Other Non Trivial Exercices of A.G.

- Conditional: An attribute Grammar for Switch
- Show the attributes plan used in the grammar
- Apply the grammar to the code generation of a fragments
- An attribute grammar checking switch for the correct presence of the clause update
- Syntactic Sugar: An attribute grammar computing the correct abstract tree of switch
- Preprocessing: An attribute grammar computing a text transliteration
- A strange iteration statement: code generation
- An attribute grammar for by-value parameters in procedure call
- An attribute grammar for by-reference parameters in procedure call
- An attribute grammar for Preprocessing optimization in expression code generation
- An attribute grammar for code optimization in expression code generation

Translation of Conditional Case: C Switch

How does it to do it ?

(Semantics of C Switch)

$$\frac{S \vdash e \rightarrow (S1,v) \qquad S1 \vdash swV\ (v)\ q \rightarrow S2}{S \vdash switch\ (e)\ q \rightarrow S2}$$

$$\frac{v \in \{w1 \ldots wk\} \qquad S \vdash c \rightarrow S1}{S \vdash swV\ (v)\ case\ w1 \ldots wk: c\ q \rightarrow S1}$$

$$\frac{v \notin \{w1 \ldots wk\} \quad S \vdash swV\ (v)\ q \rightarrow S1}{S \vdash swV\ (v)\ case\ w1 \ldots wk: c\ q \rightarrow S1}$$

$$\frac{S \vdash c \rightarrow S1}{S \vdash swV\ (v)\ default: c \rightarrow S1}$$

Extending the Simple Grammar

C::= S
S::= switch E Q
Q::= case V L : C Q
Q::= default : C
L::= V L
L::= ε
V::= Num | Char

The Attribute Grammar

C::= S {C.next:=S.next}
S::= switch E {Q.inloc:=E.loc;}
Q {S.next:=Q.next;}
Q_1::= case V {Cinit:=[quad]; emit("if" Q_1.inloc=V.loc "goto --");
L.inloc:=Q_1.inloc;}
L: {BK(Cinit+L.Cinit,quad);}
C {C-end:=[quad]; emit("goto --");
BK(L.false,quad); Q_2.inloc:=Q_1.inloc}
Q_2 {Q_1.next:=C.next+C-end+Q_2.next;}
Q::= default : C {Q.next:=C.next}
L_1::= V {Cinit:=[quad]; emit("if" L_1.inloc=V.loc "goto --");
L_2.inloc:=L_1.inloc}
L_2 {L_1.false:=L2.false; L_1.Cinit:=Cinit+L2.Cinit;}
L::= ε {L.false:=[quad]; emit("goto --"); L.Cinit:=[];}

Translation of Conditional Case: C Switch / 2

Attributes:

inloc:- location returned from the translation of the switch guard expression
- inherited of {Q,L}
- used for generating code that compares the value of the guard expression with the labels of the statements

Cinit: - list of the target incompletes that must completed with the address starting the code of the case command
- synthesized of {L}

False: - list of the target incompletes that must completed with the address starting the code of the next case guard
- synthesized of {L}
- used to generate code that transfer the control to the next case statement.

Extending the Simple Grammar

C::= S
S::= switch E Q
Q::= case V L : C Q
Q::= default : C
L::= V L
L::= ε
V::= Num | Char

The Attribute Grammar

C::= S {C.next:=S.next}
S::= switch E {Q.inloc:=E.loc;}
Q {S.next:=Q.next;}
Q_1::= case V {Cinit:=[quad]; emit("if" Q_1.inloc=V.loc "goto --");
L.inloc:=Q_1.inloc;}
L: {BK(Cinit+L.Cinit,quad);}
C {C-end:=[quad]; emit("goto --");
BK(L.false,quad); Q_2.inloc:=Q_1.inloc}
Q_2 {Q_1.next:=C.next+C-end+Q_2.next;}
Q::= default : C {Q.next:=C.next}
L_1::= V {Cinit:=[quad]; emit("if" L_1.inloc=V.loc "goto --");
L_2.inloc:=L_1.inloc}
L_2 {L_1.false:=L2.false; L_1.Cinit:=Cinit+L2.Cinit;}
L::= ε {L.false:=[quad]; emit("goto --"); L.Cinit:=[];}

Conditional Case: C Switch Applying the Translation

The Attribute Grammar

```
C::= S {C.next:=S.next}
S::= switch E {Q.inloc:=E.loc;}
          Q {S.next:=Q.next;}
Q1::= case V {Cinit:=[quad]; emit("if" Q1.inloc=V.loc "goto --");
             L.inloc:=Q1.inloc;}
         L: {BK(Cinit+L.Cinit,quad);}
         C {C-end:=[quad]; emit("goto --");
            BK(L.false,quad); Q2.inloc:=Q1.inloc}
         Q2 {Q1.next:=C.next+C-end+Q2.next;}
Q::= default : C {Q.next:=C.next}
L1::= V {Cinit:=[quad]; emit("if" L1.inloc=V.loc "goto --");
        L2.inloc:=L1.inloc}
      L2 {L1.false:=L2.false; L1.Cinit:=Cinit+L2.Cinit;}
L::= ε {L.false:=[quad]; emit("goto --"); L.Cinit:=[];}
```

Show the code generated from the translation of the fragment below, of a Simple program:

```
switch x
  case u, v, z : x:= y*w+x;
  default: x:=0;
```

Let locx, locu, locy, locz be the locations of the attribute x.loc, u.loc, u.loc, z.loc resp.

```
0   if locx=locu goto 4
1   if locx=locv goto 4
2   if locx=locz goto 4
3   goto 8
4   loc1:=locy [*] locw
5   loc2:=loc1 [+] locx
6   locx:=loc2
7   goto --
8   locx:= 0

                c.next = [7]
```

Conditional Case: Switch
Syntactic Sugar and Abstract Syntax

Noting that the this form of switch is adeguate for the Abstract Syntax of switch of C-like languages

Complete the grammar with the attributes that are needed to checking for the correct presence of *exactly one default* statement in the command switch

Extending the Simple Grammar

C::= S
S::= switch E Q
Q::= U Q
Q::= U
U::= case V L : C
U::= default : C
L::= V L
L::= ε
V::= Num | Char

Into an Attribute Grammar Checking for one default

C::= S
S::= switch E Q
Q::= U Q
Q::= U
U::= case V L : C
U::= default : C
L::= V L
L::= ε
V::= Num | Char

Conditional Case: Switch
Syntactic Sugar and Abstract Syntax

Noting that the this form of switch is adeguate for the Abstract Syntax of switch of C-like languages

Complete the grammar with the attributes that are needed to generate an Abstract Syntax having the switch default statement as the last sub-tree in the tree depth-first visit.

Extending the Simple Grammar

C::= S
S::= switch E Q
Q::= U Q
Q::= U
U::= case V L : C
U::= default : C
L::= V L
L::= ε
V::= Num | Char

Into an Attribute Grammar for correct Abstract Tree

S::= switch E Q {S.tree:=mk("switch",E.tree+Q.treeC+Q.treeD);}
Q_1::= U Q_2 {Q_1.treeC:=U.treC+Q_2.treeC; Q_1.treeD:=U.treD+Q_2.treeD;}
Q::= U {Q.treeC:=U.treC; Q_1.treeD:=U.treeD;}
U::= case V L : C {U.treeC:=mk("case",mk("label",V.tree+L.tree),C.tree);
U.treeD:=[];}
U::= default : C {U.treeD:=mk("default",C.tree);
U.treeC:=[];}
L_1::= V L_2 {L_1.tree:= [mk("",V.entry)]+L_2.tree;}
L::= ε {L.tree::=[];}
V::= Num {V.entry:=Num.entry;}
V::= Char {V.entry:=Char.entry;}

where mk("a")=["a"-<>]; mk("a",L)=["a"-L]; operazioni su liste L di tree: emptylist=[];append=+

Conditional Case: Switch
Syntactic Sugar and Abstract Syntax

Complete the grammar with the attributes that are needed to generate an Abstract Syntax having the switch default statement as the last sub-tree in the tree depth-first visit.

Show the tree that results applying the grammar to the fragment below:

```
switch x
  case u, v, z : x:= y*w+x;
  default: x:=0;
  case w, y: x:=w+z;
```

Into an Attribute Grammar for correct Abstract Tree

S::= switch E Q {S.tree:=mk("switch",E.tree+Q.treeC+Q.treeD);}
Q_1::= U Q_2 {Q_1.treeC:=U.treC+Q_2.treeC; Q_1.treeD:=U.treD+Q_2.treeD;}
Q::= U {Q.treeC:=U.treC; Q_1.treeD:=U.treeD;}
U::= case V L : C {U.treeC:=mk("case",mk("label",V.tree+L.tree),C.tree);
U.treeD:=[];}
U::= default : C {U.treeD:=mk("default",C.tree);
U.treeC:=[];}
L_1::= V L_2 {L_1.tree:= [V.tree]+L_2.tree;}
L::= ε {L.tree::=[];}
V::= Num {V.tree:=mk(Num.lexeme);}
V::= Char {V.tree:=mk(Char.lexeme);}

where mk("a")=["a"-<>]; mk("a",L)=["a"-L]; operazioni su liste L di tree: emptylist=[];append=+

Conditional Case: Switch
Syntactic Sugar and Abstract Syntax

Noting that the this form of switch is adeguate for the Abstract Syntax of switch of C-like languages

Complete the grammar with the attributes that are needed to generate a text transliteration of the switch command in a way that default statement is put at the end, whilst all the other ones stay unchanged

Extending the Simple Grammar

C::= S
S::= switch E Q
Q::= U Q
Q::= U
U::= case V L : C ;
U::= default : C ;
L::= , V L
L::= ε
V::= Num | Char

Into an Attribute Grammar Checking for one default

C::= S
S::= switch E Q {S.text:="switch"||E.text||Q.textC+Q.textD);}
Q_1::= U Q_2 {Q_1.textC:=U.textC||Q_2.textC; Q_1.textD:=U.textD||Q_2.textD;}
Q::= U {Q.textC:=U.textC; Q_1.textD:=U.textD;}
U::= case V L : C ; {U.textC:="case"||V.tetx||L.text||":||C.text||";" ;
U.textD:=[];}
U::= default : C ; {U.textD:="default"||":"||C.text);
U.textC:=[];}
L_1::= , V L_2 {L_1.text:= ","||V.text||L_2.text;}
L::= ε {L.text::=[];}
V::= Num {V.text:=Num.lexeme;}
V::= Char {V.text:=Char.lexeme;}

where || = text concatenation

A Strange Iteration Statement: iter

(Reduction Semantics of iter)

$$\frac{S \vdash e \rightarrow (S,i) \quad q \equiv c_1 \ldots c_i \ldots c_n \quad S \vdash c_i \rightarrow S1 \quad S1 \vdash \text{iter } e\ q \rightarrow S2}{S \vdash \text{iter } e\ q \rightarrow S2}$$

$$\frac{S \vdash e \rightarrow (S,i) \quad q \equiv c_1 \ldots c_i \ldots c_n \quad i \notin [1..n]}{S \vdash \text{iter } e\ q \rightarrow S}$$

Extend the Simple Grammar with the command *iter*

C::= R
R::= iter E Q
Q::= C Q
Q::= C

Extend into an attribute grammar for code translation

C::= R {C.next:= R.next;}
R::= iter {init:=quad;}
E {Q.iloc:= E.loc; Q.icount:=1; Q.iinit:=init; Q.inextc:=[]}
Q {R.next:= Q.next;}
Q_1::= {BK(Q_1.inextc,quad); Q_2.inextc:=[quad];
emit("if" Q_1.iloc [≠] Q_1.icount "goto --");}
C {BK(C.next,Q_1.iinit); emit("goto" Q_1.iinit);
Q_2.iloc:= Q_1.ilocc; Q_2.icount:=Q_1.icoun+1; Q_2.iinit:=Q_1.init;}
Q_2 {Q_1.next:= Q_2.next;}
Q::= {BK(Q.inextc,quad); Q.next:=[quad];
emit("if" Q_1.iloc [≠] Q_1.icount "goto --");}
C {BK(C.next,Q.iinit); emit("goto" Q.iinit);}

A Strange Iteration Statement: iter /2

Extend into an attribute grammar for code translation

```
C::= R {C.next:= R.next;}
R::= iter {init:=quad;}
      E {Q.iloc:= E.loc; Q.icount:=1; Q.iinit:=init; Q.inextc:=[]}
      Q {R.next:= Q.next;}
Q1::= {BK(Q1.inextc,quad); Q2.inextc:=[quad];
        emit("if" Q1.iloc [≠] Q1.icount "goto --");}
        C {BK(C.next,Q1.iinit); emit("goto" Q1.iinit);
            Q2.iloc:= Q1.ilocc; Q2.icount:=Q1.icoun+1; Q2.iinit:=Q1.init;}
        Q2 {Q1.next:= Q2.next;}
Q::= {BK(Q.inextc,quad); Q.next:=[quad];
       emit("if" Q1.iloc [≠] Q1.icount "goto --");}
       C {BK(C.next,Q.iinit); emit("goto" Q.iinit);}
```

Show the code generated from the translation of the fragment below, of a Simple program:

```
iter x
   x:= x+1;
   x:= x -1;
   while x>1 do x:=x-1;
```

```
0    if locx≠1 goto 4
1    loc1:=locx+1
2    locx:=loc1
3    goto 0
4    if locx≠2 goto
5    loc2:=locx-1
6    locx:=loc2
7    goto 0
8    if locx≠3 goto --
9    loc3:=locx>1
10   if loc3=false goto 0
11   loc4:=locx+1
12   locx:=loc4
13   goto 0

              c.next = [8]
```

Procedure Invocation: Transmission by Value

(Reduction Semantics of call-by-value)

$$\frac{S_0 \vdash e_1 \rightarrow (S_1,v_1) \;\ldots\; S_{n-1} \vdash e_n \rightarrow (S_n,v_n) \qquad S_n \vdash binding((v_1,\ldots,v_n),(x_1,\ldots,x_n)) \rightarrow S_{inv} \qquad body(S_n,p) = C \qquad pars(S_n,p) = x_1,\ldots,x_n \qquad S_{inv} \vdash C \rightarrow S_{ret}}{S_0 \vdash p(e_1,\ldots,e_n) \rightarrow S_{ret}}$$

Extend into an attribute grammar

C::= P
P::= ide (E V)
V::= , E V
V::=ε

The attribute grammar for code translation

```
C::= P {C.next:=[];}
P::= ide (E {}
          V) {emit("param" E.loc);
              list:=V.list; n:=size(list)
              while(list≠[]{
                  topp:=head(list);
                  emit("param" top);
                  list:=tail(list)}
              emit("call" ide.loc n);}
V1::= , E V2 {V1.list:=[E.loc]+V2.list;}
V::=ε {V.list:[];}
```

Procedure Invocation: Transmission by Reference

(Reduction Semantics of call-by-reference)

$$\frac{S_0 \vdash e_1 \rightarrow (S_1,l_1) \;\ldots\; S_{n-1} \vdash e_n \rightarrow (S_n,l_n) \quad S_n \vdash \text{binding}((l_1,\ldots,l_n),(x_1,\ldots,x_n)) \rightarrow S_{inv} \qquad \text{body}(S_n,p) = C \quad \text{pars}(S_n,p) = x_1,\ldots,x_n \quad S_{inv} \vdash C \rightarrow S_{ret}}{S_0 \vdash p(e_1,\ldots,e_n) \rightarrow S_{ret}}$$

Extend into an attribute grammar

C::= P
P::= ide (E V)
V::= , E V
V::=ε

The attribute grammar for code translation

```
C::= P {C.next:=[];}
P::= ide (E {}
            V) {emit("param  &"E.loc);
                 list:=V.list; n:=size(list)
                 while(list≠[]{
                     topp:=head(list);
                     emit("param  &"top);
                      list:=tail(list)}
                 emit("call" ide.loc n);}
```

V_1::= , E V_2 {V_1.list:=[E.loc]+V_2.list;}
V::=ε {V.list:[];}

Preprocessing Optimization
Expression Constant Reduction

Complete the grammar of integer expressions of Simple with the attributes and the actions that are needed to generate a text transliteration of expression in a way that constant sub-expressions are statically computed and replaced with the resulting value. For instance, the expression "3+2*x*(2+1)+10+y" must be reduced to: "x*6+y+13".

Extend into an attribute grammar

E::= E + F | F
F::= F * N | N
N::= - N | T
T::= num | ide | (E)

Two syntesized attributes for all the symbols but E'
.const: the value of the constant part computed during the traversal of the tree rooted at the symbol;
.text: the text of the variable part that cannot be solved at compile time, during the traversal of the tree rooted at the symbol;
For typing convenience, attributes will be indicated with only first letter but in capitalized form, i.e. X.C stands for X.const
Noting the grammar modification that uses the augmented symbol E'

The attribute grammar must be augmented:

E'::= E {if (E.T≠""&E.C=0){E'.T:=E.T;}
else if (E.T≠""){E'.T:=E.T++["+"] ++SR(E.C);} else E'.T:=SR(E.C);}
E_1::= E_2+F {E_1.C:=E_2.C+F.C; E_1.T:=E_2.T++F.T;}
E::= F {if (F.T≠""&F.C=0){E.T:=F.T;}
else if (F.T≠""){E.T:=F.T++["*"] ++SR(F.C);} else E.T:=SR(F.C);}
F_1::= F_2*N {F_1.C:=F_2.C*N.C; F_1.T:=F_2.T++N.T;}
F::= N {F.C:=N.C; F.T:=N.T;}
N_1::= - N_2 {N_1.C:= (N_2.C≠0)?-N_2.C:N_2.C; N_1.T:= (N_2.T≠"")?["-"]++N_2.C:"";}
N::= T {N.C= T.C; N.T:=T.T;}
T::= num {T.C:=num.value; T.T:="";}
T::= ide {T.C:=0; T.T:=ide.lexeme;}
T::= (E) {T.C:= (E.C=0 or E.T="") ? E.C : 0 ;
T.T:= (E.T≠""&E.C≠0) ? E.T++"+"++SR(E.C) : E.T;}

The domain of attribute .text is String, hence ++ is string concatenation whilst SR is Numeral-to-String conversion operator.

Preprocessing Optimization
Expression Constant Reduction / 2

Complete the grammar of integer expressions of Simple with the attributes and the actions that are needed to generate a text transliteration of expression in a way that constant sub-expressions are statically computed and replaced with the resulting value. For instance, the expression "3+2*x*(2+1)+10+y" must be reduced to: "x*6+y+13".

Extend into an attribute grammar

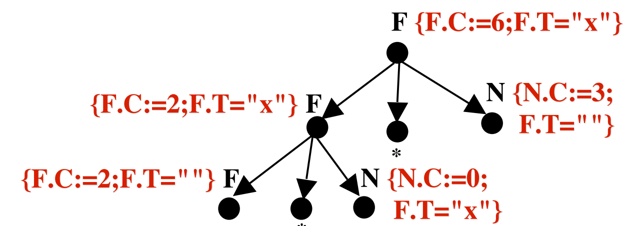
E::= E + F | F
F::= F * N | N
N::= - N | T
T::= num | ide | (E)

E' {E'.T:=...

The attribute grammar must be augmented:

E'::= E {if (E.T≠""&E.C=0){E'.T:=E.T;}
    else if (E.T≠""){E'.T:=E.T++"+"++SR(E.C);} else E'.T:=SR(E.C);}
E_1::= E_2+F {E_1.C:=E_2.C+F.C; E_1.T:=E_2.T++F.T;}
E::= F {if (F.T≠""&F.C=0){E.T:=F.T;}
    else if (F.T≠""){E.T:=F.T++"*"++SR(F.C);} else E.T:=SR(F.C);}
F_1::= F_2*N {F_1.C:=F_2.C*N.C; F_1.T:=F_2.T++N.T;}
F::= N {F.C:=N.C; F.T:=N.T;}
N_1::= - N_2 {N_1.C:= (N_2.C≠0)?-N_2.C:N_2.C; N_1.T:= (N_2.T≠"")?"-"++N_2.C:"";}
N::= T {N.C= T.C; N.T:=T.T;}
T::= num {T.C:=num.value; T.T:="";}
T::= ide {T.C:=0; T.T:=ide.lexeme;}
T::= (E) {T.C:= (E.C=0 or E.T="") ? E.C : 0 ;
    T.T:= (E.T≠""&E.C≠0) ? E.T++"+"++SR(E.C) : E.T;}

Complete with the trees around the subtree for 2*x*(2+1)



Intermediate Code Optimization
Expression Constant Reduction

Complete the grammar of integer expressions of Simple with the attributes and the actions that are needed to generate a code translation of expression in a way that constant sub-expressions are statically computed and replaced with the resulting value. For instance, the expression "3+2*x*(2+1)+10+y" must be translated as for: "x*6+y+13".

Extend into an attribute grammar

```
E::= E + F | F
F::= F * N | N
N::= - N | T
T::= num | ide | (E)
```

Two syntesized attributes in addition to the invariant .loc for the Symbols E,F,N,T:

.const: the value of the constant part computed during the traversal of the tree rooted at the symbol;

.code: true iff the translation of the string derived by the symbol modified the emit file (i.e.the attribute .loc is defined.

The attribute grammar must be augmented:

```
E'::= E {if E.code&E.const=0) then {E'.loc:=E.loc;}
          else {temp:=newtemp; E'.loc:=temp;
                if E.code then {emit(temp":= E.loc [+] "#"E.const);}
                        else emit(temp":= #"E.const);}}
E1::= E2+F {E1.const:=E2.const+F.const; E1.code:=E2.code orF.code;
            if (E2.code&F.code)then{temp:=newtemp; E1.loc:=temp;
                           emit(temp":="E2.loc[+]F.loc);E1.loc:=temp;}
            else if E2.code then {E1.loc:=E2.loc;}
                 else if F.code {E1.loc:=F.loc;} else {E1.loc:=0;//not used};}
E::= F {…complete…}
F1::= F2*N {F1.const:=F2.const*N.const; F1.code:=F2.code or N.code;
            if (F2.code&N.code)then{temp:=newtemp; F1.loc:=temp;
                           emit(temp":="F2.loc[*]N.loc);F1.loc:=temp;}
            else if F2.code then {F1.loc:=F2.loc;}
                 else if N.code {F1.loc:=N.loc;} else {F1.loc:=0;//not used};}
N::= T {…complete…}
N1::= - N2 {N1.const= -N1.const; N1.code:=N2.code; N1.loc:=N2.loc;
            if (N1.code) then emit(N1.loc:=[-]N1.loc);}
```

Intermediate Code Optimization
Expression Constant Reduction / 2

Complete the grammar of integer expressions of Simple with the attributes and the actions that are needed to generate a code translation of expression in a way that constant sub-expressions are statically computed and replaced with the resulting value. For instance, the expression "3+2*x*(2+1)+10+y" must be translated as for: "x*6+y+13".

Extend into an attribute grammar

E::= E + F | F
F::= F * N | N
N::= - N | T
T::= num | ide | (E)

The attribute grammar must be augmented:

N::= T {N.const= T.const; N.code:=T.code; N.loc:=T.loc;}
T::= num {T.const:=num.value; T.code:=false; T.loc:=0//not used}
T::= ide {T.const:=0; T.code:=true; T.loc:=ide.loc//not used}
T::= (E) {T.const:=E.const; T.code:=E.code; T.loc:=E.loc}

Two syntesized attributes in addition to the invariant .loc for the Symbols E,F,N,T:
.const: the value of the constant part computed during the traversal of the tree rooted at the symbol;
.code: true iff the translation of the string derived by the symbol modified the emit file (i.e.the attribute .loc is defined.