHandSide
                                                    ||
                  stmtVector[i].RightHandSide.RightExpression != stmtVector[j-
1].RightHandSide)
                 BEGIN
                   if(stmtVector[i].RightHandSide == stmtVector[j-1].RightHandSide)
                        BEGIN
                               tmp element :=stmtVector[i].RightHandSide
                               Add tmp element before stmtVector[j-1]
```

```
                        stmtVector[i].RightHandSide:=tmp element
                        stmtVector[j].RightHandSide:=tmp element
                    END
                END
            END
        END
    END
```

Different from Local CSE we in Global CSE we search for previous definition of each statement in all over basic blocks in the control flow graph.Then if we found such definitions I collect the basic blocks that which matching found into stmtVector. Then I local CSE from the basic blocks in the stmtVector.

8) Partial Redundancy Elimination

```
Collect Basic Blocks into BBVector[]



foreach element1 in BBVector[i]
BEGIN
      Collect expression into ExpVector[]
      if(BBVector[i].Previous.size()>1)
        BEGIN
            Collect BBVcctor[i].Previous in a PreviosVector[]
            foreach element in PreviousVEctor[]
              BEGIN
                  insert a new  bb after PreviousVector[i]
                  set new bb's previos as PreviosuVector[i]
                  set new bb's next as BBVector[i]
              END
        END
    calcute latest[] and used_out[],e_use[]
        //e_use[]:An expression is locally used in block b if it is computed at least once.
     //latest[]:An expression is in latest[b] that indicates that the last point the
     expression can be computed is at the beginning of block b.
        used_out[]: An expression is in used_out[],if it is globally used in basic block b,
then an evaluation of e
 at b will be used again along some path starting at b.

      foreach expression e in ExpVector[e]
        BEGIN
            create a temporary element t.
            if(e is in (latest[i] and used_out[i])
              BEGIN
                  insert t=e to the top of the BBVector[i]
              END
            if(e is in (e_used[i] and used_out[i])
              BEGIN
```

```
                replace e with t
            END
        END
END
```

We collect basic blocks into BBVector. Then we collect expression of each basic block into ExpVector. Then we found critial edges. Critial edge is a block with more than one predecessor. Then we insert an empty block along all edges which enter the critial edge. Then we found latest,used_out,e_use foreach basic block. Then for every expression in basic block we search for

reduncany. We created a temporary element to store e. Then we check if expression is the last point the expression can be computed is at the beginning of block and it is globally used in basic block b, then an evaluation of e at b will be used again along some path starting at b(latest and used_out).If so we insert t=e into top of the basic block we created. Then we check is expression is locally used in block b if it is computed at least once and it is globally used in basic block b, then an evaluation of e at b will be used again along some path starting at b (e_used and used_out).If so we replace original e with t

9) Tail Merging

Scope = Basic Block

```
Collect predecessors of blocks into predecs[].
IF (predecs[].size > 1)
BEGIN // if
    IF (laststatement == predecs[].lastNonContStatement)
    BEGIN // if
        Same = true;
        WHILE (predecs[].size > 0)
        BEGIN // while
            IF (predecs[].HasSuccessors == false)
            BEGIN // if
                Same = false;
                break; // while
            END // if
            ELSE IF (laststatement != predecs[].lastNonContStatement)
            BEGIN // if
                Same = false;
                break; //while
            END // if
            Predecs[].size --;
        END // while
        IF (same == true)
        BEGIN // if
            GetCopy (laststatement);
            Insert (laststatement as Beginning BB)
            WHILE (predecs[].size > 0)
            BEGIN // while
```

```
                    Remove (predecs[].lastNonContStatement);
                END // while
            END // if
        END // if
END // if
```

Explanation:
We scan backward through predecessors of blocks that have multiple predecessors looking for same sequences of instructions and we replace all but one such copy with a branch to the beginning of the remaining one.

10) Strength reduction

Scope = statements

Collect list of all expressions of type multiplication, division, modulos into MulVector[], DivVector[] and ModVector[]

```
BEGIN // strength reduction
    WHILE (MulVector[].size > 0)
    BEGIN //while
        IF (expression is multiplication by power of 2)
        BEGIN // if
            // here we have an expression like * $2^n$
            FOR (iterate as much as n's value)
                BEGIN // for
                    left_shift(a);
                END //for
        END // if
    END //while
    WHILE (DivVector[].size > 0)
    BEGIN // while
        IF(expression is division by power of 2)
        BEGIN // if
            // here we have an expression like a / $2^n$
            FOR(iterate as much as n's value)
                BEGIN // for
                    Right_shift(a);
                END // for
        END //if
    END //while
    WHILE (ModVector[].size > 0)
    BEGIN // while
        IF (expression is modulos by power of 2)
        BEGIN // if
            // here we have expression like a mod $2^n$
            a AND ($2^n$ -1);
        END //if
    END //while
```

```
END // strength reduction
```

Explanation:
We look for expressions if it has a cheaper version. If it is multiplication by power of 2 we replace this multiplication with shifting. Shifting left stands for multiplication by 2^n. If our expression is a division by power 2 we remove this calculation and put a shifting there too. Shifting right stands for division by 2^n. If our expression is modulo by power of 2 we replace this calculation with a bitwise AND. By doing this kind of things we reduce a few code but we got so much code speed.

e.g.

$2^n * a$	$a / 2^n$	$a \bmod 2^n$
	Becomes	
a << n	a >> n	a & (2^n - 1)

11) UNREACHABLE CODE ELIMINATION

SCOPE = Basic Blocks

Collect blocks into a vector[].

```
while(!again)
   BEGIN

         again:=false
         i:=2     //There should be at least two blocks.

         while (No of Blocks >= i)
            BEGIN
                  if No_Path(1,i)
                           BEGIN //true
                                    We delete the block sets its instruction number to 0,
                                    we decrease the number of block and set again to true.
                                    No of Instructions:=0
                                    No of Blocks--;
                                    Block[i]:=NULL;
                                    again:= true

                           END  //true
                  i++;//Increased i to search for other blocks

            END
```

```
   END



No_Path(1,i)
   BEGIN
         while(Block[i].Previous != NULL)
           BEGIN
                 if(Block[i].Previous == Block[i])
                   BEGIN
                         return value of the No_Path() set to TRUE;
                   END
                 else
                   BEGIN
                         return value of the No_Path() set to FALSE;
                   END
           END

   END
```

We get control flow graph. We get basic blocks into a vector.
Then we define a boolen which is initially false.
Then we iterate through the basic block vector and search for a for an empty path from entry block to basic blocks.
(We define a function called No_Path(1,i).
1 presents which is set as the entry block.
Then we search for all the basic block for No_Path i.
When No_Path returns true,means that we found an empty path which is unreachable block, we sets its instruction number to zero (we clear its instruction content),we decrease no of blocks and set again to true.
We search for an No_Path for all blocks by increasing i. Then functions iterates until again is true.

12) Jump optimizations

Scope = Basic Blocks

Collect all BB into a vector[]

```
BEGIN
    WHILE (not exit basic block)
    BEGIN // while
        IF (statement is jump)
        BEGIN //if
          IF (jump.target == labelstatement && label.nextStatement == anotherjump)
            BEGIN // if
                jump.target := anotherJump.target;
            END // if
            IF (jump.target == labelStatement && labelStatement ==
                                            jump.nextStatement)
            BEGIN // if
                //there is no need to jump
                DELETE (jump)
            END // if
        END // if
    END // while
END
```

Explanation:
We remove useless jumps in this optimization or remap their addresses for getting more efficiency. First of all we collect all basic blocks into a vector. Then until reaching the exit basic block we check whether the statement is a jump or not. If it is a jump we check it is target. If this jump goes to another jump we change first jump target statement. secondly, we check the jumps next statement if it is same as the target of the jump. It means there is an useless jump in here thus we delete this jump.

e.g.

```
JUMP X                  JUMP Y            JUMP X
...                     ...               label X                 label X
label X     →becomes    label X           ...         → becomes   ...
JUMP Y                  JUMP Y
...                     ...
label Y                 label Y
```

13) Dead object elimination

```
Scope = module
Collect both global and local objects in the code into GlobalObject[] and LocObject[]
BEGIN
    WHILE (GlobalObject[].size > 0)
    BEGIN // while
        IF (object.HasSuccessor() == true && object.HasInitial == true)
        BEGIN // if
            ObjectInUse[].insert(object)
        END // if
        GlobalObject[].size --;
    END // while
    WHILE (GlobalObject[].size > 0)
    BEGIN // while
        FOR (iterate as much as ObjectInUse[].size)
        BEGIN // for
            IF (GlobalObject[object] == ObjectInUse[object])
            BEGIN // if
                GlobalObject[object].HasUsed = true;
            END // if
        END // for
        IF (GlobalObject[object].HasUsed != true)
        BEGIN // if
            REMOVE (GlobalObject[object]);
        END // if
        GlobalObject[].size --;
    END // while
    WHILE (LocObject[].size > 0)
    BEGIN // while
        IF (object.HasSuccessor() == true && object.HasInitial == true)
        BEGIN // if
            LocObjectInUse[].insert(object)
        END // if
        LocObject[].size --;
    END // while
    WHILE (LocObject[].size > 0)
    BEGIN // while
        FOR (iterate as much as LocObjectInUse[].size)
        BEGIN // for
            IF (LocObject[object] == LocObjectInUse[object])
            BEGIN // if
                LocObject[object].HasUsed = true;
            END // if
        END // for
        IF (LocObject[object].HasUsed != true)
        BEGIN // if
            REMOVE (LocObject[object]);
        END // if
        LocObject[].size --;
```

```
    END // while
```

Explanation:
In this optimization we are looking all global and local objects and put them into a vector. First we check if it is used or not then we insert used ones into a vector. Then we compare those two vectors. Objects which are not in both are unused (dead) objects. We remove those objects with a remove () function. We used scope as module because we look for global objects. For locals function could be enough.

14) Local Forward Substitution:

```
foreach statement s in basicblock b1
    if (s is assigned -> object o1 OR o1 is assigned -> o2)
    Lookfor(copypropogation)
    endif
    else if (statement.lefhand is avaiable)
    assign it to an object o3
        if (o3 isnt global and avaiable)
        assign it as found
        endif
    foreach statement in bblock b2
    look on the all expressions in right hand statement with every depth
    put them in a vectora[]
    endfor
    collect the left expressions->vectorb[]
    foreach left expression l in vectorb[]
    if( l is used AND l is avaiable)
    take  from vectora[] , set and replace
    endif
    endfor
    endif
endfor
```

This procedure looks in a basic block for left expressions that can take forward substitution. Then it takes the righthandside of the statements. Lastly if there is availability it sets the right ones in lefts.

15) Global Forward Substitution:

```
foreach statement s in program p
    if (statement s is assigned -> object o1 OR o1 is assigned -> o2)
    Lookfor(copypropogation)
    endif
    else if (statement.lefhand is avaiable)
    assign it to an object o3
    if (o3 isnt global and avaiable)
    assign it as found
    endif
    foreach statement in bblock b
    look on the all expressions in right hand statement with every depth
    put them in a vectora[]
    endfor
    foreach left statement in program p
        foreach basic block b
        if(l is used in b)
        assign b in list l1
        endif
        endfor
        foreach basic block b1 in l1
        lookfor left expressions and collect them in vectorb[]
        endfor
        foreach left expression in vectorb[]
        take  from vectora[] , set and replace
        endfor
    endfor
endfor
```

This procedure is global version of what we did in local forward substitution. It looks in every basic block in a CFG. Then it makes lists of the basic blocks for each variable. Lastly it substitutes the righthandsides.

16) Basic Block Ordering:

```
if( CFG is not NULL)
foreach basicblock b in CFG
put b in list[]
endfor
foreach b1 in list[]
if( b1 is not an entry block OR not an exit block)
then    foreach statement s in b
        weight:= weight + calcuteweight(s)
        endfor
endif
endfor
order blocks by weight
linking them
create an exit block
```

This procedure looks in CFG and orders the basic blocks which were linked together by their weight.

17) Procedure Cloning:

There are 2 main steps for procedure cloning. Firstly, propogation and determining the maximum number of clones that can be created.
This is important, because cloning can result exponential program growth and increase of procedures in the program.After determining it, equivalent clones are merged whenever they produce the same effect on the optimization. The last phase is to applying cloning based on the decisions we obtained until the program growth reaches the limited size.

```
PROC Build_Supertrace(GRAPH,NODE)
foreach SUCC(NODE), S of basic block
        if (S is spanning tree ancestor)
        then do nothing //S is target of backedge and not member of this supertrace
        else if (S in supertrace AND not assigned yet)
                if (cloning is permitted ANDPRED_COUNT(S) == 1)
                then
                        if (S is a member of the same loop as NODE)
                        then SBHEAD(S) = SBHEAD(NODE)
                                if depth-first
                                then ENQUEUE_HEAD(S)
                                else if breadth-first
                                then ENQUEUE_TAIL(S)
                                endif
                        endif
                else if (basic block already in supertrace)
                NEWNODE = COPYNODE(S)
                SBHEAD(NEWNODE) = SBHEAD(NODE)
                else basic block must be in different supertrace
               if (cloning is permitted AND NODE is not a supertrace head)
                then NEWNODE = COPYNODE(S)
                        SBHEAD(NEWNODE) = SBHEAD(NODE)
```

```
        endif
    endif
    if (NEWNODE was created)
    then establish lexical links
    foreach SUCC(S), SS
    establish flowgraph links from NEWNODE to SS
    endfor
    add flowgraph link from NODE to NEWNODE
    remove flowgraph links from NODE to S
        if depth-first then
        ENQUEUE_HEAD(NEWNODE)
        elseif breadth-first
        then ENQUEUE_TAIL(NEWNODE)
        endif
    endif
    endfor
if (NEWNODE was created)
then RETURN TRUE
else RETURN FALSE
END Build_Supertrace
```

The function Build_Supertrace(),follows the successor links from NODE, adding basic blocks to the supertrace. Successors to the basic block that are spanning tree ancestors are targets of back edges or targets of an exit edge. If the successor of NODE is not a member of the same loop, the successor is a member of another supertrace.

```
PROC Supertrace(GRAPH)
CALL INIT_QUEUE()
LIMIT_CONDITION = maximum_size □block_count_limit □maximum_depth
NODE = flow graph entry
    while NODE != NULL do
    if (NODE is a supertrace head)
    then SBHEAD(NODE)=NODE
    ENQUEUE_HEAD(NODE)
        while queue is not empty do
        TNODE = DEQUEUE_HEAD()
        if (LIMIT_CONDITION AND Build_Supertrace(GRAPH,TNODE))
        then rebuild spanning tree for flow graph
        rebuild loops in flow graph
        endif
        endwhile
    endif
    NODE = GG_NEXT(NODE)
    endwhile
END Supertrace
```

This algorithm is the entry point for supertrace formation.Passing a flow graph to the function and set LIMIT_CONDITION to one of the defined limits. This function follows lexical links in the flow graph and enqueues basic blocks identified as supertrace heads. The inner while loop builds the supertrace by dequeuing a basic block and then calling Build_Supertrace(). Cloning basic blocks in Build_Supertrace() requires that we rebuild the flow graph spanning tree.

18) Dead Code Elimination

a) Same assignment case(very simple)

```
      Collect all assignment statements in a vector[]
      foreach element1 in vector[]
      BEGIN
            if( element1.lefthandside
                  == element1.righthandside )
            remove the element1 from vector
      END
```

Explanation

    Remove unnecessary assignment operations

ex:

    a=a; can be removed since has no effect

b) General Case

```
      Construct the DU and UD Chains for CFG in IRDUUDCHAINS[]
      Collect all the Basic Blocks in BBListVector[]
      foreach element1 in BBListVector
      BEGIN
            Collect all statements of current BB in StmtsVector[]
            foreach element2 in StmtsVector[]
            BEGIN
                  if(element2 is an Assignemt Statement)
                  if(Definitions can reach to the element2 via
IRDUUDCHAINS)
                  then remove element2 from StmtsVector[]
            END
      END
```

5.GENERAL SYNTAX SPECIFICATION:

5.1 PROJECT LANGUAGE:

Our Project is based on a framework which is coded with C++ programming language. That is why we have to use C++ as our project programming language. We will use framework while coding the optimizations but the other parts of the project will be handled by us. As a result of this , we will use the same language, as in the optimizations, with the other parts of the project. C++ will be our project language.

5.2 ANATROP CLASSES:

Every software project has its unique syntax definitions and guidelines to follow up but mostly software experts do not pay attention to this. Hellim project group will really try to obey the guideline and the syntax definitions while writing codes. It is obvious that a person, it does not matter he is a computer engineer or software expert, can have problems while checking out the codes if do not have a guideline to follow up; maybe only the author of the codes can understand it. We do not want such kind of things. As a result of our weekly meetings we decided to have standard methods for naming anatrops. It must be begin with a prefix like "*ato_*" and then it will continue with its original name which explains its functionality. Name must be unique. As an example we can show this;

For the dead code elimination optimization*: ato_deadcode.cpp* and *ato_deadcode.hpp*

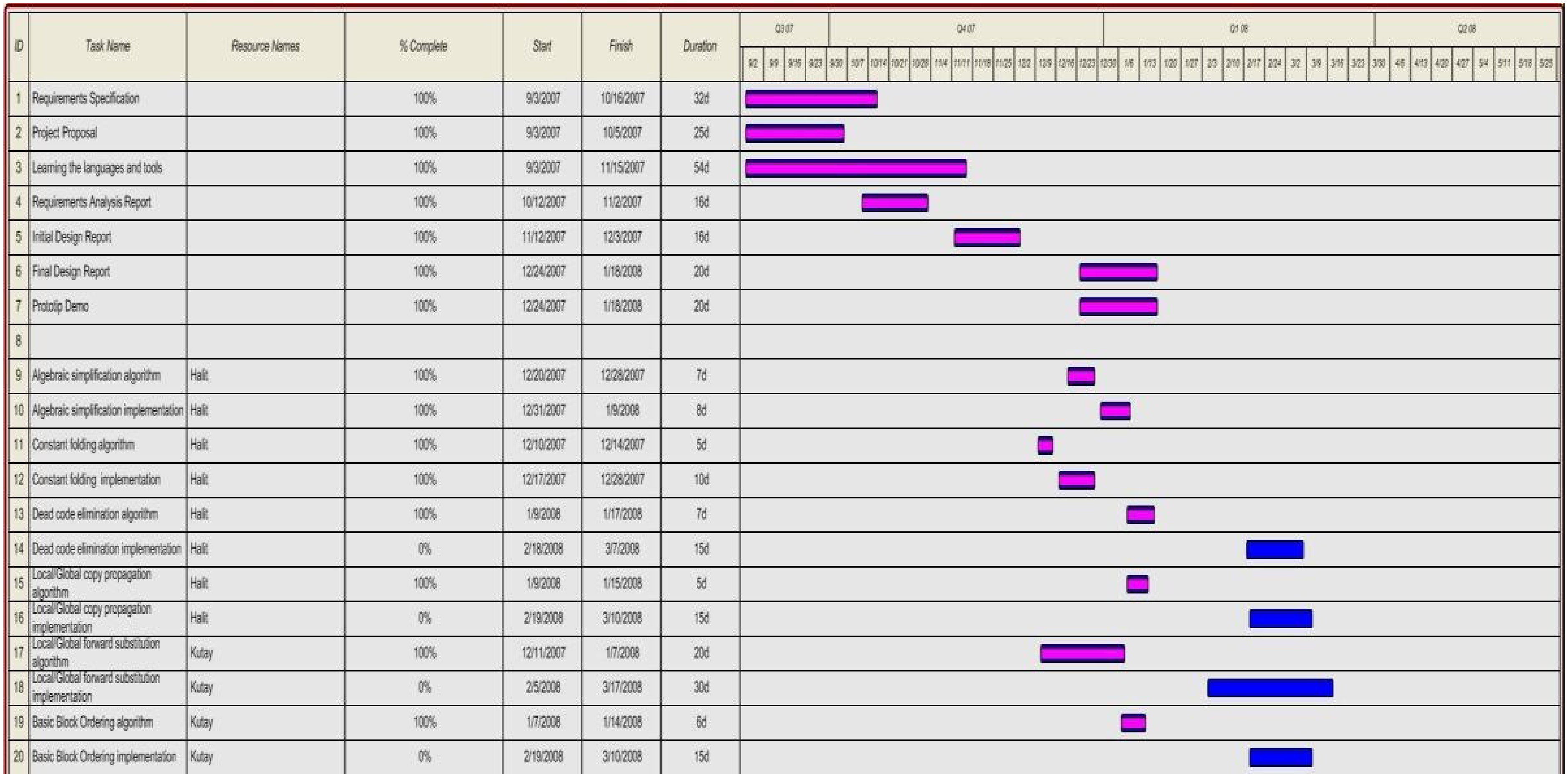
5.3 COMMENTS:

Increasing the understandability of the codes we write comments after some lines. We will use C++ comments to reduce the complexity.

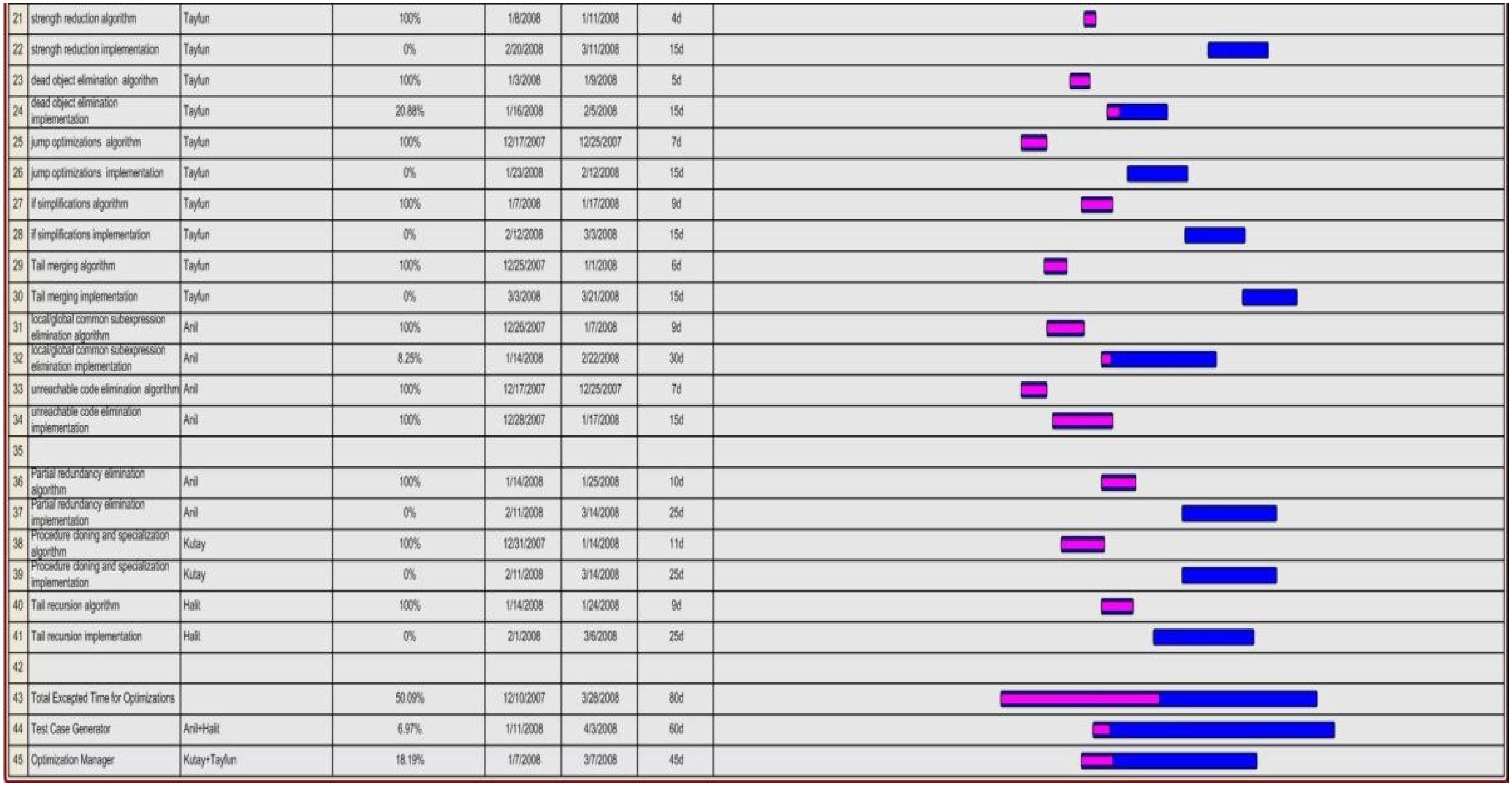
It will be like this;

// this function does that
// this call does that

6) Gannt Chart



ID	Task Name	Resource Names	% Complete	Start	Finish	Duration
1	Requirements Specification		100%	9/3/2007	10/16/2007	32d
2	Project Proposal		100%	9/3/2007	10/5/2007	25d
3	Learning the languages and tools		100%	9/3/2007	11/15/2007	54d
4	Requirements Analysis Report		100%	10/12/2007	11/2/2007	16d
5	Initial Design Report		100%	11/12/2007	12/3/2007	16d
6	Final Design Report		100%	12/24/2007	1/18/2008	20d
7	Prototip Demo		100%	12/24/2007	1/18/2008	20d
8						
9	Algebraic simplification algorithm	Halit	100%	12/20/2007	12/28/2007	7d
10	Algebraic simplification implementation	Halit	100%	12/31/2007	1/9/2008	8d
11	Constant folding algorithm	Halit	100%	12/10/2007	12/14/2007	5d
12	Constant folding implementation	Halit	100%	12/17/2007	12/28/2007	10d
13	Dead code elimination algorithm	Halit	100%	1/9/2008	1/17/2008	7d
14	Dead code elimination implementation	Halit	0%	2/18/2008	3/7/2008	15d
15	Local/Global copy propagation algorithm	Halit	100%	1/9/2008	1/15/2008	5d
16	Local/Global copy propagation implementation	Halit	0%	2/19/2008	3/10/2008	15d
17	Local/Global forward substitution algorithm	Kutay	100%	12/11/2007	1/7/2008	20d
18	Local/Global forward substitution implementation	Kutay	0%	2/5/2008	3/17/2008	30d
19	Basic Block Ordering algorithm	Kutay	100%	1/7/2008	1/14/2008	6d
20	Basic Block Ordering implementation	Kutay	0%	2/19/2008	3/10/2008	15d



21	strength reduction algorithm	Tayfun	100%	1/8/2008	1/11/2008	4d
22	strength reduction implementation	Tayfun	0%	2/20/2008	3/11/2008	15d
23	dead object elimination algorithm	Tayfun	100%	1/3/2008	1/9/2008	5d
24	dead object elimination implementation	Tayfun	20.88%	1/16/2008	2/5/2008	15d
25	jump optimizations algorithm	Tayfun	100%	12/17/2007	12/25/2007	7d
26	jump optimizations implementation	Tayfun	0%	1/23/2008	2/12/2008	15d
27	if simplifications algorithm	Tayfun	100%	1/7/2008	1/17/2008	9d
28	if simplifications implementation	Tayfun	0%	2/12/2008	3/3/2008	15d
29	Tail merging algorithm	Tayfun	100%	12/25/2007	1/1/2008	6d
30	Tail merging implementation	Tayfun	0%	3/3/2008	3/21/2008	15d
31	local/global common subexpression elimination algorithm	Anil	100%	12/26/2007	1/7/2008	9d
32	local/global common subexpression elimination implementation	Anil	8.25%	1/14/2008	2/22/2008	30d
33	unreachable code elimination algorithm	Anil	100%	12/17/2007	12/25/2007	7d
34	unreachable code elimination implementation	Anil	100%	12/28/2007	1/17/2008	15d
35						
36	Partial redundancy elimination algorithm	Anil	100%	1/14/2008	1/25/2008	10d
37	Partial redundancy elimination implementation	Anil	0%	2/11/2008	3/14/2008	25d
38	Procedure cloning and specialization algorithm	Kutay	100%	12/31/2007	1/14/2008	11d
39	Procedure cloning and specialization implementation	Kutay	0%	2/11/2008	3/14/2008	25d
40	Tail recursion algorithm	Halit	100%	1/14/2008	1/24/2008	9d
41	Tail recursion implementation	Halit	0%	2/1/2008	3/6/2008	25d
42						
43	Total Excepted Time for Optimizations		50.09%	12/10/2007	3/28/2008	80d
44	Test Case Generator	Anil+Halit	6.97%	1/11/2008	4/3/2008	60d
45	Optimization Manager	Kutay+Tayfun	18.19%	1/7/2008	3/7/2008	45d