语义分析程序及中间代码生成实验报告

17341092 梁萍佳

一、实验目的

构造TINY + 的语义分析程序并生成中间代码

二、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

三、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

四、实验过程

下面首先将给出完整的属性文法，然后根据属性文法来完成中间代码生成，从而完成利用语法分析树生成三地址中间代码的生成程序。

1、属性文法

下面给分析树的节点加上属性，其中各个属性的定义见下表：

属性名	描述	属性类型
code	表示该节点翻译出来的中间代码	综合属性
begin	表示在该节点的代码段开头的位置	综合属性
next	表示在该节点的代码段之后的第一行代码位置	继承属性
true	供布尔表达式用，表示条件为真时跳转的位置	继承属性
false	供布尔表达式用，表示条件为假时跳转的位置	继承属性
var	供表达式用，表示该节点对应的变量名	综合属性

各产生式与其对应语义规则如下，标识序号用来方便在代码中定位该语义规则对应的代码段。

其中declare函数为在符号表中记录标识符及其对应类型，其中会进行重定义的检测；check函数用于检查标识符是否为预期数据类型；newlabel会得到一个新的标记；newTvar会得到一个新的临时变量。

标识记号	产生式	语义规则
-	program->Decls Seq	program.next=newlabel; Seq.next=program.next ;program.code= Seq.code;
d1	decl->type-specifier varlist	varlist.type=type-specifier.type
d2	type-specifier->int|bool|char	type-specifier.type=int|bool|char
d3	varlist-> identifier , varlist1	decalre(identifier,varlist.type); varlist1.type=varlist.type
d4	varlist->identifier	decalre(identifier,varlist.type);
s1	Seq->S;Seq1	S.next=newlabel; Seq1.next=Seq.next; Seq.code=S.code
s1	Seq->S	S.next=Seq.next Seq.code=S.code
s2	S->repeat Seq until E	S.begin=newlabel; Seq.next=S.begin; E.true=S.next; E.false=S.begin; S.code=Label S.begin||Seq.code||E.code;
s3	S->while E do Seq end	S.begin=newlabel; Seq.next=S.begin; E.true=newlabel; E.false=S.next; S.code=Label S.begin || E.code || Label E.true || Seq.code || goto S.begin
s4	S->if E then Seq end	E.true=newlabel; E.false=S.next Seq.next=S.next; S.code=E.code|| Label E.true || Seq.code
s5	S->if E then Seq1 else Seq2 end	E.true=newlabel;E.false=newlabel;Seq1.next=S.next; Seq2.next=S.next;S.code=E.code || Label E.true || Seq1.code || goto S.next || Label E.false || Seq2.code
s6	S->identifier:=E	check(identifier.type,E.var.type);S.code= E.code || identifier:=E.var
s7	S->read identifier	S.code=read || identifier
s8	S->write E	S.code=E.code || write E.var
e1	E-> arith-exp	E.code=arith-exp.code; E.var=arith-exp.var; E.var.type=int;
e1	E->bool-exp	bool-exp.true=E.true; bool-exp.false=E.false E.var.type=bool; E.code=bool-exp.code; E.var=bool-exp.var;
e1	E->string-exp	E.var=string-exp.var; E.code=""; E.var.type=char
e2	arith-exp->term addop arith-exp1	arith-exp.var=newTvar;arith-exp.code= term.code || arith-exp1.code || arith-exp.var:=term.var addop arith-exp1.var;
e2	arith-exp->term	arith-exp.var=term.var; arith-exp.code=term.code
e3	term->factor mulop term1	term.var=newTvar;term.code= factor.code || term1.code || term.var:=factor.var mulop term1.var
e3	term->factor	term.var=factor.var; term.code=factor.code
e4	factor->(arith-exp)	factor.var=arith-exp.var; factor.code=arith-exp.code
e4	factor->number	factor.var=number; factor.code=""
e4	factor->identifier	check(identifier,int); factor.var=identifier; factor.code=""
e5	bool-exp->bterm or bool-exp1	bool-exp.var=newTvar; bterm.true=bool-exp.true;bterm.false=newLabel;bool-exp.code= bterm.code || Label bterm.false ||bool-exp1.code
e5	bool-exp->bterm	bterm.true=bool-exp.true;bterm.false=bool-exp.false;bool-exp.var=bterm.var;bool-exp.code=bterm.code
e6	bterm->bfactor and bterm1	bfactor.false=bool-exp.false;bfactor.true=newLabel;bterm.var=newTvar;bterm.code= bfactor.code || Label bfactor.true || bterm1.code
e6	bterm->bfactor	bfactor.true=bterm.true;bfactor.false=bterm.false;bterm.var=bfactor.var;bterm.code=bfactor.code
e6	bfactor->com-exp	bterm.var=com-exp.var;com-exp.true=bterm.true;com-exp.false=com-exp.false
e7	com-exp->arith-exp1 com-op arith-exp2	com-exp.var=newTvar;com-exp.code= arith-exp1.code || arith-exp2.code || if arith-exp1.var com-op arith-exp2.var goto com-exp.true || goto com-exp.false

实际程序的实现中会与上表有些许差别，例如bfactor直接用com-exp来代替了，string部分也做了比较简单化的处理。

2、翻译程序

下面首先将介绍程序中的数据结构定义，然后给出几个辅助函数，并给出一个语义规则对应的代码作为例子。

(1)数据定义

新的节点定义比语法分析的节点定义中多出了语义规则中的属性，同时有一个gen_code()方法，可以递归下降地生成中间代码。此外定义了tiny语言里面的三个数据类型，以及记录标识符的符号表。

```
//变量类型
enum var_type{t_int,t_char,t_bool};
//符号表，记录标识符及其类型
map<string,var_type> sym_table;
map<string,int> sym_line;
```

```
struct nTreenode{
    syntax_symbols symbol;
    vector<nTreenode> subtree;
    Token token;//当这个是叶子节点时就是token了
    void print_code();
    void print_tree(int dep);
    int line;//节点所在行数

    string var;
    var_type type_of_var;//type已经被前面占用了
    int begin;
    int next;
    int t_true;
    int t_false;
    vector<string> code;
    void gen_code();
};
```

(2)辅助函数

由语法分析得到的语法分析树要将其节点转换成带属性的节点，为此实现了一个transfrom函数，输入语法分析树根节点后会返回转换后的分析树根节点。

```
//将普通语法分析树转换成带属性的分析树
nTreenode transfrom(Treenode& src){
    nTreenode retnode;
    retnode.symbol=src.symbol;
    retnode.token=src.token;
    retnode.line=src.line;
    for(int i=0;i<src.subtree.size();i++){
        retnode.subtree.push_back(transfrom(src.subtree[i]));
    }
    return retnode;
}
```

此外，在语义规则中用到的几个函数如下：

```
//检查标识符
void check(string var_name,var_type type,int line){
    if(sym_table.count(var_name)==0){//未定义错误
        cout<<"Error at line "<<line<<" :"<<var_name<<" is not defined."<<endl;
        exit(0);
    }else if(sym_table[var_name]!=type){//类型错误
        string tstr[]={"'int'","'char'","'bool'"};
        cout<<"Error at line "<<line<<" :"<<" expect type "<<tstr[type];
        cout<<",but "<<var_name<<"'s type is "<<tstr[sym_table[var_name]];
        exit(0);
    }
}

//获取新的label
int newlabel(){
    static int label_cnt=0;

    cout<<"get a label "<<label_cnt<<endl;
    return label_cnt++;
```

```
}
//获取新的临时变量
string newTvar(var_type type,int line){
    static int label_cnt=0;
    char buf[10];
    do{
        label_cnt++;
        sprintf(buf,"t%d",label_cnt);
    }while(sym_table.count(buf)>0);

    declare(buf,type,line);
    cout<<"get a temp variable t"<<label_cnt<<endl;
    return string(buf);
}
```

最后，在属性计算过程中，经常有合并代码段，添加label，goto语句的需求，因此也实现了相应的几个函数来减少gen_code()中的代码量。

```
//合并代码的
void joint(vector<string>& to,vector<string>& b){
    for(int i=0;i<b.size();i++){
        to.push_back(b[i]);
    }
    b.clear();//以后都用不上了，清空以减少储存消耗
}
//将序号转换成label代码
string label_str(int label){
    char buf[20];
    sprintf(buf,"Label L%d",label);
    cout<<"placing "<<buf<<endl;
    return buf;
}
//将序号转换成goto 语句
string goto_str(int label){
    char buf[20];
    sprintf(buf,"goto L%d",label);
    cout<<"placing "<<buf<<endl;
    return buf;
}
```

(3)生成代码

由于这部分代码较多，且基本上就是按照语义规则来实现，因此只给出其中一个例子来展示。规则s2的产生式为S->repeat Seq until E；而其对应的语义规则为 S.begin=newlabel;　Seq.next=S.begin;　E.true=S.next;　E.false=S.begin;　S.code=Label S.begin||Seq.code||E.code; 而这部分对应代码如下，可以看出事实上基本上代码与语义规则差不多是一一对应的。

```
        //----------
        if(son.symbol==repeat_stmt){//S->repeat Seq until E
            if(subtree.size()!=4){
                cout<<"program bug! at repeat_stmt"<<endl;
                exit(0);
            }
            begin=newlabel();//S.begin=newlabel;
            subtree[1].next=begin;//Seq.next=S.begin;
            subtree[3].t_true=this->next;//E.true=S.next;
```

```
            subtree[3].t_false=this->begin;//E.false=S.begin;
            subtree[1].gen_code();
            subtree[3].gen_code();
            //S.code=Label S.begin||Seq.code||E.code;
            code.push_back(label_str(begin));
            joint(code,subtree[1].code);
            joint(code,subtree[3].code);
        }
        //-----------
```

五、验证结果

下面是中间代码生成的主程序，其中先后调用词法分析程序、语法分析程序，得到语法分析树后再生成中间代码，然后输出生成的中间代码。在生成中间代码的过程中也会有对应输出，以提示目前到的位置。

为了验证得到的程序的正确性，之后将给出一个正确样例及其使用本程序产生的中间代码，并分析其逻辑是否一致，最后还会给出三个有语义错误的样例，观察中间代码生成程序给出的错误提示。

```
int main(){
    char fname[100];
    scanf("%s",fname);
    FILE* source = fopen(fname, "r");
    //词法分析
    prepare(source);
//    printf("token list:\n");
//    for (int i = 0; i < tokens.size(); i++) {
//        tokens[i].print();
//        cout <<"  ";
//        if(i%5==4)cout<<endl;
//    }
//    cout<<endl<<endl;
    //语法分析
    Treenode root=parse_program();
//    cout<<endl<<"parsing result:"<<endl;
//    root.print(0);
    //分析树转换
    nTreenode n_root=transfrom(root);
//    n_root.print_tree(0);
    //生成中间代码
    n_root.gen_code();
    cout<<endl<<endl<<"int_code:"<<endl;
    //输出中间代码
    n_root.print_code();
    return 0;
}
```

正确样例

src1.txt

```
{this is an example source code}
int A,B,C,D;
char E;
bool F;
while A<C and B>D do
    read B;
    if A=1 then A:=B*C+37
           else repeat A:=A*2
        until A+C<=B+D
       end;
    write A
end
```

得到的中间代码（其中部分代码行中出现的"//"及其后面的部分，是在本报告中为解释验证中间代码正确性而加上的，原来的输出只有三地址中间代码）：

```
Label L1             //L1为while 开始的地方
if A<C goto L3       //and左端条件满足，进入L3，判断右端条件
goto L0              //and左端条件不满足，按短路计算规则，直接跳到L0，while循环之后，不进入循环
Label L3
if B>D goto L2       //and右端条件也满足，则进入L2，while循环内部第一行，进入循环
goto L0              //条件不满足，跳到L0，while循环之后，不进入循环
Label L2
read B
Label L4
if A=1 goto L6       //if条件满足，则进入L6，计算A:=B*C+37的分支
goto L7              //不满足则进入else分支
Label L6             //满足if条件的分支
t6:=B*C
t5:=t6+37
A:=t5
goto L5
Label L7             //else分支
Label L8             //L8为repeat语句开始的地方
t7:=A*2              //A:=A*2
A:=t7
t9:=A+C              //计算until条件
t10:=B+D
if t9<=t10 goto L5   //until条件满足，则退出循环
goto L8              //until条件不满足，回到repeat语句开始的地方
Label L5             //repeat循环后第一行
write A
goto L1              //回到while开始的地方
Label L0             //离开while循环
```

错误样例1

src_error1.txt，其中A被重复定义了

```
{error1 redefined identifier}
int A,B;
char A;
B:=1
```

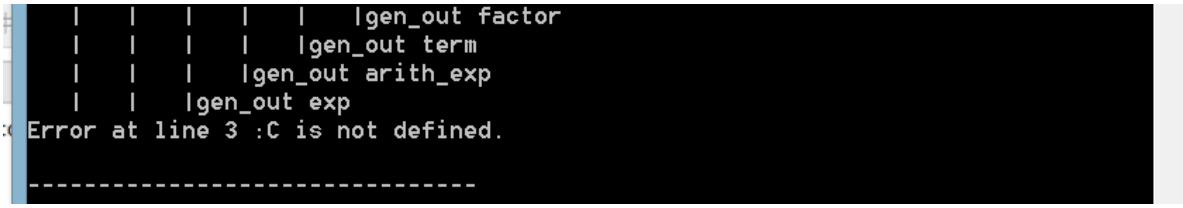
程序输出

错误样例2

src_error2.txt，其中C还未定义

```
{undefined identifier}
int A,B;
C:=1
```

程序输出

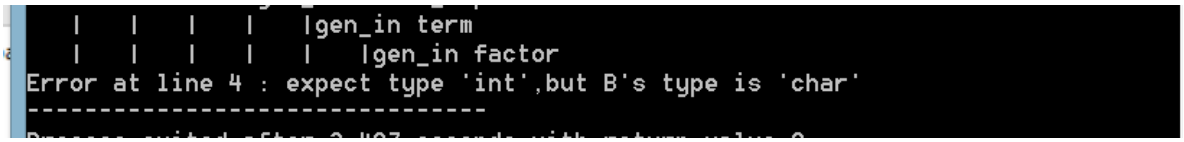


错误样例3

src_error3.txt，其中赋值操作两端类型不一致

```
{ type error }
int A;
char B;
A:=B
```

程序输出



六、实验总结

这次实验只找到了关于TINY+语义规则的部分资料，不过大部分规则在课件上也有对应介绍，因此实际需要自己补充的部分也并不多。在将语义规则实现成对应代码后，一开始生成的中间代码还是有错误，后来分析是label和goto的放置问题，在梳理过逻辑后终于正确实现了。这个中间代码生成程序还能检查语义上的错误，主要就是变量的声明问题，以及数据类型的匹配问题。在完成代码后我还给出了几个样例程序，通过实验验证程序的正确性。