Generating 3-Address Code from an Attribute Grammar

Here is a small but realistic programming language with integer and boolean data, loops and conditionals, procedures with no arguments that compute but don't return. Procedures can have local procedures. So it isn't everything that you might want, but it isn't completely trivial.

We will use an attribute grammar to automatically generate 3-address code for programs in this language.

First, here is a context-free grammar for the language:

```
P ::= program id  DECS ;  S
DECS ::= id : T | proc id ( ) DECS; S | DECS ; DECS
T ::= int |  bool
S ::=        id = E
           | if (E) then S else S
           | while (E) do S
           | id ( )
           |  begin S_LIST end
S_LIST ::= S | S_LIST ; S
E ::= id | number | true | false
           | E+E | E*E | E and E | E or E
```

Here is a program in this language:

```
program foo
        a: int;
        b: int;
        proc add( )
                x: int;
        begin
                x = a+b+1;
                a = x
        end
begin
        a = 1;
        b = 2;
        add( )
end
```

Each procedure has a symbol table containing:

a) The name of the procedure
b) A pointer to the lexically enclosing procedure
c) A pointer to its table of quadruples
d) Symbols and temporaries used in the procedure

We do the same thing for the whole program; it acts like a stand-alone procedure.

Here is the symbol table and list of quads for program foo:

name	**enclosing**	**quads**
foo	nil	
a	int	
b	int	
add	proc	

=	a	1
=	b	2
call	add	

and for procedure add:

name	**enclosing**	**quads**
add	foo	
x	int	
t1	int	

+	a	b	t1
+	t1	1	x
=	x		a

We'll make use of the following helper procedures:

- make_table(name, enclosing_scope) // makes a new symbol table and a new quad table
- add_to_table(table, name type) // adds a name and type to the symbol table
- add_proc_to_table(table, name, new_table) //
- lookup(table, id) // returns a pointer to the id's entry in the symbol table
- new_temp(table, type) // adds a new temporary variable
- emit(table, op, arg1, arg2, arg3) // adds a new quad to the table

Attributes

We will pass the symbol table down as an inherited attribute.

For each expression symbol E, E.place is the address where E is located in the symbol table.

T.type is int, boolean

Here is the attribute grammar:

```
P ::= program id  {DECS.table= make_table(id.val, nil)}
                  DECS ; {S.table=DECS.table}  S

// declarations

DECS ::= id : T  {add_to_table(DECS.table, id.val, T.type}
         | proc id ( ) {DECS$_1$.table=make_table(id.val, DECS.table);
                        add_to_table(DECS.table, id.val, DECS$_1$.table) }
                  DECS$_1$;  {S.table=DECS$_1$.table} S
         | {DECS$_1$.table=DECS.table} DECS$_1$ ;
                  {DECS$_2$.table=DECS.table} DECS$_2$
T ::= int  {T.type=int }|  bool {T.type=bool}
```

```
// expressions
E ::= id  {p = lookup(E.table, id.val);
            if (p != nil)
                    E.place = p
           else
                    error }
          | number  {E.place = new_temp (E.table, int);
                          emit(E.table, =,  E.place, number  }
          | true  {E.place= new_temp(E.table, bool);
                      emit(E.table, =, E.place, true)}
          | false {E.place=new_temp(E.table, bool)
                      emit(E.table, =, E.place, false) }
          | {$E_1$.table=E.table}$E_1$+{$E_2$.table=E.table} $E_2$
            {E.place=new_temp(E.table, int);
              emit(E.table, +, $E_1$.place, $E_2$.place, E.place) }

          // other expressions are similar
// statements

S ::=       id = {E.table=S.table} E
                  {p=lookup(S.table, id.val);
                    if (p == null)
                            error
                    else
                            emit(S.table, =, p, E.place) }
          | id ( ) {p=lookup(S.table, id.val);
                  if (p== null)
                            error
                   else
                            emit(S.table, call, p ) }
          |  begin  {S_LIST.table=S.table} S_LIST end
```

```
S_LIST ::= {S.table=S_LIST.table} S
         | {S_LIST₁.table=S_LIST.table } S_LIST₁ ;
           {S.table=S_LIST.table} S
```

We have handled everything but IF and WHILE statements, which require labels. There are two ways to handle labels: pass them around as attributes (which we have done in other examples, or leave space for them and fill them in later. Just for fun, we'll take the latter approach here.

```
S ::= if {E.table=S.table} E {M_1.table=E.table} M_1
          {emit( "if not', E.place, 'goto', ___) }
      then  {S_1.table=S.table} S_1 M_2 {emit( 'goto', ___)}
      else M_3 {S_2.table=S.table} S_2
          {patch(M_1.quad, M_3.quad);
           patch(M_2.quad, next_quad(S_2.table)}
```

M_1 ::= ε {M_1.quad=next_quad(M_1.table)}

M_2 ::= ε {M_2.quad=next_quad(M_2.table)}

M_3 ::= ε {M_3.quad=next_quad(M_3.table)}

In these rules next_quad(table) is the address of the first free quad in the table.   patch(pos, label) puts the label in the given position.