基础课程实践项目

PL/0 编译器的熟悉与 C0 语言编译器的构造

张昱　　2013 年秋

P1 预备阶段

1、实验环境熟悉

安装 Linux，使用 GCC 来编译 C 程序，如 PL/0 编译器，并运行编译生成的可执行程序。

2、PL/0 语言了解并确定与 PL/0 语言表达能力类似的 C0 语言

阅读 pl0.zip中的doc/pl0.pdf和sample/tests.pl0，理解 PL/0 语言的特征；编译PL/0 编译器，运行所生成的可执行的PL/0 编译器并用它编译运行tests.pl0。

C0 语言只有整数类型，它是 C 语言的一个子集，其程序结构包含变量声明（PL/0 中的常量定义用带赋值的变量声明来表示）、变量声明和无参函数声明（定义）。C0 语言有赋值语句、条件语句、循环语句、函数调用语句、复合语句和空语句。

注意：C0 语言没有 PL/0 语言所允许的嵌套定义的过程（即在一个过程体中定义另一个过程）。C0 语言编写出的程序是一个合法的 C 程序，可以用 GCC 编译得到可执行程序并运行。

我们将在后面的课程实验中，要求大家写出 C0 语言的词法和语法定义，并构造 C0 的编译器。

3、编写 C0 程序

用 C0 语言编写与 tests.pl0 功能相同的 tests.c，并用 GCC 编译并运行。

4、初步理解 PL/0 编译器的中间表示

对比 pl0.zip中的sample/tests.pl0 和sample/tests.res，尝试理解PL/0 编译器所用的中间表示——一种栈机器代码（在doc/pl0.pdf的 1.2 节有简要介绍）。

P2　词法分析

1、理解 PL/0 编译器的词法分析过程

PL/0 编译器的词法分析由 src/pl0.c 中的 getsym( )函数来负责完成。

2、扩展 PL/0 的词法

扩展 PL/0 编译器以支持：1）由/*开始后跟 0 个或多个字符、再以*/结尾的多行注释；2）以 0 开头后跟 0~7 这 8 种数字组成的八进制数；3）以 0x 或 0X 开头后跟 0~9、A~F、a~f 组成的十六进制数。

3、词法的形式描述

分别给出 PL/0 语言和 C0 语言所的词法规范（即用正规式描述语言中合法的单词）。

4、学习使用词法分析器的生成工具 Flex

阅读 Flex manual和一个简单的表达式语言的词法分析 例子，了解Flex的输入词法规范文件的格式，以及Flex生成的词法分析器的接口形式和实现。你可以从 flex-2.5.35.tar.gz 的examples目录下获得更多 使用Flex的例子。

5、用 Flex 生成 PL/0 的词法分析器

用 Flex 生成 PL/0 的词法分析器 getsym( )，修改 PL/0 编译器源代码 src/pl0.c，使得 PL/0

编译器能调用 Flex 生成的词法分析器来进行词法分析。

注意：

1) 要求用宏和条件编译来控制词法分析器的选择，即编译器是调用原有 pl0.c 中的 getsym( )还是调用 Flex 生成的词法分析器；

2）能快速编译和运行使用 1）中任意一种方式（原有 pl0.c 中的 getsym( )或 Flex 生成的词法分析器）构造的 PL/0 编译器。

6、P2 提交的目录结构要求

按如下目录结构提交到学生自己的 git 库的 master 分支(缺省的分支就是 master):

P2

|----README	提交内容说明，包含如何编译运行所提交的内容等
|----src/	存放源程序文件（*.h, *.c）和 Makefile
|----doc/	实验设计文档
|----bin /	帮助快速编译和运行实验内容的 shell 脚本文件
|----test/	存放测试程序

P3 PL/0 语言编译器的理解和 C0 语言的语法定义

1、识别 PL/0 编译器在词法分析阶段和语法分析阶段所处理的错误

根据 pl0.pdf中第 20~21 页提供的错误编号及含义，要求：

1）推测分别在词法分析阶段和语法分析阶段识别处理的错误种类；

2）阅读 PL/0 编译器源代码，确认分别在词法分析阶段和语法分析阶段识别出的错误种类，编写相应的测试程序并按错误类别命名保存在提交的 **P3/test/pl0/error** 目录下；

3）针对 2）中所确认的每一种错误，撰写电子文档（保存在提交的 **doc** 目录下）描述：该类错误发生的场景、该类错误发生后 PL/0 编译器的错误处理方法、你认为对这类错误的处理是否有更好的改进方案。

注意理解 (可参阅旧版以PL0 为基础的 《编译原理实践教程》)：

1）test( )函数的作用及它的三个参数的含义，能举例说明第二个参数带来的好处和引起的问题。

2）你阅读的 PL/0 编译器实现的语法和 pl0.pdf 所定义的语法是否完全一样，若不一样，请指出有区别的地方。

3）PL/0 编译器是如何把出错信息和源程序中的出错位置对应起来的。

2、写出 C0 语言的语法规范

1）用教材中习题 3.7 所介绍的扩展方式来给出 C0 语言的文法，并写入 C0 语言的设计文档（保存在提交的 **P3/doc** 目录下）中。

2）阅读 PL/0 编译器，为 PL/0 写一个 LL 文法（按照 PL/0 编译器所实现的递归下降语法分析方式）。

3）结合上述为 PL/0 编写 LL 文法的作业，为 C0 语言编写一个对应的 LL 文法。

3、理解 PL/0 编译器的实现和语言特征，编写 C0 语言程序

1）编写至少 5 个强调不同语言特征 PL/0 测试程序，在程序中以注释的形式说明该程序所强调的语言特征；然后为所编写的 PL/0 测试程序，编写对应的 C0 测试程序。

要求：所编写的测试程序按照所使用的语言不同分别置于提交的 **P3/test/pl0** 和 **P3/test/c0**

目录下。

2）理解 PL/0 编译器中使用的符号表的组织结构（包含相关的全局变量 table 和函数 position( )），说明标识符的插入、查找和删除方面的特点。

3）理解 PL/0 语言的中间代码生成：

A. 理解PL/0 编译器中使用的中间表示（在 pl0.pdf的 1.2 节、《编译原理实践教程》中有简要介绍）；

B. 理解 PL/0 编译器的代码生成函数 gen( )的作用以及三个参数的含义，gen( )中所使用的全局变量 cx、cxmax、code 分别存储什么信息；

C. 总结各个文法结构对应的中间代码结构分别是什么，其中各文法结构对应的函数中使用的、与代码生成有关的全局变量 dx、lev、levmax 等分别存储什么信息；

大家需要注意理解 PL/0 编译器中对每一种语法结构的处理，以及这种语法结构对应到 C0 中的结构形式；并在 C0 语言的设计文档中总结上述 2）和 3）的理解结果、以及 PL/0 和 C0 的语法结构对比。

P4 构造 C0 语言的分析器（词法、语法）：能分析正确的 C0 程序

一、实验内容

1、理解 bison-examples.zip，学习使用语法分析器的生成工具Bison

阅读 Bison manual（Bison是Yacc的变种）和简单的表达式语言（L-expr）的分析器构成 bison-examples.zip，了解Bison的输入文法规范文件的格式、Bison如何与Flex协作生成某个语言的分析器、所生成的分析器的接口形式和实现具有什么样的特征、以及语言的文法对生成的分析器的分析表状态空间有何影响。**请将你的理解写成文档存入提交的doc目录**。

1）查看 bison-examples.zip中Makefile文件里名为expr的编译任务，理解任务中各个命令的作用和所使用的选项的含义。

2）bison-examples.zip中config目录下的expr.y和expr1.y均能在进行语法分析的同时完成表达式的求值——语法制导的翻译，通过expr.y和expr1.y理解Yacc输入文法规范文件的格式，消化语法指导的翻译方案。

3）bison-examples.zip中src目录下的expr.tab.h、expr.tab.c和expr1.tab.c，总结expr.tab.c和expr1.tab.c的异同。说明expr.y和expr1.y中对表达式语言的不同文法表示对生成的分析器源代码expr.tab.c和expr1.tab.c有何影响。

2、为 C0 语言构造能识别正确 C0 程序的分析器，方法不限：可以用 lex+bison,也可以手工编写

可参考ANSI-C语言的 lex词法规范和 Yacc文法规范。

该项实验不要求在 C0 语言的 Yacc 输入文法规范的每个产生式选择中加入语义动作；而要求：由所编写的 C0 语言 Yacc 文法规范文件生成的分析器能正常分析完成正确的 C0 程序，而对于存在错误的程序，分析器则能报告出第 1 次遇到错误的位置。

二、提交要求

1、 包含修改后的 P 3

2、 包含一个说明提交内容以及完成/未完成/参考/问题等情况的文档

3、 包含至少 5 个 C0 程序（要求说明程序的特征）、5 个 PL/0 程序作为测试程序

4、 包含对 flex+bison 使用心得的文档以及构造分析器的问题与解答文档

5、 包含 P3 和 P4 的源代码文件以及相关的编译运行脚本文件。

P5 构造 C0 语言的分析器：能为正确的 C0 程序产生 AST

一、实验内容

1、理解 bison-examples中的抽象语法树(AST)的数据结构特点及其构造、使用和注销方法，为C0 语言设计AST结构以及相关的操作方法。

1）参考 bison-examples 中的 config/asgn2ast.y，AST 结点和树的类型定义见 include/common.h，AST 相关函数实现见 src/ast.c；符号表的相关声明和定义见 include/common.h，符号表的相关函数实现见src/symtab.c。

2）对 bison-examples中已经提供的AST结点和树类型（如结点指针类型ASTNode、AST树指针类型ASTTree）以及相应的创建结点的厂函数（如newNumber()、newPrefixExp()等）等进行扩充以表达C0 语言的AST;

3）对 bison-examples中已经提供的符号表结构和函数进行扩展，以支持对C0 程序中的函数和变量的管理。

2、在 P4 构造的分析器基础之上，利用语法制导的翻译技术，在分析器的语义动作中增加构造 AST 的代码，使得修改后的分析器能为正确的 C0 程序构造 AST。

1）思考 AST 结点在语法分析中的创建时机和信息填充时机，在分析器的相应阶段增加 AST 结点创建和信息填充等的代码。

2）编写 dump 函数，它能根据 AST 输出对应的 C0 源程序，以检验所构造的 AST 是否正确。

注意：所构造的 AST 需要保存 AST 结点对应于源程序的位置信息。

二、提交要求

1、 在 git 库中更新原有的 P3 和 P4 （如果有修改的话）

2、 新建 P5 目录，在其中包含如下子目录：

doc：存放 AST 结构和语法制导翻译等的分析、设计和实现文档；

config：存放 flex 和 yacc 文法规范描述文件

include：存放 C 程序的头文件

src：存放 C 源程序文件

test：存放测试代码，也可以统一建立一个与 P2~P5 并列的 test 目录，存放供各项目使用的测试程序

bin：存放编译运行脚本文件

3、 更新 git 库根目录下的 README，增加对这次提交内容的说明。

附 1：参考资料

- LALR分析器的生成工具Yacc的变种Bison：Bison manual、Bison源码；
- 运用Bison构造编译器的示例代码包 bison-examples.zip；
- ANSI-C语言的 lex词法规范和 Yacc文法规范；
- 一个可变目标的C编译器 LCC：它采用递归下降的语法分析。《A Retargetable C Compiler: Design and Implementation（可变目标C编译器——设计与实现）》是对 LCC的设计与实现的详细解释。

附 2：Bison 生成的分析器源码中使用的一些表及其含义

以下内容来自 Bison-2.4.1 源码中 src/tables.h 里的注释：

- YYTRANSLATE = vector mapping yylex's token numbers into bison's token numbers.
- YYTNAME = vector of string-names indexed by bison token number.
- YYTOKNUM = vector of yylex token numbers corresponding to entries in YYTNAME.
- YYRLINE = vector of line-numbers of all rules. For yydebug printouts.
- YYRHS = vector of items of all rules. This is exactly what RITEMS contains. For yydebug and for semantic parser.
- YYPRHS[R] = index in YYRHS of first item for rule R.
- YYR1[R] = symbol number of symbol that rule R derives.
- YYR2[R] = number of symbols composing right hand side of rule R.
- YYSTOS[S] = the symbol number of the symbol that leads to state S.
- YYDEFACT[S] = default rule to reduce with in state s, when YYTABLE doesn't specify something else to do. Zero means the default is an error.
- YYDEFGOTO[I] = default state to go to after a reduction of a rule that generates variable NTOKENS + I, except when YYTABLE specifies something else to do.
- YYPACT[S] = index in YYTABLE of the portion describing state S. The lookahead token's type is used to index that portion to find out what to do. If the value in YYTABLE is positive, we shift the token and go to that state. If the value is negative, it is minus a rule number to reduce by. If the value is zero, the default action from YYDEFACT[S] is used.
- YYPGOTO[I] = the index in YYTABLE of the portion describing what to do after reducing a rule that derives variable I + NTOKENS. This portion is indexed by the parser state number, S, as of before the text for this nonterminal was read. The value from YYTABLE is the state to go to if the corresponding value in YYCHECK is S.
- YYTABLE = a vector filled with portions for different uses, found via YYPACT and YYPGOTO.
- YYCHECK = a vector indexed in parallel with YYTABLE. It indicates, in a roundabout way, the bounds of the portion you are trying to examine. Suppose that the portion of YYTABLE starts at index P and the index to be examined within the portion is I. Then if YYCHECK[P+I] != I, I is outside the bounds of what is actually allocated, and the default (from YYDEFACT or YYDEFGOTO) should be used. Otherwise, YYTABLE[P+I] should be used.
- YYFINAL = the state number of the termination state.

- YYLAST (= high) the number of the last element of YYTABLE, i.e., sizeof (YYTABLE) - 1.

附 3：bison-examples.zip 的说明

本压缩包围绕 2 个小型编程语言：L-expr（简单的表达式语言）、L-asgn（赋值语句序列语言），给出如何利用 Flex 和 Bison 构造编译器，其中涉及程序位置跟踪、符号表的管理、抽象语法树的构造、错误信息管理等。

1、L-expr 语言的文法

```
input     ::=  ε | input line
line      ::=  EOL | expr EOL
expr      ::=  NUMBER
               | expr PLUS expr          # PLUS  – '+'加号
               | expr MINUS expr         # MINUS – '-'减号
               | expr MULT expr          # MULT  – '*'乘号
               | expr DIV expr           # DIV   – '/' 除号
               | MINUS expr              # MINUS – '-'负号
               | expr EXPON expr         # EXPON – '^'乘幂
               | LB expr RB              # LB, RB – '(', ')' 左右括号
```

2、L-asgn 语言的文法

```
input     ::=  ε | input line
line      ::=  EOL
               | asgnexp ';' EOL
asgnexp   ::=  IDENTIFIER ASGN exp      # ASGN  - '='赋值号
expr      ::=  NUMBER
               | IDENTIFIER              # IDENTIFIER – 标识符
               | expr PLUS expr          # PLUS  – '+'加号
               | expr MINUS expr         # MINUS – '-'减号
               | expr MULT expr          # MULT  – '*'乘号
               | expr DIV expr           # DIV   – '/' 除号
               | MINUS expr              # MINUS – '-'负号
               | expr EXPON expr         # EXPON – '^'乘幂
               | LB expr RB              # LB, RB – '(', ')' 左右括号
```

在 L-asgn 语言中，标识符代表变量，标识符需要先赋值再使用。如果程序使用了未赋值的变量，则需要给出警告并默认初值为 0。

3、bison-examples.zip 的文件目录结构说明

```
|- README 对本压缩包的说明文档
|- Makefile      用于构造语言的词法、语法分析器并将它们编译成可执行文件
|- run.sh        以./run.sh x y 命令启动 bin 子目录下的 x 编译器编译 test 子目录下的
                 y 程序
|- config        存放语言的词法和语法文法文件
   |- expr.lexL-expr 的 Flex 词法规范。
                 以-oexpr.lex.c 选项执行 flex，会根据该文件生成词法分析器 expr.lex.c
```

|- expr.y L-expr 的 Yacc 语法规范，它利用 Yacc 提供的优先级声明来消除文法的二义性。该语法规范内嵌有边分析边对表达式求值的语义动作。以-b expr -o expr.tab.c 选项执行 Bison，会根据该文件生成语法分析器 expr.tab.c 以及词法、语法分析都需使用的符号声明头文件 expr.tab.h

|- expr1.y L-expr 的 Yacc 语法规范，它通过增加文法非终结符，将文法改写成无二义的文法。可以用 Bison 处理该文件，生成 expr1.tab.h(.c)。该语法规范内嵌有边分析边对表达式求值的语义动作。

|- asgn.lex L-asgn 的 Flex 词法规范。

|- asgn.y L-asgn 的 Yacc 语法规范。该语法规范内嵌有边分析边对赋值语句求值的语义动作。

|- asgn1.y L-asgn 的 Yacc 语法规范，它与 asgn.y 的区别在于引入一个表示双目运算符的非终结符 op。

|- asgn2ast.y L-asgn 的 Yacc 语法规范。该语法规范内嵌有边分析边构造抽象语法树的语义动作。

|- asgn_err.lex L-asgn 的 Flex 词法规范，它与 asgn.lex 的区别在于引入统计尚未匹配的左括号数的全局量 lparen_num，以便在词法分析的过程中进行括号匹配跟踪。

|- asgn_err.y L-asgn 的 Yacc 语法规范。它与 asgn_err.lex 协作，支持对不合法的 L-asgn 程序的分析，提供错误恢复以及错误信息的记录并在分析结束时集中输出。

|- include 存放用户自定义的头文件

|-util.h 存放一些实用的宏、类型声明、函数声明，如动态分配相关的宏定义(如 NEW, NEW0)；线性表、线性表迭代器的数据结构的定义以及相关接口函数的声明。

|-common.h 编译器需要使用的一些公共的声明和定义，如符号表的结构、错误信息的结构、错误容器（或称错误工厂）的结构、语法树及其结点的结构定义，相关函数的声明等等

|-op.h 保存所处理的运算符及其对应的文本信息的配置文件

|-errcfg.h 保存所处理的错误类别及其对应的提示信息的配置文件

|- src 存放源程序文件

|-list.c 线性表的接口函数的实现（采用链式结构存储）、线性表迭代器 Iterator 的接口函数的实现

|-symtab.c 符号表的接口函数的实现（采用链地址结构的 Hash 表存储）

|-error.c 错误信息及其管理的接口函数的实现

|-ast.c 抽象语法树相关的接口函数的实现

|- expr.1ex.c, expr.tab.h, expr.tab.c, expr1.tab.c
利用 Bison 和 Flex 处理 config 目录下的 expr.lex, expr.y, expr1.y 生成的词法、语法分析器的源代码文件。

|- asgn.1ex.c, asgn.tab.h, asgn.tab.c, asgn1.tab.c, asgn2ast.tab.c
利用 Bison 和 Flex 处理 config 目录下的 asgn.lex, asgn.y, asgn1.y, asgn2ast.y 生成的词法、语法分析器的源代码文件。

|- asgn_err.1ex.c, asgn_err.tab.h, asgn_err.tab.c
利用 Bison 和 Flex 处理 config 目录下的 asgn_err.lex, asgn_err.y 生成的词法、语法分析器的源代码文件。

|- bin　　存放可执行文件
　|- expr, expr1, asgn, asgn1, asgn2ast, asgn_err
　　执行 make 或者 make expr (或 expr1, asgn, asgn1, asgn2ast, asgn_err) 后得到的 L-expr、L-asgn 语言的编译器的可执行文件
|- test　　存放测试程序
　|- expr.in　一个合法的 L-expr 程序
　|- asgn.in　一个合法的 L-asgn 程序
　|- asgn_err.in　一个不合法的 L-asgn 程序

4、如何使 Flex 和 Bison 生成的分析器能跟踪程序位置

bison-examples.zip中config/asgn.lex和asgn.y给出了如何使生成的分析器能跟踪程序位置的示例。这里简述要点：

1) Bison 生成的分析器缺省地用 YYLTYPE 结构类型（在生成的*.tab.h 中定义）来存储位置，其中包含 first_line、first_column、last_line、last_column 四个 int 域；并且引入 YYLTYPE 类型的全局变量 yylloc 保存当前记号的位置信息。
2) 在 asgn.lex 的声明段，增加如下代码：

```
%{int yycolumn = 1;

#define YY_USER_ACTION yylloc.first_line = yylloc.last_line = yylineno; \
    yylloc.first_column = yycolumn; yylloc.last_column = yycolumn+yyleng-1; \
     yycolumn += yyleng;
%}
%option yylineno
```

3) 在 asgn.lex 的规则段，在处理一个新行时增加对 yycolumn 修改的语义动作；
4) Bison会按缺省的位置跟踪方法为生成的分析器跟踪位置信息，见 Bison manual 3.6 Tracking Locations。在asgn.y的语义动作中，可以通过@$.first_line、@1.last_column 来访问位置信息， 其中@$表示产生式左部符号(LHS, left-hand side)对应的位置信息，@1 表示产生式右部符号(RHS, right-hand side) 对应的位置信息。

5、利用预编译指令灵活处理诸如错误类别等的可配置信息

在 bison-examples.zip中使用预编译指令灵活处理错误类别以及操作符等可配置的信息，并根据实际的配置信息生成相关类型和全局变量等。这里以操作符为例，简述处理方法：

1) include/op.h 为操作符的配置文件，其中每个操作符对应于如下形式的一行条目
   opxx(PLUS, " + ")
2) 在 include/common.h 中有如下代码段：

```
enum {
#define opxx(a, b) OP_##a,
#include "op.h"
        OPLAST
};
```

   该代码段将由 op.h 里的配置信息产生枚举常量，如 PLUS 对应的为 OP_PLUS
3) 在 src/ast.c 中有如下代码段：

```
char *opname[]={
#undef opxx
```

```
#define opxx(a, b) b,
#include "op.h"
        "Undefined Op"
};
```

该代码段将由 op.h 里的配置信息来初始化操作符名称数组 opname 的取值。

4) 用户可以在 op.h 中增加更多的操作符，而无须在修改枚举类型和 opname 的定义。

有关错误类型和对应的错误提示信息可以在 include/errcfg.h 中配置，在 include/common.h 和 src/error.c 中有相应的枚举类型、错误信息数组的定义。

6、引入宏 NEW 和 NEW0 来简化对各类结点的动态创建和清 0 操作的代码编写

```
#define NEW(p)     ((p) = malloc(sizeof *(p)))
#define NEW0(p)   memset(NEW(p), 0, sizeof *(p))
```

则若要构建结点类型为 A 的结点，则可以：

```
A *p, *q;
NEW(p);          // 创建 A 类型的结点，使 p 指向 A 类型的结点
NEW0(q);         // 创建 A 类型的结点并将所创建的结点清 0，使 q 指向该结点
```

7、引入宏 NEW 和 NEW0 来简化对各类结点的动态创建和清 0 操作的代码编写

2）引入设计模式来创建各类 AST 结点、错误信息结点等

2）引入“工厂模式”这一常用设计模式来创建各类 AST 结点、错误信息结点等