Systems Programming

Pass 1 Implementation
SIC-XE ASSEMBLER

Table of Content:

Project Description and Specifications:

Description:

The Implementation a **free formated** SIC/XE assembler, written in C++, producing code for the absolute loader. Where in phase 1 of the project Pass1 of the assembler is to be implemented so as to deliver an **intermediate** file along with **symbol** and **literal** tables that are used in pass 2 in the assembler.

Specifications

1. The pass1 is to execute by entering; pass1 <source-file-name>
2. The source file for the main program for this phase is to be named pass1.c
3. Building a parser that is capable of handling source lines that are instructions, storage declaration, comments, and assembler directives.
    a. For instructions, the parser is to minimally be capable of decoding 2, 3 and 4-byte instructions as follows:
        i. 2-byte with 1 or 2 symbolic register reference (e.g., TIXR A, ADDR S,A).
        ii. RSUB (ignoring any operand or perhaps issuing a warning).
        iii. 3-byte PC-relative with symbolic operand to include immediate, indirect, and indexed addressing.
        iv. 3-byte absolute with non-symbolic operand to include immediate, indirect, and indexed addressing.
        v. 4-byte absolute with symbolic or non-symbolic operand to include immediate, indirect, and indexed addressing.
    b. The parser is to handle all storage directives (BYTE, WORD, RESW and RESB).
4. The output of this phase should contain (at least):
    a. The symbol table.
    b. The source program in a format similar to the listing file described in your textbook except that the object code is not generated as shown below. A meaningful error message is printed below the line in which the error occurred.

Design and Data Structures:

Pass 1 implementation is mainly divided into 4 categories

Static Tables:

The directives' and operations' tables are implemented as static **Hash tables**, as only find operation is supported, unlike insertion and deletion.

Directives' and Operations' tables could be implemented either as static tables, which consumes more memory in comparison with the second implementation technique, which keeps these tables in an external file and whenever an Opcode validation is required the whole file is searched, which of course is a time consuming.

Mnemonic Opcode Table:

A singleton class that contains all Operations and their information such as:

1) Label
2) Opcode
3) Supported Format

These information is stored in a map where the key is the operation and the value is an information class that holds the previous variables.

Dynamic Tables:

Symbol table, Literal table and the group table are implemented as dynamic hash tables, giving that fact that all operations from inserting and deleting to retrieving data are supported.

The structure of all these tables are the same, as they are built using a map whose key is the label, literal and group name respectively. As for the value of the map, it is stored as a **struct** that contains the information as follows:

1) Symbol Table Information:
    a) Symbol address
    b) Symbol type (i.e. Absolute/Relative)
    c) Symbol Length
2) Literal Table Information:
    a) Literal Address
    b) Literal Length
    c) Literal Value
3) Group Table:
    a) Group Address
    b) Group Number
    c) Group Length

Each class include getters functions to retrieve all of the required informations.

Parser:

The parser classes handles each instruction line separately alongside the validation of each operand and its compatibility with the associated operation code using **Regex**.

Instruction Line Validator:

This class receives the **free formated** instruction line read from the text file and sets the line type to either one of the following:

1) Comment Line
2) Labelled line with/without operands
3) Unlabelled line with/without operands

Upon identifying the instruction line type; the label, operation, operand and comment fields are set for future getters. The class identifies the symbolic error in each field as well.

Operand Validator:

This class receives the operand and determines its type as follows:

1) Symbol
2) Immediate or indirect
3) Literal
4) Indexed Symbol
5) Expression

The output value from the getOperandType within this class is passed to operation and operand validator for further validation operations to be performed.

Operation Operand Compatibility:

It receives the operand type as well as the operations and checks their compatibility from the point of having operands which don't match the given operation such as:

1) RSUB whose operand can't be a literals as literals can't be destinations.
2) START whose operand can only be a positive 4 hex digit number.

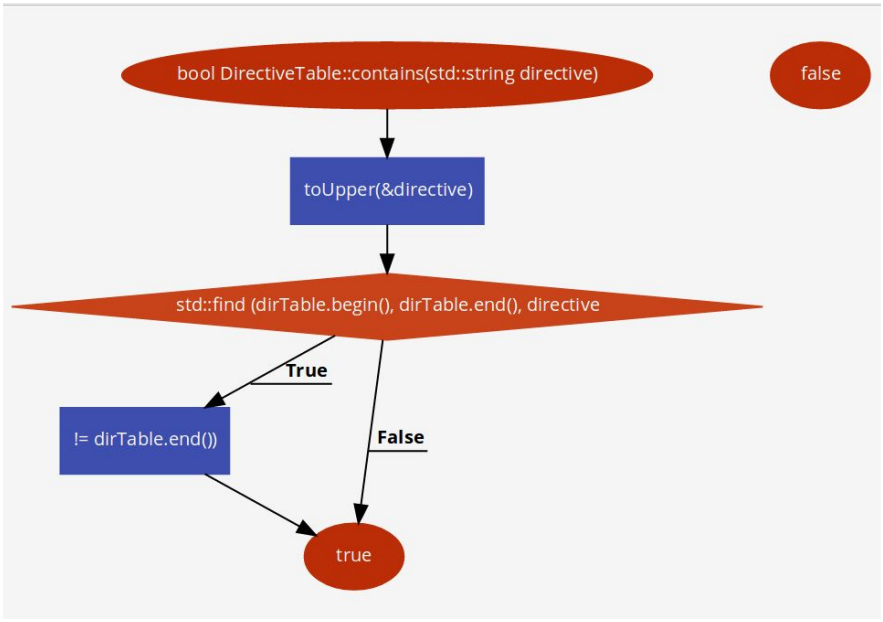
Controller:

The controller handles the overall workflow of the program, where it consists of loops and if conditions to read and check the validation of each line and generate the intermediate file, the symbol table and the literal table.

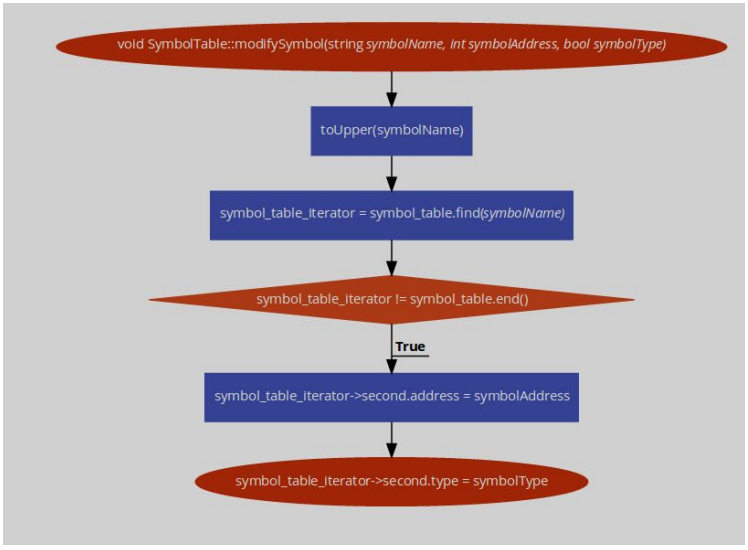
The controller generates meaningful lines' errors as well as keeping track of the locctr position and end of file.

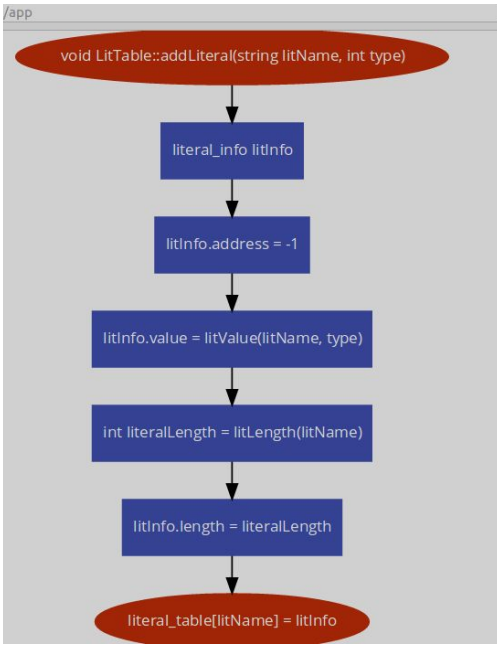
Algorithms:

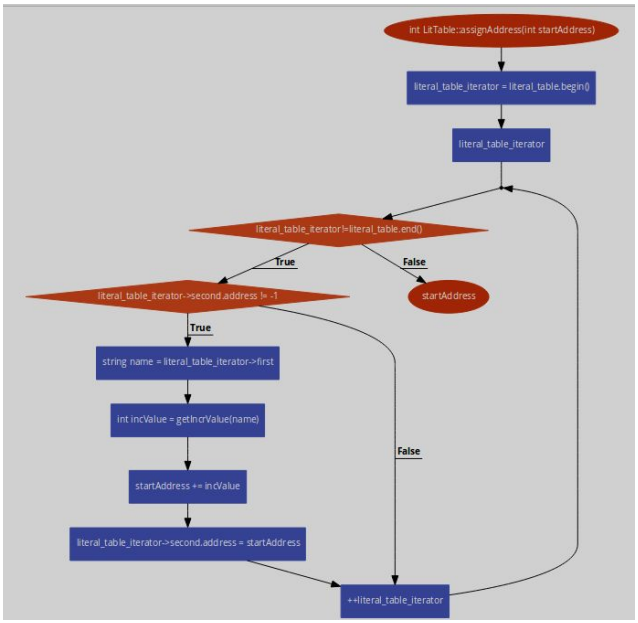
Static Tables:

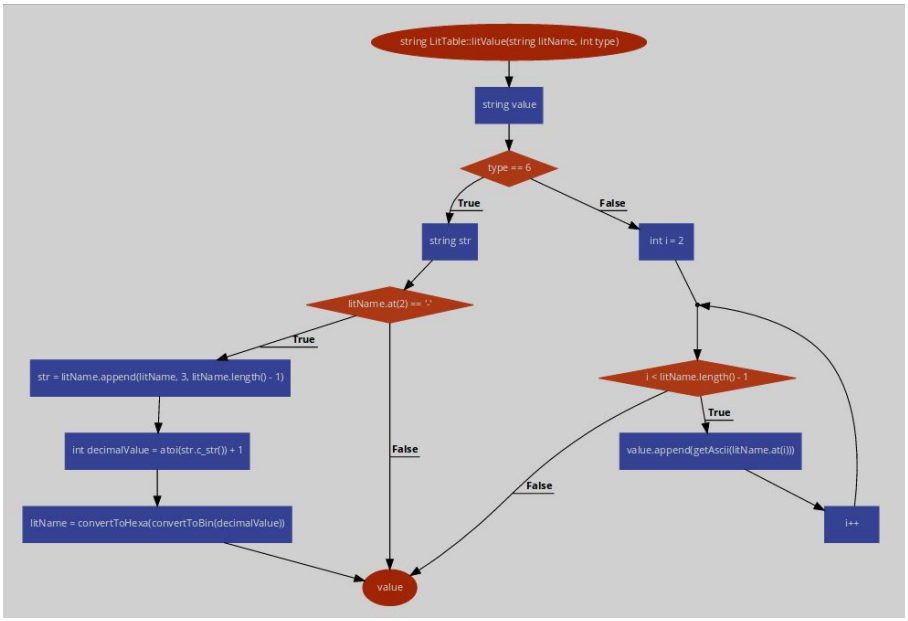
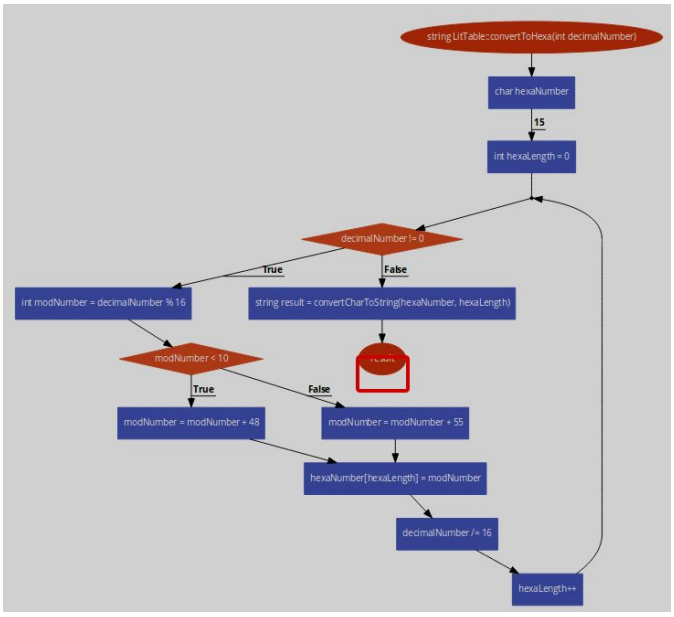
- Directive Table:
  - 

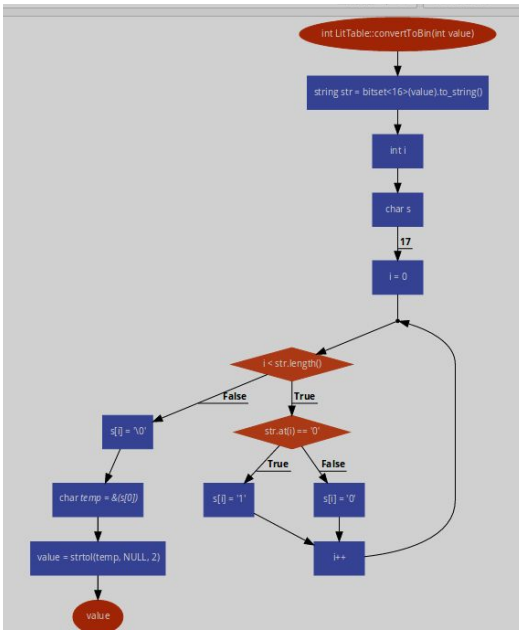
Dynamic Tables:

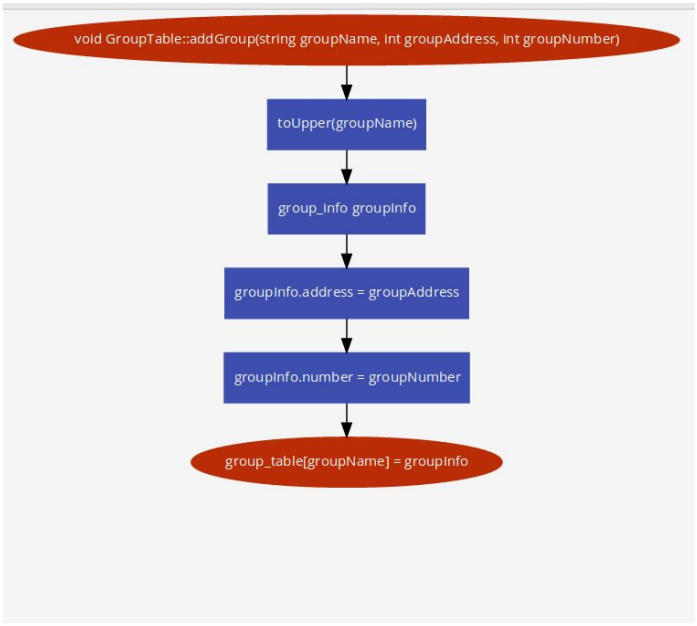
- Symbol Table:
  - 

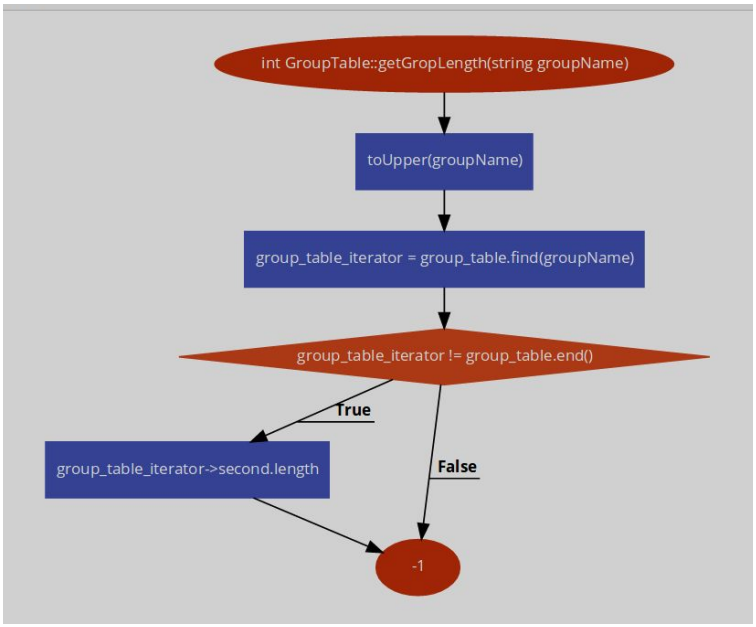
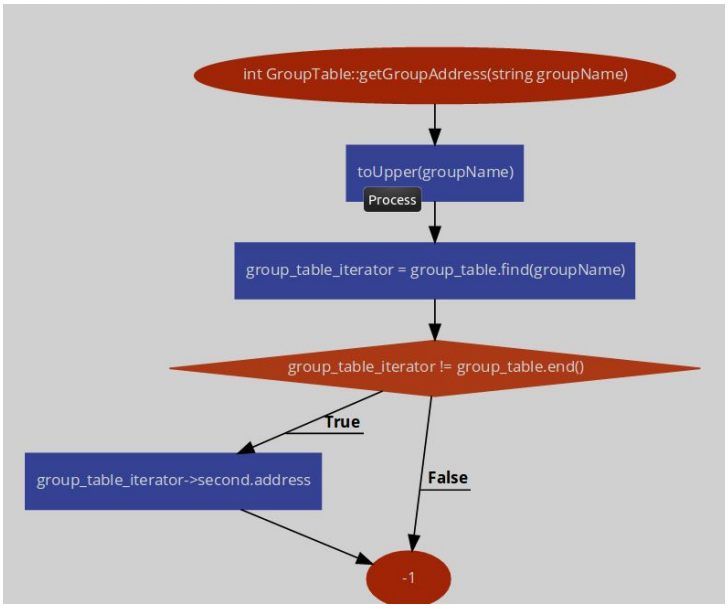
- Literal Table :
  - 

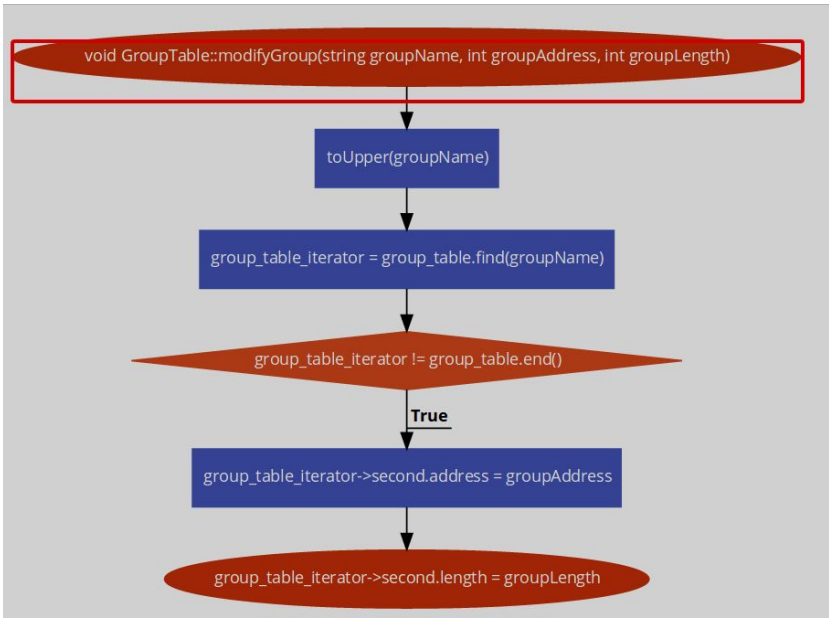
  - 

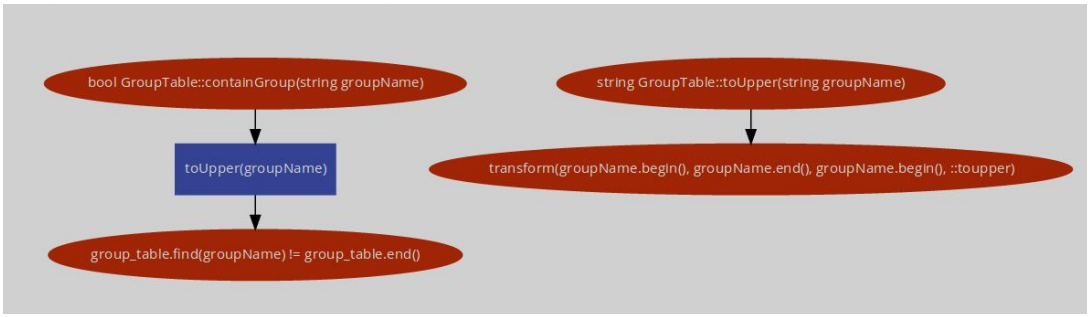
- 
string LitTable::litValue(string litName, int type)
string value
type == 6
True
False
string str
int i = 2
litName.at(2) == 'x'
True
str = litName.append(litName, 3, litName.length() - 1)
int decimalValue = atoi(str.c_str()) + 1
litName = convertToHexa(convertToBin(decimalValue))
False
i < litName.length() - 1
True
value.append(getAscii(litName.at(i)))
i++
False
value
- 
string LitTable::convertToHexa(int decimalNumber)
char hexaNumber
15
int hexaLength = 0
decimalNumber != 0
True
False
int modNumber = decimalNumber % 16
string result = convertCharToString(hexaNumber, hexaLength)
modNumber < 10
True
False
modNumber = modNumber + 48
modNumber = modNumber + 55
hexaNumber[hexaLength] = modNumber
decimalNumber /= 16
hexaLength++

- 
int LitTable::convertToBin(int value)
string str = bitset<16>(value).to_string()
int i
char s
17
i = 0
i < str.length()
False
True
s[i] = '\0'
str.at(i) == '0'
True
False
char temp = &(s[0])
s[i] = '1'
s[i] = '0'
value = strtol(temp, NULL, 2)
i++
value

- Group Table:
  - 
void GroupTable::addGroup(string groupName, int groupAddress, int groupNumber)
toUpper(groupName)
group_info groupInfo
groupInfo.address = groupAddress
groupInfo.number = groupNumber
group_table[groupName] = groupInfo

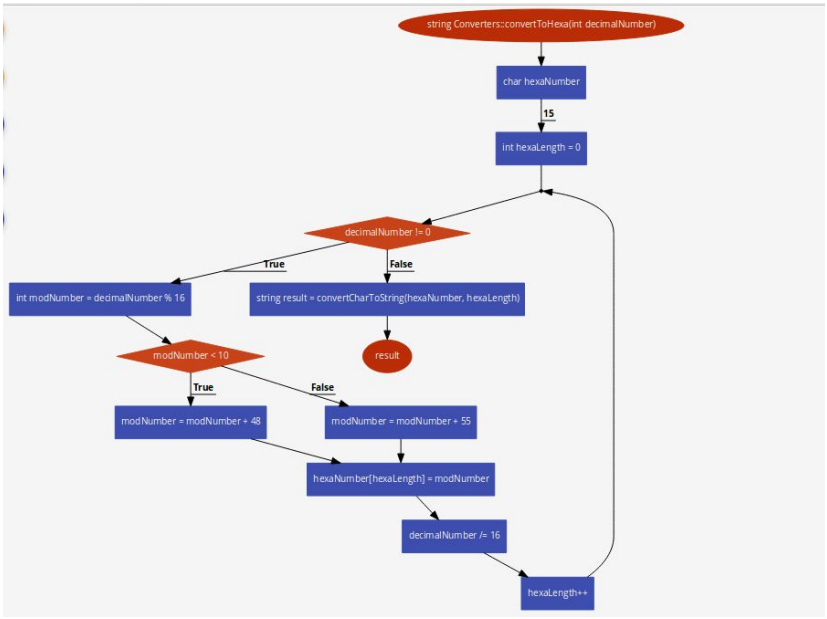
- 
int GroupTable::getGropLength(string groupName)
toUpper(groupName)
group_table_iterator = group_table.find(groupName)
group_table_iterator != group_table.end()
True
False
group_table_iterator->second.length
-1
- 
int GroupTable::getGroupAddress(string groupName)
toUpper(groupName)
Process
group_table_iterator = group_table.find(groupName)
group_table_iterator != group_table.end()
True
False
group_table_iterator->second.address
-1
-

- 

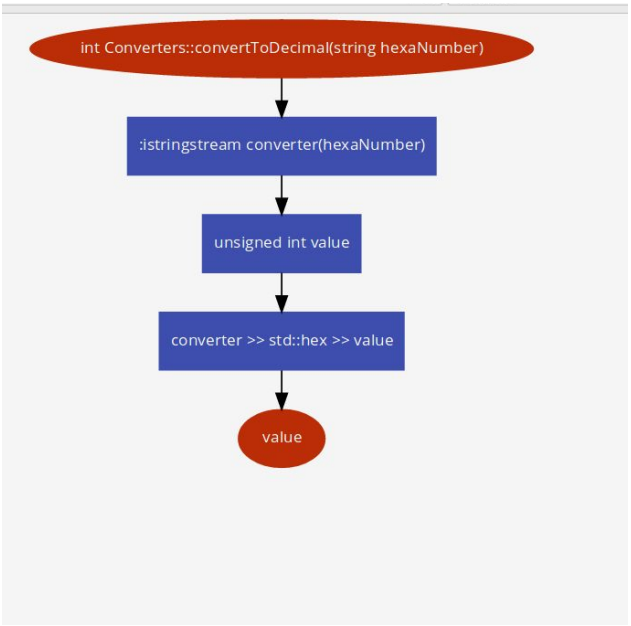
- 

Converter:

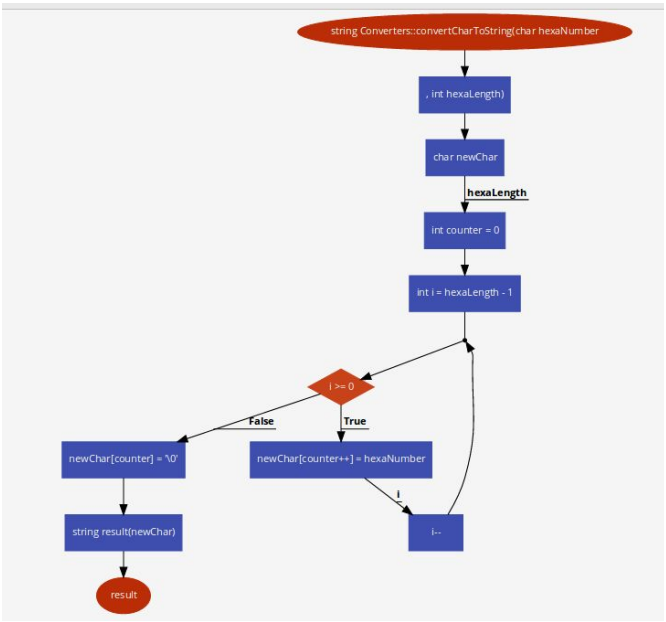
1. **Convert To Hexa:**



Convert To Decimal :

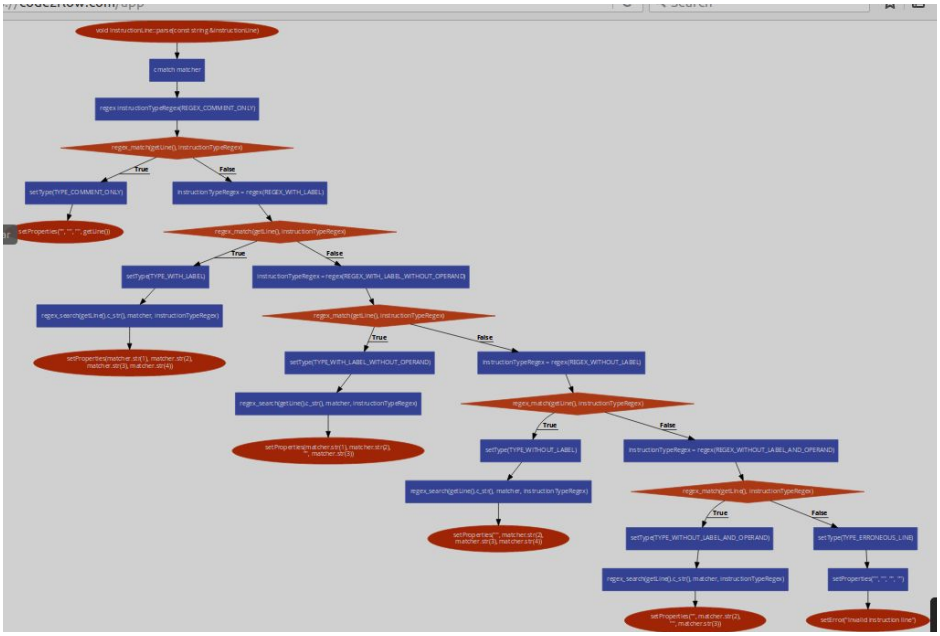


Convert Char To String:

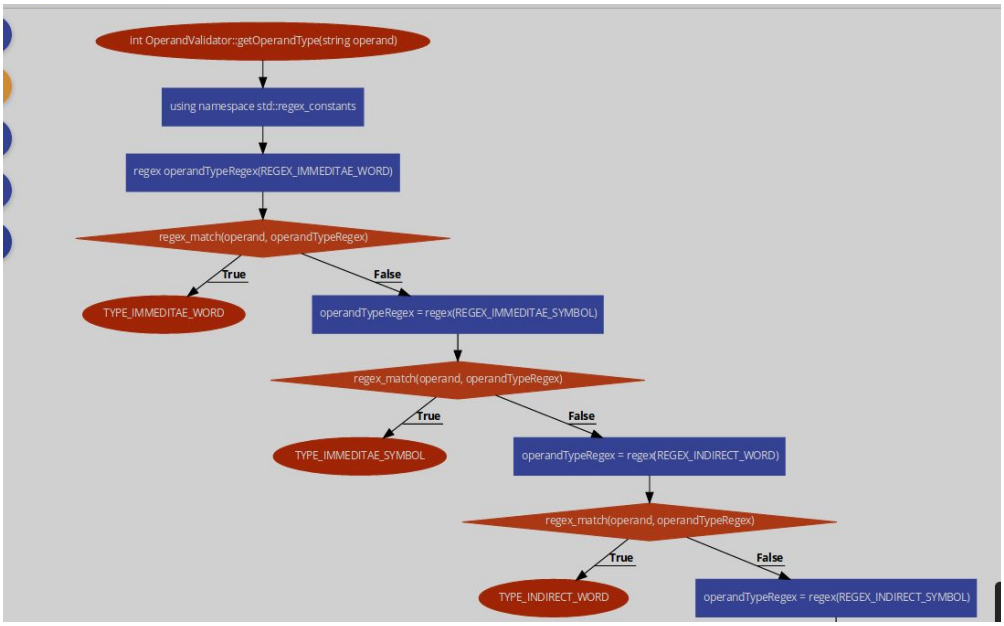


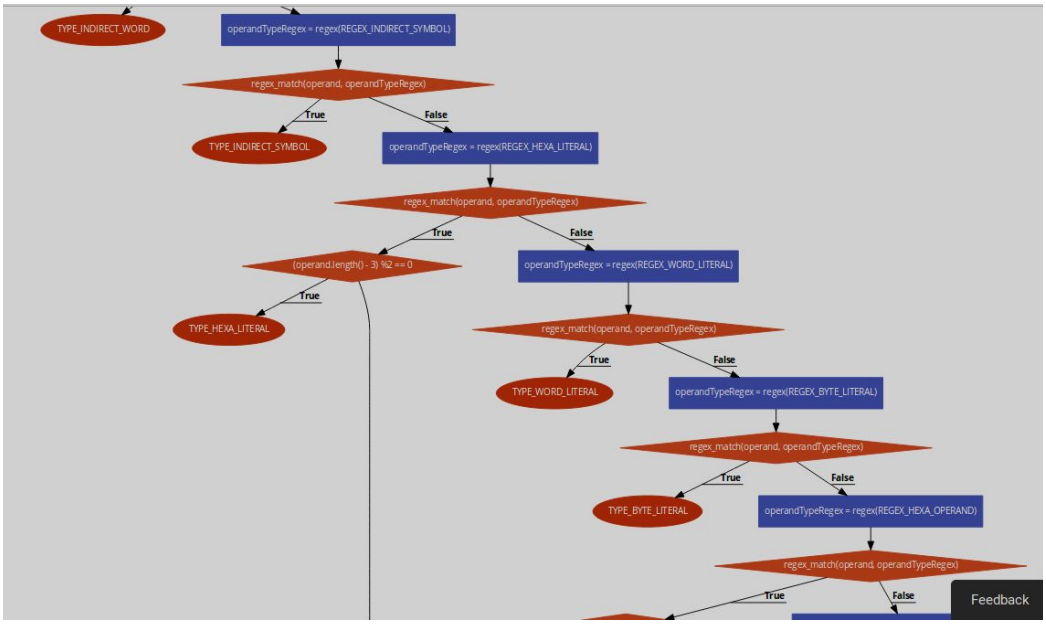
Instruction Line :

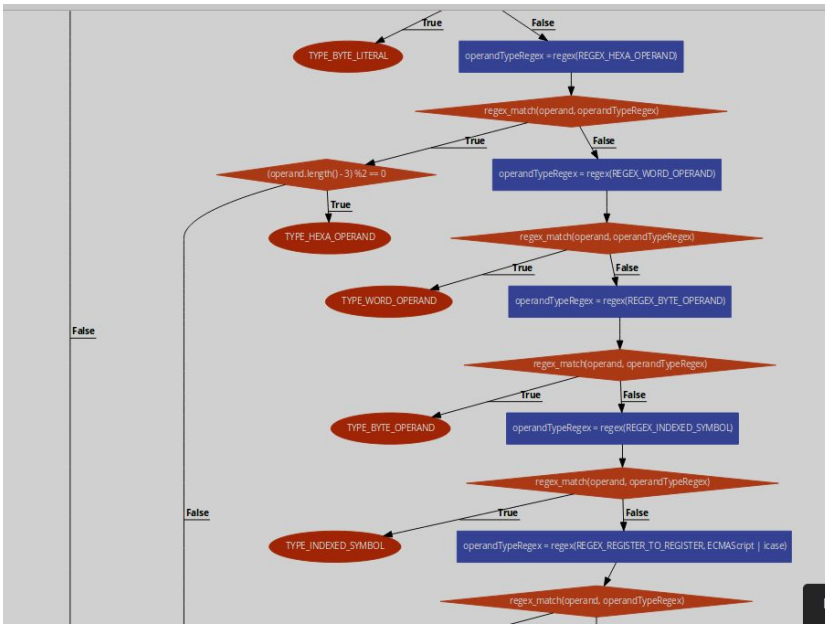
Regex :

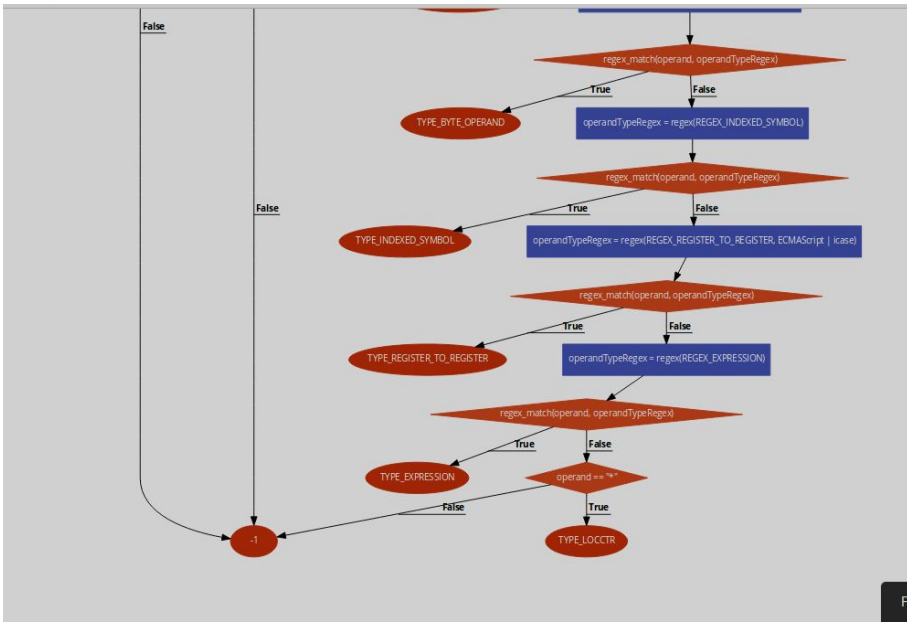


Validator:

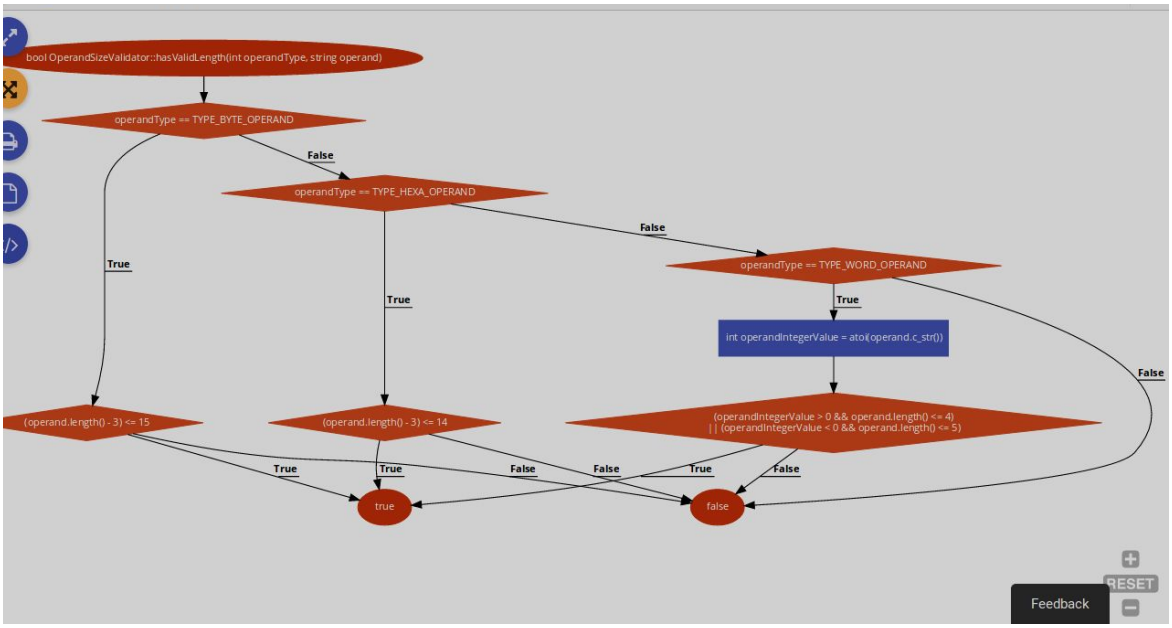
- Operand Validator:
  - 

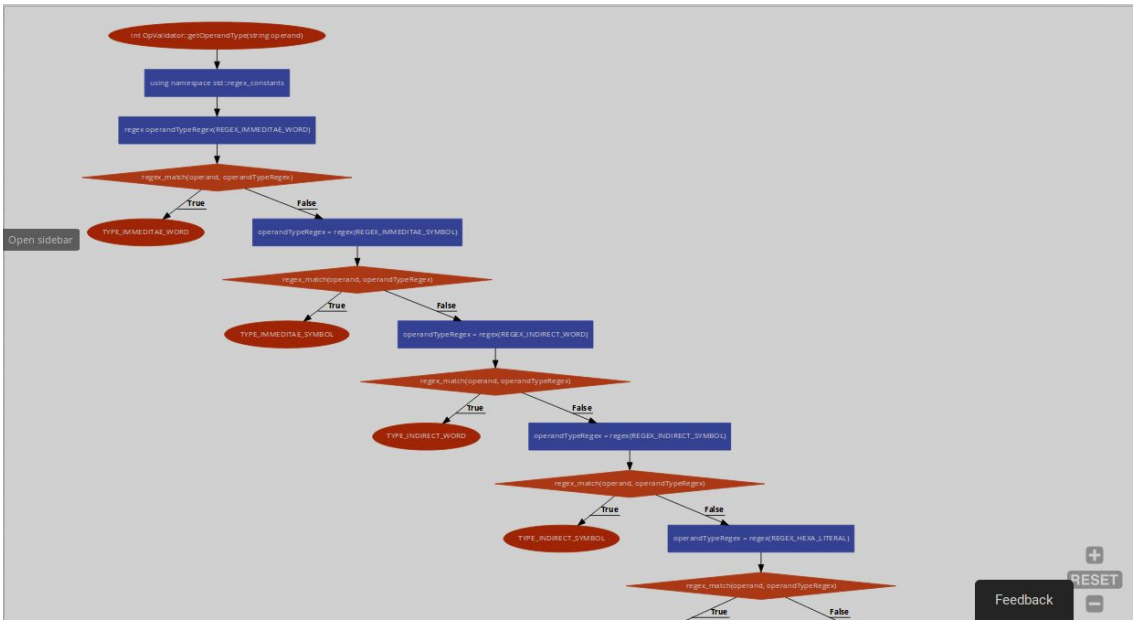
  - 

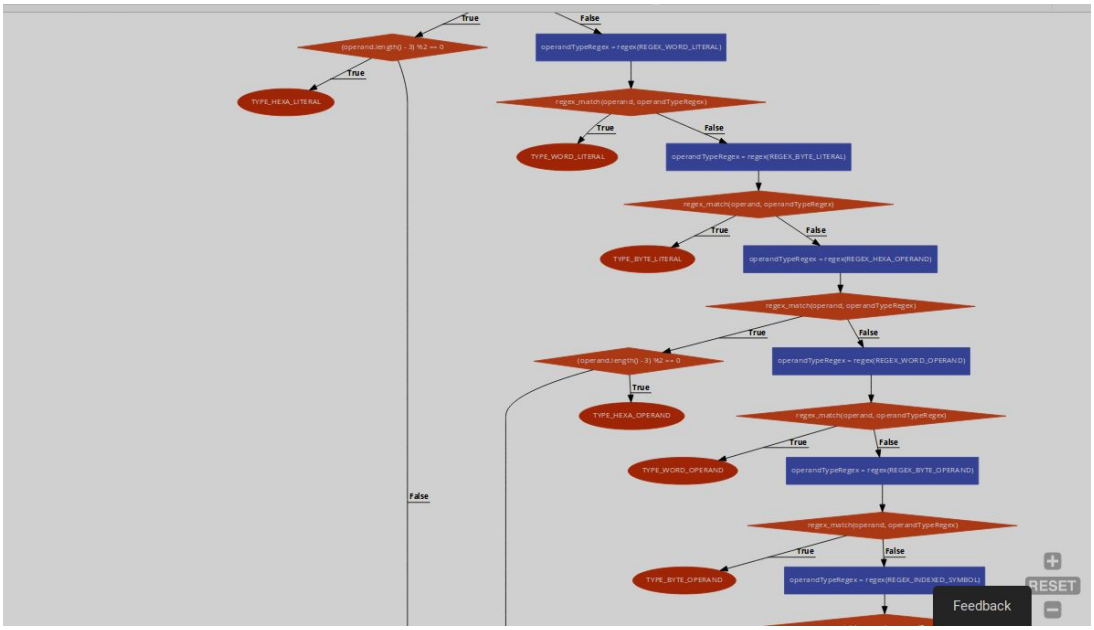
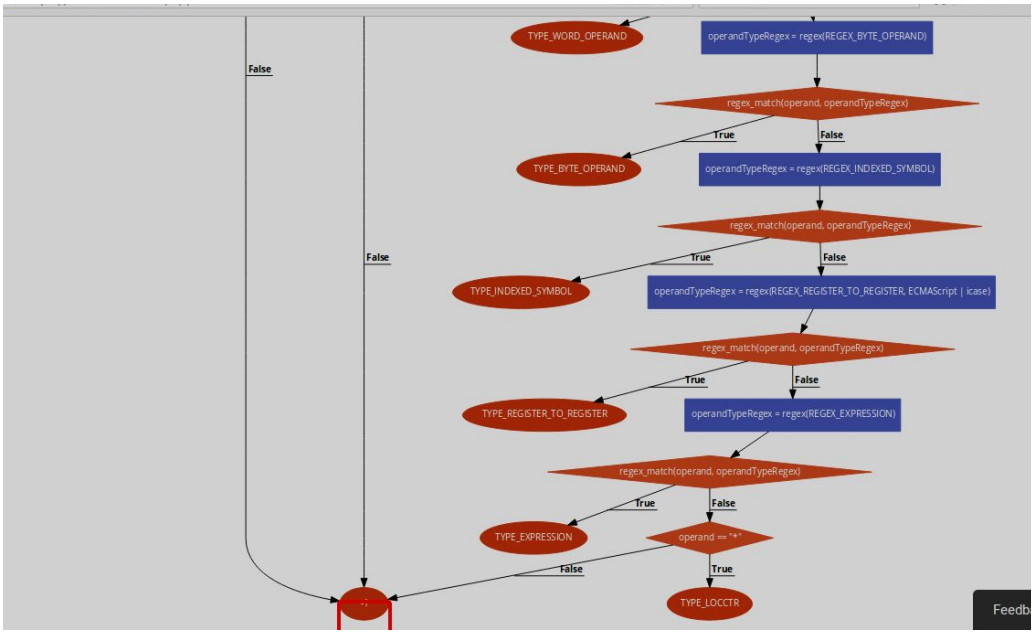
- 

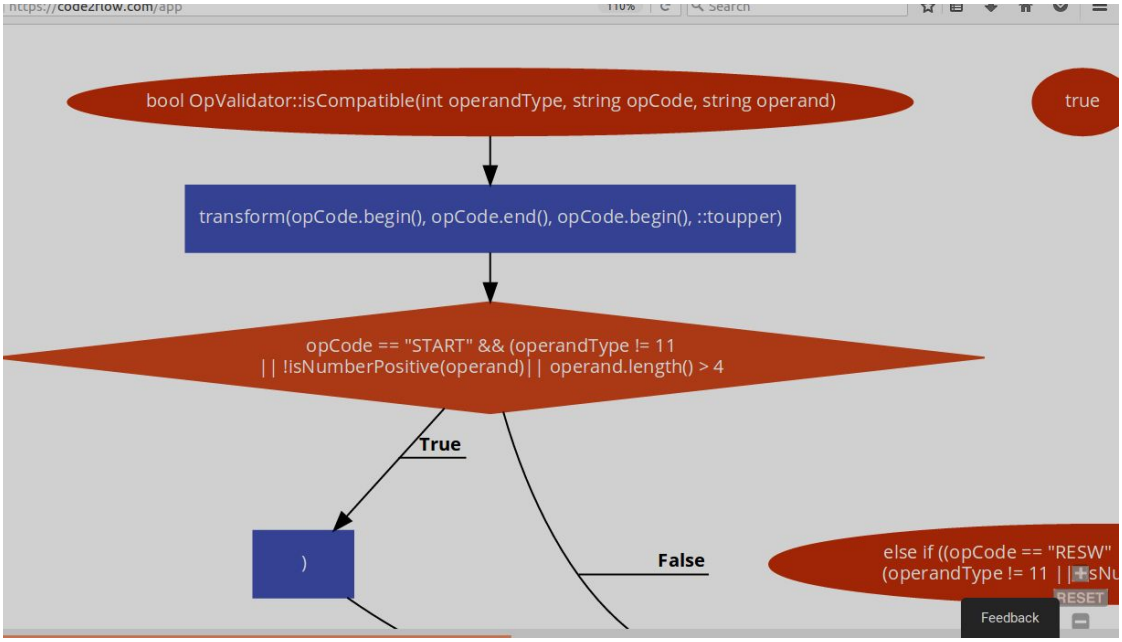
- 

- Operand size validator:

- 

- Operation code:
  - 

- 
True
False
(operand.length() - 3) %2 == 0
operandTypeRegex = regex(REGEX_WORD_LITERAL)
True
TYPE_HEXA_LITERAL
regex_match(operand, operandTypeRegex)
True
False
TYPE_WORD_LITERAL
operandTypeRegex = regex(REGEX_BYTE_LITERAL)
regex_match(operand, operandTypeRegex)
True
False
TYPE_BYTE_LITERAL
operandTypeRegex = regex(REGEX_HEXA_OPERAND)
regex_match(operand, operandTypeRegex)
True
False
(operand.length() - 3) %2 == 0
operandTypeRegex = regex(REGEX_WORD_OPERAND)
True
TYPE_HEXA_OPERAND
regex_match(operand, operandTypeRegex)
True
False
TYPE_WORD_OPERAND
operandTypeRegex = regex(REGEX_BYTE_OPERAND)
False
regex_match(operand, operandTypeRegex)
True
False
TYPE_BYTE_OPERAND
operandTypeRegex = regex(REGEX_INDEXED_SYMBOL)
RESET
Feedback
- 
TYPE_WORD_OPERAND
operandTypeRegex = regex(REGEX_BYTE_OPERAND)
False
regex_match(operand, operandTypeRegex)
True
False
TYPE_BYTE_OPERAND
operandTypeRegex = regex(REGEX_INDEXED_SYMBOL)
False
regex_match(operand, operandTypeRegex)
True
False
TYPE_INDEXED_SYMBOL
operandTypeRegex = regex(REGEX_REGISTER_TO_REGISTER, ECMAScript | icase)
regex_match(operand, operandTypeRegex)
True
False
TYPE_REGISTER_TO_REGISTER
operandTypeRegex = regex(REGEX_EXPRESSION)
regex_match(operand, operandTypeRegex)
True
False
TYPE_EXPRESSION
operand == "*"
False
True
TYPE_LOCCTR
Feedb

- 

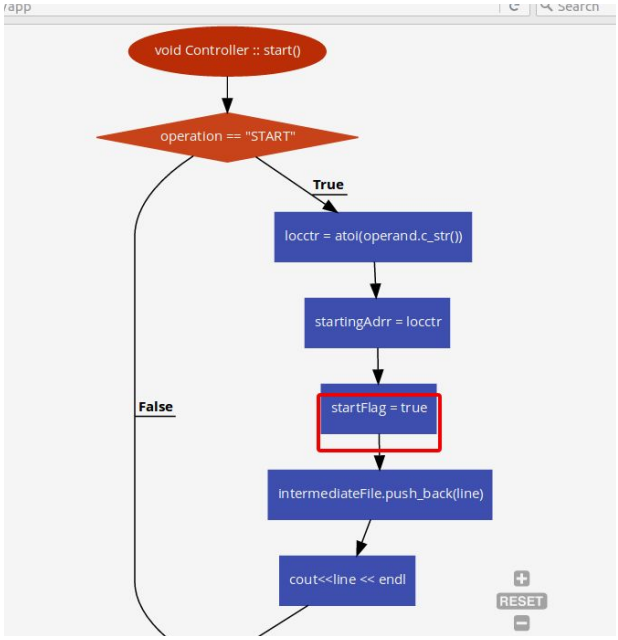
Controller:

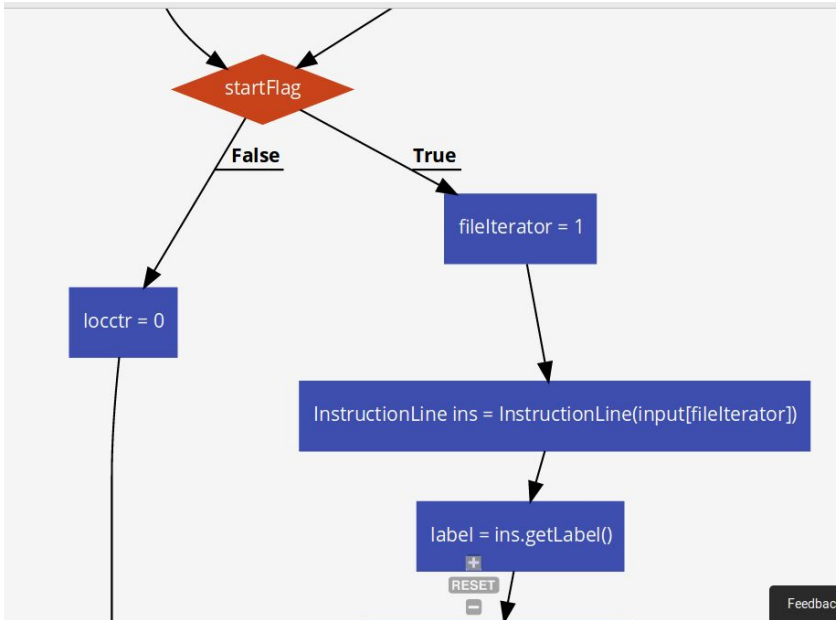
Pass 1:

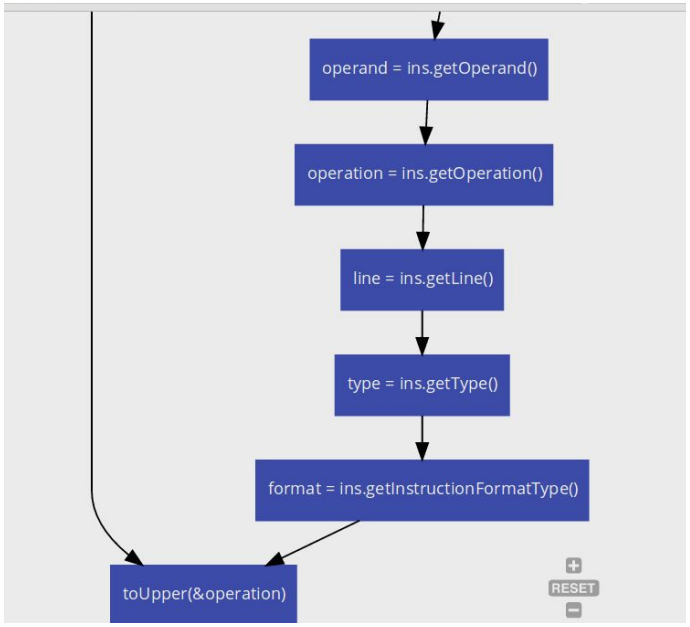
```
begin
  read first input line
  if OPCODE = 'START' then
     begin
        save #[OPERAND] as starting address
        initialize LOCCTR to starting address
        write line to intermediate file
        read next input line
     end {if START}
  else
     initialize LOCCTR to 0
  while OPCODE ≠ 'END' do
     begin
        if this is not a comment line then
           begin
              if there is a symbol in the LABEL field then
                 begin
```

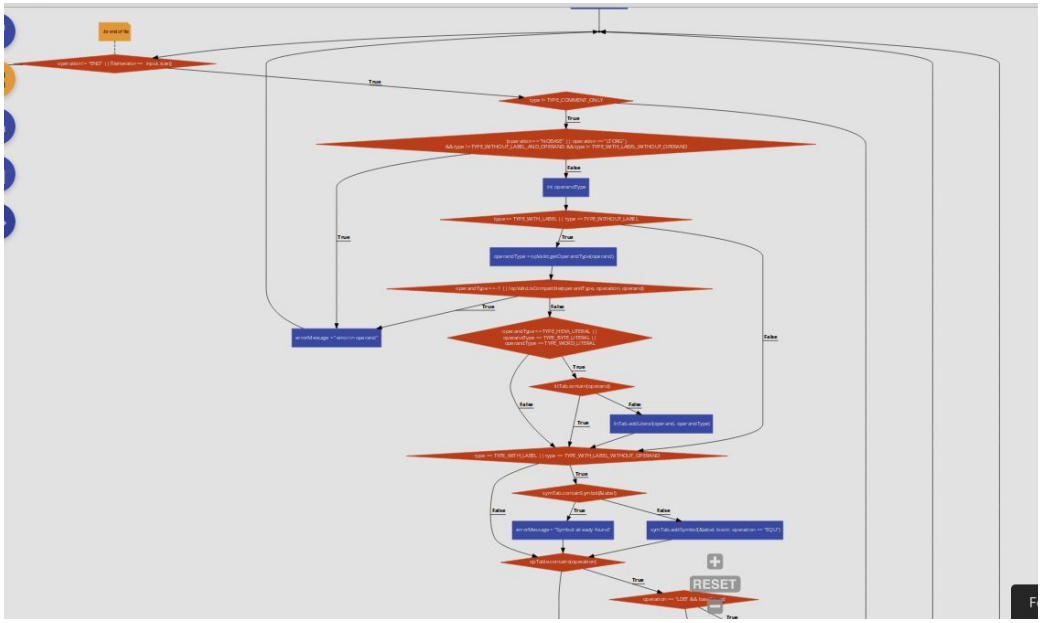
```
                search SYMTAB for LABEL
                if found then
                    set error flag (duplicate symbol)
                else
                    insert (LABEL,LOCCTR) into SYMTAB
            end {if symbol}
        search OPTAB for OPCODE
        if found then
            add 3 {instruction length} to LOCCTR
        else if OPCODE = 'WORD' then
            add 3 to LOCCTR
        else if OPCODE = 'RESW' then
            add 3 * #[OPERAND] to LOCCTR
        else if OPCODE = 'RESB' then
            add #[OPERAND] to LOCCTR

        else if OPCODE  = 'BYTE' then
            begin
                find length of constant in bytes
                add length to LOCCTR
            end {if BYTE}
        else
            set error flag (invalid operation code)
    end {if not a comment}
    write line to intermediate file
    read next input line
end {while not END}
write last line to intermediate file
save (LOCCTR - starting address) as program length
end {Pass 1}
```

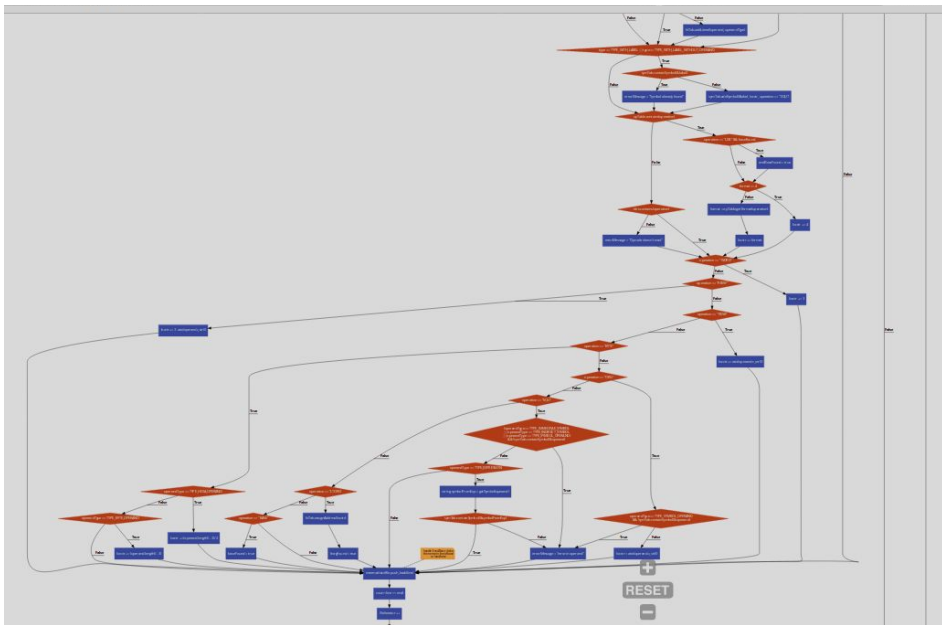
Start :

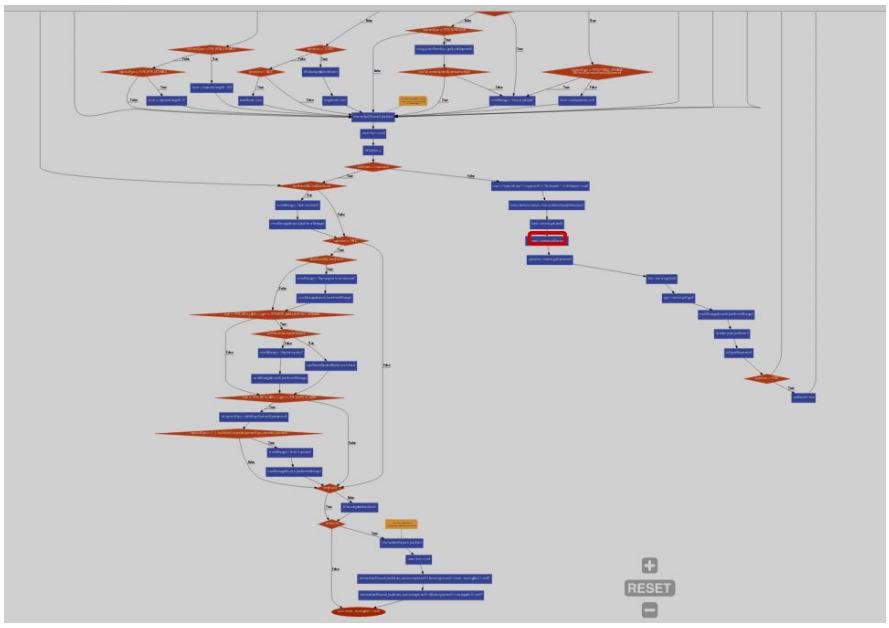

void Controller :: start()
operation == "START"
True
locctr = atoi(operand.c_str())
startingAdrr = locctr
False
startFlag = true
intermediateFile.push_back(line)
cout<<line << endl
RESET


startFlag
False
True
fileIterator = 1
locctr = 0
InstructionLine ins = InstructionLine(input[fileIterator])
label = ins.getLabel()
RESET

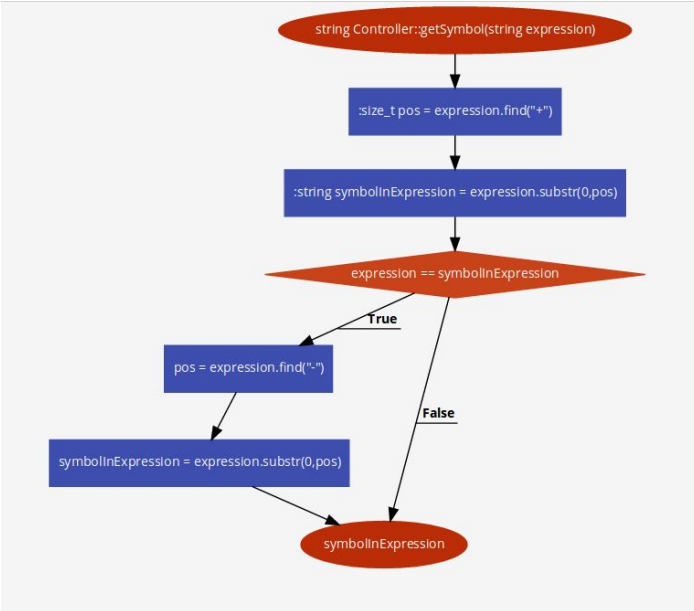

operand = ins.getOperand()
operation = ins.getOperation()
line = ins.getLine()
type = ins.getType()
format = ins.getInstructionFormatType()
toUpper(&operation)
RESET


RESET

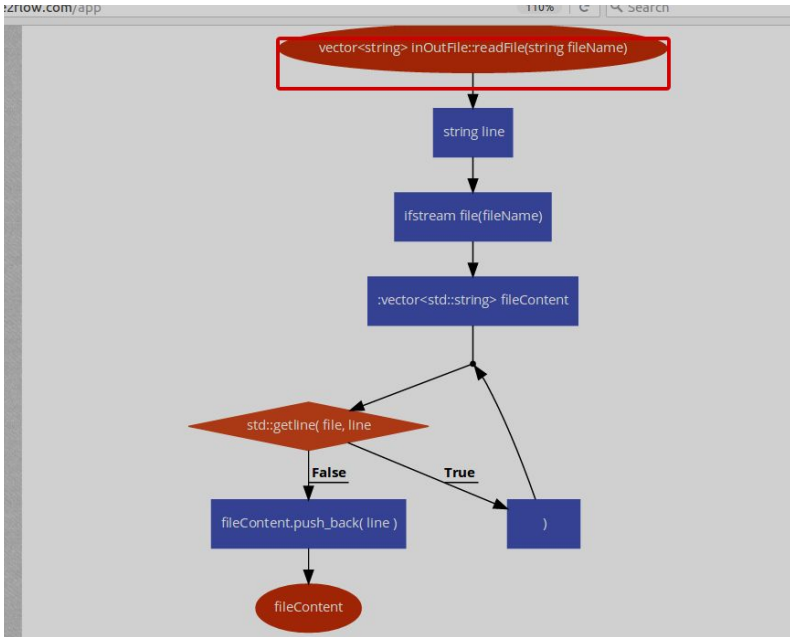




Get Symbol:



INPUT DATA :

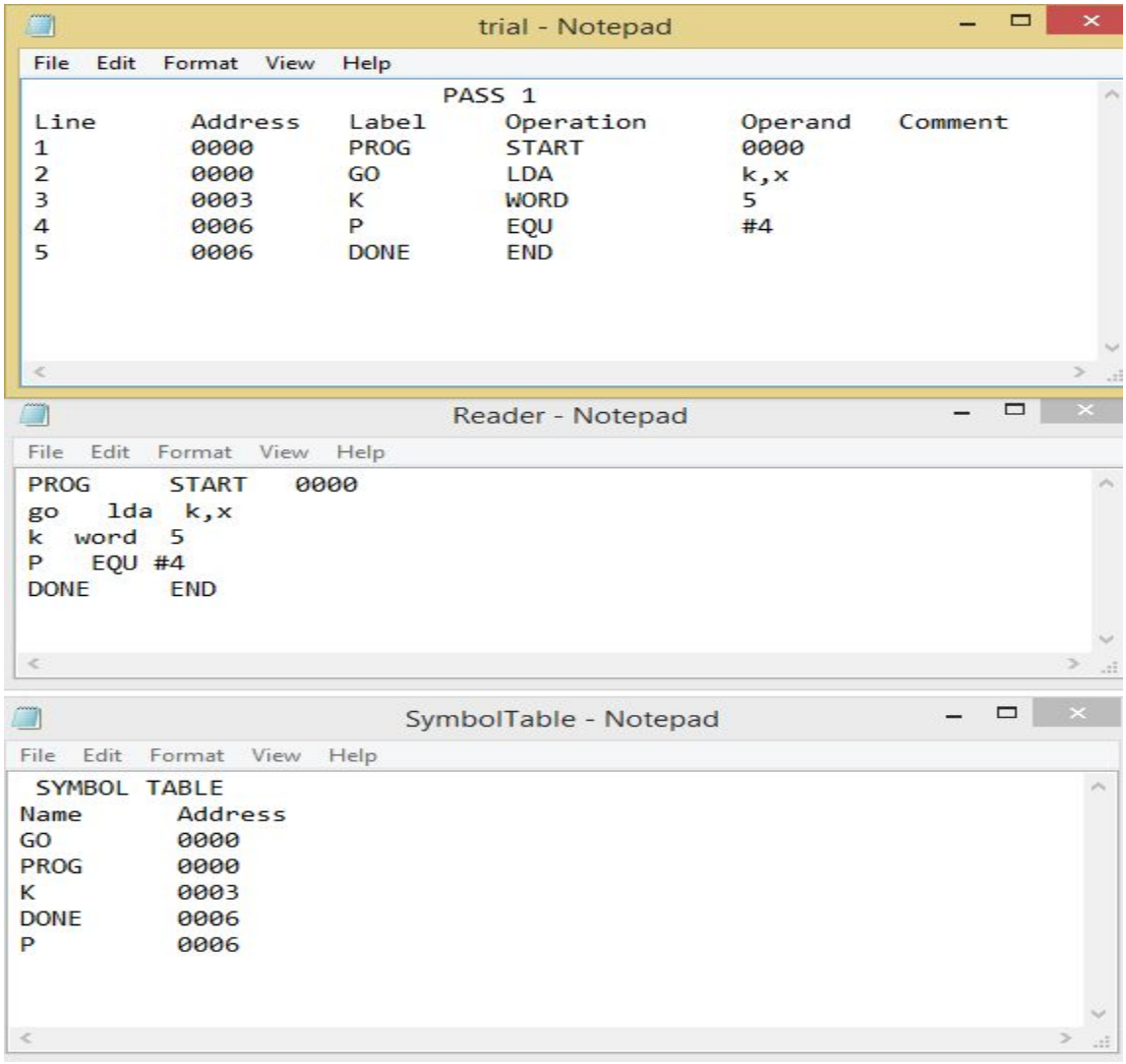


Assumptions:

1. Read File read only a file exists on the directory and not null string or directory.
2. The is no empty line every line should contain at least on field.

3. Expression (Relative - absolute) only according to Six-XE-assembler.
4. Only valid special charactors for label the $ according Six-XE-assembler.
5. All validations are based upon machine.

Sample Run :



File Edit Format View Help

PASS 1

Line	Address	Label	Operation	Operand	Comment
1	0000				
2	0000	PROGR1	START	0000	
3	0000	P	LDA	Z	
4	0003		SUB	Y	
5	0006		STA	BETA	
6	0009		LDA	V	
7	000C		ADD	W	
8	000F		DIV	BETA	
9	0012		STA	E	
10	0015	Z	WORD	9	
11	0018	Y	WORD	4	
12	001B	BETA	RESW	1	
13	001E	V	WORD	5	
14	0021	E	RESW	1	
15	0024		END	P	

SYMBOL TABLE

Name	Address
P	0000
PROGR1	0000
Z	0015
Y	0018
BETA	001B
V	001E
E	0021

```
progr1   start   0000
P        LDA     Z
         SUB     Y
         STA     BETA
         LDA     V
         ADD     W
         DIV     BETA
         STA     E
Z        WORD    9
Y        WORD    4
BETA     RESW    1
V        WORD    5
```

```
                          PASS 1
Line      Address   Label     Operation      Operand   Comment
1         0000                                         .23456789012345678901234567890123456
2         0000
3         0000              START                      .
4         0000              LDA            1000
5         0000      BGN     STA            #5
6         0003              LDCH           ALPHA
7         0006              EQU            #90
8         0009      AAA     STCH           bgn+12
9         0009              RESW           C1
10        000C              EQU                        .
11        000C      ALPHA   RESB           1
12        000F      XXX     END            ALPHA

 SYMBOL TABLE
Name      Address
BGN       0000
AAA       0009
ALPHA     000C
C1        000F
XXX       000F


.23456789012345678901234567890123456
.Label.  Opcode  The  O p e r a n d
.
         START   1000
BGN      LDA     #5
         STA     ALPHA
         LDCH    #90
aaa      equ     bgn+12
         STCH    C1
.
ALPHA    RESW    1
xxx      equ     alpha
C1       RESB    1
         END     BGN
```