编译原理实验

中国人民大学信息学院

内容

- 实验概述
- PL0/SYSY语言简介
- 词法分析器实验部分

实验设置

- 实验内容：二选一
  - PL/0语言的编译器
  - SysY语言的编译器
    - 参考：全国大学生计算机系统能力大赛—编译系统设计赛（https://compiler.educg.net/#/oldDetail?name=2022%E5%85%A8%E5%9B%BD%E5%A4%A7%E5%AD%A6%E7%94%9F%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%B3%BB%E7%BB%9F%E8%83%BD%E5%8A%9B%E5%A4%A7%E8%B5%9B%E7%BC%96%E8%AF%91%E7%B3%BB%E7%BB%9F%E8%AE%BE%E8%AE%A1%E8%B5%9B）
- 实验目的
  - 理解编译器的工作机制，掌握编译器的构造方法
  - 掌握词法分析器的生成工具LEX的用法
  - 掌握语法分析器的生成工具YACC的用法

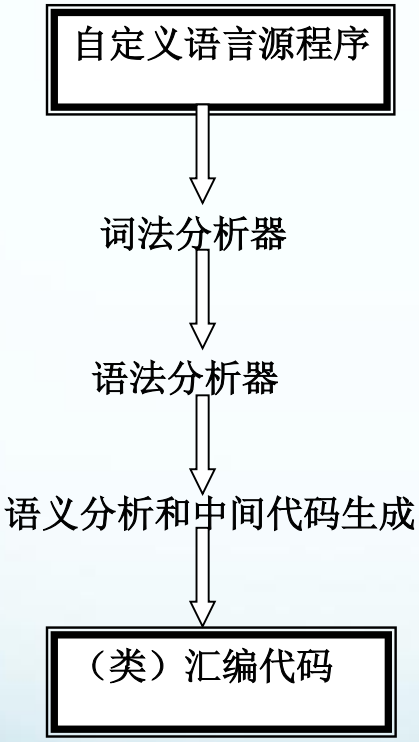
实验要求

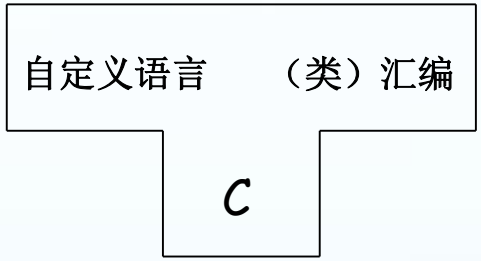
- 独立完成
  - 程序源代码（有注释，易读）
  - 文档：编译器的设计说明
    - 每个实验提交一份实验报告
    - 最后一次实验整合为整个编译器的设计说明文档
- 按时提交
  - 迟交影响成绩
- 鼓励语法扩展
- 鼓励参加“全国大学生计算机系统能力大赛—编译系统设计赛”

实验环境与成绩评定

- 成绩评定
  - 程序：70%
  - 文档：30%
- 实验分成三部分：
  - 词法分析
  - 语法分析
  - 语义分析和目标代码生成
- 实验环境
  - Windows
  - C或C＋＋

编译程序的实验流程





PL/0 语言简介

- PL/0语言是Pascal语言的子集
  - 数据类型只有整型
  - 标识符的有效长度是10，以字母开始的字母数字串
  - 数最多为14位
  - 过程无参，可嵌套（最多三层），可递归调用
  - 变量的作用域同PASCAL，常量为全局的

PL/0 语言简介

- 语句类型：
  - 赋值语句，if...then..., while...do..., read, write, call, 复合语句begin... end， 说明语句： const..., var..., procedure...
- 13个保留字：
  - if, then, while, do, read, write, call, begin, end, const, var, procedure, odd
- 扩展（三必选一）： （学号尾号模三取余）
  - 余0：For循环语句
  - 余1：整形数组类型
  - 余2：Case语句

PL0语言的文法

- EBNF范式:可说明哪些符号序列是对于某给定语言在语法上有效的程序。

- EBNF范式的符号说明
  - < >：语法构造成分，为非终结符
  - ::= ：该符号的左部由右部定义，读作“定义为”
  - |：或
  - { }：括号内的语法成分可重复
  - [ ]：括号内成分为任选项
  - ( )：圆括号内成分优先

PL0语言的文法

- <程序> ::= <分程序>.
- <分程序> ::= [<常量说明部分>][<变量说明部分>][<过程说明部分>]<语句>
- <常量说明部分> ::= CONST<常量定义>{,<常量定义>};
- <常量定义> ::= <标识符>=<无符号整数>
- <无符号整数> ::= <数字>{<数字>}
- <变量说明部分> ::= VAR<标识符>{,<标识符>};
- <标识符> ::= <字母>{<字母>|<数字>}
- <过程说明部分> ::= <过程首部><分程序>{;<过程说明部分>};
- <过程首部> ::= PROCEDURE<标识符>;
- <语句> ::= <赋值语句>|<复合语句>|<条件语句>|<当型循环语句>|<过程调用语句>|<读语句>|<写语句>|<空>

PL0语言的文法

- <赋值语句> ::= <标识符>:=<表达式>
- <复合语句> ::= BEGIN<语句>{;<语句>}END
- <条件> ::= <表达式><关系运算符><表达式>|ODD<表达式>
- <条件语句> ::= IF<条件>THEN<语句>
- <表达式> ::= [+|-]<项>{<加法运算符><项>}
- <项> ::= <因子>{<乘法运算符><因子>}
- <因子> ::= <标识符>|<无符号整数>|'（'<表达式>'）'
- <加法运算符> ::= +|-
- <乘法运算符> ::= *|/
- <关系运算符> ::= =|#|<|<=|>|>=

PL0语言的文法

- <当型循环语句> ::= WHILE<条件>DO<语句>
- <过程调用语句> ::= CALL<标识符>
- <读语句> ::= READ’（‘<标识符>{,<标识符>}’）’
- <写语句> ::= WRITE’（‘<表达式>{,<表达式>}’）’
- <字母> ::= a|b|...|X|Y|Z
- <数字> ::= 0|1|...|8|9

SysY的文法

- 语言支持 int 类型和元素为 int 类型且按行优先存储的多维数组类型，其中 int 型整数为 32 位有符号数；const 修饰符用于声明常量。
- 编译单元 CompUnit → [CompUnit] (Decl | FuncDef)
- 声明 Decl → ConstDecl | VarDecl
- 常量声明 ConstDecl → '**const**' BType ConstDef { ',' ConstDef } ';'
- 基本类型 BType → '**int**'
- 常数定义 ConstDef → Ident { '**[**' ConstExp '**]**' } '=' ConstInitVal
- 常量初值 ConstInitVal → ConstExp
- | '**{**' [ConstInitVal { '**,**' ConstInitVal }] '**}**'
- 变量声明 VarDecl → BType VarDef { ',' VarDef } ';'
- 变量定义 VarDef → Ident { '[' ConstExp ']' }
- | Ident { '[' ConstExp ']' } '=' InitVal
- 变量初值 InitVal → Exp | '{' [InitVal { ',' InitVal }] '}'
- 函数定义 FuncDef → FuncType Ident '(' [FuncFParams] ')' Block
- 函数类型 FuncType → '**void**' | '**int**'
- 函数形参表 FuncFParams → FuncFParam { ',' FuncFParam }
- 函数形参 FuncFParam → BType Ident ['[' ']' { '[' Exp ']' }]

SysY的文法

- 语句块 Block → '**{**' { BlockItem } '**}**'
- 语句块项 BlockItem → Decl | Stmt
- 语句 Stmt → LVal '**=**' Exp '**;**' | [Exp] '**;**' | Block
- | '**if**' '(Cond ')' Stmt ['else' Stmt]
- | '**while**' '(' Cond ')' Stmt
- | '**break**' ';' | '**continue**' ';'
- | '**return**' [Exp] ';'
- 表达式 Exp → AddExp　　注：SysY 表达式是 int 型表达式
- 条件表达式 Cond → LOrExp
- 左值表达式 LVal → Ident {'[' Exp ']'}
- 基本表达式 PrimaryExp → '(' Exp ')' | LVal | Number
- 数值 Number → IntConst
- 一元表达式 UnaryExp → PrimaryExp | Ident '(' [FuncRParams] ')'
- | UnaryOp UnaryExp
- 单目运算符 UnaryOp → '+' | '−' | '!' 注：'!'仅出现在条件表达式中

SysY的文法

- 函数实参表 FuncRParams → Exp { ',' Exp }
- 乘除模表达式 MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
- 加减表达式 AddExp → MulExp | AddExp ('+' | '−') MulExp
- 关系表达式 RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
- 相等性表达式 EqExp → RelExp | EqExp ('==' | '!=') RelExp
- 逻辑与表达式 LAndExp → EqExp | LAndExp '&&' EqExp
- 逻辑或表达式 LOrExp → LAndExp | LOrExp '||' LAndExp
- 常量表达式 ConstExp → AddExp 注：使用的 Ident 必须是常量

词法分析器实验

- 实验内容：用flex工具生成一个PLO/SYS语言的词法分析程序，对PLO/SYS语言的源程序进行扫描，识别出单词符号的类别，输出符号的相关信息。
- 输入：PLO/SYS 源程序
- 输出：例如把单词符号分为下面六类，然后按单词符号出现顺序依次输出各单词符号的种类和出现在源程序中的位置（行数和列数）。
  - K类（关键字）：也可以一符一种
  - I类（标识符）
  - C类 (常量)
  - O类（算符）：也可以一符一种
  - D类（界符）：也可以一符一种
  - T类（其他）

词法分析器实验

- 例如，对如下的PL0程序：

```
var a,b;
procedure test;
var t;
begin
  t := 1;
end;
   begin
            call test;
    end.
```

词法分析结果输出，

var：K，（1，1）

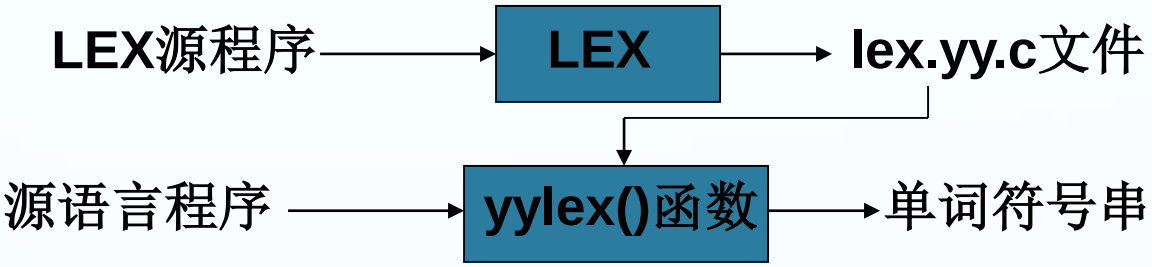
a ：I，（1，4）

，：D，（1，5）

…

LEX概述

- LEX是一个词法分析器的自动产生系统。



- LEX源程序的核心是识别规则，它由正则式和动作组成。

LEX源程序的格式

%{

声明　　- - 可选

%}

辅助定义　　- - 可选

%%

识别规则　　- - 必须有

%%

用户子程序　　- - 可选

声明

- 所有嵌在“%{”和“%}”之间的内容将被原样拷贝到lex.yy.c文件中。
- 在声明中，可以引入头文件、宏定义以及全局变量的定义。

例如：

```
%{
#include      <stdio.h>
int    num_ident, num_keyword;
%}
```

辅助定义

- 辅助定义可以用一个名字代表一个正则式。
- 辅助定义的语法是：辅助定义名　正则式

  注意：辅助定义必须从第一列写起。

  后面的辅助定义可以引用前面的辅助定义。
- 在正则式中，用“{辅助定义名}”可以引用相应的正则式。

  例如：

  NEW_LINE　(\n)

  INTEGER　　([0-9]+)

  EXPONENT　([Ee][+-] {INTEGER})

识别规则

- 识别规则由两部分组成：正则式和相应的动作。
- 正则式用于描述输入串的词法结构。
- 动作用于描述识别出某一个词形时要完成的操作。

例如：

```
%%

void      {return          T_Void;}
```

识别规则之正则式

- 正则式：由正文字符和正则式运算符组成
  - 正文字符为：计算机的字符集
    - 几个特殊的字符为：“ ”、“\t”、“\n”
  - 正则式运算符包括：.,[,],^,-,*,+,?, {,},“,\,(,),|,/,$,<,>。

正则式的写法（1）

x	匹配字符x
.	匹配除换行外的所有字符
[xyz]	字符集合，匹配字符‘x’、‘y’或‘z’
[^A-Z]	字符集合，匹配除大写字母外的所有字符
r*	正则式r出现零次或多次
r+	正则式r出现一次或多次
r?	正则式r出现零次或一次
r{2,5}	正则式r重复2至5次
r{2,}	正则式r重复2次以上（含2次）
r{4}	正则式r重复4次
{name}	辅助定义“name”的展开
“[”	字符[
\”	字符”

正则式的写法（2）

\0	空字符(ASCII 码为 0)
\123	ASCII 码为八进制数123的字符
\x2a	ASCII码为十六进制数 2a的字符
(r)	匹配正则式r
rs	正则式r后面紧跟正则式s,串联
r|s	匹配正则式r或s
r/s	当正则式r后面紧跟s时，匹配r
^r	当正则式r位于行首时，匹配r
r$	当正则式r位于行尾时，匹配r，相当于"r/\n"
<s>r	在开始条件s下匹配正则式r
<s1,s2,s3>r	在开始条件s1,s2或s3下匹配正则式r
<*>r	在任何开始条件下都匹配正则式r
<<EOF>>	遇到文件结束符时
<s><<EOF>>	在开始条件s下遇到文件结束符时

书写正则式的注意事项

- 每条规则的正则式必须从第一列写起。
- 为避免与运算符混淆，建议对所有的非字母数字字符都使用转义符“或\ 。
- 在集合中，字符之间不要留空格，否则空白符“ ”将被包含在集合中。
- 辅助定义可以使正则式更加简洁清晰。

正则式举例

[a-zA-Z0-9]　表示所有的字母和数字组成的集合；

[^ \t\n]　表示除空格、tab和换行外的所有字符组成的集合；

(\"[^"\n]*)　表示以双引号开头，后跟除双引号和换行外的若干字符组成的字符串，例如：“here is a string

a{1,5}　表示a重复1至5次形成的串的集合，即{a, aa, aaa, aaaa, aaaaa}

[A-Za-z][A-Za-z0-9]*　表示以字母开头，后跟若干字母和数字的串

([Ee][-+][0-9]+)　表示科学计数法的指数部分

识别规则之动作

- 识别规则的动作是一段C语言程序，将被原样照抄到lex.yy.c文件中。
- 缺省规则：输入串中不与任何正则式匹配的字符串将被原样照抄到输出文件中。
- 在词法分析器实验中，基本的动作就是记录单词符号的类别

书写动作的注意事项

- 动作必须从正则式所在行写起。
- 当某条规则的动作超过一条语句时，必须用大括号括起来。
- 如果希望在输出中滤去某些字符，相应的动作为空

  例如：

  [\t\n]　　{}
- 如果不希望照抄输出，就要为每一个可能出现的词形提供规则。

动作中用到的全局变量

- yytext：char *类型，指向当前正被某规则匹配的字符串。
- yyleng：整型，存储yytext中字符串的长度。被匹配的串在yytext[0] ~ yytext[yyleng-1]中。

LEX源程序举例

```
%{
        int num_lines = 0, num_chars = 0;
%}
%%
\n      {++num_lines; ++num_chars;}
.       {++num_chars;}
%%
main(){
   yylex();
   printf("# of lines = %d, # of chars = %d\n",
num_lines, num_chars );
}
```

识别规则的二义性

有时输入串中的字符可以与多条规则匹配，在这种情况下，LEX有两个处理原则：

- 能匹配最多字符的规则优先；
- 若各规则匹配的字符数目相同，先给出的规则优先。

例如，给定规则如下：

```
void            {return T_Void;}
[A-Za-z]+               {return T_Identifier;}
```

“void”将被识别为T_Void，

“voida”将被识别为T_Identifier。