编译原理实验-语义分析与中间代码生成

17341146 王程钥

1 简介

1.1 实验目的

构造TINY + 的语义分析程序并生成中间代码

1.2 实验内容

构造符号表，用C语言扩展TINY的语义分析程序，构造TINY + 的语义分析器，构造TINY + 的中间代码生成器

1.3 实验要求

能检查一定的语义错误，将TINY + 程序转换成三地址中间代码。

2 构造符号表

2.1 类定义

首先给出符号(Symbol)的类定义，包括符号名，符号类型和符号存储位置。本次实验暂时不支持递归，测试程序只包含一个main函数，所以不存储符号的辖域。

```
typedef struct Symbol
{
    SymType _type;
    char* _name;
    int _loc;
}Symbol;
```

符号类型包括以下6种（若类型暂时未知则定义为Unknown）

```
typedef enum SymType{
    _int, _char, _string, _float, _bool, _Unknown
}SymType;
```

定义符号表(Symtab)数据结构。

```
typedef struct Symtab
{
    Symbol* sym[SYMSIZE];
    int TabSize;
    int CurLoc;
}Symtab;
```

2.2 构造符号表

递归一遍语法分析生成的语法树，对于所有的终端节点，如果它们对应的token类型是key, number, char, string这几种，则它是一个变量，将其加入变量表。变量的存储位置默认从0开始计算，根据数据类型一次相加。
以下为构造符号表的主程序。

```
bool _GenSymTab(TrieNode *CurNode, Symtab *Table, SymType _type)
{
    char* NodeName = CurNode->NodeName;
    int _start = 0;
    Symbol* ret;
    if(!strcmp(NodeName, "Vars") || !strcmp(NodeName, "ForMalParams")){
        _type = GenType(CurNode->Child[0]->NodeName);
    }
    if(!strcmp(NodeName, "Func")) _start = 2;

    if(CurNode->NodeToken!=NULL){
        if(CurNode->NodeToken->type==_Key){
            if(_type==_Unknown){
                ret = SymbolExist(Table, CurNode->NodeName);
                if(ret==NULL){
                    char str[100];
                    sprintf(str, "name '%s' is not defined", CurNode->NodeName);
                    SetErrorToken(NameError, str, CurNode->NodeToken);
                    return false;
                }
                else _type = ret->_type;
            }
            else{
                InsertNewSymbol(Table, _type, CurNode->NodeName);
            }
        } else if(CurNode->NodeToken->type==_Number){
            _type = GetNumberType(CurNode->NodeName);
        } else if(CurNode->NodeToken->type==_Char){
            _type = _char;
        } else if(CurNode->NodeToken->type==_String){
            _type = _string;
        }
        SetNodeType(CurNode, _type);
    }

    for(int i=_start;i<CurNode->ChildSize;i++)
        if(!_GenSymTab(CurNode->Child[i], Table, _type)) return false;

    if(CurNode->NodeToken==NULL)
        CurNode->NodeToken = CurNode->Child[0]->NodeToken;

    return true;
}
```

另外，生成中间代码会产生一些临时变量(e.g. t0, t1)，这些变量也会加入符号表。

3 类型检测

类型检测主要做两种判断。

1. 一个运算符两边的变量/常数类型是否相同。（本次实验的编译器不支持自动类型转换）
2. 赋值符号两边的数据类型是否相同。

为了完成类型检测，需要对语法树的每个节点加存一个当前节点的数据类型（NodeType）。以下为拓展定义的TrieNode。

```
struct trie
{
    TrieNode* Child[MAXCHILD];
    int ChildSize;
    char* NodeName;
    Token* NodeToken;
    SymType NodeType;
};
```

同样地，对语法树进行一遍dfs即可。以下为类型检测的相关代码。

```
bool CheckType(TrieNode *CurNode, Symtab *Table)
{
    for(int i=0;i<CurNode->ChildSize;i++)
        if(!CheckType(CurNode->Child[i], Table)) return false;
    // dfs
    if(!strcmp(CurNode->NodeName,"Exp0") || \
       !strcmp(CurNode->NodeName,"Exp1") || \
       !strcmp(CurNode->NodeName,"Exp2") || \
       !strcmp(CurNode->NodeName,"Expression") || \
       !strcmp(CurNode->NodeName,"Assignment") ) {
        if(CurNode->ChildSize>=3){
            if(!strcmp(CurNode->NodeName,"Exp2")) {
                CurNode->NodeType = CurNode->Child[1]->NodeType;
            }
            else if(!strcmp(CurNode->NodeName,"Expression")) {
                CurNode->NodeType = _bool;
            }
            else {
                if(CurNode->Child[0]->NodeType != CurNode->Child[2]->NodeType) {
                    SetErrorToken(TypeError, "type not equal", CurNode->NodeToken);
                    return false;
                }
                CurNode->NodeType = CurNode->Child[0]->NodeType;
                if(!strcmp(CurNode->NodeName,"Exp0") || \
                   !strcmp(CurNode->NodeName,"Exp1") ) {
                    InsertNewSymbol(Table, CurNode->NodeType, CurNode->Place);
                }
            }
        }
        else {
            CurNode->NodeType = CurNode->Child[0]->NodeType;
        }
    }
    // exp0 exp1 exp2 expression
    return true;
}
```

4 中间代码生成

4.1 Place的定义

对于形如 $x := (y + z) * h$ 的表达式，使用三地址中间代码显然无法表示。所以需要引入一些中间变量。比如对于上式，可以用以下三地址码表示。

```
t0 := y + z
t1 := t0 * h
x := t1
```

我们定义中间变量Place。什么时候需要中间变量呢?

1. 对于常数，直接代入即可。
2. 对于变量Key，直接代入即可。
3. 对于表达式Expression，每步运算生成一个临时变量，临时变量用t0, t1, t2...表示。

因此再拓展定义TrieNode，加入Place。

```
struct trie
{
    TrieNode* Child[MAXCHILD];
    int ChildSize;
    char* NodeName;
    Token* NodeToken;
    SymType NodeType;
    char* Place;
};
```

4.2 中间代码生成

4.2.1 IF语句

IF语句的代码格式是这样的

```
if x op y then
   ...
else
   ...
endif
```

IF语句的中间代码格式是这样的

```
L1 : If x op y goto L3
L2 : IfFalse x op y goto L5
L3 : ...
L4 : goto L6
L5 : ...
L6 : sth Next
```

如果没有else，则可以去掉L4和L5。根据以上思路，可以写出以下代码。

```
if(!strcmp(_name, "IfStmt")) {
    TrieNode* Exp = CurNode->Child[1];
    CodeGen(Exp->Child[0]);
    CodeGen(Exp->Child[2]);
```

```
    // if
    A = Exp->Child[0]->Place;
    B = Exp->Child[1]->NodeName;
    C = Exp->Child[2]->Place;
    sprintf(str, "%d: If %s %s %s, goto %d", Lineno, A, B, C, Lineno+2);
    AddMidCode(str, Lineno); Lineno+=2;
    int PreLine = Lineno-1;

    // then
    for(int i=3;i<CurNode->ChildSize;i++){
        CodeGen(CurNode->Child[i]);
    }

    // else
    if(!strcmp(CurNode->Child[4]->NodeName, "else")){
        int IfLine = Lineno; Lineno++;
        sprintf(str, "%d: IfFalse %s %s %s, goto %d", PreLine, A, B, C,
Lineno);
        AddMidCode(str, PreLine);
        CodeGen(CurNode->Child[5]);
        sprintf(str, "%d: goto %d", IfLine, Lineno);
        AddMidCode(str, IfLine);
    }
    else {
        sprintf(str, "%d: IfFalse %s %s %s, goto %d", PreLine, A, B, C,
Lineno);
        AddMidCode(str, PreLine);
    }
}
```

4.2.2 循环语句

循环语句的代码格式是这样的

> repeat
>
> ...
>
> until x op y

循环语句的中间代码格式是这样的

> L1 : ...
> L2 : if x op y goto L1
> L3 : sth Next

根据以上规则，得到的相关代码如下。

```
if(!strcmp(_name, "RepeatStmt")) {
    // repeat
    int PreLine = Lineno;
    CodeGen(CurNode->Child[1]);
    // until
    TrieNode* Exp = CurNode->Child[3];
    CodeGen(Exp->Child[0]);
    CodeGen(Exp->Child[2]);
    A = Exp->Child[0]->Place;
    B = Exp->Child[1]->NodeName;
        C = Exp->Child[2]->Place;
```

```
    sprintf(str, "%d: If %s %s %s, goto %d", Lineno, A, B, C, PreLine);
    AddMidCode(str, Lineno++);
}
```

4.2.3 表达式

找到语法树上存在多个孩子的节点，对于每个这样的节点可以产生一个形如 $x := yopz$ 的表达式。其中x, y, z分别为当前节点及其两个孩子的Place。相关代码如下。

```
void GenExp(TrieNode *CurNode)
{
    CodeGen(CurNode->Child[0]);
    CodeGen(CurNode->Child[2]);
    char* P = CurNode->Place;
    char* A = CurNode->Child[0]->Place;
    char* B = CurNode->Child[1]->NodeName;
    char* C = CurNode->Child[2]->Place;
    char str[100];
    sprintf(str, "%d: %s := %s %s %s", Lineno, P, A, B, C);
    AddMidCode(str, Lineno++);
}
```

4.2.4 其它

其它语句的中间代码生成比较简单，在此略去。具体可见$cgen.h$。

5 实验结果

5.1 正确结果

以下为测试代码 $code.tny$

```
int main ()
    int x , y , fact;
    string str ;
    char ch ;
    x := 5 ;
    y := ( 2 + 4 ) * 5 ;
    str := "STR1!!!!" ;
    ch := 'a';

    if 0 < x then // don't compute if x <= 0
        fact := 1;
        repeat
            fact := fact * x;
            x := x - 1 ;
        until x == 0
        write fact ; // output factorial of
    else
        x := 1;
    endif

    write str ;
```

以下为符号表

```
Symbol table:
  Name   |   Type   |  Location
=========|==========|==========
       x |      int |         0
       y |      int |         4
    fact |      int |         8
     str |   string |        12
      ch |     char |        15
      t0 |      int |        16
      t1 |      int |        20
      t2 |      int |        24
      t3 |      int |        28
```

以下为生成的中间代码

```
MidCode:
0: x := 5
1: t0 := 2 + 4
2: t1 := t0 * 5
3: y := t1
4: str := "STR1!!!!"
5: ch := 'a'
6: If 0 < x, goto 8
7: IfFalse 0 < x, goto 16
8: fact := 1
9: t2 := fact * x
10: fact := t2
11: t3 := x - 1
12: x := t3
13: If x == 0, goto 9
14: OUT fact
15: goto 17
16: x := 1
17: OUT str
```

5.2 编译错误展示

变量再声明之前使用

测试代码

```
int main ()
    int x;
    x := x + y ;
```

编译器输出

```
Generate Symbol Table Fail.
code1.tny:3:11 Name Error: name 'y' is not defined
```

变量y在声明前使用，编译错误。

符号类型不匹配

测试代码

```
int main ()
    int x;
    x := "string" ;
```

编译器输出

```
Type Check Fail.
code1.tny:3:2 Type Error: type not equal
```

将字符串赋值给int，编译错误。