

**本科实验报告**

|  |  |
| --- | --- |
| 课程名称： | 编译原理 |
| 题 目： | 类C语言编译器 |
| 组 员： | 郑无恙 麦蓝 |
| 学 院： | 计算机科学与技术 |
| 专 业： | 计算机科学与技术 |
| 指导教师： | 王强 |

2021年 6月 12日

1. 序言

编译器是一个能够将某种编程语言的源代码转换为另一种编程语言（一般称为目标语言）的软件。本次实验的目的是实现一个编译器，能够将类C语言的代码转换成中间代码（汇编代码）。我们小组希望通过本次实践，加深对编译系统中每一环节工作的理解。

以下为小组分工情况。

词法分析、语法分析、语义分析：麦蓝

优化考虑、代码生成、测试案例：郑无恙

二、实验环境要求

1. Windows或Linux系统环境。Linux系统环境下，需要安装make；

2. Lex以及Bison（Yacc）；

3. clang以及llvm。

三、如何编译并运行此项目

Windows系统：运行start.bat编译本项目。编译完成后，可执行文件名为proj.exe。在使用proj.exe编译源代码前，检查当前目录是否存在testdata.c的文件。如果文件已经存在，则proj.exe将通过读取testdata.c文件对文件进行编译。因此，如果想要测试本项目编译器，需要修改testdata.c文件。完成修改后，即可对本项目编译器进行测试。

Linux系统：在项目目录中执行make命令编译本项目。完成编译后，可执行文件名为proj。后续流程与Windows系统一致，不再赘述。

当proj.exe/proj运行结束后，项目目录应该将出现一个名为output.ll文件，此文件便是针对testdata.c进行编译后所生成的llvm中间代码文件。

如果想进一步对此中间代码文件做进一步的优化，可以输入以下指令：

opt -S -O3 output.ll

便可完成对中间代码的优化。

现在需要将中间代码文件编译链接生成可执行文件，我们可以通过以下指令完成：

clang output.ll -o output

接下来，只需通过运行output.exe/output即可完成程序运行。

四、词法分析

4.1 变量定义

这部分主要使用lex实现。编写完lex后，使用命令flex -l编译生成lex.yy.c。

源代码部分：



图 1：C部分的全局声明

变量line是用作记录token的行数，当报错时会打印出出错的行号，使用户能更方便地找地错处。

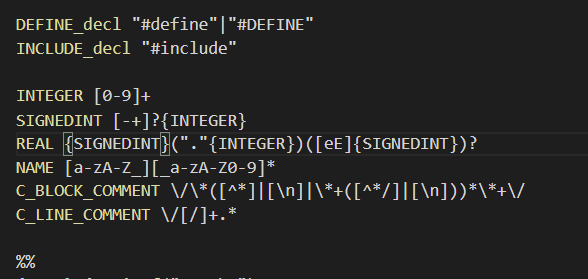


图 2：lex部分的全局声明

这部分为了方便及简化下方的正则表达式而事先对一些正则表达式进行声明

其中：

INTEGER 为整数，由于经测试，在C语言中，即使给int变量赋值的整数不用是无前置零的，即int a=0001;这个语法在C语言中是没有错的，a会被赋值为1，所以此处表明INTEGER为无符号整数的正则表达式。

SIGNEDINT 为有符号整数，由于正数可忽略前面的正数符号”+”，且’-‘符号放在中括号’[ ]’的中间会被当作别的意思，故’-‘应放在最前，所以[-+]?{INTEGER}为SIGNEDINT的正则表达式

REAL为浮点数，而浮点数必须要有小数点在其中，故第2项：(“.”{INTEGER})必须存在。而C语言也支持使用科学记数法来表示，所以第3项([eE]{SIGNEDINT})? 这个正则表达式来处理科学记数法。

NAME: 为变量名的正则表达式，C语言的变量名的规则是变量名由大小写英文字母，以及数字和下划线组成，并且第1个字符不能为数字。故C中变量的正则表达式为[a-zA-Z\_][\_a-zA-Z0-9]\*

C BLOCK COMMENT的正则表达为\/\\*([^\*]|[\n]|\\*+([^\*/]|[\n]))\*\\*+\/

大致的意思为一开始的两个字符为 / 和 \*，在之后注译的内容中不能包含 \* / ，故表示为

([^\*]|[\n]|\\*+([^\*/]|[\n]))\* ，在末尾补上面\\*+ 和 /

C的line comment的正则表达式为\/[/]+.\* ，首先检测到在两个及以上的’/’，之后由于正则表达式 . 不包括换行，故在//以后的所有元素都视为comment

4.2 正则表达式操作

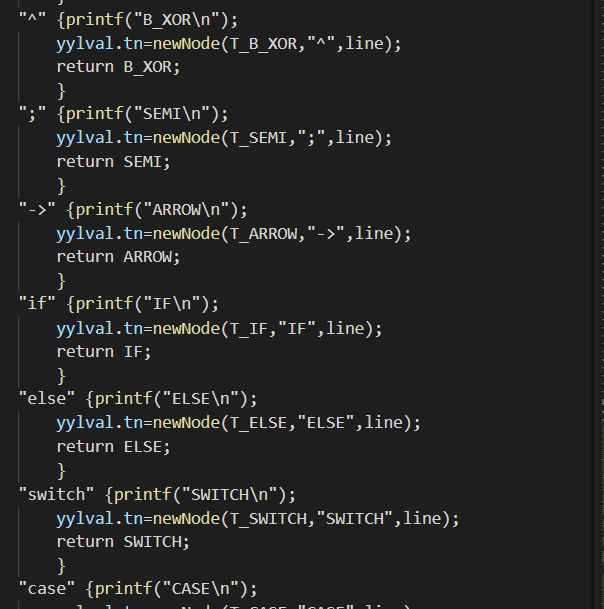


图 3：正则表达式一览

对所有会在C出现的格式符号如if, { 等等，都会按照以下的方式来进行词法分析。

对一般的terminal来说，调用newNode函数的第1个参数为一enum的值，用作type checking比较方便，第2个参数为terminal的本身的符号/本身的值，第3个参数为这个terminal所处的行数。返回值是代表该token的一个non-terminal，这里的想法是在yyparse中对词法分析中读出的token和bison本身的格式符号作一个更好的区分。

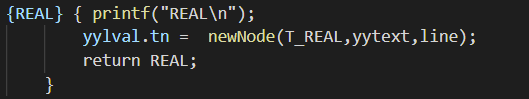


图 4：浮点数

当读到浮点数时，返回Non-terminal REAL以及通过newNode函数构造struct ASTnode，用作在下一步的parser时构造parse tree，而构造node所需的参数分别为代表读到符点数的enum: T\_REAL，value为符点数的值本身，以及行数。

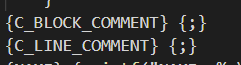


图 5：注释

当遇到comment时，甚么都不做

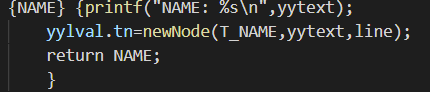


图 6：变量名

当读到变量名时，返回Non-terminal NAME以及通过newNode函数构造struct ASTnode，用作在下一步的parser时构造parse tree，而构造node所需的参数分别为代表读到符点数的enum: T\_NAME，value为变量名，以及行数。

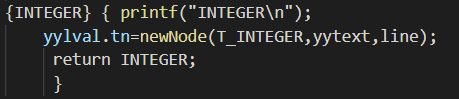


图 7：整数

当读到整数时，返回Non-terminal INTEGER以及通过newNode函数构造struct ASTnode，用作在下一步的parser时构造parse tree，而构造node所需的参数分别为代表读到符点数的enum: T\_INTEGER，value为变量名，以及行数。

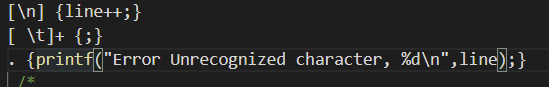


图 8：换行及tab

读到换行时，更新line变量

读到空格或tab时跳过

当没有字符能匹配时，打印Error信息并吞掉该字符

4.3 实现原理

Lex主要是实现compiler 所有步骤中的第一步: Scanner。将source code 变成一个个的tokens给下一步的parser使用。

当yyparse调用yylex函数时，yylex便会返回下一个从lex.l档中从正则表达式匹配到的token，

并把其值通过函数newNode放在struct ASTnode中返回给parser。

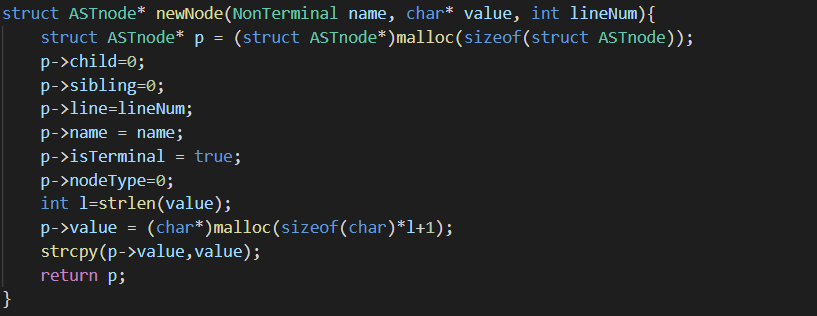


图 9：函数newNode

函数newNode是定义在ASTtree.c中的一个函数，主要的作用是把一个token中的内容包装成一个AST树的节点，使其在下一步parser中能被直接挂在树上。

其实现方法是先用malloc申请一块区域，然后对每一个ASTnode中的成员赋上从参数传来的值，最后返回这个struct

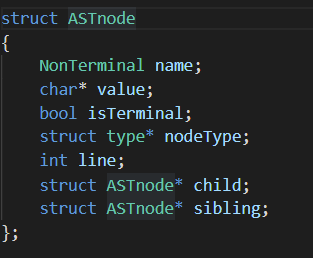


图 10：ASTnode的定义

这是ASTnode的定义，内容为AST tree中的一些必要信息。其中，name 的类型为enum。

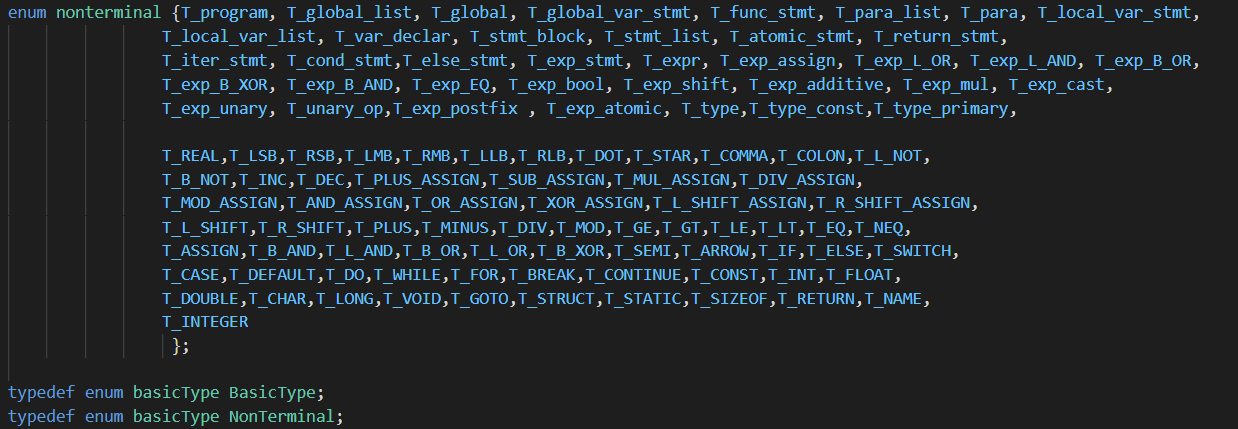


图 11：程序中所有non-terminal一览

对所有的non-terminal用一个enum来表示。

Char\* value: 为这个node的值，如为terminal，则为其本身的值，如为non-terminal，则为空(在parsing这一步)。

Bool isTerminal: 表示这一个node是否为terminal

Struct type\* nodeType: 表示这一个node的类型

其中Struct type的类型如下：

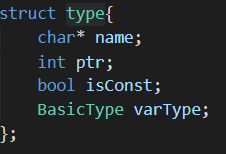


图 12：Struct type

Int ptr 表示其指针的级数。

Bool isConst表示其是否为const

BasicType varType表示其基础类型，为enum类型





图 13：BasicType

BasicType的定义如上。

而char\* name是方便在semantic analysis时使用的成员。

Int line: 表示这个node最后一个字符所在的行数

struct ASTnode\* child;

struct ASTnode\* sibling;

这个节点的儿子和兄弟的指针。

五、语法分析

5.1 变量定义

这一部分主要在yacc.y中实现，也有一些函数写在ASTtree.c中

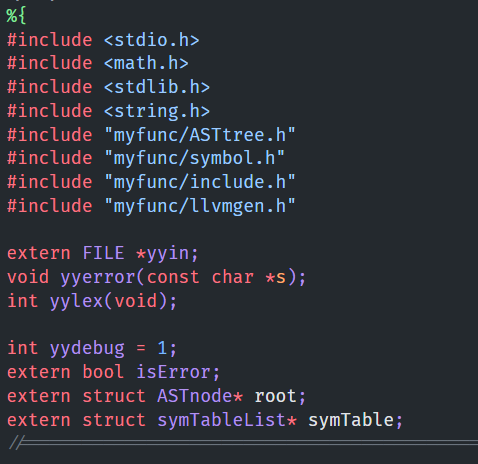


图 14：yacc中C语言部分的定义

这里我们将生成syntax tree 的部分和semantic analysis部分的函数分别放在ASTtree.c和symbol.c这两个文件中，以及所有的结构和全局变量的定义都放在include.h这个文件中，这个做法有利于项目代码的模块化。

这里其中root是syntax tree的根节点，symTable是symbol table的根节点。isError代表在语意分析这一步有没有发现编译上的错误。

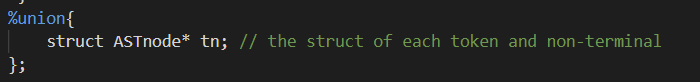


图 15：声明tn的类型

这一句主要用作声明tn的类型，而tn是每一个terminal和non-terminal的类型

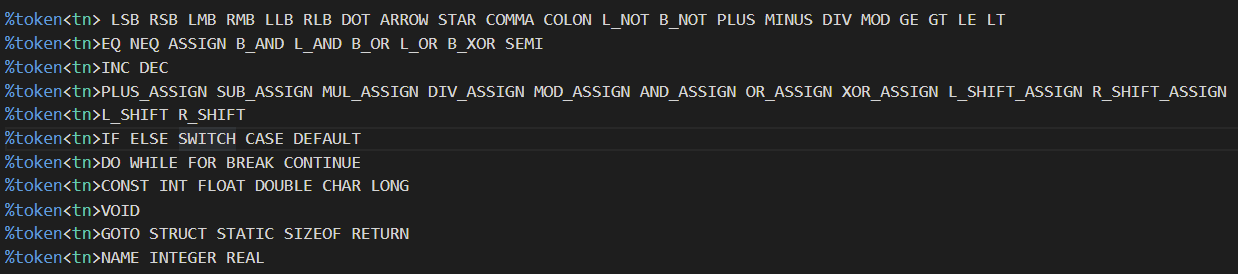


图 16：声明所有的terminal以及其类型

这是声明所有的terminal以及其类型。

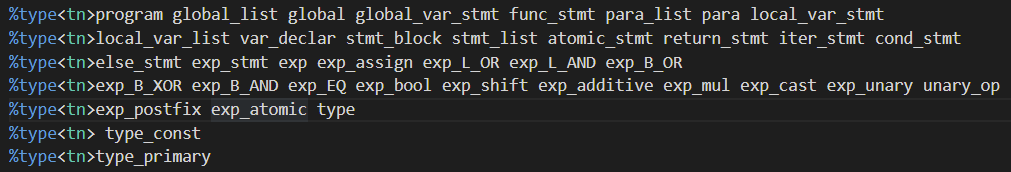


图 17：声明所有的non-terminal以及其类型

这是声明所有的non-terminal以及其类型。

由于下方的上下文无关文法已考虑到优先级的问题，所以这里没有先后顺序之分



图 18：THEN及ELSE

这两个terminal要作特别处理，因为需要消除if-else之间的shift-reduce conflict，其用法在后面会提到。

5.2 上下文无关文法

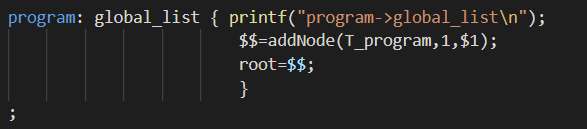


图 19：program文法

Program：程序的结束parse的点

其可以生成global\_list这个non-terminal，以及由于这是parser的最后一步，所以这一步所生成的节点为root节点，故令root等于其返回值

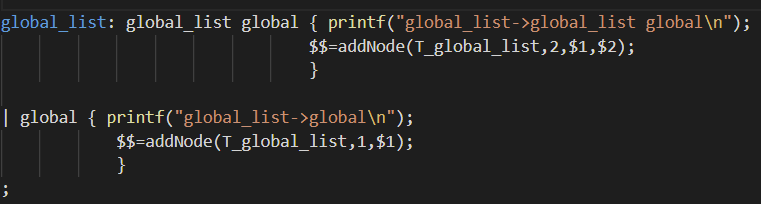


图 20：global\_list文法

global\_list生成global\_list和global，或只生成一个global，

由于C语言的代码是只由全局的声明及全局的函数所组成，故global\_list这里是指整个C语言的program。而整个程序是由不限数量的全局语句所组成，所以global\_list🡪global\_list global

这里规定一条程序至少要有一句全局语句所组成，否则会报syntax error的错误

在grammar: global\_list🡪global\_list global中，执行addNode的操作，把global\_list global的node分别挂在左边的global\_list的child 和next 下

在grammar: global\_list🡪global中，执行addNode的操作，把 global的node挂在左边的global\_list的child下

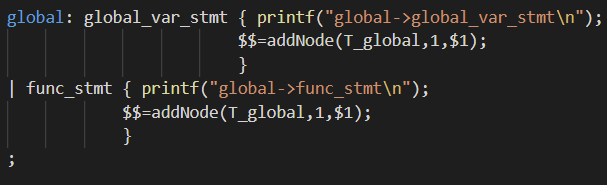


图 21：global文法

由于我们所设计的程式语言不考虑#define以及#include语句，和typedef等等，故每一个全局的语句只会是全局的变量声和函数体，所以文法如图所示

global🡪global\_var\_stmt

这句grammar表示这个全局语句为全局变量的声明

在grammar: global🡪global\_var\_stmt中，执行addNode的操作，把 global\_var\_stmt的node挂在左边的global的child下

global🡪func\_stmt

这句grammar表示这个全局语句为函数或函数体的声明

在grammar: global🡪func\_stmt中，执行addNode的操作，把 func\_stmt的node挂在左边的global的child下

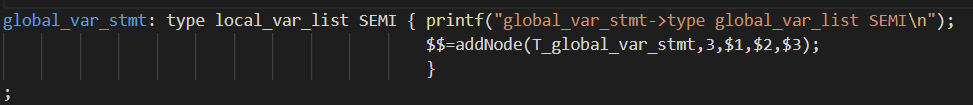


图 22：global\_var\_stmt文法

进一步看看全局函数的语法分析：

在C语言中，全局函数的变量声明是先写变量的类型，再写变量名，最后以分号作结束

在global\_var\_stmt🡪 type local\_var\_list SEMI中

执行addNode的操作，把 type的node挂在global\_var\_stmt的child下，local\_var\_stmt和semi以链表的形式挂在global\_var\_stmt的next 下

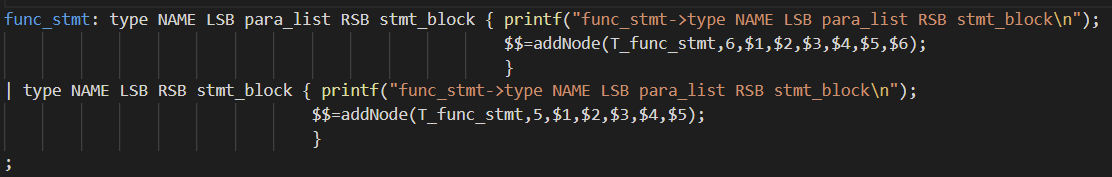


图 23：func\_stmt文法

对于函数来说，由于C语言没有局部的函数，所以只需考虑全局函数的情况，全局的函数语句可分为两种，一是函数的声明，二是函数体的定义

func\_stmt: type NAME LSB para\_list RSB stmt\_block

这句grammar表示这个函数是有至少1个参数的

type NAME LSB RSB stmt\_block

而这句则表示没有参数

同样，这两句的grammar都把type挂在func\_stmt的child中，其余的挂在func\_stmt的next中。

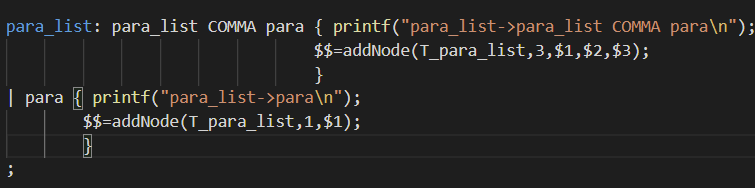


图 24：para\_list文法

这是函数的参数的语法分析，由于在func\_stmt中保证了至少有一个参数，故在这里不用考虑没有参数的情况

para\_list: para\_list COMMA para

在C语言中，参数之间用comma分隔，且没有分号作结束

同样，执行addNode操作时，这grammar把para\_list挂在func\_stmt的child中，其余的挂在func\_stmt的next中。

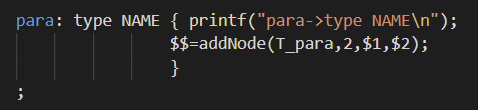


图 25：para文法

在C语言中，参数的形式有两种，一是先写参数类型，再写参数名，二是…，但由于在lex中未有对…这个符号作词法分析，故这里不考虑不限参数的做法。

执行addNode操作时，这grammar把type挂在para的child中，其余的挂在para的next中。

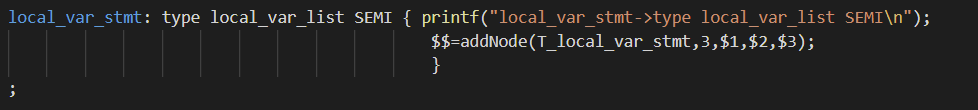


图 26：local\_var\_stmt文法

上面说过全局变量的声明，这里是局部变量的声明，与全局变量的语法分析无异，但由于两者的出现在程序的地方有很大的差别，故把它们分隔开。

执行addNode操作时，这grammar把type挂在local\_var\_stmt的child中，其余的挂在local\_var\_stmt的next中。

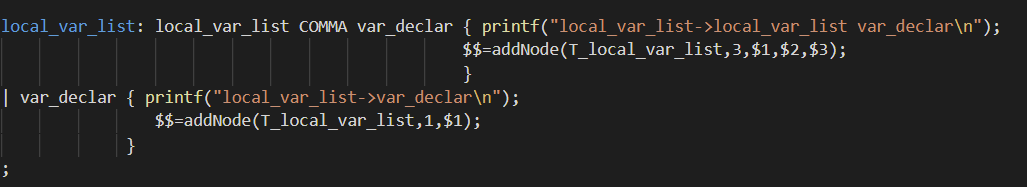


图 27：local\_var\_list文法

这是在全局变量声明或是局部变量声明中的语法分析，由于在C语言中可以在同一语句中同时声明多个变量，故这里需要作一个left recursive，每一个变量的声明使用comma分隔

执行addNode操作时，这grammar把local\_var\_list 挂在左边的local\_var\_list的child中，其余的挂在local\_var\_list的next中。

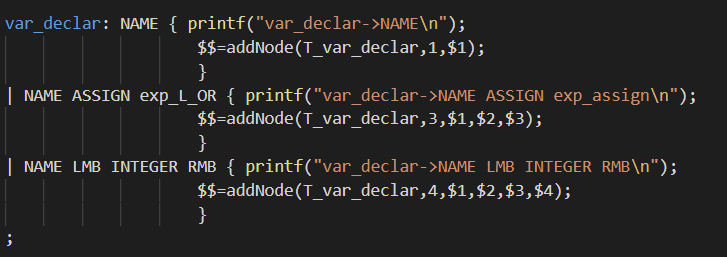


图 28：var\_declar文法

对于每一个变量的声明，可以选择赋予初始值，而这个初始值可以是一个表达式的形式，或这是一个一维数组的声明。

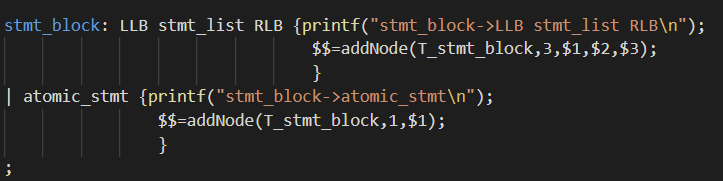


图 29：stmt\_block文法

这是一个由大括号括住的区域，一般在函数体后，及一些语句如if, while语句后出现

当只有单句语句时，也可以忽略大括号。

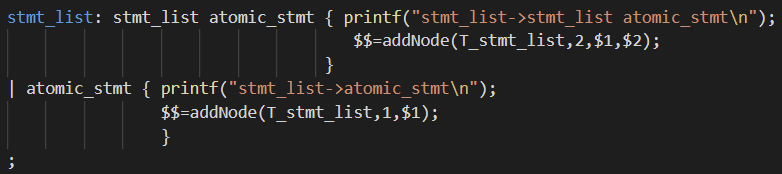


图 30：stmt\_list文法

Stmt\_list便是由很多语句组成的语句的集合，通过left recursion的方式表示stmt\_list可以由无限多的语句所组成

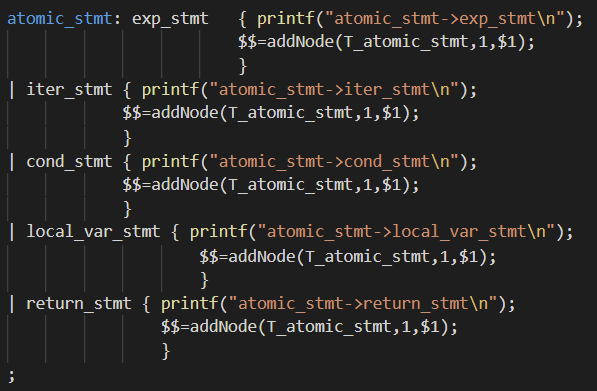


图 31：atomic\_stmt文法

在C语言中，每一句语句都可以分为下列5种语句其中之一，其中包括:

Exp\_stmt，为一般的运算表达式，可为赋值语句，可为函数调用语句，等等。

Iter\_stmt为循环语句，由while，for 以及do- while所组成

Cond\_stmt，为if-else语句，由于switch 语句可由if-else语句所替，故不实现switch语句

Local\_var\_stmt，为局部变量声明的语句，可自由选择是否初始化

Return\_stmt，为返回语句，在函数返回时使用。

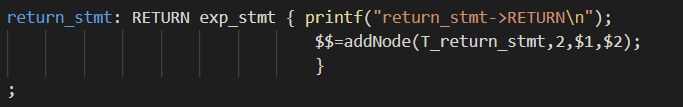


图 32：return\_stmt文法

Return\_stmt 返回语句由return这个key word以及返回值的表达式所组成。只有这一种格式

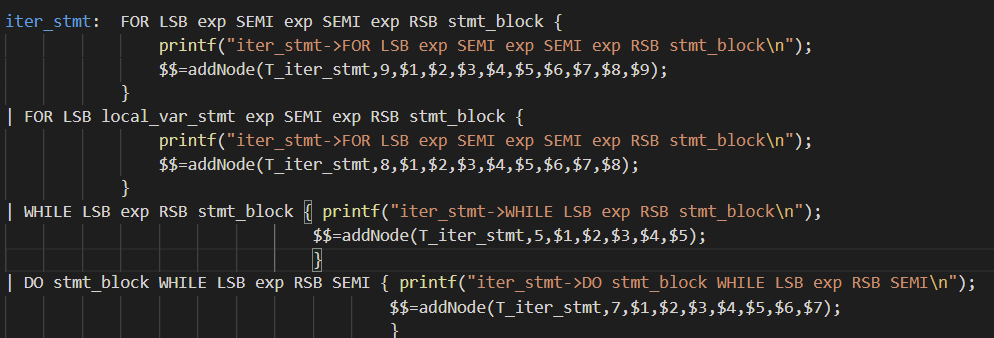


图 33：iter\_stmt文法

分别为：

在第1个分号的表达式不会定义新变量的for\_stmt

在第1个分号的表达式中定义新变量的For\_stmt

While循环

以及do-while循环

这四句grammar分别定义了循环语句的格式

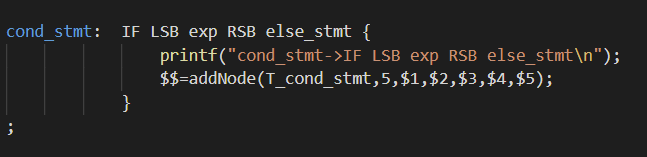


图 34：cond\_stmt文法

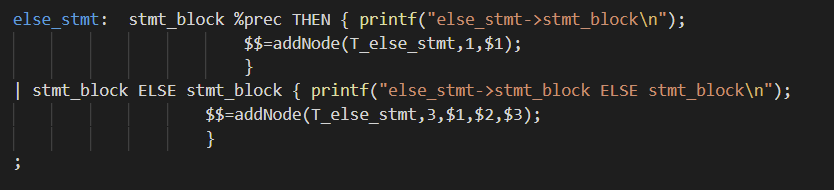


图 35：else\_stmt文法

If-else statement:

在cond\_stmt中定义了if语句的if部分的格式，但不包括If分支的代码部分

在else\_stmt中定义了两个grammar

上式

else\_stmt:  stmt\_block %prec THEN

中为单个if没有else的grammar

其stmt\_block为if分支的代码部分

下式

stmt\_block ELSE stmt\_block

中为有if有 else的grammar

其中第1个stmt\_block为if分支的代码部分，最2个stmt\_block为else的代码部分

而%prec THEN 的作用为优先匹配else，因为在上方提到

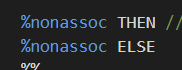


图 36：优先匹配else

ELSE的优先级要高于THEN，所以BISON会优先作shift ELSE的操作，而不是reduce

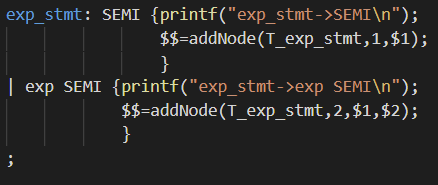


图 37：exp\_stmt文法

由于在C语言中，单独一个分号也视为一个表达式，故这里的第一条grammar用作实现这件事，而第2个grammar则表示一般的表达式

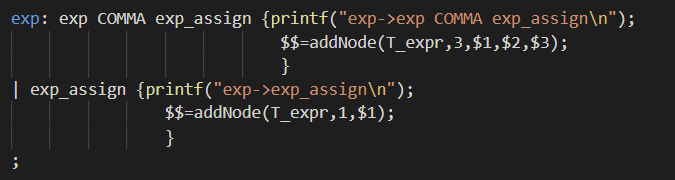


图 38：exp文法

接下来便是C语言表达式的优先级的grammar，由于在定义token时不用考虑token的先后顺序，故要用grammar来实现C语言的优先级。

在C语言中，COMMA为最低的优先级，故放在syntax tree较上方的位置

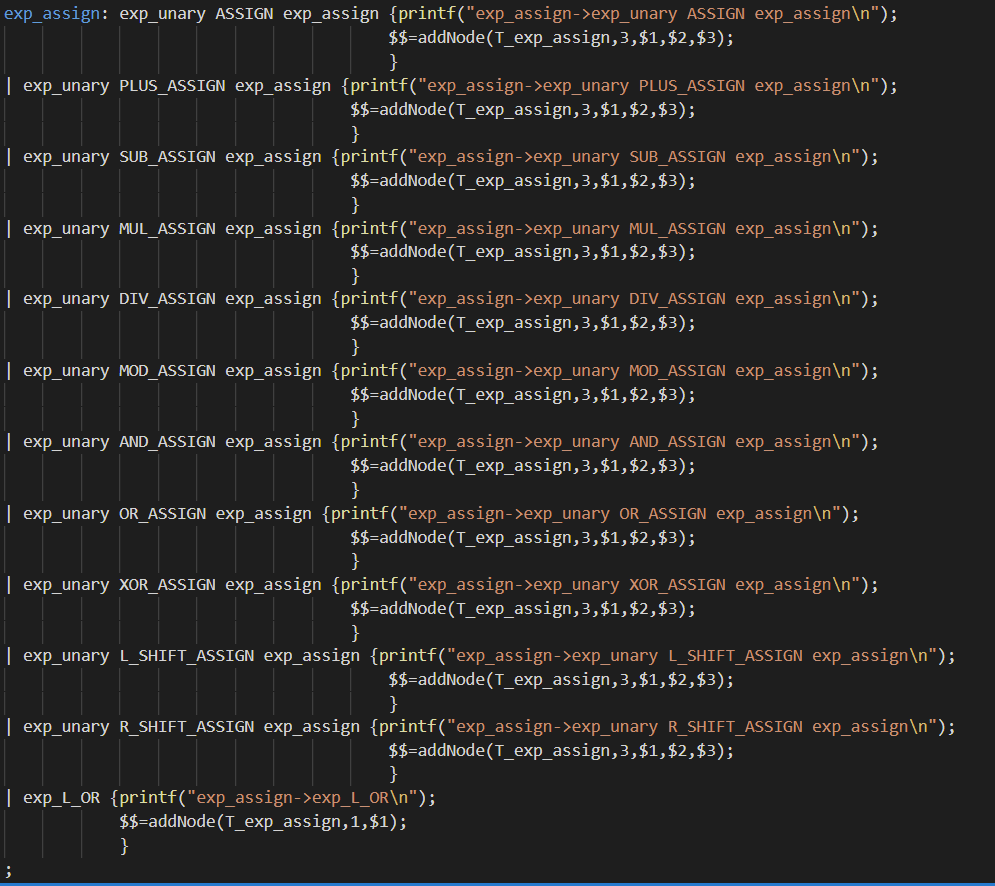


图 39：exp\_assign文法

比comma高一级优先级的便是赋值操作，这里列举了在C语言中各个赋值符号的赋值操作，而等式的左边是exp\_unary的原因是C语言允许在赋值式中被赋值的变量可以进行++，--，或\*取值等等的操作，故赋值式的左边不是变量名而是exp\_unary

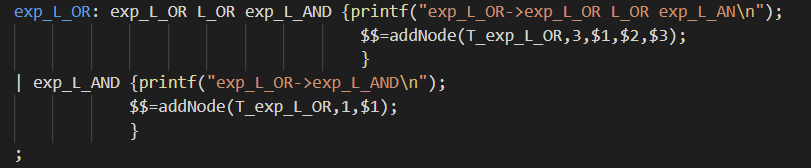


图 40：exp\_L\_OR文法

而下一个优先级便是逻辑OR，即符号 ||，而一个表达式中可以有多个逻辑OR出现，故这里需要用到left recursion

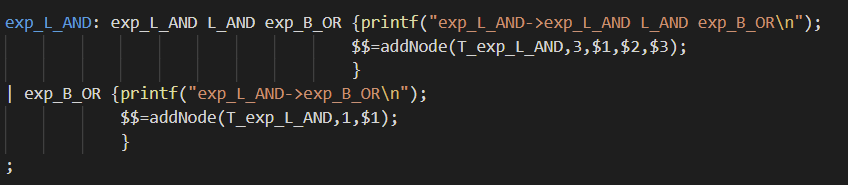


图 41：exp\_L\_AND文法

而下一个优先级便是逻辑AND，即符号 &&，而一个表达式中可以有多个逻辑AND出现，故这里需要用到left recursion

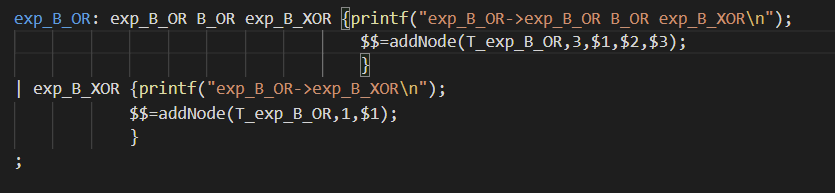


图 42：exp\_B\_OR文法

而下一个优先级便是按位或，即符号 |，而一个表达式中可以有多个按或出现，故这里需要用到left recursion

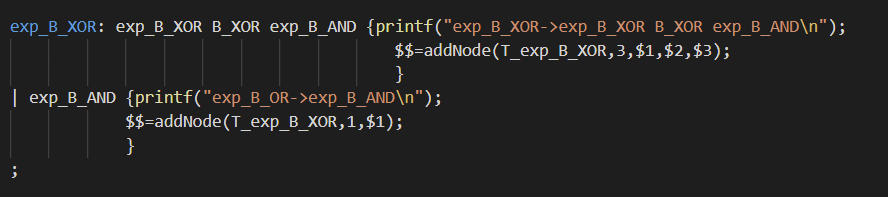


图 43：exp\_B\_XOR文法

而下一个优先级便是按位异或，即符号 ^，而一个表达式中可以有多个按异或出现，故这里需要用到left recursion

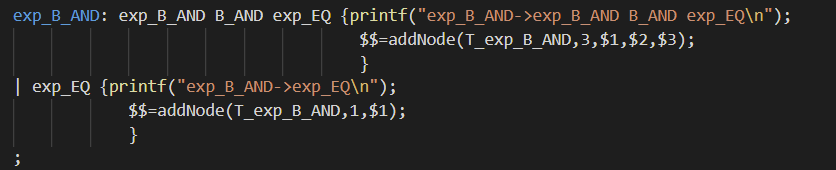


图 44：exp\_B\_AND文法

而下一个优先级便是按位AND，即符号 &，而一个表达式中可以有多个按位AND出现，故这里需要用到left recursion

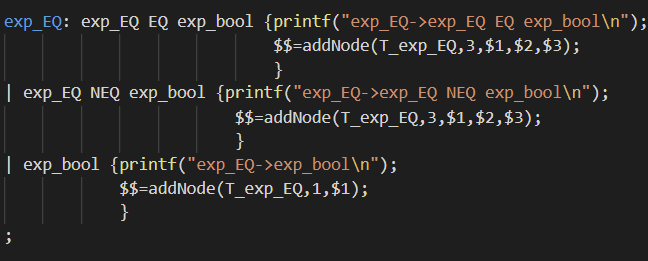


图 45：exp\_EQ文法

而下一个优先级便是比较，即== 或 !=，同样一个表达式中可以有多个比较的表达式出现，故这里需要用到left recursion

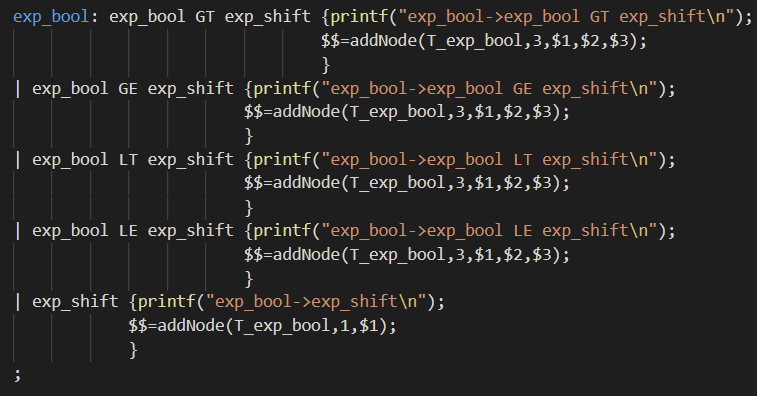


图 46：exp\_bool文法

而下一个优先级同样也是比较，但是范围比较，比等于和不等于的优先级高，即>, >= . < , <=，同样一个表达式中可以有多个比较的表达式出现，故这里需要用到left recursion

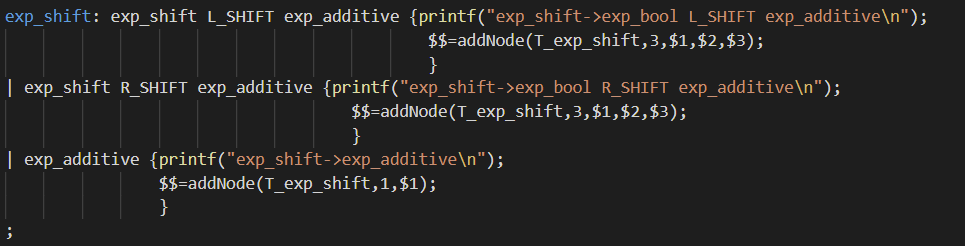


图 47：exp\_shift文法

而下一个优先级便是移位操作，即<< 或 >>，同样一个表达式中可以有多个比较的表达式出现，故这里需要用到left recursion

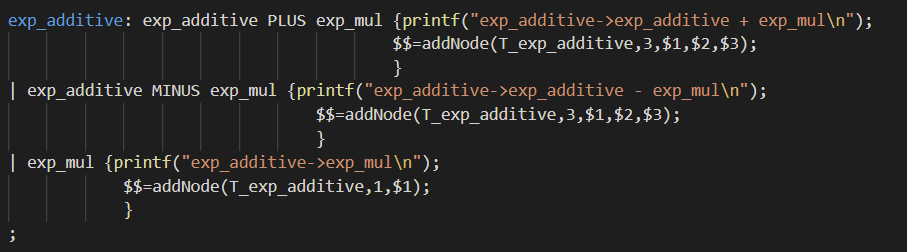


图 48：exp\_additive文法

而下一个优先级便是加减法操作，即+ 或 -，同样一个表达式中可以有多个加减的表达式出现，故这里需要用到left recursion

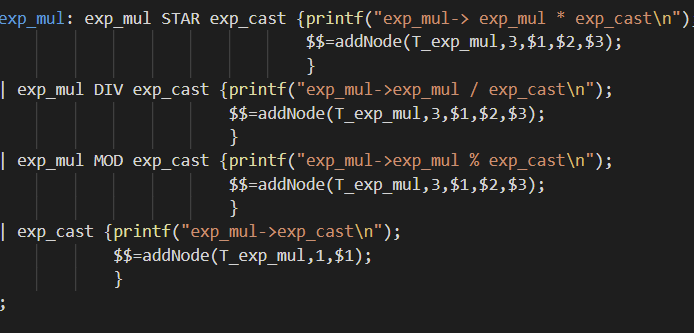


图 49：exp\_mul文法

而下一个优先级便是乘法，除法及取模操作，即\* 或 / 或 %，同样一个表达式中可以有多个乘法，除法及取模操作出现，故这里需要用到left recursion

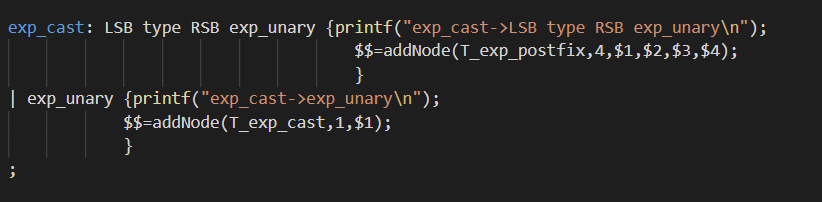


图 50：exp\_cast文法

而下一个优先级便是对变量或常数进行强制类型转换的操作，在这一级中为单目运算符的最低级，但由于赋值式的左边的变量不能被强制类型转换，所以在上面的表达式中赋值式左边的都为exp\_unary而不是exp\_cast

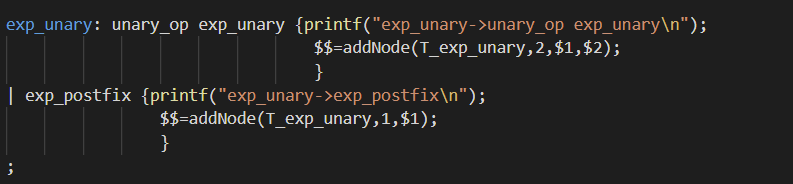


图 51：exp\_unary文法

而下一个优先级便是单目运算的操作，其运算符在表达式之前

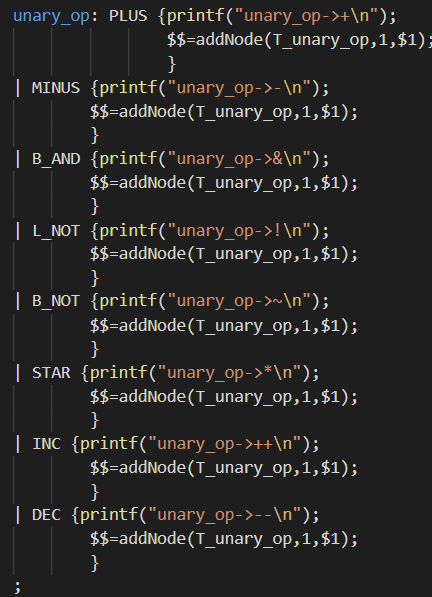


图 52：unary\_op文法

其中单目的运算如上图。

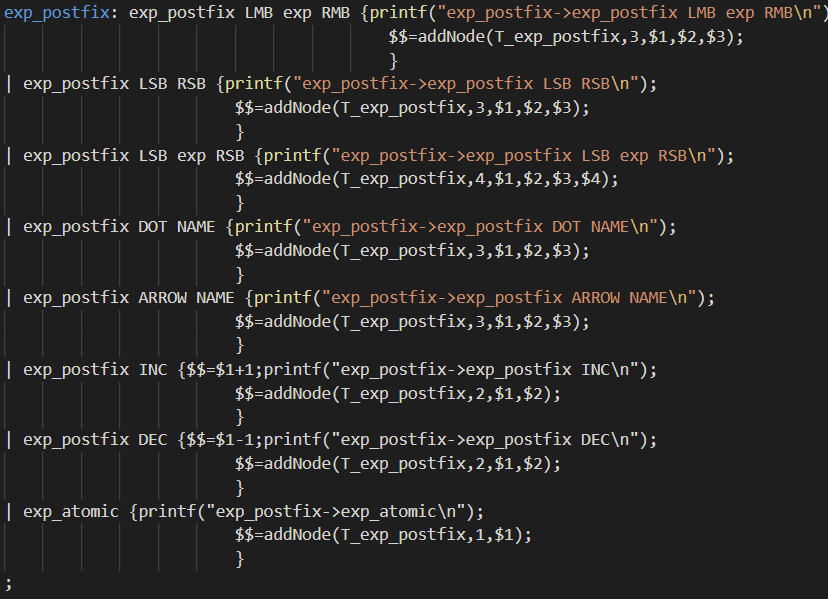


图 53：exp\_postfi文法

而下一个优先级也是单目运算的操作，其运算符在表达式之后

其操作有：

exp\_postfix LMB exp RMB

对数组中某一个元素的值

exp\_postfix LSB RSB

调用没有参数的函数

exp\_postfix LSB exp RSB

调用有参数的函数

exp\_postfix DOT NAME

取结构中某成员变量的值

exp\_postfix ARROW NAME

取该地址中的结构中某成员变量的值

exp\_postfix INC

后缀的+1

exp\_postfix DEC

后缀的-1

exp\_atomic

不能再分割的表达式

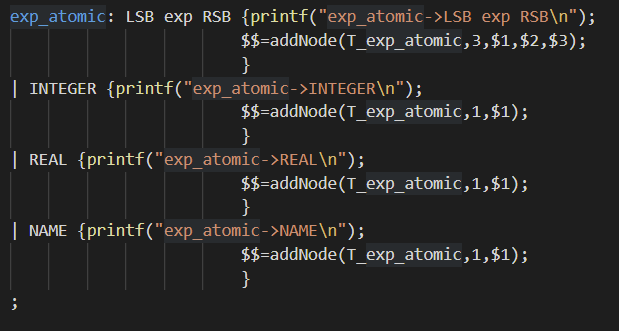


图 54：exp\_atomic文法

不能再分割的表达式，由terminal所组成，其中包括：

（1）被括号包起来的表达式；

（2）整数常数；

（3）浮点数常数；

（4）以及变量名。

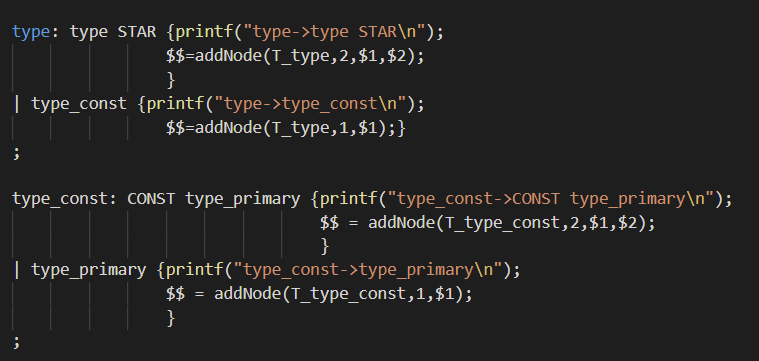


图 55：type文法

Type这个grammar为表示变量或函数返回值的类型

在这里没有考虑用户自定义的struct所构成的自定义类型，故只考虑基础类型，

即int , double char等以及指针和const

由syntax tree下至上来说明，首先知道变量的基础类型，接着判断是否为const，最后再判断这个类的型的指针的级数是多少

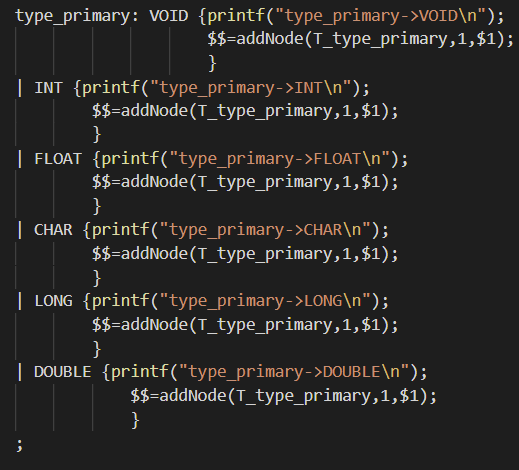


图 56：type\_primary文法

这里显示我们这个编译器所支持的基本的类型。

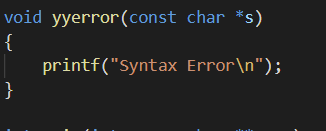


图 57：yyerror函数

当parser发现语法错误时，会报syntax error这一句。

5.3 实现原理和方法

建立syntax tree主要通过yyparse函数以及addNode函数实现



图 58：addNode函数

其主要功能是生成一棵二叉树(syntax tree)的一个重要步骤，通过不断调用这个函数最终可生成一棵syntax tree，按照num所指定的个数，按左儿子右兄弟的方法把…中所传进来的node挂在新生成的node下，并把这个node命名为其所属的non-terminal所对应的enum。

其实现方法如上图，首先申请一个区域作为新的node，接着初始化所有成员变量，最后若这个节点只有1个儿子，则若child指向其儿子，next指向null，若有多于一个儿子，则child指向第1个儿子，next指向第2个儿子，形成一个链表

最后返回这个节点的struct

语法树的打印：



图 59：travelTree函数

这个travelTree的函数的主要的作用是把parser所生成的syntax tree打印出来，通过preorder 的遍历方式，可打印出整棵syntax tree的结构。

其中参数level表示现在处于第几层，用作树每层节点的缩进，然后打印出该节点的non-terminal的名字，通过toString函数，把enum类型转成字符串的格式，以及打印出该non-terminal的最尾的字符处于第几行。

六、语义分析

6.1 符号表主要变量及函数定义

当parser步骤完成后，输出syntax tree，然后便进入semantic analysis的部分，这一步可用symCheck这个函数来概括

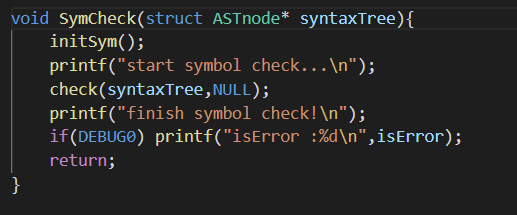


图 60：SymCheck函数

首先第一步是初始化symbol table

具体步骤如下：

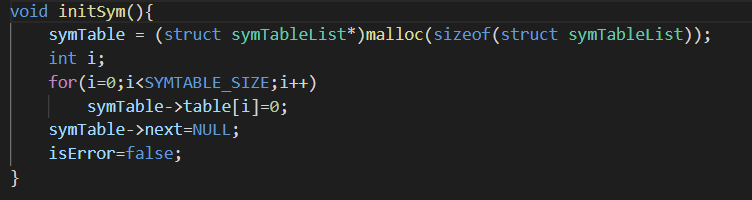


图 61：initSym函数

首先申请一块空间作为symbol table，然后对symbol table中每一个指针进行初始化，接着返回。

以下为symTableList的结构：

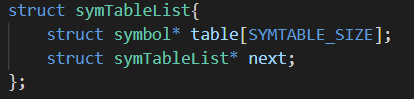


图 62：symTableList的结构

由一个symbol table以及一个链表指针所组成，这样设计的目计是区分开不同scope的变量，以达到实现命名空间的效果。

由上图可见，symbol table通过指针数组的方式实现。

以下是symbol的结构

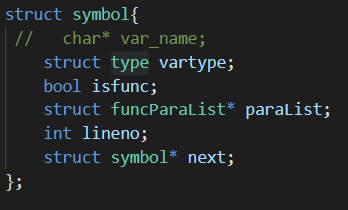


图 63：symbol的结构

Struct symbol为symbol table中存放1个元素的结构，其中包括：

该变量的类型，struct type vartype

该变量是否为函数名，bool isfunc

该变量如为函数，则其参数链表放在struct funcParaList，最右边的参数放在链表头。

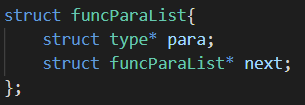


图 64：funcParaList的结构

Struct funcParaList这个struct为一普通的链表结构。其装载的value为参数类型

该变量在那一行被声明的

最后是链表指针

然后通过递归函数check对syntax tree遍历来检查有没有语法错误。

若有则不执行后续步骤，没有若继续执行。

检查函数check的实现

struct type\* check(struct ASTnode\* syntaxTree,struct type\* type\_para);

以上为check函数的定义

其中由于C语言的grammar有Synthesized attributes以及inherited attributed的grammar，所以inherited的部分使用参数作传递，synthesized的部分使用return value作传递

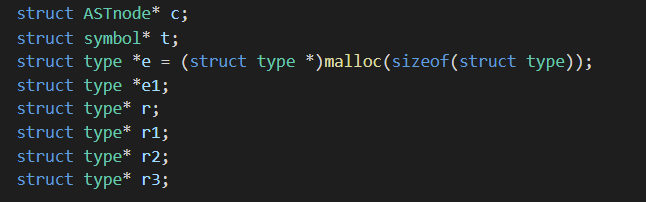


图 65：check函数中有可能会使用到的临时变量

这些是在check函数中有可能会使用到的临时变量

6.2 Check函数

Check函数根据node 的name不同，有不同的做法，所以使用了一连串的if-else来区分不同的情况。接下来对Check函数中较为典型的节点检查举例做介绍。



图 66：当前检查的节点文法属性是否为T\_iter\_stmt

如果当前检查的节点文法属性为T\_iter\_stmt时：

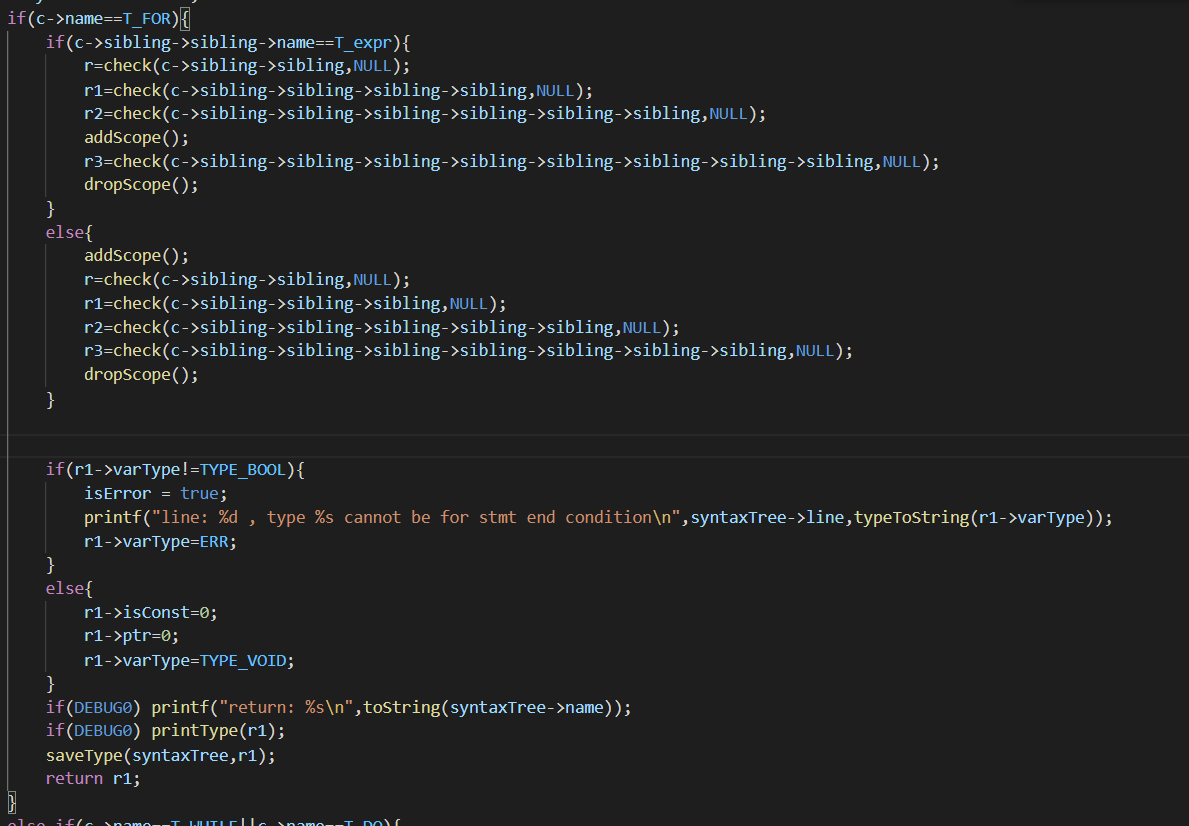


图 67：当前循环是否为T\_FOR

若这个循环为一for循环(通过检查syntaxTree->child->name判断)

若其第1个expression没有声明变量的话，执行If的分支



图 68：iter\_stmt文法

分别检查第3项:exp，第5项:exp，以及第7项的exp，接着在stmt\_block的前后增加一个命名空间，完成stmt\_block删除它，之后检查stmt\_block的正确性。

若其第1个expression没有声明变量的话，执行else的分支，并增加一个命名空间。接着分别检查第3项:local\_var\_stmt，第4项:exp，以及第6项的exp，完成检查stmt\_block的正确性，最后删除刚新增的命名空间。

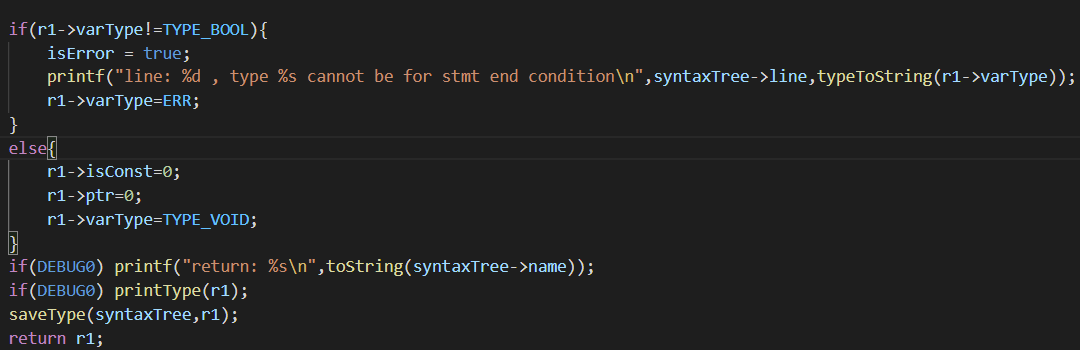


图 69：判断完结条件(r1的type)是否符合要求

退出前判断完结条件(r1的type)是否符合要求，即r1的type是否为bool类型

如果当前检查的节点为T\_var\_declar时：

分成3种情况：

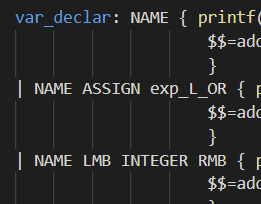


图 70：var\_declar文法

分别对应这3条grammar：

当为第1种情况时，把变量及参数中的type放进symbol table中

当为第2种情况时，把变量及参数中的type放进symbol table中，并判断表达式的返回值是否满足变量所声的类型

当为第3种情况时，把参数中的type的ptr加1，然后放进symbol table中

如果当前检查的节点为T\_REAL时，分为两种情况：一为exp\_atomic的child node为常量或变量，即下图中第2,3,4种情况，此时则直接返回child node 所返回的struct type就可以了

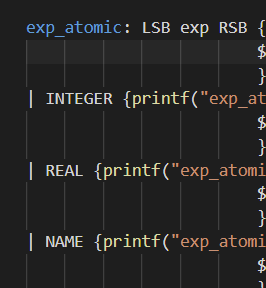


图 71：exp\_atomic文法

二为exp\_atomic的child node为一被括号括住的表达式时，即上图中的第1种情况时，先检算括号中的表达式的正确性，接着返回这个被括号活住的表达式的返回值

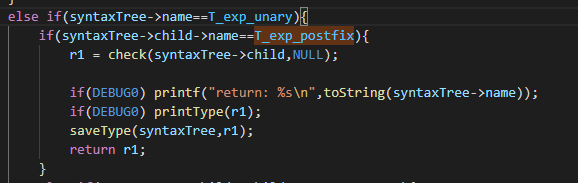


图 72：当前检查的节点文法属性是否为T\_exp\_unary

6.3 符号表其他函数



图 73：evalType函数

evalType的实现，是通过储存type这部分的syntax tree，计算出其type，并把其类型量化表示在struct中，然后返回symbol table的构造

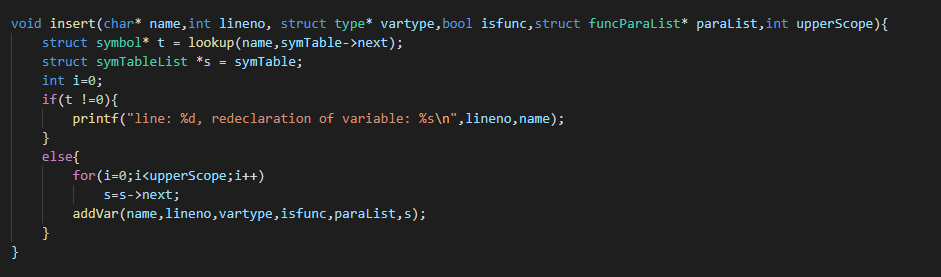


图 74：insert函数

首先在当前的scope中看看有没有相同名字的变量或函数

如没有则将指定的变量的信息通过addVar函数放入当前scope的symbol table中

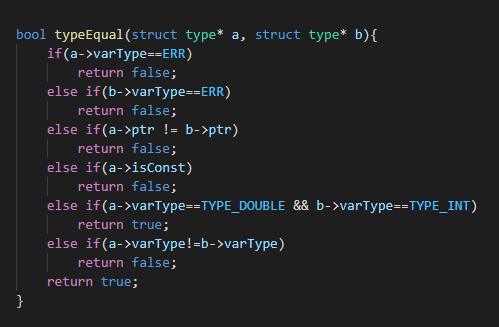


图 75：typeEqual函数

typeEqual函数，主要是判断ptr, const的值是否相同，以及type的类型可否转换

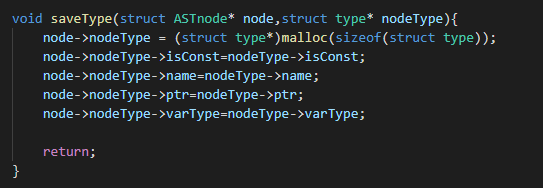


图 76：saveType函数

通过这个函数把check遍历时每个节点的返回值save在ast tree中

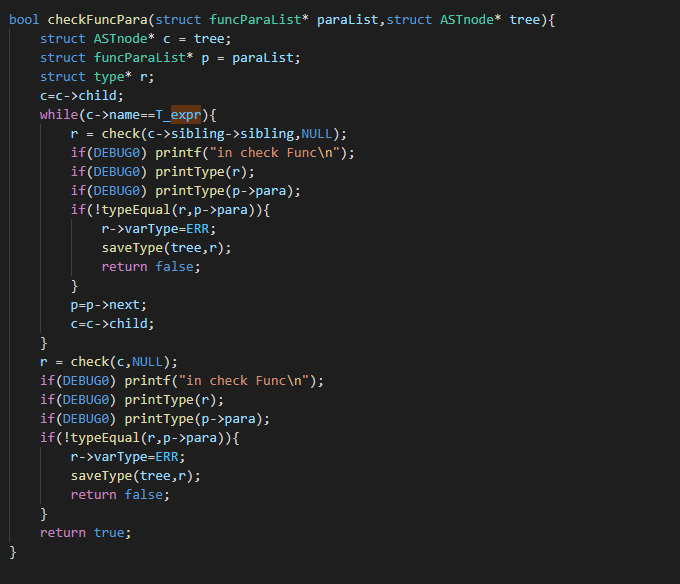


图 77：checkFuncPara函数

当用户调用有参数的函数时，用作检查函数的参数与用户输入是否匹配

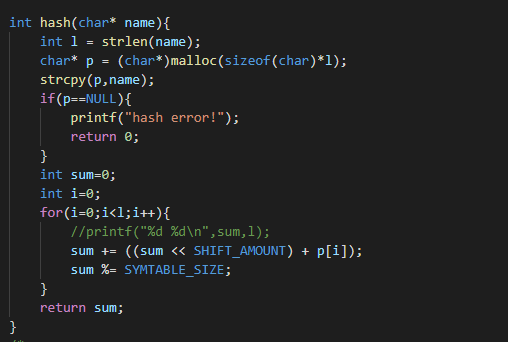


图 78：hash函数

把变量名转成Int值的一个函数

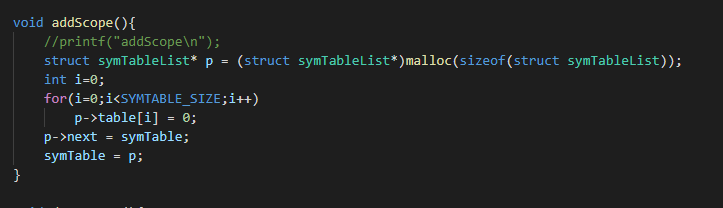


图 79：addScope函数

建立一个空的symbol table并指向上一个scope的symbol table，以达到产生新的命名空间的目的。

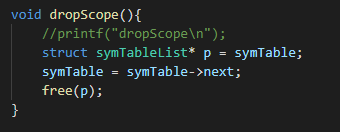


图 80：dropScope函数

用作去除最上层的scope

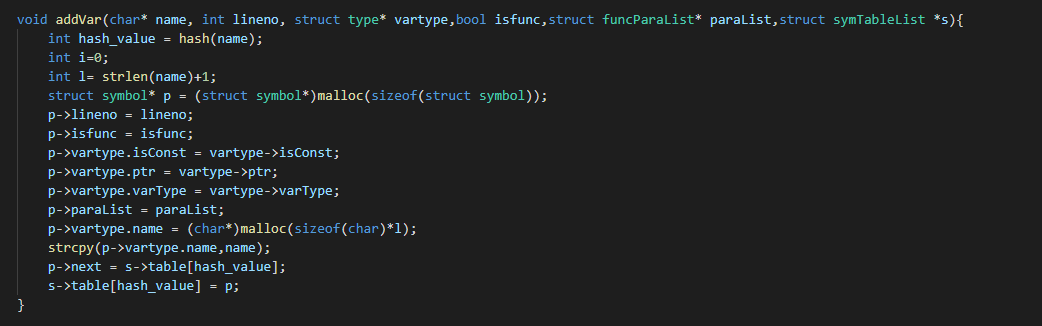


图 81：addVar函数

用作计算变量名/函数名的hash值并放入当前scope的symbol table这样的一个操作的函数

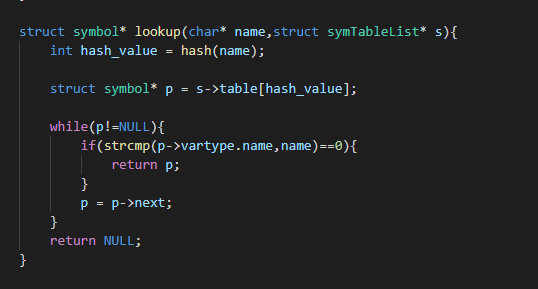


图 82：lookup函数

在参数所指定的scope中的symbol table中查找指定变量或函数

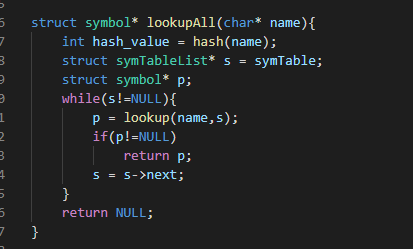


图 83：lookupAll函数

在所有的scope中的symbol table查找指定的变量或函数

七、优化考虑

7.1 常数折叠

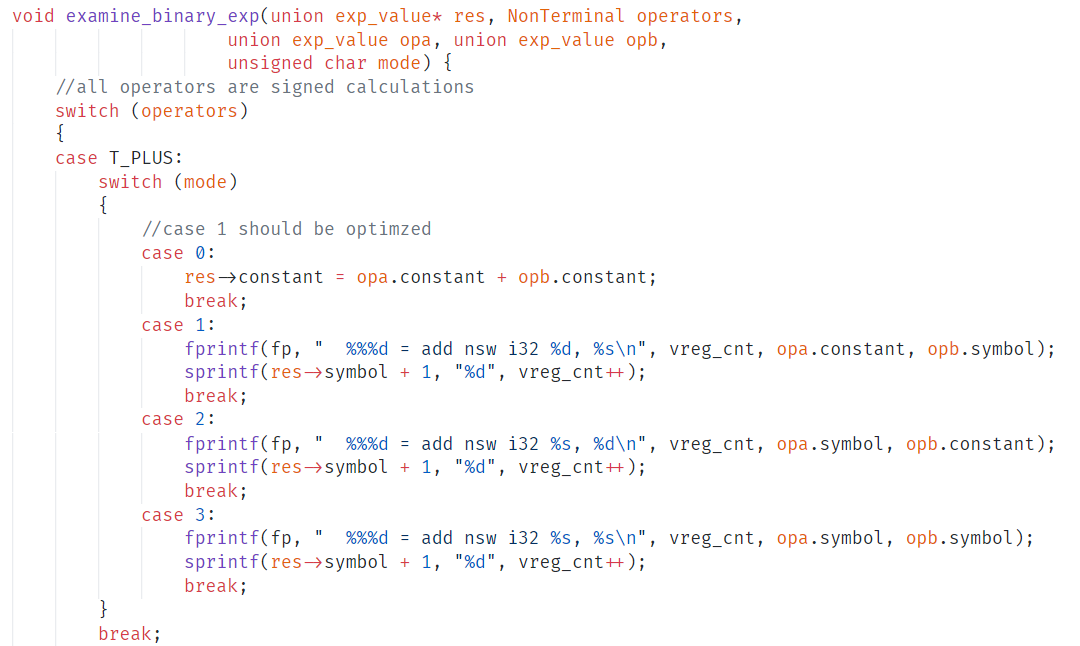


图 84：常数折叠情况一

在计算语法树邻近两个表达式节点时，如果两个节点储存的内容都是常量，则直接对节点中的常量进行运算操作，并保存到结果变量中。

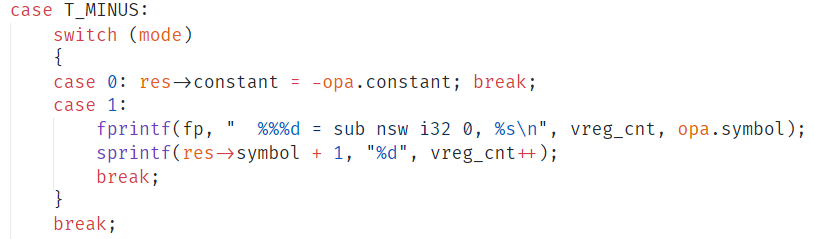


图 85：常数折叠情况二

同样地，计算一元表达式节点时，如果当前节点储存的内容也是常量，还是对节点中的常量做运算操作，保存至结果变量。

现对上述两种情况举例说明，如表达式a = 1 + 2 \* 3 – 6 / 2 + 5，程序首先会计算表达式2 \* 3以及6 / 2的结果，分别为6及3。因此，表达式变为a = 1 + 6 – 3 + 5。继续计算表达式中的常量，计算出a = 9，完成常量折叠。如果将表达式中的3变为-3。则常量计算过程变为

a = 1 – 6 – 3 + 5，a = -3。

7.2 中间代码优化

程序生成了LLVM的中间代码后，可以通过LLVM自带的opt命令对程序生成的中间代码进行优化，命令为opt -S -O3 “程序生成的中间代码文件名”，此命令对中间代码使用O3级别的优化。完成优化后，将优化后的中间代码储存在原文件中。

八、代码生成

8.1 变量及函数简介

代码生成重要由llvmgen.h以及llvmgen.c模块实现，能够对从yacc主程序接受来的语法树节点进行遍历，并根据遍历到的节点类型生成相对应的中间代码。

首先对llvmgen所使用的结构体、变量以及函数做大致介绍。

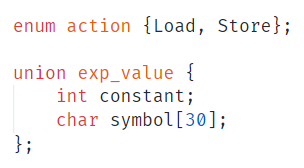


图 86：llvmgen使用的结构体

llvmgen模块使用了一个枚举类action以及一个联合类exp\_value协助中间代码的生成。

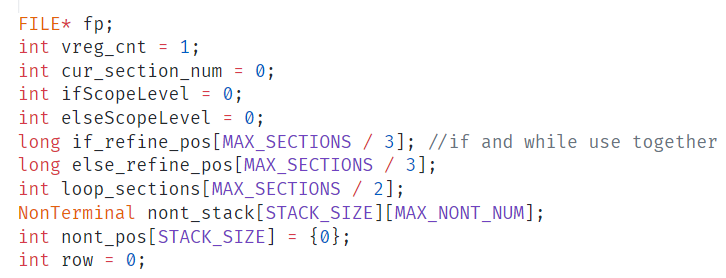


图 87：llvmgen所有全局变量声明

现对llvmgen全局变量做讲解。

fp：用于生成中间代码的文件指针；

vreg\_cnt：表示当前已经使用的虚拟寄存器数量；

cur\_seciton\_num：用于条件语句以及循环语句中，表示当前已经使用的跳转标签数量；

ifScopeLevel：表示当前while/if语句的作用域等级，与变量if\_refine\_pos搭配使用；

elseScopeLevel：表示当前else语句的作用域等级，与变量else\_refine\_pos搭配使用；

if\_refine\_pos：当程序出现if/while语句的嵌套时，此变量能够正确地为文件指针提供文件偏移地址，这些偏移地址是尚未赋值的跳转标签所处的位置；

else\_refine\_pos：与变量if\_refine\_pos作用一致，用于处理else语句中的嵌套；

loop\_sections：当遍历到迭代语句的节点时，将变量cur\_seciton\_num的值储存在此数组中；当迭代语句结束时，便可通过此数组输出正确的跳转标签。即完成迭代语句后，重新跳转回判断是否继续迭代的标签处；

nont\_stack：可以理解为一个存储文法的堆栈。保存当前遍历节点的类型，便于节点进行归约时判断当前处于yacc中的哪句文法；

non\_pos：当前文法中已经遍历的标记数；

row：当前文法堆栈的栈顶处。

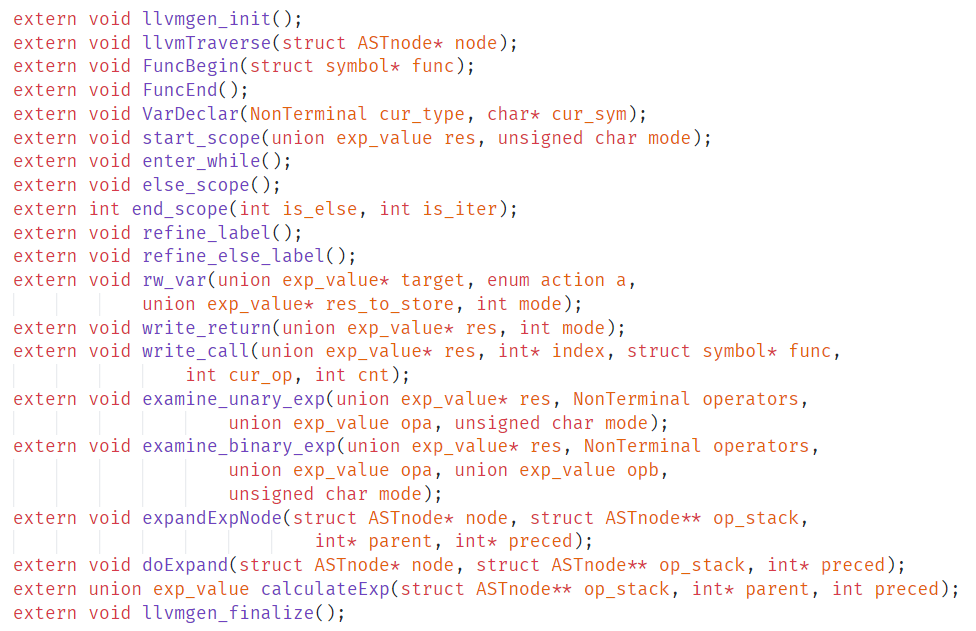


图 88：llvmgen所有函数声明

接下来介绍llvmgen各函数的功能。

llvmgen\_init：指定文件指针fp打开的文件名以及所进行的操作；

llvmTraverse：对语法树的节点进行遍历。可以认为是本模块的主入口，对当前遍历的节点所属语法做判断，并执行各语法相对应的语句以及函数调用；

FuncBegin：生成llvm函数声明的中间代码；

FuncEnd：生成结束函数声明的右大括号，并重设虚拟寄存器数及跳转标签数；

VarDeclar：生成局部变量声明的llvm中间代码；

start\_scope：当进入条件判断语句时，调用此函数生成条件判断以及标签跳转的llvm中间代码；

enter\_while：当进入循环判断语句时，调用此函数生成循环语句标签跳转的llvm中间代码；

else\_scope：当进入条件判断语句的else部分时，调用此函数生成else部分标签跳转的llvm中间代码；

end\_scope：当条件判断/循环判断语句结束后，调用此函数生成语句结束时标签跳转的llvm中间代码；

refine\_label：调用此函数对先前的条件判断/循环判断语句的跳转标签标号进行修正；

refine\_else\_label：调用此函数对先前判断语句else部分的跳转标签标号进行修正；

rw\_var：当需要读取/写入变量的值时，调用此函数生成相对应的llvm中间代码；

write\_return：当进入返回语句时，调用此函数生成返回语句的llvm中间代码；

write\_call：当表达式中存在函数调用时，调用此函数生成函数调用的llvm中间代码；

examine\_unary\_exp：当需要处理单目运算符时，调用此函数生成计算单目运算符的llvm中间代码；

examine\_binary\_exp：当需要处理双目运算符时，调用此函数生成计算双目运算符的llvm中间代码；

expandExpNode：对文法类型为exp的节点，遍历其所组成的子树，生成节点的运算优先级堆栈；

doExpand：对exp子树的节点进行深度优先搜索（Depth first search），如果当前遍历到的节点存在兄弟节点，则将当前节点保存至运算优先级堆栈中；

calculateExp：将从expandExpNode中得到的运算优先级堆栈进行计算。将当前所处堆栈位置的运算结果保存，便于后续节点使用；

llvmgen\_finalize：将中间代码所需的附加信息输出至文件中。完成输出后，关闭文件，完成中间代码生成。

8.2 llvmgen代码生成示例

现举例说明llvmgen生成中间代码的过程，假如即将需进行编译的文件内容如下：

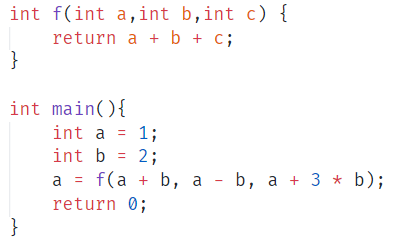


图 89：将要进行编译的文件内容

Llvmgen函数调用的流程大致如下：

llvmTraverse()->FuncBegin()

->write\_return()

->FuncEnd()

->FuncBegin() （如果变量定义没有赋予初始值，则后面这个函数将不会被调用）

->VarDeclar()->expandExpNode()->calculateExp()->rw\_var()

->VarDeclar()->expandExpNode()->calculateExp()->rw\_var()

->expandExpNode()->calculateExp()->write\_call()

->write\_return()

->FuncEnd()

九、测试案例

9.1 验证中间代码正确性

我们使用了三个不同的测试案例对编译器进行测试，此部分要求掌握LLVM中间代码的文法及规则。在了解其文法及规则后，即可验证编译器输出的中间代码是否正确。

第一个案例用于测试源代码带有复杂表达式时生成llvm中间代码的正确性。

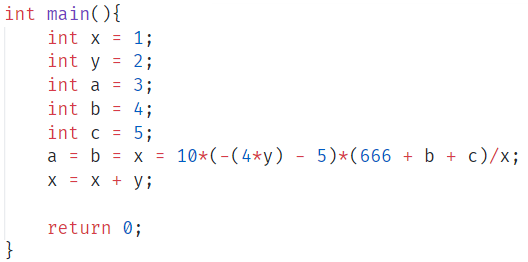


图 90：测试案例一源代码

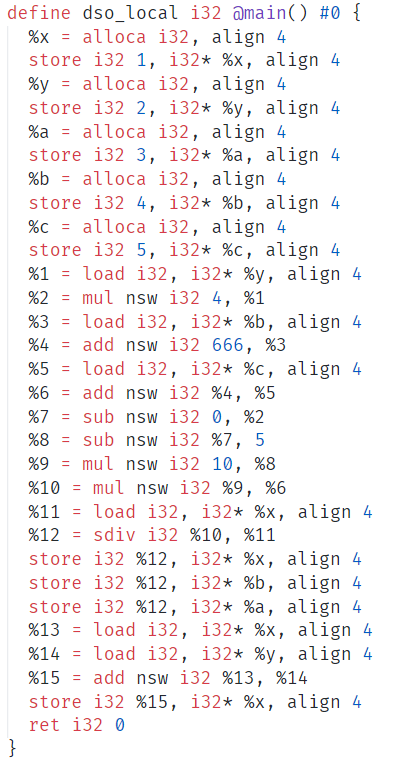


图 91：测试案例一目标代码

可以看到编译器能够生成正确的中间代码。

第二个案例测试编译器对函数声明及调用生成中间代码的正确性。

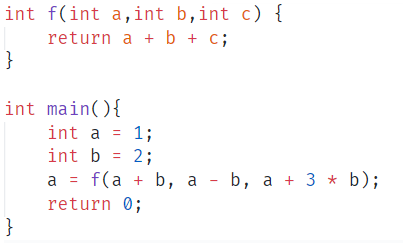


图 92：测试案例二源代码

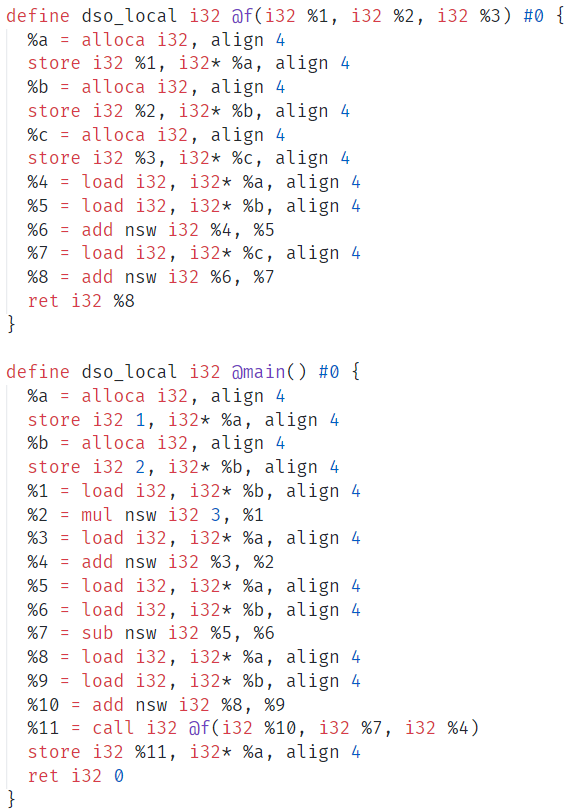


图 93：测试案例二目标代码

中间代码同样能够保持正确性。

最后一个案例测试编译器对条件判断以及循环判断语句生成中间代码的正确性。

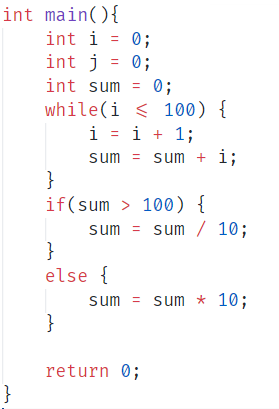


图 94：测试案例三源代码

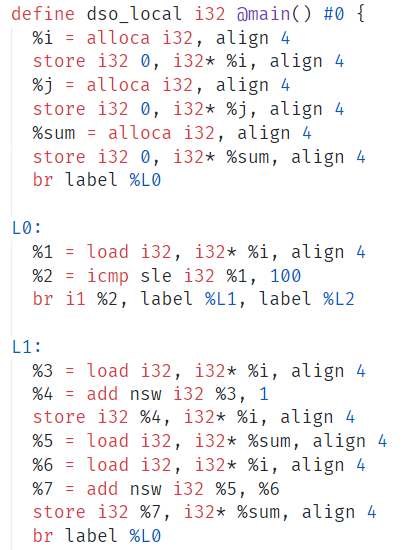


图 95：测试案例三目标代码（一）

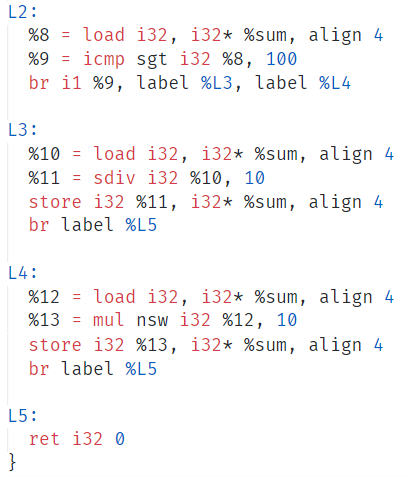


图 96：测试案例三目标代码（二）

可以看到中间代码生成依然正确。

以上的所有文件都已存放在与本报告同目录中的testset文件夹中。如果希望将中间代码生成为可执行文件，可以在命令行中输入

clang “程序生成的中间代码文件名”-o “可执行文件的文件名”

即可生成可执行文件并运行。

此外，语法树的可视化结果也已经保存在testset文件夹中。当我们的编译器完成对testdata.c的编译后，也会在相同目录下生成一个名为ASTtree.txt的文件。此文件即为语法树的可视化结果。

十、实验总结

经过本次实验的项目实践后，我们对课堂上学到的知识有了更深的理解，同时也锻炼了自己的代码水平。项目实践的过程中，我们深刻地认识到分工后了解对方工作的重要性，如果没有和队友认真跟进过代码进度的话，很有可能最终大家编写出来的接口是不兼容的。我们小组在项目的后期发生了这一情况。幸好的是，在经过一段时间的磨合后，我们的接口总算能够相兼容。这一经验也使我们了解到软件工程中保持高效团队合作的重要性，为了实现这个目标，我们必须保持长久且密切的有效沟通，了解对方当前正在面临的难题，并在关键时刻为对方提供援助。