图标

描述已自动生成

**编译原理实验报告**

|  |  |  |  |
| --- | --- | --- | --- |
| **专业** | **：** | **计算机** |  |
| **班级** | **：** | **CS1902** |  |
| **学号** | **：** | **I201920024** |  |
| **姓名** | **：** | **木林** |  |
| **电话** | **：** | **+85578449298** |  |
| **邮箱** | **：** | **792967028@qq.com** |  |

**独创性声明**

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签名：

日期：2022年 7 月 3 日

|  |  |
| --- | --- |
| 成绩 |  |
| 教师签名 |  |

目 录

[1 概述 1](#_Toc107942031)

[1.1 实验目的 1](#_Toc107942032)

[1.2 实验任务 1](#_Toc107942033)

[1.3 实验要求 1](#_Toc107942034)

[2 编译工具链的使用 3](#_Toc107942035)

[2.1 实验任务 3](#_Toc107942036)

[2.2 实验实现 3](#_Toc107942037)

[2.3 实验代码 3](#_Toc107942038)

[3 词法分析器的设计与实现 4](#_Toc107942039)

[3.1 SysY语言词法分析器的设计 4](#_Toc107942040)

[3.2 SysY语言词法分析器的实现 4](#_Toc107942041)

[3.3 实验代码 6](#_Toc107942042)

[4 语法分析器的设计与实现 9](#_Toc107942043)

[4.1 SysY语言语法分析器的设计 9](#_Toc107942044)

[4.2 SysY语言语法分析器的实现 9](#_Toc107942045)

[4.3 实验代码 10](#_Toc107942046)

[5 符号表的设计与实现 16](#_Toc107942047)

[5.1 符号表的设计 16](#_Toc107942048)

[5.2 符号表管理的实现 16](#_Toc107942049)

[6 一个L-翻译模式的递归下降翻译程序的实现 18](#_Toc107942050)

[6.1 L-翻译模式自顶向下语义计算 18](#_Toc107942051)

[6.2 一个L-翻译模式递归下降翻译程序的实现 18](#_Toc107942052)

[7 PL/0编译系统分析与修改 20](#_Toc107942053)

[7.1 PL/0编译系统分析 20](#_Toc107942054)

[7.2 PL/0编译系统的修改 20](#_Toc107942055)

[8 总结与展望 24](#_Toc107942056)

[8.1 实验总结 24](#_Toc107942057)

[8.2 实验感想 24](#_Toc107942058)

[8.3 实验展望 24](#_Toc107942059)

[参考文献 26](#_Toc107942060)

# 概述

## 实验目的

《编译原理实验》是配合《编译原理》课程理论教学独立开设的实验课程。目的在于通过一系列专门设计的实验，完成一个高级语言-SysY(C语言的一个子集)的定义和编译。熟悉业界经典的工具软件，从零开始完成一个完整的编译器，将该语言源程序编译为ARM汇编语言程序。

通过实验培养大型工程项目的设计与管理能力，提升系统软件实践能力。

## 实验任务

实验任务包括一个完整的编译器的构造过程，和三个独立的实验。完整的编译器构造过程即构造SysY语言的编译器，又细分为：词法分析，语法分析，符号表管理，语义分析，中间代码生成，代码优化和目标代码生成等阶段。三个独立的实验包括：

1. 编译工具链的使用；
2. 一个L-翻译模式的递归下降翻译程序的实现；
3. PL/0编译器的分析与修改。

## 实验要求

以上实验任务并不需要全部完成，可根据情况选做其中的部分实验(必做的实验除外)。

实验一（编译工具链的使用），实验二（词法分析），实验三（语法分析）的第１关（即语法正确性检查）是必做部分，第２关（抽象语法树的构造和输出）的６个测试用例除用例５（该用例含for语句）以外，原则上应能基本正确地输出其它５个用例的抽象语法树（输出形式不限）。实验三(语法分析)、实验七(中间代码生成)、实验九(代码优化)、实验十(功能评测)等四个实验的得分以老师最后打分为准，其它实验的成绩直接采纳平台评测的分数。

# 编译工具链的使用

## 实验任务

用交叉编译器arm-linux-gnueabihf-gcc将源程序“翻译”成arm汇编代码，再将汇编代码汇编并与Sysy2022运行时库连接，生成arm可执行代码，然后用qemu-arm虚拟机运行arm可执行程序。

## 实验实现

用arm-linux-gnueabihf-gcc 将iplusf.c编译成arm汇编代码iplusf.arm.s再次用arm-linux-gnueabihf-gcc 汇编iplusf.arm.s，同时连接SysY2022的运行时库sylib.a，生成arm的可执行代码iplusf.arm用qemu-arm运行iplusf.arm

## 实验代码

### 第一关

#!/bin/bash

gcc -D BILIBILI def-test.c -c alibaba.c

gcc -o def-test def-test.o alibaba.o

### 第二关

 #!/bin/bash

clang -S -O2 -target armv7-linux-gnueabihf bar.c -o bar.clang.arm.s

### 第三关

#!/bin/bash

arm-linux-gnueabihf-gcc -S iplusf.c -o iplusf.arm.s

arm-linux-gnueabihf-gcc iplusf.arm.s sylib.a -o iplusf.arm

qemu-arm -L /usr/arm-linux-gnueabihf/ ./iplusf.arm

# 词法分析器的设计与实现

## SysY语言词法分析器的设计

基于简单高效的原则，以及对词法和句法分析算法的扎实理解，推荐使用两种工具（Flex 和 Bison）来构建词法和句法分析程序。 如果语法正确，将生成抽象语法树。 在其中根据语言的词法规则和Flex要求的格式编辑Lex.l文件（文件名可以自己定义，但扩展名必须是.l）。 我有。 用Flex编译后，可以得到原版的Lex.yy parser.c，其中通过调用yylex()进行词法分析，每当识别出一个词，（词型码，词本身含义）就输出这个词或者 传递给解析器；

根据语言的语法规则，按照Bison要求的格式，编辑Parser.y文件，使用 Bison 编译后可以获得语法分析源程序 Parser.tab.c，调用 parser() 进行语法分析。

两者结合完成词法分析时，必须统一词的类型码。 此时，每个单词都可以通过百分号作为语法规则的终止符，在parser.y中一个一个地列出来。 同时，在 Bison 编译之后，构建了一个文件 Parser.tab.h，其中这些 ID 用作枚举常量，每个（类）词一个。 这些枚举常量被发送到 Lex.l，每个对应一个单词的类型代码； 同时，这些枚举常量中的每一个都与 Parser.y 中的单词类型代码相关。与终结器的比较 图 2-1 描绘了一个工作流程图，显示了 FLEX 和 Bison 的组合使用来开发词法和解析器。

## SysY语言词法分析器的实现

词法分析器构建技术电路的初始选择是开发一个能够准确表示所有类型单词的正则表达式。 将正则表达式描述的词法规则等价转化为相应的确定性和最小化有限自动机FA，然后构建相关的词法分析软件。

实验中的词法分析器可以利用GNU Flex词法生成器自动生成工具，需要以正则表达式（regular expressions）的形式提供词法规则。 Flex 根据上述技术路线创建指定词法规则的词法规则。 分析软件。 这个想法是创建可以正确检测各种单词的正则表达式。

高级语言词法分析器必须检测五类词：关键字（保留词）、运算符、分隔符、常量和标识符。 这里给出每个单词的类型码和匹配的符号描述，根据mini-c语言的规范如图3.1：

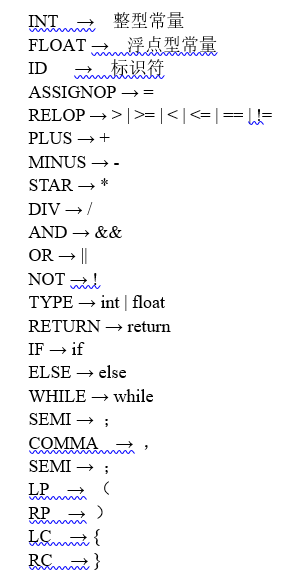


图3.1mini-c语言的规范

第一部分是定义部分，范围从 % 到 %，主要包括 C 语言宏定义，如文件包含、宏名称定义、变量和类型定义和声明。 将立即传送到词法分析器源程序 lex.yy.c 除了百分比到百分比之外，本节还定义了将在以下规则部分中使用的各种常规宏名称。

第二个组成部分是规则，一个规则包括：基于正则表达式的操作表示词法分析器检测到与正则表达式匹配的单词后，进行动作相关操作，返回单词的类型码。

当词法分析器检测到一个单词时，相关字符串保存在 yytext 中，其长度为 yyleng。

第三部分是用户子程序部分，它将被完整地转移到词法分析器源程序 lex.yy.c 中。

## 实验代码

%option noyywrap

%option yylineno

%{

#include <stdio.h>

enum Token {

    INT=258, FLOAT, VOID, CONST, RETURN, IF, ELSE, FOR, WHILE, DO, BREAK, CONTINUE,

    LP, RP, LB, RB, LC, RC, COMMA, SEMICOLON, QUESTION, COLON,

    MINUS, NOT, TILDE, ASSIGN, ADD, MUL, DIV, MOD, AND, OR,

    EQ, NE, LT, LE, GT, GE,

    ID, INT\_LIT, FLOAT\_LIT,

    LEX\_ERR

};

%}

d [0-9]

c [A-Z\_a-z]

w [\t\n ]

hex (\+|\-)?0(X|x)[0-9a-fA-F]+

oct (\+|\-)?0[0-7]+

dec (\+|\-)?[1-9]+{d}\*|0

%%

"//".\* {}

"/\*"([^\\*]|(\\*)\*[^\\*/])\*(\\*)\*"\*/"  {}

{w}+ {}

"int" {printf("%s : INT\n", yytext); return INT; }

"float" {printf("%s : FLOAT\n", yytext); return FLOAT; }

"void" {printf("%s : VOID\n", yytext); return VOID; }

"const" {printf("%s : CONST\n", yytext); return CONST; }

"return" {printf("%s : RETURN\n", yytext); ; return RETURN; }

"if" {printf("%s : IF\n", yytext); return IF; }

"else" {printf("%s : ELSE\n", yytext); return ELSE; }

"for" {printf("%s : FOR\n", yytext); return FOR; }

"while" {printf("%s : WHILE\n", yytext); return WHILE; }

"do" {printf("%s : DO\n", yytext); return DO; }

"break" {printf("%s : BREAK\n", yytext); return BREAK; }

"continue" {printf("%s : CONTINUE\n", yytext); return CONTINUE; }

"(" {printf("%s : LP\n", yytext); return LP; }

")" {printf("%s : RP\n", yytext); return RP; }

"{" {printf("%s : LC\n", yytext); return LC; }

"}" {printf("%s : RC\n", yytext); return RC; }

"[" {printf("%s : LB\n", yytext); }

"]" {printf("%s : RB\n", yytext); }

"," {printf("%s : COMMA\n", yytext); }

";" {printf("%s : SEMICOLON\n", yytext); }

"?" {printf("%s : QUESTION\n", yytext); }

":" {printf("%s : COLON\n", yytext); }

"!" {printf("%s : NOT\n", yytext); }

"=" {printf("%s : ASSIGN\n", yytext); }

"-" {printf("%s : MINUS\n", yytext); }

"+" {printf("%s : ADD\n", yytext); }

"\*" {printf("%s : MUL\n", yytext); }

"/" {printf("%s : DIV\n", yytext); }

"%" {printf("%s : MOD\n", yytext); }

"&&" {printf("%s : AND\n", yytext); }

"||" {printf("%s : OR\n", yytext); }

"==" {printf("%s : EQ\n", yytext); }

"!=" {printf("%s : NE\n", yytext); }

"<" {printf("%s : LT\n", yytext); }

"<=" {printf("%s : LE\n", yytext); }

">" {printf("%s : GT\n", yytext); }

">=" {printf("%s : GE\n", yytext); }

{c}+{d}\*{c}\*|"\_"{c}\*{d}\*{c}\* {printf("%s : ID\n", yytext); return ID; }

0[8-9]+{d}\* {printf("Lexical error - line %d : %s\n", yylineno, yytext); }

{hex}|{oct}|{dec} {printf("%s : INT\_LIT\n", yytext); return INT\_LIT; }

{d}\*f {printf("Lexical error - line %d : %s\n", yylineno, yytext); }

({d}+"."{d}\*|"."{d}+|{d}+e"-"{d}+)f? {printf("%s : FLOAT\_LIT\n", yytext); }

{d}+{c}+{d}\*{c}\* {printf("Lexical error - line %d : %s\n", yylineno, yytext); }

. {}

%%

int main(int argc, char\*\* argv) {

    if(argc > 1) {

        if(!(yyin = fopen(argv[1], "r"))) {

           perror(argv[1]);

           return  1;

        }

        while(yylex());

        fclose(yyin);

    } else printf("No source file given!\n");

    return 0;

}

# 语法分析器的设计与实现

## SysY语言语法分析器的设计

句法分析步骤中的一项关键任务是构造要构建的程序的抽象句法树 AST。 AST 与派生树不同。 删除了一些更改的单词部分，并简明地描述了程序的句法结构。 然后可以遍历抽象语法树以进行后续语义分析和中间代码创建。

例如，考虑以下语句：while (a>=1) a=a-1; 派生树和抽象语法树在图 2-3 中分别描述为左树和右树。

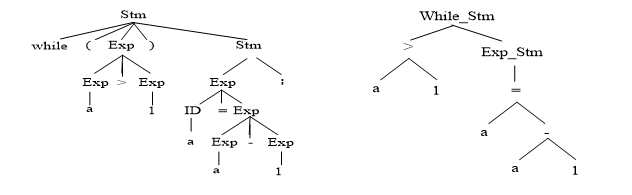


图4.1派生树和抽象语法树

## SysY语言语法分析器的实现

在mini-c语法中，有ID表示的标识符、INT表示的整数常量、IF表示关键字if、WHILE表示关键字while等终结符号。 非终结符包括ExtDefList，代表外部定义列表，CompSt，代表复合语句，等等。

当 Flex 和 Bison 一起使用时，parser.y 如何使用 lex.l 中标识的单词要做的就是在parser.y中把所有终结符（词）的类别码标识符放在 %token后面，例如：% token ID, INT, IF, ELSE, WHILE。

描述了非终结符的属性值类型。 如果必须对非终结符号进行语义计算，就会牵涉到非终结符号的属性值类型。 该类型派生自 (1) 中的联合成员，可以使用以下格式： % type<union>。 例如，在 parser.y 中：

%type <ptr> program ExtDefList

这表示非终结符 ExtDefList 属性值的类型对应于联合的成员 ptr 的类型，在这种情况下对应于指向树节点的指针。

优先级和关联性的定义 在翻译Bison文件和获取解析程序的源程序时经常会出现错误，其中大部分是shift/reduce, reduce/reduce的冲突类的错误。

## 实验代码

**ast.c文件：**

struct node \*mknode(int kind, struct node \*first, struct node \*second, struct node \*third, int pos, struct node \*forth)

{

  struct node \*T = (struct node \*)malloc(sizeof(struct node));

  T->kind = kind;

  T->ptr[0] = first;

  T->ptr[1] = second;

  T->ptr[2] = third;

  T->ptr[3] = forth;

  T->pos = pos;

  return T;

}

void display(struct node \*T, int indent)

{

  int i = 1;

  struct node \*T0;

  if (T)

  {

    switch (T->kind)

    {

    case ARRAY:

      printf("%\*c数组定义: \n", indent, ' ');

      printf("%\*cID:  %s\n", indent, ' ', T->type\_id);

      display(T->ptr[0], indent + 3);

      break;

    case ARRAY\_ARGS\_LIST:

      display(T->ptr[0], indent);

      display(T->ptr[1], indent);

      break;

    case ARRAY\_ARGS:

      printf("%\*c数组维度: \n", indent, ' ');

      display(T->ptr[0], indent + 3);

      break;

    case EXT\_DEF\_LIST:

      printf("%\*cCompUnit", indent, ' ');

      display(T->ptr[0], indent);

      display(T->ptr[1], indent);

      break;

    case EXT\_VAR\_DEF:

      printf("%\*c外部变量定义: \n", indent, ' ');

      display(T->ptr[0], indent + 3);

      printf("%\*c变量名: \n", indent + 3, ' ');

      display(T->ptr[1], indent + 6);

      break;

    case VOID:

      printf("%\*c类型:  %s\n", indent, ' ', T->type\_id);

      break;

    case TYPE:

      printf("%\*c类型:  %s\n", indent, ' ', T->type\_id);

      break;

    case EXT\_DEC\_LIST:

      display(T->ptr[0], indent);

      display(T->ptr[1], indent);

      break;

    case FUNC\_DEF:

      printf("%\*cCompUnit", indent, ' ');

      printf("%\*c`--FuncDef", indent, ' ');

      printf("%\*c函数定义: \n", indent, ' ');

      display(T->ptr[0], indent + 3);

      display(T->ptr[1], indent + 3);

      display(T->ptr[2], indent + 3);

      break;

    case FUNC\_DEC:

      printf("%\*c函数名: %s\n", indent, ' ', T->type\_id);

      if (T->ptr[0])

      {

        printf("%\*c函数形参: \n", indent, ' ');

        display(T->ptr[0], indent + 3);

      }

      else

        printf("%\*c无参函数\n", indent + 3, ' ');

      break;

    case PARAM\_LIST:

      display(T->ptr[0], indent); ，

      display(T->ptr[1], indent);

      break;

    case PARAM\_DEC:

      printf("%\*c类型: %s, 参数名: %s\n", indent, ' ',

             T->ptr[0]->type == INT ? "int" : "float", T->ptr[1]->type\_id);

      break;

    case EXP\_STMT:

      printf("%\*c表达式语句: \n", indent, ' ');

      display(T->ptr[0], indent + 3);

      break;

    case RETURN:

      printf("%\*c返回语句: \n", indent, ' ');

      display(T->ptr[0], indent + 3);

      break;

    case BREAK:

      printf("%\*c语句: \n", indent, ' ');

      break;

    case CONTINUE:

      printf("%\*c语句: \n", indent, ' ');

      break;

    case COMP\_STM:

      printf("%\*c复合语句: \n", indent, ' ');

      printf("%\*c复合语句的变量定义: \n", indent + 3, ' ');

      display(T->ptr[0], indent + 6);

      printf("%\*c复合语句的语句部分: \n", indent + 3, ' ');

      display(T->ptr[1], indent + 6);

      break;

    case STM\_LIST:

      display(T->ptr[0], indent);

      display(T->ptr[1], indent);

      break;

    case WHILE:

      printf("%\*c循环语句: \n", indent, ' ');

      printf("%\*c循环条件: \n", indent + 3, ' ');

      display(T->ptr[0], indent + 6);

      printf("%\*c循环体: \n", indent + 3, ' ');

      display(T->ptr[1], indent + 6);

      break;

    case FOR:

      printf("%\*c循环语句: \n", indent, ' ');

      printf("%\*c循环条件: \n", indent + 3, ' ');

      display(T->ptr[0], indent + 6);

      display(T->ptr[1], indent + 6);

      display(T->ptr[2], indent + 6);

      printf("%\*c循环体: \n", indent + 3, ' ');

      display(T->ptr[3], indent + 6);

      break;

    case IF\_THEN:

      printf("%\*c条件语句(IF\_THEN): \n", indent, ' ');

      printf("%\*c条件: \n", indent + 3, ' ');

      display(T->ptr[0], indent + 6);

      printf("%\*cIF子句: \n", indent + 3, ' ');

      display(T->ptr[1], indent + 6);

      break;

    case IF\_THEN\_ELSE:

      printf("%\*c条件语句(IF\_THEN\_ELSE): \n", indent, ' ');

      printf("%\*c条件: \n", indent + 3, ' ');

      display(T->ptr[0], indent + 6);

      printf("%\*cIF子句: \n", indent + 3, ' ');

      display(T->ptr[1], indent + 6);

      printf("%\*cELSE子句: \n", indent + 3, ' ');

      display(T->ptr[2], indent + 6);

      break;

    case DEF\_LIST:

      display(T->ptr[0], indent);

      display(T->ptr[1], indent);

      break;

    case VAR\_DEF:

      printf("%\*cLOCAL VAR\_NAME: \n", indent, ' ');

      display(T->ptr[0], indent + 3);

      display(T->ptr[1], indent + 3);

      break;

    case DEC\_LIST:

      printf("%\*cVAR\_NAME: \n", indent, ' ');

      T0 = T;

      while (T0)

      {

        if (T0->ptr[0]->kind == ID)

          printf("%\*c %s\n", indent + 3, ' ', T0->ptr[0]->type\_id);

        else if (T0->ptr[0]->kind == ASSIGNOP)

        {

          printf("%\*c %s ASSIGNOP\n ", indent + 3, ' ', T0->ptr[0]->ptr[0]->type\_id);

          display(T0->ptr[0]->ptr[1], indent + strlen(T0->ptr[0]->ptr[0]->type\_id) + 4);

        }

        T0 = T0->ptr[1];

      }

      break;

    case ID:

      printf("%\*cID:  %s\n", indent, ' ', T->type\_id);

      break;

    case INT:

      printf("%\*cINT: %d\n", indent, ' ', T->type\_int);

      break;

    case FLOAT:

      printf("%\*cFLAOT: %f\n", indent, ' ', T->type\_float);

      break;

    case ASSIGNOP:

    case AND:

    case OR:

    case RELOP:

    case PLUS:

    case MINUS:

    case STAR:

    case DIV:

    case MOD:

      printf("%\*c%s\n", indent, ' ', T->type\_id);

      display(T->ptr[0], indent + 3);

      display(T->ptr[1], indent + 3);

      break;

    case NOT:

    case UMINUS:

      printf("%\*c%s\n", indent, ' ', T->type\_id);

      display(T->ptr[0], indent + 3);

      break;

    case FUNC\_CALL:

      printf("%\*c函数调用: \n", indent, ' ');

      printf("%\*c函数名: %s\n", indent + 3, ' ', T->type\_id);

      display(T->ptr[0], indent + 3);

      break;

    case ARGS:

      i = 1;

      while (T)

      {

        struct node \*T0 = T->ptr[0];

        printf("%\*c第%d个实际参数表达式: \n", indent, ' ', i++);

        display(T0, indent + 3);

        T = T->ptr[1];

      }

      printf("\n");

      break;}}}

# 符号表的设计与实现

## 符号表的设计

符号表的管理是语义分析的关键组成部分。 编译器在整个编译过程中利用符号表来捕获源程序中各种名称的特征。 术语“名称”是指以下项目：程序名称、过程名称、函数名称、用户定义类型名称、变量名称、常量名称、枚举值名称、标签名称等。 维度（如果语言启用数组）、函数参数的数量、常量值和目标地址（存储单元偏移地址）等都是“特征信息”的示例。

符号表可以使用多种数据结构构建，实验可以采用多种数据结构：

(1) 序列表。 这就是本实验指南中符号表的管理方式。 符号表 symbolTable 目前是一个顺序堆栈。 堆栈顶部的索引在开始时的值为 0。 当一个符号被填充时，新的符号被填充到栈顶位置，栈顶指针加1。

(2) 哈希表

(3) 十字链表

(4) 每次进入一个作用域产生一个表，每次离开一个作用域释放一个表，形成多表结构。

符号表操作包括构建符号表、插入表项、查询表项、更新表项、删除表项、释放符号表空间等。

## 符号表管理的实现

(1) 遇到复合语句节点COM STM时，先给LEV加1，表示准备进入新的范围。 使用堆栈符号作用域TX记录符号中的作用域变量，以处理此作用域内的变量。 表中最顶端的位置，即开始位置 symbolTable 堆栈符号范围TX包含符号表symbolTable的堆栈索引。

(2) 当你想在符号表中注册一个新符号时，首先从上到下检查符号表，看层号为LEV的符号是否与现在要注册的符号同名，如果有 , 会重复。 定义不正确； 否则，新符号将在符号数据库中注册，层号为 LEV。

3）在遍历复合语句的节点COM STM的子树并准备返回其父节点后，复合语句的语义分析完成，复合语句的变量必须从符号表中擦除。 该方法首先出栈，提取复合语句作用域的起点，然后根据该值调整symbolTable.index，完成符号表的符号删除过程。

4）符号表查找操作，在AST遍历过程中，在求值各种表达式，遇到变量访问的同时，在symbolTable中，从栈顶到栈底，检查是否有相同的符号定义，如果 都是一样的，如果查询后没有发现符号，则未定义； 如果在符号表的多个位置都定义了同一个符号，那么从搜索方向上可以看出，它遵循最接近优先级的概念。 如果发现该符号，则进行额外的语义分析。

# 一个L-翻译模式的递归下降翻译程序的实现

## L-翻译模式自顶向下语义计算

L-翻译模式自顶向下语义计算的方法:

如果对于每个产生式A->X1X2……Xn，其每个语义规则中的每个属性或者是综合属性或者是Xj(1<=j<=n)的一个继承属性，而这个继承属性仅依赖于：

（1）产生式Xj的左边符号X1X2……Xj-1的属性

（2）A的继承属性

## 一个L-翻译模式递归下降翻译程序的实现

**Parses.c文件**

#include "common.h"

b\_info parseS(){

    b\_info S,R;

    int B;

    if(lookahead=='0'){}

    else if(lookahead=='1'){

            B = parseB();

            R = parseR(0,0);

            S.val = (B << R.len) + R.val;

            S.len = R.len + 1;

    }

    else{

        S.val = 0;

        S.len = 0;

    }

    return S;

}

**Parser.c文件**

#include "common.h"

#include <math.h>

b\_info parseR(int v, int l){

   b\_info R,R1;

    int B;

    switch(lookahead) {

        case '.':

            R.val = 0;

            R.len = 0;

            break;

        case '\n':

            matchToken('\n');

            R.val = 0;

            R.len = 0;

            break;

        case '0':

        case '1':

            B = parseB();

            R1 = parseR(v,l + 1);

            R.val = (B << R1.len )+ R1.val;

            R.len = R1.len + 1;

            break;

    }

    return R;

}

# PL/0编译系统分析与修改

## PL/0编译系统分析

PL/0 编译器很小（大约 5400 行 C++ 和大约 400 行语法source) 主是因为它编译的输入语言非常小。 PL/0 定义为Niklaus Wirth（Pascal 和Modula 的发明者）作为他的扩展示例的一部分1976 年的书，算法 + 数据结构 = 语言Pascal 的子集，具有嵌套过程，但没有参数； 功能; 案例，为，或重复陈述； 其他条款； 或除整数以外的任何类型。 参考版本PL/0 编译器完全按照 Wirth 定义的语言实现语言，有四个扩展：注释、原始数字 I/O、常量字符串的输出和值/结果参数。

## PL/0编译系统的修改

int main(int argc,char\*argv[])

{

bool nxtlev[symnum];

删除本行 //printf("Input pl/0 file? ");

strcpy(fname,argv[1]); //修改scanf("%s", fname);成

删除本行 //scanf("%s", fname); /\* 输入源程序文件名 \*/

fin = fopen(fname, "r"); // 注意: 在获得文件句柄后，后续程序复用fname作了其它用途，

if (fin)

{

删除本行 // printf("List object code?(Y/N)"); /\* 是否输出虚拟机代码 \*/

删除本行 // scanf("%s", fname);

listswitch = false; //修改listswitch = (fname[0]=='y' || fname[0]=='Y');成

删除本行 // printf("List symbol table?(Y/N)"); /\* 是否输出名字表 \*/

删除本行 // scanf("%s", fname);

tableswitch = false; //修改tableswitch = (fname[0]=='y' || fname[0]=='Y');成

fa1 = fopen("fa1.tmp", "w");

删除本行 // fprintf(fa1,"Input pl/0 file? ");

删除本行 // fprintf(fa1,"%s\n",fname);

init(); /\* 初始化 \*/

err = 0;

cc = cx = ll = 0;

ch = ' ';

if(-1 != getsym())

{

fa = fopen("fa.tmp", "w");

fas = fopen("fas.tmp", "w");

addset(nxtlev, declbegsys, statbegsys, symnum);

nxtlev[period] = true;

if(-1 == block(0, 0, nxtlev)) /\* 调用编译程序 \*/

{

fclose(fa);

fclose(fa1);

fclose(fas);

fclose(fin);

printf("\n");

return 0;

}

fclose(fa);

fclose(fa1);

fclose(fas);

if (sym != period)

{

error(9);

}

if (err == 0)

{

fa2 = fopen("fa2.tmp", "w"); //记录执行结果,请保留，评测程序会检查执行结果

/\* 一次性输出虚拟机代码 \*/

listallcode();

interpret(); /\* 虚拟机解释执行中间代码 \*/

fclose(fa2);

}

else

{

printf("Errors in pl/0 program");

}

}

fclose(fin);

}

else

{

printf("Can't open file!\n");

}

printf("\n");

return 0;

}

void listallcode()

{

for (int i=0; i<cx; i++)

{

printf("%d %s %d %d\n", i, mnemonic[code[i].f], code[i].l, code[i].a);

fprintf(fa2,"%d %s %d %d\n", i, mnemonic[code[i].f], code[i].l, code[i].a);

}

}

# 总结与展望

## 实验总结

编译原理实验是大学毕业三年来最具挑战性的实验。 我为这个实验工作了很多日日夜夜。 整个实验花了很长时间，我最终失败了。 使用编译原理的试用已经结束。

通过完成这四个编译原理的试用项目，我最终完成了一个 Vector 编译器。 虽然还有很多我的编译器没有涵盖的功能，以及一定的局限性和不足，但是通过这次实验，我对编译有了更好的把握。 由于我对编译器工作流程的掌握，我对编译器词法分析、语法分析、语义分析、中间代码生成和最终目标代码生成的理解和掌握程度有所提高。

## 实验感想

我做实验过中遇到了和了解有些困难如下：

1.使用flex工具创建词法分析器，然后使用正则表达式规则对程序进行词法分析。

2、在Bison中创建语法分析器，在老师提供的框架中加入新的语法规则，如for循环、break、continue语句、数组识别等，以实现对源码的语法分析。

3.改进实验导课展示功能，实现抽象语法树遍历。

4. 使用实验指导课创建符号表，然后完成符号表管理。

5. 基于以上信息，构建静态语义检查，可以检测代码中总共16个静态语义错误，如类型不匹配、break和continue语句中的位置错误、需要左值的表达式。

6、执行中间代码的创建，以及基于创建的中间代码的基本块的划分和简单的代码优化。

## 实验展望

这个实验让我深信编译原理实验是一个难点。 编译原理实验不仅巩固了课堂上描述的课程内容，也让我进一步了解了设计编程语言时应该注意的问题。 还有很多无关的材料，比如寄存器分配技术。 这个实验让我不仅巩固了自己的学术知识，还发现了很多课外信息。 而且这次编译原理实验也考验了我的编码能力，暴露了我的编码效率低，推理不严谨，考虑的不够广泛，暴露了我的很多缺陷。

# 参考文献

格式举例(没有一篇与编译技术有关，仅用于示范格式，请替换成实际参考的文献)：

[1] 闫明礼，张东刚. CFG桩复合地基技术及工程实践（第二版）. 北京：中国水利水电出版社，2006

[2] M. Chalfie, S. R. Kain. Green fluorescent protein: properties, applications, and protocols. Hoboken, New Jersey: Wiley-interscience, 1998

[3] 詹向红，李德新. 中医药防治阿尔茨海默病实验研究述要. 中华中医药学刊，2004, 22(11): 2094-2096

[4] E. S. Lein, M. J. Hawrylycz, N. Ao, M. Ayres, A. Bensinger, A. Bernard, et al. Genome-wide atlas of gene expression in the adult mouse brain. Nature, 2007, 445(7124): 168-176

[5] M. L. Bouxsein, S. K. Boyd, B. A. Christiansen, R. E. Guldberg, K. J. Jepsen, R. Müller. Guidelines for assessment of bone microstructure in rodents using micro–computed tomography. Journal of Bone and Mineral Research, 2010, 25(7): 1468-1486

[6] S. Yang, J. Liu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau. Principled Schedulability Analysis for Distributed Storage Systems using Thread Architecture Models. In: Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018), Carlsbad, CA, USA, October 8-10, 2018. USENIX Association, 2018: 161-176

[7] T. Yao, J. Wan, P. Huang, X. He, F. Wu, C. Xie. Building efficient key-value stores via a lightweight compaction tree. ACM Transactions on Storage, 2017, 13(4): 1-28