Report of Compiler Project

Linghan Zheng
Department of Computer Science and Engineering
SJTU
Shanghai,China
Student ID : 5130309485

1. INTRODUCTION

This project is divided into two parts: 1) Implement a simplified compiler front-end including a lexical analyzer and a syntax analyzer with the tool Flex and Yacc; 2) Implement a code generator which returns a LLVM assembly program. In the project, the source language is Small-C, a C-like language containing a subset of the C programming language; while the target code is LLVM instructions that can be run on LLVM.

The report will contain three main sections : Section 2 will describe the implementation of lexical and syntax analyzer; Section 3 will describe the implementation of code generator; Section 4 will show some examples.

2. A LEXICAL-SYNTAX ANALYZER

2.1 lex.l

The lexical analyzer reads in the Small-C source code and recognize tokens according to regular definitions.

The first part contains necessary C declarations and includes to use throughout the lex specifications.

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <assert.h>
    #include <string.h>
    #include "y.tab.h"

    extern void yyerror(char *s);
    int line=0;
    struct Node
    {
        ......
    };

    struct Node* setNode(char *n,char *t,int l)
    {
        ......
    }

    int getValue(char *txt)
    {
        ......
    }
%}
```

Here, I created a struct named **Node** which is the basal element of a parse tree.

```
struct Node
    {
        char name[100];     // char name[100] records the name of the node;
        char type[10];   // char type[10] records the type of the node such as
                         "COMMA" for "," and "ID" for an identifier;
        int line;   // int line records where the token appears;
        int value;   // int value records the value of a "INT" node
        struct Node *son; // The last two items will be used to create a parse tree
        struct Node *bro;
    };
```

The function "setNode" is used to create the node for each token. It works to record the name, type and line number. Besides, it gets the value when getting a "INT" node with the help of function "getValue".

Then we define some regular expressions:

```
    delim   [ \t]
    ws  {delim}+                        // for white space
    letter [A-Za-z]
```

```
digit   [0-9]
id  {letter}({letter}|{digit})*            //for identifiers
int  0?(x|X)?(([0-7]+)|([0-9A-F]+))      // for integers including hex and oct
```

In the next part, translation rules are written to describe the corresponding actions when meeting different tokens.

Take “if” as an example:

```
if        {yylval.node=setNode(yytext, "IF",line);return(IF);}
```

When read in “if”, the lexical analyzer will set a Node with the name as“if”, type as “IF” and record the current line number. At last, return IF to .y file to notify syntax analyzer that there exists a “IF” token.

To sum up, Lex source is a table of regular expressions and corresponding program fragments. As each expression appears in the input, the corresponding fragment is executed.

By using the command: **$flex lex.l** , lex complier will get “**lex.yy.c**” from “**lex.l**”.

2.2 parse.y

Yacc is an LALR parser generator. **parse.y** can parser the token recognized by **lex.l** file and check whether they follow the grammar rules.

parse.y file also consists of three parts:

```
Definitions
(tokens)
%%
Rules
%%
User code
```

1) Definition -- C source code, include files,etc.

```
%{

    #include ......                  // include files
    void yyerror(char *s);            // to handle errors

    struct Node   { ...... };  // struct Node, the same as definitions lex.l

    FILE *ifile,*ofile;         // help read and write files
```

```
        /*Flags*/        // contain flags which help code generator

        struct Node *root;          // root of a parse tree
        struct Node *makeTree(char *n,int num,...)    { ..... }
        // function to load a new node on the tree

        struct Node *empty(char *n)      { ...... }
        // function to load a new node with an empty son on the tree

        struct symbol   // a struct to record the information of variables
        {
            ......        // this part will be explained in detail in next section
        };
        struct symbol* symTable[27][20];   // symbol Table, explain in 3.7
        ......
        void _PROGRAM(struct Node *root);   // functions for code generators
    ......
%}
```

2) Definition -- Tokens

All terminal symbols recognized by lexical analyzer should be declared through **%token.**

```
%union{ struct Node* node;}
%token <node> IF
......
```

Using the above sentences, I point out that every token is a Node.

For operators, **%left** or **%right** shows its associativity. Operators in the same group have the same precedence; across groups, precedence increases going down. Thus **LP RP LB RB DOT** have the highest precedence while **ASSIGNOP** has the lowest.

3) Grammar Rules

```
//programs, the root node for a parse tree
PROGRAM : EXTDEFS    ;

//external definitions
EXTDEFS : EXTDEF EXTDEFS   |  /*empty*/     ;

//one external definition
EXTDEF : SPEC EXTVARS SEMI            // such as:   int a;
     | SPEC FUNC STMTBLOCK      // such as:   int main() {}
     ;
```

```
//external variables : declaration | dec , extvars | empty
EXTVARS : DEC | DEC COMMA EXTVARS | /*empty*/
    ;

// types: int | struct
SPEC : TYPE   | STSPEC   ;

STSPEC : STRUCT OPTTAG  LC DEFS RC // struct name { ... } -- for declaration
    | STRUCT ID       // struct name   -- for use
    ;

OPTTAG :  ID  |   /*empty*/  ;


//variables : id | for array | for struct element
VAR : ID   | VAR LB INT RB | VAR DOT ID;
FUNC : ID LP PARAS RP     ; // example "function ( paras )"
PARAS : PARA COMMA PARAS |  PARA|  /*empty*/  ;
PARA : SPEC VAR   ; // example "int a"
STMTBLOCK : LC DEFS STMTS RC    ;     // { declaration statement }
STMTS : STMT STMTS   | /*empty*/;
STMT : EXP SEMI
    | STMTBLOCK
    | RETURN EXP SEMI
    | IF LP EXP RP STMT ESTMT
    | FOR LP FEXP SEMI FEXP SEMI FEXP RP STMT
    | CONT SEMI
    | BREAK SEMI
    ;
ESTMT : ELSE STMT   |/*empty*/   ;        // statements after "else"
DEFS : DEF DEFS   | /*empty*/  ;          // define
DEF : SPEC DECS SEMI ;                      // declarations
DECS : DEC COMMA DECS    | DEC  ;
DEC : VAR|  VAR ASSIGNOP INIT   ;           // examples " var" "var = 1"
INIT : EXP|  LC ARGS RC   ;          // EXP for a simple var,
                                     while LC ARGS RC for an array
EXP : EXP HBOP EXP           // example   a * b
    | EXP LBOP EXP                   / / example   a + b
    | EXP SHIFT EXP         // example   a >> b
    | EXP COMP EXP          // example   a >= b
    | EXP EQU EXP      // example   a = b
    | EXP NE EXP       // example   a != b
    | EXP BAND EXP          // example   a & b
    | EXP BXOR EXP          // example   a ^ b
    | EXP BOR EXP      // example   a | b
    | EXP LAND EXP          // example   a && b
    | EXP LOR EXP      // example   a || b
    | LNOT EXP         // example !x
    | PIN EXP          // example i++
    | PDE EXP          // example i--
    | BNOT EXP         // example ^x
    | LBOP EXP         // example -a
    | LP EXP RP             // (expression)
    | ID LP ARGS RP
    | ID ARRS               // example id[1]
    | ID ARRS ASSIGNOP EXP            // example id[1]=1
    | EXP DOT ID            // example st.xx
    | EXP DOT ID ASSIGNOP EXP              // example st.xx = 1

    | INT                   ;
```

```
FEXP : EXP |  /*empty*/  ; // expressions which can be empty, FOR
ARRS : LB EXP RB ARRS   | /*empty*/   ;  // [i] for arrays
ARGS : EXP COMMA ARGS   | EXP  ;    // arguments for function
```

Since that it's difficult to explain all of the rules, I won't waste more time to talk about them them one by one now. But I will introduce them more clearly when describing code generator in section 4.
According to the codes, we also have some other {action} added after the rules, such as

```
ARGS : EXP COMMA ARGS {$$=makeTree("ARGS",3,$1,$2,$3);}
ARRS :| /*empty*/ {$$=empty("ARRS");}
```

In "**makeTree**" function, the first argument "ARGS" means that we create a new Node named as ARGS, and it has 3 sons "EXP" "COMMA" "ARGS" (the three Nodes have already been created before). The new node will remember the pointer of its first son, and then the first son will remember the pointer of its next brother. The following is a vivid picture to show the relationship:

```
ARGS
    --(son)-->  EXP  --(bro) -->  COMMA  --(bro) -->  ARGS
```

Then, the function returns a struct Node * pointer and assign it to $$.
In “**empty**” function, we also create a new Node, but its son is NULL.

Therefore, along with the process of parse, we can get a parse tree with Node as element. Then, by using DFS, we can print the tree.

4) User Code
To be explained in Code Generator section.

3. CODE GERERATOR

In this part, I created a lot functions to help translate the code, such as _PROGRAM() and _EXTDEFS(). Usually one non-terminal has one corresponding function with the same name. However, considering that some symbols have different features in different situations, I created more than one functions for them, _VARNOINIT() and _VARINIT(), _DECIN() and _DECEX(), for example. Besides, there also exist some symbols that have no corresponding functions because of their strong relationship with others. At the same time, it is easy to notice that there are some functions named _IFAND(), showHead() and showFoot(). They are designed to handle some special problems.

During the implementation, I realized the test-cases one by one. Thus, I re-edited these functions over and over again, adding some sentences once a while. But to make the report looks more complete, I decided to explain my codes one function after another, and used some examples to make it more clear.

3.1 void showHead() and void showFoot()

➢ Source Codes:

```
void showHead()
{
    printf("target datalayout =
\"e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-
f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:3
2:64-S128\"\n");
    printf("target triple = \"x86_64-pc-linux-gnu\"\n\n");

    if(nodeFlag!=NULL)
    {
        _DECIN(nodeFlag);
        printf("\n; Function Attrs: nounwind\n");
        printf("declare void @llvm.memcpy.p0i8.p0i8.i64(i8* nocapture, 
i8* nocapture readonly, i64, i32, i1) #2\n\n");
        nodeFlag=NULL;
    }

    if(rFlag==1 || wFlag==1)
        printf("@.str = private unnamed_addr constant [3 x i8] 
c\"%%d\\00\", align 1\n");
}

void showFoot()
{
    if(rFlag==1)
        printf("\ndeclare i32 @__isoc99_scanf(i8*, ...) #1\n");
    if(wFlag==1)
        printf("\ndeclare i32 @printf(i8*, ...) #1\n");

    printf("\nattributes #0 = { nounwind uwtable 
\"less-precise-fpmad\"=\"false\" \"no-frame-pointer-elim\"=\"true\" 
\"no-frame-pointer-elim-non-leaf\" \"no-infs-fp-math\"=\"false\" 
\"no-nans-fp-math\"=\"false\" \"stack-protector-buffer-size\"=\"8\" 
\"unsafe-fp-math\"=\"false\" \"use-soft-float\"=\"false\" }\n" );

    if(rFlag==1 || wFlag==1)
    printf("attributes #1 = { \"less-precise-fpmad\"=\"false\" 
\"no-frame-pointer-elim\"=\"true\" \"no-frame-pointer-elim-non-leaf\" 
\"no-infs-fp-math\"=\"false\" \"no-nans-fp-math\"=\"false\" 
\"stack-protector-buffer-size\"=\"8\" \"unsafe-fp-math\"=\"false\" 
\"use-soft-float\"=\"false\" }\n");

    printf("\n!llvm.ident = !{!0}\n");
    printf("\n!0 = metadata !{metadata !\"Ubuntu clang version 
3.4-1ubuntu3 (tags/RELEASE_34/final) (based on LLVM 3.4)\"}");
}
```

➢ Explanation:

By observation, every .ll file has a fixed “head” and “foot” showing some information about the program. For programs with “read” and “write” there are some additional sentences. rFlag and wFlag are used to show whether we have the two functions (In fact, they exist in every test-case, but I still use a judgment to decide whether to add these information.) _DECIN() part will be explained in 3.10.

*3.2 void _PROGRAM(struct Node *root)*

- Source Codes:

```
void _PROGRAM(struct Node *root)
{
    showHead();
    _EXTDEFS(root->son);
    showFoot();
}
```

- Explanation:

This function is quite easy. For any .c file read in, it was firstly parsed and a parse tree has already been built. Then in the main() function, the root of the tree will be sent to function _PROGRAM(root);

Void _PROGRAM(struct Node *root) reads in root as the parameter. Then it calls showHead() to print out the "head" information. Next, it calls _EXTDEFS() and sends the son of root, which is a node named EXTDEFS, as the parameter. Finally, it shows the foot information by calling showFoot();

*3.3 void _EXTDEFS(struct Node *n)*

- Source Codes:

```
void _EXTDEFS(struct Node *n)
{
    BlockLevel=0;
    if(n->son==NULL) return;
    struct Node *p=n->son;
    _EXTDEF(p);
    _EXTDEFS(p->bro);
}
```

- Explanation:

According to the grammar rule, EXTDEFS may have two possible methods of extension: EXTDEF EXTDEFS | empty. So I added a judgment that if(n->son==NULL), which means it has no son, we don't do anything and just return. Otherwise, we call EXTDEF and EXTDEFS in order to handle its two son-nodes. Notice that there is one strange sentence: BlockLevel = 0; it initializes BlockLevel, shows now we are out of a function. You will know why we need this in function _STMTBLOCK();

*3.4 void _EXTDEF(struct Node *n)*

➢ Source Codes:

```
void _EXTDEF(struct Node *n)
{
    struct Node *son1=n->son;
    struct Node *son2=son1->bro;
    struct Node *son3=son2->bro;

    if(!strcmp(son2->name,"EXTVARS")) //SPEC EXTVARS SEMI
    {
        struct Node *son11=son1->son;  //TYPE or STSPEC
        if(!strcmp(son11->name,"int"))  //int ...
            _EXTVARSTYPE(son2);
        else  //struct ...
            _EXTDEFSTRUCT(n);
    }
    else //SPEC FUNC STMTBLOCK
    {
        _FUNC(son2);
        _STMTBLOCK(son3);
    }
}
```

➢ Explanation:

Similarly, there are two possibilities. 1) SPEC EXTVARS SEMI: this rule is to define something outside of a function. SPEC has two forms TYPE and STSPEC, the first is for “int” while the next one is for “struct”. Since we have different methods to handle the two situations, I designed function “EXTVARSTYPE” and “EXTDEFSTRUCT” to deal with them separately. 2) SPEC FUNC STMTBLOCK: this rule is very important because it actually contains a complete function. But it’s easy to handle it here that we just call _FUNC() and _STMTBLOCK().

*3.5 void _EXTVARSTYPE(struct Node *n)*

➢ Source Codes:

```
void _EXTVARSTYPE(struct Node *n)
{
        if(n->son==NULL) return;
        if(n->son->bro==NULL) //DEC
            _DECEX(n->son);
        else //DEC COMMA EXTVARS
        {
            _DECEX(n->son);
            _EXTVARSTYPE(n->son->bro->bro);
        }
    }
```

➢ Explanation:

It handles EXTVARS nodes with INT as its brother. If it doesn't have a son,just return; if it only has one son, call _DECEX() to print declarations; otherwise, call _DECEX first to handle the first declarations and then call itself to handle the next declarations.

*3.6 void _DECEX(struct Node *n)*

➢ Source Codes:

```
    void _DECEX(struct Node *n)
    {
        exDecFlag=1;
        if(n->son->bro==NULL) //VAR
            _VARNOINIT(n->son);

        else//VAR ASSIGNOP INIT
            _VARINIT(n->son);
        exDecFlag=0;
    }
```

➢ Explanation:

It helps to declare outside of a function. _VARNOINIT() and _VARINIT() will handle VAR and VAR ASSIGNOP INIT respectively. Because VAR will have different codes to handle ex- and in- declarations, we should let it know now we want to declare outside. Thus, exDecFlag is set as 1 before calling VAR and it is set as 0 again after handling by VAR.

*3.7 void _EXTDEFSTRUCT(struct Node *n)*

➢ Source Codes(part 1):

```
void _EXTDEFSTRUCT(struct Node *n)
{
    struct Node *stspec=n->son->son;

    struct Node *stname=stspec->son->bro;
    if(stname->bro==NULL) //STRUCT ID
    {
        char *stid=(char*)malloc(sizeof(char)*100);
        stid=stname->name;

        struct Node *tmp=n->son;

        while(tmp)
        {
            char *id=(char*)malloc(sizeof(char)*100);
            id=tmp->bro->son->son->son->name;

            int group;
            group=id[0]-'a';
            if(group<0)group=id[0]-'A';
            if(id[0]=='_')group=26;

            int i=0;
            while(symTable[group][i]) i++;
            symTable[group][i]=(structsymbol*)malloc(sizeof(struct
symbol));
            struct symbol *sym=symTable[group][i];

            sym->head=(char*)malloc(sizeof(char));
            strcpy(sym->head,"@");
            strcpy(sym->word,id);
            strcpy(sym->st,stid);
            sym->type='e'; // struct element

            printf("@%s = common global %%struct.%s zeroinitializer,
align 4\n",id,stid);

            tmp=tmp->bro->son->bro;
        }
    }
```

- Explanation(part 1):

This part is to handle STRUCT ID. When meeting this code, we are actually declare instantiating an object of a struct. For example: struct something A; STRUCT ID represents “struct something” part.

Here, I have to stress a sequence of instructions which begins from “char *id = ...” and ends with “struct symbol *sym ...”. If you read through all my project, you will find this sequence appears a lot of times. First, pointer “char *id” records the name of the variable we meet. Then, “int group” records which group it should in. In the part “2.2 1) Definitions”, I have showed the **symTable[27][20]**, 27 contains 26 letters and underline “_”; 20 means we can at most record 20 variables in each group. Next, “int i” helps to find an empty space for new symbol. Once a suitable space is found, new symbol will be created to store the new variable. “struct symbol *sym” is set to represent this new symbol. Now, we can set or get information of it.

In struct **symbol**, we have these information:

①char *head; records head of different kinds of symbols, for example ‘@’ for global variable and ‘%’ for local one.

②char word[100]; records the name of the symbol.

③ char type; records the type, ‘g’ meas “global”, ‘l’ means “local”, ‘a’ means “argument”, ‘e’ means “element” and ‘s’ means “struct”.

④int size; if the variable is a array, “size” records its size.

⑤char st[100]; if the variable is an element of a struct, it records its struct name.

⑥ int stno; for an element of a struct, it records the location in a struct. For example, struct XX { int A; int B }, B’s stno is 2.

⑦int reg; records the register number of a variable.

⑧int mem; rerods the memory number of a variable.

After getting sym, we set the head \word \stid for the declaration of the new struct element. Then print the sentence “.. .common global %struct ...” in .ll file to declare the element.

➢ Source Codes(part 2):

```
else // STRUCT OPTTAG { DEFS }
    {
        char *id=(char*)malloc(sizeof(char)*100);
        id=stname->son->name;

        int group;
        group=id[0]-'a';
        if(group<0)group=id[0]-'A';
        if(id[0]=='_')group=26;

        int i=0;
        while(symTable[group][i]) i++;
        symTable[group][i]=
            (struct symbol*)malloc(sizeof(struct symbol));
        struct symbol *sym=symTable[group][i];

        sym->head=(char*)malloc(10*sizeof(char));
        strcpy(sym->head, "%struct. ");
        strcpy(sym->word,id);
        sym->type='s';

        printf("%s%s = type {",sym->head,sym->word);
        stEleNum=0;
        _DEFSSTRUCT(stname->bro->bro,id);
        printf(" }\n");
    }
}
```

➢ Explanation(part 2):

This part helps to define a struct outside the function. For example, struct XX { int A; int B; int C}　Similarly, we create a new symbol, and then set its head \name \type. Next, call _DEFSTURCT() to print the information of the element of this struct, like ABC in the example above.

3.8 void _DEFSTRUCT(struct Node *n,char *stname)

void _DEFSTRUCT(struct Node *n,char *stname)

➢ Source Codes:

```
void _DEFSSTRUCT(struct Node *n,char *stname)
{
    if(n->son==NULL)return;
    else
    {
        _DEFSTRUCT(n->son,stname);
        _DEFSSTRUCT(n->son->bro,stname);
    }
}

void _DEFSTRUCT(struct Node *n,char *stname)
{
    struct Node *dec = n->son->bro->son;

    char *id=(char*)malloc(sizeof(char)*100);
```

```
        id=dec->son->son->name;

        int group;
        group=id[0]-'a';
        if(group<0)group=id[0]-'A';
        if(id[0]=='_')group=26;

        int i=0;
        while(symTable[group][i]) i++;
        symTable[group][i]=
            (struct symbol*)malloc(sizeof(struct symbol));
        struct symbol *sym=symTable[group][i];

        strcpy(sym->word,id);
        strcpy(sym->st,stname);
        sym->type='i'; // in struct
        sym->stno=stEleNum;
        if(stEleNum==0)
            printf(" i32");
        else
            printf(", i32");
        stEleNum++;
    }
```

➢ Explanation:

"char *stname" is special in these two functions. We set this parameter because it's important to remember which struct the element belongs. In _DEFSTRUCT, we create a new symbol again, and then record its name \stname \stno. "stEleNum"helps to remember the location in a struct.

*3.9 void _VARNOINIT(struct Node *n)*

➢ Source Codes(part 1):

```
    void _VARNOINIT(struct Node *n)
    {
        if(n->son->bro==NULL)  //ID
        {
            char *id=(char*)malloc(sizeof(char)*100);
            id=n->son->name;

            // store the new var in symTable
            int group;
            group=id[0]-'a';
            if(group<0)group=id[0]-'A';
            if(id[0]=='_')group=26;

            int i=0;
            while(symTable[group][i]) i++;
            symTable[group][i]
               =(struct symbol*)malloc(sizeof(struct symbol));
            struct symbol *sym=symTable[group][i];

            if(exDecFlag==0) // DECIN
            {
               sym->head=(char*)malloc(sizeof(char));
               strcpy(sym->head,"%");
               strcpy(sym->word,id);
               sym->type = 'l';  //local
               printf("  %s%s = alloca i32, align 4\n",
                     sym->head,sym->word);
            }
            else // DECEX
            {
               sym->head=(char*)malloc(sizeof(char));
               strcpy(sym->head,"@");
               strcpy(sym->word,id);
               sym->type = 'g';  //global
               printf("%s%s = common global i32 0, align 4\n",
                    sym->head,sym->word);
            }
    }
```

➢ Explanation(part 1):

_VARNOINIT() handles the variables without assignment. This part shows the instructions to deal with ID case. At first, we also create a new symbol to remember the new variable. Then we check where the variable is declared inside the function or outside the function. For the first case, we set its head as “%”, type as “l”, and print a sentence which looks like “ %var = alloca i32, align 4 ”; for the second case, we set its head as “@”, type as “g”, and print a sentence which lools like “@var = common global i32 0, align 4 ”. I know .ll file should contain these sentences by observation.

➢ Source Codes(part 2):

```
    else // VAR [ INT ]
    {
        int num = n->son->bro->bro->value;

        char *id=(char*)malloc(sizeof(char)*100);
        id=n->son->son->name;

        int group;
        group=id[0]-'a';
        if(group<0)group=id[0]-'A';
        if(id[0]=='_')group=26;

        int i=0;
        while(symTable[group][i]) i++;
        symTable[group][i]
            =(struct symbol*)malloc(sizeof(struct symbol));
        struct symbol *sym=symTable[group][i];
        sym->size=num;

        if(exDecFlag==0) // DECIN
        {
            sym->head=(char*)malloc(sizeof(char));
            strcpy(sym->head,"%");
            strcpy(sym->word,id);
            sym->type = 'l';  //local
            printf("  %s%s = alloca [%d x i32], align 4\n",
                    sym->head,sym->word,num);
        }

        else // DECEX
        {
            sym->head=(char*)malloc(sizeof(char));
            strcpy(sym->head,"@");
            strcpy(sym->word,id);
            sym->type = 'g';  //global
            printf("%s%s = common global [%d x i32] zeroinitializer, align 
4\n",sym->head,sym->word,num);
        }
    }
```

- Explanation(part 2):

This part shows the instructions to deal with VAR[int] case. The only difference between these instructions and those above is the sentence to be printed.

*3.10 void _VARINIT(struct Node *n)*

- Source Codes:

```
void _VARINIT(struct Node *n)
{
    struct Node *init=n->bro->bro;

    if(n->son->bro==NULL)  //ID = INIT
    {
        int value = init->son->son->value;
        char *id=(char*)malloc(sizeof(char)*100);
        id=n->son->name;

        // store the new var in symTable
        int group;
        group=id[0]-'a';
        if(group<0)group=id[0]-'A';
        if(id[0]=='_')group=26;

        int i=0;
        while(symTable[group][i]) i++;
        symTable[group][i]
            =(struct symbol*)malloc(sizeof(struct symbol));
        struct symbol *sym=symTable[group][i];

        if(exDecFlag==0) // DECIN
        {
            sym->head=(char*)malloc(sizeof(char));
            strcpy(sym->head,"%");
            strcpy(sym->word,id);
            sym->type = 'l';  //local
        printf("  %s%s = alloca i32, align 4\n",sym->head,sym->word);
        printf("  store i32 %d, i32* %s%s, align 4\n",
                value,sym->head,sym->word);
        }
        else // DECEX
        {
            sym->head=(char*)malloc(sizeof(char));
            strcpy(sym->head,"@");
            strcpy(sym->word,id);
            sym->type = 'g';  //global
    printf("%s%s = global i32 %d, align 4\n",sym->head,sym->word,value);
```

```
        }
}

else  // VAR [ INT ] = INIT
{
    int num = n->son->bro->bro->value;

    char *id=(char*)malloc(sizeof(char)*100);
    id=n->son->son->name;

    // store the new var in symTable
    int group;
    group=id[0]-'a';
    if(group<0)group=id[0]-'A';
    if(id[0]=='_')group=26;

    int i=0;
    while(symTable[group][i]) i++;
    symTable[group][i]
        =(struct symbol*)malloc(sizeof(struct symbol));
    struct symbol *sym=symTable[group][i];
    sym->size=num;

    if(exDecFlag==0) // DECIN
    {
        sym->head=(char*)malloc(sizeof(char));
        strcpy(sym->head,"%");
        strcpy(sym->word,id);
        sym->type = 'l';  //local

        if(nodeFlag!=NULL) // first declare
        {
    printf("\n@%s.%s = private unnamed_addr constant [%d x i32] [",
            funcName, sym->word, num);
            _ARGSINIT(init->son->bro);
            printf("], align 4\n");
        }

        else // declare inside function
        {
        printf("  %s%s = alloca [%d x i32], align 4\n",
                sym->head,sym->word,num);
        printf("  %%%d = bitcast [%d x i32]* %%%s to i8*\n",
                regNum,num,sym->word);
            printf("  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %%%d, i8* 
bitcast ([%d x i32]* @%s.%s to i8*), i64 8, i32 4, i1 false)\n",
                regNum++,num,funcName,sym->word);
            }
        }
        else // DECEX
        {
            sym->head=(char*)malloc(sizeof(char));
            strcpy(sym->head,"@");
            strcpy(sym->word,id);
            sym->type = 'g';  //global
            printf("%s%s = global [%d x i32] [",
                sym->head,sym->word,num);
            _ARGSINIT(init->son->bro);
            printf("], align 16\n");
        }
    }
}
```

➢ Explanation:

This function is very similar to _VARNOINIT(). It also deals with ID and VAR[int] cases respectively, and it also discusses DECIN and DECEX in each case. The biggest difference is that we have to call _ARGSINIT() to handle the assignment. Besides, you may notice that there is a sequence of codes bold. The reason is that for an array like var[2]={3,4}; we have to print a sentence looks like “@functionName.var = private unnamed_addr constant [size x i32] [i32 3, i32 4], align 4”. Besides, this is why I call _DECIN() in showHead();

*3.11 void _ARGSINIT(struct Node *n)*

➢ Source Codes:

```
    void _ARGSINIT(struct Node *n)
    {
        struct Node *exp = n->son;
        if(n->son->bro==NULL) //EXP
            printf("i32 %d",exp->son->value);

        else // EXP COMMA ARGS
        {
            printf("i32 %d, ",exp->son->value);
            _ARGSINIT(exp->bro->bro);
        }
}
```

➢ Explanation:

This function helps _VARINIT() to finish printing the arguments, like “[i32 3, i32 4]” in the above example.

*3.12 void _FUNC(struct Node *n)*

➢ Source Codes:

```
void _FUNC(struct Node *n)
{
    regNum = 1;
    tempReg = 0;

    printf("\n; Function Attrs: nounwind uwtable\n");
    printf("define i32 @%s(",n->son->name);
    struct Node *paras = n->son->bro->bro;
    struct Node *pson = paras->son;
    if(pson==NULL) paraFlag = 0;
    else
    {
        paraFlag = 1;
        _PARAS(paras);
    }
    printf(") #0 ");
}
```

➢ Explanation:

It prints the title of a function like “main()” in the source codes. For those functions having parameters, we have to call _PARAS() to print them.

*3.13 void _PARAS(struct Node *n) and void _PARA(struct Node *n)*

➢ Source Codes:

```
void _PARAS(struct Node *n)
{
    if(n->son==NULL)return;
    if(n->son->bro==NULL)
```

```
        _PARA(n->son);
    else
    {
        _PARA(n->son);
        printf(", ");
        _PARAS(n->son->bro->bro);
    }
}

void _PARA(struct Node *n)
{
    struct Node *para = n->son->bro->son;

    paraArr[paraNum] = (char*)malloc(sizeof(char)*60);
    strcpy(paraArr[paraNum],para->name);
    paraNum++;
    printf("i32 %%%s",para->name);
}
```

➢ Explanation:

As I said just now, _PARAS() and _PARA() are used to print the parameters of a function. For example, we will print “define i32 @gcd(i32 %x, i32 %y) #0” before the codes of gcd function in .ll file, and _PARA() is used to printf(i32 %x, i32 %y). What’s more, it is necessary to remember the parameters. Thus, I designed a table named paraArr to remember that we have parameters for the function. Then, in the codes of gcd function, we will print sentence “%2 = alloca i32, align 4\n store i32 %x, i32* %2, align 4” to store the parameter in a register.

*3.14 void _STMTBLOCK(struct Node *n) and void _STMTS(struct Node *n)*

➢ Source Codes:

```
void _STMTBLOCK(struct Node *n)
{
    if(!BlockLevel)  // for function block
    {
        printf("{\n");
        printf("  %%%d = alloca i32, align 4\n",regNum++);
        int i=0;
        while(paraArr[i])
        {
            printf("  %%%d = alloca i32, align 4\n",regNum);
            printf("  store i32 %%%s, i32* %%%d, align 4    ;
               store  para \n",paraArr[i],regNum);
            char* id = (char*)malloc(sizeof(char)*100);
            strcpy(id,paraArr[i]);

            //store the paraments in symTable
            int group;
            group=id[0]-'a';
            if(group<0)group=id[0]-'A';
            if(id[0]=='_')group=26;

            //find a vacuous space of argument
            int j=0;
            while (symTable[group][j]) j++;
            symTable[group][j]
                   =(struct symbol*)malloc(sizeof(struct symbol));

            struct symbol *sym=symTable[group][j];
            strcpy(sym->word,id);
            sym->head=(char*)malloc(sizeof(char));
            strcpy(sym->head,"%");
            sym->type = 'a';
            sym->mem = regNum;

            regNum++;free(id);
            free(paraArr[i]);paraArr[i]=NULL;
            i++;
        }
    }

    struct Node *defs=n->son->bro;
    struct Node *stmts=defs->bro;
    _DEFS(defs);
    if(!BlockLevel && paraNum==0)
       printf("  store i32 0, i32* %%1\n");
    paraNum=0;
    _STMTS(stmts);

    if(!BlockLevel) printf("}\n");
}

void _STMTS(struct Node *n)
{
        if(n->son==NULL)return;
        _STMT(n->son);
        _STMTS(n->son->bro);
    }
```

➢ Explanation:

If(!BlockLevel) means this statement block is used for a function. Then we have some specific sentences to print out. When there are some parameters, we have to record them, set their head as ‘a’ and remember its location after storing. _STMTBLOCK() calls _DEFS() and _STMTS().

STMTS represents a series of statements, so _STMTS() calls _STMT() and itself to deal with the statements one by one.

*3.15 void _STMT(struct Node *n)*

➢ Source Codes(easy part):

```
    void _STMT(struct Node *n)
    {
        struct Node *son1=n->son;
        if(son1->bro==NULL)  //STMTBLOCK
        {
            BlockLevel++;
            _STMTBLOCK(son1);
            BlockLevel--;
            return;
        }

        if(!strcmp(son1->name,"EXP"))  //EXP SEMI
        {
            _EXP(son1);
            return;
        }
```

➢ Explanation(easy part):

This part is so easy to understand that I don’t want to waste time to explain them. One point to notice: BlockLevel will increase before calling _STMTBLOCK again and decrease after calling.

➢ Source Codes(IF part):

```
if(!strcmp(son1->name,"if"))  //IF LP EXP RP STMT ESTMT
{
    struct Node *exp=son1->bro->bro;
    struct Node *stmt=exp->bro->bro;
    struct Node *estmt=stmt->bro;

    if(!strcmp(exp->son->bro->name,"&&")) // exp => exp && exp
    {
        _IFAND(n);
        return;
    }

     _EXP(exp);

    if(!strcmp(exp->son->bro->type,"DOT")) // if(st.id)
    {
        printf("  %%%d = icmp ne i32 %%%d, 0\n",
            regNum,regNum-1);
        regNum++;
    }

    // if (x&1)  (x|1)  (x^1)
    if(!strcmp(exp->son->bro->name,"&") ||
        !strcmp(exp->son->bro->name,"|") ||
        !strcmp(exp->son->bro->name,"^"))
    {
        printf("  %%%d = icmp ne i32 %%%d, 0\n",
            regNum,regNum-1);
        regNum++;
    }


    if(estmt->son!=NULL)  //with else
    {
        //DOT CASE not finished

        if(tfFlag==1) //if(1==1)
        {
            _STMT(stmt);
            tfFlag=-1;
            return;
        }

        else if(tfFlag==0) //if(1==2)
```

```
{
    _STMT(estmt->son->bro);
    tfFlag=-1;
    return;
}


int labelT=regNum;

printf("  br i1 %%%d, label %%%d",
       regNum-1,labelT);
long loc1=ftell(ofile); // the loc of %labelF

//first STMT
regNum++;
_STMT(stmt);
int labelF=regNum;

// back to loc1,add the number
fseek(ofile,loc1,0);
printf(", label %%%d\n",labelF);

// repeat STMT
printf("\n; <label>:%d (T)\n",labelT);
regNum=labelT+1;
_STMT(stmt);
long loc2=ftell(ofile);// the loc of br

// first ESTMT
regNum=labelF+1;
_STMT(estmt->son->bro);
int after=regNum;

// back to loc2
fseek(ofile,loc2,0);
printf("  br label %%%d\n",after);

// repeat ESTMT
printf("\n; <label>:%d (F)\n",labelF);
regNum=labelF+1;
_STMT(estmt->son->bro);
printf("  br label %%%d\n",after);

// after
```

```
        printf("\n; <label>:%d (A)\n",after);
        regNum=after+1;

    }

    else   //without else
    {
        int label=regNum;

        printf("  br i1 %%%d, label %%%d",
               regNum-1,label);
        long loc=ftell(ofile); // the loc of %label2

        //STMT
        regNum++;
        _STMT(stmt);
        int after=regNum;

        // back to loc,add the number
        fseek(ofile,loc,0);
        printf(", label %%%d\n",after);

        // repeat STMT
        printf("\n; <label>:%d (T)\n",label);
        regNum=label+1;
        returnFlag=0;
        _STMT(stmt);
        if(returnFlag==0)
            printf("  br label %%%d\n",after);
        returnFlag=0;

        // after
        printf("\n; <label>:%d (A)\n",after);
        regNum=after+1;
    }
}
```

- Explanation(IF part):

This part is very important! There are two situations, one is with "else", while the other is without "else". The usual execution order is like the following figure. Obviously, it is important to know where is the next instruction. That's why we need sentences like "br i1 %5, label %6, label %10" or "br label %15" in .ll file. The problem is how can we know the location of these locations.

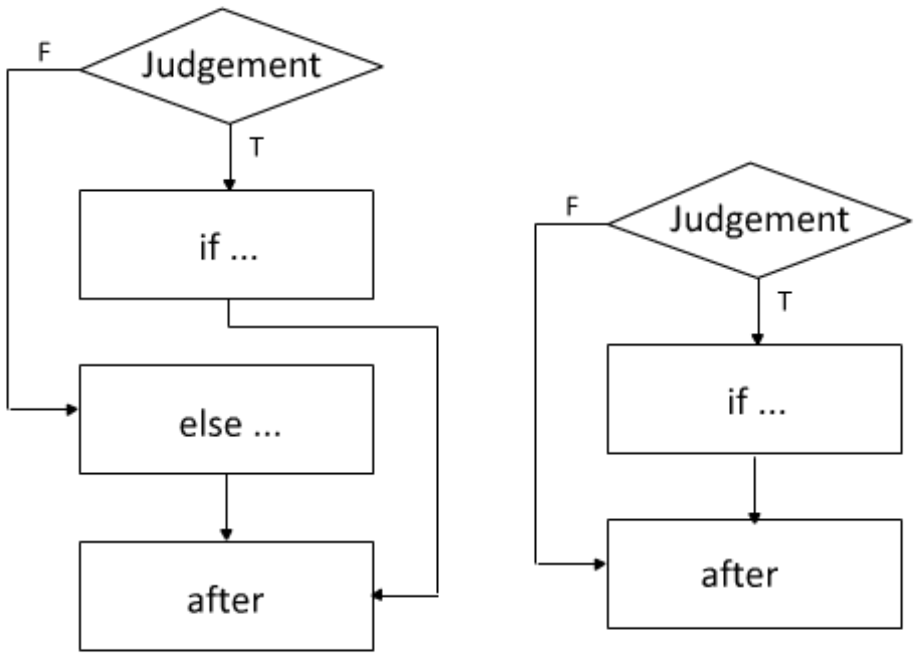


Fig 1. Structure of IF-ELSE case and ONLYIF case

However, it is not easy since that before we get the label of “else...”, the codes of “if...” part have to be translated first; before we get the label of “after”, the codes of “else...” part have to be translated first. To solve the problem, I took a brave try. Maybe it is not the best way, but at least it works.

I used two functions, one is **ftell()** and the other is **fseek()**. The idea is quite easy. IF-ELSE case: I first translate STMT of “if...” part, get the label of else, which is labelF. Then I go back to judgment part, and print the number of labelF. Next I translate STMT of “if...”part again and then “else” part, get the label of “after”. Now again, I go back to the end of “if...” part, print the label of “after” and translate “else...”part again. By doing so, I can finish a complete translation with right labels of every block. the process of IF case is very similar.

There are some other points to notice: 1) When there exists “&&” symbol, cases are much more difficult, thus, I created a special function __IFAND to handle. 2) For judgment like “if(A.x)”, we have to print one sentence “ %d = icmp ne i32 %d-1, 0” to show this is a judgment instead of a simple variable.

- Source Codes(RETURN part):

```
if(!strcmp(son1->name,"return")) //RETURN EXP SEMI
    {
        returnFlag = 1;
        struct Node *exp=son1->bro;
        if(strcmp(exp->son->name,"0")) //no return 0;
        {
            _EXP(exp);
            printf("  store i32 %%%d, i32* %%1\n",regNum-1);
            printf("  %%%d = load i32* %%1\n",regNum);
            printf("  ret i32 %%%d\n",regNum++);
        }
        else
            printf("  ret i32 0\n");//return 0;
    }
```

- Explanation(RETURN part):

Two possible "return" codes will appear: return 0; and return exp;

A simple "return 0;" is easy to deal with. "ret i32 0" is enough. "return exp" is more complex. First, we have to call _EXP() to translate the exp part. After translating, a sentence like " store i32 %d, i32 * %1" has to be print, which means that the result of exp is stored in %1. Then "%d = load i32 * %1;" load the value, and "ret i32 %d". In this way, the result of exp can be returned.

- Source Codes(FOR part):

```
if(!strcmp(son1->name,"for")) // FOR LOOP
    {
        struct Node *exp1 = son1->bro->bro;
        struct Node *exp2 = exp1->bro->bro;
        struct Node *exp3 = exp2->bro->bro;
        struct Node *stmt = exp3->bro->bro;

        _FEXP(exp1);

        int start = regNum;
        printf("  br label %%%d\n",start);

        printf("\n; <label>:%d (J)\n",start);
        regNum=start+1;
        _FEXP(exp2);

        long loc=ftell(ofile);
        int judge=regNum-1;
        int after=regNum;

        // execute the code after for loop
        regNum=after+1;
        _STMTS(n->bro);
        int label=regNum;
        fseek(ofile,loc,0);
        printf("  br i1 %%%d, label %%%d, label %%%d\n",
            judge,label,after);

        printf("\n; <label>:%d (A)\n",after);
        regNum=after+1;
        _STMTS(n->bro);
        n->bro->son=NULL; // already exe after,set NULL

        printf("\n; <label>:%d (LOOP)\n",label);
        regNum=label+1;
        _STMT(stmt);
        printf("  br label %%%d\n",regNum);

        printf("\n; <label>:%d (C-A)\n",regNum++);
        _FEXP(exp3);
        printf("  br label %%%d\n\n",start);
    }
```

➢ Explanation(FOR part):

This is another very important part. The standard structure is shown as the left figure.But I change its order into the right one.

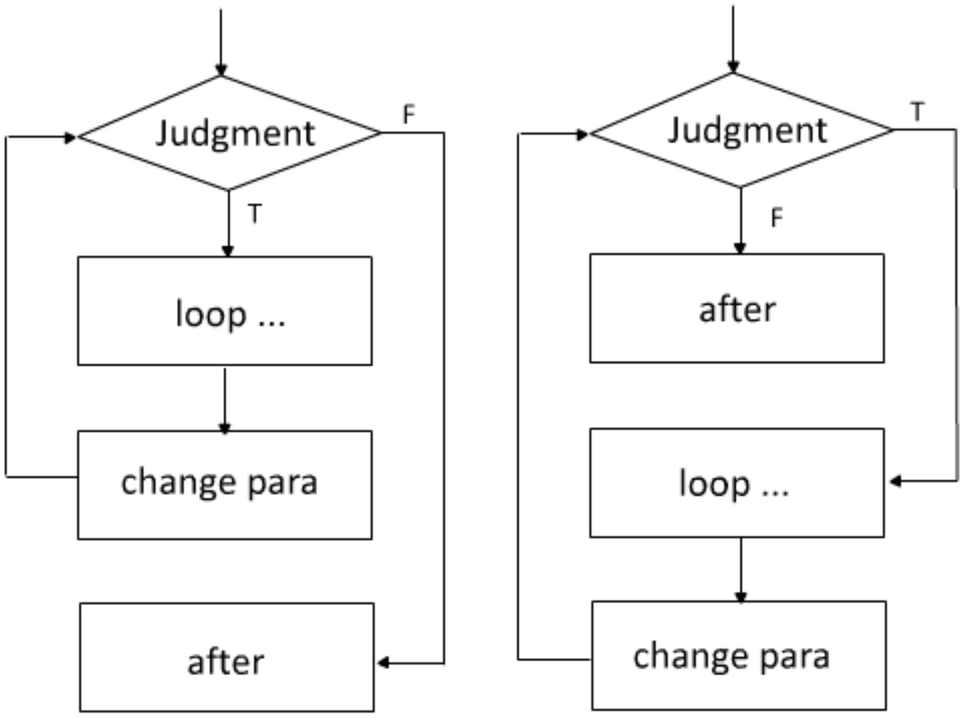


Fig 2. (a) the standard order (b) my order

The idea is very similar to IF. One problem is that, if I use the standard order, I have to re-translate “loop”and ”change para”, however, the two parts are far longer than “after” part in our test-cases. Thus, I decided to change its order and in this way, statement in “loop” part will only be translated once, saving much time.

*3.16 void _IFAND (struct Node *n)*

➢ Source Codes:

```
void _IFAND(struct Node *n)
{
    struct Node *exp = n->son->bro->bro;
    struct Node *stmt = exp->bro->bro;

    struct Node *exp3 = exp->son->bro->bro;
    struct Node *exp1 = exp->son->son;
    struct Node *exp2 = exp1->bro->bro;

    int after;

    // to get int after
    _EXP(exp1);
    long loc=ftell(ofile);
    int label=regNum;

    regNum++;
    _EXP(exp2);
    regNum++;
    _EXP(exp3);
    regNum++;
    _STMT(stmt);
    after = regNum;

    fseek(ofile,loc,0);
    printf("  br i1 %%%d, label %%%d, label %%%d\n",
           label-1,label,after);

    printf("\n; <label>:%d (J2)\n",label);
    regNum = label+1;
    _EXP(exp2);
    printf("  br i1 %%%d, label %%%d, label %%%d\n",
           regNum-1,regNum,after);
```

```
        printf("\n; <label>:%d (J3)\n",regNum);
        regNum++;
        _EXP(exp3);
        printf("  br i1 %%%d, label %%%d, label %%%d\n",
                regNum-1,regNum,after);

        printf("\n; <label>:%d (T)\n",regNum);
        regNum++;
        _STMT(stmt);
        printf("  br label %%%d\n",regNum);

        printf("\n; <label>:%d (A)\n",regNum);
        regNum++;
    }
```

➢ Explanation:

This function is designed especially for "if(A && B && C)" instruction. The order of this instruction is as follows.

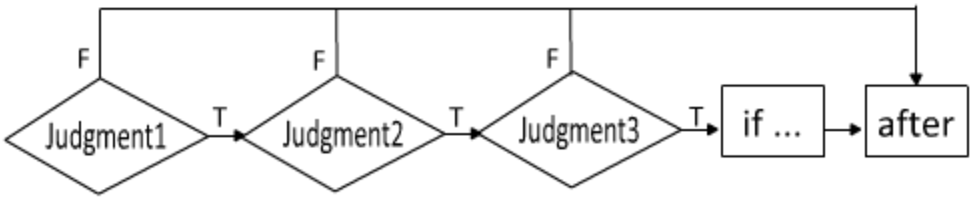


Fig 3. Order of if(A && B && C);

Because of its complex structure, the translation is a little bit hard to understand. But the idea is just like that of IF. I translated judgment1\2\3 and if-statement one by one and at last get the label of "after" part. Then, re-translated them and added the label information this time.

3.17 void _DEFS(...) void _DEF(...) void _DECS(...) void _DECIN(...)

➢ Source Codes:

```
    void _DEFS(struct Node *n)
    {
        if(n->son==NULL)return;
        else
        {
            _DEF(n->son);
            _DEFS(n->son->bro);
        }
    }
```

```
void _DEF(struct Node *n)
{
    _DECS(n->son->bro);
}

void _DECS(struct Node *n)
{
    struct Node *son1=n->son;
    struct Node *son2=son1->bro;
    if(son2==NULL)
    {
        _DECIN(son1);
    }
    else
    {
        _DECIN(son1);
        _DECS(son2->bro);
    }
}

void _DECIN(struct Node *n)
{
    if(n->son->bro==NULL) //VAR
        _VARNOINIT(n->son);

    else//VAR ASSIGNOP INIT
        _VARINIT(n->son);
}
```

- Explanation:

  The four functions are used to deal with definitions inside a function.

3.18 char _EXP(struct Node *n)*

- Source Codes(INT case):

```
    if(son1->bro==NULL)  //INT
    {
        char *value = (char*)malloc(sizeof(char)*20);
        sprintf(value,"%d",son1->value);return value;
    }
```

- Explanation(INT case):

_EXP() is the most significant function of this project. The biggest difference is that it can return a char*. In the simplest INT case, it get the value of INT exp, and return its value as a string.

- Source Codes(read case):

```
    if(!strcmp(son1->name,"read")) //read case
        {
            struct Node *exp = son2->bro->son;

            if(!strcmp(exp->son->name,"EXP")) // read struct.ID
            {
            char *st= (char*)malloc(sizeof(char)*100);
            strcpy(st,exp->son->son->name);
            char *id = (char*)malloc(sizeof(char)*100);
            strcpy(id,exp->son->bro->bro->name);

            int group;
            group=id[0]-'a';
            if(group<0)group=id[0]-'A';
            if(id[0]=='_')group=26;

            //find the argument
            int i=0;
            while (strcmp(id,symTable[group][i]->word)) i++;
            struct symbol *sym=symTable[group][i];

            printf("  %%%d = call i32 (i8*, ...)* @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8]* @.str, i32 0, i32 0), i32* getelementptr inbounds (%%struct.%s* @%s, i32 0, i32 %d))\n",
                regNum++,sym->st,st,sym->stno);

            }

            else // read ID
            {
            char *id = (char*)malloc(sizeof(char)*100);
            readFlag=1;
            id = _EXP(exp);
            readFlag=0;
            printf("  %%%d = call i32 (i8*, ...)* @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8]* @.str, i32 0, i32 0), i32* %s)\n",regNum++,id);
            }
            return NULL;
        }
```

➢ Explanation(read case):

In small-c language, we created a standard function named “read”, which is similar to “scanf” in c language but simpler. The object of read can be a struct element as well as ID. In the former case, we find the element in symbol Table first, and then print out the information including its struct name, number in a struct. In the latter case, we can easily print information about its ID name. The translation is imitating the sentence “scanf(“%d”,x)”.

➢ Source Codes(write case):

```
    if(!strcmp(son1->name,"write")) //write case
        {
            char *args = (char*)malloc(sizeof(char)*100);
            struct Node *son3 = son2->bro;
            args = _EXP(son3->son);
            if(!strcmp(son3->son->son->type,"INT"))  //write(int)
                printf("  %%%d = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([3 x i8]* @.str, i32 0, i32 0), i32 %s)\n",regNum,_EXP(son3->son));

            else if(!strcmp(son3->son->son->type,"LBOP")) //write(-int)
                printf("  %%%d = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([3 x i8]* @.str, i32 0, i32 0), i32 -%s)\n",regNum,_EXP(son3->son->son->bro));

            else
                printf("  %%%d = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([3 x i8]* @.str, i32 0, i32 0), i32 %%%d)\n",regNum,regNum-1);

            regNum++;return NULL;
        }
```

➢ Explanation(write case):

Write case is easy. I just finished it by imitating different examples of printf.

- Source Codes(ID ARRS case):

* The codes of this part is very long and there are too many same sentences, so I don’t want to put them here. You may find them in parse.y file.

- Explanation(ID ARRS case):

There are four main situations of EXP that begins with “ID ARRS”. They are:

1) ID (no ARRS). Example: abc. In this case, we find the variable in symbol table first, and them print the corresponding load instructions for variables of different types. However, when the ID appears in read function, we don’t need to load. Actually, I think **an Error detector** could be added here if it cannot be found in symTable.

2) ID (no ARRS) ASSIGNOP EXP. Example: x = 1; or x = a+b; Firstly, expression on the right side should be sent to _EXP(); then, we find ID in symTable, and translate it. If ASSIGNOP is “=” and the R-expression is a simple int, store the value into the ID; if the R-expression is not int, we store the value in newest register into the ID. For other ASSIGNOPs, we have more information to print out, such as “add” instruction for +=.

3) ID ARRS. Example: a[1] or a[i]. If the index is int, get the array “a” and choose the element we want, then load it into a register. Otherwise, we have to compute the value of index first. After _EXP(), a sentence “ %d = sext i32 %d to i64” is needed to extent the index value. Till then, we can find the element in array and load it.

4) ID ARRS ASSIGNOP EXP. Example: a[1] = 2; or a[i] = y; Similar to 2), expression on the right side should be dealt with at the beginning, and then we use the methods in 3) to get the element and then store the value.

- Source Codes(BOP case):

```
if(!strcmp(OP,"*"))strcpy(show,"mul nsw");
if(!strcmp(OP,"/"))strcpy(show,"sdiv");
if(!strcmp(OP,"%"))strcpy(show,"srem");
if(!strcmp(OP,"-"))strcpy(show,"sub nsw");
if(!strcmp(OP,"+"))strcpy(show,"add nsw");
if(!strcmp(OP,"<<"))strcpy(show,"shl");
if(!strcmp(OP,">>"))strcpy(show,"ashr");
if(!strcmp(OP,">"))strcpy(show,"icmp sgt");
if(!strcmp(OP,">="))strcpy(show,"icmp sge");
if(!strcmp(OP,"<"))strcpy(show,"icmp slt");
if(!strcmp(OP,"<="))strcpy(show,"icmp sle");
if(!strcmp(OP,"=="))strcpy(show,"icmp eq");
if(!strcmp(OP,"!="))strcpy(show,"icmp ne");
if(!strcmp(OP,"&"))strcpy(show,"and");
if(!strcmp(OP,"^"))strcpy(show,"xor");
if(!strcmp(OP,"|"))strcpy(show,"or");
```

- Explanation(BOP case):

The codes above are the most important part of BOP. The main idea of BOP is to compute op1 and op2 first and then translate instructions depending on its BOP symbol and expression type.

- Source Codes(!EXP case):

```
if(!strcmp(son1->type,"LNOT")) // !exp
    {
        char* tmp = _EXP(son2);
        printf("  %%%d = icmp eq i32 %%%d, 0\n",regNum,regNum-1);
                regNum++;
    }
```

- Explanation(!EXP case):

In fact, !EXP is simple. These is still one point to stress. When clang .ll file, you will find that for expression (!y), the translation is “icmp ne...”, but I put it as “icmp eq”. In the first case, branch instruction will be “br il, label yF, label yT”; in the latter case, branch instruction will be “br il label yT, label yF”, which is easier to handle.

- Source Codes(func-call case):

```
if(!strcmp(son2->name,"("))  // ID (ARGS) -- function( , )
    {
        if(!strcmp(son2->bro->son->son->type,"INT")) // args is int
        {
            printf("  %%%d = call i32 @%s(i32 %d)\n",
                regNum++,son1->name,son2->bro->son->son->value);
            return NULL;
        }

        _ARGSFUNC(son2->bro);
        printf("  %%%d = call i32 @%s(",regNum++,son1->name);
        int i=0;

        while(paraArr[i])
        {
            ......
        }
        printf(")\n");
```

```
            return NULL;
        }
```

➢ Explanation(func-call case):

These sequence of codes deal with function call. Since that some functions may have arguments, we have to call _ARGSFUNC() to get these arguments, and while paraArr[i] != NULL, information about these arguments are printed.

➢ Source Codes(DOT case):

```
    if(!strcmp(son2->type,"DOT"))  // EXP DOT ID ...
    {
        if(son2->bro->bro==NULL) // EXP DOT ID
        {
            ......
        printf("  %%%d = load i32* getelementptr inbounds (%%struct.%s*
@%s, i32 0, i32 %d), align 4\n",regNum++,sym->st,st,sym->stno);
        }

        else // EXP DOT ID ASSIGNOP EXP
        {
            ......
            if(!strcmp(rexp->son->type,"INT"))
            {
            printf("   store  i32  %d,  i32*  getelementptr  inbounds
(%%struct.%s* @%s, i32 0, i32 %d), align 4\n",
            rexp->son->value,sym->st,st,sym->stno);
            }

            else
            {
            _EXP(rexp->son->bro);

            printf("   store  i32  %%%d,  i32*  getelementptr  inbounds
(%%struct.%s*     @%s,     i32     0,     i32     %d),     align
4\n",regNum-1,sym->st,st,sym->stno);
            }
        }
```

➢ Explanation(DOT case):

This part was designed deal with struct. 1) EXP DOT ID, such as AA.xx.The procedure is as follows: find the id in symbol table -> get the information like struct name and location in struct -> load it from the struct. 2) EXP DOT ID ASSIGNOP EXP, such as AA.xx = 1. Similarly, find the id in symbol table -> handle the expression on the right side -> store the value in struct element.

- Source Codes(++i case):

```
if(!strcmp(son1->type,"PIN"))  // ++ EXP
    {
        char* id=(char*)malloc(sizeof(char)*100);
        id = _EXP(son1->bro);
        printf("  %%%d = add nsw i32 %%%d, 1\n",regNum,regNum-1);
        printf("  store i32 %%%d, i32* %s, align 4\n",regNum,id);
        regNum++;
    }
```

- Explanation(++i case):

PIN means “personal increase”. It is easy to deal with. We can just divide “++EXP” into two parts, “++” and “EXP”. Then we use _EXP() to process exp and print “add” information to process “++”. At last, we store the value back into the variable.

*3.19 void _FEXP(struct Node *n)*

- Source Codes:

```
void _FEXP(struct Node *n)
{
    if(n->son==NULL)return;

    _EXP(n->son);
    if(n->son->son->bro->bro==NULL)  // if fexp: x
    {
        printf("  %%%d = icmp ne i32 %%%d, 0\n",
            regNum,regNum-1);
                regNum++;
    }
}
```

- Explanation:

FEXP is designed for “for expression”. It is special because its son can be NULL. If the EXP is a simple ID, we have to add a sentence “... icmp ne ...”, which is used for “judgment” in for expression.

*3.20 void _ARGSFUNC(struct Node *n)*

- Source Codes:

```
void _ARGSFUNC(struct Node *n)
{
    if(n->son->bro==NULL) //EXP
    {
        char* args = (char*)malloc(sizeof(char)*100);
        args = _EXP(n->son)+1;
        paraArr[paraNum] = (char*)malloc(sizeof(char)*60);
        if(tempReg) // 参数为计算表达式
            strcpy(paraArr[paraNum],"***");
        else
            strcpy(paraArr[paraNum],args);
        paraNum++;
    }

    else // EXP, ARGS
    {
        char* args = (char*)malloc(sizeof(char)*100);
        args = _EXP(n->son)+1;
        paraArr[paraNum] = (char*)malloc(sizeof(char)*60);
        if(tempReg) // 参数为计算表达式
            strcpy(paraArr[paraNum],"***");
        else
            strcpy(paraArr[paraNum],args);
        paraNum++;

        _ARGSFUNC(n->son->bro->bro);
    }
}
```

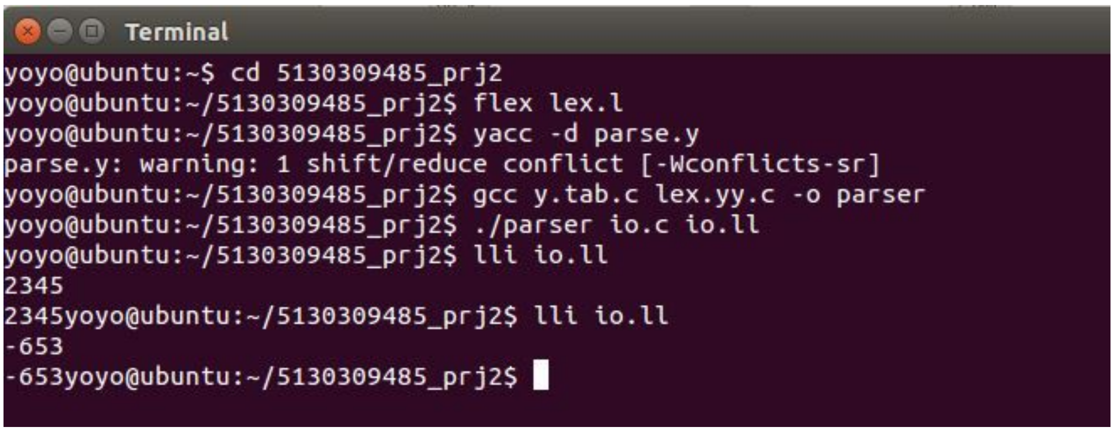
- Explanation:

_ARGSFUNC() is used to deal with argument in function. It stores information of arguments and if the argument is computed by an expression(tempReg stores the results), we store “***” as the name of argument to note.

Now, I have already explained all the functions to generate the codes. Maybe it’s sounds not so difficult now, but it really needs time to design them little by little. I am happy that I finally finish these functions by myself! The next section will show some examples of test-cases.

4.TESTCASES

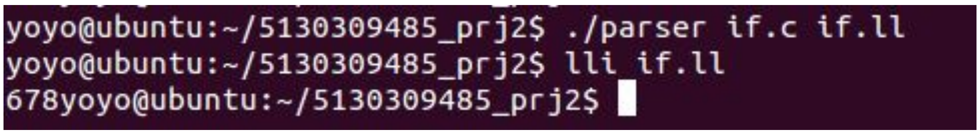
4.1 io.c



```
yoyo@ubuntu:~$ cd 5130309485_prj2
yoyo@ubuntu:~/5130309485_prj2$ flex lex.l
yoyo@ubuntu:~/5130309485_prj2$ yacc -d parse.y
parse.y: warning: 1 shift/reduce conflict [-Wconflicts-sr]
yoyo@ubuntu:~/5130309485_prj2$ gcc y.tab.c lex.yy.c -o parser
yoyo@ubuntu:~/5130309485_prj2$ ./parser io.c io.ll
yoyo@ubuntu:~/5130309485_prj2$ lli io.ll
2345
2345yoyo@ubuntu:~/5130309485_prj2$ lli io.ll
-653
-653yoyo@ubuntu:~/5130309485_prj2$
```

Fig 4. Example of io.ll

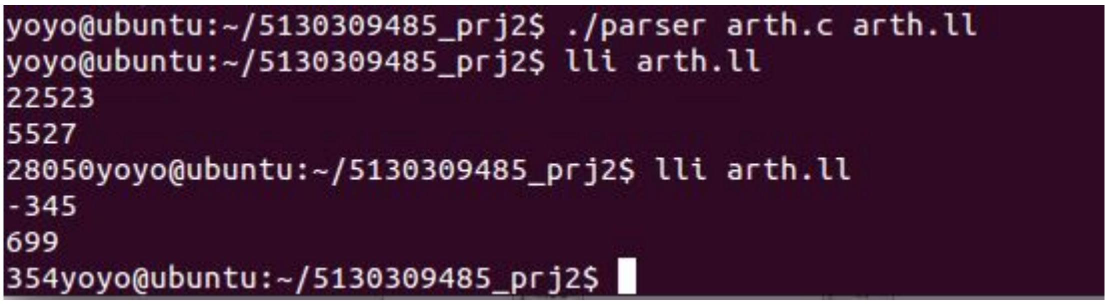
4.2 if.c



```
yoyo@ubuntu:~/5130309485_prj2$ ./parser if.c if.ll
yoyo@ubuntu:~/5130309485_prj2$ lli if.ll
678yoyo@ubuntu:~/5130309485_prj2$
```

Fig 5. Example of if.ll

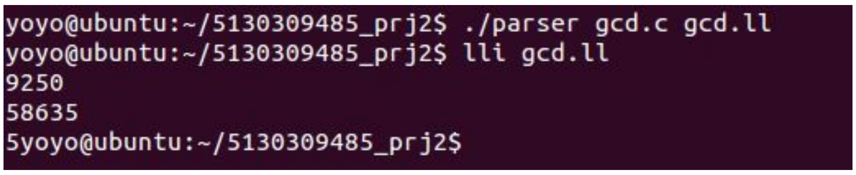
4.3 arth.c



```
yoyo@ubuntu:~/5130309485_prj2$ ./parser arth.c arth.ll
yoyo@ubuntu:~/5130309485_prj2$ lli arth.ll
22523
5527
28050yoyo@ubuntu:~/5130309485_prj2$ lli arth.ll
-345
699
354yoyo@ubuntu:~/5130309485_prj2$
```

Fig 6. Example of arth.ll

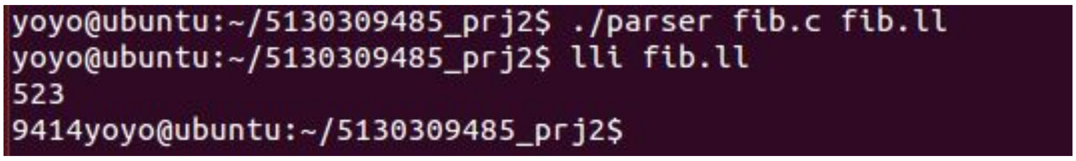
4.4 gcd.c



```
yoyo@ubuntu:~/5130309485_prj2$ ./parser gcd.c gcd.ll
yoyo@ubuntu:~/5130309485_prj2$ lli gcd.ll
9250
58635
5yoyo@ubuntu:~/5130309485_prj2$
```

Fig 7. Example of gcd.ll

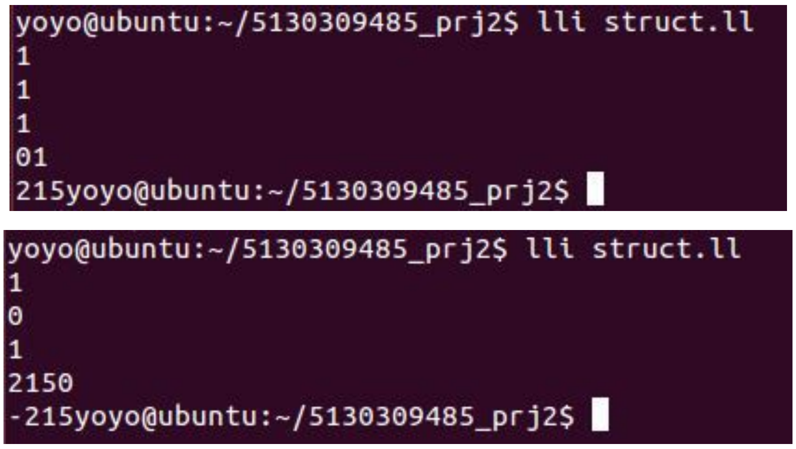
4.5 fib.c



```
yoyo@ubuntu:~/5130309485_prj2$ ./parser fib.c fib.ll
yoyo@ubuntu:~/5130309485_prj2$ lli fib.ll
523
9414yoyo@ubuntu:~/5130309485_prj2$
```

Fig 8. Example of fib.ll

4.6 struct.c

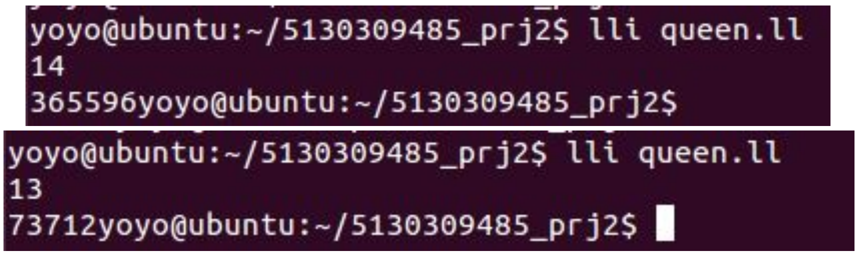


```
yoyo@ubuntu:~/5130309485_prj2$ lli struct.ll
1
1
1
01
215yoyo@ubuntu:~/5130309485_prj2$
```

```
yoyo@ubuntu:~/5130309485_prj2$ lli struct.ll
1
0
1
2150
-215yoyo@ubuntu:~/5130309485_prj2$
```

Fig 9. Example of struct.ll

4.7 queen.c



```
yoyo@ubuntu:~/5130309485_prj2$ lli queen.ll
14
365596yoyo@ubuntu:~/5130309485_prj2$
```

```
yoyo@ubuntu:~/5130309485_prj2$ lli queen.ll
13
73712yoyo@ubuntu:~/5130309485_prj2$
```

Fig 10. Example of queen.ll

5.ACKNOWLEGMENT

First ans foremost, I would like to show my deepest gratitude to my RA, Lerong Chen, who have guided me on this project with great patience. Without his enlightening instruction, impressive kindness and clear explanation, I could not have completed my project.

I would also like to thank my parents for their encouragement and support.