GCC Frontend HOWTO

Table of Contents

GCC Frontend HOWTO

v 1.1, August 2002

Sreejith K Menon

Creating a new GCC front end

---

8. Demo front end

9. See also

---

1. Introduction

This document shows the steps required for creating a new GCC front end. It helps you to create a compiler of your own with the help of the GNU Compiler Collection. Basic information about tools like Bison and Flex is also provided to make the document self contained.

I assume that you have sound knowledge of the C programming language. A general idea about compilers will help you understand the document better. If you wish to make experiments of your own, please download the source code of GCC from http://gcc.gnu.org.

1.1 New Versions of the document

This version of the document will help you in developing basic language constructs. Succeeding revisions will be focussed on more complex issues.

1.2 Feedback and Corrections

This document may have mistakes in it because I am writing from my practical experiments with the front end. There may be faults in the way I have grasped things. Please inform me about the mistakes, so that I can correct them in the next version. I always welcome suggestions and criticisms. I can be contacted at sreejithkmenon@yahoo.com

1.3 Why I have written this

I started out trying to add a small language to the GNU Compiler Collection. Even though there is an excellent manual which describes GCC internals, I found it a bit intimidating to the newbie hacker. I thought of documenting my experiments so that even novice programmers can start tinkering with the complex GCC code base.

1.4 Distribution Policy

1.5 Acknowledgements

This document is the by–product of an intensive `code reading' experiment conducted at the Government Engineering College, Trichur (GECT). Students were asked to read and tinker with the source code of reasonably complex systems software to give them a feel of how large systems are designed and maintained – some of us concentrated on the GCC front end, others on the back end (we hope to have a document on hacking the GCC backend soon!). Those who have a flair for Operating Systems had a chance to work on the Linux file system, scheduler, VM subsystem and the networking stack.

I am indebted to the Free Software community as a whole for giving me a chance to play with the source of useful(and exciting!) programs. My thanks to the faculty of the Department of Computer Science at GECT for their commitment to education. Thanks to Mr.Pramode C.E for leading me to Linux and compilers.

I am grateful to Tim Josling, who has created a small beautiful front end, which helped me in all my experiments.

2. Some general ideas about Compilers

A compiler is a translator which accepts programs in a source language and converts them into programs in another language which is most often the assembly code of a real (or virtual) microprocessor. Designing real compilers is a complex undertaking and requires formal training in Computer Science and Mathematics.

The compilation process can be divided into a number of subtasks called phases. The different phases involved are

1. Lexical analysis
2. Syntax analysis
3. Intermediate code generation
4. Code Optimization
5. Code generation.

Symbol tables and error handlers are involved in all the above phases. Let us look at these stages.

1. Lexical analysis

   The lexical analyzer reads the source program and emits tokens. Tokens are atomic units, which represent a sequence of characters. They can be treated as single logical entitities. Identifiers, keywords, constants, operators, punctuation symbols etc. are examples of tokens. Consider the C statement,

   ```
   return 5;
   ```

   It has 3 tokens in it. The keyword 'return', the constant 5 and the punctuation semicolon.
2. Syntax analysis

   Tokens from the lexical analyzer are the input to this phase. A set of rules (also known as productions) will be provided for any programming language. This defines the grammar of the language. The syntax analyzer checks whether the given input is a valid one. ie. Whether it is permitted by the given grammar. We can write a small set of productions.

```
Sentence        -> Noun Verb
Noun            -> boy  | girl | bird
Verb            -> eats | runs | flies
```

This is a grammar of no practical importance (and also no meaning). But it gives a rough idea about a grammar.

A parser for a grammar takes a string as input. It can produce a parse tree as output. Two types of parsing are possible. Top down parsing and bottom up parsing. The meaning is clear from the names. A bottom up parser starts from the leaves and traverse to the root of the tree.

In the above grammar if we are given "bird flies" as input, it is possible to trace back to the root for a valid 'Sentence'. One type of bottom−up parsing is a "shift−reduce" parsing. The general method used here is to take the input symbols and push it in a stack until the right side of a production appears on top of the stack. In that case it is reduced with the left side. Thus, it consists of shifting the input and reducing it when possible. An LR (left to right) bottom up parser can be used.

3. Intermediate Code Generation.

   Once the syntactic constructs are determined, the compiler can generate object code for each construct. But the compiler creates an intermediate form. It helps in code optimization and also to make a clear−cut separation between machine independent phases (lexical, syntax) and machine dependent phases (optimization, code generation).

   One form of intermediate code is a parse tree. A parse tree may contain variables as the terminal nodes. A binary operator will be having a left and right branch for operand1 and operand2.

   Another form of intermediate code is a three−address code. It has got a general structure of A = B op C, where A, B and C can be names, constants, temporary names etc. op can be any operator. Postfix notation is yet another form of intermediate code.
4. Optimization

   Optimization involves the technique of improving the object code created from the source program. A large number of object codes can be produced from the source program. Some of the object codes may be comparatively better. Optimization is a search for a better one (may not be the best, but better).

   A number of techniques are used for the optimization. Arithmetic simplification, Constant folding are a few among them. Loops are a major target of this phase. It is mainly because of the large amount of time spent by the program in inner loops. Invariants are removed from the loop.
5. Code generation

   The code generation phase converts the intermediate code generated into a sequence of machine instructions. If we are using simple routines for code generation, it may lead to a number of redundant loads and stores. Such inefficient resource utilization should be avoided. A good code generator uses its registers efficiently.
6. Symbol table

   A large number of names (such as variable names) will be appearing in the source program. A compiler needs to collect information about these names and use them properly. A data structure used for this purpose is known as a symbol table. All the phases of the compiler use the symbol table in one way or other.

Symbol table can be implemented in many ways. It ranges from the simple arrays to the complex hashing methods. We have to insert new names and information into the symbol table and also recover them as and when required.

7. Error handling.

   A good compiler should be capable of detecting and reporting errors to the user in a most efficient manner. The error messages should be highly understandable and flexible. Errors can be caused because of a number of reasons ranging from simple typing mistakes to complex errors included by the compiler (which should be avoided at any cost).

---

3. Compiler Tools

Two tools, Flex and Bison, are provided for the compiler development. If you are having a general idea regarding them you can skip the next two sections, since I have got nothing new to say.

3.1 Flex

Flex is a fast lexical analyzer generator. As explained, the first phase of building a compiler is lexical analysis. Flex is an efficient tool for performing the pattern matching on text. In the absence of Flex we will have to write our own routines for obtaining tokens from the input text.

But with flex we can provide the regular expression to be matched and the action to be taken when a perfect match is found. Our file will have an extension .l, which shows that it is a valid lex file. The output of the flex is a file called lex.yy.c. It has a routine yylex() defined in it. The file, lex.yy.c can be compiled and linked with the '−lfl' library to produce the executable.

One or two examples will make the things clearer. Create a small file, say lex.l with the following contents.

```
%%

"good"  { printf("bad"); }

%%
```

Produce the executable with the commands

```
lex lex.l
cc lex.yy.c −lfl
```

Run the executable. We find that for each occurrence of the string "good", a replacement with the string "bad" is made. For any other input, the input is just echoed. We here have our first lesson − the default action is to just copy the input to the output.

The general structure of the flex file is

```
definitions
%%
rules
```

```
%%
user code
```

The definitions may contain a 'name definition'. For example,

```
DIGIT [0-9]
```

It defines "DIGIT" to be a regular expression, which matches a single digit. The main purpose of name definition is to simplify the scanner specification.

Next is the 'rules' section of the flex file. The general form is

```
pattern action
```

where the pattern may be a regular expression. The action should reside on the same line of pattern. The patterns and actions are described below.

In the 'rules' section it is permissible to use variable declarations enclosed in %{ }. It should appear before the first rule. They are local to the scanning routine.

Let us look at an example.

```
%{
#define WHILE 1
#define IF 2
%}

%%
while           {return WHILE; }
if              {return IF; }
%%

main()
{
        int val;
        while (val = yylex())
                printf("%d",val);
        printf("final =%d\n", val);
}
```

In the above program for the occurrence of "while" and "if", the corresponding integer value is printed. At the end, for an EOF, the program terminates by returning zero.

Patterns

Here I have only mentioned about the general patterns, which will be required in our compiler construction. For a complete list you are encouraged to refer the manual page.

```
`x'             match the character x.
`.'             any character except the newline.
`[xyz]'         either an `x' or a `y' or a `z'.
```

```
`[a-z]'         either an `a' or a `b' ... or a `z'.
`[^A-Z]'        any character except an uppercase letter.
`r*'            zero or more r's.
`r+'            one or more r's.
`r?'            zero or one r's.
`{name}'        the expansion of the "name" description.
                (As explained above).
`\x'            if x is an `a', `b', `f', `n', `r', `t' or `v'
                then the ANSI C representation of \x. Otherwise
                a literal `x'.
`\0'            the NULL character.
`(r)'           match an r.
                parentheses to override precedence.
`rs'            concatenation of r and s.
`r|s'           either an r or an s
```

The regular expressions mentioned above are arranged in the decreasing order of precedence. The topmost one has the highest precedence. In case of any confusion you can make use of parentheses to explicitly show what you mean.

Generally, the first question that will be coming in our mind will be − What happens when multiple matches are found. In that case the scanner chooses the one with the maximum length. That is, if we have a file,

```
%%

"ab"            {printf("first"); }
"abc"           {printf("second"); }

%%
```

and we are providing the input "abcd" then the two possibilities are "firstcd" and "secondd". The scanner prefers only the second.

But consider the case when the lengths are same. Then the rule given first in the file will get preference over the other.

Once the match is clearly defined then the corresponding action provided can be executed. The text corresponding to the match is available in 'yytext' and its length in 'yyleng', both global values. It is better to avoid local names starting with 'yy' because of its extensive use by the scanner and parser. Its avoidance also contributes to better readability.

```
%%

[0-9]+  {printf("The value of first yytext is %s\n",yytext);}
[a-z]+  {printf("The value of sec yytext is %s\n",yytext);}

%%
```

Actions

We find that for each pattern given in the lex file it has an associated action. The action can be any arbitrary C code. It is possible to use the constructs like 'return' to return a value to the caller of yylex. In our compiler we need only simple actions, which can be understood by anyone having some knowledge with the C language.

The above mentioned details are more than enough for our compiler. For the beginners it is highly recommended to try out the various examples and check the different variations of the regular expressions. Before proceeding to the next section you should have a basic idea regarding the Flex.

3.2 Bison

Once we get used with the lexical analyzer, we are ready to meet its best companion − the parser generator, Bison. Given a description for an LALR(1) context−free grammar, it is the duty of Bison to generate a C program to parse that grammar. As explained, the second stage of compiler construction is parsing. We are supplied with the tokens from the lex. We have to define a grammar for a language and see whether the given input is a valid one.

Before proceeding let us look what a context free grammar is and what we mean by terminals and nonterminals.

A context free grammar is a finite set of nonterminals, each of which represents a language. The language represented by the nonterminals is defined recursively in terms of each other and with the help of primitive symbols called terminals.

Thus in simple words terminals are those which can't be further subdivided whereas nonterminals are formed from the grouping of smaller constructs. It is possible to subdivide the nonterminals.

As an example, consider the grammar

Note: I haven't used the bison syntax in this example.

```
A -> Ab
A -> b
```

It denotes all the strings having only one or more b's. Here A is a nonterminal because it can be divided further using the given productions. But b is a terminal symbol because it is not possible to further divide it.

Suppose we are given a string "bbb". We have to check whether it is accepted by the above productions. Assume the start symbol is 'A'.

```
A -> Ab       {rule −1}
  -> Abb      {rule −1}
  -> bbb      {rule −2} and thus accepted.
```

In Bison, generally the terminal symbols are represented in uppercase Ex := NUM (say, for a number) or by using character literal as in the case of '+'. As we expect, the nonterminals are represented by using lowercase letter. Ex := exp. (We'll obey this rule when we switch to Bison examples.).

Flex and Bison work with perfect mutual understanding. A Bison grammar rule may say that "an expression is made of a number followed by a plus sign followed again by a number". The flex whenever sees a number informs the bison that it has found a number. That is it informs the presence of a token to the parser.

The grammar rule is only concerned whether the given input obeys the rules. Suppose we are given a terminal symbol NUM. The grammar rules no longer bother whether we are having a value 1 as NUM or whether the value is 100. For the grammar all the numbers are just the terminal symbols NUM. But we may be certainly interested in the value of NUM. Here comes the importance of 'Semantic Values' and 'Semantic Actions'.

Associated with each grammar rule the parser allows us to define certain actions. For the above example,

```
A -> b { printf("We have found a `b'\n"); }
```

In most cases the actions may not be simple as the above one. Suppose we are implementing a small calculator, the semantic action may be to perform an addition operation.

The terminals and nonterminals present in the grammar can have an associated value. The value is extracted using the symbol '$n' where n is an integer. A rule can have a semantic value. (Actually it is the value of the nonterminal represented by that rule). It is defined by using the symbol '$$'.

For example,

```
exp: exp '+' exp        {$$ = $1 + $3;}
```

which stands for exp –> exp '+' exp. The contents of{ } denote the semantic action. The semantic actions are generally made of C statements.

In the above example, consider the right hand side of the production. The first exp is denoted by '$1'. The terminal symbol '+' is represented by '$2' and the last exp is denoted by '$3'. We find here that it is possible for a terminal symbol like '+' to have no associated semantic value. The value associated with the grammar is '$$' which is the sum of the first and third token.

Suppose we are also having a rule,

```
exp: NUM        {$$ = $1;}
```

Let the given input be '1 + 2'. Then the tokens 1 and 2 will match the NUM. The semantic value of the rule exp: exp '+' exp would be 3 due to the corresponding semantic action.

Bison Grammar File

The general form of a bison parser file is

```
%{
C DECLARATIONS
%}

BISON DECLARATIONS

%%
GRAMMAR RULES
%%

ADDITIONAL C CODE
```

The C declarations generally contain the #include's and other declarations. The bison declarations handle the terminals and nonterminals. The productions explained in the above section form the Grammar rules. Additional C code contains the rest of the programs used (if needed).

The ideas will be clearer with an example. Consider a small grammar capable of taking lines of expressions and returning their values.

The lexical file, lex.l is given. No explanations are given for the file. In case of any doubt refer the Flex section.

```
%{
#include"parse.tab.h"
#include<stdio.h>
%}
%%
[0-9]+                  {yylval=atoi(yytext);return NUM;}
"+"                     {return '+';}
"*"                     {return '*';}
"-"                     {return '-';}
"\n"                    {return '\n';}
"/"                     {return '/';}
%%
```

The parser file, parse.y is also given.

```
%{
#include<stdio.h>
%}

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

%start line

%%

line:
       /* empty */
     |line exp '\n' {printf("%d\n",$2);}
     | error '\n';

exp:      exp '+' exp {$$ = $1 + $3;}
        | exp '*' exp {$$ = $1 * $3;}
        | exp '-' exp {$$ = $1 - $3;}
        | exp '/' exp { if ($3 == 0)
                                $$ = 0;
                        else
                                $$ = $1/$3;}
        | NUM          {$$ = $1;};
%%

yyerror()
{
        printf("Error detected in parsing\n");
}

main()
{
        yyparse();
}
```

Let us explore the program line by line. Also let us look how the program works with the lexical file.

The C declaration part is simple. Here we are using only stdio.h. If required other header files can also be included. The second part of the bison file consists of the bison declarations. Every terminals that are used in the file have to be declared here. Implicit terminals such as a character literal needn't be mentioned. Here we are only dealing with a single terminal called NUM. Others are character literals such as '\n', '+' etc.

```
%token NUM
```

completes the declaration.

In the expression we will be handling a number of operators such as '+', '−', '*' and '/'. The different operators are having different precedence. (For example, '/' will be having more precedence than the '+'. '+' and '−' have the same precedence). Also the operators will be having different associativity. All the operators in our code are left associative. This information is passed to the parser with the following declarations

```
%left  ->  for left associative.
%right ->  for right associative.
```

The order in which the declarations are made defines the precedence. Higher the line number, higher will be the precedence. If we are declaring "%left '/'" under "%left '+'", the '/' will have higher precedence. As expected declarations on same line denote equal precedence.

"%start" gives information to the parser about the start symbol. If not defined the first production is taken as the starting one.

Now let us move on to the Grammar rules. The first rule states that a line can be empty. No semantic actions are associated with that. The second rule

```
line:   line exp '\n' {printf("%d\n",$2); }
```

is the important one. It says that a line can be an expression followed by a newline. The left recursion used in the rule is just a technique used to parse multiple lines. You can avoid it if you are interested in parsing only a single line. The semantic action associated with the above rule is to print the value of the expression.

The rule − line : error '\n' will be explained later.

The rules for expression are simple. It just suggests that an expression can be an expression followed by any given operator and an expression. The rule exp: NUM provides a way to move out of the recursive rules. The semantic actions are just to perform the various operations.

The last section of the bison file defines the other C declarations. We have included only two functions. The main function just invokes the parser; and yyerror routine prints the error message. The function yyerror is invoked whenever the parser meets a parse error. The rule

```
line: error '\n'
```

is included to detect the error and consider the error as just another rule. If we are not including the production, the parser will terminate as soon as it meets an error. The nonterminal 'error' is implicitly declared in the parser and we can use them without any declaration.

Let us now look at the working of the parser and scanner. Suppose we provide the input "1+2". The scanner returns the token NUM whenever it finds a number. Also the value is stored in the global variable 'yylval' of the scanner. The parser checks whether the input is a valid one (according to the rules provided) and if it is, the corresponding actions are performed with the semantic values supplied.

But the problem is that the terminal NUM was declared only in the parser file. It has to be used in the lexical file. The problem is avoided by using the command

```
bison -d parse.y
```

It causes the creation of the file parse.tab.h, which includes all the required declarations. We can include it in the lexer.

Test and understand the working of the scanner and parser. Create the files given above and produce the executable with the following commands

```
lex lex.l
bison -d parse.y
cc parse.tab.c lex.yy.c -lfl
```

The above mentioned example is a simple one capable of recognizing only simple lines of expressions. But what we are going to deal is a compiler creation for a small programming language. Although the basic ideas are same, we have to acquire more understanding about the parser to work with a programming language. Keeping this in mind let us look at another example.

We create a new language with the following constructs − variable declarations, assignments and print statements. The lexical file is more or less same as the old one. But the parser file is different. The two files are given − lex.l and parse.y.

```
lex.l

%{
#include"parse.tab.h"
#include<stdio.h>
#include<string.h>
%}
%%

[0-9]+                  {yylval.val=atoi(yytext);return NUM;}
"print"                 {return PRINT;}
"declare"               {return DECL;}
[a-z]([0-9]|[a-z])*     {yylval.str= strdup(yytext);return NAME;}
"+"                     {return '+';}
"*"                     {return '*';}
"-"                     {return '-';}
"\n"                    {return '\n';}
"/"                     {return '/';}
"="                     {return '=';}

%%

parse.y

%{
#include<stdio.h>
```

```
#include<string.h>
#include<stdlib.h>

        struct node{
                char *name;
                int value;
        };
        static struct node* sym_table[100];
%}
%union{
        char *str;
        int val;
}

%token <val> NUM
%token <str> NAME
%token PRINT
%token DECL
%left '+' '-'
%left '*' '/'

%type <val> exp

%start input

%%
input: /* empty */
        | input line ;

line:
       exp '\n'          {}
      | DECL NAME '\n'    {return_value($2);}
      | NAME '=' exp '\n' {assign_value($1,$3);}
      | PRINT NAME '\n'   {printf("%d\n",return_value($2));}
      | error ;

exp:      exp '+' exp {$$ = $1 + $3;}
        | exp '*' exp {$$ = $1 * $3;}
        | exp '-' exp {$$ = $1 - $3;}
        | exp '/' exp { if ($3 == 0)
                                $$ = 0;
                        else
                                $$ = $1/$3;}
        | NUM         {$$ = $1;}
        | NAME        {$$ = return_value($1);};
%%

yyerror()
{
        printf("Error detected in parsing\n");
}

int assign_value(char *s,int symvalue)
{
        char *symname;
        int len,i;
        len=strlen(s) + 1;
        symname=malloc(sizeof(char *) * len);
        strcpy(symname,s);
        for(i=0;sym_table[i];i++)
                if(!strcmp(symname,sym_table[i]->name)){
                        sym_table[i]->value=symvalue;
```

```
                        return symvalue;
                }
        sym_table[i]=malloc(sizeof(struct node));
        sym_table[i]->name=symname;
        sym_table[i]->value=symvalue;
        return symvalue;
}

int return_value(char *s)
{
        char *symname;
        int len,i;
        len=strlen(s) + 1;
        symname=malloc(sizeof(char *) * len);
        strcpy(symname,s);
        for(i=0;sym_table[i];i++)
                if(!strcmp(symname,sym_table[i]->name))
                        return sym_table[i]->value;
        sym_table[i]=malloc(sizeof(struct node));
        sym_table[i]->name=symname;
        sym_table[i]->value=0;
        return 0;
}

main()
{
        yyparse();
}
```

In the parser file we find a new declaration %union. It is used to define the entire list of possible types. In the first example we had to work with only integers. But here the values can have more than one type. This information is passed through %union declaration. Since more than one type exists, the type information has to be specified for all the terminals and nonterminals whose semantic values are used in the grammar rules. It is shown in angle brackets. In the example we are making use of semantic values of NAME and NUM but not of PRINT and DECL. Similar changes are also made in the lexical file for 'yylval'.

```
%type <val> exp
```

is used to define the nonterminal and to specify the type.

The rest of the file is easy to understand. Whenever we see a new identifier we insert it into the symbol table. For new identifiers the initial value is made to be zero. Assignment statements cause the specified value to be stored in the symbol table. Two functions − assign_value and return_value are used for the symbol table manipulations.

4. GCC Front End

GCC (GNU Compiler Collection) is essentially divided into a front end and a back end. The main reason for this division is for providing code reusability. As all of us know GCC supports a variety of languages. This includes C, C++, Java etc.

If you want to introduce a new language into GCC, you can. The only thing you have to do is to create a new front end for that language.

The back end is same for all the languages. Different front ends exist for different languages. So creating a compiler in GCC means creating a new front end. Experimentally let us try to introduce a new language into the GCC.

We have to keep certain things in mind before we introduce a language. The first thing is that we are adding a segment to a huge code. For perfect working we have to add some routines, declare some variables etc. which may be required by the other code segments. Secondly some errors produced by us may take the back end into an inconsistent state. Little information will be available to us about the mistake. So we have to go through our code again and again and understand what went wrong. Sometimes this can be accomplished only through trial and error method.

4.1 Tree and rtl

Let me give a general introduction to tree structure and rtl. From my experience a person developing a new small front end need not have a thorough idea regarding tree structure and rtl. But you should have a general idea regarding these.

Tree is the central data structure of the gcc front end. It is a perfect one for the compiler development. A tree node is capable of holding integers, reals ,complex, strings etc. Actually a tree is a pointer type. But the object to which it points may vary. If we are just taking the example of an integer node of a tree, it is a structure containing an integer value, some flags and some space for rtl (These flags are common to all the nodes). There are a number of flags. Some of them are flags indicating whether the node is read only , whether it is of unsigned type etc. The complete information about trees can be obtained from the files − tree.c, tree.h and tree.def. But for our requirement there are a large number of functions and macros provided by the GCC, which helps us to manipulate the tree. In our program we won't be directly handling the trees. Instead we use function calls.

RTL stands for register transfer language. It is the intermediate code handled by GCC. Each and every tree structure that we are building has to be changed to rtl so that the back end can work with it perfectly. I am not trying in anyway to explain about rtl. Interested readers are recommended to see the manual of GCC. As with the trees GCC provides us a number of functions to produce the rtl code from the trees. So in our compiler we are trying to build the tree and convert it to rtl for each and every line of the program. Most of the rtl routines take the trees as arguments and emit the rtl statements.

So I hope that you are having a vague idea about what we are going to do. The tree building and rtl generation can be considered as two different phases. But they are intermixed and can't be considered separately. There is one more phase that the front end is expected to do. It is the preliminary optimization phase. It includes techniques like constant folding, arithmetic simplification etc. which we can handle in the front end. But I am totally ignoring that phase to simplify our front end. Our optimization is completely dependent on the back end. I also assume that you are perfect programmers. It is to avoid error routines from front end. All the steps are taken to simplify the code as much as possible.

From now onwards our compiler has only three phases − lexical, syntax and intermediate code generation. Rest is the head ache of the back end.

---

5. Installing the GCC

Let us begin with the preliminaries. I assume that you have down loaded the source code of GCC. The different steps of installation of GCC are given with it. But I add it here for completeness.

Like most GNU software GCC must be configured before it is built. Let the GCC source code be in a directory called 'srcdir'. Create a new directory 'objdir' where the GCC is to be built. Create one more directory called 'local' where the tools will be accumulated. Let our current directory be /home/name. Now we have /home/name/srcdir, /home/name/objdir and /home/name/local.

To configure GCC use the command

```
cd objdir
```

```
/home/name/srcdir/configure --prefix=/home/name/local/
```

When we complete the creation of our language add one more option,

```
--enable-languages=demo
```

where demo is the new language we are going to develop. The second step required is the building process. It is accomplished with the command

```
make bootstrap-lean
```

The last step is the final installation procedure. It is done with the command

```
make install
```

We can start our procedure of developing a new language.

6. Getting started

According to the formal GNU philosophy, each language that is present in the GCC (ie. for languages having separate front ends) should have a subdirectory of its own. So the first thing the GCC expects from us is a separate directory . Let us create a new subdirectory in the srcdir/gcc with the name of our language (say 'demo').

As explained before gcc expects a number of files and functions. In our directory also there should be some files present. The first thing that should be present is a make file. It is divided into two, Make−lang.in and Makefile.in. Both are part of the main make file of gcc.

Make−lang.in is actually included in the main make file of the gcc and from there (objdir/gcc) it calls the make in the language directory. So the filenames should be provided with full path names. Don't try to give relative ones. The file gives information about the source files of our language. So it is here that we should specify the files that is required for our language.

Makefile.in is used to create the make file in the language directory. It is included in the language directory.

The third file expected by the gcc is config−lang.in. It is used by the file srcdir/gcc/configure (a shell script).

The fourth file gcc expects is lang−specs.h. This file helps in the modification of the gcc driver program. The driver has to understand the presence of our new language. This is neatly accomplished by this file. It is here that the details like extensions of our language exists. You can specify them according to your taste.

Now the most important thing. No one is expecting you to create all these files from scratch. The best method is to copy these files from any existing directory and make the relevant changes. The changes may include some modifications in the name of the language, the files used by us, the extensions of our choice etc.

All the information provided are from my observation and sometimes there may be variations in the above details.

6.1 Call back routines

As already explained we are going to use a major part of the gcc for compiling our language. So it is our responsibility to define some functions and variables which are used by the back end. Most of them are of no direct help to us. But back end expects it, so we should provide it.

As in the above case it is better to copy from some existing front ends. But let's have a general idea of what each function is. If you find this section boring you can skip this section but don't skip the inclusion of these routines in your program.

```
type_for_size(unsigned precision, int unsignedp)
```

It returns a tree of integer type with number of bits given by the argument precision. If unsignedp is nonzero, then it is unsigned type, else it is signed type.

```
init_parse(char *filename)
```

It initialize parsing.

```
finish_parse()
```

It does the parsing cleanup.

```
lang_init_options()
```

Language specific initialization option processing.

```
lang_print_xnode(FILE *file,tree t,int i)
```

Required by gcc. Don't know what exactly it does.

```
type_for_mode(enum machine_mode mode,int unsignedp)
```

It returns a tree type of the desired mode given by us. mode represents machine data type like whole number. unsignedp, as usual is used for obtaining an unsigned type or else a signed type is returned.

```
unsigned_type(tree type_node)
```

Returns the unsigned version of type_node.

```
signed_type(tree type_node)
```

Returns the signed version of type_node.

```
signed_or_unsigned_type(int unsignedp, tree type)
```

Returns signed or unsigned tree node depending upon unsignedp.

```
global_bindings_p()
```

Returns nonzero if we are currently in the global binding level.

```
getdecls()
```

Returns the list of declarations in the current level, but in reverse order.

```
kept_level_p()
```

It is nonzero when a 'BLOCK' must be created for the current level of symbol table.

```
pushlevel(int ignore)
```

Enter a new binding level. A symbol name used before in another binding level is covered when entered into a new level.

```
poplevel(int keep, int reverse, int functionbody)
```

Removes a new level created by pushlevel. The symbol table status is regained (which was present before the pushlevel).

```
insert_block(tree block)
```

Insert block at the end of the list of subblocks of the current binding level.

```
set_block(tree block)
```

Sets the block for the current scope.

```
pushdecl(tree decl)
```

Inserts the declaration, decl into the symbol table and returns the tree back.

```
init_decl_processing()
```

Initializes the symbol table. It sets global variables and inserts other variables into the symbol table.

```
lang_decode_option(int a, char **p)
```

It decodes all language specific options that cannot be decoded by the GCC. Returns 1 if successful, otherwise 0.

```
lang_init()
```

Performs all the initialization steps required by the front end. It includes setting certain global variables.

```
lang_finish()
```

Performs all front end specific clean up.

`lang_identify()`

Returns a short string identifying the language to the debugger.

`maybe_build_cleanup(tree decl)`

Creates a tree node, which represents an automatic cleanup action.

`incomplete_type_error(tree value, tree type)`

Prints an error message for invalid use of incomplete type.

`truthvalue_conversion(tree expr)`

It returns the same expr, but in a type which represents truthvalues.

`mark_addressable(tree expr)`

Marks expr as a construct which need an address in storage.

`print_lang_statics()`

Prints any language−specific compilation statics.

`copy_lang_decl(tree node)`

It copies the declarations if DECL_LANG_SPECIFIC is nonzero.

`print_lang_decl(FILE *file, tree node, int indent)`

Outputs the declaration for node with indentation depth indent to the file, file.

`print_lang_type(FILE *file, tree node, int indent)`

Outputs the type for node with indentation depth indent to the file, file.

`print_lang_identifier(FILE *file, tree node, int indent)`

Outputs the identifier for node with indentation depth indent to the file, file.

`int_lex()`

Performs whatever initialization steps are required by the language dependent lexical analyzer.

`set_yydebug()`

Sets some debug flags for the parser.

`yyerror(char *s)`

Routine to print parse error message.

`language_string`

A character string to hold the name of our language. say, demo.

`flag_traditional`

A variable needed by the file dwarfout.c

`error_mark_node`

A tree node used to define errors. It represents a partial tree. It is of great help when some errors occur in the syntax analysis phase.

`integer_type_node, char_type_node, void_type_node`

Clear from the names.

`integer_zero_node, integer_one_node`

Constants of type integer_type_node with values 0 and 1 respectively.

---

7. Creating our own front end

7.1 Our Aim

We are going to develop a small new language with the following constructs − assignments, variable declarations, if and while statements etc. We are not going to directly compile our program but we are making the GCC to do the required operations and produce the machine code. Let's have a name to our language. I have given the name "demo", since it shows how to develop a new compiler. The syntax I am going to choose is a combination of Pascal and C so that it would be familiar to programmers of both the languages.

Our role is to clearly specify the syntax of our language and create a tree structure for the language. We will be creating the RTL from the tree structure. After that, it is the duty of the back end to produce an optimized output. We will be concentrating only on the tree structure creation and rtl conversion.

A number of functions and macros for handling the trees and rtl will be specified. Also a short description of what each does is given. The language that is being explained deals with only integer values so that we can avoid unnecessary type conversion complications.

7.2 Expressions

Let me start our language with the basic unit, expression. Here we have to perform only tree structure creation. It is possible with the help of a function called 'build'.

build(PLUS_EXPR, type, left, right) returns a tree for addition operation. Here left and right are two trees supplied by us for addition. We have one and only one type − the integer type. So there is no confusion regarding that.

Similarly, we have build1 used for operations like negations as shown

build1(NEGATE_EXPR,type,expression) where expression is the tree to be negated. It is used in case of unary operators.

And at last we can create a tree for simple whole numbers as shown

build_int_2(int num, num >=0 ? 0 : −1) to create a tree for the specific number. Actually, the two parameters passed are the low and high values of type HOST_WIDE_INT. But let us understand it like this − We have to set the second parameter of the function to show the correct sign. This is actually a macro invoking the function build_int_2_wide.

Now we have an idea regarding the creation of trees for building expressions. Let us directly write the parsing segment for tree creation for expressions.

```
exp:  NUM     { $$ = build_int_2 ($1, $1 >= 0 ? 0 : -1); }
     | exp '+' exp  { $$ = build (PLUS_EXPR, integer_type_node, $1, $3); }
     | exp '-' exp  { $$ = build (MINUS_EXPR, integer_type_node, $1, $3); }
     | exp '*' exp  { $$ = build (MULT_EXPR, integer_type_node, $1, $3); }
     | exp '/' exp  { $$ = build (TRUNC_DIV_EXPR, integer_type_node, $1, $3); }
     | exp '%' exp  { $$ = build (TRUNC_MOD_EXPR, integer_type_node, $1, $3); }
     | '-' exp %prec NEG { $$ = build1 (NEGATE_EXPR, integer_type_node, $2); }
     | '(' exp ')'  { $$ = $2; }
```

Now I am directly giving the lexical file required by demo language. No explanation is provided because of its simplicity.

```
%%
[ \n\t]         ;                   // white space
[0-9]+          {yylval.ival = atoi(yytext); return NUM;}   // integers
var             {return VAR;}       // variable declaration
return          {return RETURN;}    // return
if              {return IF;}        // if
then            {return THEN;}      // then
else            {return ELSE;}      // else
endif           {return ENDIF;}     // endif
while           {return WHILE;}     // while
endwhile        {return ENDWHILE;}  // endwhile
begin           {return BEGIN;}     // begin
end             {return END;}       // end
"<"             {return LT;}        // less than
">"             {return GT;}        // greater than
"=="            {return EQ;}        // equal to
[a-zA-Z]+       {yylval.name = strdup(yytext); return NAME;}
                                // function/variable name
.               {return yytext[0];} // match all single characters
%%
```

Let's also present the grammar we are going to develop so that I needn't repeat it always.

```
%union{
        tree exp;       //Tree node developed by us.
        int ival;       //Integer value for constants.
        char *name;     //Name of function or variables.
}



input:            fndef body ;

fndef:            NAME '(' ')';
```

```
body:             BEGIN declarations compstmts END;

declarations:     /*empty */
                | declarations VAR NAME

compstmts:        /*empty */
                | compstmts statements;

statements:       exp
                | assignstmt
                | ifstmt
                | whilestmt
                | returnstmt;

assignstmt:       NAME '=' exp;

whilestmt:        head loopbody;
head:             WHILE exp
loopbody:         compstmts ENDWHILE

ifstmt:           ifpart thenpart elsepart;
ifpart:           IF exp;
thenpart:         THEN compstmts;
elsepart:         ELSE compstmts ENDIF;

returnstmt:       RETURN exp;

exp:              NUM
                | exp '+' exp
                | exp '-' exp
                | exp '*' exp
                | exp '/' exp
                | NAME;
```

I will be developing our "demo" language in a step by step procedure. Starting from expressions we will move steadily to the end. At each stage a new construct will be introduced in the language. So in the final stage the "demo" language obeys the above grammar in all respect.

In the bison code above, I haven't provided the semantic actions for the productions. It will be introduced as we are studying to develop trees and rtl conversion.

Also in each stage I will be providing you only the required bison segments. You can look at the above grammar to understand the overall working.

7.3 Functions

Let us insert our expression inside a function. A large number of steps have to be invoked for a function. It ranges from setting up the parameters to the announcement of our declaration. The required tree and rtl functions are described in a step by step manner below. Most of the functions dealing with trees start with a 'build' and will be returning a tree structure. (I haven't explicitly given this fact everywhere. You should understand it by the way the functions involving trees are used.) We are going to build the tree using the functions provided by the GCC. Using the tree structure we will be creating the rtls.

For the functions provided by the GCC, the arguments are explained only if it is of any relative importance.

Let us write the routine for function handling. The given code builds a function with name, "name".

```
build_function_decl (char *name)
{
tree param_list;  //param_list is of type tree. Similarly others.
tree param_type_list;
tree fntype;
tree fndecl; // Consider it as a global value.
             // It will be required quite often.
```

/*First of all we have to arrange the parameters that are involved in the function. Suppose, we have "hello(int a, int b)" then a and b form the parameters. a and b have to be made available to the function. But since this is our first attempt, we are assuming that the function doesn't involve any parameters. So param_list is made to be a NULL_TREE (which is a NULL of type tree). */

```
param_list = NULL_TREE;
```

/*Next is the type of the parameters. The function always take a fixed number of parameters. Since we don't have any parameters in the current example, we are invoking the function, tree_cons as shown. The first field is a purpose field and the last one is a chain field (for linking together different types like integers). We will be explaining more about this when we pass some parameters. */

```
param_type_list = tree_cons(NULL_TREE, void_type_node, NULL_TREE);
```

/*Alright, we are done with the parameters. ie. the parameters and the type. We needn't bother about them. Now let's look at the function. The first thing is the type of the function. It depends on the return type and the parameter type. We consider our function as one returning an integer value. So the first argument of build_function is integer. The second one deals with parameters. The type of the parameters is given in the param_type_list. */

```
fntype = build_function_type(integer_type_node, param_type_list);
```

/*Next is the function declaration. It is possible with the function build_decl. The first argument says that it is a function declaration. The second one involves a function, get_identifier. It returns a tree whose name is "name". If an identifier with that name has been previously referred to, then the same tree node is returned. Since this is the first time we are using this, a new tree node is returned. The last argument deals with the type of the function. */

```
fndecl = build_decl(FUNCTION_DECL, get_identifier(name), fntype);
```

/*Here comes the flags. They are invoked through special macros given below*/

/*If nonzero, it means external reference. Needn't allocate storage. There is a definition elsewhere. But we need to make a definition here itself and hence zero. */

```
DECL_EXTERNAL(fndecl) = 0;
```

/* non zero means function can be accessed from outside the module.*/

```
TREE_PUBLIC(fndecl) = 1;
```

/* non zero means function has been defined and not declared. */

```
TREE_STATIC(fndecl) = 1;
```

/* It declares the argument of the function (which is stored in param_list)*/

```
DECL_ARGUMENTS(fndecl) = param_list;
```

/* It makes the declaration for the return value.*/

```
DECL_RESULT(fndecl) =
        build_decl(RESULT_DECL, NULL_TREE, integer_type_node);
```

/*It declares the context of the result. ie. We inform the result, that its scope is fndecl.*/

```
DECL_CONTEXT( DECL_RESULT( fndecl)) = fndecl;
```

/*Creates the rtl for the function declaration. The first argument gives the tree for the function declaration The second parameter deals with assembler symbol name. We don't want it here. We make it NULL. Let's look at third and fourth parameters later. Interested readers can refer toplev.c */

```
rest_of_decl_compilation (fndecl, NULL_PTR, 1, 0);
```

/*This gives idea regarding where the declaration is in the source code */

```
DECL_SOURCE_FILE( fndecl) = input_filename;
DECL_SOURCE_LINE( fndecl) = 1;
```

/* This function is just used to print the name of the function "name", on stderr if required */

```
announce_function( fndecl);
```

/* Let the GCC know the name of the function being compiled.*/

```
current_function_decl = fndecl;
```

/* It holds the tree of bindings. Just a method used here to make a partial tree. Don't bother about that. */

```
DECL_INITIAL( fndecl) = error_mark_node;
```

/* All tree and rtl are allocated in temporary memory. Used per function. */

```
temporary_allocation();
```

/*pushlevel is explained in call back. Here, it requires a push at the start of any function. */

```
pushlevel(0);
```

/*create function rtl for function definition. */

```
make_function_rtl( fndecl);
```

/*Generate rtl for the start of a function, fndecl. The second and third parameters denote the file and the line */

```
init_function_start(fndecl, input_filename, 1);
```

/*Let's start the rtl for a new function. It also sets the variables used for emitting rtl. The second parameter shows that there is no cleanup associated with. If it is made nonzero, cleanup will be run when a return statement is met. */

```
expand_function_start(fndecl, 0);
```

/*It generates the rtl code for entering a new binding level.*/

```
expand_start_bindings(0);
```

```
} //end of build_function_decl
```

All the above mentioned functions are invoked when, one enters a function. When we are leaving the function certain other things have to be done. These are explained in the code below.

build_function() {

/*Let's build the tree and emit the rtl for the return statement. In order to avoid an extra tree variable, I have included the tree creation and rtl conversion in a single statement. First build a tree of type result for 'fndecl'. I am always returning zero from our simple function. If you intend to return any other value, replace integer_zero_node with the other corresponding tree structure. expand_return creates the rtl for the tree. */

```
expand_return (build (MODIFY_EXPR, void_type_node, DECL_RESULT(fndecl),
                integer_zero_node));
```

/*Emit rtl for the end of bindings. Just like start bindings */

```
expand_end_bindings (NULL_TREE, 1, 0);
```

/* We have pushed. So don't forget to pop */

```
poplevel (1, 0, 1);
```

/*Emit rtl for the end of function. Just like starting */

```
expand_function_end (input_file_name, 1, 0);
```

/*Compile the function, output the assembler code, Free the tree storage. */

```
rest_of_compilation (fndecl);
```

/*We are free now */

```
current_function_decl=0;
```

/* Free everything in temporary store. Argument 1 shows that, we have just finished compiling a function */

```
permanent_allocation (1);
}
```

We have understood the working of functions and expressions. Let's add the required semantic actions for functions and expressions.

```
input:  fndef body      { build_function(); };
```

```
fndef:  NAME '(' ')'    { build_function_decl ($1); };


exp:  NUM     { $$ = build_int_2 ($1, $1 >= 0 ? 0 : -1); }
     | exp '+' exp  { $$ = build (PLUS_EXPR, integer_type_node, $1, $3); }
     | exp '-' exp  { $$ = build (MINUS_EXPR, integer_type_node, $1, $3); }
     | exp '*' exp  { $$ = build (MULT_EXPR, integer_type_node, $1, $3); }
     | exp '/' exp  { $$ = build (TRUNC_DIV_EXPR, integer_type_node, $1, $3); }
     | exp '%' exp  { $$ = build (TRUNC_MOD_EXPR, integer_type_node, $1, $3); }
     | '-' exp %prec NEG { $$ = build1 (NEGATE_EXPR, integer_type_node, $2); }
     | '(' exp ')'  { $$ = $2; }
```

Example demo program 1

```
foo()
   begin
        1+2*3
   end
```

7.4 Variable Declaration

What is our current status? We have with us a function and an expression, which is good for nothing. Our language will be more beautiful with the presence of variables, which can be assigned and returned.

So the next step is to declare variables. As usual we have to build trees and convert it to rtl. When we are making a variable declaration as

```
var a;    // I prefer PASCAL syntax. Just a matter of taste.
```

we have to store the value of 'a' because when it is used later in an expression, say a+1, we have to use the same 'a'.

We can define two arrays of our own var_name[ ] and var_decls[ ] as shown.

```
char *var_name[100];    //global array ie. initialized to zero.
tree var_decls[100];    //global array
```

I hope the reader has understood that the first one is used to store the name of variables in "demo" and the second one to store the corresponding tree structures that are built.

Let's directly look at the code.

```
void add_var(char *name)
{
  int i;
/*Add the name given, as the last name in the array */
  for(i=0;var_name[i];i++);
/* Store the name */
  var_name[i] = name;
/*Build the tree structure for the variable declared
All the parameters are explained before.
Thus we have name of the variable stored in var_name[] and
tree stored in var_decls[ ]*/
  var_decls[i] =
```

```
  build_decl (VAR_DECL, get_identifier(name), integer_type_node);
/* Explained before*/
  DECL_CONTEXT (var_decls[i]) = fndecl;
/*We are just making the initial value of the variable declared
as zero. This is a matter of taste. */
  DECL_INITIAL (var_decls[i]) = integer_zero_node;
/*push to the current scope. Explained before*/
  pushdecl (var_decls[i]);
/* Emit the rtl for the variable declaration*/
  expand_decl (var_decls[i]);
/* Emit the rtl for the initialization. ie. Initialized to zero*/
  expand_decl_init (var_decls[i]);
}
```

From now onwards I am not going to give the bison file fully. I'll be providing only the segments modified by us in each stage. Please do refer the bison file provided above in case of any doubt.

The bison segment for variable declaration is given by

```
declarations:    /*empty */
              |  declarations VAR NAME          { add_var($3); };
```

Combining the variable declaration with the above syntax for functions and expressions, we have another example of a "demo" program.

Example demo program 2

```
foo()
  begin
        var a
        var b
        1+2*3-1
  end
```

7.5 Assignments

A number of variables and expressions one below the other doesn't help us in any way to do anything useful. For that we need assignments. Let's study the tree structure building and rtl conversion for assignments.

The general format of an assignment statement is "a = 10". Operations required for it is to store the value, 10 in 'a'. We had created the tree structure for 'a' and stored it in var_decls[] at the time of variable declaration. So our duty is to retrieve the tree structure of 'a' and assign the value, say 10 in it.

We seek the help of a function called "get_var" to retrieve the tree structure of the variable. The code is given below

```
tree get_var(char *name)
{
  int i;
/*Search for the name "name" in the variable name table, var_name[]
  for(i=0;var_name[i];i++)
/*If found, return the corresponding tree structure of "name"
     if( !strcmp (var_name[i], name))
        return var_decls[i];
}
```

The above function gets us the tree structure of the variable. The only task left is to build the tree structure for assignment and the rtl conversion. It is achieved by the following function.

```
make_assign (tree vartree, tree valtree)
{
  tree assign;

/* Create the tree. Explained before.
  assign =
     build (MODIFY_EXPR, integer_type_node, vartree, valtree);

/*Non zero values means that there is a side effect and re evaluation
  of the whole expression could produce a different value. The
  optimization phase takes this into consideration.
*/
  TREE_SIDE_EFFECTS (assign) = 1;

/* Indicates that the tree node is used */
  TREE_USED (assign) = 1;

/* Emit the rtl for the assign tree */
  expand_expr_stmt (assign);
}
```

We have also studied the tree creation and rtl conversion for assignment construct. Now it is time to give the semantic action in the bison file for assignment.

```
assignstmt:     NAME = exp      { make_assign ( get_var($1), $3); };
```

7.6 Expressions revisited

The expressions that we are using now is capable of working with numbers only. ie. We can give expressions as "1+2", "1+2*3" etc. But it is not possible for us to work with variable names as "a+1", "b+a−5" etc.

Let us modify our expressions to include variable names also. The task of inclusion is simple. We have to get the tree structure of the variable name. We have a function "get_var" in our hand which is capable of doing it.

So let us look at the semantic action for this

```
exp:    NAME            { $$ = get_var ($1); };
```

Example demo program 3

```
hello()
  begin
        var i
        var j
        var k
        i = 10
        j = 20
        k = i + j
  end
```

7.7 Return

The above program would be more beautiful if it is possible for us to return the sum calculated. So our next step will be the inclusion of return construct.

I have already mentioned about the tree creation for 'return' when I explained the 'function'. But at that time we returned only zero. Now let us return an expression in the place of zero.

```
ret_stmt (tree expr)
{
  tree ret;
/* build the tree node for return. The arguments are explained
    before. 'fndecl' is the global variable that we have defined
    before, for our function.
*/
  ret =
      build (MODIFY_EXPR, integer_type_node, DECL_RESULT(fndecl),
              expr);

/*emits the rtl */
  expand_return (ret);
}
```

Let's look the bison segment straightly

```
returnstmt:     RETURN exp      { ret_stmt ($2) ; };
```

Example demo program 4

```
hello()
  begin
        var i
        var j
        var k
        i = 10
        j = 20
        k = i + j
        return k
  end
```

7.8 Conditional statement

In the case of 'if' construct our task is different from the above. We have to clearly give information to GCC regarding where the if construct begins, where the 'else' part begins and where the if construct ends.

GCC supplies us rtl statements, which are capable of performing the above tasks. The rtl statements are given below

```
expand_start_cond (tree cond, int exitflag)
```

It generates an rtl for the start of an if−then. 'cond' is the expression whose truth is to be checked. Consider exitflag to be zero.

```
expand_start_else ()
expand_end_cond ()
```

These causes the rtl code generation for start of 'else' and for the end of 'if construct' respectively.

Now we can use the above functions in our bison file.

```
ifstmt   :       ifpart thenpart elsepart        { expand_end_cond (); };
ifpart   :       IF exp                  { expand_start_cond ($2, 0); };
thenpart:        THEN compstmts          { expand_start_else (); };
elsepart:        ELSE compstmts ENDIF            ;
```

Example demo program 5

```
example()
  begin
        var x
        var y
        x = 100
        if (x) then
                y = 1
        else
                y = 0
        endif
        return y
  end
```

7.9 Loops

Loops are same as conditional statements. We have to provide the start of the loop, end of the loop and the expression to be checked for truthness.

The rtl statements meant for the above operations are

```
struct nesting *expand_start_loop (int exit_flag)
expand_exit_loop_if_false (struct nesting *whichloop, tree cond)
expand_end_loop()
```

The first one denotes the rtl for start of a loop. 'exit_flag' is zero. The return type is struct nesting *, which is used in the second statement. The second statement is used for generating a conditional jump to exit the current loop, if 'cond' evaluates to zero. The third statement is the rtl for end of a loop. It generates a jump back to the top of the loop. The ideas will be clearer with the bison file.

```
whilestmt:       head loopbody   { expand_end_loop(); };

head :           WHILE exp       { struct nesting *loop;
                                   loop = expand_start_loop (0);
                                   expand_exit_loop_if_false (loop, $2);
                                 };

loopbody:        compstmts ENDWHILE      ;
```

Example demo program 6

```
test()
  begin
        var i
        i = 100
        while (i)
                i = i-1
        endwhile
        return i
  end
```

8. Demo front end

A front end for our demo language is provided at http:// www.gec−fsug.org/gcc−front/demo.tar.gz.

It has been tested with the GCC (version 2.95.3) back end. You can make experiments with the front end. Try to add more constructs. Constructs such as for loops, repeat until, case structure, else if, can be easily added in our demo language.

If you're successful with the creation of a new front end, please do send it to us, so that it would be a great help for the newcomers in the field of GCC front ends.

9. See also

When I began the creation of a new front end the only source of information was an article 'Writing a Compiler Front End' from 'Using Maintaining and Enhancing Cobol for the GNU Compiler Collection' by Tim Josling.

Details regarding GCC can be obtained from the home page of GCC. Information was also obtained from the GCC manual that is available with each distribution of GCC.

But the vital source of information is the front ends of other languages. But I'll never recommend them for the beginner. Once you are on track then they will be of great help to you.