

**本科实验报告**

|  |  |
| --- | --- |
| 课程名称： | 编译原理 |
| 姓 名： | 刘一辰、黄海烽、周思颖 |
| 学 院： | 计算机科学与技术学院 |
| 系： | 计算机系 |
| 专 业： | 计算机科学与技术 |
| 学 号： | 3180102886、3180102339、3180104714 |
| 指导教师： | 李莹 |

目录

[第一章 词法分析 3](#_Toc74675675)

[1.1 Lex 4](#_Toc74675676)

[1.2具体实现 4](#_Toc74675677)

[1.2.1 定义段 4](#_Toc74675678)

[1.2.2 词法规则段 4](#_Toc74675679)

[第二章 语法分析 8](#_Toc74675680)

[2.1 YACC 8](#_Toc74675681)

[2.2抽象语法树 9](#_Toc74675682)

[2.2.1 ASTNode类 9](#_Toc74675683)

[2.2.2 ExprNode类和StatNode类 9](#_Toc74675684)

[2.2.3 Program类 10](#_Toc74675685)

[2.2.4 ConstValue类 10](#_Toc74675686)

[2.2.5 SkyArrayType类 10](#_Toc74675687)

[2.2.6 SkyType类 11](#_Toc74675688)

[2.2.7 Identifier类 12](#_Toc74675689)

[2.2.8 VarDec类 12](#_Toc74675690)

[2.2.9 ConstDec类 13](#_Toc74675691)

[2.2.10 FuncDec类 14](#_Toc74675692)

[2.2.11 ClassBody类和ClassDec类 14](#_Toc74675693)

[2.2.12 BinaryExpr类 15](#_Toc74675694)

[2.2.13 CompoundStat类 16](#_Toc74675695)

[2.2.14 ArrayReference类和ClassRef类 16](#_Toc74675696)

[2.2.15 FuncCall类 17](#_Toc74675697)

[2.2.16 PointerNode类 17](#_Toc74675698)

[2.2.17 GlobalArea类 18](#_Toc74675699)

[2.2.18 其他类 19](#_Toc74675700)

[2.3语法分析的具体实现 19](#_Toc74675701)

[第三章 语义分析 21](#_Toc74675702)

[3.1 LLVM概述 21](#_Toc74675703)

[3.2 LLVM IR 22](#_Toc74675704)

[3.2.1 IR布局 22](#_Toc74675705)

[3.2.2 IR上下文环境 22](#_Toc74675706)

[3.2.3 IR核心类 22](#_Toc74675707)

[3.3 IR生成 22](#_Toc74675708)

[3.3.1 运行环境设计 22](#_Toc74675709)

[3.3.2 类型系统 23](#_Toc74675710)

[第四章 中间代码生成 23](#_Toc74675711)

[第五章 优化以及进阶主题 23](#_Toc74675712)

[第六章 测试结果 23](#_Toc74675713)

[6.1 测试用例1：快速排序 23](#_Toc74675714)

[6.1.1 测试用例1代码 23](#_Toc74675715)

[6.1.2输入与输出 24](#_Toc74675716)

[6.2 测试用例2：矩阵乘法 24](#_Toc74675717)

[6.2.1 测试用例2代码 25](#_Toc74675718)

[6.2.2 输入与输出 25](#_Toc74675719)

[6.3 测试用例3：选课助手 26](#_Toc74675720)

[6.3.1 测试用例3代码 26](#_Toc74675721)

[6.3.2 输入与输出 30](#_Toc74675722)

# 第一章 词法分析

词法分析是计算机科学中将字符序列转换为标记（token）序列的过程。标记经常使用正则表达式进行定义，像Lex一类的词法分析器生成器就支持使用正则表达式。语法分析器读取输入字符流、从中识别出语素、最后生成不同类型的标记。

## 1.1 Lex

一般而言，一个Lex源程序分为三部分，三部分之间以符号%%分隔。第一部分为定义段，第二部分为词法规则段，第三部分为辅助函数段。

## 1.2具体实现

### 1.2.1 定义段

Sky的Lex源程序在定义区导入了需要的头文件，包括：

* stdio.h（C语言标准输入输出头文件）
* string（C++ std::string头文件）
* nodeList.h（抽象语法树头文件）
* sky.tab.h（yacc生成的词法分析器头文件），

然后声明了lex需要的yywrap函数。

%{

#include<stdio.h>

#include<string>

#include "nodeList.h"

#include "sky.tab.h"

extern "C" int **yywrap**() {return 1;}

using namespace std;

%}

### 1.2.2 词法规则段

首先，需要排除空格、水平制表符、回车换行、注释的干扰，方法是把他们解析为空格。

[ \t\n]                       {  }

"//"[^\n]\*                    { }

然后解析关键字、运算符和分隔符，由于这些字符是固定的，所以正则表达式只需要也是固定字符。

"var"                         {return VAR;}

"let"                         {return LET;}

"new"                         {return NEW;}

"delete"                      {return DELETE;}

"func"                        {return FUNCTION;}

"return"                      {return JUMP\_RETURN;}

"if"                          {return IF;}

"else"                        {return ELSE;}

"for"                         {return FOR;}

"while"                       {return WHILE;}

"in"                          {return IN;}

"continue"                    {return JUMP\_CONTINUE;}

"break"                       {return JUMP\_BREAK;}

"class"                       {return CLASS;}

"this"                        {return THIS;}         */\*自身指针\*/*

"lambda"                      {return LAMBDA;}

"int"                         {return TYPE\_INT;}

"int\*"                        {return TYPE\_INT\_POINTER;}

"int64"                       {return TYPE\_INT\_64;}

"int64\*"                      {return TYPE\_INT\_64\_POINTER;}

"char"                        {return TYPE\_CHAR;}

"char\*"                       {return TYPE\_CHAR\_POINTER;}

"float"                       {return TYPE\_FLOAT;}

"float\*"                      {return TYPE\_FLOAT\_POINTER;}

"double"                      {return TYPE\_DOUBLE;}

"double\*"                     {return TYPE\_DOUBLE\_POINTER;}

"bool"                        {return TYPE\_BOOL;}

"bool\*"                       {return TYPE\_BOOL\_POINTER;}

"{"                           {return('{');}

"}"                           {return('}');}     

"("                           {return('(');}      

")"                           {return(')');}

"["                           {return('[');}

"]"                           {return(']');}

"."                           {return('.');}

","                           {return(',');}

":"                           {return(':');}

";"                           {return(';');}

"="                           {return('=');}        /\*赋值\*/

"+="                          {return ADD\_ASSIGN;}

"-="                          {return SUB\_ASSIGN;}

"\*="                          {return MUL\_ASSIGN;}

"/="                          {return DIV\_ASSIGN;}

"%="                          {return MOD\_ASSIGN;}

"&="                          {return AND\_ASSIGN;}

"^="                          {return XOR\_ASSIGN;}

"|="                          {return OR\_ASSIGN;}

"!"                           {return OPER\_NOT;}

"+"                           {return OPER\_PLUS;}

"-"                           {return OPER\_MINUS;}

"\*"                           {return('\*');}

"/"                           {return OPER\_DIV;}

"%"                           {return OPER\_MOD;}

">>"                          {return OPER\_RIGHT;}

"<<"                          {return OPER\_LEFT;}

"->"                          {return OPER\_PTR;}

"&&"                          {return OPER\_AND;}

"&"                           {return('&');}

"||"                          {return OPER\_OR;}

"<"                           {return OPER\_LT;}

"<="                          {return OPER\_LE;}

">"                           {return OPER\_GT;}

">="                          {return OPER\_GE;}

"=="                          {return OPER\_EQ;}

"!="                          {return OPER\_NE;}

标识符是由字母或下划线开头，由字母、数字和下划线组成，并且不能是关键字的字符串。Sky编译器在词法分析阶段只校验标识符是否符合规则，而不会校验其是否存在。不同于运算符，标识符需要额外保存字符串值。

[a-zA-Z\_][a-zA-Z0-9\_]\*        {

                                yylval.sVal = new char[yyleng+1];

**strcpy**(yylval.sVal, yytext);

                                //**printf**("%s\n", yytext);

                                return IDENTIFIER;

                              }

其他需要额外保存值的单词：

* 在函数、类型等单词中，词法分析器需要记录字符串串值以使得语法分析器能区分是哪个函数或类型。

"true"|"false"                {

                                bool tmp;

**sscanf**(yytext, "%d", &tmp);

                                yylval.bVal = tmp;

                                return BOOLEAN;

                              }

"\_\_init\_\_"                    {

                                yylval.sVal = new char[yyleng+1];

**strcpy**(yylval.sVal, yytext);

                                return INIT;

                              }

"\_\_del\_\_"                     {

                                yylval.sVal = new char[yyleng+1];

**strcpy**(yylval.sVal, yytext);

                                return DEL;

                              }

"main"                        {

                                yylval.sVal = new char[yyleng+1];

**strcpy**(yylval.sVal, yytext);

                                //**printf**("%s\n", yytext);

                                return MAIN;

                              }

* 字符型使用以'开头结尾，中间为任意字符的正则表达式识别，将中间字符存储。
* 字符串型使用以＂开头结尾，中间为＼开头或不包括＼和＂的任意字符的正则表达式识别，将中间字符串存储。

\'.\'                         {

                                yylval.cVal = yytext[1];

                                return CHAR;

                              }

\'\\[nt0]\'                   {

                                if (yytext[2] == 'n') {

                                    yylval.cVal = '\n';

                                }

                                if (yytext[2] == 't') {

                                    yylval.cVal = '\t';

                                }

                                if (yytext[2] == '0') {

                                    yylval.cVal = '\0';

                                }

                                return CHAR;

                              }

\"(\\.|[^\\\"])\*\"   {

                       yylval.sVal = new char[yyleng-1];

**memcpy**(yylval.sVal, yytext+1, **strlen**(yytext)-2);

                       yylval.sVal[yyleng-2] = '\0';

                       //**printf**("%s\n", yylval.sVal);

                       return STRING;

                     }

* 对于单精度浮点型、双精度浮点型和整型，在词法分析阶段使用C语言转换为对应类型存储。

[0-9]+\.[0-9]+                {

                                double tmp;

**sscanf**(yytext, "%lf", &tmp);

                                yylval.dVal = tmp;

                                return DOUBLE;

                              }

[0-9]+\.[0-9]+[fF]            {

                                float tmp;

**sscanf**(yytext, "%f[fF]", &tmp);

                                yylval.fVal = tmp;

                                return FLOAT;

                              }

[0-9]+                        {

                                int tmp;

**sscanf**(yytext, "%d", &tmp);

                                yylval.iVal = tmp;

                                return INTEGER;

                              }

# 第二章 语法分析

## 2.1 YACC

yacc（Yet Another Compiler Compiler），是Unix/Linux上一个用来生成编译器的编译器（编译器代码生成器）。yacc生成的编译器主要是用C语言写成的语法解析器（Parser），需要与词法分析器Lex一起使用，再把两部分产生出来的C程序一并编译。

yacc的输入是巴科斯范式（BNF）表达的语法规则以及语法规约的处理代码，输出的是基于表驱动的编译器，包含输入的语法规约的处理代码部分。

与Lex相似，yacc的输入文件由以%%分割的三部分组成，分别是定义段、规则段和程序段。三部分的功能与Lex相似，不同的是规则段的正则表达式替换为CFG，在定义段要提前声明好使用到的终结符以及非终结符的类型。

## 2.2抽象语法树

### 2.2.1 ASTNode类

**ASTNode**类是一个抽象类，其意义为"抽象语法树的节点"，这是抽象语法树所有节点的共同祖先。该类拥有一个用于生成中间代码的纯虚函数**convertToCode**和一个默认析构函数**~ASTNode**。

class **ASTNode**{

public:

    virtual **Value** \***convertToCode**() = 0;

    virtual **~ASTNode**() = default;

};

### 2.2.2 ExprNode类和StatNode类

**ExprNode**和**StatNode**是大部分实体类的父类。

* **ExprNode**类是表达式类，它的子类的特征是可获得值或可更改值，也就是左值或者右值，比如二元表达式或变量。

*// the node for expression*

*// expression has return value*

class **ExprNode**: public **ASTNode** {

public:

**SkyTypes** type;

};

* **StatNode**类是语句类，它的子类的特征是该类会进行操作，比如赋值、比较、条件控制等。

*// the node for statement*

*// statement doesn't have return value*

class **StatNode**: public **ASTNode** {

public:

*//    void forward();*

    void **backward**();

*//    BasicBlock \*afterBB{};*

};

### 2.2.3 Program类

**Program**类的意义是程序，该类是最顶层的实体类，包括全局区域globalArea和main函数对象mainFunc。

*// Program is split into GlobalArea and MainFunction*

class **Program**: public **StatNode** {

public:

**Program**(**GlobalArea** \*globalArea, **FuncDec** \*mainFunc): **globalArea**(globalArea), **mainFunc**(mainFunc) { }

**Value** \***convertToCode**() override;

**GlobalArea**\* globalArea;

**FuncDec** \*mainFunc;

private:

};

### 2.2.4 ConstValue类

**ConstValue**类的意义是常量节点，由于常量的类型很多，所以**ConstValue**是一个抽象类，具体由**SkyInt**、**SkyDouble**、**SkyFloat**、**SkyChar**、**SkyCharPointer**、**SkyBool**六个子类完成，通过**getType**和**getValue**函数获得真实的值。

class **ConstValue**: public **ExprNode**{

public:

    virtual **SkyVarType** **getType**() = 0;

    virtual **ConstValueUnion** **getValue**() = 0;

    virtual **ConstValue** \***operator-**() = 0;

**Constant**\* **create**();

};

### 2.2.5 SkyArrayType类

**SkyArrayType**类的意义是处理各种数组的类型。

*// Node for array type*

*// Example:*

*//      int[10]    =>   type = SKY\_INT,  size = 10*

class **SkyArrayType**: public **StatNode** {

public:

**SkyArrayType**(**SkyType** \*type, int size): **type**(type), **size**(size) { }

**Value** \***convertToCode**() override;

**SkyType** \*type;

    int size;

};

### 2.2.6 SkyType类

**SkyType**类的意义是Sky支持的类型，包括数组、空类型、变量类型、函数类型和自动类型。

*// all the types in Sky*

*// including:   SKY\_ARRAY : array type   Example: int[10]*

*//              SKY\_VAR : simple types (SKY\_INT, SKY\_FLOAT, ...)*

*//              SKY\_VOID : now is only used in the function return type, which means the function has no return value*

*//              SKY\_FUNC : function type*

*//              SKY\_AUTO : auto type (type inference)*

class **SkyType**: public **StatNode** {

public:

    explicit **SkyType**(**SkyArrayType** \*arrayType): **arrayType**(arrayType), **type**(SKY\_ARRAY) { }

    explicit **SkyType**(**SkyVarType** \*varType): **varType**(varType), **type**(SKY\_VAR) { }

    explicit **SkyType**(**SkyFuncType** \*funcType): **funcType**(funcType), **type**(SKY\_FUNC) { }

    explicit **SkyType**(**SkyAutoType** \*autoType): **autoType**(autoType), **type**(SKY\_AUTO) { }

**SkyType**(): **type**(SKY\_VOID) { }

**Value** \***convertToCode**() override;

**Constant**\* **Create**();

**Type**\* **toLLVMType**();

*//    Constant\* initValue(ConstValue \*v = nullptr);*

**SkyArrayType** \*arrayType{};

**SkyVarType** \*varType{};

**SkyFuncType** \*funcType{};

**SkyAutoType** \*autoType{};

**SkyTypes** type;

};

### 2.2.7 Identifier类

**Identifier**的意义是标识符，包括一个name字段。

*// In human terms, it can be seen as the name of something(variable, function, class, const, ...)*

class **Identifier**: public **ExprNode** {

public:

    explicit **Identifier**(char\* name): **name**(name) { }

**Value** \***convertToCode**() override;

    char\* name;

};

### 2.2.8 VarDec类

**VarDec**类的意义是变量声明，由变量名**Identifier**、变量类型**SkyType**和表达式**ExprNode**三部分构成。

*// Node for variable declaration*

*// Example:*

*//      name = expression*

*//      name : type*

class **VarDec**: public **StatNode** {

public:

**VarDec**(**Identifier** \*id, **SkyType** \*type, **ExprNode**\* expr): **id**(id), **type**(type), **expr**(expr), **global**(false) { }

**VarDec**(**Identifier** \*id, **SkyFuncType** \*funcType): **id**(id) {

        type = new **SkyType**(funcType);

    }

    bool **isGlobal**() const {

        return this->global;

    }

    void **setGlobal**() {

        this->global = true;

    }

**Value** \***convertToCode**() override;

**Identifier** \*id;

**SkyType** \*type;*// if type == SKY\_AUTO, need Type Inference*

**ExprNode** \*expr{};*// when type == SKY\_AUTO, calculate this expr to get the type*

    bool global{};*// whether is the global variable*

};

### 2.2.9 ConstDec类

**ConstDec**类的意义是常量声明，由常量名**Identifier**、常量值**ConstValue**和常量类型**SkyType**三部分构成。

*// Node for const declaration*

*// Example:*

*//      name = value*

*// The type of the value will be recognized in the parsing phase*

class **ConstDec**: public **StatNode** {

public:

**ConstDec**(**Identifier** \*id, **ConstValue** \*cv): **id**(id), **value**(cv), **global**(false) {

        type = new **SkyType**(new **SkyVarType**(value->**getType**()));

    }

**Value** \***convertToCode**() override;

    bool **isGlobal**() const {

        return this->global;

    }

    void **setGlobal**() {

        this->global = true;

    }

**Identifier** \*id;

**ConstValue** \*value;

**SkyType** \*type;

    bool global;*// whether is the global const*

};

### 2.2.10 FuncDec类

**FuncDec**类的意义是函数声明，由函数名**Identifier**、函数类型**SkyFuncType**两部分构成。

*// Node for function declaration*

*// Example:*

*//      func func\_name(paraList) compound\_statement      (retType = SKY\_VOID)*

*//      func func\_name(paraList) -> retType compound\_statement*

class **FuncDec**: public **StatNode** {

public:

**FuncDec**(**Identifier** \*name, **SkyFuncType** \*funcType): **id**(name), **funcType**(funcType) { }

**Value** \***convertToCode**() override;

private:

**Identifier** \*id;

**SkyFuncType** \*funcType;

};

### 2.2.11 ClassBody类和ClassDec类

**ClassBody**类的意义是类的主体，由构造函数init、析构函数del、函数声明列表funcList三部分构成。

*// Node for class body*

*// Class body must contain init and del function*

*// Example:*

*//      func \_\_init\_\_(paraList) -> retType compound\_statement*

*//      func \_\_del\_\_(paraList) -> retType compound\_statement*

*//      funcDecList*

class **ClassBody**: public **StatNode** {

public:

**ClassBody**(**FuncDec** \*init, **FuncDec** \*del, **FuncDecList** \*funcList): **initFunc**(init), **delFunc**(del), **funcList**(funcList) { }

**Value** \***convertToCode**() override { return nullptr; }

private:

**FuncDec** \*initFunc, \*delFunc;

**FuncDecList** \*funcList;

};

**ClassDec**类的意义是类的声明，由类名name、父类father、类的主体body三部分构成。

*// Node for class declaration*

*// Example:*

*//      class class\_name { class\_body }*

*//      class class\_name : father { class\_body }*

class **ClassDec**: public **StatNode** {

public:

**ClassDec**(**Identifier** \*name, **Identifier** \*father, **ClassBody** \*body): **name**(name), **father**(father), **body**(body) { }

**Value** \***convertToCode**() override { return nullptr; }

private:

**Identifier** \*name, \*father;*// father can be nullptr*

**ClassBody** \*body;

};

### 2.2.12 BinaryExpr类

**BinaryExpr**类的意义是二元表达式，节点存储有左表达式、右表达式和操作符。

*// Node for binary expression*

*// Example:*

*//      leftExpr op rightExpr*

*// The priority is solved in the parsing phase, so just do the operation directly*

class **BinaryExpr**: public **ExprNode** {

public:

**BinaryExpr**(**ExprNode** \*left, **BinaryOperators** op, **ExprNode** \*right): **left**(left), **op**(op), **right**(right) { }

**Value** \***convertToCode**() override;

private:

**ExprNode** \*left, \*right;

**BinaryOperators** op;

};

### 2.2.13 CompoundStat类

**CompoundStat**类的意义是复合语句，即有一系列语句组成的语句列表，该类由**StatList**组成。

*// Node for compound statement*

*// Example:*

*//      { statement\_list }*

class **CompoundStat**: public **StatNode** {

public:

**CompoundStat**() {

        statList = new **StatList**();

    }

    explicit **CompoundStat**(**StatList** \*statList): **statList**(statList) { }

**Value**\* **convertToCode**() override;

**StatList** \*statList;

};

### 2.2.14 ArrayReference类和ClassRef类

**ArrayReference**类的意义是数组成员的引用，**ClassRef**类的意义是类成员的引用。

*// Node for array element reference*

*// Example:*

*//      arrName[index]*

*// index is saved as an expression(ExprNode)*

class **ArrayReference**: public **ExprNode** {

public:

**ArrayReference**(**Identifier** \*id, **ExprNode** \*subInd): **id**(id), **subInd**(subInd) { }

**Value** \***convertToCode**() override;

**Value**\* **getValueI**();

private:

**Identifier** \*id;

**ExprNode** \*subInd;

};

*// Node for class member reference*

*// Example:*

*//      className.classMemberName*

class **ClassRef**: public **ExprNode** {

public:

**ClassRef**(**Identifier** \*id, **Identifier** \*childId): **id**(id), **childId**(childId) { }

**Value** \***convertToCode**() override { return nullptr; }

private:

**Identifier** \*id, \*childId;*// id: className,  childId: classMemberName*

};

### 2.2.15 FuncCall类

**FuncCall**类的意义是函数调用，由函数名id和参数列表args两部分组成，该类同时继承自**ExprNode**类和**StatNode**类。

*// Node for function call*

*// It can be an expression or a statement, so it is inherited from both ExprNode and StatNode*

class **FuncCall**: public **ExprNode**, public **StatNode** {

public:

**FuncCall**(**Identifier** \*id, **ExprList** \*args): **id**(id), **args**(args) { }

**Value** \***convertToCode**() override;

**Value** \***callSysIO**();

private:

**Identifier** \*id;*// the function name*

**ExprList** \*args;*// the arguments of the function*

};

### 2.2.16 PointerNode类

**PointerNode**类的意义是指针节点。

*// Node for Pointer*

*// Example:*

*//      \*(a+2)*

*//      \*a*

*//      \*a[10]*

*//      \*func(1,2)*

*//      \*a.b*

*// a+2, a, a[10], func(1,2), a.b above are all saved as an expression(ExprNode)*

*// and the expression value should be calculated first, then use the '\*'*

class **PointerNode**: public **ExprNode** {

public:

    explicit **PointerNode**(**ExprNode** \*expr): **expr**(expr) { }

**Value** \***convertToCode**() override;

**ExprNode** \*expr;

};

### 2.2.17 GlobalArea类

**GlobalArea**类的意义是全局区域，该类由常量声明constDecList、变量声明varDecList、函数声明funcDecList、类声明classDecList四部分构成。

*// GlobalArea can only do some definition*

*// including const, variable, function and class*

*// Especially, function and class can only be defined in the GlobalArea*

class **GlobalArea**: public **StatNode** {

public:

**GlobalArea**() {

        constDecList = new **ConstDecList**();

        varDecList = new **VarDecList**();

        funcDecList = new **FuncDecList**();

        classDecList = new **ClassDecList**();

    }

    void **addConstDec**(**ConstDecList** \*cd) {

        for (auto & constDec : \*cd) {

            constDec->**setGlobal**();

        }

        constDecList->**insert**(constDecList->**end**(), cd->**begin**(), cd->**end**());

    }

    void **addVarDec**(**VarDecList** \*vd) {

        for (auto & varDec : \*vd) {

            varDec->**setGlobal**();

        }

        varDecList->**insert**(varDecList->**end**(), vd->**begin**(), vd->**end**());

    }

    void **addFuncDec**(**FuncDec** \*fd) {

        funcDecList->**push\_back**(fd);

    }

    void **addClassDec**(**ClassDec** \*cd) {

        classDecList->**push\_back**(cd);

    }

**Value** \***convertToCode**() override;

private:

**ConstDecList** \*constDecList{};*// list of const declaration*

**VarDecList** \*varDecList{};*// list of variable declaration*

**FuncDecList** \*funcDecList{};*// list of function declaration*

**ClassDecList** \*classDecList{};*// list of class declaration*

};

### 2.2.18 其他类

还有一些其他继承自**StatNode**类的语句类，如**IfStat**类、**ForStat**类、**WhileStat**类和**JumpStat**类等。

## 2.3语法分析的具体实现

首先在定义段声明好终结符和非终结符类型。

%union {

    int iVal;

    float fVal;

    double dVal;

    char cVal;

    char\* sVal;

    bool bVal;

    Program \*program;

    GlobalArea \*globalArea;

    ConstDec \*constDec;

    ConstDecList \*constDecList;

    ConstValue \*constValue;

    VarDec \*varDec;

    VarDecList \*varDecList;

    SkyType \*skyType;

    SkyVarType \*skyVarType;

    SkyArrayType \*skyArrayType;

    FuncDec \*funcDec;

    CompoundStat \*compoundStat;

    StatList \*statList;

    IfStat \*ifStat;

    JumpStat \*jumpStat;

    ExprNode \*expression;

    ExprList \*exprList;

    AssignStat \*assignStat;

    ClassDec \*classDec;

    Identifier \*identifier;

    ClassBody \*classBody;

    FuncDecList \*funcDecList;

    StatNode \*statement;

    VarDecListNode \*varDecListNode;

    ConstDecListNode \*constDecListNode;

    SkyFuncType \*skyFuncType;

}

%type<program>                          program

%type<globalArea>                       global\_area

%type<constDec>                         const\_expr

%type<constDecList>                     const\_list

%type<constDecListNode>                 const\_declaration

%type<constValue>                       const\_value

%type<varDec>                           var\_expr

%type<varDecList>                       var\_list para\_list

%type<varDecListNode>                   var\_declaration

%type<skyType>                          type\_declaration

%type<skyVarType>                       var\_type

%type<skyArrayType>                     array\_type\_declaration

%type<funcDec>                          func\_declaration main\_func class\_init class\_del

%type<compoundStat>                     compound\_statement

%type<statList>                         statement\_list

%type<ifStat>                           branch\_statement

%type<statement>                        for\_statement statement

%type<jumpStat>                         jump\_statement

%type<expression>                       expression expression\_or expression\_and expr expr\_shift term factor number

%type<exprList>                         expression\_list

%type<assignStat>                       assign\_statement

%type<classDec>                         class\_declaration

%type<identifier>                       inherit\_part name

%type<classBody>                        class\_body

%type<funcDecList>                      func\_declaration\_list

%type<skyFuncType>                      func\_type lambda\_expression

%token<iVal> INTEGER

%token<fVal> FLOAT

%token<dVal> DOUBLE

%token<cVal> CHAR

%token<sVal> STRING IDENTIFIER MAIN INIT DEL

%token<bVal> BOOLEAN

%token  PRINT SCAN

        VAR LET NEW DELETE LAMBDA

        FUNCTION JUMP\_BREAK JUMP\_CONTINUE JUMP\_RETURN

        IF ELSE FOR WHILE IN

        CLASS THIS

        TYPE\_INT TYPE\_INT\_POINTER TYPE\_INT\_64 TYPE\_INT\_64\_POINTER

        TYPE\_CHAR TYPE\_CHAR\_POINTER

        TYPE\_FLOAT TYPE\_FLOAT\_POINTER TYPE\_DOUBLE TYPE\_DOUBLE\_POINTER

        TYPE\_BOOL TYPE\_BOOL\_POINTER

        ADD\_ASSIGN SUB\_ASSIGN MUL\_ASSIGN DIV\_ASSIGN MOD\_ASSIGN AND\_ASSIGN XOR\_ASSIGN OR\_ASSIGN

        OPER\_PLUS OPER\_MINUS OPER\_DIV OPER\_MOD OPER\_RIGHT OPER\_LEFT OPER\_PTR OPER\_AND OPER\_OR OPER\_NOT

        OPER\_LT OPER\_LE OPER\_GT OPER\_GE OPER\_EQ OPER\_NE

        LF

接着按从上往下的顺序构造语法树，文法详见sky.y文件。

# 第三章 语义分析

## 3.1 LLVM概述

LLVM是一套编译器基础设施项目，它以C++写成，包含一系列模块化的编译器组件和工具链，用来开发编译器前端和后端。它是为了任意一种编程语言而写成的程序，利用虚拟技术创造出编译时期、链接时期、运行时期以及“闲置时期”的优化。

* 前端：LLVM最初被用来取代GCC中的代码产生器，许多GCC的前端已经可以与其运行，LLVM目前支持Ada、C语言、C++、D语言、Fortran等语言的编译。
* 中间端：LLVM的核心是中间端表达式（Intermediate Representation，IR），一种类似汇编的底层语言。IR是一种强类型的精简指令集，并对目标指令集进行了抽象。LLVM支持三种表达形式：人类可读的汇编，在C++中对象形式和序列化后的bitcode形式。
* 后端：LLVM已经支持多种后端指令集，包括ARM、Qualcomm Hexagon、MIPS、Nvidia并行指令集等。

## 3.2 LLVM IR

LLVM IR是LLVM的核心所在，通过将不同高级语言的前端变换成LLVM IR进行优化、链接后再传给不同目标的后端转换成为二进制代码，前端、优化、后端三个阶段互相解耦，这种模块化的设计使得LLVM优化不依赖于任何源码和目标机器。

### 3.2.1 IR布局

### 3.2.2 IR上下文环境

### 3.2.3 IR核心类

### 3.3 IR生成

### 3.3.1 运行环境设计

### 3.3.2 类型系统

# 第四章 中间代码生成

# 第五章 优化以及进阶主题

# 第六章 测试结果

## 6.1 测试用例1：快速排序

### 6.1.1 测试用例1代码

|  |
| --- |
| var a: int[10010];  var n: int;  func qsort(l : int, r : int) -> int {  var i: int, j: int, mid: int, tmp: int;  i = l;  j = r;  mid = a[(l + r) >> 1];  while (i <= j) {  while (a[i] < mid) { i = i + 1; }  while (a[j] > mid) { j = j - 1; }  if (i <= j) {  tmp = a[i];  a[i] = a[j];  a[j] = tmp;  i = i + 1;  j = j - 1;  }  }  //printf("%d %d %d %d@", l, r, i, j);  //var k: int;  //for k in [0, n, 1] {  // printf("%d ", a[k]);  //}  if (l < j) { qsort(l, j); }  if (i < r) { qsort(i, r); }  return 0;  }  func main() -> int {  scanf("%d", &n);  var i: int;  for i in [0, n, 1] {  scanf("%d", &a[i]);  }  qsort(0, n - 1);  for i in [0, n, 1] {  printf("%d@", a[i]);  }  return 0;  } |

### 6.1.2输入与输出

## 6.2 测试用例2：矩阵乘法

### 6.2.1 测试用例2代码

|  |
| --- |
| var m\_a: int, n\_a: int;  var m\_b: int, n\_b: int;  var a: int[410], b: int[410];  var i: int, j: int, k: int;  func main() -> int {  scanf("%d%d", &m\_a, &n\_a);  for i in [0, m\_a, 1] {  for j in [0, n\_a, 1] {  scanf("%d", &a[i \* n\_a + j]);  }  }  scanf("%d%d", &m\_b, &n\_b);  for i in [0, m\_b, 1] {  for j in [0, n\_b, 1] {  scanf("%d", &b[i \* n\_b + j]);  }  }  if (n\_a != m\_b) {  printf("Incompatible Dimensions@");  return 0;  }  var tmp: int;  for i in [0, m\_a, 1] {  for j in [0, n\_b, 1] {  tmp = 0;  for k in [0, n\_a, 1] {  tmp = tmp + a[i \* n\_a + k] \* b[k \* n\_b + j];  }  printf("%10d", tmp);  }  printf("@");  }  return 0;  } |

### 6.2.2 输入与输出

## 6.3 测试用例3：选课助手

### 6.3.1 测试用例3代码

|  |
| --- |
| var name: char[1000], pre: char[10000], grade: int, credit: int;  var sum\_attempt: int, sum\_complete: int, sum\_remain: int, sum\_grade: int;  var num\_complete: int, num\_remain: int;  var names\_complete: char[10000], names\_remain: char[10000];  var pres: char[100000];  var pre\_name: char[1000];  var st\_complete: int[10000], st\_pres: int[10000], st\_remain: int[10000];  func check(id: int) -> int {  //printf("%d@", st);  if (pres[st\_pres[id]] == '\0') {  return 1;  }  //printf("%d@", st);  var tmp\_len: int, top: int, orr: int, andd: int;  var stack: int[5];  tmp\_len = 0;  top = 0;  orr = 0;  andd = 0;  var i: int, j: int, k: int;  var flag: int;  for i in [st\_pres[id], st\_pres[id+1], 1] {  var ch: char;  ch = pres[i];  if (ch == ',' || ch == ';' || ch == '\0') {  pre\_name[tmp\_len] = '\0';  top = top + 1;  stack[top] = 0;  for j in [0, num\_complete, 1] {  flag = 1;  for k in [st\_complete[j], st\_complete[j+1], 1] {  //printf("%c=?%c ", names\_complete[k], pre\_name[k - st\_complete[j]]);  if (names\_complete[k] != pre\_name[k - st\_complete[j]]) {  flag = 0;  break;  }  }  //printf("%d@", flag);  if (flag == 1) {  //printf("%d", j);  stack[top] = 1;  break;  }  }  if (andd > 0) {  if (stack[top - 1] == 1 && stack[top] == 1) {  stack[top - 1] = 1;  } else {  stack[top - 1] = 0;  }  top = top - 1;  andd = 0;  }  if (orr > 1) {  if (stack[top - 1] == 1 || stack[top] == 1) {  stack[top - 1] = 1;  } else {  stack[top - 1] = 0;  }  top = top - 1;  orr = orr - 1;  }  if (ch == ',') {  andd = andd + 1;  }  if (ch == ';') {  orr = orr + 1;  }  if (ch == '\0') {  if (orr > 0) {  if (stack[top - 1] == 1 || stack[top] == 1) {  stack[top - 1] = 1;  } else {  stack[top - 1] = 0;  }  top = top - 1;  orr = orr - 1;  }  break;  }  tmp\_len = 0;  } else {  pre\_name[tmp\_len] = ch;  tmp\_len = tmp\_len + 1;  }  }  return stack[1];  }  func main() -> int {  var i: int, j: int;  while (1 > 0) {  var len\_name: int, len\_pre: int, ch: char;  var scanfRet:int, fuyi:int;  scanfRet = scanf("%c", &ch);  fuyi = -1;  if (scanfRet == fuyi) { break; }  var flag: int;  flag = 1;  while (ch == '\n') {  scanfRet = scanf("%c", &ch);  if (scanfRet == fuyi) { flag = 0; break; }  }  if (flag == 0) {  break;  }  len\_name = 0;  while (ch != '|') {  name[len\_name] = ch;  len\_name = len\_name + 1;  scanf("%c", &ch);  }  name[len\_name] = '\0';  scanf("%d|", &credit);  len\_pre = 0;  scanf("%c", &ch);  while (ch != '|') {  pre[len\_pre] = ch;  len\_pre = len\_pre + 1;  scanf("%c", &ch);  }  pre[len\_pre] = '\0';  scanf("%c", &ch);  grade = -1;  if (ch == 'A') { grade = 4; }  if (ch == 'B') { grade = 3; }  if (ch == 'C') { grade = 2; }  if (ch == 'D') { grade = 1; }  if (ch == 'F') { grade = 0; }  if (grade >= 0) {  sum\_grade = sum\_grade + grade \* credit;  sum\_attempt = sum\_attempt + credit;  }  var st: int;  if (grade > 0) {  st = st\_complete[num\_complete];  for i in [st, st+len\_name+1, 1] {  names\_complete[i] = name[i - st];  }  sum\_complete = sum\_complete + credit;  num\_complete = num\_complete + 1;  st\_complete[num\_complete] = st + len\_name + 1;  } else {  st = st\_pres[num\_remain];  for i in [st, st+len\_pre+1, 1] {  pres[i] = pre[i - st];  }  st\_pres[num\_remain+1] = st+len\_pre+1;  st = st\_remain[num\_remain];  for i in [st, st+len\_name+1, 1] {  names\_remain[i] = name[i - st];  }  st\_remain[num\_remain+1] = st+len\_name+1;  sum\_remain = sum\_remain + credit;  num\_remain = num\_remain + 1;  }  }  var GPA: double;  if (sum\_attempt == 0) {  printf("GPA: 0.0@");  } else {  var aa: int, bb: int, cc: int;  aa = sum\_grade / sum\_attempt;  bb = (sum\_grade % sum\_attempt) \* 10 / sum\_attempt;  cc = (sum\_grade % sum\_attempt) \* 100 / sum\_attempt - bb \* 10;  //printf("%d %d %d %d %d@", aa, bb, cc, sum\_grade, sum\_attempt);  if (cc >= 5) {  bb = bb + 1;  if (bb == 10) {  aa = aa + 1;  bb = 0;  }  }  printf("GPA: %d.%d@", aa, bb);  }  printf("Hours Attempted: %d@", sum\_attempt);  printf("Hours Completed: %d@", sum\_complete);  printf("Credits Remaining: %d@", sum\_remain);  printf("@Possible Courses to Take Next@");  var num: int;  num = 0;  for i in [0, num\_remain, 1] {  if (check(i) == 1) {  num = num + 1;  printf(" ");  for j in [st\_remain[i], st\_remain[i+1]-1, 1] {  printf("%c", names\_remain[j]);  }  printf("@");  }  }  if (num == 0) {  if (num\_remain == 0) {  printf(" None - Congratulations!@");  } else {  }  }  return 0;  } |

### 6.3.2 输入与输出