华中科技大学

编译原理实验报告

专　　业：　　计算机科学与技术

班　　级：

学　　号：

姓　　名：

电　　话：

邮　　箱：

目 录

1 概述

1.1 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。

1.2 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。

1.3 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。

根据自己实际完成情况，据实撰写实验报告。

2 编译工具链的使用

2.1 实验任务

了解编译过程，掌握常见编译器的使用，具体包括 gcc 编译器、clang 编译器、arm-linux-gnueabihf-gcc 交叉编译器和 qemu-arm 虚拟机等编译工具的使用。了解并熟悉上述编译器的编译选项和可选命名参数，具体实现将源语言代码编译生成所需的目标代码。

2.2 实验实现

（1）GCC 编译器的使用

使用 gcc 编译器连编 def-test.c 和 alibaba.c，生成 def-test 二进制可执行代码。且使用宏定义参数-DBILIBILI 定义宏 BILIBILI，完成任务要求。具体命令为：

```
gcc -o def-test -DBILIBILI def-test.c alibaba.c
```

-o 指定输出文件，-DBLIBILI 进行宏定义。

（2）CLANG 编译器的使用

使用 clang 编译器将 bar.c 翻译成优化级别 O2 的 armv7 架构-linux 系统-符合 gnueabihf 嵌入式二进制接口规则并支持 arm 硬浮点的汇编代码 bar.clang.arm.s。具体命令为：

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

-S 表明完成编译但不执行汇编，产生汇编文件；-O2 设置优化级别为 O2；-target armv7-linux-gnueabihf 进行架构指定。

（3）交叉编译器 arm-linux-gnueabihf-gcc 和 qemu-arm 虚拟机的使用

使用 arm-linux-gnueabihf-gcc 交叉编译器，将 iplusf.c 编译成 arm 汇编代码 iplusf.arm.s，并设置具体 arm 型号为 arm7，具体命令为：

```
arm-linux-gnueabihf-gcc -S -o iplusf.arm.s iplusf.c -march=armv7
```

使用 arm-linux-gnueabihf-gcc 交叉编译器，将 iplusf.arm.s 连接 sylib.a 生成 arm 的可执行代码 iplusf.arm，设置 O2 优先级别。具体命令为：

```
arm-linux-gnueabihf-gcc -O2 -o iplusf.arm iplusf.arm.s sylib.a
```

将上述生成的 iplusf.arm 可执行代码在 qemu-arm 虚拟机上执行，库路径为 /usr/arm-linux-gnueabihf/，具体命令为：

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

使用上述方法的好处在于多数服务器为 X86 的 Linux 服务器，这使得在搭建编译器中的比较环节，如目标 arm 汇编代码的比较，十分麻烦。但使用交叉编译器与虚拟机则可以有效解决 arm 服务器资源有限的问题，从而方便后续搭建编译器正确性的判断。

3 词法分析器的设计与实现

3.1 SysY 语言词法分析器的设计

为了完成 Sys Y 语言的词法分析器，需要掌握词法分析辅助工具 flex 的使用，以及 Sys Y 语言的词法规则。

flex 语法分成 3 部分，分别为辅助定义部分、规则部分和用户子程序段部分。辅助定义部分进行选项设置，C 语言代码的头文件与全局定义，以及词法正则式的辅助定义。规则部分进行正则式的编写。用户子程序段进行辅助函数的定义以及相关测试代码的编写。

（1）辅助定义部分设计

设置 yylineno 可选项，自动管理行号。设置 noyywrap 选项，由于测试是单文件扫描，故无需设置 yywarp。接着在 C 代码体，即%{<代码体>%}，中设置头文件与 TOKEN 单词种类码，可以采用 enum 类型进行管理。接着进行正则式定义，在该处定义的正则式可被规则部分引用，方便正则式生成。

为方便后续 Token 识别，在此处定义正则式用以识别标识符、int 整型字面量、float 浮点型字面量、单行注释、多行注释、非法标识符、非法数值。

（2）规则部分设计

在辅助定义部分定义单词种类码的基础上，当识别到某个单词时，返回对应的 TOKEN 值。在此次词法测试中，还需输出识别的单词及其 TOKEN 值。flex 提供 yytext 变量保存识别的单词字符串。

值得注意的是，对于标识符、int 整型字面量、float 浮点型字面量、单行注释、多行注释、非法标识符、非法数值的识别可直接引用辅助定义部分的正则式。另外，对标识符的识别应该放置在关键字识别之后。

（3）用户子程序部分设计

在此部分进行词法分析测试，使用 flex 提供的 yyin 存放待测试文件的指针，并循环调用 yylex()进行词法分析。

3.2 SysY 语言词法分析器的实现

根据上述 3 部分设计思路，完成对应的具体实现。

（1）辅助定义部分的具体实现

辅助定义部分结构为：

```
%option noyywrap
%option yylineno
%{
C 头文件、单词种类码定义
%}
正则式定义
```

单词种类码采用 enum 类型定义，将所有识别代词别名定义在此。具体实现为 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};，设置初 TOKEN 值为 258。

定义标识符，标识符的首字符必须为字母或‘_’，且后续只能出现字母、数字或‘_’。具体定义为：

```
ID    [a-zA-Z_][a-zA-Z0-9_]*
```

定义 int 型字面值，可以为十进制数、八进制数或十六进制数，其中十进制数超过个位数的不能以 0 开头，八进制数以 0 开头且后续出现的数字不得大于 7，十六进制数以 0x 或 0X 开头且后续值得出现 a-fA-F 以及 0-9 范围内的字符。具体定义为：

```
INT_LIT [1-9][0-9]*|0[0-7]*|(0x|0X)[0-9a-fA-F]+
```

定义 float 型字面值，举例一些具有代表性的字面值。带小数点 2.0 或.04；带科学计数 4e-04，且 float 值后都可带 f 或 F。由此定义 EXP 辅助正则式 e 或 E 后可跟正负号，后跟数字。具体定义为：

```
EXP [Ee][+-]?[0-9]+
FLOAT_LIT ([0-9]*\.[0-9]+|[0-9]+\.){EXP}?(f|F)?|[0-9]+{EXP}(f|F)?
```

定义单行注释和多行注释。单行注释在识别到“//”后一直识别到换行符为

止。多行注释则识别到“/*”一直到识别“*/”为止。具体定义为：

```
SingleLineComment     "//".*$
MultilineComment      "/*"(.|\n)*"*/"
```

定义非法标识符和非法数值。当以数字开头时，若后续产生字母或“_”时，识别为非法标识符，同时可以识别非法 float 型值，如 2f 等，以及一些特别的十六进制数 0xz 等。对于非法 int 型值则主要判别非法八进制，如 089 等。具体定义为：

```
Invalid ([0-9]+[a-zA-z]+[a-zA-Z0-9]*)
Invalid_int (0[0-7]*(8|9)[0-9]*)
```

（2）规则部分的具体实现

对于该部分实现，则可分成单关键字和用户自定义单词的识别，对于单关键字则直接进行识别“关键字”，例如识别 int 关键字，具体实现为：

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

其中{}内为指定动作，yytext 存放识别单词，返回对应的 TOKEN 值。

而对于用户自定义单词，如标识符，可以引用辅助部分的相关定义，使用引用方式进行识别，具体实现为：

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

而对于错误单词识别，则可使用上述定义的非法标识符识别与数值识别。具体实现为：

```
{Invalid} {printf("Lexical error - line %d : %s\n",yylineno,yytext); return LEX_ERR;}
{Invalid_int} {printf("Lexical error - line %d : %s\n",yylineno,yytext); return LEX_ERR;}
```

特别的，ID 等自定义符号的识别需要放置在关键字识别之后进行，以免识别出错。对于单独出现的空格符、换行符、制表符等进行跳过，而对于进行上述所有识别判断后仍无法识别的则同样输出错误。

（3）用户子程序部分的具体实现

该部分主要实现 yyin 的指针获取，具体为 yyin = fopen(argv[1], "r");。以及对词法分析的循环调用，具体为 while(yylex());。

完成上述 3 部分实现后，各部分间通过%%分隔。

4 语法分析器的设计与实现

4.1 SysY 语言语法分析器的设计

在词法分析的基础上，使用 bison 设计语法分析器，识别语法错误，并在正确的情况下生成抽象语法树 AST。对于语法分析器的设计可以分为两部分，语法检查与 AST 的生成。

（1）语法检查设计

语法检查的关键则是进行语法规则的构建，这里使用 bison 辅助语法分析器的构建，bison 的语法规则使用 L-翻译模式，故在自底向上语法语法分析的同时生成抽象语法树。在用户自定义函数部分，调用 yyparse()进行语法分析。

首先进行辅助定义设计，使用%union 进行类型别名定义，方便后续查阅代码。接着使用%type <ptr>定义非终结符语义值类型，在此为节点类型，因为后续进行抽象语法树搭建时，采用的是对非终结符进行 mknode 节点构建。%token <type_int> INT 和%token <type_float> FLOAT 分别识别 int 类型数值和 float 类型数值，%token <type_id> ID RELOP TYPE VOID 识别标识符或类型关键字以及模块化比较符。%token 后直接进行词法分析 TOKEN 值的定义，从而无需在词法分析器内重复定义。

接着进行优先级的定义来避免可能存在的冲突，%left 表明该操作符左结合，%right 表明该操作符右结合，对于上述声明有以下关系：先声明的优先级更低，后声明的优先级更高。具体定义为：

```
%left ASSIGNOP
%left OR
%left AND
%left RELOP
%left PLUS MINUS
%left STAR DIV
%right UMINUS NOT
```

除此之外，还需对移进规约冲突进行相关说明。使用%noassoc 声明上述冲突错误，该语法内存在 if-then 与 if-then-else 的移进规约冲突，具体声明如下：

```
%nonassoc LOWER_THEN_ELSE
%nonassoc ELSE
```

该声明表示 LOWER_THEN_ELSE 的优先级低于 ELSE，具体在特定语法规则内使用%prec LOWER_THEN_ELSE 将该优先级覆盖到此语法规则内从而解决冲突。对于 Sys Y 语言而言，则在单 IF-THEN 时进行优先级说明，从而在后续识别到 ELSE 后，由于 LOWER_THEN_ELSE 的优先级更低，从而进行移进，解决了移进-规约冲突。

随后进行语法规则的定义。定义树的首节点为 CompUnit，当语法分析规约于此时，表明该程序无语法错误，可以进行抽象语法树的打印。其下层语法规则为外部定义列表，列表内含一个外部定义。外部定义包括外部变量定义、带返回值的函数定义、不带返回值的函数定义即错误定义。

外部变量定义中含变量类型、外部变量列表和 SEMI 分号，变量类型为 INT 或 FLOAT，而外部变量列表允许采用<变量>，<变量>的方式连续定义，变量可以为单标识符，也可以为数组。

而对于函数定义，则含返回类型、函数声明和函数体，返回类型为 INT、FLOAT 或 VOID，函数声明采用函数名(<形参类型><形参 ID>，...)的方式定义，而函数体进行‘{’‘函数内容项’‘}’的分析。

可以进行变量定义列表与语句列表的交替识别。变量定义列表包含一个或多个变量定义，对于变量定义识别，与外部变量识别大体无异，区别在于节点类型不一，为了后续抽象语法树遍历时区分节点。语句列表包含一个或多个语句，而语句识别，则包含表达式语句 Exp 识别，语句块 Compst 识别、Return 语句识别、IF-Then 和 IF-THEN-Else 语句识别、WHILE 语句识别和 FOR 语句识别。表达式语句则包含操作符表达式、数值识别和标识符识别（含数组）。

最后进行 yyerror 的自定义。此次任务中需要自定义 yyerror 函数，输出语法错误行号与列号，可以使用 bison 管理的行列号进行输出。当产生语法错误时，会自动调用 yyerror 函数，输出我们所需的错误信息。自定义语法错误信息为：

```
fprintf(stderr, "%s:%d Grammar Error at Line %d Column %d: ",filename,yylloc.first_line,yylloc.first_line,yylloc.first_column);
```

（2）抽象语法树的设计

采用语法制导的抽象语法树生成，在上述提到非终结符为 ptr 类型，即节点类型，在进行语法分析时可自底向上进行节点的连接从而生成语法树，节点连接函数为 mknode，返回一个节点指针，将综合节点建立为该节点的孩子，并为该节点添加部分属性，如行号信息等。

当语法分析到 CompUnit 时，表明程序语法无误，调用抽象语法树输出函数 display，进行语法树深度优先遍历输出语法树。display 函数识别传入节点的节点类型标志 kind，进行对应的文字的输出，同时为了美观，可以通过参数传递缩进值，方便测试查看。

4.2 SysY 语言语法分析器的实现

对于上述设计中提到两部分的主要内容，进行具体实现的举例说明：

（1）for 语句

在语句识别中，包含 for 语句识别。对于 for 语句包含 4 个部分，单词表达式、条件表达式、末尾循环体和 for 语句循环体，其中可将前 3 个部分识别为循环条件 ForList，循环体则识别为 Stmt，即可能为语句块，或是单语句。将这 2 个节点作为 for 语句节点的孩子。ForList 则将 3 部分作为自己的孩子。具体实现为：

```
Stmt: FOR LP ForList RP Stmt {$$=mknode(FOR,$3,$5,NULL,yylineno);}
      | …
      ;
ForList:    Exp SEMI Exp SEMI Exp {$$=mknode(FOR_LIST,$1,$3,$5,yylineno);}
      ;
```

在进行抽象语法树生成时，遍历到 FOR 节点时，打印提示信息，分别进行循环条件和循环体的遍历，display(T->ptr[0],indent+6);与 display(T->ptr[1],indent+6);。

（2）数组定义

数组包含标识符 ID、数组维度列表。定义数组节点标记为 ARRY_DEC，数组维度列表为 ArryList 节点。标识符 ID 直接存入 ARRY_DEC 节点内。ArryList 为该节点孩子。而 ArryList 节点以‘[’ Exp ‘]’ ArryList 右递归分析。将 Exp 和 ArryList 作为孩子节点。具体实现为：

```
ArryDec:    ID ArryList {$$=mknode(ARRY_DEC,$2,NULL,NULL,yylineno);
              strcpy($$->type_id,$1);}
            ;
ArryList:   LB Exp RB ArryList {$$=mknode(ARRY_LIST,$2,$4,NULL,yylineno);}
            | LB Exp RB          {$$=mknode(ARRY_LIST,$2,NULL,NULL,yylineno);}
            ;
```

进行 AST 输出时，遍历到 ARRAY_DEC 节点，输出该节点存储的标识符，继续遍历 ArryList 孩子。在 ARRY_LIST 首节点中进行 while 循环遍历并输出各数组维度信息。

（3）if-then 语句

该语句的识别包括 2 个部分，即条件表达式 Exp 和语句内容 Stmt。将 Exp 和 Stmt 作为 IF_THEN 节点的孩子。特别的，在设计中提出使用定义优先级方式解决移进-规约冲突在此体现，添加%prec LOWER_THEN_ELSE 在语法后，当后续出现 ELSE 时，由于上述定义 LOWER_THEN_ELSE 的优先级低于 ELSE，故进行移进操作。具体实现为：

```
Stmt :    IF LP Exp RP Stmt %prec LOWER_THEN_ELSE
            {$$=mknode(IF_THEN,$3,$5,NULL,yylineno);}
          | ...
          ;
```

遍历到 IF_THEN 节点时，输出 if-then 信息，再遍历两个节点即可。

（4）外部变量定义

在 ExtDef 识别归约时，可能是一个外部变量定义。外部变量定义包括变量类型、变量列表和分号。而变量定义生成节点 Specifier，其中包括 type 信息和关键字信息。而变量列表则采用右递归构建，主要识别标识符或数组，具体实现为：

```
ExtDef:     Specifier ExtDecList SEMI     {$$=mknode(EXT_VAR_DEF,$1,$2,NULL,yylineno);}
            | ...
            ;
Specifier: TYPE      {$$=mknode(TYPE,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);$$->type=
            !strcmp($1,"int")?INT:FLOAT;}
            ;
ExtDecList:   VarDec          {$$=$1;}
            | VarDec COMMA ExtDecList {$$=mknode(EXT_DEC_LIST,$1,$3,NULL,yylineno);}
            ;
```

AST 遍历到对应节点后输出信息，并遍历孩子。

5 符号表的设计与实现

5.1 符号表的设计

符号表管理中，符号表数据结构的设计十分关键，其中包含了定义变量和函数的各种属性。通过符号表管理，可以有效进行后续的语义分析和中间代码生成。符号表表项数据结构如下：

```
struct symbol
{
   char name[33];   //变量或函数名
   int level;        //层号，外部变量名或函数名层号为 0，形参名为 1，每到 1 个复合语句层号加 1，退出减 1
   int type;         //变量类型或函数返回值类型
   int paramnum;     //形式参数个数
   int arraynum;     //数组维度
   char alias[10]; //别名，为解决嵌套层次使用，使得每一个数据名称唯一
   char flag;        //符号标记，函数：'F' ;变量：'V'; 参数：'P'; 数组：‘A’
   char offset;      //外部变量和局部变量在其静态数据区或活动记录中的偏移量
   char Ftype[20]; //函数类型，返回类型（参数类型，...）
   char Atype[20]; //数组类型，数组维度信息（[.][.]...）
};
```

其中，变量或函数名存放在 name 中。level 存放该变量所在层号，外部变量名或函数名层号都为 0，而形参为 1，进入复合语句时层号增 1，退出时减 1。type 记录变量类型，通过 TOKEN 值记录。paramnum 和 arraynum 分别记录形参个数与数组维度。alias 解决嵌套层次的变量名重名问题。flag 进行符号标记，用于判别该变量类型，F 表示函数名、V 表示变量、P 表示形参、A 表示数组。offset 用于记录当前变量的偏移量，在目标代码生成时使用，暂不考虑。Ftype 和 Atype 存放完整的函数类型和数组类型，完整函数类型包括返回参数类型和形参类型，完整数组类型包括维度信息。

符号表采用线性表、单符号表组织。将上述表项组织成结构数组，将该结构数组与表项号 index 组合产生符号表结构体。为了在退出复合语句块时能够更快定位到低层表项，通过构造结构体，保存各层的首表项信息，类似栈结构。具体

数据结构为：

```
struct symboltable
{
   struct symbol symbols[MAXLENGTH];
   int index;
} symbolTable;
struct symbol_scope_begin
{
   int TX[30];
   int top;
} symbol_scope_TX;
```

其中结构体 symbolTable 为顺序表、单符号表，结构体 symbol_scope_begin 用于保存栈顶层号与上层符号表的表项号 index，用于快速删除某一层所有表项。

除搭建符号表数据结构外，还需要对符号表进行操作方法的定义，具体包括初始化符号表、填表、进入复合语句体与退出复合语句体的表项修改、展示符号表等功能。在进行语法树遍历时，调用所需函数进行符号表管理。

5.2 符号表管理的实现

（1）符号表初始化函数 void InitSymbolTable()

将符号表栈顶 symbol_scope_TX.top 初始化为 0，将表序列号 symbolTable.index 初始化为 0。

（2）符号表显示函数 void DisplaySymbolTable()

直接输出符号表的所有填写内容的表项，即表中记录的 index 值。对于 F 类型表项，除输出基本信息外，还需要输出具体参数类型 Ftype 和形参个数 paramnum；对于 A 类型表项，除输出基本信息外，还需要输出具体参数类型 Atype 和维度信息 arraynum；对于 V 和 P 类型表项，则直接输出基本信息，如类型值等。

（3）进入复合语句体函数 void enterScope()

将层号增 1，将 symbol_scope_TX 的 top 置为层号，将上一层的最后一个表项入栈。

（4）退出复合语句体函数 void exitScope()

将 symbol_scope_TX 保存的上一层最后表项赋给符号表的 index，进行删除属于该层所有表项的操作。层号自减，将层号赋给 symbol_scope_TX 的 top 值。

（5）插入表项函数 void InsertSymbolTable(char *name,int type,int level,char flag,int num,char *moretype)

将符号表的 index 作为插入表项序列号，传入的参数 name，type，level，flag 可直接添加进表项中。参数中的 num 和 moretype 为补充说明，当 flag 为 F 类型时，num 为形参个数，moretype 存放函数类型的完整定义；当 flag 为 A 类型时，num 为维度数目，moretype 存放数组类型的完整定义。将其一同加入表项中。最后将 index 增 1，记录已添加表项数。

插入表项函数在变量声明时使用，即在 VarDec 节点处。由于函数体声明、外部变量声明、局部变量声明和形参定义都会到达该节点，故需要辅助信息的辅助判断。通过采用继承属性的传递，从而区别到达该节点的前驱节点。由于遍历抽象语法树是自顶向下遍历的，故继承属性的传递十分自然。

进入与退出复合语句体函数主要在函数定义语句块、复合语句块使用。由于函数名为原层号，而形参为高一阶层号，则对于函数定义语句块，则在 func_dec 节点调用 enterScope()，将形参定义在复合语句体内。而在 func_def 节点调用 exitScope()，在遍历完函数体节点的所有后续节点后，退出该复合体语句。其他复合体语句，如 IF-THEN、IF-THEN-ELSE、FOR、WHILE 等语句则在对应节点处进行 enterScope() 和 exitScope() 的调用。具体在进入该节点时调用 enterScope()，在遍历完节点孩子后退出时调用 exitScope()。

而符号表显示函数用于进行测试，在实际语义分析中无需调用，此处仅为了方便调试与测试。

6 一个 L-翻译模式的递归下降翻译程序的实现

6.1 L-翻译模式自顶向下语义计算

L-翻译模型中包含继承属性和综合属性。采用自顶向下方法进行语义计算无非是解决两个问题：继承属性如何传递，综合属性如何获取。采用递归下降子程序法实现自顶向下语义计算时，对于上述两个问题则是通过参数传递与返回值传递解决的。

非终结符的递归子程序中，形参为该非终结符的继承属性，返回值为该非终结符的综合属性。继承属性向下传递，综合属性向上传递。在该子程序内，则进行语法规则的分析，即继承属性传递，调用另外的非终结符子程序，语义计算，综合属性的返回。同时还需对错误信息进行输出。

6.2 一个 L-翻译模式递归下降翻译程序的实现

在进行将二进制小数转换成十进制小数的递归下降子程序法构建时，以 S 和 R 的递归下降子程序构建为例。

S 的相关产生式为 S→B {R.ival = B.val; R.ilen = 1} R { S.val = R.val; S.len := R.len }，其中 R.ival 和 R.ilen 为继承属性，R.val，R.len，S.val，S.len，B.val 为综合属性，该产生式的 SELECT 集为{‘0’，‘1’}。由于 S 无继承属性，故无形参；有两个综合属性，可将其构建为结构体，统一作为返回值返回。使用 lookahead 查看当前字符是否为 SELECT，否则报错。接着进行继承属性的赋值和 B 与 R 递归子程序调用，最后填写 S 的综合属性并返回。

R 的相关产生式为 R→B {R1.ival=2 * R.ival + B.val; R1.ilen = R.ilen + 1} R1 {R.val = R1.val; R.len = R1.len }和 R→ ε {R.val = R.ival; R.len:=R.ilen }，其中 R.ival 和 R.ilen 为继承属性，R.val、R.len 和 B.val 为综合属性。R→BR 的 SELECT 集为{‘0’，‘1’}，R→ ε 的 SELECT 集为{‘.’，‘\n’}。将 R 的 ival 和 ilen 作为该递归子程序的形参，接着识别 lookahead，当为 0 或 1 时，进行 R1 继承属性赋值，语义计算，R 递归

子程序的调用，随后返回 R 的综合属性；当为‘.’或‘\n’时，直接进行综合属性的赋值，并返回。给出该具体实现：

```
#include "common.h"

// R->BR | ε      select(R->BR)={'0','1'}; select(R-> ε) ={'.','\n'}
b_info parseR(int v, int l){
      b_info r1,r;
      int Bval,R1ival,R1ilen;
      if(lookahead == '0' || lookahead == '1'){
            Bval = parseB();
            R1ival = 2*v+Bval;
            R1ilen = l + 1;
            r1 = parseR(R1ival,R1ilen);
            r.val = r1.val;
            r.len = r1.len;
      }
      else if(lookahead == '.' || lookahead == '\n'){
            r.val = v;
            r.len = l;
      }
      else{
            printf("syntax error\n");
              exit(0);
      }
      return r;
}
```

7 静态语义分析

7.1 静态语义分析的设计

静态语义分析主要关注综合属性和继承属性的选取与传递，相关属性放置在通用节点定义里，对于节点定义，除了上述实验中使用的基础属性外，如节点类型、层号等，还需要添加综合属性：paramnum、paramtype、arraynum、arraytype 用来传递函数和数组的补充信息；继承属性：flag、breakflag、continueflag、returnflag 和 LorR 用于相关语义计算。

对于综合属性，由于生成抽象语法树是自底向上的，故直接采用语法制导方式获取。拿形参个数 paramnum 和形参类型 paramtype 的获取举例，将其语义计算直接加入翻译模式中，具体代码为：

```
VarList:   ParamDec {$$=mknode(PARAM_LIST,$1,NULL,NULL,yylineno);$$->paramnum=1;
             strcpy($$->paramtype,$1->paramtype);}
         | ParamDec COMMA VarList   {$$=mknode(PARAM_LIST,$1,$3,NULL,yylineno);
             $$->paramnum=$3->paramnum+1;strcpy($$->paramtype,$1->paramtype);
             strcat($$->paramtype,",");strcat($$->paramtype,$3->paramtype);}
         ;
```

在形参规约时，对于无参情况，paramnum 直接置为 0，且 paramtype 置为 void。对于有参情况，见具体代码中，初始化 paramnum 为 1，接着在后续规约中进行加 1 即可，而 paramtype 则采用 strcat 连接该形参类型和以获取形参类型，由此当规约到上层节点时，综合属性可直接计算得到。

对于继承属性，由于遍历抽象语法树是自顶向下的，故在遍历语法树时进行传递。拿 breakflag 为例，为 0 时表示在该处不允许出现 break，为 1 时表示可以出现。首先在函数定义节点处定义 breakflag 为 0，在后续节点遍历时，该属性继承双亲节点属性。当到达 while 和 for 语句节点时，将该属性置为 1，后续仍然进行继承，表示允许在 while 和 for 循环体内出现 break。在 break 节点处进行判断，该节点的 breakflag 属性为 0 时，产生语义错误，表明在不该出现 break 的地方出现了 break。其他继承属性类似，与具体的语义分析和语义判错相对应。

7.2 语义分析的实现

首先将语义错误信息定义在一个 enum 类型中，enum error_list{OK,UNDECLARED,...,WITHOUT_RETURN_VALUE}，接着定义错误信息输出函数，函数原型为 void printError(int lineno,enum error_list error,char *errorident)。其中 lineno 为错误发生行号、error 为错误的 enum 类型值，errorident 为错误补充信息，如发生错误的标识符等。函数体内进行 switc-case 的分情况输出。

（1）遍历符号表进行语义检查

语义分析的主体在符号表的遍历上，符号表遍历函数原型为 int SearchSymbolTable(char *name,int type,int level,char flag,int num,char *moretype,int choice,int LorR)，其中 name、type、level、flag、num、moretype 为调用该函数节点的属性信息，choice 进行遍历选择，提供两种遍历模式，LorR 用于判断左右值的语义问题。而函数返回错误类型，若无错误则返回 OK。

将重定义 redefinition 和同名定义 redeclared 归为第一类遍历表模式。主要用于变量定义、函数定义节点处。在遍历过程中，若发现有相同 name 的表项，且两者的 flag 和其他属性值相同，则表明产生 redefinition 错误；若属性值不相同，则产生 redeclared 错误。

对于其他错误类型归为第二类遍历表模式。主要用在变量的使用节点处。首先遍历符号表，寻找序列号最大的与传入参数名 name 一致的表项。若查询失败，则表明未定义 undeclared；查询成功，则进行其他语义错误的判断。

若传入 flag 为变量类型：若该表项为数组，则产生数组引用不当错误；若该表项为函数，且作为左值出现，产生左值表示错误；若该表项为函数，且作为右值出现，产生错误使用操作符错误。

若传入 flag 为函数类型：若该表项不是函数，则产生变量当函数调用；若该表项为函数，而表项的形参个数小于实际使用参数，则产生实际参数传入过多错误，相反若大于则产生实际参数传入过少错误。

若传入 flag 为数组类型：若该表项不是数组，则产生非数组类型的使用错误；若该表项为数组，但其中引用的变量不是 int 型，则产生下标错误。

（2）通过属性传递进行语义检查

上述设计中对节点属性进行了扩充，补充了 breakflag、continueflag、returnflag 继承属性。

breakflag 进行 break 位置正确性判断，当继承到 break 节点时的 breakflag 为 0 时产生错误信息，在设计中已进行详细解释，在此不赘述。continueflag 则进行 continue 位置正确性的判断，与 break 类型。

returnflag 则传递函数返回值信息，在函数声明时，若标识符为 VOID，则表明无参返回，returnflag 置 0，否则置 1。将该属性逐节点传递，当遍历到在 return 语句时进行判断。若 returnflag 为 0，而该语句有参数返回，则产生返回值与函数声明不一致的错误；若 returnflag 为 1，而该语句无参数返回，同样产生语义错误。

8 PL/0 编译系统分析与修改

8.1 PL/0 编译系统分析

Pl/0 是一种极简化的 PASCAL 语言，而 PASCAL 是一种结构化编程语言。而 PL/0 编译系统则是面向该语言的编译程序，将 PL/0 翻译成中间代码 P-CODE。该中间代码可在 P-CODE 虚拟机上允许。

同样的，PL/0 编译系统包括词法分析、语法分析、语义分析和中间代码生成。词法分析主要由函数 getsym()完成，getsym 会调用 getch()。getsym 返回下一个 token，返回的 token 存储在全局变量 symbol(enum 类型)，其值则存储于全局变量 id(标识符、关键字、界符、运算符)或 num(数字)。PL/0 的文法是 LL(1)的，其语法分析采用自顶向下分析法，具体采用递归下降子程序法，即为每个语法单位编写一个子程序。PL/0 的语义分析和中间代码生成差不多都在 block()完成，gen()只负责将生成的中间代码记录到中间代码数组，并将数组下标+1 而已。

8.2 PL/0 编译系统的修改

对 PL/0 编译系统进行修改，使其一次性输出全部代码。

（1）main()函数形参和 main()内代码的修改

将 main 函数参数补充完整，为 int main(int argc,char *argv[])，并将 argv[1]传入的文件名作为源程序文件，写入 fname 中，strcpy(fname,argv[1]);。接着注释掉函数体内的辅助输入输出代码。将 listswitch 和 tableswitch 置为 false，不需要输出虚拟机中间过程代码和符号表。同时删除相关文件操作，如 fclose(fa1)等。

（2）listallcode()的修改

输出生成的目标代码，cx 为代码长度，循环输出至评测文件 fa2 即可。fprintf(fa2,"%d %s %d %d\n", i, mnemonic[code[i].f], code[i].l, code[i].a);

（3）getch()函数的修改

将该函数内的从 fin 读取字符操作删除，即删除 fscanf(fin,"%c", &ch);。

9 总结与展望

9.1 实验总结

本次实验主要完成了常用编译器的使用、编译器前端设计与实现、符号表管理的设计与实现、出错处理的设计与实现、L-翻译模式的递归下降翻译程序的实现、PL/0 编译系统的分析与修改。其中前端设计与实现包括词法分析、语法分析和语义分析。

编译工具链的使用中，完成了对常见编译工具 gcc 编译器、clang 编译器、arm-linux-gnueabihf-gcc 交叉编译器和 qemu-arm 虚拟机的使用。

词法分析器的设计与实现中，完成了使用 flex 进行词法分析器的构建。完成了标识符、int 型字面量、float 型字面量、单行与多行注释、其他关键字的正则式的构建。完成了词法错误产生的相关正则式的构建。

语法分析器的设计与实现中，完成了使用 bison 进行语法分析器的构建。完成了 for 语句与数组产生式的添加、操作符的添加、yyerror 的自定义，完成了抽象语法树生成与遍历的补充和修改，使其符合修改后的语法规则，完成了对优先级关系的深度理解。

符号表管理的设计与实现中，完成了符号表数据结构的构建，完成了符号表管理函数的构建，包括初始化符号表、插入表项、显示符号表、进入与退出复合语句块等函数的构建。

静态语义分析的设计与实现中，完成了节点属性的扩充，完成了出错处理程序的构建，完成了综合属性和继承属性的传递，完成了符号表的遍历函数，通过上述构建实现最终的语义分析。

L-翻译模式的递归下降翻译程序的实现中，掌握了使用自顶向下的具体实现方法，了解了非终结符综合属性和继承属性的传递，完成了递归下降子程序的构建。

PL/0 编译系统的分析与修改中，了解了 PL/0 编译系统的整体架构和组成成

分，完成了对 PL/0 编译器的修改，使其能够一次性输出 P-CODE 代码。

9.2 实验感想

通过此次编译原理课程实践，教会了我如何将编译原理课堂上的理论知识转化为实际的操作应用，将原理知识转化为实践能力。通过对编译器前端的搭建，让我感受到构建编译器的不易，并且每个环节都十分具有考验，并且遇到了许许多多的问题。譬如，在进行语义分析时，若直接在原语法分析搭建的语法规则上进行，会产生诸多冲突，于是当是经过内心的挣扎后，选择重新构建语法规则架构，从而符合综合属性自底向上，继承属性自顶向下的传递思路。除此之外，在各环节的细节处理上十分考验细心与耐心，这使我养成了做事不急躁，办事有耐心的习惯。经过种种困难的磨砺，最终有了编译器的半成品，还是感到收获满满，成就感满满。

对于实验建议方面，有一点建议十分强烈。在符号表实现环节，进行符号表展示时需要按照测试机的方式完美输出才能过关，而那一关的符号表显示并不是必要的内容，在生成格式上花了大量时间，但之后的实验并不设计符号表的显示，从而有些舍本逐末了。希望在以后能够改善这一点，比如加入老师检查的因素，对于强格式化测试可以采用人工检查的方式赋分，而不是机械的满足十分严格的输出格式。

9.3 实验展望

本次实验由于其他比赛和课程堆积的原因，没有完成一个完整的编译器，而只是搭建了编译器的前端和一些补充内容的实现，感到有些失望。曾经目标是编写完中间代码生成，希望这个任务能在有机会进行学习补充，从而加深对编译器的理解与认识。

之后的学习与生活可能不会再接触到编译器内部实现了，可编译原理里的各种思想将永远铭记于心，可能在来日的某一次毕业设计，某一篇论文编写，亦或是生活中的某一时刻，带来灵感上的触动。

参考文献

[1] 王生原，董渊. 编译原理（第 3 版）. 北京：清华出版社，2015

[2] 袁春风，余子濠. 计算机系统基础（第 2 版）. 北京：机械工业出版社，2018