

**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 4](#_Toc75531922)

[1.1 Lex 4](#_Toc75531923)

[1.2具体实现 4](#_Toc75531924)

[1.2.1 定义段 4](#_Toc75531925)

[1.2.2 词法规则段 5](#_Toc75531926)

[第二章 语法分析 9](#_Toc75531927)

[2.1 YACC 9](#_Toc75531928)

[2.2抽象语法树 9](#_Toc75531929)

[2.2.1 ASTNode类 9](#_Toc75531930)

[2.2.2 ExprNode类和StatNode类 9](#_Toc75531931)

[2.2.3 Program类 10](#_Toc75531932)

[2.2.4 ConstValue类 10](#_Toc75531933)

[2.2.5 SkyArrayType类 11](#_Toc75531934)

[2.2.6 SkyType类 11](#_Toc75531935)

[2.2.7 Identifier类 12](#_Toc75531936)

[2.2.8 VarDec类 12](#_Toc75531937)

[2.2.9 ConstDec类 13](#_Toc75531938)

[2.2.10 FuncDec类 14](#_Toc75531939)

[2.2.11 ClassBody类和ClassDec类 14](#_Toc75531940)

[2.2.12 BinaryExpr类 15](#_Toc75531941)

[2.2.13 CompoundStat类 16](#_Toc75531942)

[2.2.14 ArrayReference类和ClassRef类 16](#_Toc75531943)

[2.2.15 FuncCall类 17](#_Toc75531944)

[2.2.16 PointerNode类 18](#_Toc75531945)

[2.2.17 GlobalArea类 18](#_Toc75531946)

[2.2.18 其他类 19](#_Toc75531947)

[2.3语法分析的具体实现 19](#_Toc75531948)

[第三章 语义分析 22](#_Toc75531949)

[3.1 LLVM概述 22](#_Toc75531950)

[3.2 LLVM IR 22](#_Toc75531951)

[3.2.1 IR布局 22](#_Toc75531952)

[3.2.2 IR上下文环境 23](#_Toc75531953)

[3.3 IR生成 23](#_Toc75531954)

[3.3.1 运行环境设计 23](#_Toc75531955)

[3.3.2 类型系统 24](#_Toc75531956)

[3.3.3变量的创建与获取 25](#_Toc75531957)

[3.3.4 函数的定义与调用 26](#_Toc75531958)

[3.3.5 二元运算的实现 29](#_Toc75531959)

[3.3.6 条件控制语句的实现 31](#_Toc75531960)

[3.3.7 赋值语句的实现 37](#_Toc75531961)

[第四章 目标代码生成 38](#_Toc75531962)

[第五章 优化以及进阶主题 39](#_Toc75531963)

[5.1 一些优化 39](#_Toc75531964)

[5.2 进阶主题 39](#_Toc75531965)

[第六章 测试结果 39](#_Toc75531966)

[6.1 测试用例1：快速排序 39](#_Toc75531967)

[6.1.1 测试用例1代码 39](#_Toc75531968)

[6.1.2输入与输出 40](#_Toc75531969)

[6.2 测试用例2：矩阵乘法 41](#_Toc75531970)

[6.2.1 测试用例2代码 41](#_Toc75531971)

[6.2.2 输入与输出 42](#_Toc75531972)

[6.3 测试用例3：选课助手 42](#_Toc75531973)

[6.3.1 测试用例3代码 42](#_Toc75531974)

[6.3.2 输入与输出 46](#_Toc75531975)

# 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

LLVM IR语言自上而下主要分为：module、function、block、instruction。其中。

* Module: 每个IR文件被称为一个Module，其中包含了全局的符号信息（包括全局变量、function的声明与定义）、对其他module的依赖信息以及目标代码的环境信息等。
* Function: 函数由传入参数和若干block构成。进入函数后会顺序执行block，在block执行过程中可能会出现由block内instruction指定的跳转执行。函数是代码层面组织可执行语句的基本模块。
* Block: 由若干条instruction组成。
* Instruction：是操作的最小单元，每一个instruction是一个基本指令，包括：运算指令、逻辑指令、跳转指令、函数调用与返回执行、定义指令等。

### 3.2.2 IR上下文环境

在LLVM-IR中，主要需要使用由LLVM提供的两个上下文环境：

1. LLVM::Context
2. LLVM:IRBuilder

其中的Context主要提供上下文环境，负责提供用户定义符号如：全局变量、局部变量、函数等的储存记录以及查询。并且提供其上下文环境。

其中的IRBuilder主要是负责创建相关语句、变量、函数等，即在IR中添加相应的instruction。是构建LLVM-IR的十分重要的一个上下文环境。

## 3.3 IR生成

### 3.3.1 运行环境设计

LLVM-IR为我们提供了IR上下文环境，可以通过LLVM::Context来实现对于变量函数等变量的查询。但是为了简便以及对一些其他信息的维护，我们将LLVM::Context与LLVM::IRBuilder进行封装，然后额外维护一些信息，定义如下：

**static** LLVMContext context;

**static** IRBuilder<> builder(context);

**class** ConvertEngine{

**private**:

Module \*module;

**private**:

Function \*scan, \*print;

**private**:

Function \*main;

stack<Function\*> funcList;

vector<BasicBlock\*> breakBlock;

vector<BasicBlock\*> continueBlock;

**public**:

map<string, SkyArrayType\*> arrayMap;

**bool** flagIsReturn;

};

（类内的函数部分已被隐去，函数会在下文中被介绍）

接下来我们解释一下ConvertEngine内的成员变量的含义与用途：

* \*module: 该module是由LLVM中所定义的Module，与前文中提到的LLVM中的Module是同样的概念。
* \*scan, \*print: 语言所必要的就是对于数据的读取与输出，这两个指针分别指向输入函数和输出函数。在Sky语言中，我们数据读入函数和数据输出函数均设置为系统函数，由ConvertEngine在构造时直接自动生成。
* funcList: 在函数的递归调用时，函数的递归调用栈是十分重要的数据结构，是用以维护函数在递归时正确执行、正确返回、正确使用变量所非常重要的结构。在LLVM中没有提供的默认递归栈可供调用，所以为了维护栈上函数指针，在ConvertEngine中维护了函数递归调用栈，来保存栈上的所有指针。相应的，在进行函数递归调用以及退出时，也要操作该栈。
* breakBlock, continueBlock: 跳转逻辑是以块为单位进行的。在一个块中进行continue或者break时实质上是要进行跳转。这两个结构即保存当此时进行continue或者break指令时，实质是应该跳转到哪一个block中继续执行。
* arrayMap: 由于在Sky语言中没有指针，所以对于数组的组织是定义了一个单独的array类型，该类型是LLVM中并不默认支持的，所以在使用时实际上是将其转化为了指针，然后通过自行维护相关数组长度信息来实现的。所以对于该类变量，需要使用特殊的储存方式和获取方式。
* flagIsReturn: 该变量是一个简单的公用flag，用于解决block内跳转语句后不应有语句的问题。当开始执行Block时，该变量会被置为0,当执行到return语句或跳转语句时该变量会被置为1。当执行非return或跳转语句时发现该变量为1时，会将该变量置0并直接退出当前block。

### 3.3.2 类型系统

类型定义见抽象语树部分

相关类型的转化实现如下：

*/\**

*\* TO LLVM type : this function is used to cast our type : SkyType to LLVM type*

*\* \*/*

Type \* SkyType::toLLVMType() {

**if**(type == SKY\_VAR) {

**switch** (\*varType) {

**case** SkyVarType::SKY\_INT:

**return** builder.getInt32Ty();

**case** SkyVarType::SKY\_INT\_64:

**return** builder.getInt64Ty();

**case** SkyVarType::SKY\_CHAR:

**return** builder.getInt8Ty();

**case** SkyVarType::SKY\_FLOAT:

**return** builder.getFloatTy();

**case** SkyVarType::SKY\_DOUBLE:

**return** builder.getDoubleTy();

**case** SkyVarType::SKY\_BOOL:

**return** builder.getInt1Ty();

*// TODO: pointers*

**case** SkyVarType::SKY\_INT\_POINTER:

**return** llvm::Type::getInt32PtrTy(context);

**case** SkyVarType::SKY\_CHAR\_POINTER:

**return** llvm::Type::getInt8PtrTy(context);

**case** SkyVarType::SKY\_INT\_64\_POINTER:

**return** llvm::Type::getInt64PtrTy(context);

**case** SkyVarType::SKY\_FLOAT\_POINTER:

**return** llvm::Type::getFloatPtrTy(context);

**case** SkyVarType::SKY\_DOUBLE\_POINTER:

**return** llvm::Type::getDoublePtrTy(context);

**case** SkyVarType::SKY\_BOOL\_POINTER:

**return** llvm::Type::getInt1PtrTy(context);

}

} **else** **if** (type == SKY\_ARRAY) { *// Array type is special*

**return** ArrayType::get(arrayType->type->toLLVMType(), arrayType->size);

} **else** **return** llvm::Type::getVoidTy(context);

}

### 3.3.3变量的创建与获取

Value \*VarDec::convertToCode() {

**if** (type->type == SKY\_AUTO && expr != **nullptr**) {

type->type = expr->type;

type->varType = &(expr->varType);

}

**if** (type->type == SKY\_ARRAY) {

engine.arrayMap[id->name] = type->arrayType;

}

**auto** varType = type->toLLVMType();

Value \* ret = **nullptr**;

**if**(isGlobal()) { *// Difference is that : we should not pass initValue to it.*

ret = **new** GlobalVariable(\*engine.getModule(), varType, false, GlobalValue::ExternalLinkage, type->Create(), id->name);

} **else** {

ret = CreateEntryBlockAlloca(engine.nowFunction(), id->name, varType);

}

**auto** assign = **new** AssignStat(id, expr);

assign->convertToCode();

**delete** assign;

**return** ret;

}

变量的创建要分为两种情况：全局变量的创建、局部变量的创建。其不同之处是储存的空间不同，所以调用的LLVM接口不同。同时全局变量会进行默认初始化。

同时，在执行完变量定义后，需要执行赋值语句进行赋值。因为可能会出现定义与赋值连写的语法。

接下来我们来看一下变量的获取：

Value\* ConvertEngine::findVarByName(string varName) {

**auto** nowFunc = funcList.top();

**auto** result = nowFunc->getValueSymbolTable()->lookup(varName);

**if**(result != **nullptr**) **return** result;

result = module->getGlobalVariable(varName);

**if**(result == **nullptr**){

**throw** VarNotFound(varName + " not found");

}

**return** result;

}

该函数是是现在ConvertEngine中的变量查找函数，实际上是对LLVM提供的函数查找接口进行了封装，首先通过函数调用栈获取当前所在达函数指针，然后在当前函数内部搜索是否存在当前名称的变量。如果没有找到变量则在全局范围内进行搜索，即局部变量可以屏蔽全局变量。

获取到正确的函数了之后会返回Value\*以进行调用。

### 3.3.4 函数的定义与调用

*/\**

*\* Function declaration : this function is used to declare function*

*\* 1. construct params*

*\* 2. convert body*

*\* 3. end*

*\* \*/*

Value \* FuncDec::convertToCode() {

funcType->funcName = id->name;

**return** funcType->convertToCode();

}

Value \* SkyFuncType::convertToCode() {

vector<Type\*> args;

**if** (paraList != **nullptr**) {

**for** (**auto** &it: \*paraList) {

args.push\_back(it->type->toLLVMType());

}

}

*// func (args) -> retType {}*

**auto** funcType = FunctionType::get(retType->toLLVMType(), args, false);

**auto** func = Function::Create(funcType, GlobalValue::ExternalLinkage, funcName, engine.getModule());

engine.enterFunction(func);

BasicBlock \*funcBlock = BasicBlock::Create(context, "function begin", func, **nullptr**);

builder.SetInsertPoint(funcBlock);

*// calc params*

**auto** iterToPara = func->arg\_begin();

**if** (paraList != **nullptr**) {

**for** (**auto** &it: \*paraList) {

**auto** mem = CreateEntryBlockAlloca(func, it->id->name, it->type->toLLVMType());

builder.CreateStore(iterToPara++, mem);

}

}

Value \* ret = **nullptr**;

**if** ( retType->type != SkyTypes::SKY\_VOID ){

ret = CreateEntryBlockAlloca(func, funcName, retType->toLLVMType());

}

body->convertToCode();

engine.exitFunction();

*// Maintain the function stack*

**if** (engine.funcStackSize()) {

**auto** nowFunc = engine.nowFunction();

builder.SetInsertPoint(&(nowFunc->getBasicBlockList().back()));

}

**return** func;

函数的定义需要对函数的具体参数进行具体的定义，即函数的定义需要处理：

1. 函数参数的定义（包括传入参数和返回参数）
2. 函数的实例：在声明过程中，需要通过LLVM接口创建Function实例。
3. 函数内的基本块：block是必要的，为该函数创建block并且设定为函数入口点。
4. 对于函数具体语句内容的实现，调用AST树中子节点的convert函数来实现
5. 对于函数递归调用栈的维护，需要调用ConvertEngine中的相关函数进行函数递归调用栈的维护。

接下来我们来看一看如何进行函数的调用：

*/\**

*\* Function Call: this function is used to call function*

*\* \*/*

Value \*FuncCall::convertToCode() {

**auto** func = engine.getModule()->getFunction(id->name);

**if** (func == **nullptr**) { *// If this function is not implement*

**throw** FuncNotFound(\*(**new** string(id->name)) + " not found");

}

*// System function should have its own call function, because it needs some special settings.*

**if** (strcmp(id->name, "printf") == 0 || strcmp(id->name, "scanf") == 0) {

**return** callSysIO();

}

*// Get all params*

vector<Value\*> inputArgs;

**auto** funcNeed = func->arg\_begin();

**if** (args != **nullptr**) {

**for** (**auto** &it : \*args) {

**if** (funcNeed->hasNonNullAttr()) {

**auto** \*addr = engine.findVarByName(**dynamic\_cast**<Identifier \*>(it)->name);

inputArgs.push\_back(addr);

} **else** {

inputArgs.push\_back(it->convertToCode());

}

funcNeed++;

}

}

*// Create Call sentences*

Value \*ret = builder.CreateCall(func, inputArgs, "callFunc");

**return** ret;

}

函数调用主要分为以下几部分：

1. 搜索函数名，查看是否定义/实现
2. 获取到对应的函数后，将传入参数与函数所需求的参数按顺序匹配并传入。此处使用到了ConvertEngine中对变量的查找。
3. 调用函数

### 3.3.5 二元运算的实现

*/\**

*\* Binary operator function:*

*\* Support :*

*\* add, sub, mul, div, equ, neq, LT, GT, LE, GE, and, or, xor, sl, rl*

*\* Note: all params of binary operator need to be the same type*

*\* \*/*

Value \*calcOp(Value\* left, Value\* right, BinaryOperators op) {

**auto** type1 = left->getType();

**auto** type2 = right->getType();

*// check the float flag*

**bool** floatFlag = type1->isFloatTy() || type2->isFloatTy();

**bool** doubleFlag = type1->isDoubleTy() || type2->isDoubleTy();

**switch** (op) {

**case** OP\_PLUS:

**if** (floatFlag || doubleFlag) {

**return** builder.CreateFAdd(left, right, "addFloat");

} **else** {

**return** builder.CreateAdd(left, right, "addInt");

}

**case** OP\_MINUS:

**if** (floatFlag || doubleFlag) {

**return** builder.CreateFSub(left, right, "subFloat");

} **else** {

**return** builder.CreateSub(left, right, "subInt");

}

**case** OP\_MUL:

**if** (floatFlag || doubleFlag) {

**return** builder.CreateFMul(left, right, "mulFloat");

} **else** {

**return** builder.CreateMul(left, right, "mulInt");

}

**case** OP\_DIV:

**if** (floatFlag || doubleFlag) {

**return** builder.CreateFDiv(left, right, "divSigned");

} **else** {

**return** builder.CreateSDiv(left, right, "divFloat");

}

**case** OP\_EQ:

**return** builder.CreateICmpEQ(left, right, "equal");

**case** OP\_NE:

**return** builder.CreateICmpNE(left, right, "neq");

**case** OP\_GT:

**return** builder.CreateICmpSGT(left, right, "gt");

**case** OP\_LT:

**return** builder.CreateICmpSLT(left, right, "lt");

**case** OP\_GE:

**return** builder.CreateICmpSGE(left, right, "ge");

**case** OP\_LE:

**return** builder.CreateICmpSLE(left, right, "le");

**case** OP\_AND:

**return** builder.CreateAnd(left, right, "and");

**case** OP\_OR:

**return** builder.CreateOr(left, right, "or");

**case** OP\_XOR:

**return** builder.CreateXor(left, right, "xor");

**case** OP\_MOD:

**return** builder.CreateSRem(left, right, "mod");

**case** OP\_LEFT:

**return** builder.CreateShl(left, right, "shl");

**case** OP\_RIGHT:

**return** builder.CreateAShr(left, right, "shr");

**case** OP\_PTR:

**return** **nullptr**;

}

}

该模块主要是通过将我们所定义的运算与LLVM中默认提供的运算进行一一匹配。匹配之后直接通过IRBuilder来创建相关的语句。

### 3.3.6 条件控制语句的实现

#### 3.3.6.1 if语句的实现

*/\**

*\* If Status:*

*\* if (condition) {*

*\* -then-*

*\* } else {*

*\* -else-*

*\* }*

*\* -common-*

*\* condition -> then/else -> common*

*\* Note: LLVM do not permit any code after br or ret in the same basic block,*

*\* so we must check this condition.*

*\* \*/*

Value \*IfStat::convertToCode() {

Value \*condValue = condExpr->convertToCode();

condValue = builder.CreateICmpNE(condValue, ConstantInt::get(Type::getInt1Ty(context), 0, true), "if");

**auto** func = engine.nowFunction();

*// Create 3 basic blocks to contain codes*

**auto** thenCond = BasicBlock::Create(context, "thenCond", func);

**auto** elseCond = BasicBlock::Create(context, "elseCond", func);

**auto** common = BasicBlock::Create(context, "common", func);

*// then*

**auto** branch = builder.CreateCondBr(condValue, thenCond, elseCond);

builder.SetInsertPoint(thenCond);

engine.flagIsReturn = false;

thenStat->convertToCode();

**if** (!engine.flagIsReturn) {*// Do not create br, is jump is in body*

builder.CreateBr(common);

}

engine.flagIsReturn = false;

thenCond = builder.GetInsertBlock();

*// else*

builder.SetInsertPoint(elseCond);

engine.flagIsReturn = false;

**if** (elseStat != **nullptr**) {

elseStat->convertToCode();

}

**if** (!engine.flagIsReturn) { *// Do not create br, is jump is in body*

builder.CreateBr(common);

}

engine.flagIsReturn = false;

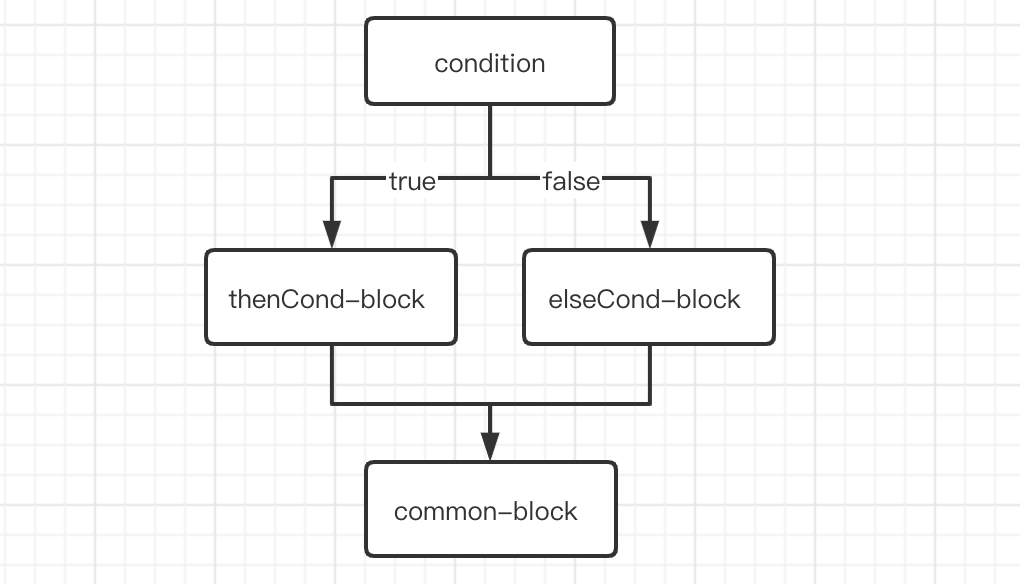
elseCond = builder.GetInsertBlock();

builder.SetInsertPoint(common);

**return** branch;

}

为了实现If的语句控制，我们需要定义3个block来进行处理。其执行关系如下：



其中的common-block是为了使两种不同的分支最后都合并到同一个block中所定义的空block，只起到控制作用。剩余的thenCond-block与elseCond-block分别与if语句的两个代码块部分相对应。

#### 3.3.6.2 for 循环语句的实现

*/\**

*\* for Status:*

*\* for condition {*

*\* -loop-*

*\* }*

*\* -breakBlock-*

*\* condition -> loop/breakkBlock, loop->condition*

*\* Note: LLVM do not permit any code after br or ret in the same basic block,*

*\* so we must check this condition.*

*\* \*/*

Value \*ForStat::convertToCode() {

**auto** func = engine.nowFunction();

Value \* startValue = start->convertToCode();

Value \* endValue = end->convertToCode();

Value \* stepValue = step->convertToCode();

Value \* varValue = engine.findVarByName(forVar->name);

builder.CreateStore(startValue, varValue);

*// Create 3 basic blocks to contain codes*

BasicBlock \*condition = BasicBlock::Create(context, "condition", func);

BasicBlock \*loop = BasicBlock::Create(context, "loopCode", func);

BasicBlock \*breakLoop = BasicBlock::Create(context, "breakLoop", func);

engine.enterLoop(breakLoop, condition);

*// condition*

builder.CreateBr(condition);

builder.SetInsertPoint(condition);

**auto** nowValue = forVar->convertToCode();

**auto** condValue = builder.CreateICmpSLT(nowValue, endValue);

condValue = builder.CreateICmpNE(condValue, ConstantInt::get(Type::getInt1Ty(context), 0, true));

**auto** branch = builder.CreateCondBr(condValue, loop, breakLoop);

condition = builder.GetInsertBlock();

*// loop*

builder.SetInsertPoint(loop);

engine.flagIsReturn = false;

body->convertToCode();

Value \* newVarValue = builder.CreateAdd(nowValue, stepValue); *// add step*

builder.CreateStore(newVarValue, varValue);

**if** (!engine.flagIsReturn) {

builder.CreateBr(condition);

}

engine.flagIsReturn = false;

loop = builder.GetInsertBlock();

builder.SetInsertPoint(breakLoop);

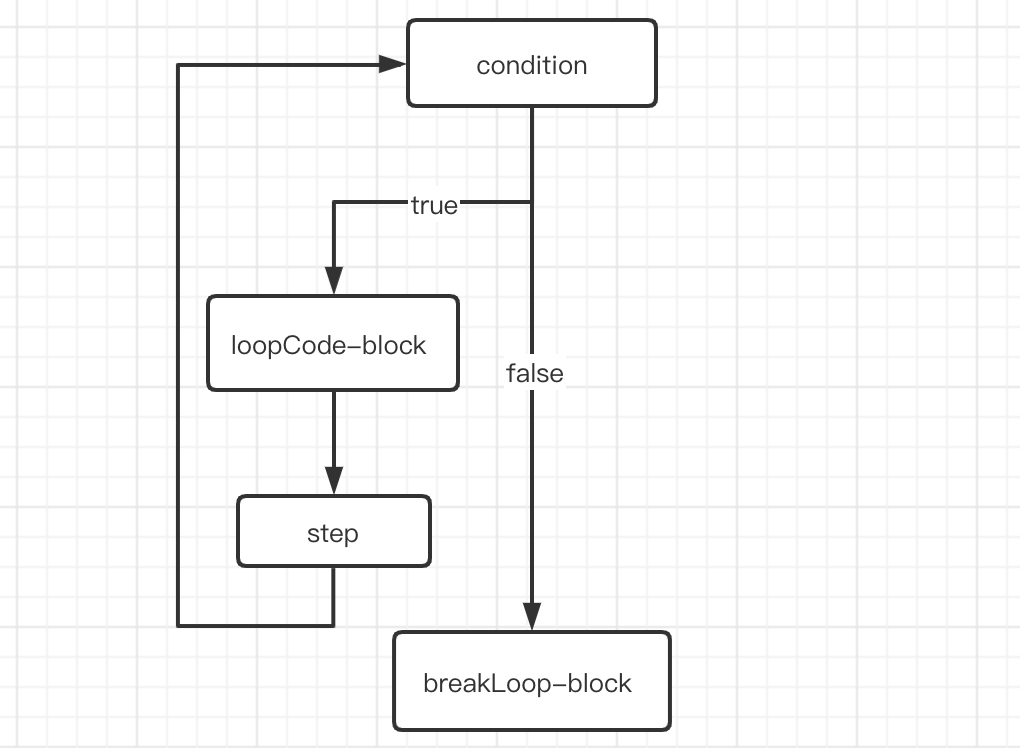
engine.exitLoop();

**return** branch;

}

For 语句实际上需要4个Block，但是为了更加简单，我们将下图中的step block直接添加在loopCode-block尾部来实现循环变量的修改。

For 循环的breakLoop-block会被储存到convertEngine中的相关数据结构中用以进行break语句的处理。Continue语句的处理就比较简单了，可以直接跳转到condition中来实现。



#### 3.3.6.3 while语句的实现

*/\**

*\* While Status:*

*\* while condition {*

*\* -loop-*

*\* }*

*\* -breakBlock-*

*\* condition -> loop/breakkBlock, loop->condition*

*\* Note: LLVM do not permit any code after br or ret in the same basic block,*

*\* so we must check this condition.*

*\* \*/*

Value \*WhileStat::convertToCode() {

**auto** func = engine.nowFunction();

BasicBlock \* condition = BasicBlock::Create(context, "condition", func);

BasicBlock \* loop = BasicBlock::Create(context, "loop", func);

BasicBlock \* breakLoop = BasicBlock::Create(context, "breakLoop", func);

engine.enterLoop(breakLoop, condition);

*// condition*

builder.CreateBr(condition);

builder.SetInsertPoint(condition);

**auto** condValue = cond->convertToCode();

condValue = builder.CreateICmpNE(condValue, ConstantInt::get(Type::getInt1Ty(context), 0, true));

**auto** branch = builder.CreateCondBr(condValue, loop, breakLoop);

condition = builder.GetInsertBlock();

*// loop*

builder.SetInsertPoint(loop);

engine.flagIsReturn = false;

body->convertToCode();

**if** (!engine.flagIsReturn) {

builder.CreateBr(condition);

}

engine.flagIsReturn = false;

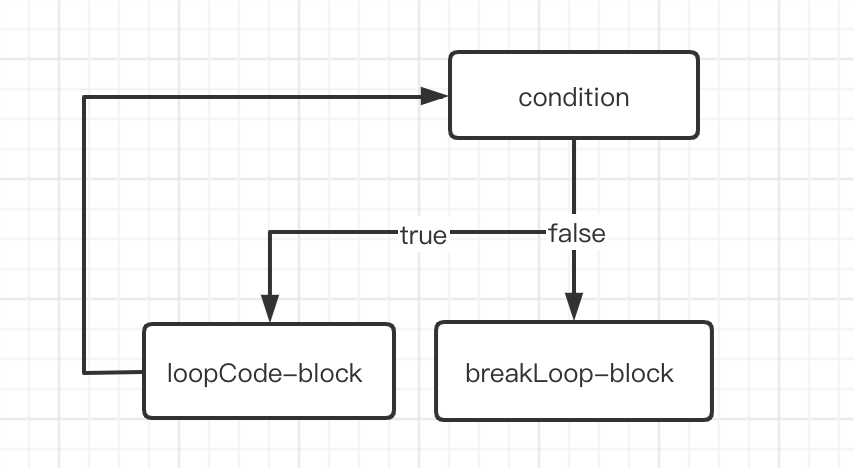
builder.SetInsertPoint(breakLoop);

engine.exitLoop();

**return** branch;

}

While语句的实现与for语句的实现是及其相似的，区别就在于while语句中不再处理step，所以该部分实际上删去了for循环的step部分。



#### 3.3.6.4 其他控制语句

*/\**

*\* Jump Status: Control codes*

*\* - break : exit now loop*

*\* - continue : directly enter the condition part in current loop*

*\* - return : exit the function and return value*

*\* Note : No code is allowed to be write after JumpStat, so we used a flag to pass a signal.*

*\* \*/*

Value \*JumpStat::convertToCode() {

engine.flagIsReturn = true;

**switch** ( type ) {

**case** BREAK:

**return** builder.CreateBr(engine.getCurBreakBlock());

**case** CONTINUE:

**return** builder.CreateBr(engine.getCurContinueBlock());

**case** RETURN:

**if** (retExpr != **nullptr**) {

**return** builder.CreateRet(retExpr->convertToCode());

} **else** {

**return** builder.CreateRetVoid();

}

}

**return** **nullptr**;

}

我们的语言支持其他三种控制指令：break, continue, return

其中return指令由于有LLVM的直接支持，所以可以直接调用IRBuilder来实现。剩余的两种控制指令则使用ConvertEngine中所记录的当前环境的break跳转点和continue跳转点，通过创建Br来进行实现。

### 3.3.7 赋值语句的实现

*/\**

*\* AssignStart : This function is used to load value of right to left identifier (arrayReference)*

*\* \*/*

Value \*AssignStat::convertToCode() {

Value \*res = **nullptr**;

**switch** (type) {

**case** ID\_ASSIGN: *// Identifier = value\**

**return** builder.CreateStore(expr->convertToCode(), engine.findVarByName(id->name));

**case** ARRAY\_ASSIGN: *// ArratReference = value\**

**return** builder.CreateStore(expr->convertToCode(), arrayRef->getValueI());

**case** LAMBDA\_ASSIGN: *// identifier = lambda*

**int** funcID = engine.pushNewFunction();

**dynamic\_cast**<SkyFuncType \*>(expr)->funcName = engine.getFuncNameByID(funcID);

expr->convertToCode();

**return** builder.CreateStore(builder.getInt32(funcID), engine.findVarByName(id->name));

}

**return** **nullptr**;

}

直接通过LLVM中提供的Store接口来进行变量的赋值。

# 第四章 目标代码生成

由于我们使用了LLVM框架，所以可以直接调用LLVM的接口来输出LLVM-IR代码。

**void** ConvertEngine::compileToFile(string fileName) {

error\_code ErrInfo;

raw\_ostream \*out = **new** raw\_fd\_ostream(fileName, ErrInfo, sys::fs::CreationDisposition());

module->print(\*out , **nullptr**);

}

上面的函数就可以直接输出LLVM-IR语言的代码，但是这并不是可以直接运行的代码。所以为了生成目标代码，我们进行调用了llc来将LLVM-IR语言编译成asm文件，然后调用平台本地的gcc编译器编译汇编代码，最终就可以得到可以直接在目标平台上执行的代码。

我们将这个过程直接写在了main函数中，用户可以直接调用该编译器来将一份sky语言代码直接编译成可执行文件，编译过程主函数如下：

**int** main(**int** argc, **char**\*\* argv){

**if** (argc >= 2) {

yyin = fopen(argv[1], "r");

} **else** {

puts("Invalid param.");

**return** 0;

}

yyparse();

**if** (root == **nullptr**) **return** -1;

root->convertToCode();

**if** (argc != 3) {

engine.compileToFile("compileOut.ir");

system("llc compileOut.ir");

system("gcc compileOut.ir.s -o compileOut");

} **else** {

string fileName = string(argv[2]);

engine.compileToFile(fileName+".ir");

system(("llc " + fileName+".ir").c\_str());

system(("gcc " + fileName + ".ir.s" + " -o " + fileName).c\_str());

}

**return** 0;

}

# 第五章 优化以及进阶主题

## 5.1 一些优化

我们的编译器是比较简洁的编译器，所以具有很快的编译速度，可以快速对项目程序进行编译。

除了本身的简洁之外，我们还对一些细节进行了优化：

1. 尽量减少跳转语句的出现：一种更加简单的方法是将for循环的step部分作为单独的block来出现，但是这样无疑会凭空多出一次跳转指令性导致程序运行效率降低。所以我们将step部分直接附加在主循环体的尾部。
2. 强类型的要求。为了防止类型自动转化导致的用户使用错误的类型而导致的隐秘的难以调试的错误，我们在进行表达式计算时要求所有参与运算的数据必须是同一类型，如果类型不匹配需要进行手动转化。

## 5.2 进阶主题

我们实现了一个不是很进阶的主题：类型推断

类型推断是指在进行变量定义是可以不显式指定变量的类型只进行初始化赋值。当编译器发现被定义变量没有声明类型时，会去通过初始化的数值的类型来自动推断该变量的类型。

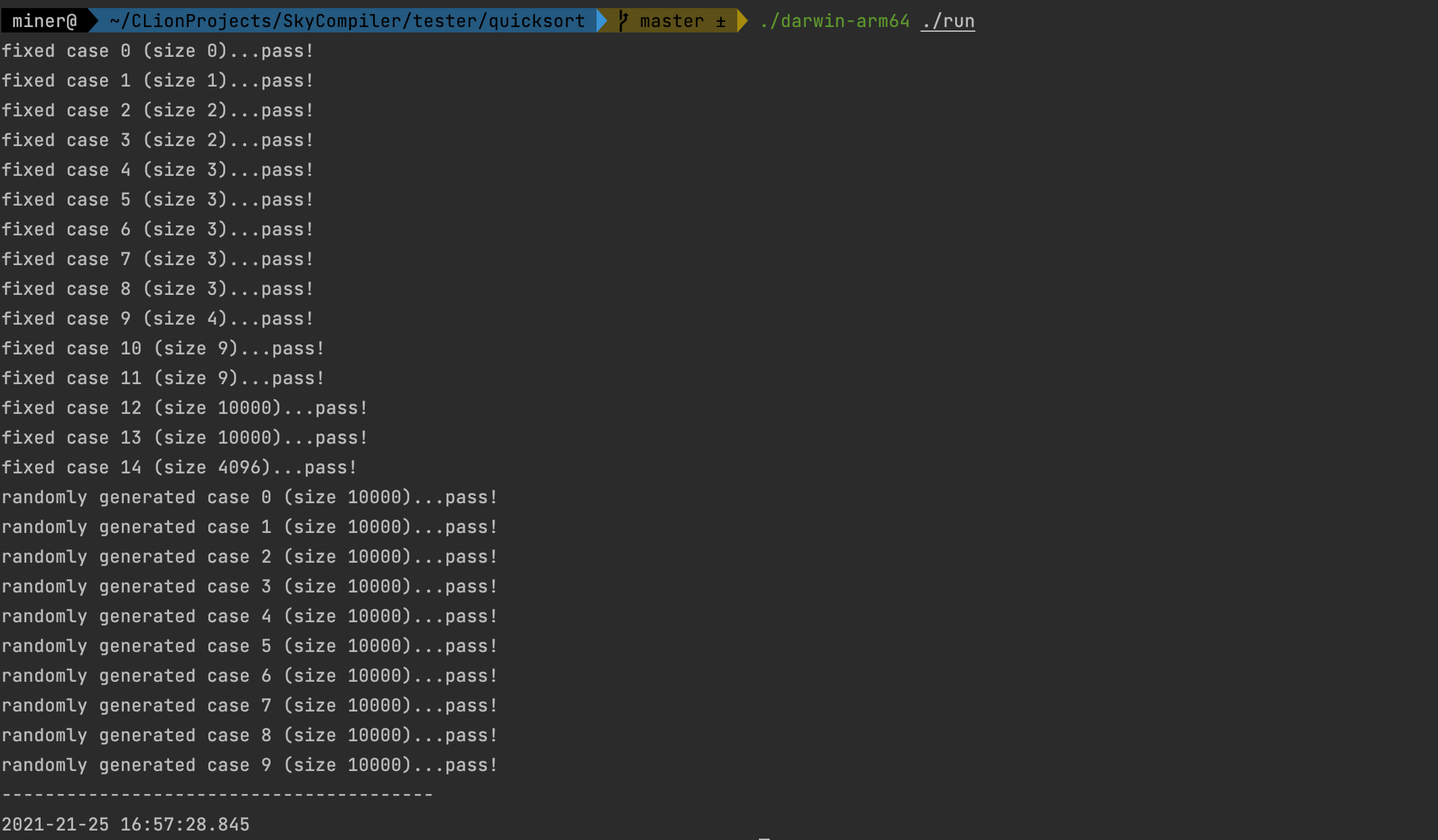
# 第六章 测试结果

## 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

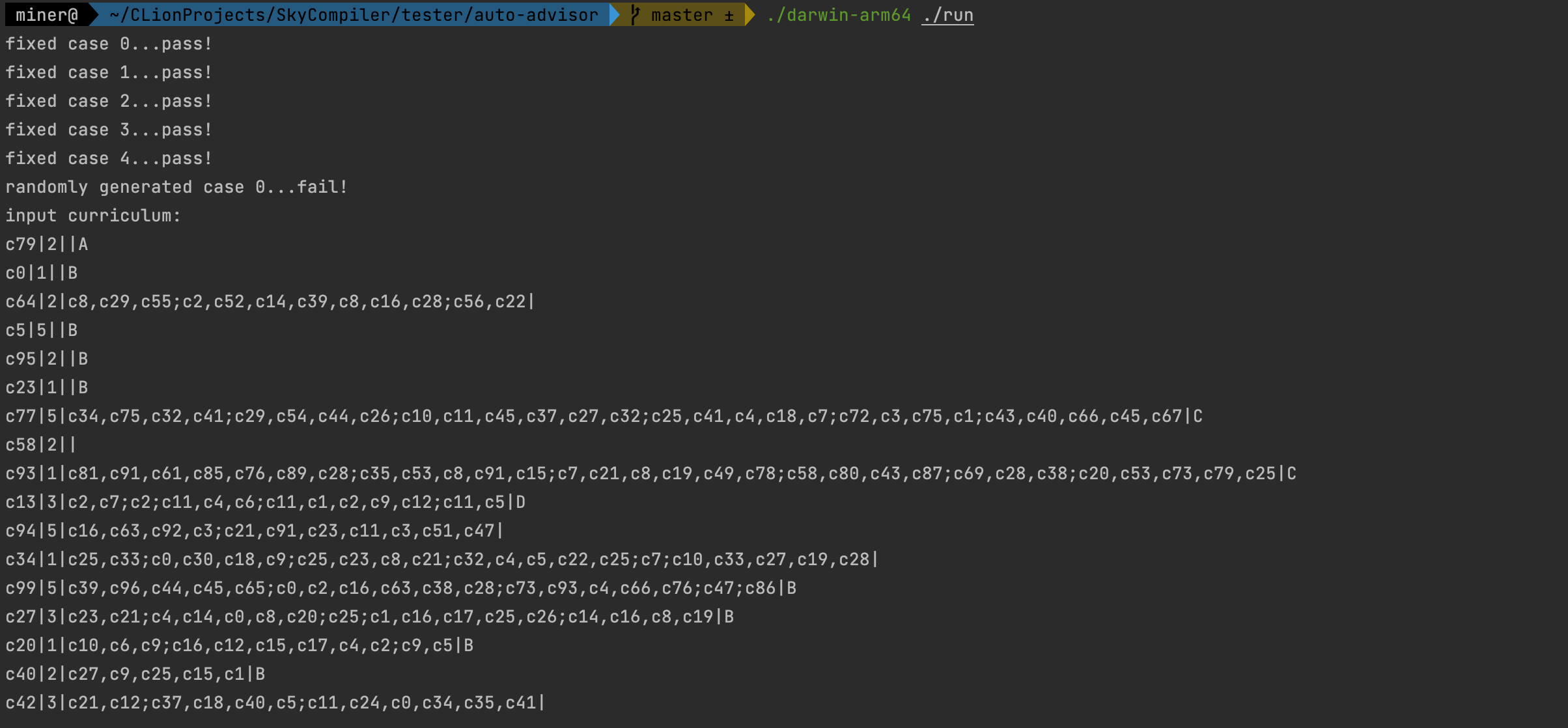


## 6.3 测试用例3：选课助手

### 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



注：因为助教提示该道题目的随机测试数据有问题，所以我们认为通过了前面5个case就是通过了这道题目。