将 PL/0 编译器改造为 C0 编译器

PB09210183 何春晖

2012.03.24

1 C0 的词法

词法的名字以程序为准，从这里可以看出 PL/0 词法和 C0 词法间的对应关系。

1.1 关键字

```
ifsym  -> "if"
procsym  -> "void"
varsym  -> "int"
whilesym -> "while"
```

1.2 符号

```
becomes  -> "="
eql  -> "=="
neq  -> "!="
leq  -> "<="
lss  -> "<"
geq  -> ">="
gtr  -> ">"
plus  -> "+"
minus  -> "-"
times  -> "*"
slash  -> "/"
oddsym  -> "%"
lparen  -> "("
rparen  -> ")"
begsym   -> "{"
endsym   -> "}"
comma  -> ","
semicolon-> ";"
```

1.3 标识符

```
id -> [_a-zA-Z][a-zA-Z0-9]*
```

1.4 数字

```
number  -> num10 | num16 | num8
num10 -> [1-9][0-9]*|0
num16 -> 0[xX][0-9a-f]+
num8 -> 0[0-7]+
```

1.5 注释

```
comment -> \/\*[^\*]*\*(([^\*\/][^\*]*)?\*)*\/
```

2 C0 的语法和语义

为了语法清晰起见，固定符号的终结符用引号引用原文，而不用上节定义的名字。非终结符以大写起头。

按照扩展方式的语法表示如下：

```
Program -> Block
Block   -> [Vardecl] [Procdecl] Stmts
Vardecl -> "int" Vardef {, Vardef} ";"
Vardef  -> ident ["=" number]
ProcDecl-> "void" ident "(" ")" "{" Block "}"
Stmt    -> ident StmtOpt | "{" Stmts "}" |
           "if" "(" Cond ")" Stmt | "while" "(" Cond ")" Stmt | ";"
StmtOpt -> "=" Exp ";" | "(" ")" ";"
Stmts   -> Stmt Stmts | E
Cond    -> Exp CondOpt
CondOpt -> "%" "2" | RelOp Exp
RelOp   -> "=="|"!="|"<"|">"|"<="|">="
Exp     -> ["+"|"-"] Term {"+" Term | "-" Term}
Term    -> Factor {"*" Factor | "/" Factor}
Factor  -> ident | number | "(" Exp ")"
```

可见相比 PL/0 语言的主要区别有：

- Program 不以句点结尾，而是 **以文件到达结尾作为程序的结尾**；
- C0 的函数不能向 PL/0 一样只有一句，而 **必须以大括号包围**。因此 Block 中 Stmt 相应改为 Stmts，ProcDecl 中 Block 要有大括号包围；

- 合并了常量和变量的声明，**若声明时使用 ident “=”number 的模式，则认为 ident 是常量，若只用 ident，则认为是变量**；
- 分号不再只是语句间的分隔符，而要求 **每个语句必须以分号结束**，因此 Stmt 相应修改，并加上空语句“;”；
- **函数调用没有“call”前缀**，与赋值语句前缀相同，因此要做提左因子以满足 LL(1) 文法要求；
- **odd 函数用“%2”代替**，并提左因子；
- 其他细节，如左右括号。

至此，经过文法相应修改，PL/0 文法已经基本变为 C0 的风格。但实践上还有两点不同：

- **C0 不像 PL/0 可以过程嵌套定义**；
- **PL/0 最外层的 Stmt 块在 C0 中没有对应成分，而 C0 中 main 函数在 PL/0 中没有对应成分**。

针对这两点不同，将问题简单化，并没有选择继续修改文法，而是在语义动作上做修改。

针对第一点，因为在 block 函数中，用 lev 来判断嵌套层次。因此，在递归调用 block 前判断 lev，若进入嵌套则报语义错。

针对第二点，视 main 为普通函数定义。然后基于同样原理，判断 lev，若处于最外层，则自动加上一个 Stmts 的机器代码。代码的内容是调用 main 函数。若找不到 main 函数，则报语义错。这样就满足了 main 函数首先调用的要求，也隐去了 PL/0 中最后一段 Stmts 成分。

这样 PL/0 语言成分就对应到了 C0 上。

2.1 中间代码

2.1.1 符号表组织和结构

符号表目前定义在 parser.c 中，叫 table，结构各个元素意义为：

- name：符号名
- kind：符号类型，有常量、变量和过程
- val：常量的值
- level：符号的层次
- addr：过程或变量的地址

tx 是 **当前层次上** 符号表第一个未分配的表项下标。

对符号表的操作有：

- 登记新符号：将指定类型的当前符号顺序存入符号表，使用 enter 函数完成
- 查找符号：查找符号在当前层次对应的符号表项，使用 position 函数完成

- 进入和退出层次：此工作是在 block 函数中实现的，进入一层时，置 tx1=tx 保存；退出一层时，置 tx=tx1 恢复

可见，对符号表是利用类似栈的结构进行管理的。这种管理保证了局部的符号不会在其他地方被引用。

2.1.2 中间代码的组织

中间代码是全局量，定义在 glo.h 中，是一个叫 code 的指令数组，里面存放翻译的中间代码。代码生成还依赖于两个全局量 cx 和 dx。

- cx 是 code 中当前要生成的指令的下标
- dx 是在当前层次堆栈中第一个可分配数据的下标

其中 dx 也要根据分析层次的进出类似 tx 一样地调整，以完成局部变量的分配。

对 code 进行生成的函数是 gen，在 parser.c 中，此函数仅仅简单将指令的操作码、层次和地址顺序登入 code 中。

2.1.3 中间代码的生成

表达式和条件表达式

- 常量：先从符号表中查出数值，再生成 lit，载入值到栈顶。
- 变量：先从符号表中查出地址偏移 a 和层次 l，生成 lod，载入 base(l)+a 处的值到栈顶。
- 数值：直接生成 lit，载入值到栈顶。
- 运算符：生成 opr 指令对顶部元素计算。

语句

- 赋值：先产生右部表达式代码，再从符号表查处变量的偏移和层次，生成 sto 指令。
- 函数（过程）调用：先从符号表中查出地址偏移和层次，生成 cal 指令。
- 条件和循环：先生成条件表达式的代码，再生成 jpc、jmp 和语句的代码。

过程块

- 开始：生成一个空 jmp 指令，用来跳转到首过程代码处执行。
- 变量定义：生成 int 指令，在运行时栈上分配空间。**由于用 cal 指令调用函数时，该指令会在栈上存下动态链信息、返址和静态链信息，因此生成的 int 指令至少要开辟 3 的空间**，因此 dx 初值赋值为 3。
- 过程定义：递归地做过程块的代码生成，并维护 lev、tx、dx 变量。
- 代码开始：修改前面生成的空 jmp 指令，使目标地址指向此处。
- 代码结束：生成 opr(0,0) 指令返回

2.2 错误处理

按照《PL/0 编译器的错误与错误恢复》一文中的改进方案设计错误处理。

2.2.1 修改 test 语义

在 parser.c 中：修改 test 定义为 int test(s1, s2)。

意义为：若 sym 在 s1 中，吃掉 sym，返回 1；否则报错，并向后同步扫描到 s1 或 s2 中，返回 0；若同步到 s1 中，吃掉 sym，若同步到 s2 中，保持 sym 不被吃掉。

经过这样的修改后，代码不少部分得以简化。并且所有的语法错均由 test 函数检测，因此不必附加错误号。

2.2.2 增加 miss_error 函数

针对原文中提出的报错信息的修改，在 lexer.c 中增加 miss_error 函数，可以向 miss_error 函数传递一个集合，表示编译器期望接收到的符号集合。

2.2.3 去除 Follow 集合检测

由代码清晰性的要求，去除了所有 Follow 集合的检验，即所有 fsys 参数被删除。

对 Follow 的检测改由调用者显式调用 test 函数检测，而非被调用者经由 fsys 间接检测。

2.2.4 结果

以遗漏分号的例子检测：

```
void main()
{
    int a, b;
    a=3
    b=4;
}
```

报错为：

```
 ****     ^ 0
errmsg(0): miss ';' token
```

显然友好不少。