小型编译器框架构建与PL/0语言解释器实例化

——用户手册

nekko

PL/0语言解释器

PL/0编译器的编译

```
>>> g++ -o PL-0 PL-0.cpp -std=c++11
```

PL/0源程序的编译运行

```
>>> PL-0 [source file]
```

其中`[source file]`是源文件（应以`.pas`作为文件名后缀），即存储的标准PL/0源代码文件；在运行过程中会有`Lexer.out`等中间文件生成，用于观察编译结构。

标准PL/0语言文法

标准PL/0语言的文法定义如下（请**严格遵守**下列规则进行PL/0语言源代码的编写）：

```
program = block ".";
block = [ "const" ident "=" number {"," ident "=" number} ";"]
        [ "var" ident {"," ident} ";"]
        { "procedure" ident ";" block ";" } statement;
statement = [ ident ":=" expression | "call" ident
              | "?" ident | "!" expression
              | "begin" statement {";" statement } "end"
              | "if" condition "then" statement
              | "while" condition "do" statement ];
condition = "odd" expression |
            expression ("="|"#"|"<"|"<="|">"|">=") expression;
expression = [ "+"|"-"] term { ("+"|"-") term};
term = factor {("*"|"/") factor};
factor = ident | number | "(" expression ")";
```

标准PL/0语言的语法主要有以下几部分组成：

```
四则运算：+,-,*,/
不等关系：=,#(不等于),<,<=,>,>=
输入：? [变量]
输出：! [表达式]
定义过程：procedure [过程名]; [语句块];
语句块：[(可选)常量声明]; [(可选)变量声明]; 语句;
语句：选择语句、循环语句、读写语句、赋值语句、子程序调用语句、多语句块
注释：用一对配对的大括号包含，如：{ 这是一段注释 }
```

PL/0程序应以若干基本语句组成。一个样例程序（作用为读入两个变量、交换两个变量的值、输出）如下：

```
var a, b;
procedure swap;
var c; { temp variable for swap }
begin
    c := a;
    a := b;
    b := c
end;
begin
    ? a; ? b;
    call swap;
    ! a; ! b
end.
```

标准PL/0语言简明教程

PL/0的数据全部采用int类型，即**32位有符号整数**；所有的标识符（常量名、变量名和子程序名）应以下划线或者字母开头，并由下划线、字母或数字组成；在变量被赋值前，不保证变量的值为0（可能是任意随机数）；局部变量只有在子程序内部语句有效，局部变量优先级高于全局变量；如果重复定义局部变量或者全局变量，不保证程序能正确运行（`undefined behavior`）；子程序不可嵌套定义；源代码应当只出现数字、英文字母和必须的半角符号；源程序文件名请以`.pas`结尾，运行过程中会产生`Lexer.out`等中间文件。

主程序应分为四个部分组成，即**全局常量声明**、**全局变量声明**、**子程序声明**（零个或多个）和**主程序代码**。

全局常量声明的形式为：`const 常量1=值1, 常量2=值2, ...;`。

全局常量声明的形式为：`var 变量1, 变量2, ...;`。

子程序声明的形式为：

```
procedure 子程序名;
子程序局部常量声明;
子程序局部变量声明;
子程序代码;
```

相邻两条语句应以分号`;`分割（除了`end`前），在主程序结尾（即最后一条语句）后应加上句点`.`作为标识。

可以通过`变量名 := 表达式`来进行对变量的赋值。

可以通过`if 条件判断 then 执行语句`来执行`执行语句`（当`条件判断`为真时）。

可以通过`while 条件判断 do 执行语句`来执行`执行语句`若干次（只要`条件判断`为真时）。

可以通过`begin 语句1; 语句2; ... end`来执行多条语句。

可以通过`call 子程序名`来调用**已经定义过**的子程序。

可以通过`? [变量]`来读入一个int类型整数；可以通过`! [表达式]`来输出表达式所对应的值并换行（需要为int类型整数）。

可以通过两个配对的大括号来定义一段注释。

测试程序

下面的测试程序将输出 1 ～ 100 中的质数。

```
const max = 100;
var arg, ret;
procedure isprime; {declare procedure `isprime`}
var i;
begin
    ret := 1;
    i := 2;
    while i < arg do
    begin
        if arg / i * i = arg then
        begin
            ret := 0;
            i := arg
        end;
        i := i + 1
    end
end;
procedure primes;
begin
    arg := 2;
    while arg < max do
    begin
        call isprime;
        if ret = 1 then ! arg;
        arg := arg + 1
    end
end;
call primes.
```

客制化编译器框架

本编译器框架分为**分词**与**抽象语法树构建**两部分。

正则库

首先通过`Lexer lexer;`来声明一个分词库，可以通过`lexer.feed()`填入正则表达式和回调函数（处理匹配后的行为）。

通过**正则表达式**和**回调函数**，客制化匹配到分词后的行为（实现**分词功能**），一个可匹配整数和标识符的例子如下：

```
lexer.feed("[0-9][0-9]*",
           [&](string raw) {
               bpb("[number]", raw);
               tpb(TokenType :: number, raw);
           })
    .feed("[_a-zA-Z][_a-zA-Z0-9]*", // identifiers
          [&](string raw) {
              bpb("[ident]", raw);
              tpb(TokenType :: ident, raw);
          });
```

之后使用`lexer.match(src)`方法，传入字符串`src`对其进行分词匹配。

语法解析与抽象语法树生成

首先需要定义枚举`enum Nonterminal`来预定义非终结符号（应以`__Nonterminal_Count`结尾，用于统计非终结符号个数）；之后使用字符串数组`const char* symbols[]`来**依次**存储每一项非终结符号的对应名称。

之后可以使用`Parser parser;`来定义一个语法库，并通过`parser.getTerminalNode(val)`方法获取字符串`val`的终止节点；通过`parser.getNonterminalNode(name)`获取枚举`name`的非终止节点；通过`(Node*) -> add()`方法可以添加一条CFG规则。

通过自定义的`LL(1)`文法，可以录入文法表达式，进行分词结果的解析。

之后通过`Node* astroot = parser.match(parser.getNonterminalNode(program));`生成以`N(program)`为根节点的抽象语法树(AST)，下面是一个例子。

```
#define T(val) (parser.getTerminalNode(val))    // Terminal
#define N(name) (parser.getNonterminalNode(name)) // Nonterminal
N(statement) -> add({ N(ident), T(":="), N(expression) })
    -> add({ T("call"), N(ident) })
    -> add({ T("?"), N(ident) })
    -> add({ T("!"), N(expression) })
    -> add({ T("begin"), N(statement), N(statementpri), T("end") })
    -> add({ T("if"), N(condition), T("then"), N(statement) })
    -> add({ T("while"), N(condition), T("do"), N(statement) });
```

之后就可以得到抽象语法树，用户可以对`astroot`所代表的AST进行purify或进一步编译执行。