AdvCalc2IR Documentation

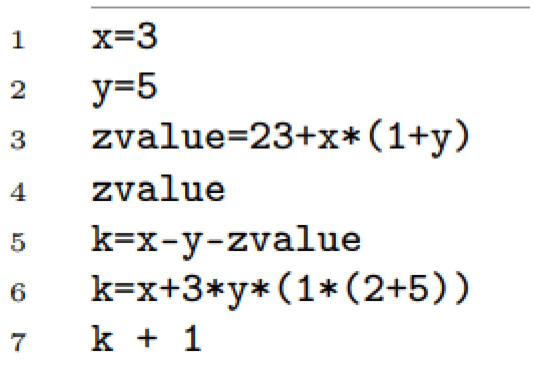
Introduction

This is a command line program that converts arithmetic expressions to LLVM IR language.

Program Interface

Program can be started by typing ./advcalc2ir and giving it file path as parameter in console. Program reads lines from the file. Any invalid expression will be printed. Ex: “Error on line 8!”. After reading lines program will create a file in LLVM IR format named filename.ll.

Input File

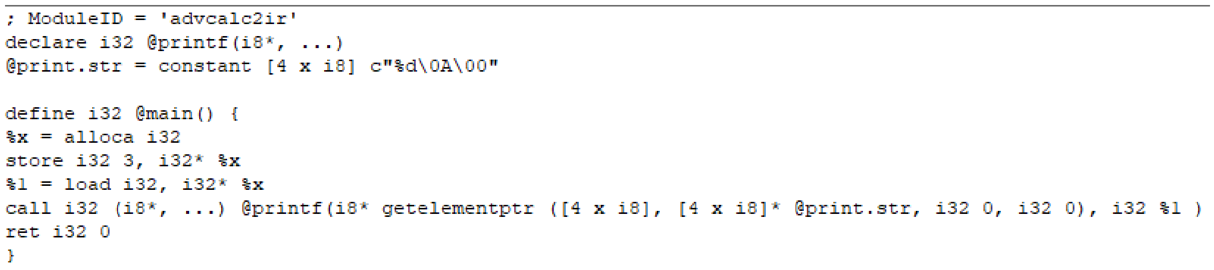


```
x=3
y=5
zvalue=23+x*(1+y)
zvalue
k=x-y-zvalue
k=x+3*y*(1*(2+5))
k + 1
```

Example Format

The file lines should be formatted as algebraic expression. It can include “=” sign to assign values to variables. Any syntax error will be printed as error.

Output File



```
; ModuleID = 'advcalc2ir'
declare i32 @printf(i8*, ...)
@print.str = constant [4 x i8] c"%d\0A\00"

define i32 @main() {
%x = alloca i32
store i32 3, i32* %x
%1 = load i32, i32* %x
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %1 )
ret i32 0
}
```

Example Output File

Program will create output file that in LLVM IR format.

Program Structure

Program calculates the expressions via creating or editing structs. To process the expression, there are 6 main sections that handle the process. They are main, Lexer, Parser, Calculator, VariableController, ProcessCounter. It can be guessed that the project is designed as a basic compiler because the origin of processing arithmetic expression is the same as processing programming language.

Lexer

Lexer is a system that gets an input then creates meaningful tokens that are called lexemes by filtering unnecessary characters and gathering meaningful words.

Example:
input: (x + 57) / asd
outTokenArray: (“(“ , Left Parenthesis), (“x“ , Variable), (“+“ ,Plus), (“57“ , Number), (“)“ , Right Parenthesis), (“/“ ,Divide), (“asd“ ,Variable)
outSize: 7

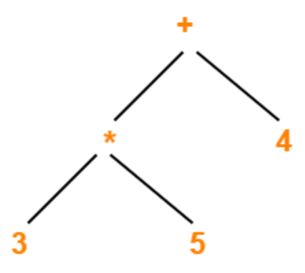
Parser

Parser is a system that reads lexemes then creates a parse tree for arithmetic expressions. To create a parse tree, the system looks at the BNF Grammar then applies the rules of the grammar. The tree will handle the precedence of the operators via looking at the BNF Grammar.

The BNF Grammar for this program:

```
<bitor> ::= <bitor> | <bitor> "|" <bitor>
<bitand> ::= <sum> | <bitand> "&" <sum>
<sum> ::= <fac> | <sum> "+" <fac> | <sum> "-" <fac>
<fac> ::= <par> | <fac> "*" <par> | <fac> "/" <par> | <fac> "%" <par>
<par> ::= <func> | "(" <bitor> ")"
<func> ::= <number> | <var> | <funcname> "(" <bitor> "," <bitor> ")" | <funcname> "(" <bitor>
")"
<number> ::= <digit> | <digit> <number>
<digit> ::= [0-9]
<var> ::= <alpha> | <alpha> <var>
<alpha> ::= [a-z] | [A-Z]
<funcname> ::= "xor" | "ls" | "rs" | "lr" | "rr" | "not"
```

Example:
It can be visualized for this input: 3 * 5 + 4



Calculator

Calculator takes an input as the root of the parse tree then recursively writes the necessary process to output file.

Variable Controller

While these 3 systems are running they will require variable value. This system will handle this request. It takes input as a variable name then makes reservations in RAM if necessary.

Process Counter

Process Counter count number of variables created. It is used in creation of %1, %2, %3

Main

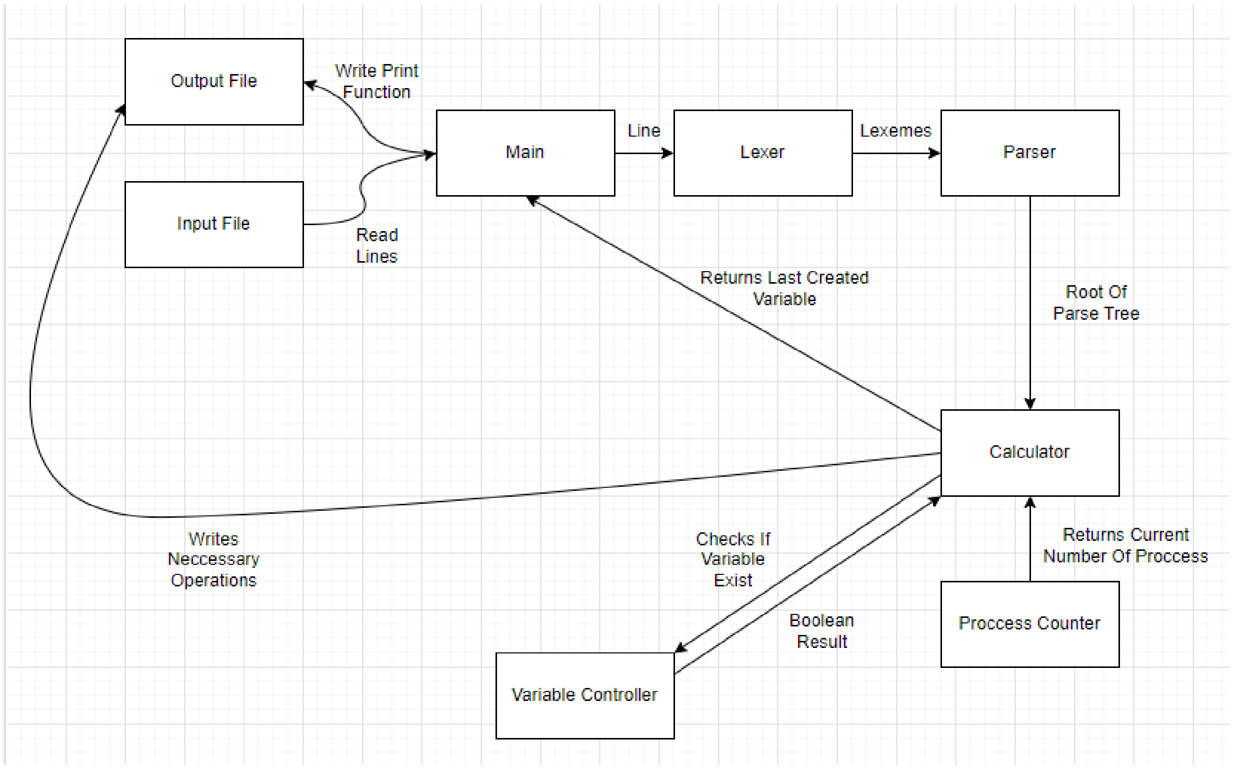
This is the command line handler part. It reads inputs from the file then sends it to systems then it will write the result to output file. We can say that this is the client side of the code.

Structs and Enums

All structs and enums defined in Structures.h.

- **enum TokenType**: It will be used in Node struct that will give clues about what info Node carries.
- **struct Token**: It will help the creating Node struct instances.
- **struct Node**: Parse Tree element, if it is leaf node, it will carry data. Else have references of other nodes.
- **struct VariableStruct**: It will keep the variable data and name.

The high-level program structure diagram shown below:



Functions

Lexer:

Syntax:

void GetLexemes(const char line[], Token outTokenArray[], int* outSize)

Parameters:

- const char line[]: the expression
- Token outTokenArray[]: it will be filled with lexemes, it is a return variable from parameter
- int* outSize: it will be count of lexemes, it is a return variable from parameter

---

Parser:

Syntax:

void PrepareParser(int TokenCount, const Token* TokenPtr);

Description:

Before the parse processes, the system should be prepared. This function prepare the system

Parameters:

- int TokenCount: token amount
- const Token* TokenPtr: token array

Syntax:

void DisposeParser(struct Node* startPoint);

Description:

Free the parse tree

Parameters:

- struct Node* startPoint: root of the tree

Syntax:

bool anyErrorOccurred ();

Description:

Check is there any error

Return:
    true if error occurred.

Syntax:
    struct Node* ParseBitAnd();

Description:
    Creates parse tree

Return:
    Root of the tree

Calculator:

Syntax:
    char* calculate(const struct Node* startPoint);

Description:
    Evaluates the parse tree, then writes

Parameters:
- const struct Node* startPoint: root of the parse tree

Return:
    Last created variable

Variable Controller:

Syntax:
    void setVariableValue(const char* name, long long value);

Description:
    Sets variable and save it

Parameters:
- const char* name: variable name
- long long value: variable value

Syntax:
    bool checkIfVariableExist(const char* name);

Description:
    Checks if variable exist or not

Parameters:
- const char* name: variable name

Return:

True if exist else False

Syntax:

```
void disposeVariables();
```

Description:

Free the saved variables

Process Counter:

Syntax:

```
int getProcessCount();
```

Description:

it returns a number that unique id of process

Return:

Last process number

Syntax:

```
char* getVarProcessName();
```

Description:

Creates unique process name such as %1, %2, %3

Return:

Unique process name
