语义分析器

周炎亮 2018202196

一、实验目的

结合前两个词法分析器和语法分析器，实现语义分析器，对PL/0代码及自定义的扩展部分编译生成类pcode语句，并利用interpret对其解释执行。（程序执行方法见README.md）

二、扩展部分介绍

- 常量的值**不可修改**（及对应标识符只能被引用而无法被修改）
- 变量的数据类型支持**整型**、**字符型**（单个字符）和**浮点型**，支持相应运算。
- 变量可支持**一维数组**，数组定义的形式为“标识符(表达式:表达式)”，引用的形式为“标识符(表达式)”，其中表达式只能为**整型**。
- 浮点型支持**科学计数法**。
- 字符变量在存储时以**ascii码**的形式存储，因此可将字符类型和整型相互切换。但字符型和实型、整型和实型之间无法相互切换。
- 支持过程嵌套。
- 支持**for循环语句**，形式为“for 赋值语句 to/downto 表达式 [step 表达式] do 语句”，to表示变量不能减小，downto表示变量不能增大。其中表达式和赋值语句可支持**实型**。且其中表达式在编译时已计算出结果，即作为**常量**，若表达式中含有变量且值在for循环过程中该边，则不影响该表达式值。
- 支持**repeat循环语句**，形式为“repeat 语句 until 条件表达式”。
- read和write语句支持**列表**，即read和write语句中可有多个表达式或标识符，以逗号分隔。
- 支持扩展else的条件语句，形式为“if 条件 then 语句 else 语句”

三、EBNF范式

```
<程序> ::= <分程序>.
<分程序> ::= <说明部分><语句>
<说明部分> ::= <常量说明部分>|<常量说明部分><变量说明部分>|<常量说明部分><过程说明部分>|<常量说明部分><变量说明部分><过程说明部分>|<变量说明部分><过程说明部分>|<变量说明部分>|<过程说明部分>
<常量说明部分> ::= CONST<常量定义集>;
<常量定义集> ::= <常量定义集>,<常量定义>|<常量定义>
<常量定义> ::= IDENTIFIER:=INTEGER
<变量说明部分> := <变量说明部分>VAR<var变量说明部分集>;|<变量说明部分>CHAR<char变量说明部分集>;|<变量说明部分>REAL<real变量说明部分集>;|VAR<var变量说明部分集>;|CHAR<char变量说明部分集>;|REAL<real变量说明部分集>;
<var变量说明部分集> := <var变量说明部分集>,IDENTIFIER|<var变量说明部分集>,IDENTIFIER'('<表达式>,<表达式>')'|IDENTIFIER'('<表达式>,<表达式>')'|IDENTIFIER
<char变量说明部分集> := <char变量说明部分集>,IDENTIFIER|<char变量说明部分集>,IDENTIFIER'('<表达式>,<表达式>')'|IDENTIFIER'('<表达式>,<表达式>')'|IDENTIFIER
<real变量说明部分集> := <real变量说明部分集>,IDENTIFIER|<real变量说明部分集>,IDENTIFIER'('<表达式>,<表达式>')'|IDENTIFIER'('<表达式>,<表达式>')'|IDENTIFIER
<过程说明部分> ::= <过程说明部分><过程首部><分程序>;|<过程首部><分程序>;
<过程首部> ::= PROCEDURE IDENTIFIER;
<语句集> ::= <语句集>;<语句>|<语句>
<语句> ::= <赋值语句>|<复合语句>|<条件语句>|<当型循环语句>|<repeat型循环语句>|<for型循环语句>|<过程调用语句>|<读语句>|<写语句>|<空>
```

```
<赋值语句> ::= IDENTIFIER:=<表达式>|IDENTIFIER'('<表达式>')':=<表达式>
<复合语句> ::= BEGIN<语句集>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>|IF<条件>THEN<语句>ELSE<语句>
<表达式> ::= <表达式>+<项>|<表达式>-<项>|+<项>|-<项>|<项>
<项> ::= <项>*<因子>|<项>/<因子>|<因子>
<因子> ::= '('<表达式>')'|<标识符>|IDENTIFIER'('<表达式>')'|INTEGER|FLOAT|STRING
<关系运算符> ::= =|#|<|<=|>|>=
<当型循环语句> ::= WHILE<条件>DO<语句>
<repeat型循环语句> ::= REPEAT<语句>UNTIL<条件>
<for型循环语句> := FOR<赋值语句>TO<表达式>STEP<表达式>DO<语句>|FOR<赋值语句>TO<表达式
>DO<语句>|FOR<赋值语句>DOWNTO<表达式>STEP<表达式>DO<语句>|FOR<赋值语句>DOWNTO<表达式
>DO<语句>
<过程调用语句> ::= CALL IDENTIFIER
<读语句> ::= READ'('<标识符集>')'
<标识符集> ::= <标识符集>,<标识符>|<标识符>
<写语句> ::= WRITE'('<表达式集>')'
<表达式集> ::= <表达式集>,<表达式>|<表达式>
```

此EBNF范式是在之前语法分析器实验的基础上，结合本次语义实验的修改版。其中如变量说明部分定义了三种不同的变量说明集，看似一样，但其实是为了方便编译时确定变量的类型。

四、类pcode代码

由于增加了扩展部分，因此对输出的类pcode代码及interpret函数进行了较大修改。

- 对于LOD指令，将其扩展为LODI和LODF，分别表示读入的数据的类型为整型（因为字符以ascii码存储，故也可作为整型读入）或浮点型。同理，STO也扩展为STOI和STOF。
- 对于OPR的1-13的a值（对应不同的数值操作），将这些操作中原本无关的l值改为0、2代表对整型或字符型的操作，1代表对浮点型的操作。
- 对于OPR中a为14的写语句和16的读语句，也根据l值的不同输出或输入相应的类型的值。
- 对于JPC语句，也对l增加意义，当l为0时，与原始一致，即栈顶值非真时跳至a语句；当l为1时，则相反，即栈顶值为真时跳至a语句。
- 增加了I2R和R2I指令，即将整型转为浮点型和将浮点型转为整型。对应的a为0时，表示对栈顶元素进行该操作；a为1时，表示对次栈顶元素进行该操作。

对于interpret，将原始int类型的l修改为`enum {Int, Real, Char}`，将原始int类型的a修改为同时存放整型、浮点型和字符型的union结构DATA（便于LIT读取浮点型数据）。根据上述思路对解释部分函数进行相应修改。

五、代码说明

- 使用了display表的方式维护符号表，其中记录对应标识符的类型，名称，值，所在层次，偏移量，指向的下一个标识符的地址，是否为数组，数组下界及数组上界。
- 语义分析中部分符号，定义如下union结构储存相应数据：

```
%union{
      struct{
            union{
                  int val;
                  char str;
                  float real;
            }data;
            int kind;   //int 0, real 1, char 2
            char *str;
      }var;
}
```

- 每次对标识符进行定义或引用时，会先调用if_declared函数判定标识符是否已被定义，进行相关判断后在调用find_sign函数确定标识符位置。
- 对于while、for以及repeat循环语句，使用教材上相应回填方法进行分析，对于每个语句分别定义一个地址表记录该跳转的地址。

报错处理

出现以下情况时，编译器会报错（此处省略不符合语法的情况）

- 数组下标为实型
- 定义标识符时，该标识符已被定义
- 引用的标识符未被定义
- 对常量标识符赋值
- 数组越界
- ODD对实型进行操作
- 对常量标识符进行读操作
- 引用的过程名未被定义

六、实验结果截图

```
var n,number;
procedure gcd;

    if n>1 then
        begin

            number := number * (n - 1);
            n := n - 1;
            call gcd;
        end;
begin
    read(n);
    number := n;
    call gcd;
    write(number);
end.
```

```
(base) triode@DESKTOP-SCDSMTP:/mnt/f/课件/_大二下/编译原理/语义分析器/new$ ./a.out < recursion.txt
(base) triode@DESKTOP-SCDSMTP:/mnt/f/课件/_大二下/编译原理/语义分析器/new$ ./interpret
6
720
```

扩展部分：

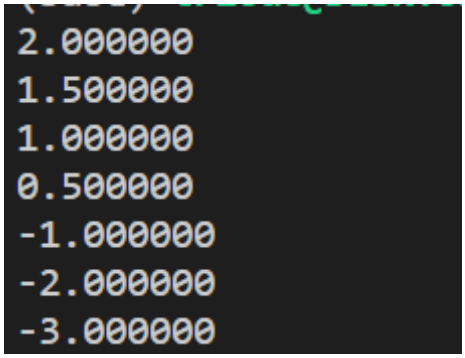
```
const cc=1;
var a,i;
real b,c;

begin

    for b := 2 downto 0.5 step -0.5 do
        write(b);

    repeat
    begin
        b := b - 1;
        write(b);
    end
    until b < -2;

end.
```




```
const cc=1;

char d;
char e(1:3);
var f(2:3);
begin

    read(d);
    write(d);

    d := "b";
    write(d);

    e(1) := 66;
    e(2) := 67;
    write(e(1),e(2));

    write(cc);
end.
```

七、总结

优点：

- 在要求的基础上，对for语句增加了支持步长乃至浮点类型。
- read和write函数支持多参数。

不足：

- 只能实现一维数组而不支持更高维度。
- 对于数组元素，因为编译时已将数组下标的表达式计算出值存在pcode代码中，因此不支持循环调用数组元素。

通过语义实验，在前两次实验的基础上，将整个编译器的三个部分综合在了一起，进一步加深了编译器的尤其是语义生成中间代码部分的理解，也熟悉了类 pcode 代码和解释器运行的简单实现方式。掌握了自己实现编译器的方式，完成了自定义语法的编译实现，对编译器工作和实现方式有了初步的理解和实践经验。