Report of Compiler Principle

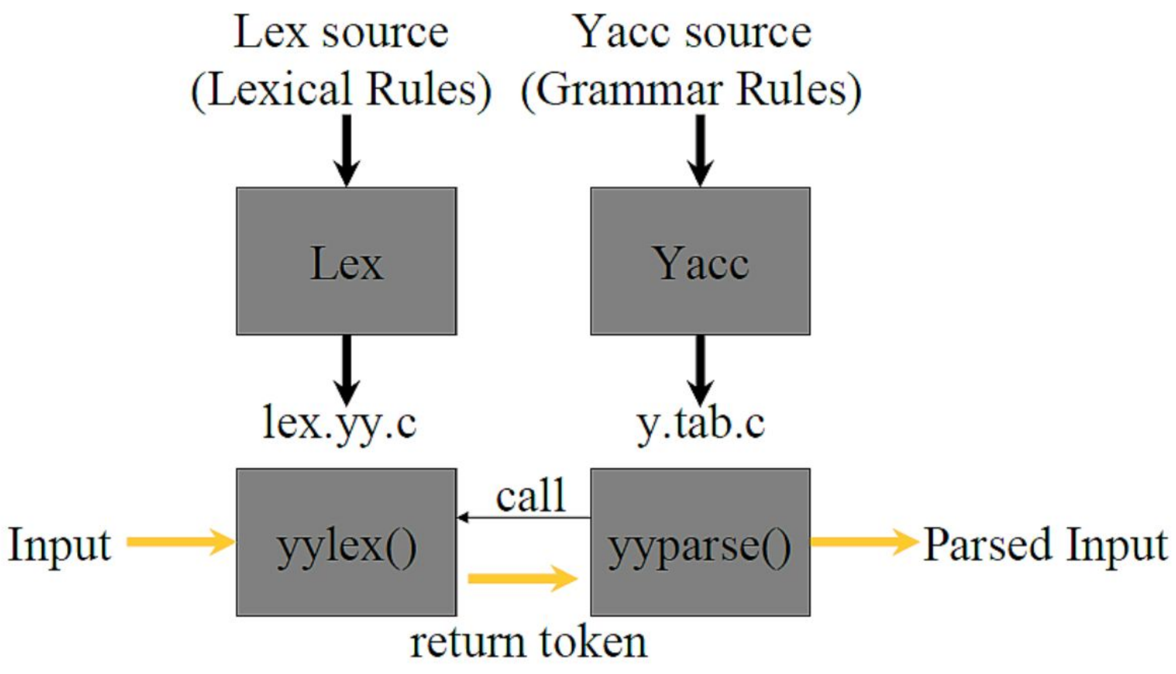
Project 2

Dongyu Jia (5120309607)
01/20/2016

Introduction

In this project, you are required to implement a code generator to translate the intermediate representation, which is produced by your syntax analyzer implemented in project 1, into LLVM instructions. Your code generator should return a LLVM assembly program, which can be run on LLVM (http://llvm.org/). After finishing this project, you will get a compiler, which can translate Small-C source programs to LLVM assembly programs.

Lexical Analyzer



Here we will use flex as the tool of lexical analyzer and we can work under the framework with only spcifying the regex expression and the according action.

Examples

Since the logic are pretty the same for all symbols, we use define to avoid duplicate and if you want to change some logic, you only need to fix the define instead of fixing them one by one.

```
#define printReturn(x)  return x;
#define load yylval.string = strdup(yytext);
……..
"/" {load printReturn(DIV_OP)}
"%" {load printReturn(MOD_OP)}
"<" {load printReturn(LT_OP)}
```

Line number

For the debug purpose for me and for whoever use this compiler, it is very important that you tell user why and where it does not work.

```
[\n] {yylineno = yylineno+1;}
```

Numbers

We will only handle positive value here, as for minus sign before it, we will handle it later in semantic analyzer. Note that we will treat number start with 0, and 0x, 0X as valid number. And alphabet will only appear in literals start with 0x or 0X.

```
([0-9]*|0[xX][0-9a-fA-F]+)          { load printReturn(INT) }
```

Comments

For comments, there are two formats.
one line comments is handled by such:

```
"//"(.)*  {}
```

And comments block will be handled by a more trick strategy which involves state transition:

```
%x COMMENT
...
 "/*"       BEGIN(COMMENT);
<comment>{
[^*]*
"*"+[^*/]*
\n
"*"+"/"        BEGIN(INITIAL);//   ******/
}
```

Syntax Analyzer

In the file smallc.y, we define the syntax rules. We define precedence and translation rules. Along with which we also specify the structure of the node so that we can get a parse tree after parsing the file.

Precedence of IF and IF ELSE Statement

For expression IF LP EXPS RP STMT and IF LP EXPS RP STMT ELSE STMT, if we don’t give them precedence, then the parser may have a confilct that may be both pattern can be used , so we need to specify the precedence that IF LP EXPS RP STMT ELSE STMT has higher priority than the other one.

```
%nonassoc  IF_NO_ELSE
%nonassoc ELSE_AFTER_IF
....

STMT:
...
| IF LP EXPS RP STMT %prec IF_NO_ELSE { $$ = getNodeInstance(yylineno,"STMT" ,"STMT: if
( EXPS ) STMT", 2, $3,$5); }
| IF LP EXPS RP STMT ELSE STMT %prec ELSE_AFTER_IF { $$ =
getNodeInstance(yylineno,"STMT", "STMT: if ( EXPS ) STMT else STMT", 3, $3,$5,$7);}
```

Error Message

we override the error funciton yyerror that it will give the line number for better debug purpose.

```
void yyerror(char *s)
{
       fflush(stdout);
       fprintf(stderr,"yyerror: %d :%s %s\n",yylineno,s,yytext);
}
```

Node reduction and modification

Reduction

In the given material, there are some grammer like:

```
STSPEC → STRUCT OPTTAG LC DEFS RC | STRUCT ID
OPTTAG → ID | ϵ

EXTDEF → SPEC EXTVARS SEMI | SPEC FUNC STMTBLOCK
SPEC → TYPE | STSPEC

STMT → IF LP EXP RP STMT ESTMT  | ...
ESTMT → ELSE STMT | ϵ
```

I change the grammer by merging the rules. so that I can translate the code in a higher level of the grammer tree. I merge the rules to this:

```
STSPEC → STRUCT ID LC DEFS RC
        | STRUCT LC DEFS RC
        | STRUCT ID
```

```
EXTDEF → TYPE EXTVARS SEMI
       | TYPE FUNC STMTBLOCK
       | STSPEC SEXTVARS SEMI //(by introtucing SEXTVARS to distinguish struct def)


STMT → IF LP EXP RP STMT
       | IF LP EXP RP STMT ELSE STMT
       |...
```

Modification

Since situation like if (EXP) will not be acceptable if EXP is empty, I introduce EXPS as such:

```
EXP -> EXPS
       | ϵ
```

EXPS are defined as exp before but it can not be empty and modify if(EXP) to if(EXPS)

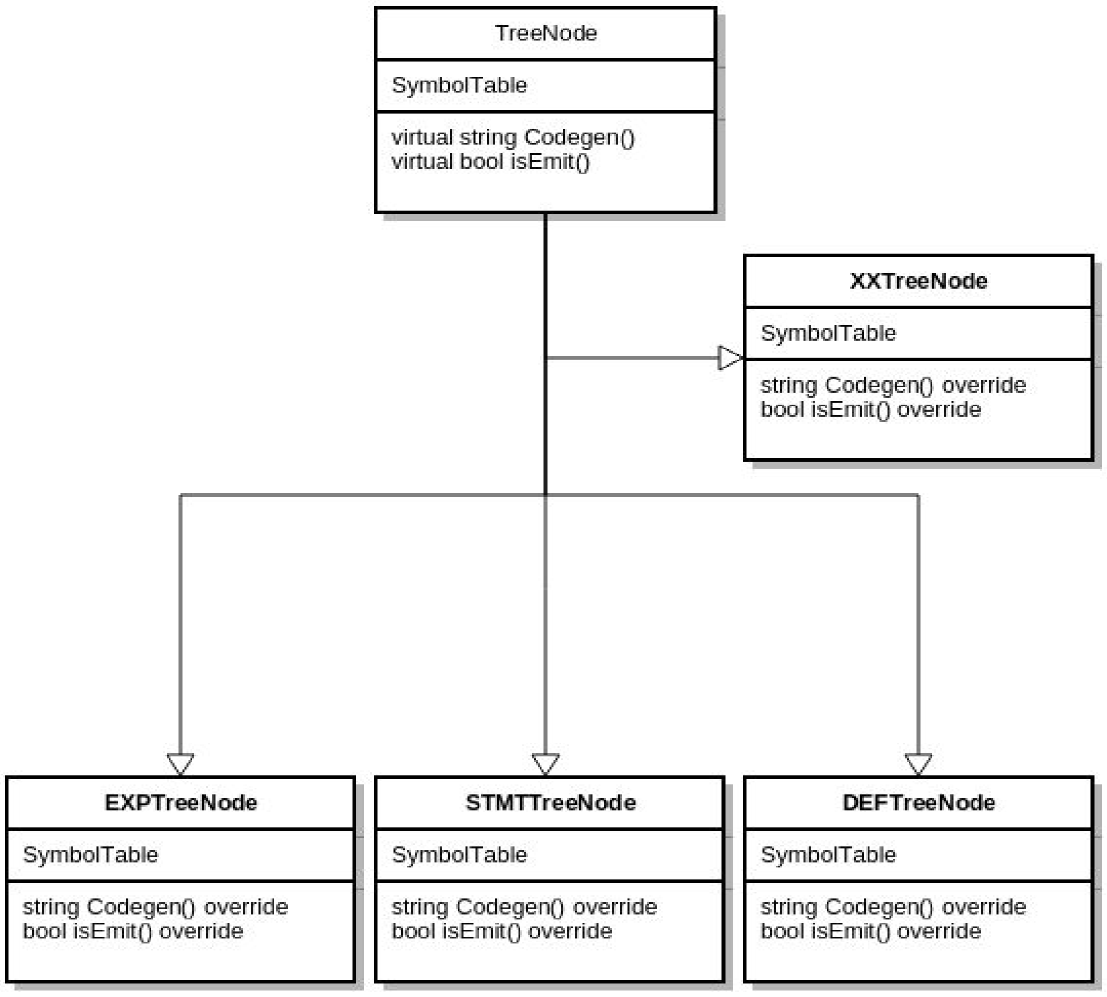
Semantic Analyzer

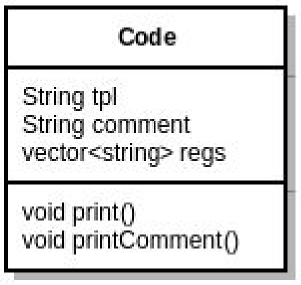
We use polymorphism as our main method in generating parse tree. Our Data structure is as such, there is a base class called TreeNode. And there are a lot of class inheriting the base class and will override the function Codegen() and isEmit();

Derived from the base class, we implement more than 20 derived class such as EXPTreeNode , STMTTreeNode etc.

Function getNodeInstance will judge on the input argument and decide which kind of TreeNode class to return. And that class will have it's own implementation of Codegen(). When parent node is called function Codegen(), it can directly call children class to excute their Codegen() without knowing what kind of child class it is calling.

isEmit is for optimization purpose, if some class override the default function, then the node will be neglected if isEmit() function return false. This is for dead code elimination purpose.





Codegen() will generate code by pushing back Code class to a vector of code. Code is stored by lines or by instructions. It include template which idicate the instrcution selection, comment that for debug purpose and register used in instruction. In our implementation , comment will include the infomation of the path from the calling node to the root node.

Parse Tree Generation

Examples

I will demostrate how for expression is constructed.

```
STMT:  FOR LP EXP SEMI EXP SEMI EXP RP STMT { $$ = getNodeInstance(yylineno,
"STMT","STMT: for ( EXP ; EXP ; EXP ) STMT", 4, $3,$5,$7,$9); }
```

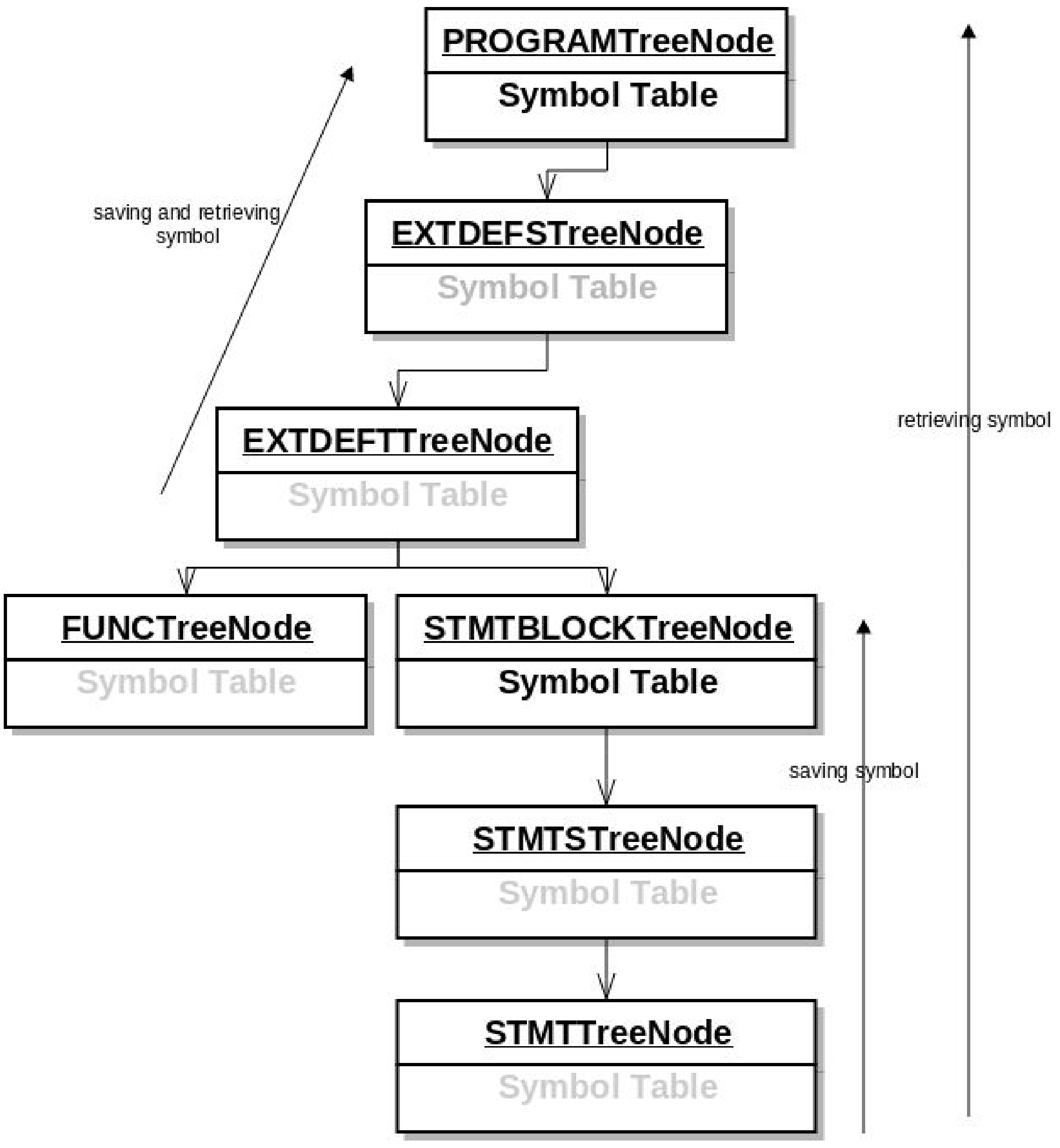
getNodeInstance will return a TreeNode class based on arguments, here we pass “STMT” into the function and this funciton will return a STMTTreeNode Class. And it will has 4 children, pointer of which will be represented as $3, $5, $7, $9. And the returned STMTTreeNode may be treated as a child by other TreeNode.

Symbol Table

Symbol table is stored in each TreeNode, we make this design to support the senario where same name appear in different layer of code like:

```
int a;
int main{
        int a=0;
        int c=0;
        if(a=0){
                int c=1;
        }
        return 0;
}
```

Only STMTBLOCKTreeNode and PROGRAMTreeNode has meaningful symbol table, for other type of nodes, the symbol table should be empty.



As the figure suggests, when a node want to retrieving a symbol, it will first look up to the nearest STMTBOLCK node or PROGRAM node, and check if the symbol is in the table, if not find it, it will futher go to upper node until arrive root node.

For saving process, toward the root direction, it will save this symbol to the nearest symbol table.

The symbol table is resposible to make a map between the varible in smallc to a unique name in LLVM. And Symbol table in Program holds global variable, while Symbol table in STMTBOLCK holds local variable. Other node's symbol table will be empty.

Error Checking

Auto conversion

For situation of :

```
if (exps)
for(exp;exp;exp)
```

The value inside if bracket and second value of exp should be a bool value, but generally in most grammer, int, string and other varible that are not empty or not zero will be concidered as true. Here after the calculation of the exps, we will do the auto conversion, if and i32 is returned , it will be compared to zero to generate a bool value.

operand type checking

1. for Dot and [] , we will first check if the operand is valid or not. If it is not valid, we will log warning and exit the program.
2. for return statment, I will check if the return register is a i32 format.

Resevered word

Id can not be reserved word, this is implemented by the flex and yacc, it will treat reserve word a symbol and parse it by that symbol's way. It will generate an error in syntaxtree building process.

Main entrance

We will check is the function main exist after we finish parsing the code.

Usage after declaration

This part is implemented with symbol table, when encountered a symbol, it will try to retrive the symbol in the path of node toward the root. If no such symbol is found, it will generate a error message and exit with -1.

Break & continue

We use a stack to maintain if program is inside a for loop, in each for loop, we will push the break label and continue label into the stack, when encounter a break or continue, it will first check if the stack is empty and then jump to the label on the top of the stack.

Function checking and Struct definition checking

When program call a funciton or declare a struct, we will check if it has been defined globally.

Intermediate Representation

Learning LLVM basic knowledge

small c	translation template
read	%s = call i32 (i8*, ...)* @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8]* @.str, i32 0, i32 0), i32* %s)
write (other funciton call are similar)	%s = call i32 (i8*, ...)* @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8]* @.str, i32 0, i32 0), i32* %s)
continue, break	br label %%%s
return	ret i32 %s
int a	%s = allocal i32, align 4 %s = common global i32 0, align 4
int a[4]	%s = common global [4 x i32] zeroinitializer, align 4 %s = alloca [4 x i32] , align 4
struct a={ …}	%s = type {i32, i32…}
a=b	%s= load i32* %b, align 4 save i32 %s , i32* %a, align 4
a+b (other binary operation is similar)	%s = add i32 %1, %2
a==b (other compare operation is similar)	%s = icmp eq i32 %a, %b
++a (other unary operand is similar)	%a= load i32* %ptr_a, align 4 %s =add i32 %a, 1 save i32 %s, i32* %ptr_a

Register Allocation

Alougth llvm can have infinite variables and llvm will do register allocation inside it, we still do register allocation for practice.

Note that llvm does not allow the following grammer:

```
%r3 = add i32 %r1, %r2
....
%r3 = add i32 %r1, %r4
```

This means that one variable can only be assigned a value once. Because of that I named my register with two parts like: %r_45.fp323. The first part is %r_45, meaning that it is a register, and it is register 45, fp323 is short for figerprints 323. Finger prints is to make sure that the register is not the same for llvm. But if we are going to help llvm implement register allocation, then the fingerprint part can be neglected.

There are three cases where we need to handle register.

1. read/write return value.
   a. when read and write are called, a function call will return a int. And that int will be abandoned.
2. EXP calculation
3. initialization of the array,
   a. we need to first get the pointer to the value
   b. then store value to that pointer, after that this pointer is of no use.

For first and third situation, since they will be abandoned after one time use. They will handled by adhoc method. We do register allocation for the second situation.

Our register allocation Algorithm is like this:

1. we maintain an int registerNumber and set<stirng> freeRegister.
2. if freeReg is called, then the reg is add to freeRegister.
3. if allocateReg is called, then it will give the first element of freeRegister(also the smallest one since set is implemented as BST) if freeRegister is empty then increment registerNumber and return a newly created name.

freeReg is called by the programmer who write the compiler (i.e me). basically I will freeReg all regs except the return one when finish the calculation of the EXPS.

Optimization

Unused function

For unused funciton , I will directly remove that EXTDEFSTreeNode.

Unused struct definition

For unused struct definition, I will directly remove that EXTDEFSTreeNode.

Conclusion

My compiler has passed the given test and other test like quicksort, prime number, recursive if , and int array test.

My program is published in github: https://github.com/jiady/compiler-llvm. Feel free to download, comment or modify it.

You can see the commit history here:
https://github.com/jiady/compiler-llvm/commits?author=jiady

online doc:
https://docs.google.com/document/d/1tONBUmLnBsV1VjflQ5XMpU3WLNFiC7uWsflQ-jzs26c/pub

Philosophy

Programing for debug

At the very begining I realize that this is such a big project that without a great debug method it will be very hard for me to make it. I have majorly two way on this.

1. I make plenty of log and comment in generated llvm code to make sure every move is correct. Every generated code is along with a comment that indicate structure infomation of the parse tree. And I can easily see the parents of that node and so on.
2. I check the assumptions before doing any logic, for example I will check if this node's number is correct, if the content is valid. I check like content==”STMTBLOCK: XXXXX” instead of content.at(0)==’S’. What I want to do is to make every move clear. I will check if I can't find a valid symbol or some symbol is duplicate. I will exit immediately instead of letting the program running on errors.

Avoid duplication

It is common that some logic may need to be changed against the design during coding. And it will be painful if you write the same logic code for hundrends of times. When you need to change it , you will find it is impossible.

1. I use a lot of #define and helper funtion to avoid duplication.
2. Why polymorphism? why not simple if else to decide calling which funtion to generate code. The quesiton should be asked as why the same infomation need to be given twice? The tree structure is given in the syntax tree already, why should you use if else to judge it again? So polymorphism is used to avoid duplication and make it more elegant.

Design before you start coding

If you do not have a big picture(design) and start coding, believe it or not, you are writing shit.

Reference

http://www.cs.sjtu.edu.cn/~jiangli/teaching/CS308/projects/LexIntroduction.pdf
http://www.cs.sjtu.edu.cn/~jiangli/teaching/CS308/projects/SetupEnvironment.pdf
http://www.cs.sjtu.edu.cn/~jiangli/teaching/CS308/projects/YaccIntroduction.pdf
http://www.cs.sjtu.edu.cn/~jiangli/teaching/CS308/projects/project2.pdf
http://www.cs.sjtu.edu.cn/~jiangli/teaching/CS308/projects/project1.pdf
http://www.cs.sjtu.edu.cn/~jiangli/teaching/CS308/projects/LLVMIntroduction.pdf
http://www.cs.sjtu.edu.cn/~jiangli/teaching/CS308/projects/ProjectIntroduction.pdf
https://github.com/BinaryMelody/Compiler_Principle
https://github.com/vendisky/Compiler-Principles-Project-2
https://github.com/linzebing/compiler
http://llvm.org/docs/LangRef.html