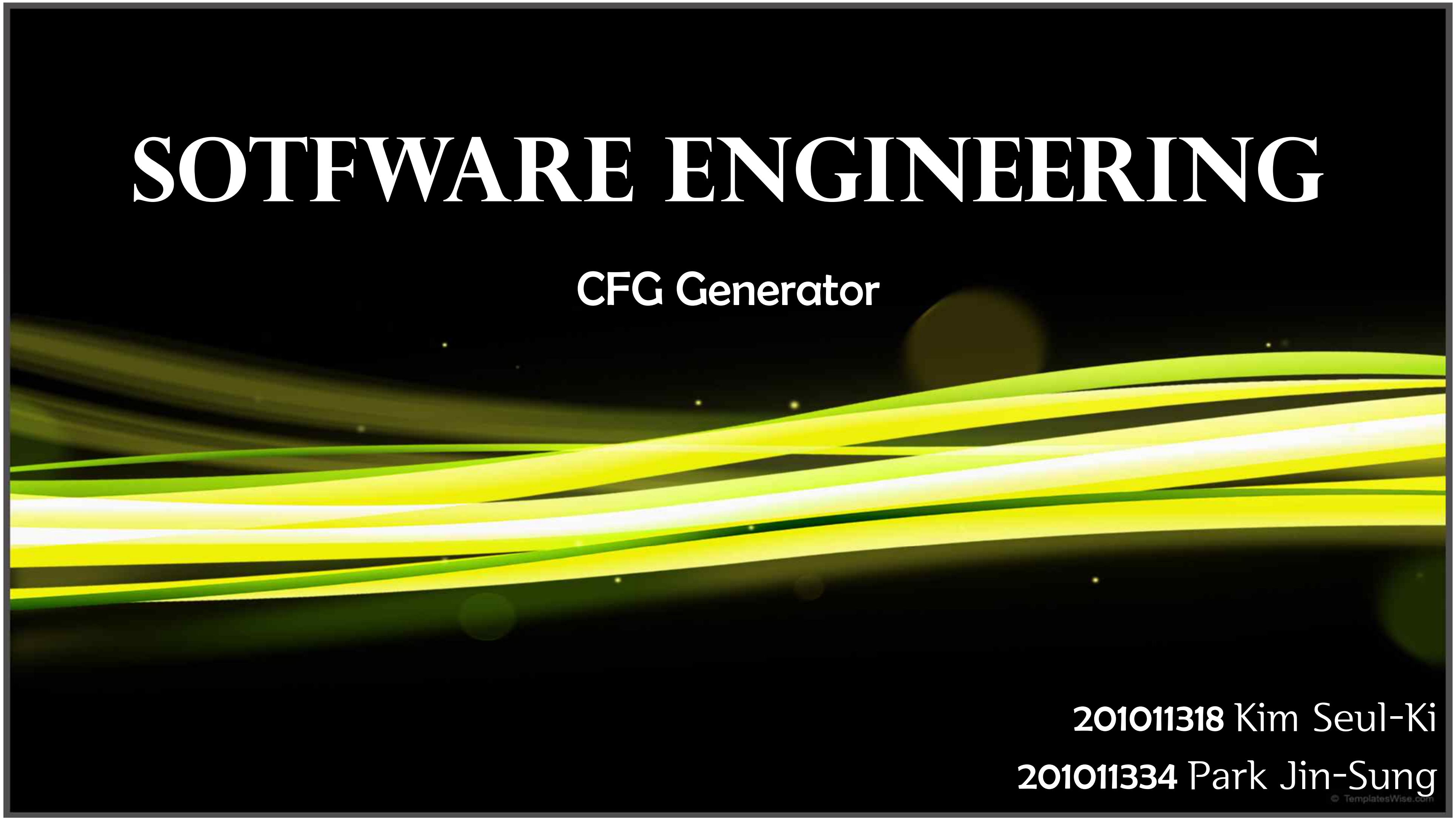


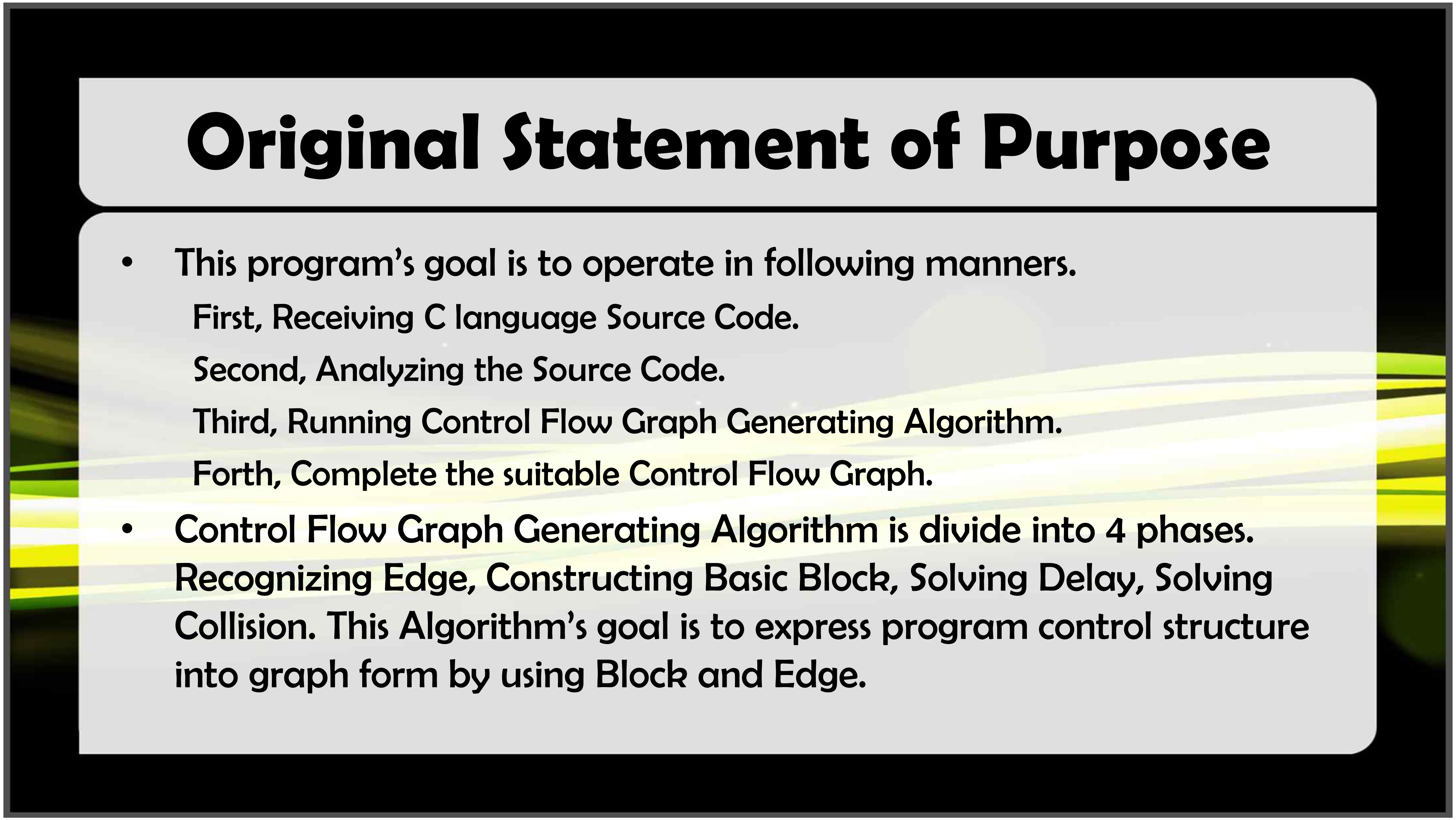
SOTFWARE ENGINEERING

CFG Generator

201011318 Kim Seul-Ki

201011334 Park Jin-Sung

1. SA & SD



Original Statement of Purpose

- This program's goal is to operate in following manners.

  First, Receiving C language Source Code.

  Second, Analyzing the Source Code.

  Third, Running Control Flow Graph Generating Algorithm.

  Forth, Complete the suitable Control Flow Graph.

- Control Flow Graph Generating Algorithm is divide into 4 phases. Recognizing Edge, Constructing Basic Block, Solving Delay, Solving Collision. This Algorithm's goal is to express program control structure into graph form by using Block and Edge.

Modified Statement of Purpose

- This program's goal is to operate in following manners.

  First, Receiving C language Source Code.

  Second, Analyzing the Source Code.

  Third, Running Control Flow Graph Generating Algorithm.

  Forth, Complete the suitable Control Flow Graph and Print message to console depends on Result.

1. When CFG generating is started, This program prints 'Start message' to console.
2. After finishing CFG generating, This program prints only 'Success message' and saves CFG into the File.

Modified Statement of Purpose

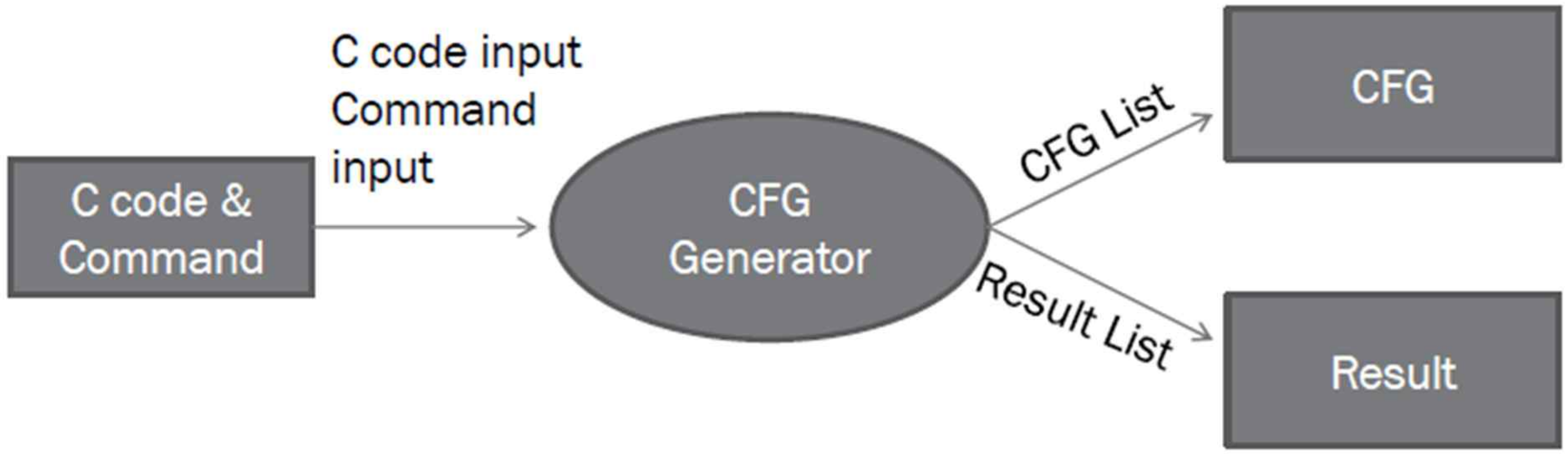
3. If CFG generating is failed, This program prints ‘Error Message’ to console and closes all processes.
4. This program handles C language Source code only.
5. This program converts only Main() Function parts.
6. If this source code hasn’t block like a ‘{}’ or path to the file is incorrect, CFG Generator is terminated by handling Error because it can’t generate CFG.

Modified Statement of Purpose

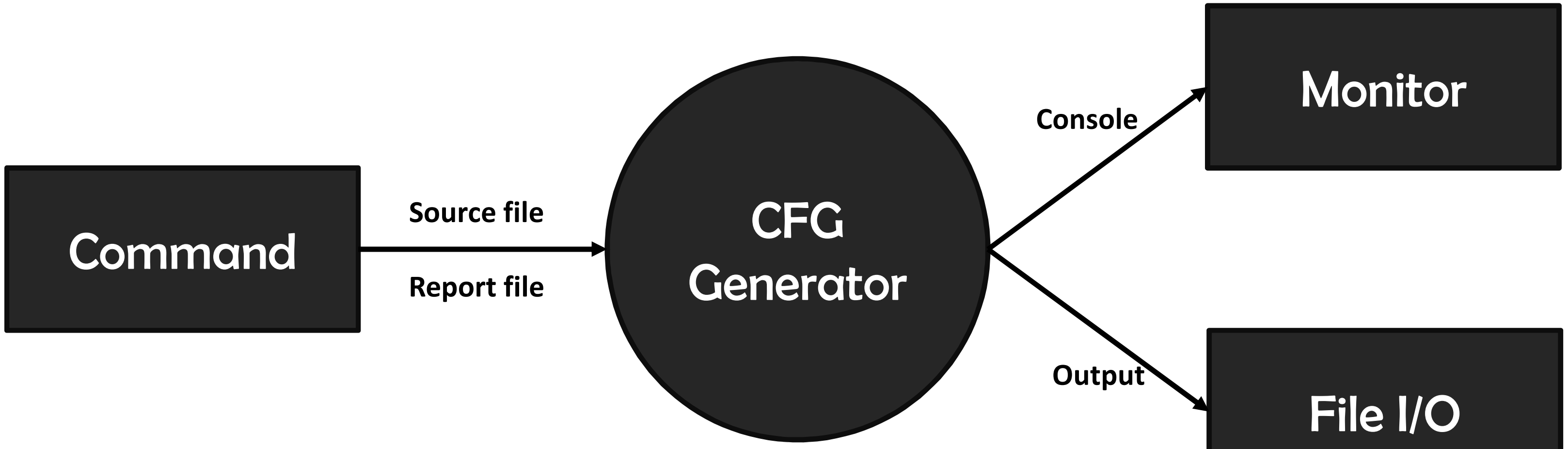
7. This program has no consideration for whole compile error. The program consider only 'CFG generating error' which is necessary for generating CFG.(e.g. Lack of Block like '{}' or Nonexistence of Main() Function.)
8. After generating CFG is completed, This program saves Basic Block List and Edge List of CFG on each line into the File.
9. If the path to the C Source Code File is wrong, Help message is printed.
   - Help Message follows the form like a next line.

```
$./cg <source code filepa(*.c)> <report filepath(*.txt)>
```

Original System Context Diagram



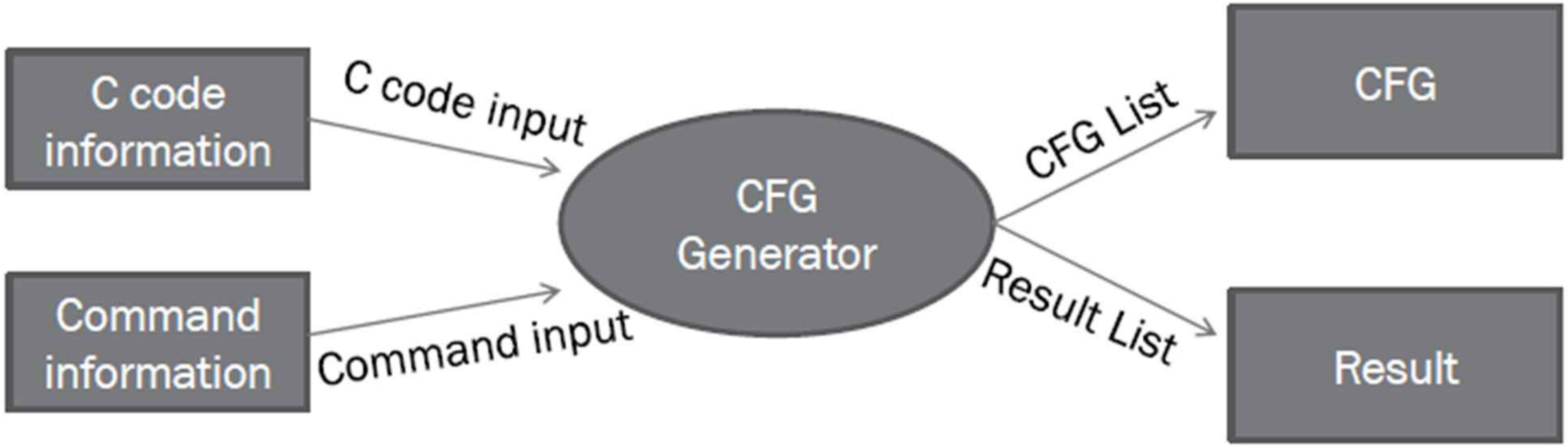
Modified System Context Diagram



Event List

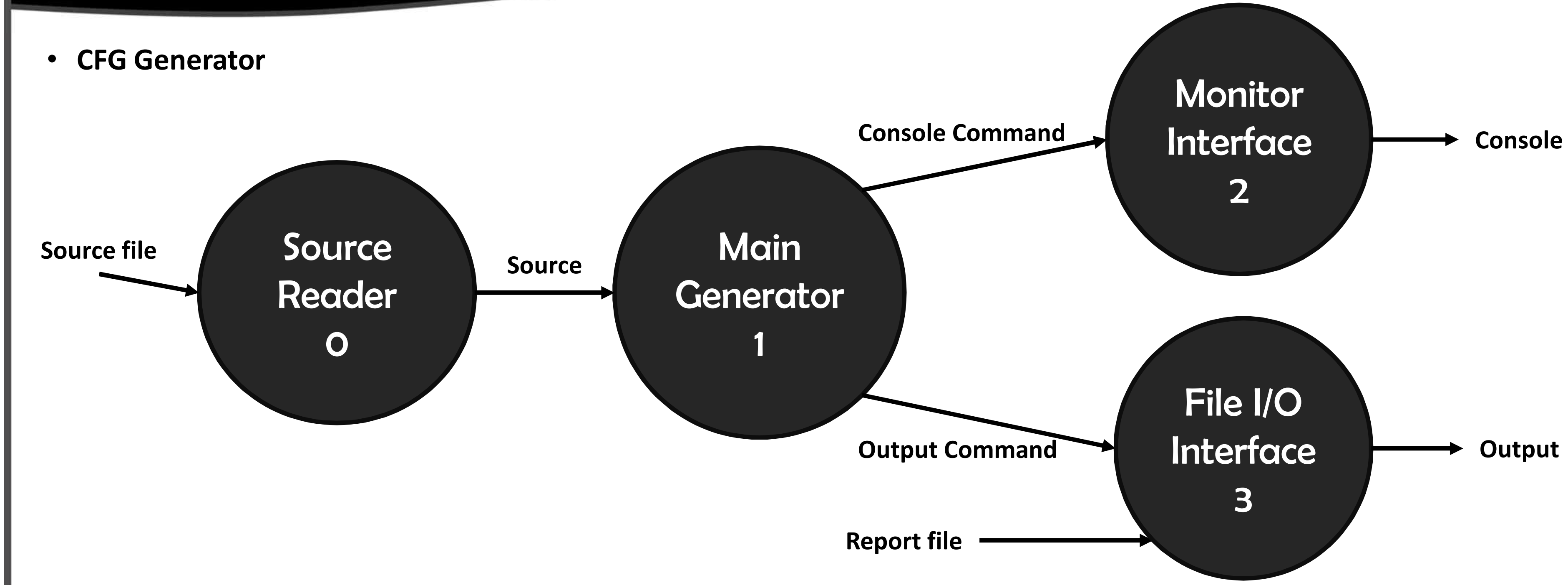
Input/Output Event	Description	Format/Type
Source File	The path to the C Source Code	String(*.c)
Report File	The path to the File of Complete CFG	String(*.txt)
Console	Success/Start/Error message that will be printed on console.	Message that will be printed on console.
Output	String that includes Basic Block and Edges in CFG	String that will be printed on File.

Original
Data Flow Diagram – Level 0



Modified
Data Flow Diagram – Level 0

- CFG Generator



Data Dictionary – Level 0

Input/Output Event	Description	Format/Type
Source	The String that is converted from Source File.	String/char *
Console Command	String for printing on Console.	String/char *
Output Command	String for writing on File.	String/char *

Process Specification– Level 0

Name	Source Reader
Reference Number	0
Input	Source File
Output	Source
Description	This process extracts whole Source from Source File by using File I/O. If Source File isn't correct path, the process saves NULL in Source.

Process Specification– Level 0

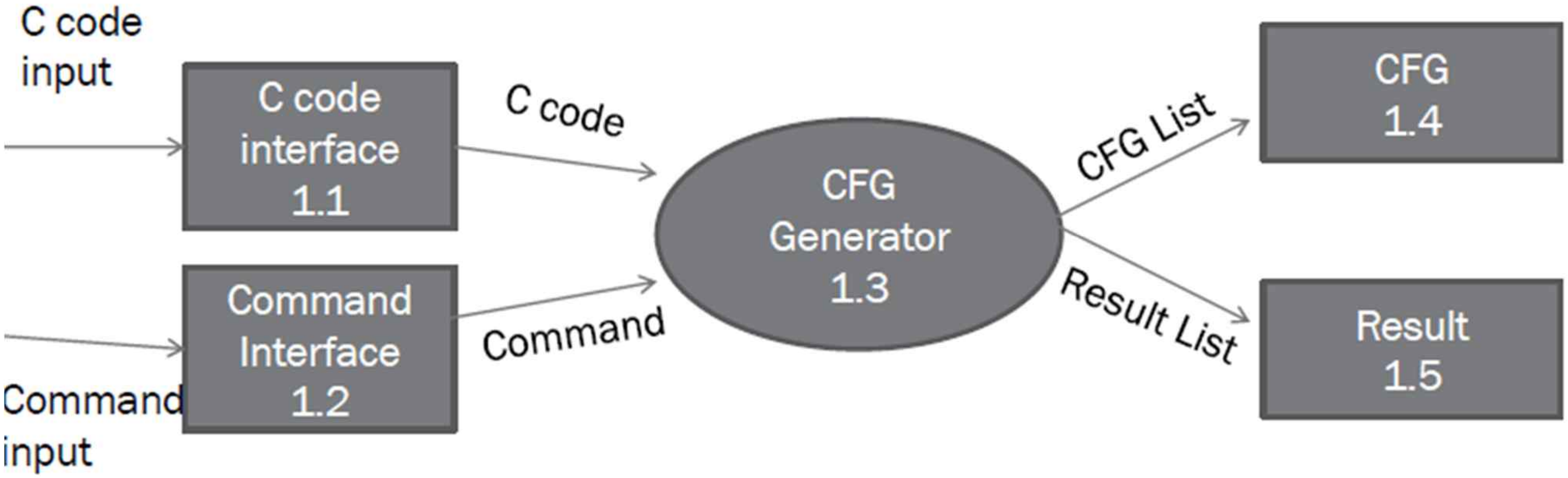
Name	Main Generator
Reference Number	1
Input	Source
Output	Console Command, Output Command
Description	First, This process parses the received Source. Next, the process prints the Error message or Success message. Last, the process writes CFG to the Source File.

Process Specification– Level 0

Name	Monitor Interface
Reference Number	2
Input	Console Command
Output	Console
Description	This process takes Console Command and prints the console message on the monitor.

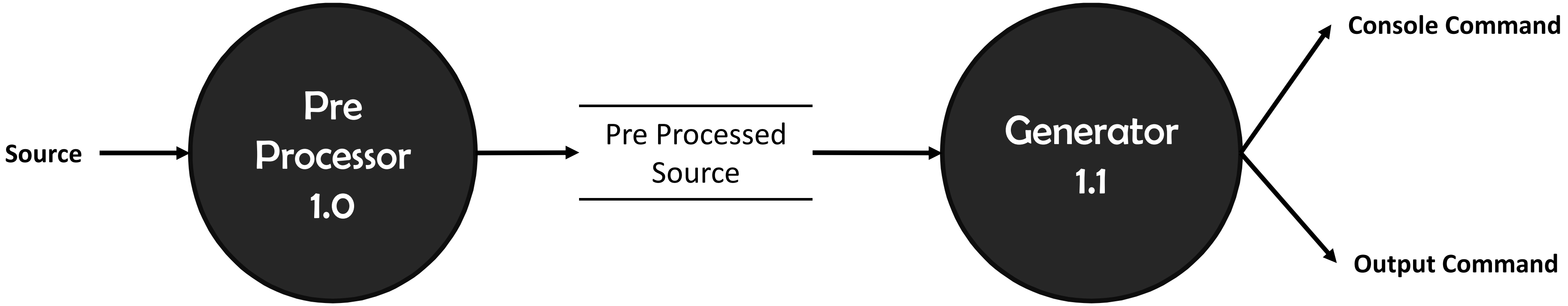
Name	File I/O Interface
Reference Number	3
Input	Output Command
Output	Output
Description	This process gets Output Command and outputs Output to Report File.

Original
Data Flow Diagram – Level 1



Modified
Data Flow Diagram – Level 1

- **Main Generator 1**



Data Dictionary – Level 1

Input/Output Event	Description	Format/Type
Pre Processed Source	String of main function is extracted from the Source.	String/char *

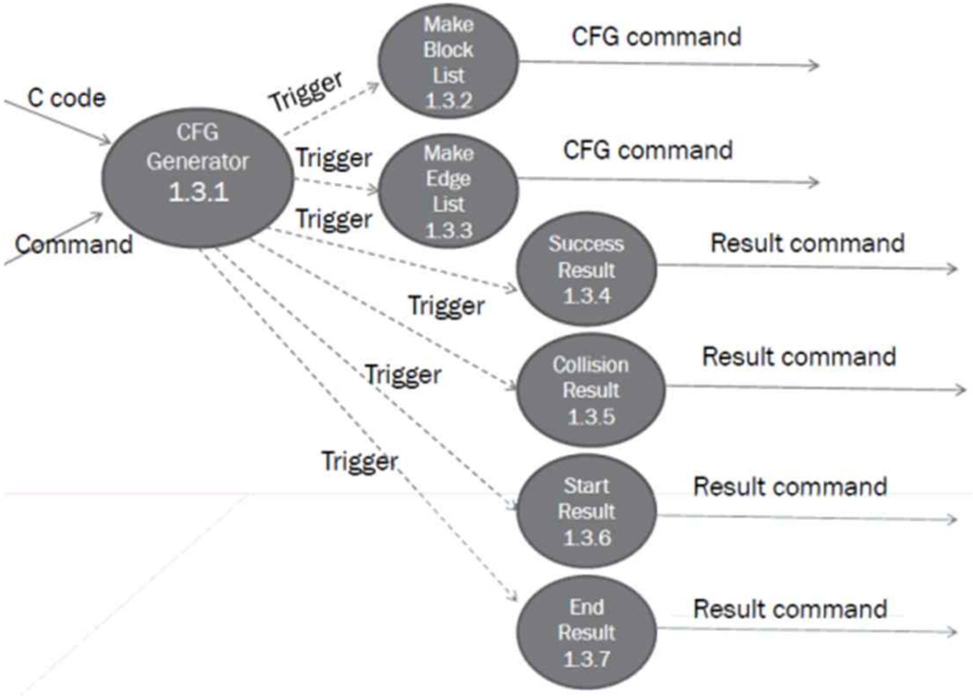
Process Specification– Level 1

Name	Pre Processor
Reference Number	1.0
Input	Source
Output	Pre Processed Source
Description	This process extracts the inside of main function from Source and saves that at the Pre-processed Source. If Source is NULL, the process saves NULL at the Pre-processed Source.

Process Specification– Level 1

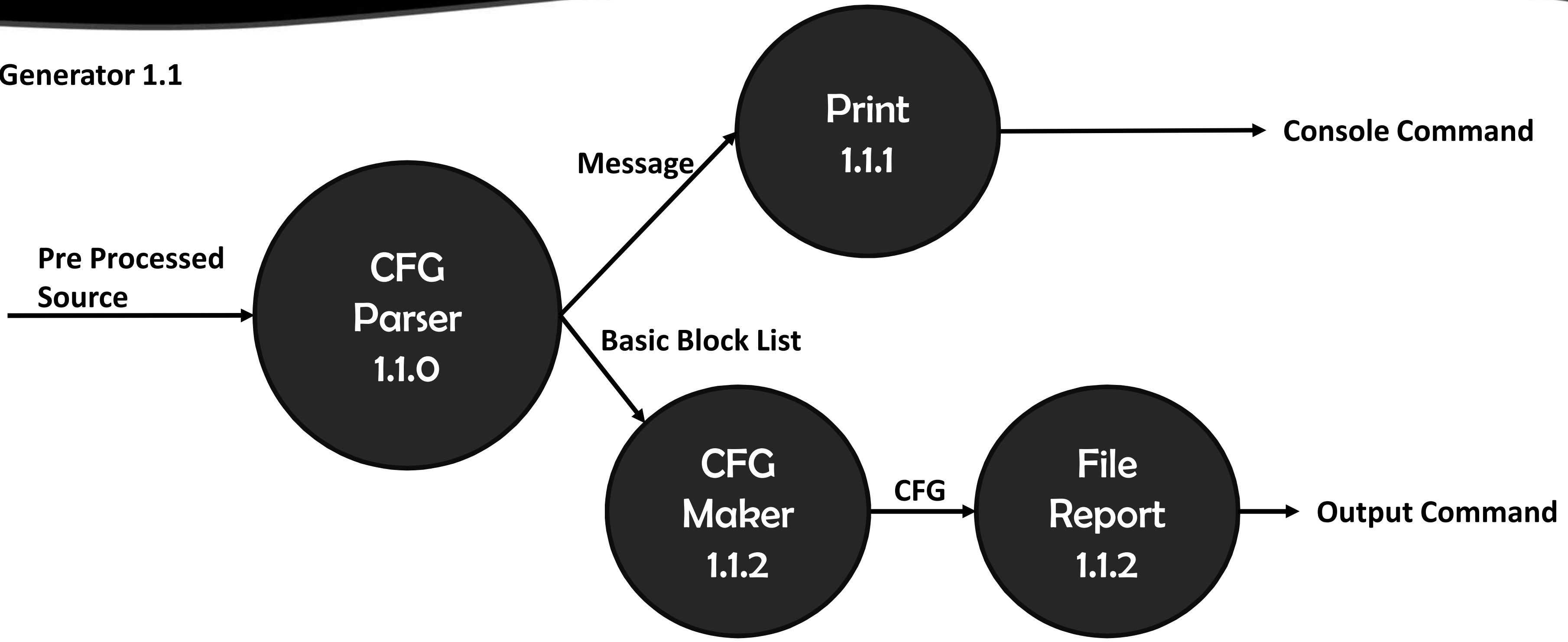
Name	Generator
Reference Number	1.1
Input	Pre Processed Source
Output	Output Command, Console Command
Description	After Converting from pre-processed source to CFG form, this process outputs appropriate Output Command and Console Command.

Original
Data Flow Diagram – Level 2



Modified
Data Flow Diagram – Level 2

- Generator 1.1



Data Dictionary – Level 2

Input/Output Event	Description	Format/Type
Message	String included the Error/Success/Start message.	String/char *
Basic Block List	List of the Basic Blocks that is made in each phases. The Basic Block is connected with each others.	Connected List and Source
CFG	The final result of connected Basic Blocks	Basic Block is connected with each others./Graph

Process Specification– Level 2

Name	CFG Parser
Reference Number	1.1.0
Input	Pre Processed Source
Output	Message, Basic Block List(+ Data structure)
Description	This process parses Pre-processed Source to Basic Block List. If the process fails to parsing, the process outputs Fail message.(In this case, Basic Block List isn't outputted.) Or If the process successes to parsing, the process outputs Basic Block List.(In this case, Message isn't outputted.)

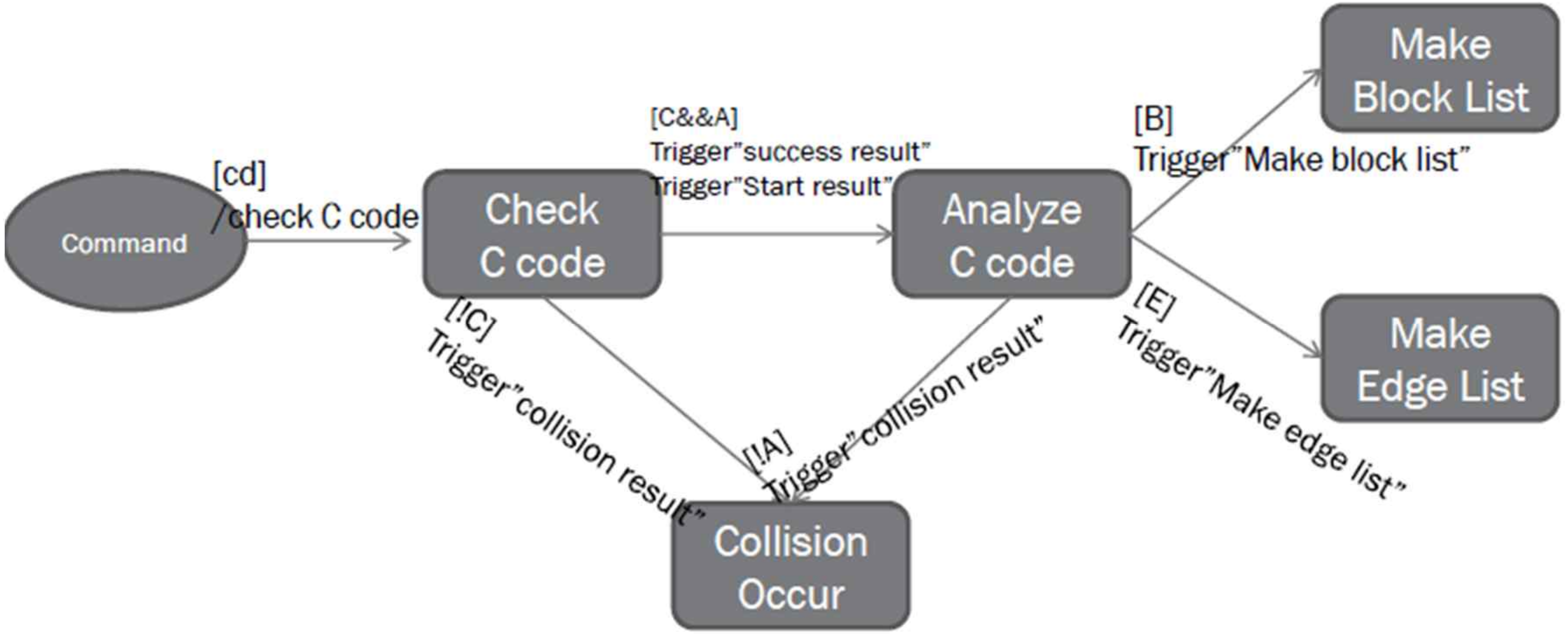
Process Specification– Level 2

Name	Print
Reference Number	1.1.1
Input	Message
Output	Console Command
Description	This process takes Message and converts to Console Command.

Process Specification– Level 2

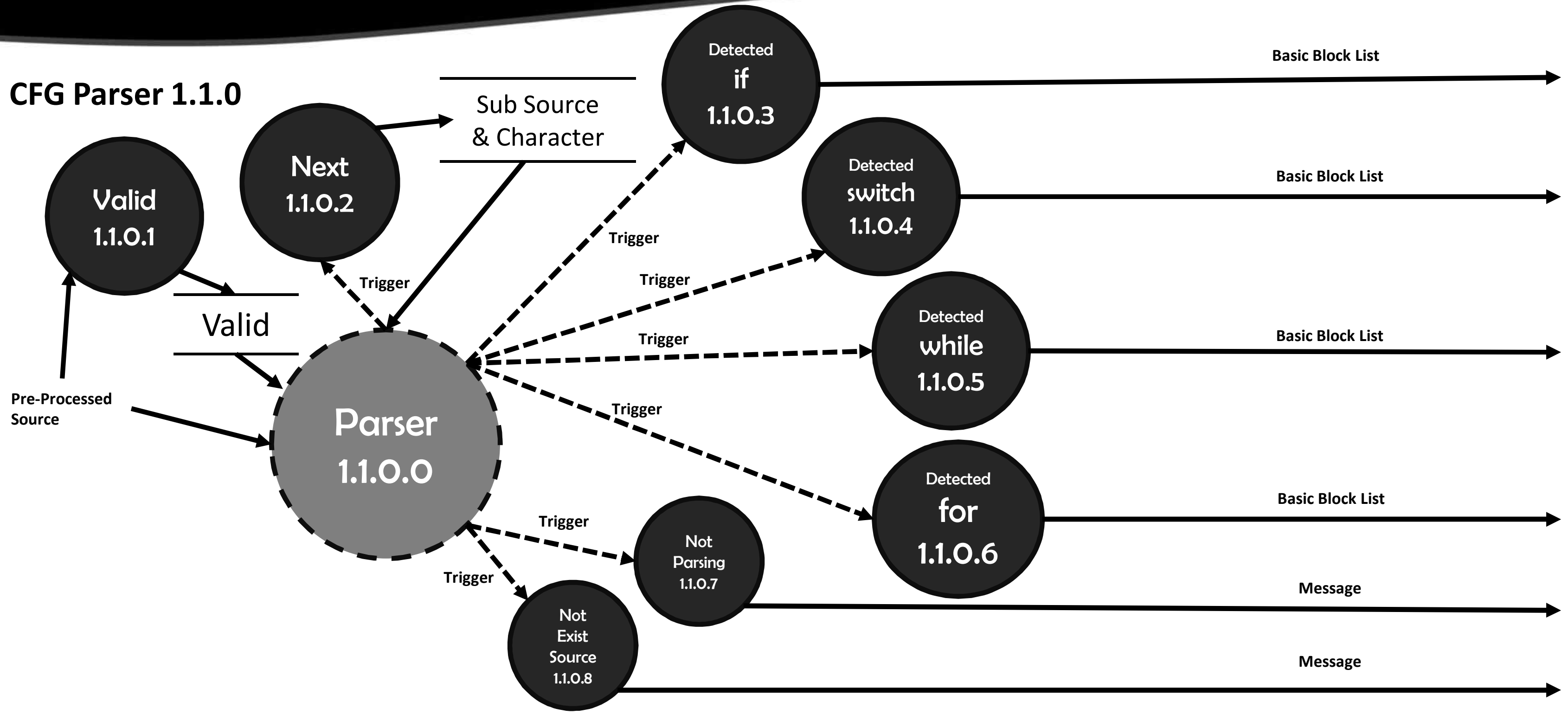
Name	File Report
Reference Number	1.1.2
Input	CFG
Output	Output Command
Description	While this process explores CFG in hierarchy structure form, this process extracts Basic Blocks and Edges. And this process outputs them to Output Command.

Original
Data Flow Diagram – Level 3



Modified
Data Flow Diagram – Level 3

- **CFG Parser 1.1.0**



Data Dictionary – Level 3

Input/Output Event	Description	Format/Type
Valid	The result of checking pre-processed source is correct.	String/ char *
Sub Source	(After the), A piece of the source that is combination of letters. The letters are outputted from the Next process.	String / char *
Character	Words from the Next process.	Words / char

Process Specification– Level 3

Name	Parser
Reference Number	1.1.0.0
Input	Pre Processed Source(PPS), Valid(V), Sub Source(SS), Character(C)
Output	Trigger
Description	If PPS is NULL, this process triggers the Not Exist Source process. Next, this process calls the Next process in each time and gets C and SS. And according to their condition, the process triggers appropriate Detected process and Statement process.

Process Specification– Level 3

Name	Valid
Reference Number	1.1.0.1
Input	Pre Processed Source
Output	Valid
Description	While this process explores Pre-processed Source, the process checks parenthesises is correct.

Process Specification– Level 3

Name	Next
Reference Number	1.1.0.2
Input	Trigger
Output	Sub Source(SS), Character(C)
Description	This process saves one letter at the Character.(One letter is extracted from Pre-processed Source.) After SS is cleared, the process saves the word at the Sub Source. (the word is made by combinating letters from Character.)

Process Specification– Level 3

Name	Detected if
Reference Number	1.1.0.3
Input	Trigger
Output	Basic Block List
Description	This process calls the Next process and makes Basic Blocks by parsing whole 'If statement'. Next, the process saves them at the Basic Block List.

Process Specification– Level 3

Name	Detected switch
Reference Number	1.1.0.4
Input	Trigger
Output	Basic Block List
Description	This process calls the Next process and makes Basic Blocks by parsing whole 'Switch statement'. Next, the process saves them at the Basic Block List.

Process Specification– Level 3

Name	**Detected while**
Reference Number	1.1.0.5
Input	Trigger
Output	Basic Block List
Description	This process calls the Next process and makes Basic Blocks by parsing whole 'While statement'. Next, the process saves them at the Basic Block List.

Process Specification– Level 3

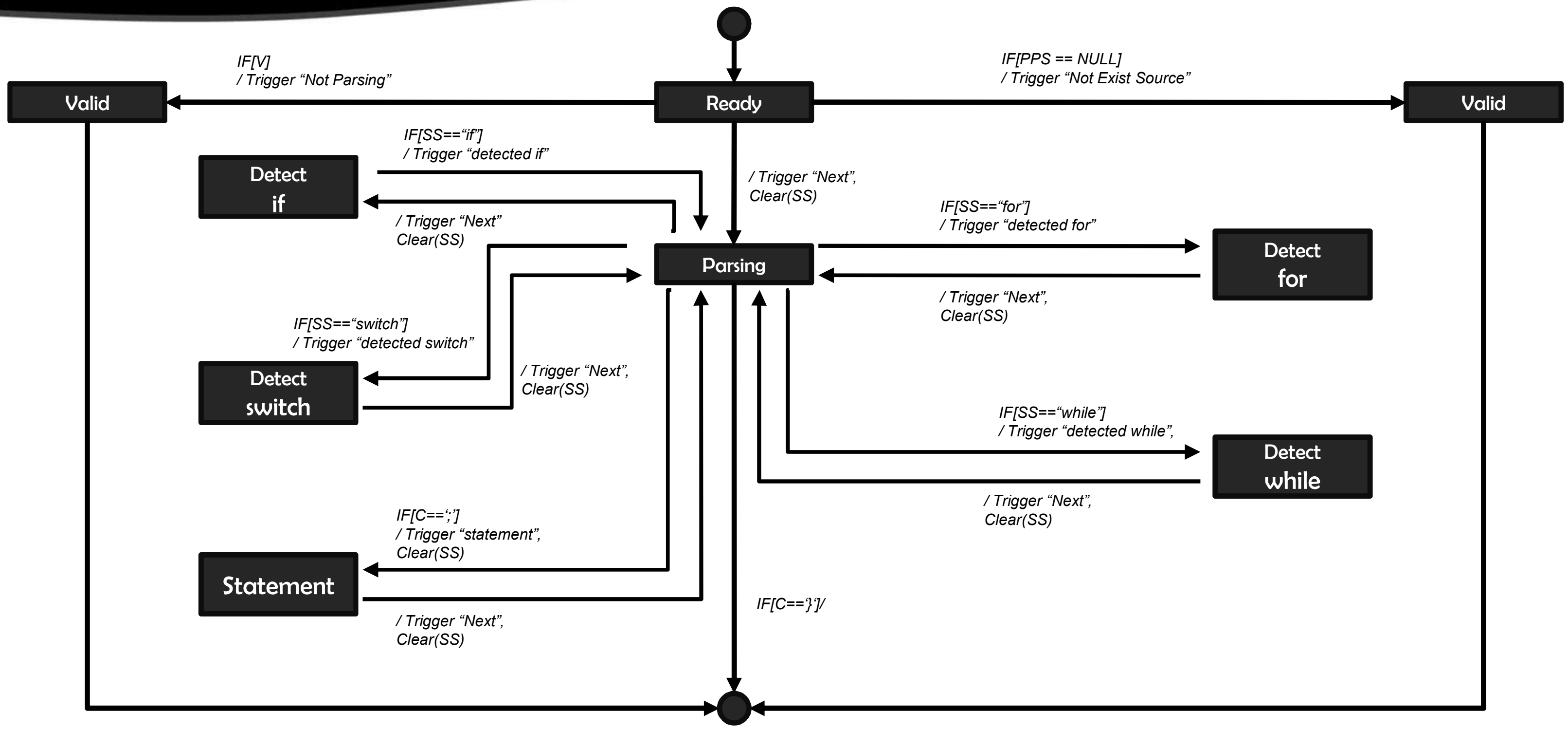
Name	Detected for
Reference Number	1.1.0.6
Input	Trigger
Output	Basic Block List
Description	This process calls the Next process and makes Basic Blocks by parsing whole 'For statement'. Next, the process saves them at the Basic Block List.

Process Specification– Level 3

Name	Not Parsing
Reference Number	1.1.0.7
Input	Trigger
Output	Message
Description	This Process saves ‘Error’ at the Message.

Name	Not Exist Source
Reference Number	1.1.0.8
Input	Trigger
Output	Message
Description	This process saves ‘Usage’ at the Message.

State
Transition Diagram – Level 4



Valid

IF[V]
/ Trigger “Not Parsing”

Ready

IF[PPS == NULL]
/ Trigger “Not Exist Source”

Valid

Detect
if

IF[SS==“if”]
/ Trigger “detected if”

/ Trigger “Next”
Clear(SS)

/ Trigger “Next”,
Clear(SS)

Parsing

IF[SS==“for”]
/ Trigger “detected for”

Detect
for

/ Trigger “Next”,
Clear(SS)

IF[SS==“switch”]
/ Trigger “detected switch”

Detect
switch

/ Trigger “Next”,
Clear(SS)

IF[SS==“while”]
/ Trigger “detected while”,

Detect
while

/ Trigger “Next”,
Clear(SS)

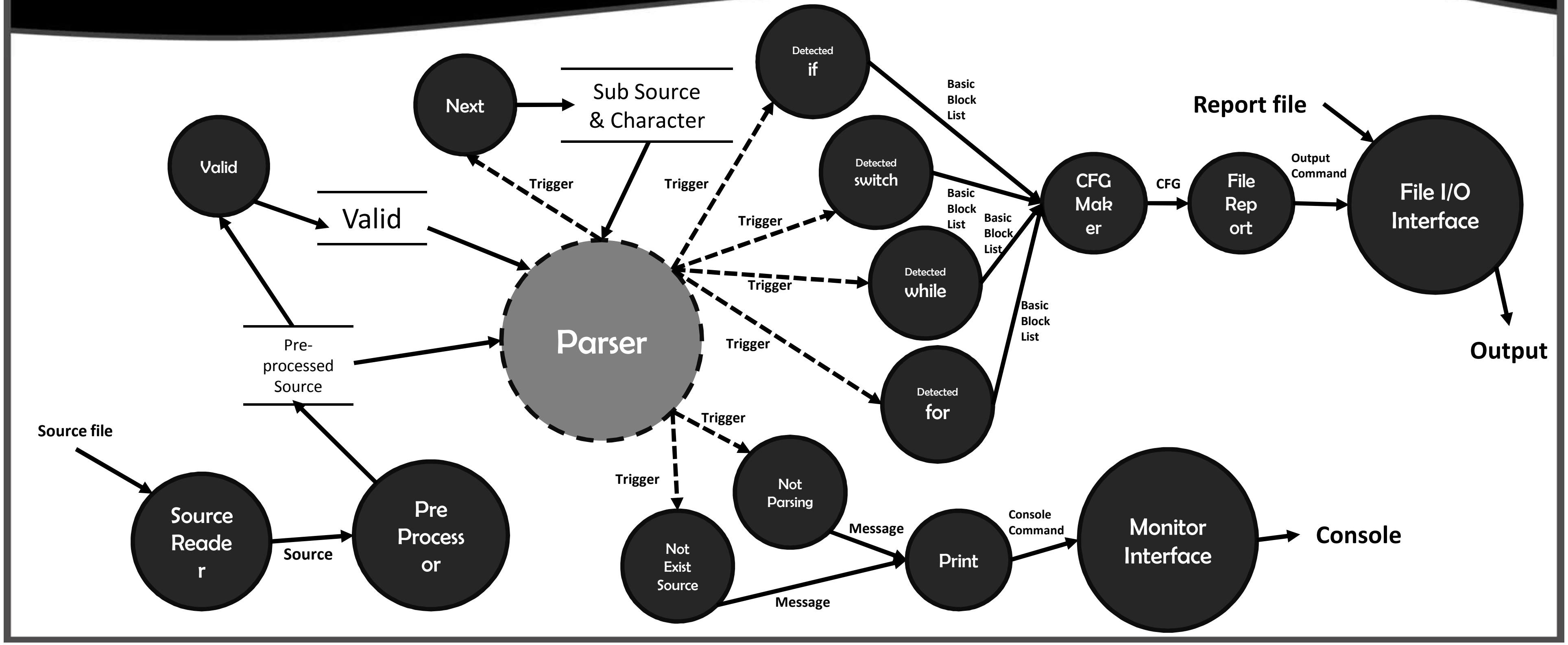
IF[C==‘;’]
/ Trigger “statement”,
Clear(SS)

Statement

/ Trigger “Next”,
Clear(SS)

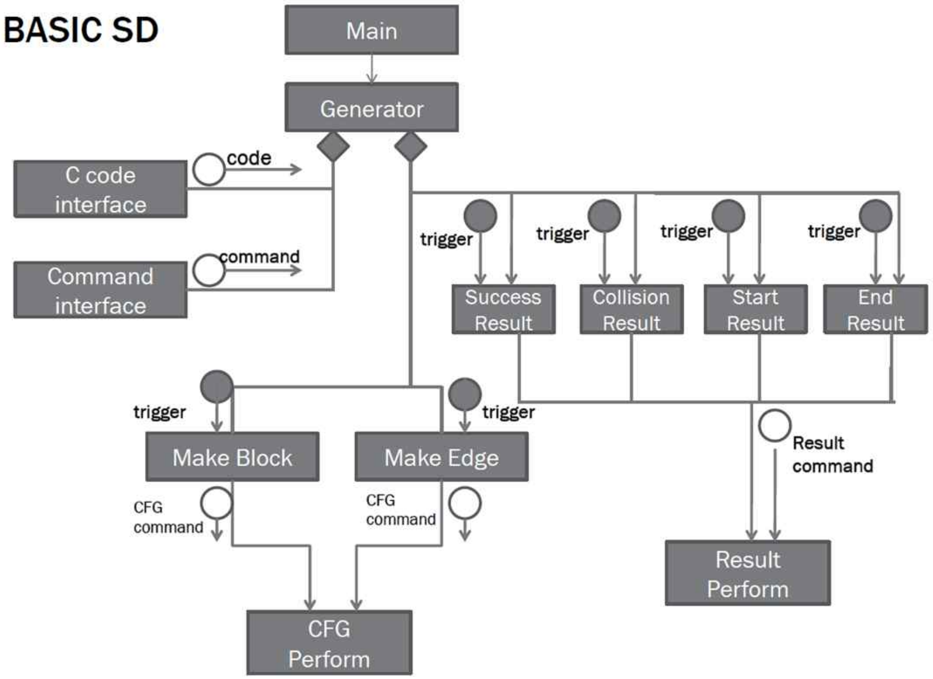
IF[C==‘}‘]/

Modified
Total Data Flow Diagram

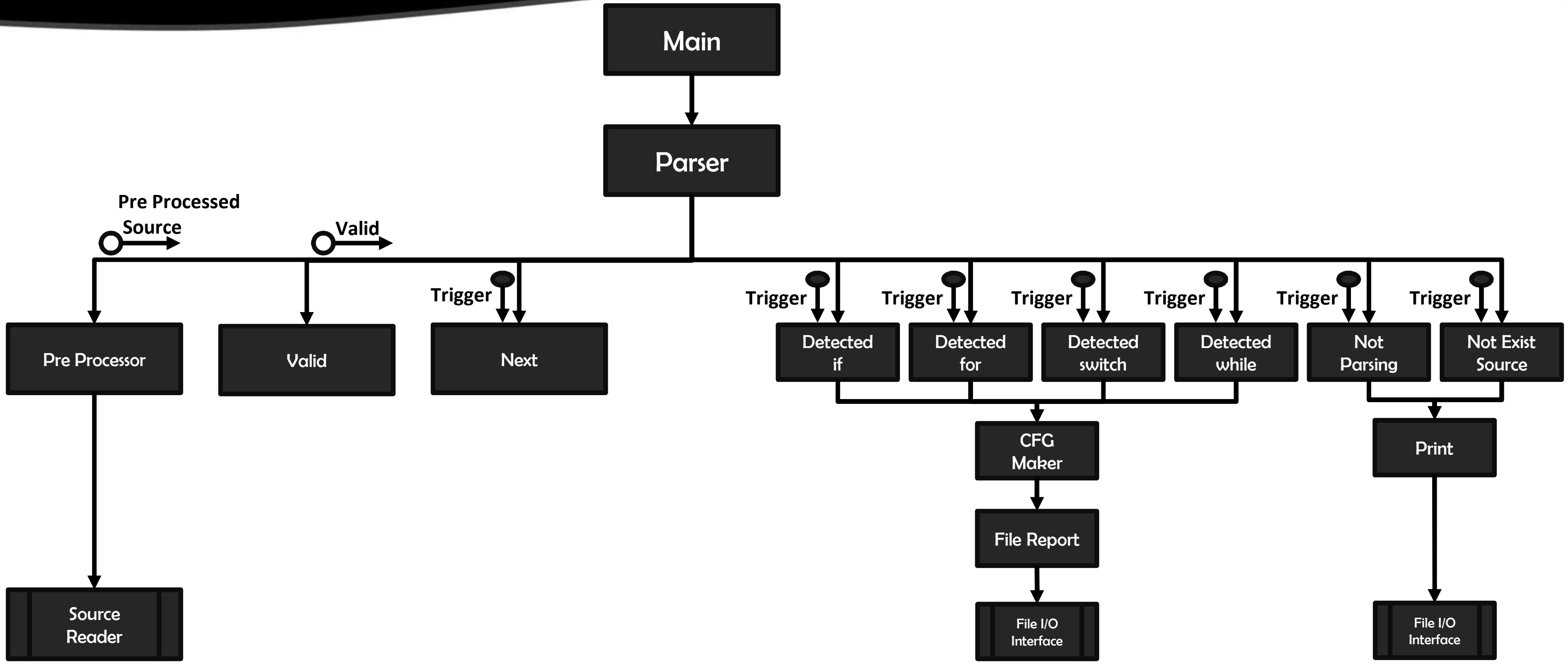


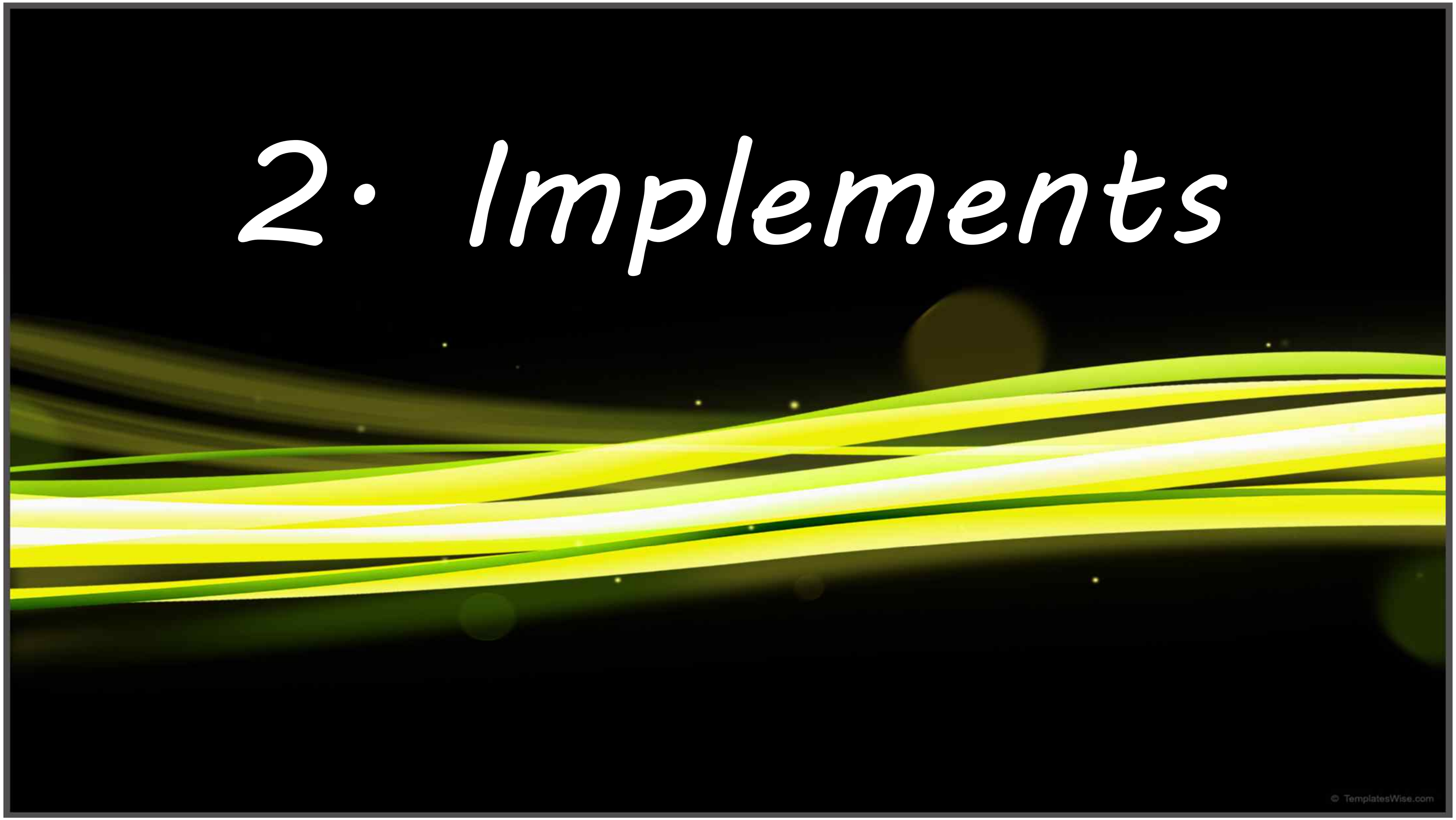
Original
Structured Charts

BASIC SD



Modified Structured Charts





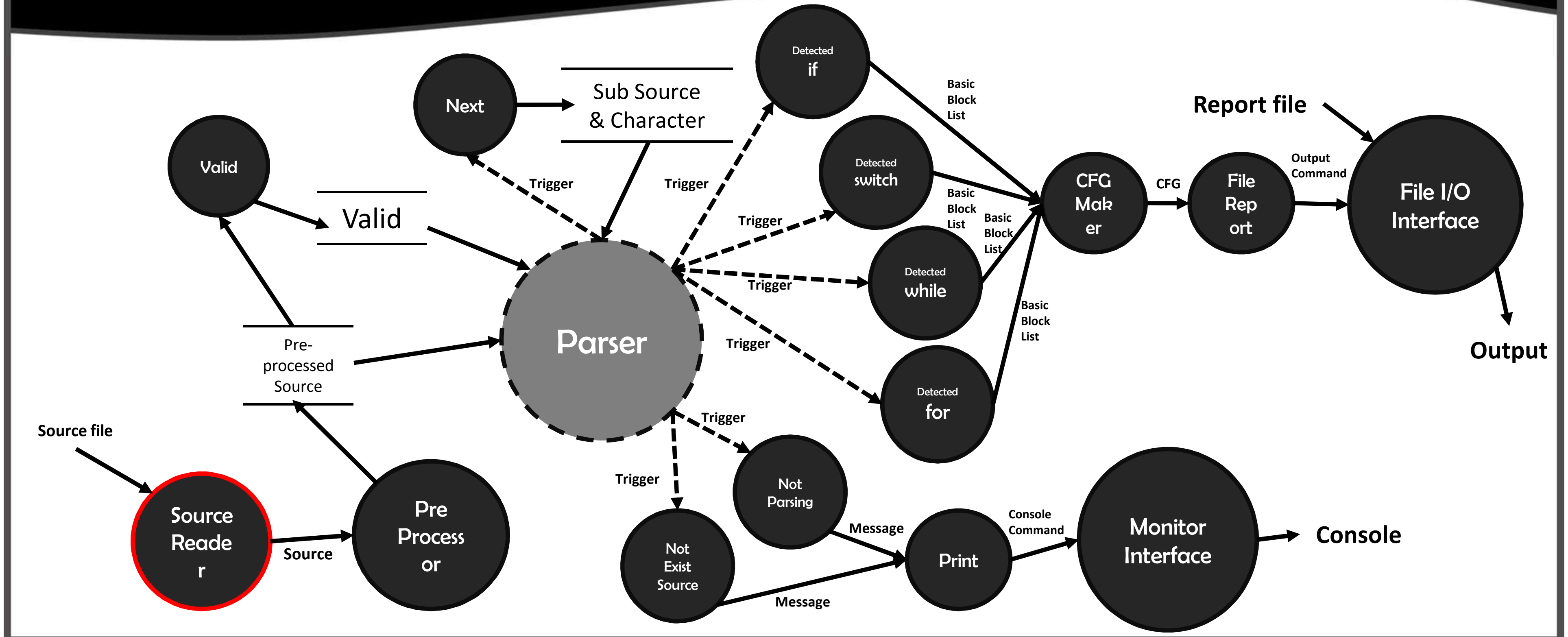
2· Implements

Source File & Header File

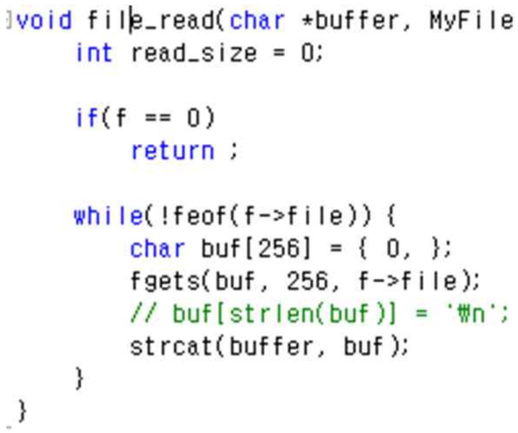
- **Source File**
  main.c main_generator.c file.c list.c report.c
  Stdafx.c utils.c CFG.c detected_if.c detected_for.c
  detected_switch.c detected_while.c

- **Header File**
  main_generator.c CFG.h file.h list.h
  report.h stdafx.h utils.h

Modified
Total Data Flow Diagram



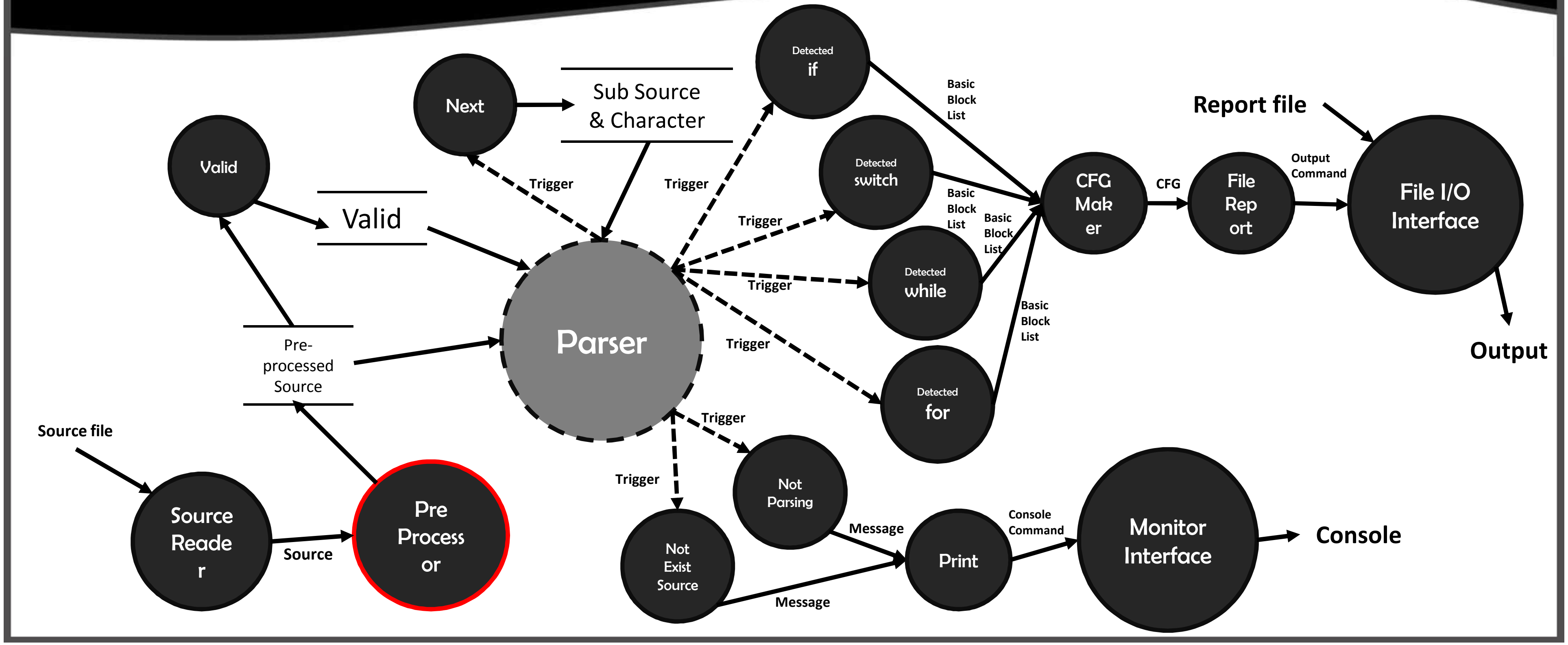
Source Reader



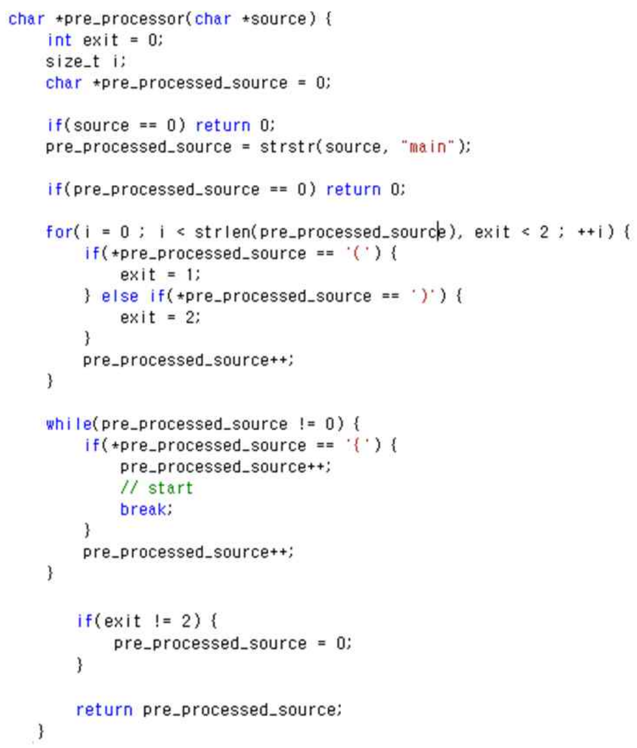
```
void file_read(char *buffer, MyFile *f) {
    int read_size = 0;

    if(f == 0)
        return ;

    while(!feof(f->file)) {
        char buf[256] = { 0, };
        fgets(buf, 256, f->file);
        // buf[strlen(buf)] = '₩n';
        strcat(buffer, buf);
    }
}
```

- **Those Function that its name starts ‘file_’ is function for writing or reading File.**
- **Then, file_read function reads the File and moves them to ‘buffer’ memory.**

Modified Total Data Flow Diagram



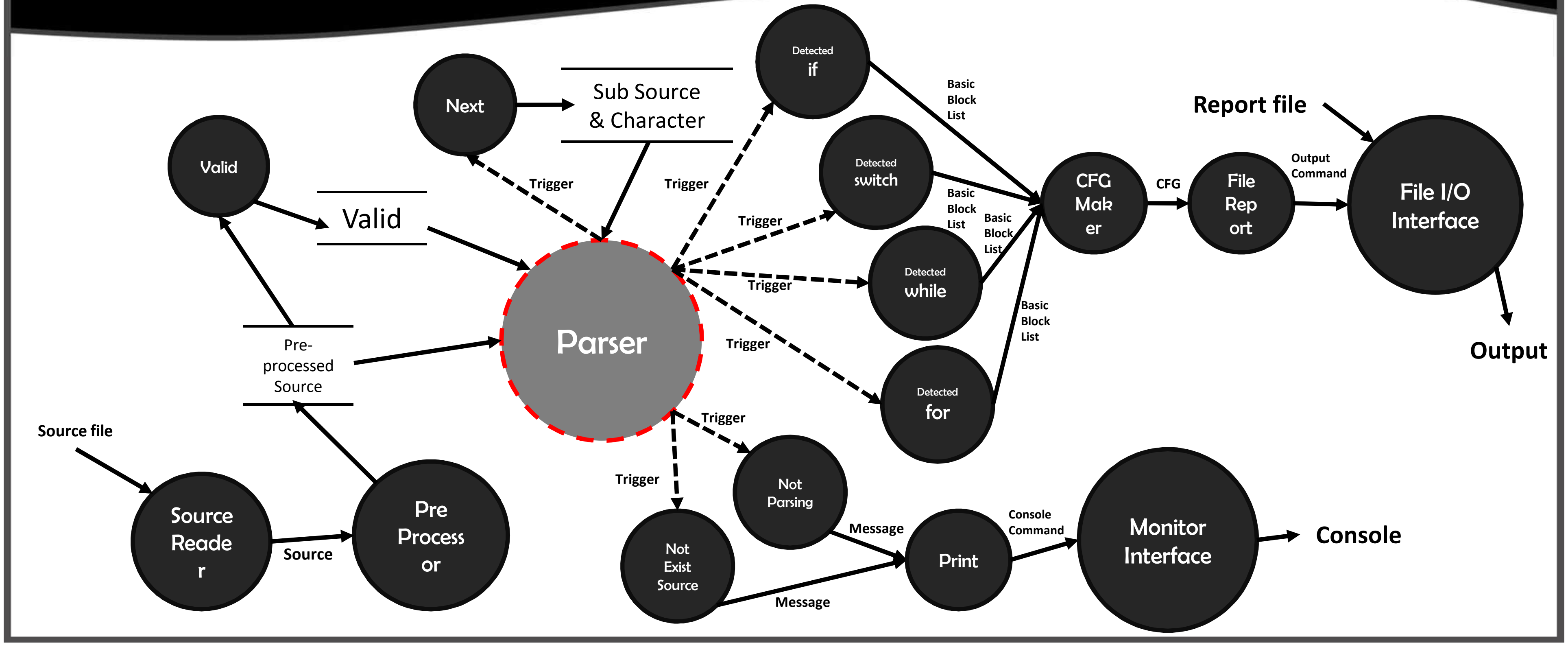
Pre-processor



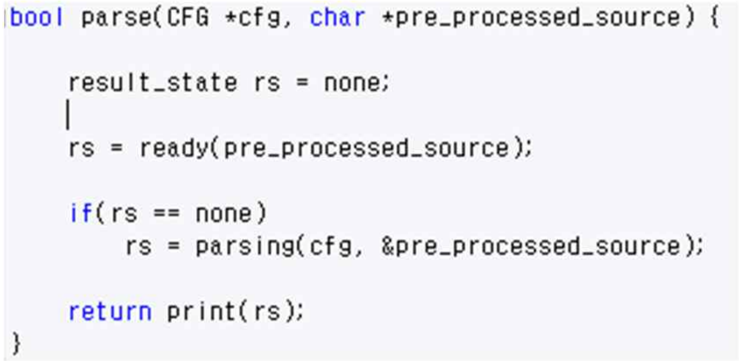
```
char *pre_processor(char *source) {
    int exit = 0;
    size_t i;
    char *pre_processed_source = 0;

    if(source == 0) return 0;
    pre_processed_source = strstr(source, "main");

    if(pre_processed_source == 0) return 0;

    for(i = 0 ; i < strlen(pre_processed_source), exit < 2 ; ++i) {
        if(*pre_processed_source == '(') {
            exit = 1;
        } else if(*pre_processed_source == ')') {
            exit = 2;
        }
        pre_processed_source++;
    }

    while(pre_processed_source != 0) {
        if(*pre_processed_source == '{') {
            pre_processed_source++;
            // start
            break;
        }
        pre_processed_source++;
    }


       if(exit != 2) {
           pre_processed_source = 0;
       }

       return pre_processed_source;
   }
```

- **Pre-precessor function gets Source from file_read function and extracts main() function.**
- **This function’s role is modifying String before Parsing.**

Modified
Total Data Flow Diagram



Parse



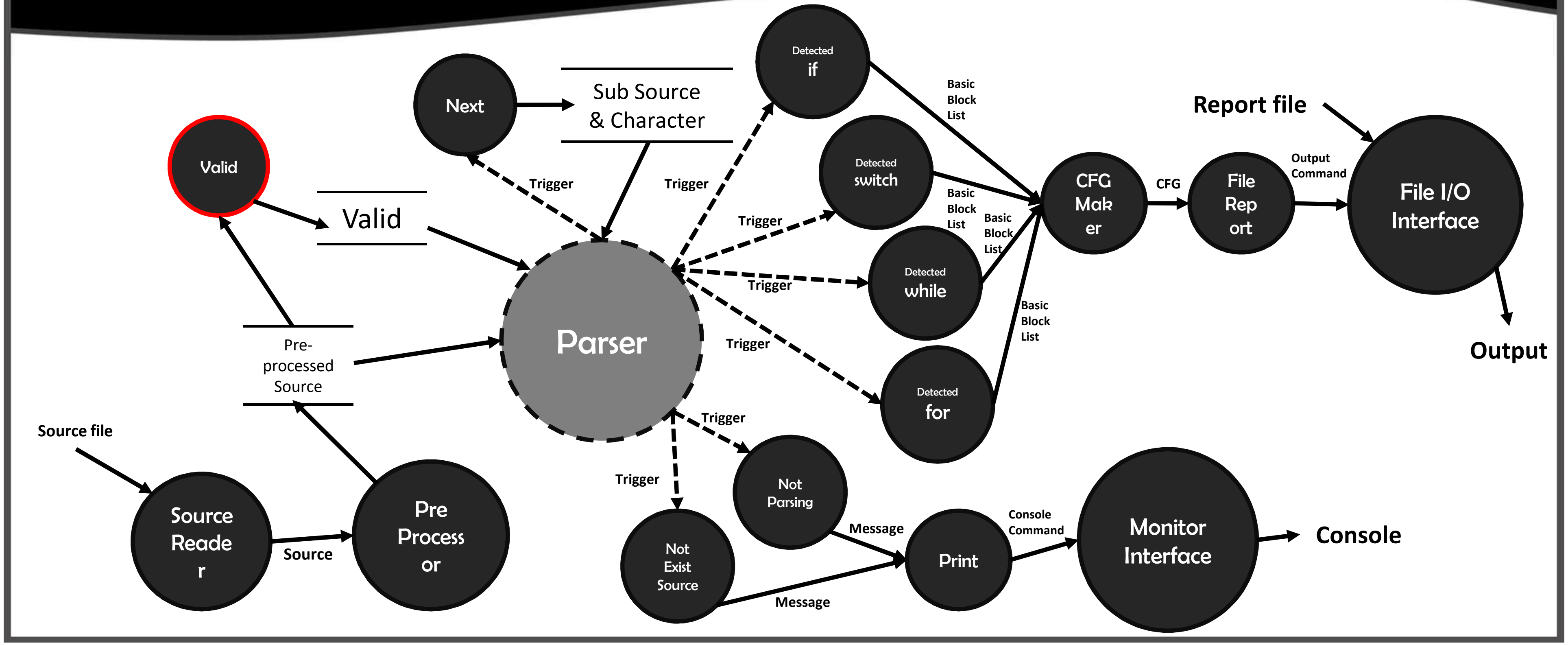
```
bool parse(CFG *cfg, char *pre_processed_source) {

    result_state rs = none;

    rs = ready(pre_processed_source);

    if(rs == none)
        rs = parsing(cfg, &pre_processed_source);

    return print(rs);
}
```

- **This function corresponds ‘Parser’ process.**

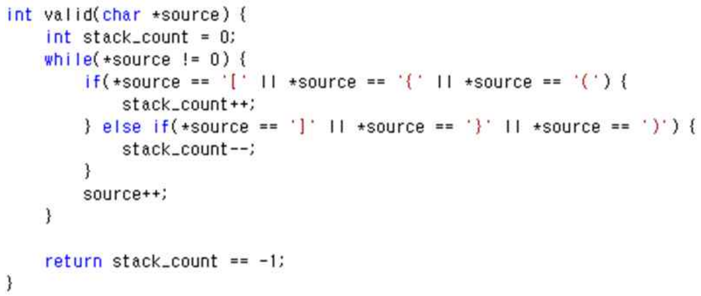
- **Starting ‘ready’ state first, It becomes ‘parsing’ state after checking valid.**

- **If It is failed, This function calls print according to result_state.**

Modified
Total Data Flow Diagram



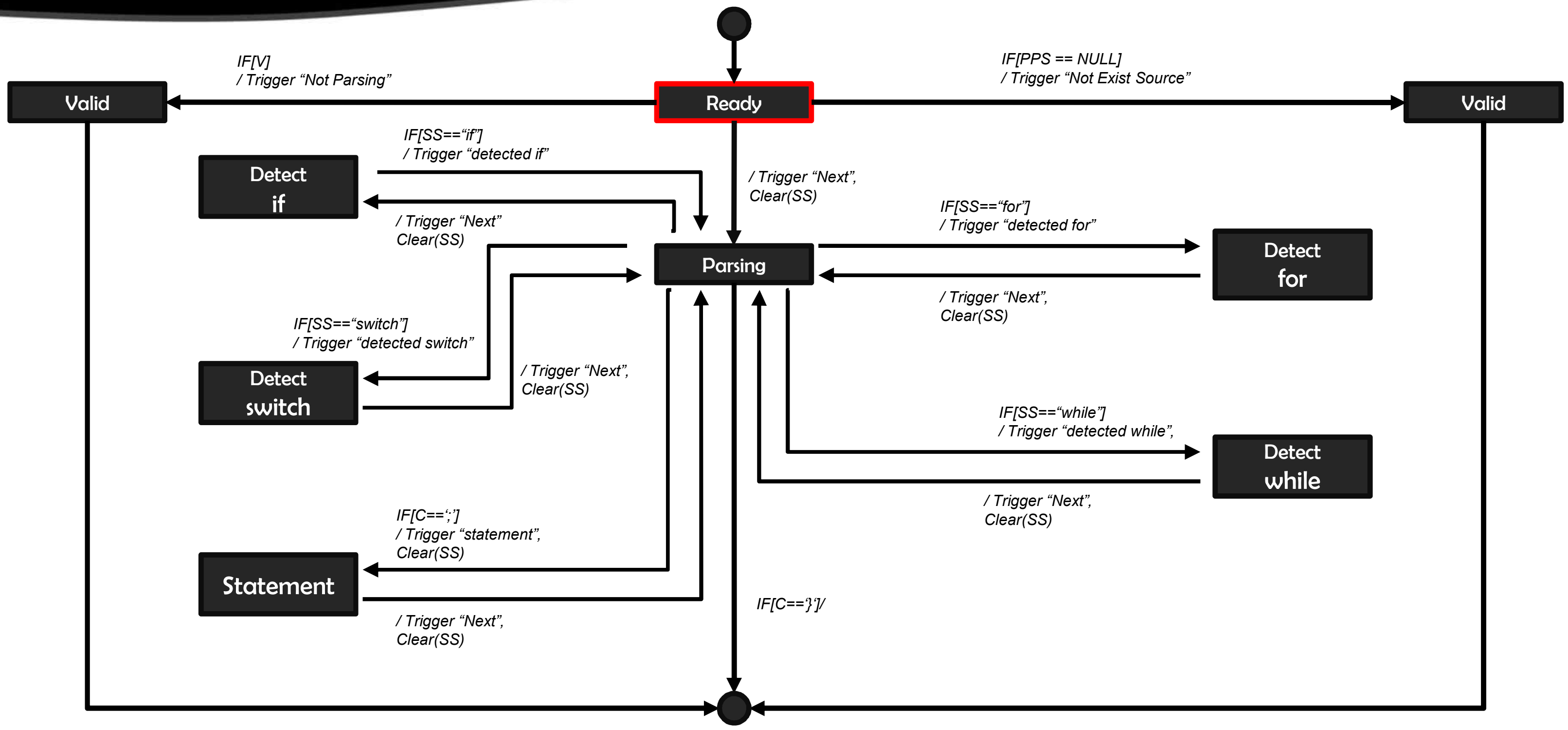
Valid



```c
int valid(char *source) {
    int stack_count = 0;
    while(*source != 0) {
        if(*source == '[' || *source == '{' || *source == '(') {
            stack_count++;
        } else if(*source == ']' || *source == '}' || *source == ')') {
            stack_count--;
        }
        source++;
    }

    return stack_count == -1;
}
```

- **This function checks parenthesis is correct.**

State Transition Diagram – Level 4



IF[V]
/ Trigger “Not Parsing”

Valid

Ready

IF[PPS == NULL]
/ Trigger “Not Exist Source”

Valid

IF[SS==“if”]
/ Trigger “detected if”

Detect
if

/ Trigger “Next”
Clear(SS)

/ Trigger “Next”,
Clear(SS)

IF[SS==“for”]
/ Trigger “detected for”

Parsing

Detect
for

/ Trigger “Next”,
Clear(SS)

IF[SS==“switch”]
/ Trigger “detected switch”

Detect
switch

/ Trigger “Next”,
Clear(SS)

IF[SS==“while”]
/ Trigger “detected while”,

Detect
while

/ Trigger “Next”,
Clear(SS)

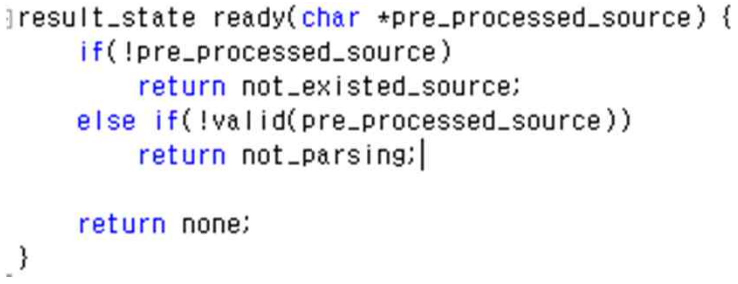
IF[C==‘;’]
/ Trigger “statement”,
Clear(SS)

Statement

/ Trigger “Next”,
Clear(SS)

IF[C==‘}’]/

Ready

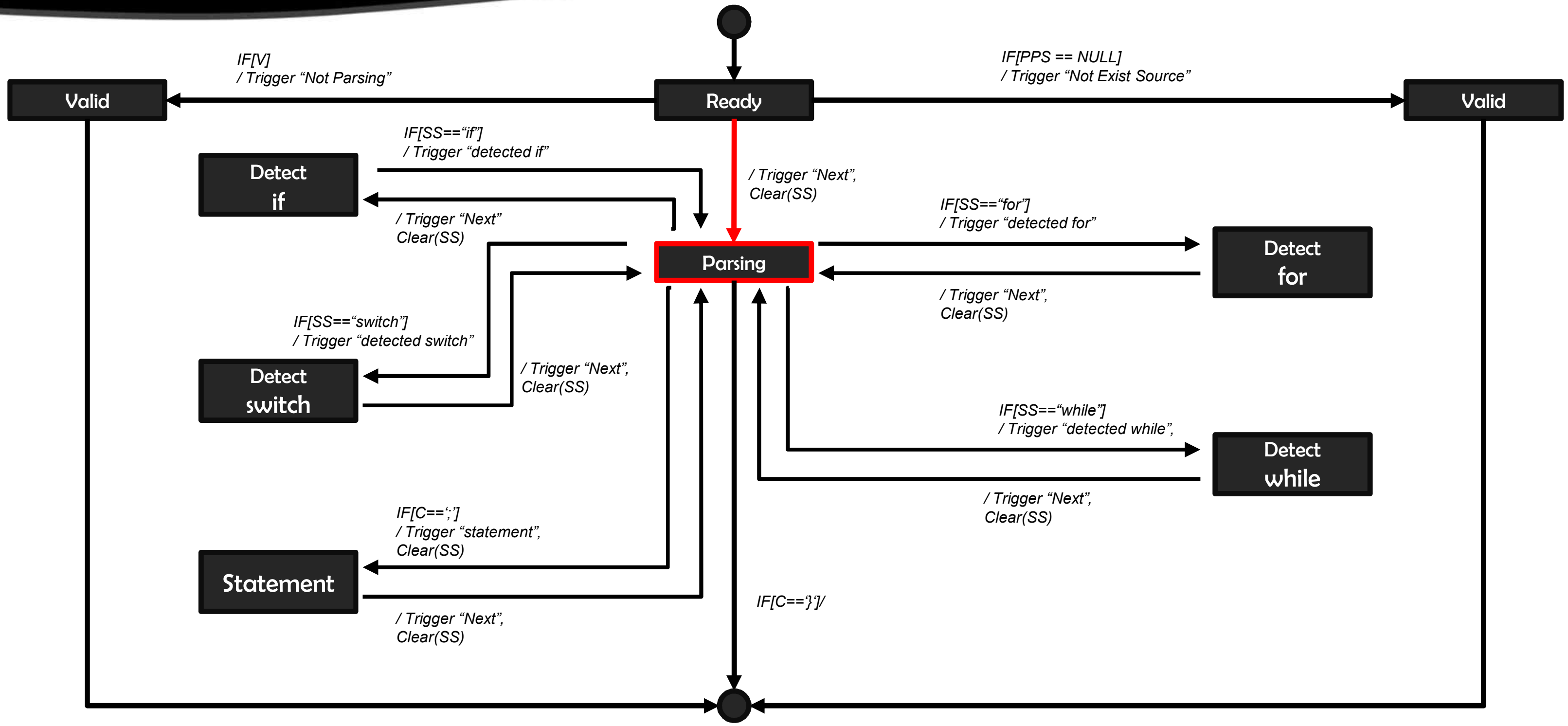


```
result_state ready(char *pre_processed_source) {
    if(!pre_processed_source)
        return not_existed_source;
    else if(!valid(pre_processed_source))
        return not_parsing;

    return none;
}
```

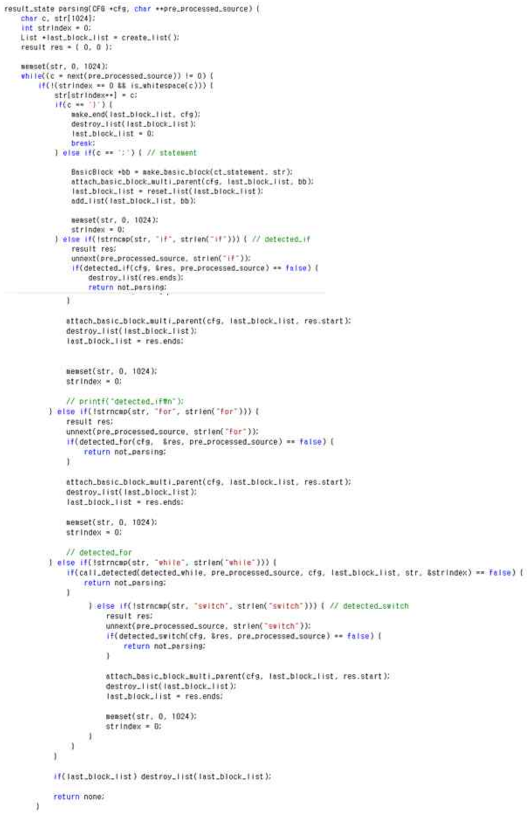
- **If Pro-processed Source is NULL, this function return not_existed_source.**

- **If Valid is NULL, this function return not_parsing.**

State
Transition Diagram – Level 4



Parsing



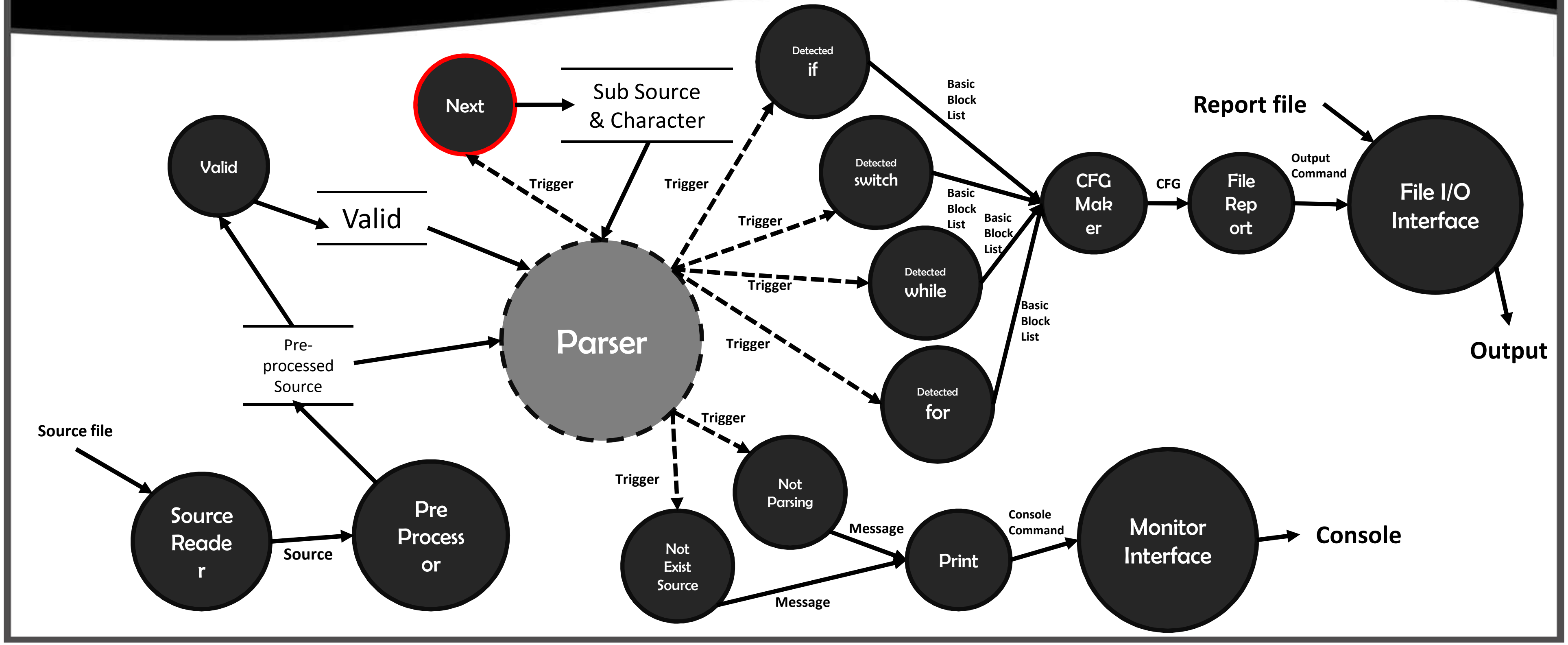
```
result_state parsing(CFG *cfg, char **pre_processed_source) {
    char c, str[1024];
    int strIndex = 0;
    List *last_block_list = create_list();
    result res = { 0, 0 };

    memset(str, 0, 1024);
    while((c = next(pre_processed_source)) != 0) {
        if(!(strIndex == 0 && is_whitespace(c))) {
            str[strIndex++] = c;
            if(c == '}') {
                make_end(last_block_list, cfg);
                destroy_list(last_block_list);
                last_block_list = 0;
                break;
            } else if(c == ';') { // statement

                BasicBlock *bb = make_basic_block(ct_statement, str);
                attach_basic_block_multi_parent(cfg, last_block_list, bb);
                last_block_list = reset_list(last_block_list);
                add_list(last_block_list, bb);

                memset(str, 0, 1024);
                strIndex = 0;
            } else if(!strncmp(str, "if", strlen("if"))) { // detected_if
                result res;
                unnext(pre_processed_source, strlen("if"));
                if(detected_if(cfg, &res, pre_processed_source) == false) {
                    destroy_list(res.ends);
                    return not_parsing;
                }

                attach_basic_block_multi_parent(cfg, last_block_list, res.start);
                destroy_list(last_block_list);
                last_block_list = res.ends;


                memset(str, 0, 1024);
                strIndex = 0;

                // printf("detected_if₩n");
            } else if(!strncmp(str, "for", strlen("for"))) {
                result res;
                unnext(pre_processed_source, strlen("for"));
                if(detected_for(cfg,  &res, pre_processed_source) == false) {
                    return not_parsing;
                }

                attach_basic_block_multi_parent(cfg, last_block_list, res.start);
                destroy_list(last_block_list);
                last_block_list = res.ends;

                memset(str, 0, 1024);
                strIndex = 0;

                // detected_for
            } else if(!strncmp(str, "while", strlen("while"))) {
                if(call_detected(detected_while, pre_processed_source, cfg, last_block_list, str, &strIndex) == false) {
                    return not_parsing;
                }

                    } else if(!strncmp(str, "switch", strlen("switch"))) { // detected_switch
                        result res;
                        unnext(pre_processed_source, strlen("switch"));
                        if(detected_switch(cfg, &res, pre_processed_source) == false) {
                            return not_parsing;
                        }

                        attach_basic_block_multi_parent(cfg, last_block_list, res.start);
                        destroy_list(last_block_list);
                        last_block_list = res.ends;

                        memset(str, 0, 1024);
                        strIndex = 0;
                    }
                }
            }

            if(last_block_list) destroy_list(last_block_list);

            return none;
        }
```

- **Parsing function gets CFG and Pre-processed Source. And the function parses the Pre-processed Source.**

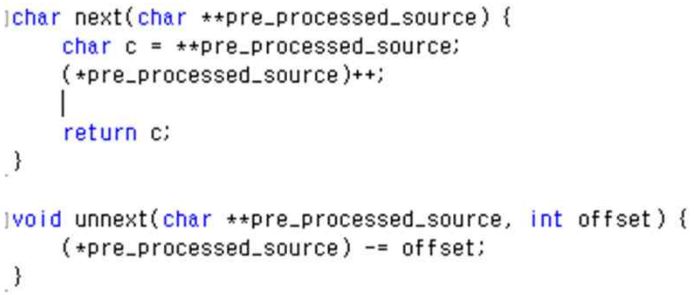
- **This function checks whether the String is if or while or for or switch from Pro-processed Source and Sub String by the prefix.**

- **After checking, This function calls appropriate detected function.**

Modified
Total Data Flow Diagram



Next, Unnext



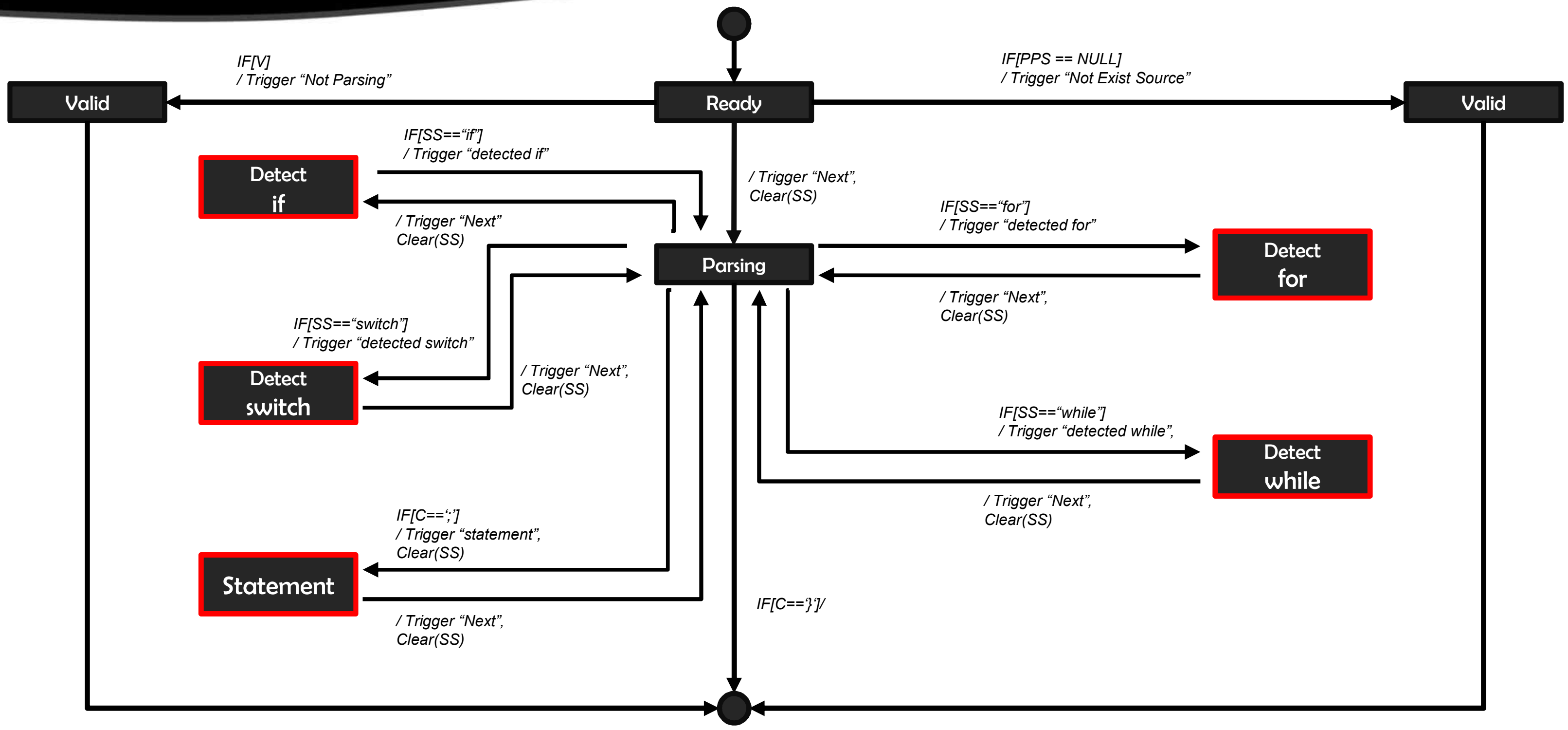
```
char next(char **pre_processed_source) {
    char c = **pre_processed_source;
    (*pre_processed_source)++;

    return c;
}

void unnext(char **pre_processed_source, int offset) {
    (*pre_processed_source) -= offset;
}
```

- This function needs for controls Pre-processed Source forward or backward.

- In case of ‘next’, the function returns character while moves forward.

- In case of ‘unnext’, the function cancels the parts as much as ‘offset’.

State Transition Diagram – Level 4



Valid

IF[V]
/ Trigger "Not Parsing"

Ready

IF[PPS == NULL]
/ Trigger "Not Exist Source"

Valid

IF[SS=="if"]
/ Trigger "detected if"

Detect if

/ Trigger "Next"
Clear(SS)

/ Trigger "Next",
Clear(SS)

IF[SS=="for"]
/ Trigger "detected for"

Parsing

Detect for

/ Trigger "Next",
Clear(SS)

IF[SS=="switch"]
/ Trigger "detected switch"

Detect switch

/ Trigger "Next",
Clear(SS)

IF[SS=="while"]
/ Trigger "detected while",

Detect while

/ Trigger "Next",
Clear(SS)

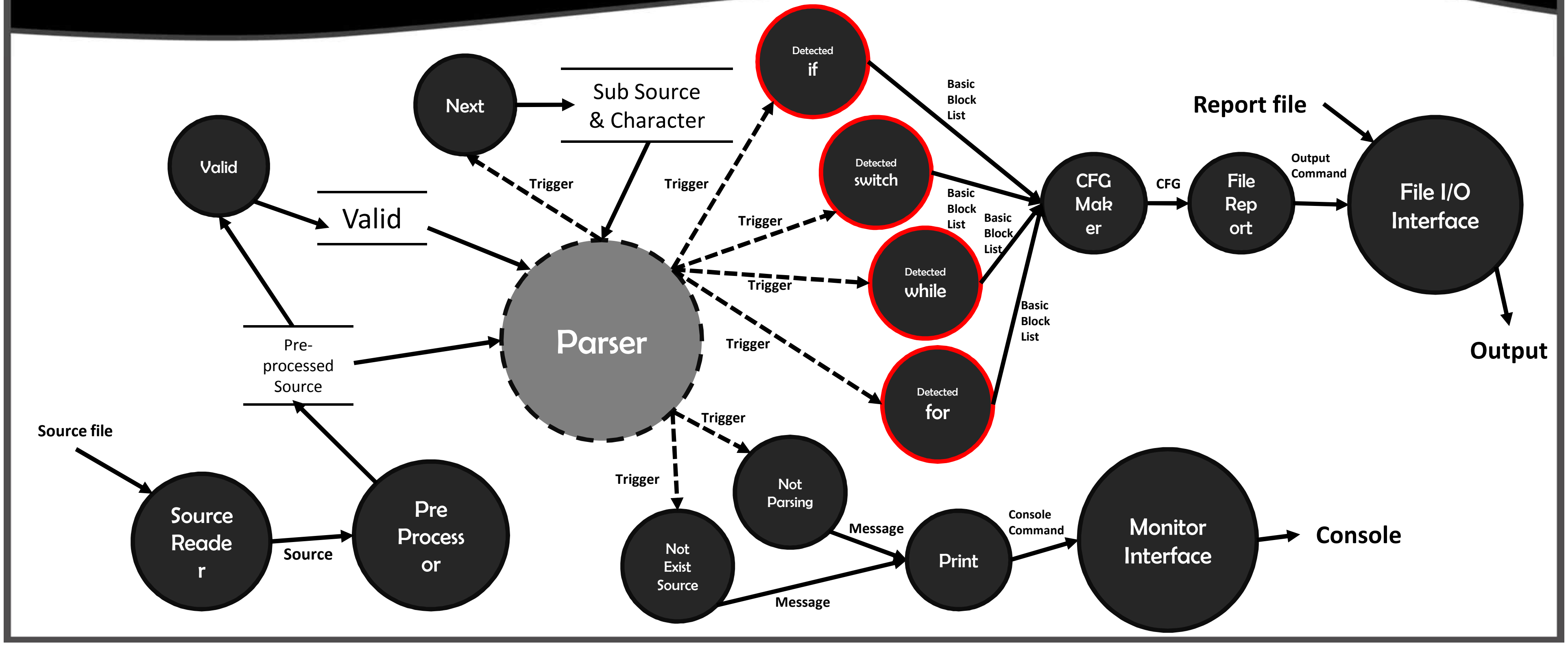
IF[C==';']
/ Trigger "statement",
Clear(SS)

Statement

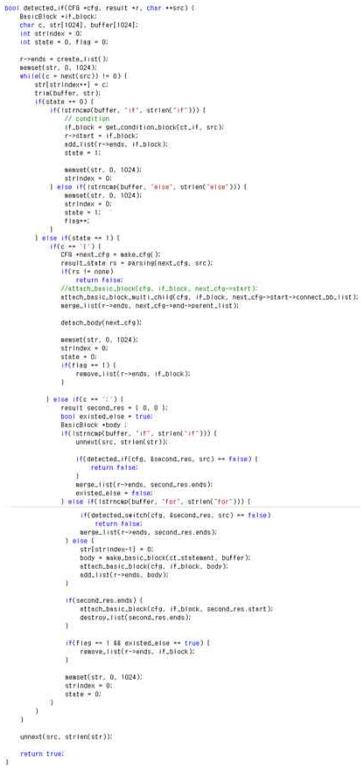
IF[C=='}']/

/ Trigger "Next",
Clear(SS)

Modified
Total Data Flow Diagram



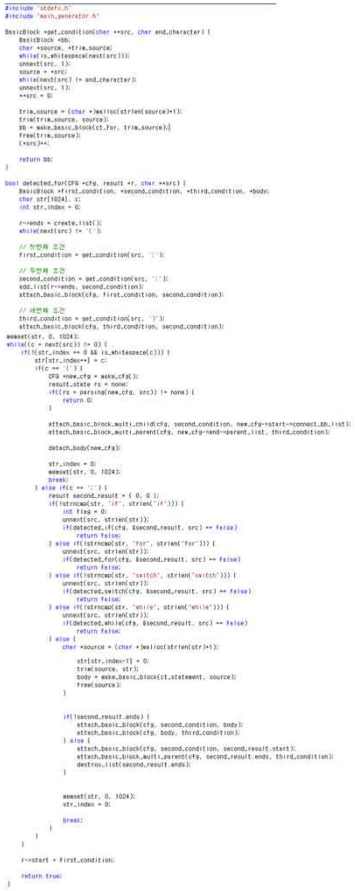
Detected If



```
bool detected_if(CFG *cfg, result *r, char **src) {
    BasicBlock *if_block;
    char c, str[1024], buffer[1024];
    int strindex = 0;
    int state = 0, flag = 0;

    r->ends = create_list();
    memset(str, 0, 1024);
    while((c = next(src)) != 0) {
        str[strindex++] = c;
        trim(buffer, str);
        if(state == 0) {
            if(!strncmp(buffer, "if", strlen("if"))) {
                // condition
                if_block = get_condition_block(ct_if, src);
                r->start = if_block;
                add_list(r->ends, if_block);
                state = 1;

                memset(str, 0, 1024);
                strindex = 0;
            } else if(!strncmp(buffer, "else", strlen("else"))) {
                memset(str, 0, 1024);
                strindex = 0;
                state = 1;
                flag++;
            }
        } else if(state == 1) {
            if(c == '{') {
               CFG *next_cfg = make_cfg();
               result_state rs = parsing(next_cfg, src);
               if(rs != none)
                   return false;
               //attach_basic_block(cfg, if_block, next_cfg->start);
               attach_basic_block_multi_child(cfg, if_block, next_cfg->start->connect_bb_list);
               merge_list(r->ends, next_cfg->end->parent_list);

               detach_body(next_cfg);

               memset(str, 0, 1024);
               strindex = 0;
               state = 0;
               if(flag == 1) {
                   remove_list(r->ends, if_block);
               }

           } else if(c == ';') {
               result second_res = { 0, 0 };
               bool existed_else = true;
               BasicBlock *body ;
               if(!strncmp(buffer, "if", strlen("if"))) {
                   unnext(src, strlen(str));

                   if(detected_if(cfg, &second_res, src) == false) {
                       return false;
                   }
                   merge_list(r->ends, second_res.ends);
                   existed_else = false;
               } else if(!strncmp(buffer, "for", strlen("for"))) {
                    if(detected_switch(cfg, &second_res, src) == false)
                        return false;
                    merge_list(r->ends, second_res.ends);
                } else {
                    str[strindex-1] = 0;
                    body = make_basic_block(ct_statement, buffer);
                    attach_basic_block(cfg, if_block, body);
                    add_list(r->ends, body);
                }

                if(second_res.ends) {
                    attach_basic_block(cfg, if_block, second_res.start);
                    destroy_list(second_res.ends);
                }

                if(flag == 1 && existed_else == true) {
                    remove_list(r->ends, if_block);
                }

                memset(str, 0, 1024);
                strindex = 0;
                state = 0;
            }
        }
    }

    unnext(src, strlen(str));

    return true;
}
```

- **When ‘If statement’ is checked, This function is called.**
- **This function is operated internally by calling detected function and parse function recursively.**
- **If parsing is failed, this function returns ‘false’.**

Detected For



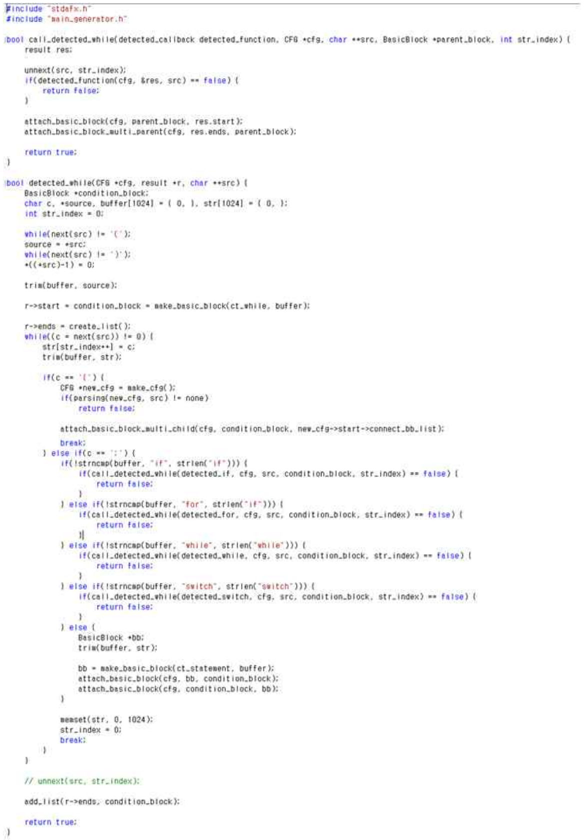
```c
#include "stdafx.h"
#include "main_generator.h"

BasicBlock *get_condition(char **src, char end_character) {
    BasicBlock *bb;
    char *source, *trim_source;
    while(is_whitespace(next(src)));
    unnext(src, 1);
    source = *src;
    while(next(src) != end_character);
    unnext(src, 1);
    **src = 0;

    trim_source = (char *)malloc(strlen(source)+1);
    trim(trim_source, source);
    bb = make_basic_block(ct_for, trim_source);
    free(trim_source);
    (*src)++;

    return bb;
}

bool detected_for(CFG *cfg, result *r, char **src) {
    BasicBlock *first_condition, *second_condition, *third_condition, *body;
    char str[1024], c;
    int str_index = 0;

    r->ends = create_list();
    while(next(src) != '(');

    // 첫번째 조건
    first_condition = get_condition(src, ';');

    // 두번째 조건
    second_condition = get_condition(src, ';');
    add_list(r->ends, second_condition);
    attach_basic_block(cfg, first_condition, second_condition);

    // 세번째 조건
    third_condition = get_condition(src, ')');
    attach_basic_block(cfg, third_condition, second_condition);
memset(str, 0, 1024);
while((c = next(src)) != 0) {
    if(!(str_index == 0 && is_whitespace(c))) {
        str[str_index++] = c;
        if(c == '{') {
            CFG *new_cfg = make_cfg();
            result_state rs = none;
            if((rs = parsing(new_cfg, src)) != none) {
                return 0;
            }

            attach_basic_block_multi_child(cfg, second_condition, new_cfg->start->connect_bb_list);
            attach_basic_block_multi_parent(cfg, new_cfg->end->parent_list, third_condition);

            detach_body(new_cfg);

            str_index = 0;
            memset(str, 0, 1024);
            break;
        } else if(c == ';') {
            result second_result = { 0, 0 };
            if(!strncmp(str, "if", strlen("if"))) {
                int flag = 0;
                unnext(src, strlen(str));
                if(detected_if(cfg, &second_result, src) == false)
                    return false;
            } else if(!strncmp(str, "for", strlen("for"))) {
                unnext(src, strlen(str));
                if(detected_for(cfg, &second_result, src) == false)
                    return false;
            } else if(!strncmp(str, "switch", strlen("switch"))) {
                unnext(src, strlen(str));
                if(detected_switch(cfg, &second_result, src) == false)
                    return false;
            } else if(!strncmp(str, "while", strlen("while"))) {
                unnext(src, strlen(str));
                if(detected_while(cfg, &second_result, src) == false)
                    return false;
            } else {
                char *source = (char *)malloc(strlen(str)+1);

                    str[str_index-1] = 0;
                    trim(source, str);
                    body = make_basic_block(ct_statement, source);
                    free(source);
                }


                if(!second_result.ends) {
                    attach_basic_block(cfg, second_condition, body);
                    attach_basic_block(cfg, body, third_condition);
                } else {
                    attach_basic_block(cfg, second_condition, second_result.start);
                    attach_basic_block_multi_parent(cfg, second_result.ends, third_condition);
                    destroy_list(second_result.ends);
                }


                memset(str, 0, 1024);
                str_index = 0;

                break;
            }
        }
    }

    r->start = first_condition;

    return true;
}
```

- When 'For statement' is checked, This function is called.

- This function use recursive call internally like 'detected If'.

- If parsing is failed, this function returns 'false'.

Detected While



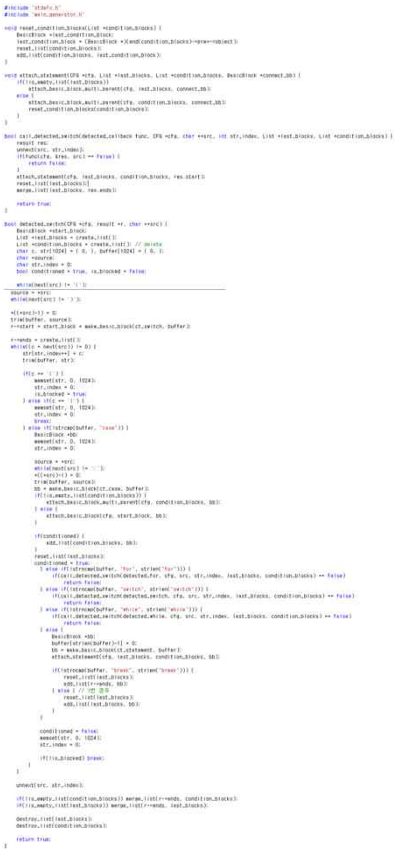
```c
#include "stdafx.h"
#include "main_generator.h"

bool call_detected_while(detected_callback detected_function, CFG *cfg, char **src, BasicBlock *parent_block, int str_index) {
    result res;

    unnext(src, str_index);
    if(detected_function(cfg, &res, src) == false) {
        return false;
    }

    attach_basic_block(cfg, parent_block, res.start);
    attach_basic_block_multi_parent(cfg, res.ends, parent_block);

    return true;
}

bool detected_while(CFG *cfg, result *r, char **src) {
    BasicBlock *condition_block;
    char c, *source, buffer[1024] = { 0, }, str[1024] = { 0, };
    int str_index = 0;

    while(next(src) != '(');
    source = *src;
    while(next(src) != ')');
    *((*src)-1) = 0;

    trim(buffer, source);

    r->start = condition_block = make_basic_block(ct_while, buffer);

    r->ends = create_list();
    while((c = next(src)) != 0) {
        str[str_index++] = c;
        trim(buffer, str);

        if(c == '{') {
            CFG *new_cfg = make_cfg();
            if(parsing(new_cfg, src) != none)
                return false;

            attach_basic_block_multi_child(cfg, condition_block, new_cfg->start->connect_bb_list);

            break;
        } else if(c == ';') {
            if(!strncmp(buffer, "if", strlen("if"))) {
                if(call_detected_while(detected_if, cfg, src, condition_block, str_index) == false) {
                    return false;
                }
            } else if(!strncmp(buffer, "for", strlen("if"))) {
                if(call_detected_while(detected_for, cfg, src, condition_block, str_index) == false) {
                    return false;
                }
            } else if(!strncmp(buffer, "while", strlen("while"))) {
                if(call_detected_while(detected_while, cfg, src, condition_block, str_index) == false) {
                    return false;
                }
            } else if(!strncmp(buffer, "switch", strlen("switch"))) {
                if(call_detected_while(detected_switch, cfg, src, condition_block, str_index) == false) {
                    return false;
                }
            } else {
                BasicBlock *bb;
                trim(buffer, str);

                bb = make_basic_block(ct_statement, buffer);
                attach_basic_block(cfg, bb, condition_block);
                attach_basic_block(cfg, condition_block, bb);
            }

            memset(str, 0, 1024);
            str_index = 0;
            break;
        }
    }

    // unnext(src, str_index);

    add_list(r->ends, condition_block);

    return true;
}
```

- When ‘While statement’ is checked, This function is called.

- This function use recursive call internally like ‘detected If’.

- If parsing is failed, this function returns ‘false’.

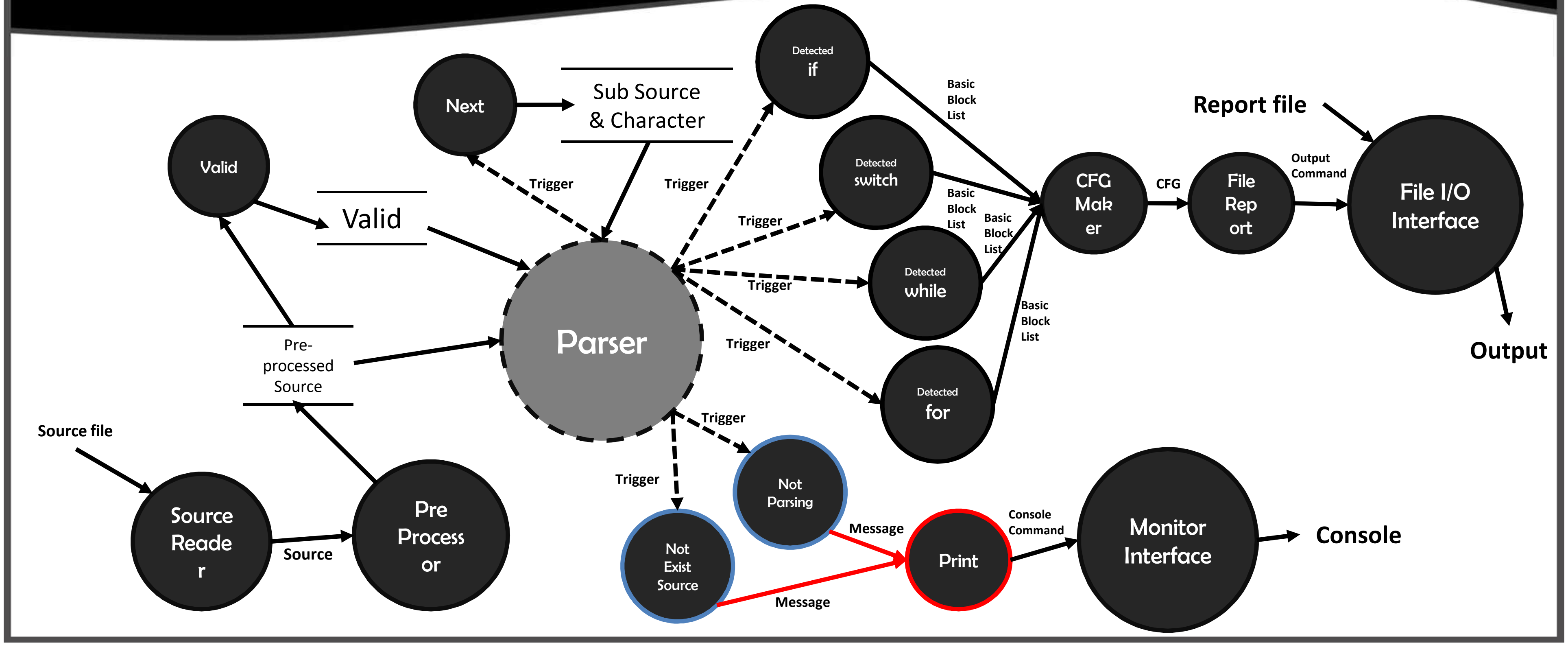
Detected Switch



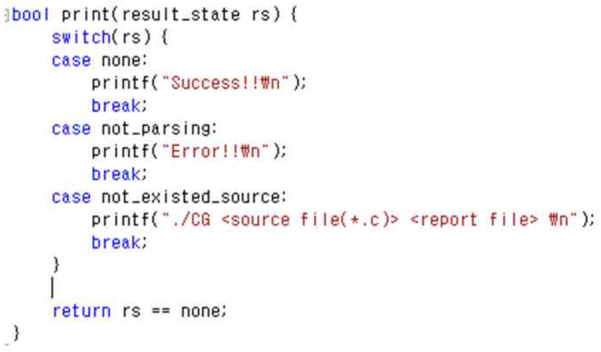
```
#include "stdafx.h"
#include "main_generator.h"

void reset_condition_blocks(List *condition_blocks) {
    BasicBlock *last_condition_block;
    last_condition_block = (BasicBlock *)(end(condition_blocks)->prev->object);
    reset_list(condition_blocks);
    add_list(condition_blocks, last_condition_block);
}

void attach_statement(CFG *cfg, List *last_blocks, List *condition_blocks, BasicBlock *connect_bb) {
    if(!is_empty_list(last_blocks))
        attach_basic_block_multi_parent(cfg, last_blocks, connect_bb);
    else {
        attach_basic_block_multi_parent(cfg, condition_blocks, connect_bb);
        reset_condition_blocks(condition_blocks);
    }
}

bool call_detected_switch(detected_callback func, CFG *cfg, char **src, int str_index, List *last_blocks, List *condition_blocks) {
    result res;
    unnext(src, str_index);
    if(func(cfg, &res, src) == false) {
        return false;
    }
    attach_statement(cfg, last_blocks, condition_blocks, res.start);
    reset_list(last_blocks);
    merge_list(last_blocks, res.ends);

    return true;
}

bool detected_switch(CFG *cfg, result *r, char **src) {
    BasicBlock *start_block;
    List *last_blocks = create_list();
    List *condition_blocks = create_list(); // [illegible]
    char c, str[1024] = { 0, }, buffer[1024] = { 0, };
    char *source;
    char str_index = 0;
    bool conditioned = true, is_blocked = false;

    while(next(src) != '(');
   source = *src;
   while(next(src) != ')');

   *((*src)-1) = 0;
   trim(buffer, source);
   r->start = start_block = make_basic_block(ct_switch, buffer);

   r->ends = create_list();
   while((c = next(src)) != 0) {
       str[str_index++] = c;
       trim(buffer, str);

       if(c == '{') {
           memset(str, 0, 1024);
           str_index = 0;
           is_blocked = true;
       } else if(c == '}') {
           memset(str, 0, 1024);
           str_index = 0;
           break;
       } else if(!strcmp(buffer, "case")) {
           BasicBlock *bb;
           memset(str, 0, 1024);
           str_index = 0;

           source = *src;
           while(next(src) != ':');
           *((*src)-1) = 0;
           trim(buffer, source);
           bb = make_basic_block(ct_case, buffer);
           if(!is_empty_list(condition_blocks)) {
               attach_basic_block_multi_parent(cfg, condition_blocks, bb);
           } else {
               attach_basic_block(cfg, start_block, bb);
           }

           if(conditioned) {
               add_list(condition_blocks, bb);
           }
           reset_list(last_blocks);
           conditioned = true;
            } else if(!strncmp(buffer, "for", strlen("for"))) {
                if(call_detected_switch(detected_for, cfg, src, str_index, last_blocks, condition_blocks) == false)
                    return false;
            } else if(!strncmp(buffer, "switch", strlen("switch"))) {
                if(call_detected_switch(detected_switch, cfg, src, str_index, last_blocks, condition_blocks) == false)
                    return false;
            } else if(!strncmp(buffer, "while", strlen("while"))) {
                if(call_detected_switch(detected_while, cfg, src, str_index, last_blocks, condition_blocks) == false)
                    return false;
            } else {
                BasicBlock *bb;
                buffer[strlen(buffer)-1] = 0;
                bb = make_basic_block(ct_statement, buffer);
                attach_statement(cfg, last_blocks, condition_blocks, bb);

                if(!strncmp(buffer, "break", strlen("break"))) {
                    reset_list(last_blocks);
                    add_list(r->ends, bb);
                } else { // [illegible]
                    reset_list(last_blocks);
                    add_list(last_blocks, bb);
                }
            }

            conditioned = false;
            memset(str, 0, 1024);
            str_index = 0;

            if(!is_blocked) break;
        }
    }

    unnext(src, str_index);

    if(!is_empty_list(condition_blocks)) merge_list(r->ends, condition_blocks);
    if(!is_empty_list(last_blocks)) merge_list(r->ends, last_blocks);

    destroy_list(last_blocks);
    destroy_list(condition_blocks);

    return true;
}
```

- When 'Switch statement' is checked, This function is called.
- This function use recursive call internally like 'detected If'.
- If parsing is failed, this function returns 'false'.

Modified Total Data Flow Diagram



Print

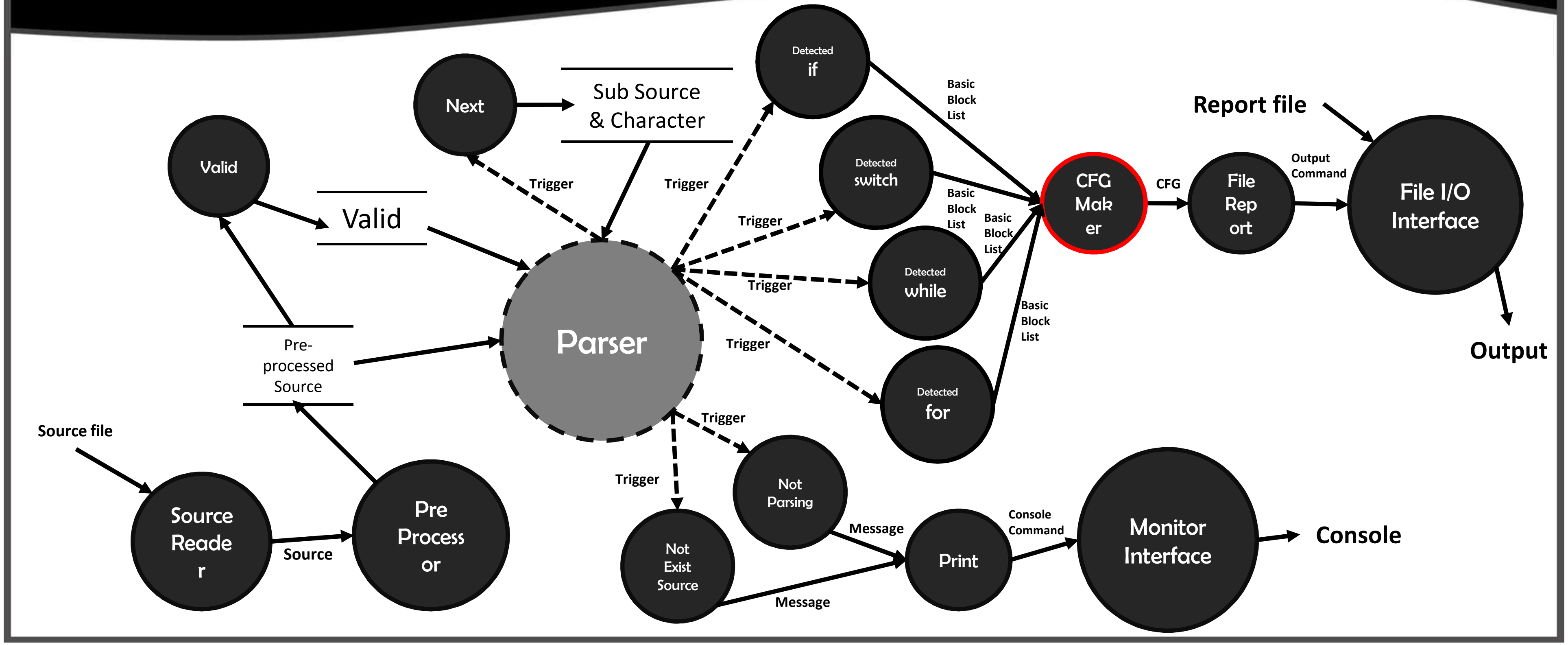


```
bool print(result_state rs) {
    switch(rs) {
    case none:
        printf("Success!!₩n");
        break;
    case not_parsing:
        printf("Error!!₩n");
        break;
    case not_existed_source:
        printf("./CG <source file(*.c)> <report file> ₩n");
        break;
    }

    return rs == none;
}
```

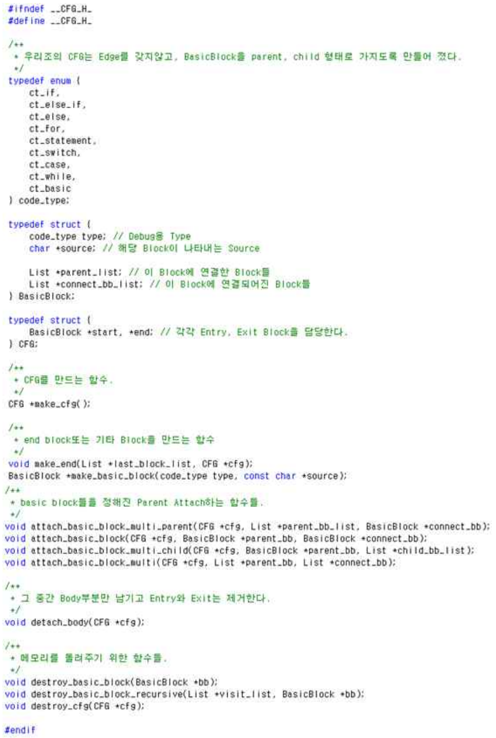
- **According to Result State(rs), This function decides Message.**

- **Case : rs = none,**
  - Success Message
- **Case : rs = not_parsing,**
  - Error Message
- **Case : rs = not_existed_source**
  - Help Message

Modified
Total Data Flow Diagram



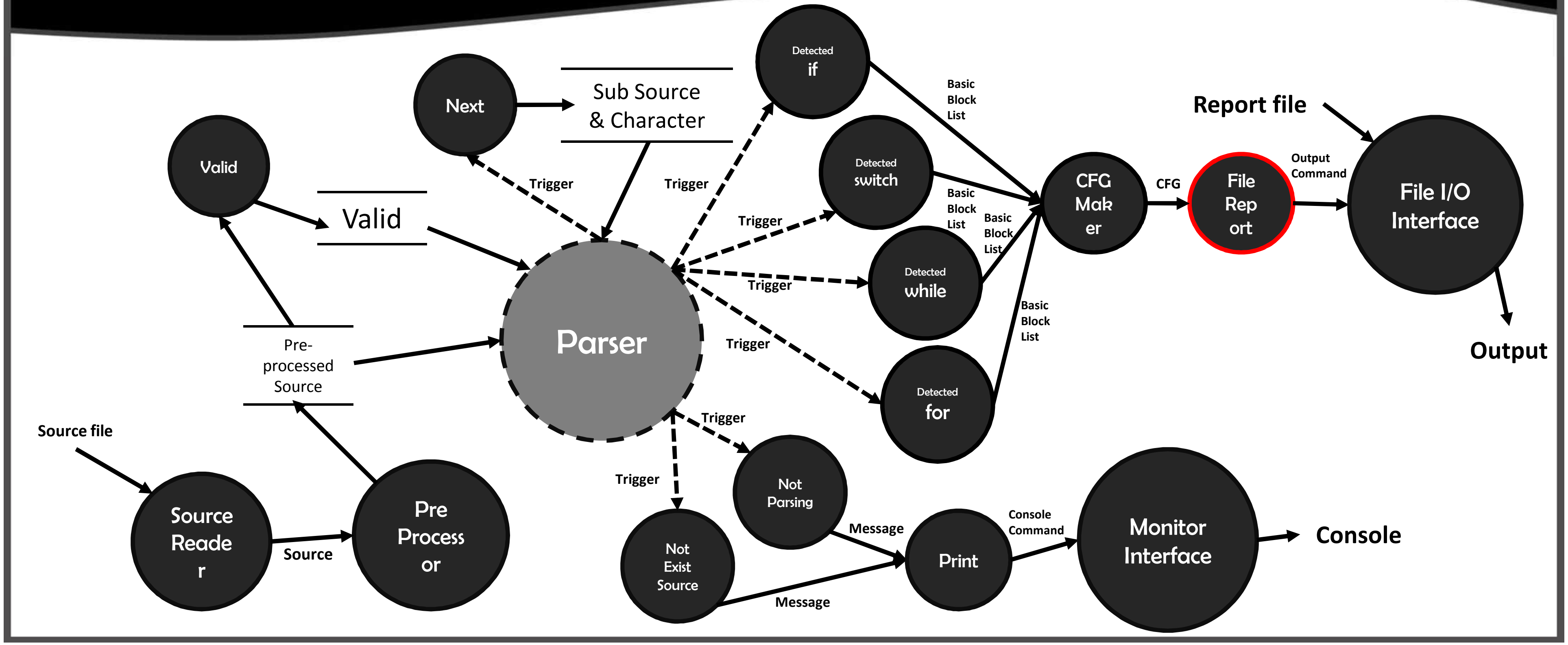
CFG Maker



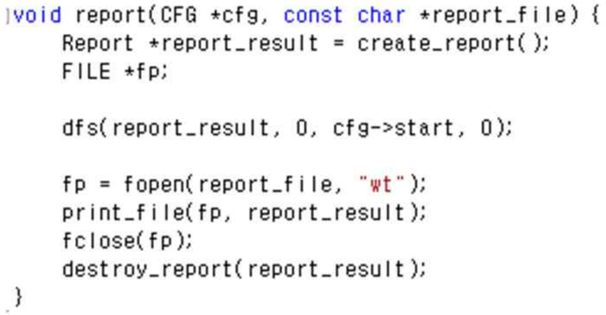
```c
#ifndef __CFG_H_
#define __CFG_H_

/**
 * 우리조의 CFG는 Edge를 갖지않고, BasicBlock을 parent, child 형태로 가지도록 만들어 졌다.
 */
typedef enum {
    ct_if,
    ct_else_if,
    ct_else,
    ct_for,
    ct_statement,
    ct_switch,
    ct_case,
    ct_while,
    ct_basic
} code_type;

typedef struct {
    code_type type; // Debug용 Type
    char *source; // 해당 Block이 나타내는 Source

    List *parent_list; // 이 Block에 연결한 Block들
    List *connect_bb_list; // 이 Block에 연결되어진 Block들
} BasicBlock;

typedef struct {
    BasicBlock *start, *end; // 각각 Entry, Exit Block을 담당한다.
} CFG;

/**
 * CFG를 만드는 함수.
 */
CFG *make_cfg( );

/**
 * end block또는 기타 Block을 만드는 함수
 */
void make_end(List *last_block_list, CFG *cfg);
BasicBlock *make_basic_block(code_type type, const char *source);
/**
 * basic block들을 정해진 Parent Attach하는 함수들.
 */
void attach_basic_block_multi_parent(CFG *cfg, List *parent_bb_list, BasicBlock *connect_bb);
void attach_basic_block(CFG *cfg, BasicBlock *parent_bb, BasicBlock *connect_bb);
void attach_basic_block_multi_child(CFG *cfg, BasicBlock *parent_bb, List *child_bb_list);
void attach_basic_block_multi(CFG *cfg, List *parent_bb, List *connect_bb);

/**
 * 그 중간 Body부분만 남기고 Entry와 Exit는 제거한다.
 */
void detach_body(CFG *cfg);

/**
 * 메모리를 돌려주기 위한 함수들.
 */
void destroy_basic_block(BasicBlock *bb);
void destroy_basic_block_recursive(List *visit_list, BasicBlock *bb);
void destroy_cfg(CFG *cfg);

#endif
```

- Our CFG hasn’t Edge.
- Alternatively, Basic Block has parent and child.
- This parts includes CFG generating function, Making end block or extra block function, Attaching basic block function, Restoring memory function/

Modified Total Data Flow Diagram



Report



```
void report(CFG *cfg, const char *report_file) {
    Report *report_result = create_report();
    FILE *fp;

    dfs(report_result, 0, cfg->start, 0);

    fp = fopen(report_file, "wt");
    print_file(fp, report_result);
    fclose(fp);
    destroy_report(report_result);
}
```

- **Because CFG hasn’t edge, We should make edge through Basic Block’s information.**

- **If linked node of parents is two or more, It’s critical edge.**

- **If parents’ ID is larger than current’s ID, It’s Back edge.**

3. Test

Q n A

THANK YOU!!