

cfglp: A CFG (Control Flow Graph) Language Processor (version 2.0)

fglp: A CFG (Control Flow Graph) Language Processor

version 2.0)

[day Khedker](#) and Tanu Kanvar

[Department of Computer Science & Engg., IIT Bombay](#)

- [Home](#)
- [About cfglp](#)
- [An overview of compilation](#)
- [Functionality of cfglp](#)
- [Using cfglp](#)
- [Assignments involving enhancing cfglp level 0](#)
- [Design of cfglp level 0](#)
 - [Grammar](#)
 - [Data structures](#)
 - [Interpretation](#)
 - [Possible improvements](#)
- Download cfglp level 0
 - [Complete tarball](#)
 - [Individual files](#)
- [Reference implementations](#)
- [Using Flexc++ and Bisonc++](#)
 - [Recognizing numbers](#)
 - [Recognizing expressions](#)
 - [Computing Expressions](#)
 - [Installing Flexc++ and Bisonc++](#)

about cfglp

The language processor cfglp has been implemented for UG courses cs302+cs306: Language Processors (theory and lab) at IIT Bombay. It was designed and implemented by Uday Khedker. Tanu Kanvar refactored version 1.0 and revised it thoroughly to create version 2.0. All copyrights are served by Uday Khedker. This implementation has been made available purely for academic purposes without any warranty of any kind.

If you are new to the area of compilers, you may want to [see this for an overview](#) before proceeding further.

The main motivation behind this implementation has been to create a small well-crafted code base for UG compiler construction class such that it can be enhanced systematically by students. We begin with a very small language and enhance it using well defined increments. *Each step supports new source language features rather than a new phase in a compiler or an interpreter.* In other words, each step results in

- a complete working compiler which generates code which can be executed, and
- a complete working interpreter which executes all programs that can be written with the allowed source language features.

The complexity of understanding the complete compilation is controlled by restricting the input language instead of omitting phases or modules of the language processor. We believe that this provides a much better feel of an integrated working system than the traditional approach of enhancing the compiler phase-wise.

We welcome feedback and suggestions on improving cfglp, as also bug reports.

functionality of cfglp

This language processor takes as input GCC tree-cfg (control flow graph) dumps. Based on command line parameters, the read program is either interpreted or compiled to generate equivalent spim assembly language program.

The overall functionality supported by cfglp is best described by the usage printed by it:

```
Usage: cfglp [options] [file]
Options:
  -help          Show this help
  -tokens        Show the tokens in file.toks (or out.toks)
  -ast           Show abstract syntax trees in file.ast (or out.ast)
  -symtab        Show the symbol table of declarations in file.sym (or out.sym)
  -program       Show the complete program read by cfglp in file.prog (or out.prog)
                 (-program option cannot be given with -tokens, -ast, or -symtab)
  -eval          Interpret the program and show a trace of the execution in file.eval (or
out.eval)
  -compile       Compile the program and generate spim code in file.spim (or out.spim)
                 (-eval and -compile options are mutually exclusive)
                 (-compile is the default option)
  -lra           Do local register allocation to avoid redundant loads within a basic block
  -icode         Compile the program and show the intermediate code in file.ic (or out.ic)
                 (-eval and -icode options are mutually exclusive)
  -d             Demo version. Use stdout for the outputs instead of files
```

For simplicity, we have divided cfglp into two parts: Interpreter and Compiler. Current description on this page is applicable to the interpreter only and the options -symtab, -program, -compile, -lra, and -icode are not available. The compiler description will be added later and the page will be reorganized.

cfglp: A CFG (Control Flow Graph) Language Processor (version 2.0)

Using cfglp

The required input for *cfglp* can be created by using GCC-4.6 by giving the command `gcc -fdump-ee-cfg file.c`. This would generate a dump file called `file.c.013t.cfg`. There are two limitations of the default *cfg* dump by GCC:

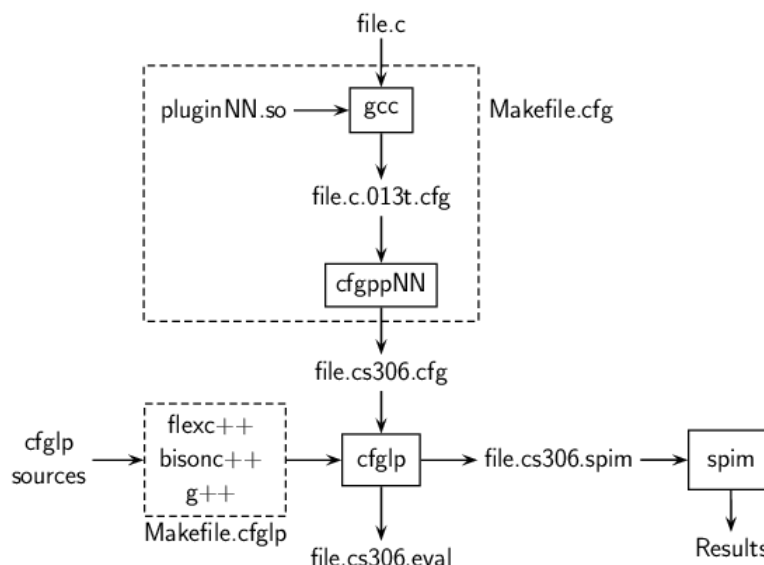
- GCC does not dump the declaration of global variables in the dump file.
- GCC introduces many temporary variables in the dump file
 - to hold the intermediate results so that the expressions can be broken down into smaller expressions, and
 - to load the global variables and address taken variables (viewed as memory) into local variables (viewed as registers).

In order to simplify the input for *cfglp*, a dynamic plugin for GCC is used and the generated dump file is re-processed using a pre-processor called *cfgpp*. We have provided binaries for these for both 32 bit intel based ubuntu as well as 64 bit intel based ubuntu.

Using *cfglp* involves the following three steps:

- Preparing the input for *cfglp*.
- Building *cfglp*.
- Executing *cfglp*.

These steps are illustrated in the figure below. The number NN appearing in the figure refer to 32 or 64 depending upon the word size of the machine.



The steps are explained below.

- *Preparing the input for cfglp*. This can be done using a makefile called *Makefile.cfg* which, given an input file *test.c*, invokes the dynamic plugin within GCC and pre-processes the dumps to eventually produce a *cfg* file called *test.cs306.cfg*. The Makefile assumes that *test.c* is present in a subdirectory called *test_files* and generates the *test.cs306.cfg* file in the same subdirectory. The command to invoke the makefile for input file *test.c* is: `make FILE=test.c -f Makefile.cfg`.

Here is the makefile for [32 bit intel based ubuntu](#) and for [64 bit intel based ubuntu](#). Note that we expect that you have gcc-4.6 installed on your machine. The dumps created by other versions of GCC may be slightly different and may have to be modified to meet the *cfglp* requirements.

- *Building cfglp*. This can be done using a makefile called *Makefile.cfglp*. The command to invoke this makefile is: `make -f Makefile.cfglp`.

Here is the makefile for [32 bit intel based ubuntu](#) and for [64 bit intel based ubuntu](#).

- *Executing cfglp*. The command to run *cfglp* on a *cfg* file generated in the first step is: `./cfglp test_files/test.cs306.cfg`.

Please see the [README](#) and [USER-MANUAL](#) files for more details.

cfglp: A CFG (Control Flow Graph) Language Processor (version 2.0)

Assignment Involving Enhancing cfglp 0

the moment cfglp accepts *level 0* language that defines pre-processed cfg dumps corresponding to programs that consist of a (possibly empty) list of global declaration statements followed by the definition of the `main` procedure. No other procedure definition is allowed.

- [Home](#)
- [About cfglp](#)
- [An overview of compilation](#)
- [Functionality of cfglp](#)
- [Using cfglp](#)
- [Assignments involving enhancing cfglp level 0](#)
- [Design of cfglp level 0](#)
 - [Grammar](#)
 - [Data structures](#)
 - [Interpretation](#)
 - [Possible improvements](#)
- Download cfglp level 0
 - [Complete tarball](#)
 - [Individual files](#)
- [Reference implementations](#)
- [Using Flexc++ and Bisonc++](#)
 - [Recognizing numbers](#)
 - [Recognizing expressions](#)
 - [Computing Expressions](#)
 - [Installing Flexc++ and Bisonc++](#)

- The lone `main` procedure has the following restrictions:
 - It can have a return statement but cannot return a value. Thus it does not have a return type.
 - It consists of a (possibly empty) list of local declaration statements followed by a (possibly empty) list of assignment statements.
 - The declarations can introduce only integer variables (introduced by the type specifier `int`). No other type or data structure is allowed.
 - The list of statements does not have control flow statements or procedure calls.
- An assignment statement has the following restrictions:
 - The right hand side must be an integer variable or an integer number.

For examples of valid and invalid inputs, please see the [test files directory](#).

- The `.c` files are valid test cases. These when passed through `cfgpp` produce a corresponding `.cs306.cfg` file. These files in turn when used as input to `cfglp` produce a valid output.
- The `.ecfg` files are invalid test cases. These have been created by manually editing the GCC generated files because GCC does not generate an invalid dump. The `.ecfg` files when passed as input to `cfglp` will either reject the files or will give invalid output.

For subsequent levels are defined by incrementally enhancing the input language given to GCC as follows:

1. *Level 1*. This level extends level 0 by supporting intraprocedural control flow statements and evaluation of conditions in the following ways:
 - Conditional expression `e1?e2:e3`, while, do-while, for, if, and if-else statements may exist in the C source program.
 - `switch` and `goto` statements are not present in the C program.
 - The boolean conditions controlling the control flow consist of the six comparison operators (`<`, `<=`, `>`, `>=`, `==`, and `!=`) and three logical operators (`&&`, `||`, and `!`).
 - The values used for comparison are only integer values.
 - The comparison expressions can also appear on the right hand sides of assignment statements.
 - In some cases, GCC optimizes a statement
 - `A = (A < B) ? A : B;` to `A = MIN_EXPR <B, A>;`
 - `A = (A > B) ? A : B;` to `A = MAX_EXPR <B, A>;`
 such optimizations are not expected to be handled.
2. *Level 2*. This level extends level 1 in the following ways:
 - Apart from integers, single precision floating point types (introduced by the type specifier `float`) are supported.
 - Expressions appearing in the right hand side of an assignment or in comparisons may contain the following arithmetic operators: `+`, `-`, `*`, `/`, and unary minus. Note that the expressions do not have any side effects.

3. *Level 3*. This level extends Level 2 by supporting function calls (including recursion). In this level, we have multiple functions. Besides, the functions may take arguments and may return results.

4. *Level 4*. This level extends Level 3 by supporting data and procedure encapsulation supported by classes.

It may be advisable to implement the above enhancements in smaller steps in the increasing order of complexity. A possible sequence of assignments focussed in each step could be defined as follows:

- Group A: Extend the interpreter to levels 1, 2, and 3.
- Group B: Extend the compiler to levels 1, 2, and 3.
- Group C: Extend the interpreter to level 4.
- Group D: Extend the compiler to level 4.

cfglp: A CFG (Control Flow Graph) Language Processor (version 2.0)

Reference Implementations

In order to help check your implementation, executables of reference implementation for different levels will be provided as the semester progresses. Currently, the reference implementation for Level 1 interpreter has been provided.

- [Home](#)
- [About cfglp](#)
- [An overview of compilation](#)
- [Functionality of cfglp](#)
- [Using cfglp](#)
- [Assignments involving enhancing cfglp level 0](#)
- [Design of cfglp level 0](#)
 - [Grammar](#)
 - [Data structures](#)
 - [Interpretation](#)
 - [Possible improvements](#)
- [Download cfglp level 0](#)
 - [Complete tarball](#)
 - [Individual files](#)
- [Reference implementations](#)
- [Using Flexc++ and Bisonc++](#)
 - [Recognizing numbers](#)
 - [Recognizing expressions](#)
 - [Computing Expressions](#)
 - [Installing Flexc++ and Bisonc++](#)

Level 1 Interpreter	
32 bit intel ubuntu	64 bit intel ubuntu
Tarball	Tarball
gcc-4.6 plugin	gcc-4.6 plugin
pre-processor cfqpp	pre-processor cfqpp
cfglp level 1 executable	cfglp level 1 executable
Makefile.cfg	Makefile.cfg
Test files for level 1	Test files for level 1
README file	README file
User Manual	User Manual

We would appreciate hearing about bugs if you come across any.

Design of cfglp level 0

cfglp embodies object oriented design and implementation. The implementation language is C++. Bisonc++ and flexc++ have been used for parsing and scanning. Bisonc++ specifications create an object of class parser which is then invoked in the main function. The overall compilation flow is the classical sequence of

- parse and construct ASTs and symbol table (invoke the scanner when a token is needed, and typecheck during parsing), and
- evaluate (store the results in the symbol table), or
- compile (assign offsets to variables, generate code in intermediate form by invoking optional register allocation, emit assembly instructions during compilation).

The main classes that constitute cfglp are:

- program, procedure, basic block, abstract syntax trees
- symbol table (persistent as well as volatile symbol table for scope analysis)
- register and instruction descriptor
- intermediate code

Level 0 Grammar

A program consists of a sequence of a declaration statement list which is a list of global variables, a procedure name, and finally the procedure body (in level 0 we have a single procedure).

```

program: declaration_statement_list
        procedure_name
        procedure_body
        | procedure_name
        procedure_body
        ;

```

The first rule matches if the program contains global variables, otherwise the second rule matches. The associated action saves the list of global variables and checks that the names of the global variables and the procedure name are distinct. If no global variables are declared in the user program then the global variable list is NULL. Once the body of the procedure is processed, the procedure is added with its procedure name to a map of procedures (in level 0 we have only one procedure).

Our procedure header indicates that the procedure does not have a return type associated with it (because it does not return a value) and does not have any parameters. The procedure body is

enclosed in a pair of braces '{' and '}' and contains an optional list of declarations followed by a list of basic blocks.

cfglp: A CFG (Control Flow Graph) Language Processor (version 2.0)

- [Home](#)
- [About cfglp](#)
- [An overview of compilation](#)
- [Functionality of cfglp](#)
- [Using cfglp](#)
- [Assignments involving enhancing cfglp level 0](#)
- [Design of cfglp level 0](#)
 - [Grammar](#)
 - [Data structures](#)
 - [Interpretation](#)
 - [Possible improvements](#)
- [Download cfglp level 0](#)
 - [Complete tarball](#)
 - [Individual files](#)
- [Reference implementations](#)
- [Using Flexc++ and Bisonc++](#)
 - [Recognizing numbers](#)
 - [Recognizing expressions](#)
 - [Computing Expressions](#)
 - [Installing Flexc++ and Bisonc++](#)

```

procedure_name: NAME '(' ')'
;

procedure_body: '{'
                declaration_statement_list
                basic_block_list
            '}'
            | '{'
              basic_block_list
            '}'
;

```

The action associated with this rule checks if the list of local variables have been declared in the user program, and saves the list of local variables and the list of basic blocks.

The declaration statement list is defined recursively as follows:

```

declaration_statement_list: declaration_statement_list
                           declaration_statement
                           | declaration_statement
                           ;
declaration_statement: INTEGER NAME ';'
;

```

Note that the same non-terminal is used for lists of global as well as local variables and the position where it occurs (inside of a procedure or outside it) tells us whether it is global or local. A declaration of a variable consists of the type name (only INTEGER in level 0) and the NAME of the variable followed by semicolon (';'). The action checks variables for possible redeclaration within the same scope. Variables thus recognized, are pushed in a symbol table.

A basic block is a labelled list of groups of executable statements.

```

basic_block_list: basic_block_list
                  basic_block
            '}'
            | basic_block
            ;
basic_block: BB_NUMBER
             executable_statement_list
            ;

```

In the current level we have only assignment statements and an optional return statement.

```

executable_statement_list: assignment_statement_list
                           | assignment_statement_list
                           RETURN
                           ;

```

At level 0 an assignment statement can involve only a variable or an integer number on the right hand side and only a variable on the left hand side.

```

assignment_statement_list : /* empty */
                           | assignment_statement_list
                           assignment_statement
                           ;
assignment_statement : variable '=' variable ';'
                     | variable '=' constant ';'
                     ;
variable : NAME
;

```

cfglp: A CFG (Control Flow Graph) Language Processor (version 2.0)

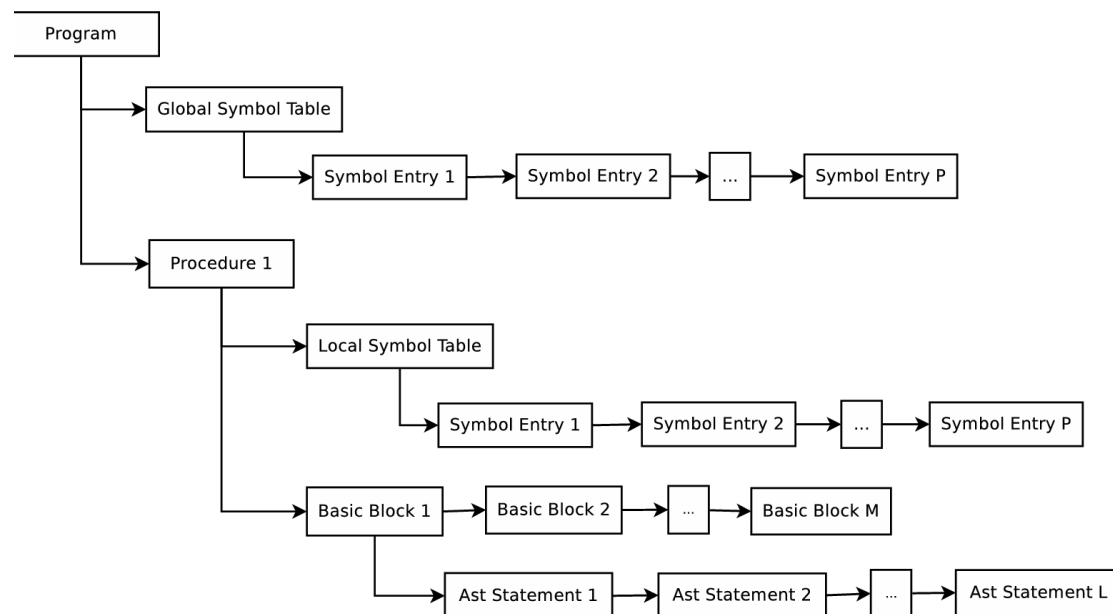
- [Home](#)
- [About cfglp](#)
- [An overview of compilation](#)
- [Functionality of cfglp](#)
- [Using cfglp](#)
- [Assignments involving enhancing cfglp level 0](#)
- [Design of cfglp level 0](#)
 - [Grammar](#)
 - [Data structures](#)
 - [Interpretation](#)
 - [Possible improvements](#)
- Download cfglp level 0
 - [Complete tarball](#)
 - [Individual files](#)
- [Reference implementations](#)
- [Using Flexc++ and Bisonc++](#)
 - [Recognizing numbers](#)
 - [Recognizing expressions](#)
 - [Computing Expressions](#)
 - [Installing Flexc++ and Bisonc++](#)

```
constant : INTEGER_NUMBER
;
```

ie associated actions check that a variable has been declared and the types of the left hand side and e right hand side are same.

Level 0 Data Structures

ie data structure of the interpreter for Level 0 is as shown below:



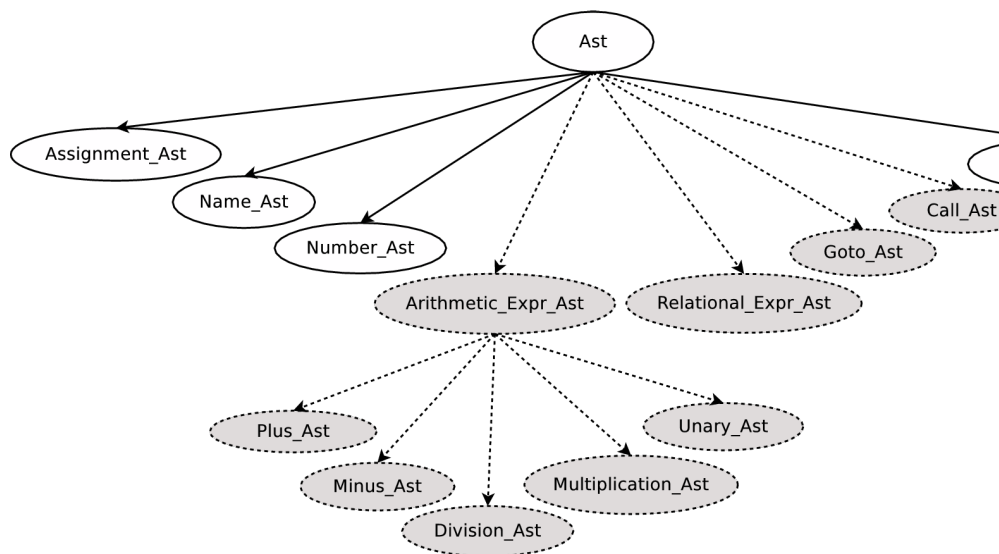
ie program object is an instance of class Program. It contains:

- A global_symbol_table object which is an instance of class Symbol_Table. It is a list of symbol_entry objects which are instances of class Symbol_Table_Entry.
- A procedure object which is an instance of class Procedure. It contains
 - A local_symbol_table object which is an instance of class Symbol_Table. It is a list of symbol_entry objects which are instances of class Symbol_Table_Entry.
 - A basic_block_list object which is a list of pointers to instances of class Basic_Block.

Each basic block contains a list of objects representing AST Statements (Abstract Syntax Tree). They are represented by an abstract class Ast as shown below. The following classes are derived from Ast in Level 0: Name_Ast, Number_Ast, Assignment_Ast, and Return_Ast.

cfglp: A CFG (Control Flow Graph) Language Processor (version 2.0)

- [Home](#)
- [About cfglp](#)
- [An overview of compilation](#)
- [Functionality of cfglp](#)
- [Using cfglp](#)
- [Assignments involving enhancing cfglp level 0](#)
- [Design of cfglp level 0](#)
 - [Grammar](#)
 - [Data structures](#)
 - [Interpretation](#)
 - [Possible improvements](#)
- [Download cfglp level 0](#)
 - [Complete tarball](#)
 - [Individual files](#)
- [Reference implementations](#)
- [Using Flexc++ and Bisonc++](#)
 - [Recognizing numbers](#)
 - [Recognizing expressions](#)
 - [Computing Expressions](#)
 - [Installing Flexc++ and Bisonc++](#)



The Ast classes described in gray colour are to be included in the subsequent levels.

Level 0 Interpretation

During interpretation, the local and global variables are copied into a separate data structure called `eval_env` which is an instance of class `Local_Environment` which stores the variable names and their formation for each procedure call (including recursive function calls). Note that this data structure is not necessary in level 0 because the values could be stored in the symbol table but this design has been chosen keeping in mind the enhancements in the subsequent levels.

Interpretation is performed by traversing the basic blocks and visiting each AST in the basic block. Each AST is evaluated and the values of the variables in `eval_env` are updated. This requires recording the targets of goto statements also. Further, subsequent levels would need recording the values of visiting point variables also. In order to implement a generic evaluate procedure across all ASTs, the values are stored using objects of a class called `Eval_Result`. We derive three classes `Eval_Result_Value` (used to store values of variables) and `Eval_Result_BB` (used to store the targets of gotos in level 1). We also derive classes `Eval_Result_Value_Int` and `Eval_Result_Value_Float` (this distinction will be required in level 2).

Current Limitations and Possible Improvements

The design and implementation of cfglp has some limitations which we expect to overcome in subsequent versions. We welcome more additions to the following list.

- The use of a map to store symbol table details is fine for global and local variables where the order does not matter. However, for formal parameters, the order matters (because of positional correspondence between actual and formal parameters). I have been able to circumvent this problem by explicitly storing the sequence number but perhaps a better design is possible.
- We would like to introduce an optimizer generator which works with cfglp to allow the users to specify analyses and transformations declaratively and automatically construct optimizers that work within cfglp.

Using Flexc++ and Bisonc++

We use flexc++ and bisonc++ to generate C++ code for the scanner and parser. They provide a cleaner interface compared to flex and bison which generates C code by default but can be made to generate C++ code. For a detailed documentation, please visit the original [flexc++ web page](#) and the [bisonc++ web page](#). They contain extensive documentation. Here we provide some details to help your understanding.

We introduce flexc++ and bisonc++ in three steps keeping the examples as simple as possible:

cfglp: A CFG (Control Flow Graph) Language Processor (version 2.0)

- [Home](#)
- [About cfglp](#)
- [An overview of compilation](#)
- [Functionality of cfglp](#)
- [Using cfglp](#)
- [Assignments involving enhancing cfglp level 0](#)
- [Design of cfglp level 0](#)
 - [Grammar](#)
 - [Data structures](#)
 - [Interpretation](#)
 - [Possible improvements](#)
- Download cfglp level 0
 - [Complete tarball](#)
 - [Individual files](#)
- [Reference implementations](#)
- [Using Flexc++ and Bisonc++](#)
 - [Recognizing numbers](#)
 - [Recognizing expressions](#)
 - [Computing Expressions](#)
 - [Installing Flexc++ and Bisonc++](#)

- First we construct a [scanner to recognize numbers](#). This requires using flexc++ without bisonc++.
- Its [tarball can be downloaded](#) or [files can be seen here](#).
- In the second step, we construct [a scanner and a parser to recognize expressions](#). This requires cooperation between the scanner and parser in that the scanner must pass tokens to the parser. This requires us to modify the generated class descriptions. We use five grammars beginning with simplest of expressions to more complicated expressions. Its [tarball can be downloaded](#) or [the files can be seen here](#).
- In the final step, we construct [a calculator to compute the values of expressions](#). This requires further interaction between the scanner and the parser because apart from the tokens, the token values also need to be passed from the scanner to the parser. This requires us to change the classes further to share variables. Its [tarball can be downloaded](#) or [the files can be seen](#).

We explain the interaction between the scanner and the parser in details in the third step.

Using Flexc++ to Create a Scanner to Recognize Numbers

We provide a simple example of using flexc++ to create a standalone scanner. Since the generated scanner is called from main.cc rather than from within a parser, the class Scanner generated by flexc++ is sufficient and there is no need to manually change it unlike in the next two cases.

We reproduce the scanner script below. It defines tokens using regular expressions defined over character classes and prints the lexemes found along with the token that they match.

```
digit [0-9]
operator [-+*/]

%%
{digit}+ { std::cout << "Found a number whose lexeme is '" << matched() << "'\n"; }
{operator} { std::cout << "Found an operator whose lexeme is '" << matched()[0] << "'\n"; }
. { std::cout << "Found an unrecognized character '" << matched()[0] << "'\n"; }
```

Please [download the tarball](#) or [see the files](#). Please see the README file and Makefile for more details.

Using Flex++ and Bisonc++ together to Recognize Expressions

We provide a simple example of using flexc++ and bisonc++ to create a parser and scanner that recognize valid expressions. Now the scanner returns a token to the parser (eg. return Parser::NUM;).

Scanner script	Parser scripts
<pre>exp.ll %% [0-9]+ { return Parser::NUM; } [-+*/] { return matched()[0]; } \n { return matched()[0]; } [\t] { } %%</pre>	<pre>exp1.yy %start E %token NUM %% E : NUM { cout << "found an expression consists of a number\n"; } ; exp2.yy %start E %token NUM %% E : NUM { cout << "found an expression consists of a number\n"; } E '+' E { cout << "found a plus expression\n"; } ; exp3.yy %start E %token NUM %left '+' %% E : NUM { cout << "found an expression consists of a number\n"; } E '+' E { cout << "found a plus expression\n"; }</pre>

cfglp: A CFG (Control Flow Graph) Language Processor (version 2.0)

- [Home](#)
- [About cfglp](#)
- [An overview of compilation](#)
- [Functionality of cfglp](#)
- [Using cfglp](#)
- [Assignments involving enhancing cfglp level 0](#)
- [Design of cfglp level 0](#)
 - [Grammar](#)
 - [Data structures](#)
 - [Interpretation](#)
 - [Possible improvements](#)
- Download cfglp level 0
 - [Complete tarball](#)
 - [Individual files](#)
- [Reference implementations](#)
- [Using Flexc++ and Bisonc++](#)
 - [Recognizing numbers](#)
 - [Recognizing expressions](#)
 - [Computing Expressions](#)
 - [Installing Flexc++ and Bisonc++](#)

```

;

%start E
%token NUM
%left '+'
%left '*'

exp4.yy %%
E : NUM      { cout << "found an expression consist of a number\n"; }
  | E '+' E   { cout << "found a plus expression\n"; }
  | E '*' E   { cout << "found a mult expression\n"; }
;

%start E
%token NUM
%left '+' '-'
%left '*' '/'

exp5.yy %%
E : NUM      { cout << "found an expression consist of a number\n"; }
  | E '+' E   { cout << "found a plus expression\n"; }
  | E '*' E   { cout << "found a mult expression\n"; }
  | E '-' E   { cout << "found a sub expression\n"; }
  | E '/' E   { cout << "found a div expression\n"; }
;

```

Please [download the tarball](#) or [see the files](#). Please see the README file and Makefile for more details.

Simple Expressions Calculator Example

We use a simple calculator program to explain the interaction between a scanner and a parser generated using flexc++ and bisonc++. The calculator computes the result of a single expression consisting of numbers and operators + and *.

Here are the scanner and parser specifications for our calculator, along with the associated actions to perform the computation are:

Scanner script	Parser script
<pre> % 0-9]+ { ParserBase::STYPE__ *val = getSval(); *val = atoi(matched().c_str()); return Parser::NUM; } [+*] { return matched()[0]; } \n { return matched()[0]; } [\t] { } . { string error_message; error_message = "Illegal character `" + matched(); error_message += "` on line " + lineNr(); cerr << error_message; exit(1); } %% </pre>	<pre> %token NUM %start Start %left '+' %left '*' %% Start : Expr '\n' { cout << ; Expr : Expr '+' Expr { \$\$ = \$1 + Expr '*' Expr { \$\$ = \$1 * NUM { \$\$ = \$1; ; </pre>

Please [download the tarball](#) or [see the files](#).

In this grammar we have four terminals (or tokens): NUM, +, *, and \n and two non-terminals Start and Expr. The operators + and * have been declared to be left associative and * is declared to have a higher precedence than + (because its associativity specification appears lower). The action `$$ = $1 +`

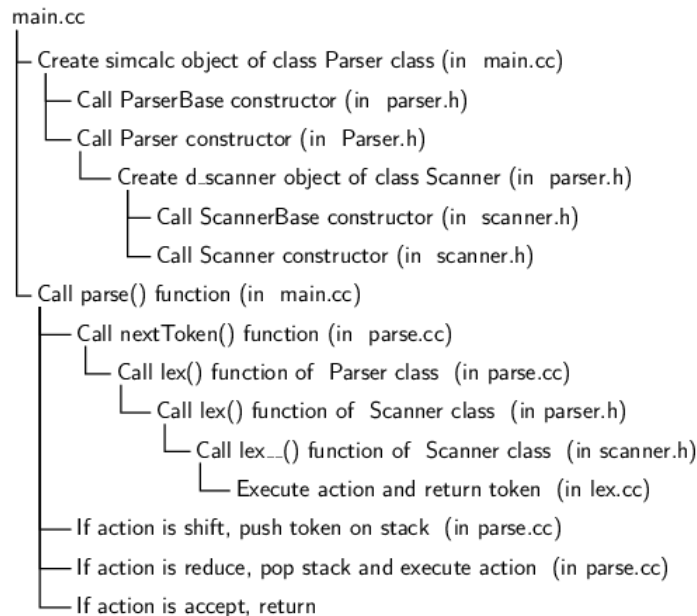
cfglp: A CFG (Control Flow Graph) Language Processor (version 2.0)

- [Home](#)
- [About cfglp](#)
- [An overview of compilation](#)
- [Functionality of cfglp](#)
- [Using cfglp](#)
- [Assignments involving enhancing cfglp level 0](#)
- [Design of cfglp level 0](#)
 - [Grammar](#)
 - [Data structures](#)
 - [Interpretation](#)
 - [Possible improvements](#)
- Download cfglp level 0
 - [Complete tarball](#)
 - [Individual files](#)
- [Reference implementations](#)
- [Using Flexc++ and Bisonc++](#)
 - [Recognizing numbers](#)
 - [Recognizing expressions](#)
 - [Computing Expressions](#)
 - [Installing Flexc++ and Bisonc++](#)

↗ adds up the values associated with the two occurrences of Expr on the right hand side and assigns to the value of the Expr appearing on the left hand side. The action { \$\$ = \$1; } assigns the value of M to Expr.

ie scanner is defined by an object of class Scanner which inherits from a base class ScannerBase. Similarly, the parser is defined by an object of class Parser which inherits from a base class ParserBase.

we create an object called simcalc as an instance of class Parser and then call its parse() function. The constructor of Parser creates an object called d_scanner as an instance of the Scanner class. The parse() function receives tokens from the lex() function and takes the parsing decisions. This has been illustrated below.



Once the tokens are recognized by the scanner, and their values have to be passed by the scanner to the parser. In this case, after the scanner identifies a certain sequence of digits as a lexeme corresponding to the token NUM, it should also compute the value of the number from the lexeme and make it available to the parser. We explain this interaction below by first showing the default Scanner class generated by flexc++ and the default Parser class generated by bisonc++. Then we show how these classes are modified.

Automatically generated Scanner and Parser classes are as shown below

```

class Scanner: public ScannerBase
{
public:
    explicit Scanner(std::istream &in = std::cin,
                    std::ostream &out = std::cout);

    Scanner(std::string const &infile,
            std::string const &outfile);

    // $insert lexFunctionDecl
    int lex();

private:
    int lex__();
    int executeAction__(size_t ruleNr);

    void print();
    void preCode();
};
  
```

```

class Parser: public ParserBase
{
    // $insert scannerobject
    Scanner d_scanner;

public:
    int parse();

private:
    void error(char const *msg);
    int lex();

    void print();

    // support functions for parse():
    void executeAction(int ruleNr);
    void errorRecovery();
    int lookup(bool recovery);
    void nextToken();
};
  
```

cfglp: A CFG (Control Flow Graph) Language Processor (version 2.0)

- [Home](#)
- [About cfglp](#)
- [An overview of compilation](#)
- [Functionality of cfglp](#)
- [Using cfglp](#)
- [Assignments involving enhancing cfglp level 0](#)
- [Design of cfglp level 0](#)
 - [Grammar](#)
 - [Data structures](#)
 - [Interpretation](#)
 - [Possible improvements](#)
- Download cfglp level 0
 - [Complete tarball](#)
 - [Individual files](#)
- [Reference implementations](#)
- [Using Flexc++ and Bisonc++](#)
 - [Recognizing numbers](#)
 - [Recognizing expressions](#)
 - [Computing Expressions](#)
 - [Installing Flexc++ and Bisonc++](#)

Manually updated classes (see the lines in red) to enable transfer of values from the Scanner to the Parser

```
class Scanner: public ScannerBase
{
    public:
        explicit Scanner(std::istream &in = std::cin,
                        std::ostream &out = std::cout);

        Scanner(std::string const &infile,
                std::string const &outfile);

        // $insert lexFunctionDecl
        int lex();

    private:
        ParserBase::STYPE__ * dval;

        int lex__();
        int executeAction__(size_t ruleNr);

        void print();
        void preCode();

    public:
        void setSval(ParserBase::STYPE__ * val);
        ParserBase::STYPE__ * getSval();
};
```

```
class Parser: public ParserBase
{
    // $insert scannerobject
    Scanner d_scanner;

    public:
        Parser()
        {
            d_scanner.setSval(&d_val__);
        }

        int parse();

    private:
        void error(char const *msg);
        int lex();

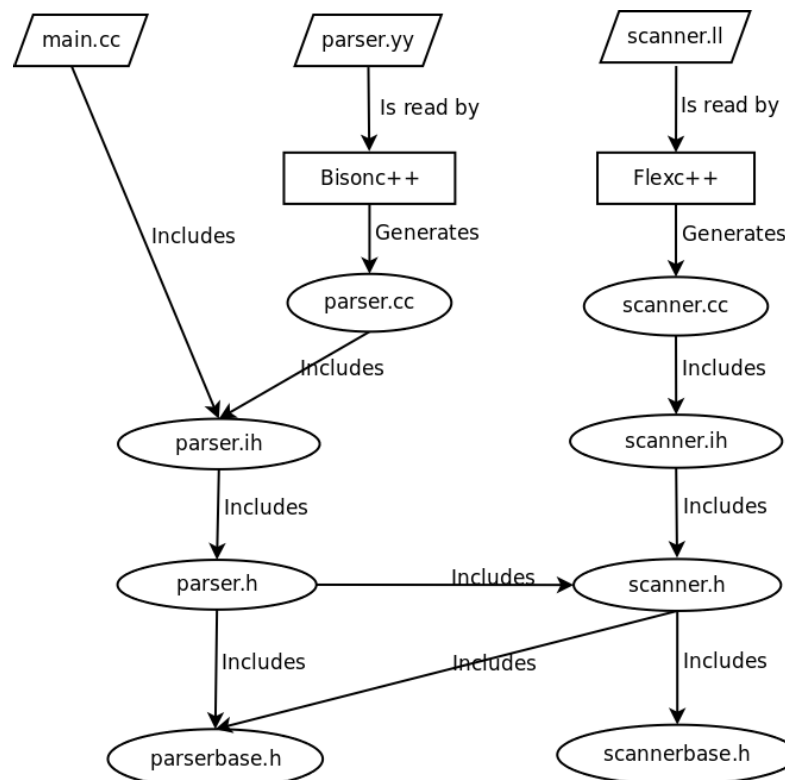
        void print();
        // support functions for parse():
        void executeAction(int ruleNr);
        void errorRecovery();
        int lookup(bool recovery);
        void nextToken();
};
```

The class Parser has a protected member `d_val__` which it inherits from the base class `ParserBase`. We make the address of this variable available to the Scanner class by adding new constructor for the Parser class which calls the function `d_scanner.setSval` (`d_scanner` is the scanner object used by the Parser object). This is a function that we have added in the Scanner class. This function stores the pointer in a private variable `dval`. Whenever the script in `scanner.ll` has to make the value available to the parser, it gets the pointer using the function `getSval` and stores the value in the address which happens to be the address of the variable `d_val__` in an object of the Parser class.

File inclusions are a little tricky and have been illustrated below. The files at the top are the files that we write. The `parser.yy` and `scanner.ll` are the scripts that are read by `bisonc++` and `flexc++` which then generate the `parser.cc` and `scanner.cc` files and the associated header files.

cfglp: A CFG (Control Flow Graph) Language Processor (version 2.0)

- [Home](#)
- [About cfglp](#)
- [An overview of compilation](#)
- [Functionality of cfglp](#)
- [Using cfglp](#)
- [Assignments involving enhancing cfglp level 0](#)
- [Design of cfglp level 0](#)
 - [Grammar](#)
 - [Data structures](#)
 - [Interpretation](#)
 - [Possible improvements](#)
- Download cfglp level 0
 - [Complete tarball](#)
 - [Individual files](#)
- [Reference implementations](#)
- [Using Flexc++ and Bisonc++](#)
 - [Recognizing numbers](#)
 - [Recognizing expressions](#)
 - [Computing Expressions](#)
 - [Installing Flexc++ and Bisonc++](#)



Installing Flexc++ and Bisonc++

The only flip side of using flexc++ and bisonc++ is that they require multiple steps for installation. Our system administrators have installed them in all machines in NSL. If you want to install them on your machines, please read on.

The steps for installing Flexc++ and Bisonc++ on ubuntu (12.04+) on a 64 bit intel processor based machine have been given below. For a 32 bit intel processor based machine, please replace **amd64** by **86** in the installation of libbobcat and libbobcat-dev.

We need to install the following packages. Note that the specific version numbers (or higher versions) are important. Wherever possible, we suggest using the apt-get command (synaptic package manager could do just as well) minimising the need of installation from .deb files or source files.

- g++ version 4:4.6.3-1ubuntu5.
 - Use the command `sudo apt-get install g++`.
- bisonc++ version 2.09.03-1.
 - Use the command `sudo apt-get install bisonc++`.
- icmake version 7.16.01-1ubuntu1
 - Use the command `sudo apt-get install icmake`.
- libbobcat version 3_3.01.00-1
 - Visit the site <https://launchpad.net/ubuntu/quantal/amd64/libbobcat3/3.01.00-1>.
 - Check for all dependencies and their versions listed on the site. You may want to use the synaptic package manager to find out whether the dependencies have been installed on your system. Install all missing dependencies.
 - Download the libbobcat3_3.01.00-1_amd64.deb file and use the command `sudo dpkg -i libbobcat3_3.01.00-1_amd64.deb`.
- libbobcat-dev version 3.01.00-1. These are the development libraries (note the suffix -dev).
 - Visit the site <https://launchpad.net/ubuntu/quantal/amd64/libbobcat-dev3/3.01.00-1>.
 - Check for all dependencies and their versions listed on the site. You may want to use the synaptic package manager to find out whether the dependencies have been installed on your system. Install all missing dependencies.
 - Download the libbobcat-dev_3.01.00-1_amd64.deb file and use the command `sudo dpkg -i libbobcat-dev_3.01.00-1_amd64.deb`.
- flexc++ version 0.98.00.
 - Visit the site <https://launchpad.net/ubuntu/+source/flexc++/0.98.00-1>.
 - Check for all dependencies and their versions listed on the site. You may want to use the

cfglp: A CFG (Control Flow Graph) Language Processor (version 2.0)

- [Home](#)
- [About cfglp](#)
- [An overview of compilation](#)
- [Functionality of cfglp](#)
- [Using cfglp](#)
- [Assignments involving enhancing cfglp level 0](#)
- [Design of cfglp level 0](#)
 - [Grammar](#)
 - [Data structures](#)
 - [Interpretation](#)
 - [Possible improvements](#)
- Download cfglp level 0
 - [Complete tarball](#)
 - [Individual files](#)
- [Reference implementations](#)
- [Using Flexc++ and Bisonc++](#)
 - [Recognizing numbers](#)
 - [Recognizing expressions](#)
 - [Computing Expressions](#)
 - [Installing Flexc++ and Bisonc++](#)

synaptic package manager to find out whether the dependencies have been installed on your system. Install all missing dependencies.

- Download the source code archive file `flexc++_0.98.00.orig.tar.gz`.
- Untar the source code using the command `tar zxvf flexc++_0.98.00.orig.tar.gz`.
- Change the directory using `cd flexc++-0.98.00`.
- Follow the steps given in the file `INSTALL`.