MIDDLE EAST TECHNICAL UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

CENG 491

SENIOR DESIGN PROJECT

REQUIREMENT ANALYSIS 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

Tables of Content

1 Problem Definition and Project Scope

QuickC is a retarget able and optimizing compiler framework. It is simply designed to produce high quality, easily maintainable compilers in a short time.

QuickC is;

- Developer friendly
- Extensible
- Modular
- Flexible

Our aim is to implement optimizations for the framework also by the help of framework in order to decrease run time and to have efficient programs.

We will also implement a Test Case Generator for the framework in order to generate random C codes. We will try to develop the Generator as rich as possible to test our implementations.

We will also develop the Optimization Manager which reads an external file during compilation. Manager will specialize order and number of optimizations for given functions and modules.

2 LITERATURE SURVEY

2.1 What is a Compiler?

A compiler is a computer program (or set of programs) that translates text written in a programming language (*source code*) into another programming language (*object code*). Commonly the output has a form suitable for processing by other programs (e.g., a linker), but it may be a human-readable text.
Compiler design can be separated to three main approaches:

- Front end
- Middle end
- Back end

The front end is generally considered to be where syntactic and semantic processing takes place, along with translation to a lower level of representation (than source code).

The middle end is usually designed to perform optimizations on a form other than the source code or machine code. This source code/machine code independence is intended to enable generic optimizations to be shared between versions of the compiler supporting different languages and target processors.

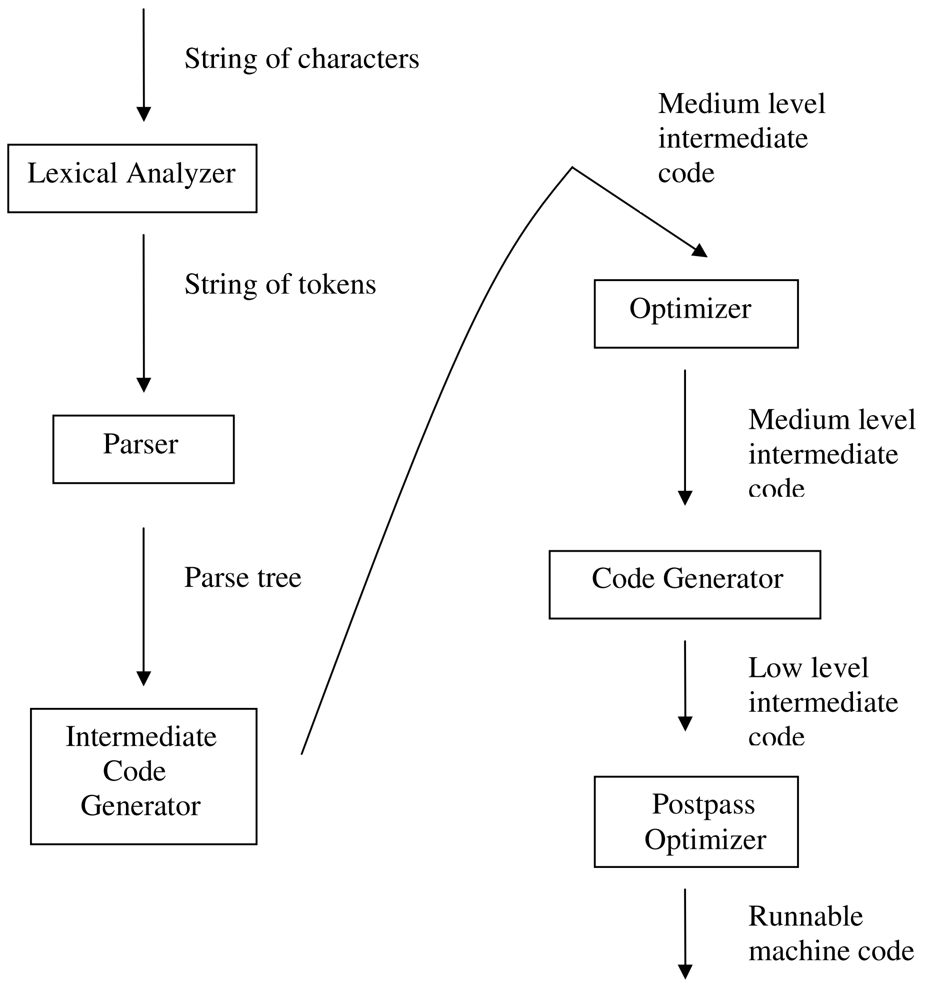
The back end takes the output from the middle. It may perform more analysis, transformations and optimizations that are for a particular computer. Then, it generates code for a particular processor and OS.

2.2 Back end

The main phases of the back end include the following:

1. Analysis: This is the gathering of program information from the intermediate representation derived from the input. Typical analyses are dataflow analysis to build use-define chains, dependence analysis, alias analysis, pointer analysis, etc. Accurate analysis is the basis for any compiler optimization.
2. Optimization: the intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms. Popular optimizations are inline expansion, dead code elimination, constant propagation, loop transformation , register allocation or even automatic parallelization.
3. Code generation

2.3 Structure of a Compiler



Muchnick, Steven S ., Advanced Compiler Design and Implementation, pp. 7-10

2.4 Some Compilers in Market

Intel C++ Compiler: It describes a group of C/C++ compilers from Intel. Compilers are available for Linux, Microsoft Windows and Mac OS X. Intel supports compilation for its IA-32, Intel 64, Itanium 2, and XScale processors. The Intel C++ Compiler for x86 and Intel 64 features an automatic vectorizer that can generate SSE, SSE2, and SSE3 SIMD instructions, the embedded variant for Intel Wireless MMX and MMX 2

Intel tunes its compilers to optimize for its hardware platforms to minimize stalls and to produce code that executes in the smallest number of cycles. The Intel C++ Compiler supports three separate high-level techniques for optimizing the compiled program Interprocedural Optimization (IPO), Profile-Guided Optimization (PGO) and High Level Optimizations (HLO).

Profile-Guided Optimization refers to a mode of optimization where the compiler is able to access data from a sample run of the program across a representative input set. The data would indicate which areas of the program are executed more frequently, and which areas are executed less frequently.

High Level Optimizations are optimizations performed on a version of the program that more closely represents the source code. High level optimizations include loop interchange, loop fusion, loop unrolling, loop distribution, data prefetch, and more. High level optimizations are usually very aggressive and may take considerable compilation time.

Interprocedural Optimization applies typical compiler optimizations (such as constant propagation) but using a broader scope that may include multiple procedures, multiple files, or the entire program.

GNU Compiler Collection: It is a set of compilers produced for various programming languages by the GNU Project. GCC is a key component of the GNU toolchain, and as well as being the official compiler of the GNU system, GCC has been adopted as the standard compiler most other modern Unix-like computer operating systems, including Linux, the BSD family and Mac OS X

Optimization: Optimization on trees does not generally fit into what most compiler developers would consider a front end task, as it is not language dependent and does not involve parsing. GCC developers have given this part of the compiler the somewhat contradictory name the "middle end." These optimizations include dead code elimination, partial redundancy elimination, global value numbering, sparse conditional constant propagation, and scalar replacement of aggregates. Array dependence based optimizations such as automatic vectorization are currently being developed.

The exact set of GCC optimizations varies from release to release as it develops, but includes the standard algorithms, such as loop optimization, jump threading, common subexpression elimination, instruction scheduling, and so forth.

Borland C/C++ Builder: It is a popular rapid application development (RAD) environment produced for writing programs in the C/C++ programming language.

Dev-C++: It is a free integrated development environment (IDE) distributed under the GNU General Public License for programming in C/C++. It is bundled with the open source MinGW compiler.

Microsoft Visual Studio: 2005 version includes

- Visual C# 2005 Express Edition

- Visual C++ 2005 Express Edition: The compile and build system feature, precompiled header files, "minimal rebuild" functionality and incremental link: these features significantly shorten turn-around time to edit, compile and link the program, especially for large software projects.

3 Technical Analysis

3.1 Importance of Code Optimization

The code can be much more efficient if compiler optimizes it.

The most important optimizations are those operate on loops, global register allocation and instruction scheduling among the others.

3.2 Information About Optimizations

3.2.1 Constant Folding

A technique, where constant sub expressions are evaluated at compile time.

This optimization should cause little penalty in compile time, but cause run time to decrease greatly.

Ex:

```
return 3+5 >> return 8
```

3.2.2 Dead Code Elimination

A technique, which removes unreachable code that is never executed.

Ex:

```
if(0)
a = 1;      >>    This code becomes
else               a = 2;
a = 2;
```

Also elimination of variables whose values are not used.

Ex:

```
x = y + 1;         y = 1;
y = 1;      >>     x = z * 2;
x = z * 2;
```

3.2.3 Unreachable Code Elimination

It is an optimization that eliminate code that cannot possibly executed

-Code guarded by conditional that is always false

-Code without any branches to it

-Code that cannot possibly be executed, regardless of the input data

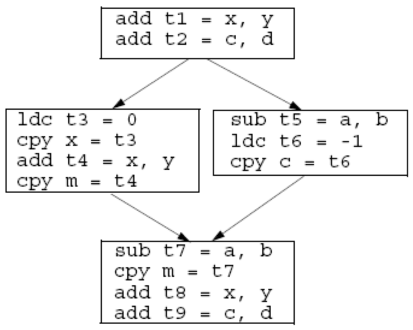
causes unreachable code.

3.2.4 Global Common Subexpression Elimination

- An occurrence of an expression is a *common subexpression* if there is another occurrence of the expression whose evaluation always precedes this one in execution order and if the operands of the expression remain unchanged between the two evaluations

- *CSE is usually done in two phases:*
  - *Local CSE is done within each basic block while the intermediate code for the basic block is constructed*
  - *Global CSE is done later across an entire flowgraph representing a procedure*

Global Common Subexpression



```
add t1 = x, y
add t2 = c, d
```

```
ldc t3 = 0
cpy x = t3
add t4 = x, y
cpy m = t4
```

```
sub t5 = a, b
ldc t6 = -1
cpy c = t6
```

```
sub t7 = a, b
cpy m = t7
add t8 = x, y
add t9 = c, d
```

- **Availability of an expression E at point P**
  - DEFINITION: Along every path to P in the flow graph:
    - E must be evaluated at least once
    - no variables in E redefined after the last evaluation
  - Observations: E may have different values on different paths

3.2.5 Global Forward Substitution

An optimization in which the result of an assignment can be propagated forward through a program. For example, the assignment B=C followed by A=B can be replaced by A=C if B is not used elsewhere in the program.

3.2.6 Basic Block Ordering

It is an optimization that improves instruction cache hit rates by placing frequently executed blocks together. So that the improvement of code locality leads to a reduced number of cache conflicts.

A basic block ordering is usually performed within the procedure boundaries and results on an effective reduction of misses , especially if the procedure maps inside the instruction cache.

a)Bottom-up: Basicly , it tries to reduce number of taken branches (conditional and unconditional)

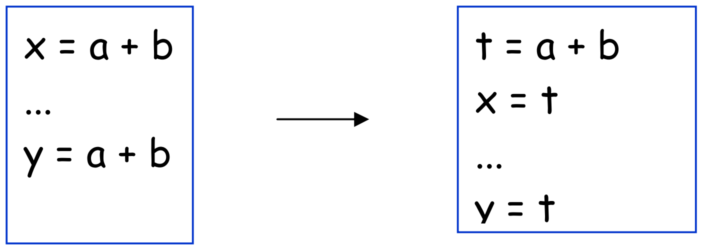
It can be named as forming chains of basic blocks. That chins should be placed as group as straight-line code in a bottom-up method. In this technique each basic block i considered as the head of tail of a chain. Then the merging of two different chains, in the order of large to small by looking at the arcs or block graphs.
Note: There is no must to be transitive and loops are allowed.

b)Top down: Basically , it tries to make conditional branches become forward conditionals.
The algorithm:

1) Place entry basic block.
2) A- Successor that is connected to the last places block by the largest count is selected.
   B- If all successors have already been selected, pick among unselected blocks with one largest connection to already selected.

3.2.7 Local Common Subexpression Elimination

The compiler detects multiple uses of the same expression or subexpression. The value is calculated only once and reused where possible superfluous expression calculations are eliminated from the code.



```
x = a + b
...
y = a + b
```

```
t = a + b
x = t
...
y = t
```

```
c = a + b
d = m * n
e = b + d
f = a + b
g = - b
h = b + a
a = j + a
k = m * n
j = b + d
a = - b
if m * n go to
L
```

```
t1 = a + b
c = t1
t2 = m * n
d = t2
t3 = b + d
e = t3
f = t1
g = -b
h = t1 /*
commutative */
a = j + a
k = t2
j = t3
```

3.2.8 Forward Substitution

Forward substitution is transformation that substitutes the right-hand side of an assignment statement for occurrences of the left-hand side variable which is especially useful in conjuction with symbolic data dependence.

In programs, temp variables are used to hold commonly used subexpressions or offsets in a local area. Holding a register too long, causes spills. So, these kind of substitutions might be useful.

Example:

```
a=b+2

c=b+2
d=a*b
```

```
t1=b+2
a=t1

c=t1
d=a*b
```

3.2.9 Jump Optimizations

In compiler theory, Jump threading is a compiler optimization. In this pass, conditional jumps in the code that branch to identical or inverse tests are detected, and can be "threaded" through a second conditional test. This is easily done in a single pass through the program, following acyclic chained jumps until you arrive at a fixed point.

3.2.10 Dead Object Elimination

In compiler theory, dead object elimination is a compiler optimization used to reduce program size by removing objects which does not affect the program. Dead object includes variables and functions that can never be executed (unreachable object), and code that only affects dead variables and functions, that is variables and functions that are irrelevant to the program.

3.2.11 Strength Reduction

In compiler theory, Strength reduction is a compiler optimization where a function of some systematically changing variable is calculated more efficiently by using previous values of the function. In a procedural programming language this would apply to an expression involving a loop variable and in a declarative language it would apply to the argument of a recursive function.

E.g.

```
          f x = ... (2**x) ... (f (x+1)) ...

becomes       f x = f' x (2**x)

where         f ' x z = ... z ... (f' (x+1) 2*z) ...
```

Here the expensive operation (2**x) has been replaced by the cheaper 2*z in the recursive function f'. This maintains the invariant that z = 2**x for any call to f'.

3.3 How is Code Optimized?

Source code is translated into a medium-level intermediate code and optimizations that are largely architecture-independent are done on it. Then the code is translated to a low-level form and further optimizations that are mostly architecture dependent are done on it. (mixed model of optimization)

Source code is translated to a low-level intermediate code and all optimizations are done on that dorm of code. (low-level model of optimization)

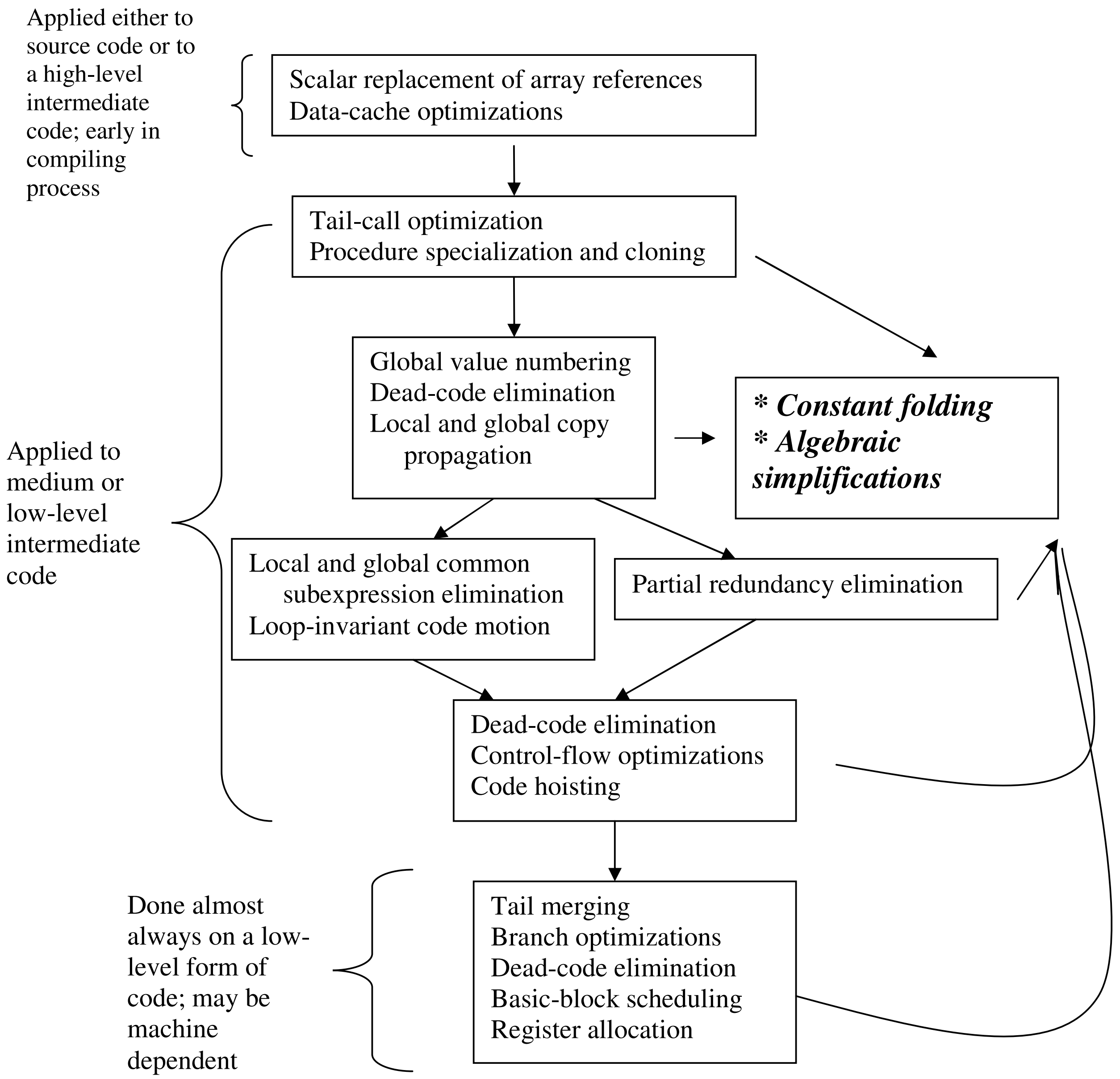
In both models optimizer analyzes and transforms the intermediate code to eliminate unused generality.

3.4 Advantages of Optimizations

Optimizations generally improve performance, although it is entirely possible that they may decrease it or make no difference for some (or even all) possible inputs to a given program.

In most cases, a particular optimization improves (or at least does not worsen) performance.

In general, we attempt to be as aggressive as possible in improving code, but never make it incorrect.


Applied either to source code or to a high-level intermediate code; early in compiling process
Scalar replacement of array references
Data-cache optimizations
Tail-call optimization
Procedure specialization and cloning
Global value numbering
Dead-code elimination
Local and global copy propagation
* Constant folding
* Algebraic simplifications
Applied to medium or low-level intermediate code
Local and global common subexpression elimination
Loop-invariant code motion
Partial redundancy elimination
Dead-code elimination
Control-flow optimizations
Code hoisting
Done almost always on a low-level form of code; may be machine dependent
Tail merging
Branch optimizations
Dead-code elimination
Basic-block scheduling
Register allocation

**** Constant folding and Algebraic simplifications can be applied many times, in any order***

Muchnick, Steven S . , Advanced Compiler Design and Implementation, pp. 325-327

<u>4 Specifications</u>

<u>4.1 Information about the Framework</u>

In this project, we are going to work on the framework QuickC, provided by CStar group.

First of all, QuickC is a framework which is developed on design by contract. It is Object-Oriented and developed with C++. Framework has its own intermediate representation in order to be used with different language front-ends.

Also, framework is documented with Doxygen. Doxygen is a documentation system which is used for creating reference manual from a set of documented source files. Detailed description of functions and classes can be presented with Doxygen. The advantage of Doxygen is, it makes documentation a must and helps with understanding the modules of the framework.

The framework has a debug/trace utility which helps with debugging and implementations and also supports Dwarf debugging information standard.

Framework has a core utility class, IRBuilder, Intermediate Representation Builder. Basic blocks are created automatically by IRBuilder while adding and removing statements. Also it updates Control Flow Graph. This makes easier development of optimizations and transformations.

The first task of the framework is to translate front-end abstract syntax tree in to the intermediate representation which helps with to apply transformations in any order and amount. Intermediate representation is low level enough to represent language constructs of different languages.

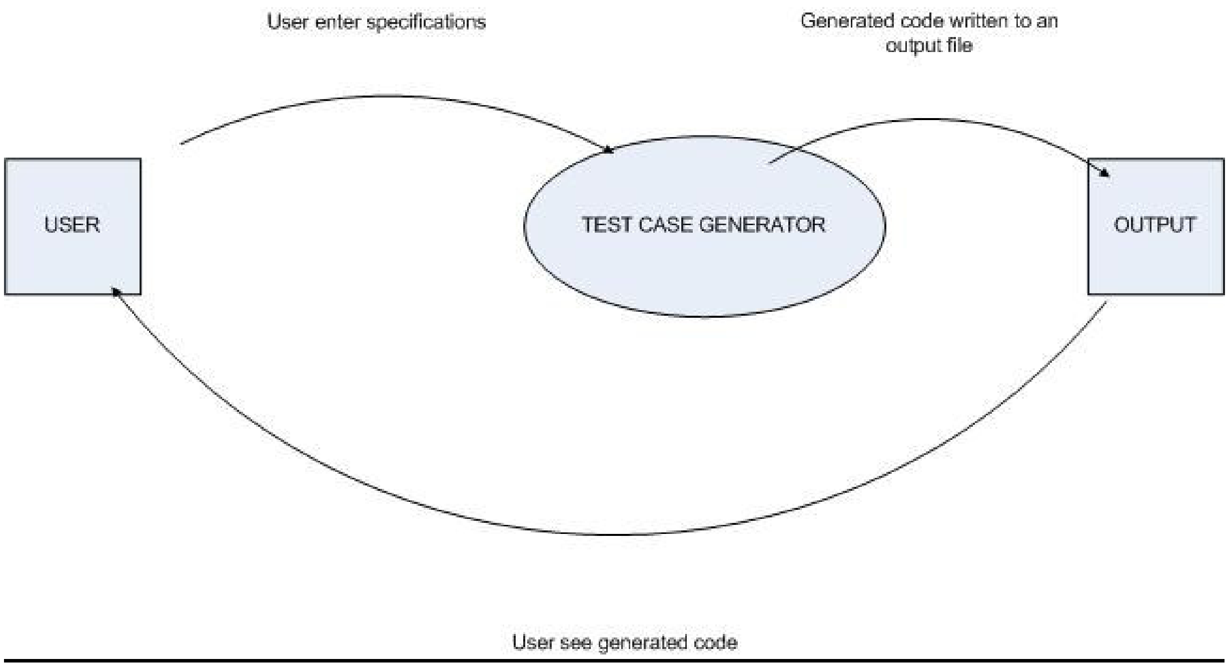
$$\begin{tikzcd}
\textbf{IRProgram} \arrow[d] & \\
\textbf{IRModule} \arrow[d] & \text{Includes unordered list of modules} \\
\textbf{IRFunction} \arrow[d] & \text{Includes unordered list of functions} \\
\textbf{CFG} \arrow[d] & \text{Every function has a corresponding Control Flow Graph. One entry and one exit basic block} \\
\textbf{IRBB} \arrow[d] & \text{Directed graph data structure of basic blocks.} \\
\textbf{IRStmt} & \text{Ordered list of statements. Statements may have zero or more expression trees depending on the type of statement.}
\end{tikzcd}$$

Analysis, transformation and optimization of intermediate representation is done by ANATROPS. They can be called in any amount and order. You can call an ANATROP on its set of valid scope like on basic block, a function, a module or a program.

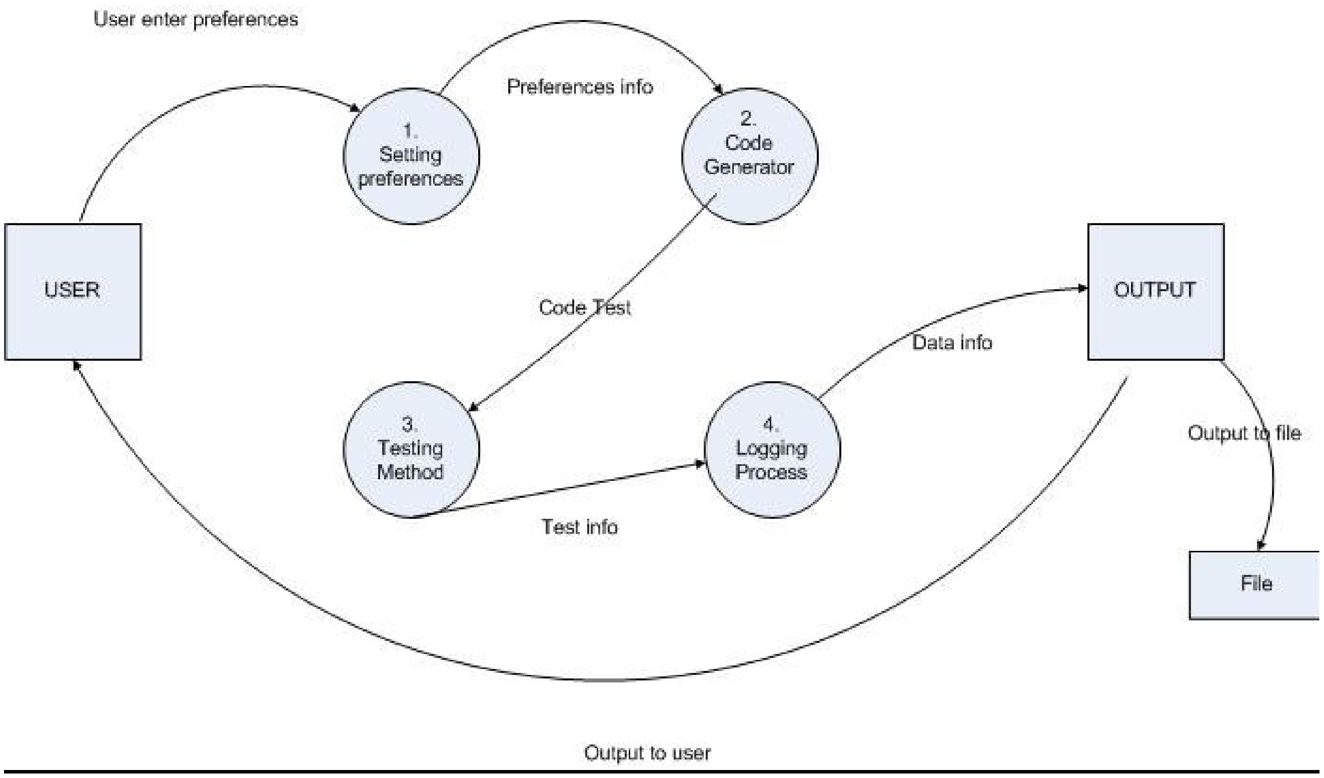
4.2 Test Case Generator

Test case generator is a tool which we will generate random C programs by using the framework. As we will see in the data flow diagrams user enter preferences for the program that want to generate. User will determine the percentages of the constructs. We will generate the program with a code generation algorithm. Then the program will the tested and the result will be passed to the logging process. This log and code will pass to the output process and code will be saved to a file and it will be printed to the screen.

Level 0 DFD


User enter specifications
Generated code written to an output file
USER
TEST CASE GENERATOR
OUTPUT
User see generated code

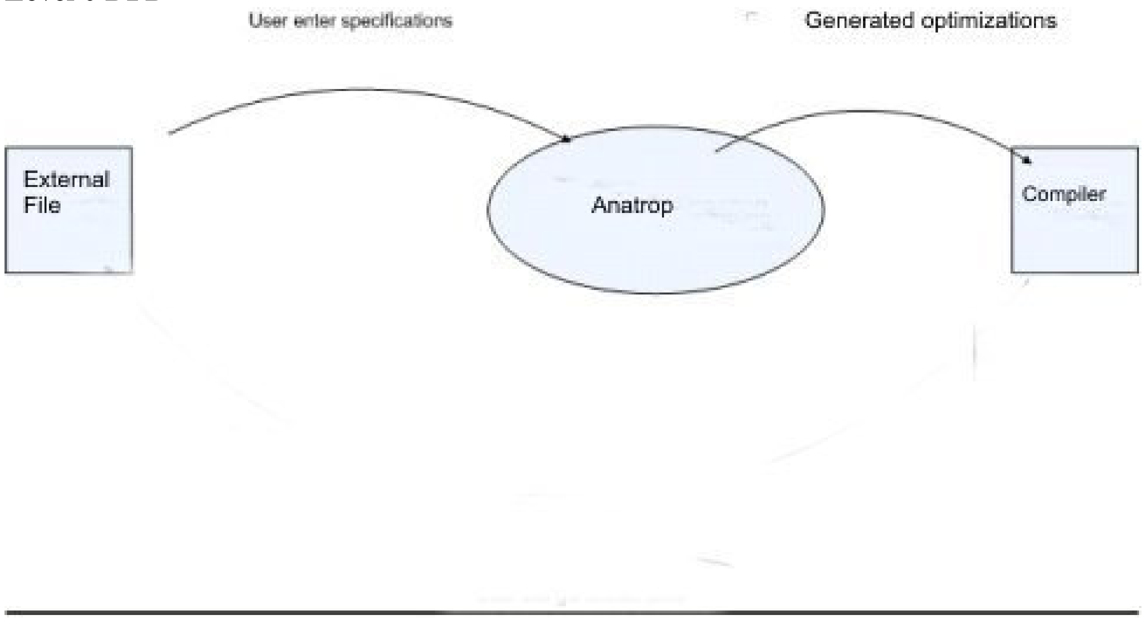
Level 1 DFD



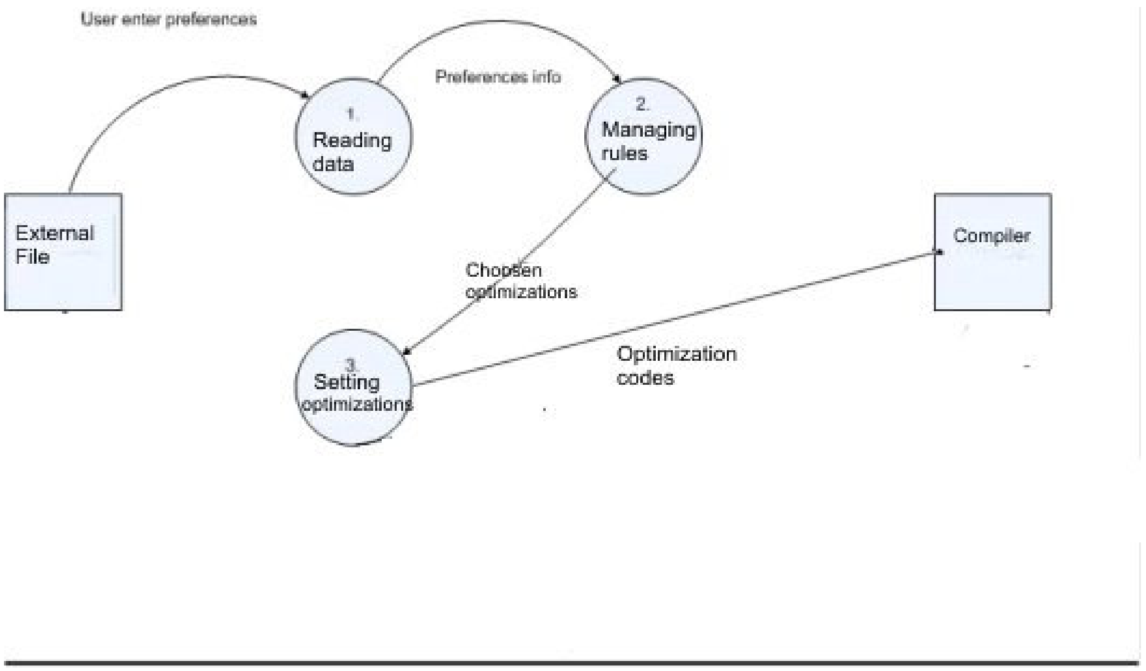
4.3 Optimization Manager

Optimization manager is an effective and advanced tool for managing the optimizations that will be used in compile time of a program. This manager is an assistant for advanced users. Some optimizations can be useful and some can affect badly to the program. That situation changes because of the characteristic of codes and used functions in program.So the user will specify the optimizations which are chosen to be useful.
Workflow of the optimization manager is very simple. Actually , the manager is an anatrop in the framework. By reading an external file , which consists of a program that sets execution order and options , the anatrop will set the optimizations.

Level 0 DFD



Level 1 DFD



<u>5 Project Model:</u>

In our project we will make compiler optimizations. We have a ready framework and we will use it for producing the code. Requirements are clear but can change in the future, additional requirements can show up. At the beginning we will do must-optimizations which are given by company.

Optimizations: Constant folding, basic block ordering, dead code elimination, local/global forward substitution, strength reduction, unreachable code elimination, dead object elimination, local/global copy propagation, local/global common subexpression elimination, jump optimizations, if simplifications, tail merging. Team members shared those optimizations fairly. There will be bonus optimizations. In addition to optimizations we are going to produce a test case generator and an optimization manager. Test case generator will produce random c codes, around 1000 – 2000 lines. We are going to test our optimizations on these codes. It must be as rich as possible. Optimization manager will be capable of reading an external file during compilation. This file will contain the execution order of the optimizations. Our team has 4 members, so we are a small project group and we have chosen our project model by thinking of this. XP Model is the best for the small groups like us. We got the requirements from the company as clear as possible but we know there will be additions to those requirements. We will start the project by following XP model properties.

Requirements are shaped then designing; coding, testing and releasing will come. This cycle will be processed several times. Between each cycle we will have feedback from the company. We must be flexible about the applying style the company can make lots of changes according to our works. We will use simplest design ever. This makes our job clear. We have divided the team. We have made two pairs. Those pairs will work together in their own area.

Products will be integrated at the end. Every member of the team will use the same framework and same language. We are going to produce our codes in c++. Time by time we will make time estimations. Those estimations will show us that which part of the project is done and what we have ahead. Our friendship will ease the communication. Our continuous connection will help us for informing each other about what we are doing. Writing the code, testing and maintenance will be our heavy part of the job. In addition to this we are going to make advanced documentation. This is not common for XP Model but for the course professor and assistant it is important.

6 Project Schedule

Project Task Set

Framework Activities

- Initial Design √
- Design
- Programming
- Testing

Task Set

- Requirements specification √
- Learning the languages and tools √
- Optimizations
- Test case generator
- Optimization manager
- Testing

List of deliverables

Documentation

- Project Proposal √
- Requirement analysis report
- Initial Design Report
- Final design report

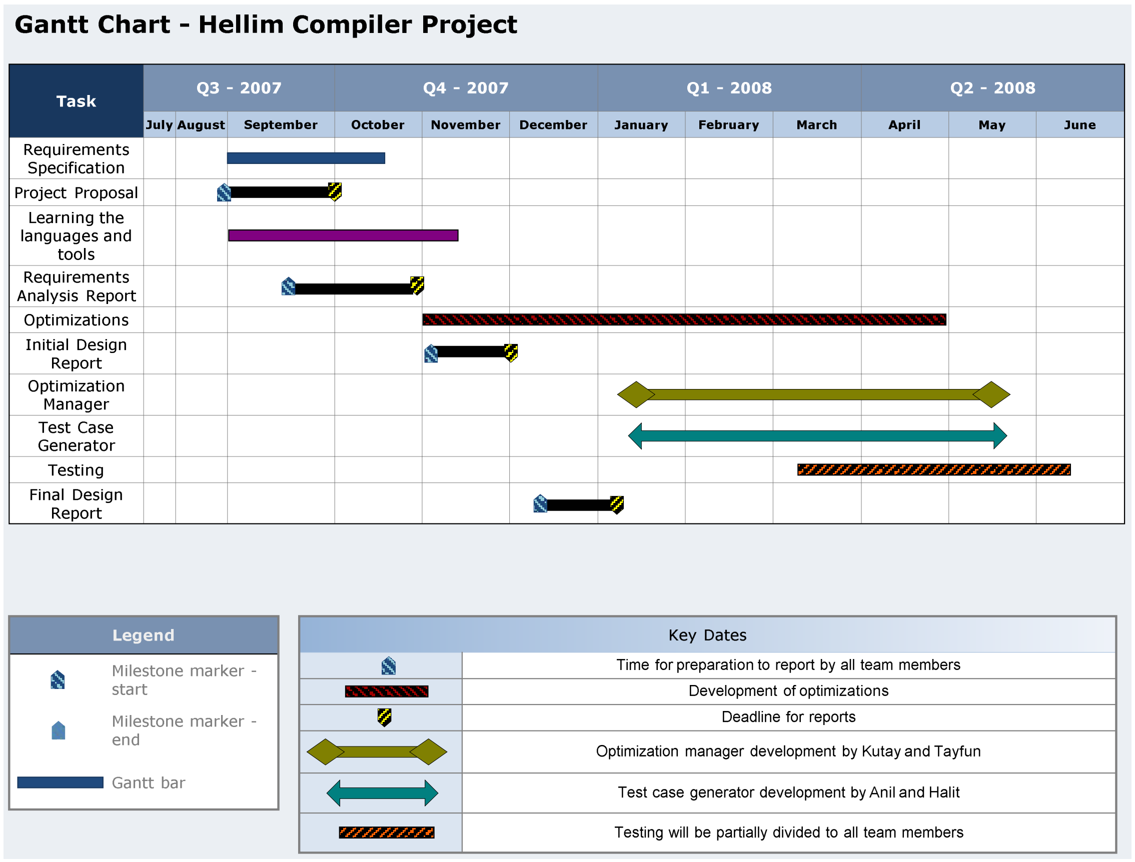
Functional Decomposition of these tasks

Requirements Specification
    Internet Search √
Learning the languages and tools
    Determining the proper language √
    Determining the tutorials and manuals √
    Studying the tutorials √
Optimizations
    Learning how to use optimization
    Implementation
Test case generator
    Test case generator initial design

  - Test case generator initial implementation
  - Test case generator revised design
  - Test case generator complete implementation
- Optimization manager
  - Optimization manager initial design
  - Optimization manager initial implementation
  - Optimization manager revised design
  - Optimization manager complete implementation

- Testing
  - Test of optimizations with benchmark tools
  - Comparing test results with GCC
  - Testing the test case generator
  - Testing the optimization manager

Gannt Chart



Key Dates	
▨	Time for preparation to report by all team members
▬	Development of optimizations
▨	Deadline for reports
◆—◆	Optimization manager development by Kutay and Tayfun
⟷	Test case generator development by Anil and Halit
▬	Testing will be partially divided to all team members

7 Risk Management Plan

Unavoidably, risks may occur during various times of a software project. There should be a risk management to reduce the damage of risks which may occur and for safety of project. So a risk analysis is a must.

Team:

-Misunderstanding the responsibilities: Each member has to recognize his responsibility. Otherwise the work will be done again unnecessarily and precious time will be wasted.

-Lack of aims and standards: Each part of the project should have an aim and done with some standards. Otherwise quality of job will be reduced.

-Unavailability of member(s): Because of exams or assignments of other courses, some members become unavailable. This condition can reduce productivity.

Project:

-Lack of time: Time is a precious factor for this project. Unnecessary jobs or not obeying the schedule will reduce time.

-Changes of requirements: If the requirements are not specified well, later on there can be major changes which slow down the project.

-Design and implementation: Design and coding phase of project may become harder due to some complicated algorithms.

-Technology: Because of framework's complicated structure, sometimes risks may occur.

Risk Table:

Risk	**Probability**	**Impact**
Misunderstanding the responsibilites	%10	Negligible
Lack of aims and standards	%30	Critical
Unavailability of member(s)	%30	Critical
Lack of time	%20	Negligible
Changes of requirements	%15	Negligible
Design and implementation	%25	Critical
Technology	%20	Marginal