Tutorial 05: CS31003: Compilers:

[M-05] IC Translation: Control, Type, Array & Function

Indranil Sengupta
Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

isg@iitkgp.ac.in
ppd@cse.iitkgp.ac.in

October 17, 2020

Doubts from the Week

Problem: Control Construct Grammar (do-while & for)

Consider the control construct grammar extended with 'do-while' and 'for':

1-2:	S	$\rightarrow$	$\{ L \} \mid$ **id** $= E$ **;**
3-5:	S	$\rightarrow$	**if** (B) S $\mid$ **if** (B) S **else** S $\mid$ **while** (B) S
6-7:	S	$\rightarrow$	**do** S **while (** B **);** $\mid$ **for (** E **;** B **;** E **)** S
8-9:	L	$\rightarrow$	$L\ S \mid S$
10-13:	E	$\rightarrow$	**id** $\mid$ **num** $\mid E + E \mid E = E$

1. Convert the grammar with back-patching rules
2. Write semantic actions for 'do-while' and 'for'. Actions for other productions are to be used from the lecture slides
3. Using the actions translate the following to 3 address. Show the annotated parse tree and the reduction actions.
   1. ```
      sum = 0; n = 5;
      for(i = 1; i < n; i = i + 1) sum = sum + i;
      ```
   2. ```
      sum = 0; n = 5; i = 1;
      do
          sum = sum + i; i = i + 1;
      while (i < n);
      ```
   3. ```
      sum = 0; n = 5; i = 1;
      do
          j = 1;
          while (i > j) j = j + 2;
          if (i == j) sum = sum + i;
          i = i + 1;
      while (i < n);
      ```
      ```

Solution: Back-patching Control Construct Grammar (`do-while` & `for`)

1:	S	$\rightarrow$	$\{\ L\ \}$
2:	S	$\rightarrow$	$\textbf{id} = E\ \textbf{;}$
3:	S	$\rightarrow$	$\textbf{if}\ (B)\ M\ S_1$
4:	S	$\rightarrow$	$\textbf{if}\ (B)\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$
5:	S	$\rightarrow$	$\textbf{while}\ M_1\ (B)\ M_2\ S_1$
6:	S	$\rightarrow$	$\textbf{do}\ M_1\ S_1\ M_2\ \textbf{while}\ \textbf{(}\ B\ \textbf{);}$
7:	S	$\rightarrow$	$\textbf{for}\ \textbf{(}\ E_1\ \textbf{;}\ M_1\ B\ \textbf{;}\ M_2\ E_2\ N\ \textbf{)}\ M_3\ S_1$
8:	L	$\rightarrow$	$L_1\ M\ S$
9:	L	$\rightarrow$	S
10:	E	$\rightarrow$	$\textbf{id}$
11:	E	$\rightarrow$	$\textbf{num}$
12:	M	$\rightarrow$	ϵ // Marker rule
13:	N	$\rightarrow$	ϵ // Fall-through Guard rule

Solution: Back-patching Control Construct Grammar with Actions (`do-while` & `for`)

6: $S \rightarrow$ **do** $M_1\ S_1\ M_2$ **while (B);**

$\{$ $backpatch(B.truelist, M_1.instr);$
$backpatch(S_1.nextlist, M_2.instr);$
$S.nextlist = B.falselist;$ $\}$

7: $S \rightarrow$ **for (** E_1 **;** $M_1\ B$ **;** $M_2\ E_2\ N$ **)** $M_3\ S_1$

$\{$ $backpatch(B.truelist, M_3.instr);$
$backpatch(N.nextlist, M_1.instr);$
$backpatch(S_1.nextlist, M_2.instr);$
$emit(\text{"}goto\text{"}\ \ M_2.instr);$
$S.nextlist = B.falselist;$ $\}$

Solution: 1.3.1

Reductions :

$E_1 \rightarrow num_0$
$S_1 \rightarrow id_{sum} = E_1$
$L_1 \rightarrow S_1$
$M_1 \rightarrow \epsilon$
$E_2 \rightarrow num_5$
$S_2 \rightarrow id_n = E_2$
$L_2 \rightarrow L_1 M_1 S_2$
$M_2 \rightarrow \epsilon$
$E_3 \rightarrow id_i$
$E_4 \rightarrow num_1$
$E_5 \rightarrow E_3 = E_4$
$M_3 \rightarrow \epsilon$
$E_6 \rightarrow id_i$
$E_7 \rightarrow id_n$
$B_1 \rightarrow E_6 < E_7$
$M_4 \rightarrow \epsilon$
$E_8 \rightarrow id_i$
$E_9 \rightarrow id_i$
$E_{10} \rightarrow num_1$
$E_{11} \rightarrow E_9 + E_{10}$
$E_{12} \rightarrow E_8 = E_{11}$
$N_1 \rightarrow \epsilon$
$M_5 \rightarrow \epsilon$
$E_{13} \rightarrow id_{sum}$
$E_{14} \rightarrow id_i$
$E_{15} \rightarrow E_{13} + E_{14}$
$S_3 \rightarrow id_{sum} = E_{15}$
$S_4 \rightarrow for(E_5; M_3 B_1; M_4 E_{12} N_1) M_5 S_3$
$L_3 \rightarrow L_2 M_2 S_4$
$S_5 \rightarrow L_3$

TAC:

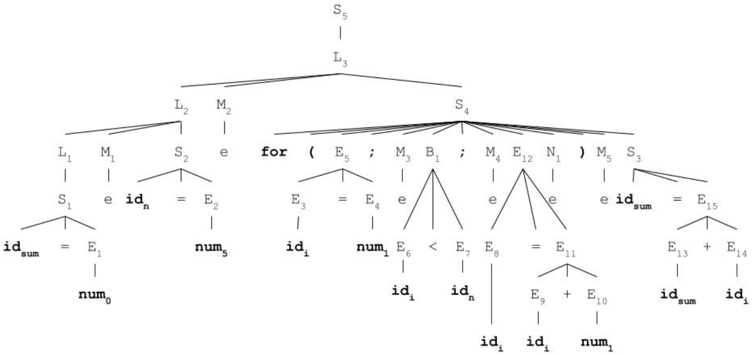
```
100: t0 = 0
101: sum = t0
102: t1 = 5
103: n = t1
104: t2 = 1
105: i = t2
106: if i < n goto 112 // BP ( B1.TL, M5.I )
107: goto ...
108: t3 = 1
109: t4 = i + t3
110: i = t4
111: goto 106           // BP ( N1.NL, M3.I )
112: t5 = sum + i
113: sum = t5
114: goto 108
```

Actions:

```
S1.NL = {}               B1.FL = {107}
L1.NL = S1.NL = {}       M4.I = {108}
M1.I = {102}             N1.NL = {111}
S2.NL = {}               M5.I = {112}
L2.NL = S2.NL = {}       S3.NL = {}
M2.I = {104}             S4.NL = B1.FL = {107}
M3.I = {106}             L3.NL = S4.NL = {107}
B1.TL = {106}            S5.NL = L3.NL = {107}
```

```
sum = 0; n = 5;
for(i = 1; i < n; i = i + 1) sum = sum + i;
```

Solution: 1.3.1



Solution: 1.3.2

Reductions :

$E_1 \rightarrow num_0$
$S_1 \rightarrow id_{sum} = E_1$
$L_1 \rightarrow S_1$
$M_1 \rightarrow \epsilon$
$E_2 \rightarrow num_5$
$S_2 \rightarrow id_n = E_2$
$L_2 \rightarrow L_1 M_1 S_2$
$M_2 \rightarrow \epsilon$
$E_3 \rightarrow num_1$
$S_3 \rightarrow id_i = E_3$
$L_3 \rightarrow L_2 M_2 S_3$
$M_3 \rightarrow \epsilon$
$M_4 \rightarrow \epsilon$
$E_4 \rightarrow id_{sum}$
$E_5 \rightarrow id_i$
$E_6 \rightarrow E_4 + E_5$
$S_4 \rightarrow id_{sum} = E_6$
$L_4 \rightarrow S_4$
$M_5 \rightarrow \epsilon$
$E_7 \rightarrow id_i$
$E_8 \rightarrow num_1$
$E_9 \rightarrow E_7 + E_8$
$S_5 \rightarrow id_i = E_9$
$L_5 \rightarrow L_4 M_5 S_5$
$S_6 \rightarrow L_5$
$M_6 \rightarrow \epsilon$
$E_{10} \rightarrow id_i$
$E_{11} \rightarrow id_n$
$B_1 \rightarrow E_{10} < E_{11}$
$S_7 \rightarrow do\ M_4 S_6 M_6 while(B_1)$
$L_6 \rightarrow L_3 M_3 S_7$
$S_8 \rightarrow L_6$

```
sum = 0; n = 5; i = 1;
do
    sum = sum + i; i = i + 1;
while (i < n);
```

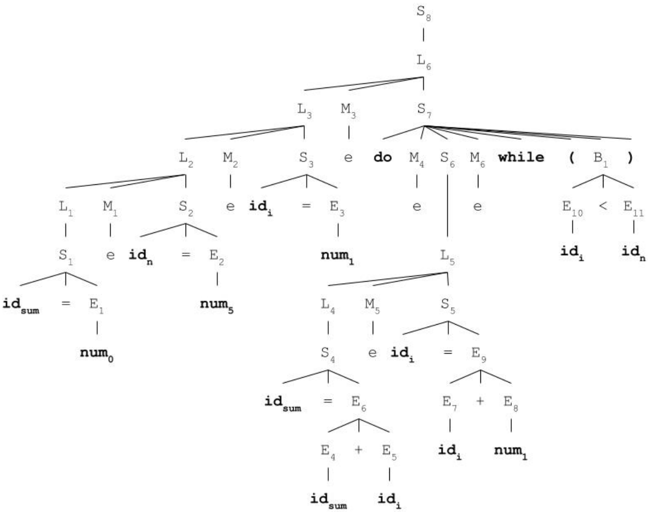
TAC:

```
100: t0 = 0
101: sum = t0
102: t1 = 5
103: n = t1
104: t2 = 1
105: i = t2
106: t3 = sum + i
107: sum = t3
108: t4 = 1
109: t5 = i + t4
110: i = t5
111: if i < n goto 106 //BP(B1.TL,M4.I)
112: goto ...
```

Actions:

```
S1.NL = {}
L1.NL = S1.NL = {}
M1.I = {102}
S2.NL = {}
L2.NL = S2.NL = {}
M2.I = {104}
S3.NL = {}
L3.NL = S3.NL = {}
M3.I = {106}
M4.I = {106}
S4.NL = {}
L4.NL = S4.NL = {}
M5.I = {108}
S5.NL = {}
L5.NL = S5.NL = {}
S6.NL = L5.NL = {}
M6.I = {111}
B1.TL = {111}
B1.FL = {112}
S7.NL = B1.FL = {112}
L6.NL = S7.NL = {112}
S8.NL = L6.NL = {112}
```

Solution: 1.3.2



Solution: 1.3.3

Reductions :

$E_1 \rightarrow num_0$
$S_1 \rightarrow id_{sum} = E_1$
$L_1 \rightarrow S_1$
$M_1 \rightarrow \epsilon$
$E_2 \rightarrow num_5$
$S_2 \rightarrow id_n = E_2$
$L_2 \rightarrow L_1 M_1 S_2$
$M_2 \rightarrow \epsilon$
$E_3 \rightarrow num_1$
$S_3 \rightarrow id_i = E_3$
$L_3 \rightarrow L_2 M_2 S_3$
$M_3 \rightarrow \epsilon$
$M_4 \rightarrow \epsilon$
$E_4 \rightarrow num_1$
$S_4 \rightarrow id_j = E_4$
$L_4 \rightarrow S_4$
$M_5 \rightarrow \epsilon$
$M_6 \rightarrow \epsilon$
$E_5 \rightarrow id_i$
$E_6 \rightarrow id_j$
$B_1 \rightarrow E_5 > E_6$
$M_7 \rightarrow \epsilon$
$E_7 \rightarrow id_j$
$E_8 \rightarrow num_2$
$E_9 \rightarrow E_7 + E_8$
$S_5 \rightarrow id_j = E_9$
$S_6 \rightarrow while\ M_6 (B_1) M_7 S_5$
$L_5 \rightarrow L_4 M_5 S_6$
$M_8 \rightarrow \epsilon$
$E_{10} \rightarrow id_i$
$E_{11} \rightarrow id_j$
$B_2 \rightarrow E_{10} == E_{11}$
$M_9 \rightarrow \epsilon$
$E_{12} \rightarrow id_{sum}$
$E_{13} \rightarrow id_i$
$E_{14} \rightarrow E_{12} + E_{13}$
$S_7 \rightarrow id_{sum} = E_{14}$
$S_8 \rightarrow if (B_2) M_9 S_7$
$L_6 \rightarrow L_5 M_8 S_8$
$M_{10} \rightarrow \epsilon$
$E_{15} \rightarrow id_i$
$E_{16} \rightarrow num_1$
$E_{17} \rightarrow E_{15} + E_{16}$
$S_9 \rightarrow id_i = E_{17}$
$L_7 \rightarrow L_6 M_{10} S_9$
$S_{10} \rightarrow L_7$
$M_{11} \rightarrow \epsilon$
$E_{18} \rightarrow id_i$
$E_{19} \rightarrow id_n$
$B_3 \rightarrow E_{18} < E_{19}$
$S_{11} \rightarrow do\ M_4 S_{10} M_{11} while (B_3)$
$L_8 \rightarrow L_3 M_3 S_{11}$
$S_{12} \rightarrow L_8$

Solution: 1.3.3

TAC:

```
100: t0 = 0
101: sum = t0
102: t1 = 5
103: n = t1
104: t2 = 1
105: i = t2
106: t3 = 1
107: j = t3
// BP ( B1.TL, M7.I )
108: if i > j goto 110
// BP ( L5.NL, M8.I )
109: goto 114
110: t4 = 2
111: t5 = j + t4
112: j = t5
113: goto 108
// BP ( B2.TL, M9.I )
114: if i == j goto 116
// BP ( L6.NL, M10.I )
115: goto 118
116: t6 = sum + i
117: sum = t6
118: t7 = 1
119: t8 = i + t7
120: i = t8
// BP ( B3.TL, M4.I )
121: if i < n goto 106
122: goto ...
```

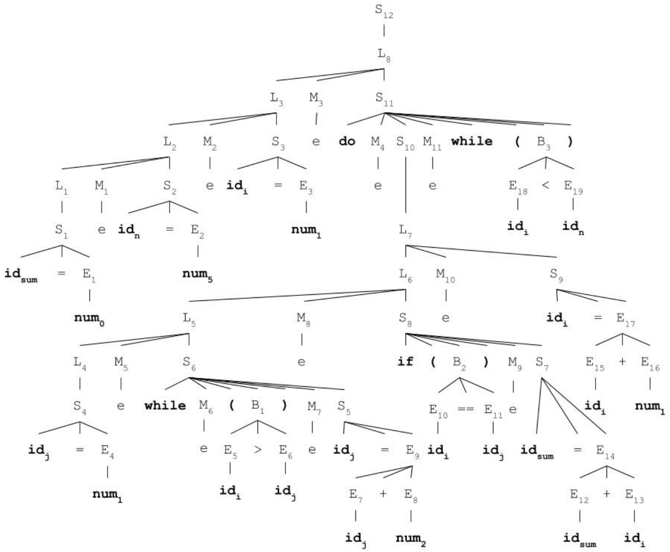
Actions:

```
S1.NL = {}
L1.NL = S1.NL = {}
M1.I = {102}
S2.NL = {}
L2.NL = S2.NL = {}
M2.I = {104}
S3.NL = {}
L3.NL = S3.NL = {}
M3.I = {106}
M4.I = {106}
S4.NL = {}
L4.NL = S4.NL = {}
M5.I = {108}
M6.I = {108}
B1.TL = {108}
B1.FL = {109}
M7.I = {110}
S5.NL = {}
```

```
S6.NL = B1.FL = {109}
L5.NL = S6.NL = {109}
M8.I = {114}
B2.TL = {114}
B2.FL = {115}
M9.I = {116}
S7.NL = {}
S8.NL = B2.FL U S7.NL = {115}
L6.NL = S8.NL = {115}
M10.I = {118}
S9.NL = {}
L7.NL = S9.NL = {}
S10.NL = L7.NL = {}
M11.I = {121}
B3.TL = {121}
B3.FL = {122}
S11.NL = B3.FL = {122}
L8.NL = S11.NL = {122}
S12.NL = L8.NL = {122}
```

```
sum = 0; n = 5; i = 1;
do
    j = 1;
    while (i > j) j = j + 2;
    if (i == j) sum = sum + i;
    i = i + 1;
while (i < n);
```

Solution: 1.3.3



Problem: Control Construct Grammar (`switch`)

Consider the following grammar for switch control

$$
\begin{array}{lll}
S & \rightarrow & \textbf{switch (} E \textbf{) } S_1 \\
S & \rightarrow & \textbf{case num: } S_1 \\
S & \rightarrow & \textbf{default: } S_1
\end{array}
$$

1. Design a scheme for translation using:
    1. Synthesized Attributes
    2. Inherited Attributes
2. According to your scheme will cases occurring later (or earlier) in the switch sequence take more time to be processed during execution due to the serial nature of test-and-go of the cases? Can this significantly affect performance if there are a large number of cases in a switch? What improvisation/s can you do in your scheme to make this efficient at run-time so that all cases can be hit (almost) at constant time?

Solution: Control Construct Grammar (`switch`) (1)

$$S \rightarrow \textbf{switch (} E \textbf{) } S_1$$
$$S \rightarrow \textbf{case num: } S_1$$
$$S \rightarrow \textbf{default: } S_1$$

Using Mutually Exclusive "case" Clauses - Unlike C	
Synthesized Attributes	Inherited Attributes

Synthesized Attributes

	Code to Evaluate E into **t**
	goto test
L_1:	Code for S_1
	goto next
L_2:	Code for S_2
	goto next
	...
L_{n-1}:	Code for S_{n-1}
	goto next
L_n:	Code for S_n
	goto next
test:	**if t** $=$ V_1 **goto** L_1
	if t $=$ V_2 **goto** L_2
	...
	if t $=$ V_{n-1} **goto** L_{n-1}
	goto L_n
next:	

Inherited Attributes

	Code to Evaluate E into **t**
	if t != V_1 **goto** L_1
	Code for S_1
	goto next
L_1:	**if t !=** V_2 **goto** L_2
	Code for S_2
	goto next
L_2:	
	...
L_{n-2}:	**if t !=** V_{n-1} **goto** L_{n-1}
	Code for S_{n-1}
	goto next
L_{n-1}:	Code for S_n
next:	

Solution: Control Construct Grammar (`switch`) (2)

1. The cases occurring later in the switch sequence will take more time to be processed during the execution due to the serial nature of test-and-go of the cases. As the cases are processed one after the other, the cases occurring later naturally will be processed only after all the cases occurring before.
2. The performance of this scheme is of linear order, so the performance increases linearly with the number of cases. For a very large number of cases this will result in poor performance.
3. The scheme can be made more efficient by using hash table. The hash table entries will have the value as the key and a corresponding label of the statement. If the value is not found in the hash table then jump to default case is generated. With a good hash function for the table, all cases can be hit in almost constant time.
4. If all the case values lie in a small range [min, max] where max-min is small and all the distinct values are a significant fraction of (max-min) then max-min number of buckets can be created. Where j-min contains the label for case j statement. Any bucket unfilled will contain the default statement label. This hits all the cases in constant time, but with constraint that all case values lie in small range.

Problem: Control Construct Grammar (`break` & `continue`)

Design suitable schemes to translate **break** and **continue** statements:

$$S \rightarrow \textbf{break;}$$
$$S \rightarrow \textbf{continue;}$$

Solution: Control Construct Grammar (`break` & `continue`)

Extra attributes will be required here for S:

1. **S.break**: A list (indices of) quads having dangling exits occurring due to *break* for statement S
2. **S.continue** : A list (indices of) quads having dangling exits occurring due to *continue* for statement S

Actions for *break* and *continue*:

$$
\begin{array}{lll}
S & \rightarrow & \textbf{break;} \\
 & & \quad \{\ S.break = makelist(nextinstr); \\
 & & \quad\ \ S.continue = NULL; \\
 & & \quad\ \ S.nextlist = NULL;\ \} \\
 & & \\
S & \rightarrow & \textbf{continue;} \\
 & & \quad \{\ S.continue = makelist(nextinstr); \\
 & & \quad\ \ S.break = NULL; \\
 & & \quad\ \ S.nextlist = NULL;\ \}
\end{array}
$$

Solution: Control Construct Grammar (`break` & `continue`)

Actions to handle break and continue in S:

1. The break and continue statements are expected in loop constructs and are merged until a loop construct translation is reached.
2. On a loop statement translation the dangling exits of all the break statements are passed onto statement nextlist. These indices will be later backpatched with the corresponding index out of the loop.
3. On a loop statement translation the dangling exits of all the continue statements are backpatched with the index for the corresponding start of the loop.

Example for *for* statement translation:

$$
\begin{array}{lll}
S & \rightarrow & \textbf{for (}\ E_1\ \textbf{;}\ M_1\ B\ \textbf{;}\ M_2\ E_2\ N\ \textbf{)}\ M_3\ S_1 \\
& & \quad \{\ backpatch(B.truelist, M_3.instr); \\
& & \quad\ \ backpatch(N.nextlist, M_1.instr); \\
& & \quad\ \ backpatch(S_1.nextlist, M_2.instr); \\
& & \quad\ \ backpatch(S_1.continue, M_2.instr); \\
& & \quad\ \ emit("goto"\ \ M_2.instr); \\
& & \quad\ \ S.nextlist = merge(S_1.break, B.falselist); \\
& & \quad\ \ S.break = NULL; \\
& & \quad\ \ S.continue = NULL;\ \}
\end{array}
$$

Problem: Type Declaration: 1

We have seen that the translation for the variable type declaration grammar is inconvenienced due to the presence of inherited attributes:

$$
\begin{array}{llll}
0: & P & \rightarrow & M\ D \\
1: & D & \rightarrow & T\ V\ \textbf{;}\ D \\
2: & D & \rightarrow & \epsilon \\
3: & V & \rightarrow & V\ \textbf{,}\ \textbf{id} \\
4: & V & \rightarrow & \textbf{id} \\
5: & T & \rightarrow & B \\
6: & B & \rightarrow & \textbf{int} \\
7: & B & \rightarrow & \textbf{float} \\
8: & M & \rightarrow & \epsilon
\end{array}
$$

We have discussed four approaches for handling the inherited attributes:

- **Global Marker**
- **Lazy Action**
- **Bison Stack**
- **Grammar Rewrite**

Using suitable example/s compare and contrast the above approaches.

Problem: Type Declaration: 2

Using the variable type declaration grammar translate the following declarations using the technique as marked. Show the parse tree, the steps in translation, and the symbol table.

1. *By Global Marker*

```
int a, b, c; float d;
```

2. *By Lazy Action*

```
int a;
float d;
int b
int c;
```

3. *By Bison Stack*

```
int a, b;
float c, d;
```

4. *By Grammar Rewrite*

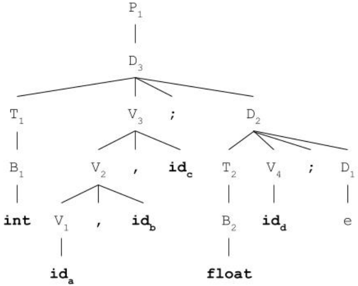
```
int a;
int b;
float c, d;
```

Solution: Type Declaration: 2 (1)

$$
\begin{array}{lcl}
B_1 & \rightarrow & \textbf{int} \\
T_1 & \rightarrow & B_1 \\
\\
V_1 & \rightarrow & id_a \\
V_2 & \rightarrow & V_1\,,\,id_b \\
V_3 & \rightarrow & V_2\,,\,id_c \\
B_2 & \rightarrow & \textbf{float} \\
T_2 & \rightarrow & B_2 \\
\\
V_4 & \rightarrow & id_d \\
D_1 & \rightarrow & \epsilon \\
D_2 & \rightarrow & T_2\,V_4\,;\,D_1 \\
D_3 & \rightarrow & T_1\,V_3\,;\,D_2 \\
P_1 & \rightarrow & D_3
\end{array}
$$

Name	Type	Size	Offset
a	integer	4	0
b	integer	4	4
c	integer	4	8
d	float	8	12

```
offset = 0
B1.type = integer; B1.width = 4;
t = integer; w = 4;
T1.type = integer; T1.width = 4;
offset = 4;
offset = 8;
offset = 12;
B2.type = float; B2.width = 8;
t = float; w = 8;
T2.type = float; T2.width = 8;
offset = 20;
```

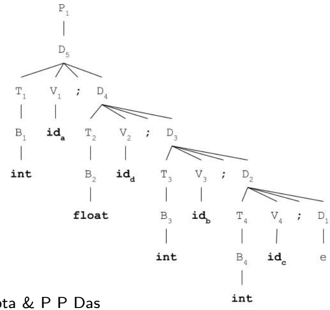


Solution: Type Declaration: 2 (2)

$$
\begin{array}{lcl}
B_1 & \rightarrow & \textbf{int} \\
T_1 & \rightarrow & B_1 \\
V_1 & \rightarrow & id_a \\
B_2 & \rightarrow & \textbf{float} \\
T_2 & \rightarrow & B_2 \\
V_2 & \rightarrow & id_d \\
B_3 & \rightarrow & \textbf{int} \\
T_3 & \rightarrow & B_3 \\
V_3 & \rightarrow & id_b \\
B_4 & \rightarrow & \textbf{int} \\
T_4 & \rightarrow & B_4 \\
V_4 & \rightarrow & id_c \\
D_1 & \rightarrow & \epsilon \\
D_2 & \rightarrow & T_4\, V_4;\, D_1 \\
D_3 & \rightarrow & T_3\, V_3;\, D_2 \\
D_4 & \rightarrow & T_2\, V_2;\, D_3 \\
D_5 & \rightarrow & T_1\, V_1;\, D_4 \\
P_1 & \rightarrow & D_5
\end{array}
$$

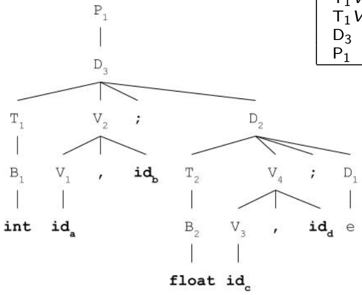
```
B1.type = integer; B1.width = 4;
T1.type = integer; T1.width = 4;
V1.list = {a};
B2.type = float; B2.width = 8;
T2.type = float; T2.width = 8;
V2.list = {d};
B3.type = integer; B3.width = 4;
T3.type = integer; T3.width = 4;
V3.list = {b};
B4.type = integer; B4.width = 4;
T4.type = integer; T4.width = 4;
V4.list = {c};
offset = 0;
```

Name	Type	Size	Offset
a	integer	4	0
d	float	8	4
b	integer	4	12
c	integer	4	16



Solution: Type Declaration: 2 (3)

$$
\begin{array}{lcl}
B_1 & \rightarrow & \textbf{int} \\
T_1 & \rightarrow & B_1 \\
V_1 & \rightarrow & id_a \\
V_2 & \rightarrow & V_1 \, , \, id_b \\
B_2 & \rightarrow & \textbf{float} \\
T_2 & \rightarrow & B_2 \\
V_3 & \rightarrow & id_c \\
V_4 & \rightarrow & V_4 \, , \, id_d \\
D_1 & \rightarrow & \epsilon \\
D_2 & \rightarrow & T_2 \, V_4 ; \, D_1 \\
D_3 & \rightarrow & T_1 \, V_2 ; \, D_2 \\
P_1 & \rightarrow & D_3
\end{array}
$$

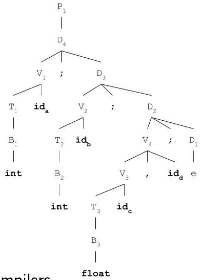


Stack	Action
B_1	$B_1.type$ = integer; $B_1.width = 4$;
T_1	$T_1.type$ = integer; $T_1.width = 4$;
$T_1 \, id_a$	
$T_1 \, V_1$	a.type = integer; a.width = 4;
$T_1 \, V_1 \, , \, id_b$	
$T_1 \, V_2$	b.type = integer; b.width = 4;
$T_1 \, V_2 ; \, B_2$	$B_2.type$ = float; $B_2.width = 8$;
$T_1 \, V_2 ; \, T_2$	$T_2.type$ = float; $T_2.width = 8$;
$T_1 \, V_2 ; \, T_2 \, id_c$	
$T_1 \, V_2 ; \, T_2 \, V_3$	c.type = float; c.width = 8;
$T_1 \, V_2 ; \, T_2 \, V_3 \, , \, id_d$	
$T_1 \, V_2 ; \, T_2 \, V_4$	d.type = float; d.width = 8;
$T_1 \, V_2 ; \, T_2 \, V_4 ; \, D_1$	
$T_1 \, V_2 ; \, D_2$	
D_3	
P_1	

Name	Type	Size	Offset
a	integer	4	0
b	integer	4	4
c	float	8	8
d	float	8	16

Solution: Type Declaration: 2 (4)

$$
\begin{array}{lll}
B_1 & \rightarrow & \textbf{int} \\
T_1 & \rightarrow & B_1 \\
V_1 & \rightarrow & T_1\, id_a \\
B_2 & \rightarrow & \textbf{int} \\
T_2 & \rightarrow & B_2 \\
V_2 & \rightarrow & T_2\, id_b \\
B_3 & \rightarrow & \textbf{float} \\
T_3 & \rightarrow & B_3 \\
V_3 & \rightarrow & T_3\, id_c \\
V_4 & \rightarrow & V_3\,,\, id_d \\
D_1 & \rightarrow & \epsilon \\
D_2 & \rightarrow & V_4\,;\, D_1 \\
D_3 & \rightarrow & V_2\,;\, D_2 \\
D_4 & \rightarrow & V_1\,;\, D_3 \\
P_1 & \rightarrow & D_4
\end{array}
$$




```
offset = 0;
B1.type = integer; B1.width = 4;
T1.type = integer; T1.width = 4;
offset = 4;
V1.type = integer; V1.width = 4;
B2.type = integer; B2.width = 4;
T2.type = integer; T2.width = 4;
offset = 8;
V2.type = integer; V2.width = 4;
B3.type = float; B3.width = 8;
T3.type = float; T3.width = 8;
offset = 16;
V3.type = float; V3.width = 8;
offset = 24;
V4.type = float; V4.width = 8;
```

Name	Type	Size	Offset
a	integer	4	0
b	integer	4	4
c	float	8	8
d	float	8	16

Problem: Type Conversion: 1

Consider the Grammar:

1: $S \rightarrow E_1$ = E_2 ;
2: $E \rightarrow E_1$!= E_2
3: $E \rightarrow E_1$ == E_2
4: $E \rightarrow E_1$ **?** E_2 **:** E_3
5: $E \rightarrow$ **id**
6: $E \rightarrow$ **num**

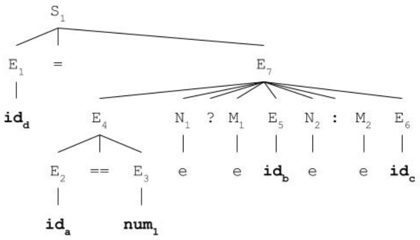
where **id**, **num**, and all E's (except E_1 in rule 4) are of type **int**. E_1 in rule 3 is naturally of type **bool** where an automatic conversion from **int** is carried out in the context.
Using the translation schemes discussed in the lectures of Module 5, translate the following to 3 address. Show the parse trees and attributes in steps.

1. ```
   int a, b, c, d;
   d = a == 1 ? b : c;
   ```
2. ```
   int a, b, c, d;
   d = a ? b : c;
   ```

Solution: Type Conversion: 1 (1)

```
E1.loc = d, E1.type = int
E2.loc = a, E2.type = int
E3.loc = t0, E3.type = int
E4.type = bool
E4.TL = {101}
E4.FL = {102}
N1.NL = {103}
M1.I = {104}
E5.loc = b, E5.type = int
N2.NL = {104}
M2.I = {105}
E6.loc = c, E6.type = int
E7.loc = t1, E7.type = int
I = {106}
I = I U {108} = {106,108}
```

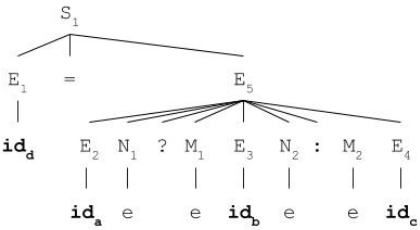
```
100: t0 = 1
101: if a == t0 goto 104
// BP ( E4.TL, M1.I )
102: goto 105     // BP ( E4.FL, M2.I )
103: goto 109     // BP ( N1.NL, 109 )
104: goto 107     // BP ( N2.NL, 107 )
105: t1 = c
106: goto 109     // BP ( I, 109 )
107: t1 = b
108: goto 109     // BP ( I, 109 )
109: d = t1
```



Name	Type	Size	Offset
a	int	4	0
b	int	4	4
c	int	4	8
d	int	4	12
t0	int	4	16
t1	int	4	20

Solution: Type Conversion: 1 (2)

```
E1.loc = d, E1.type = int
E2.loc = a, E2.type = int
N1.NL = {100}
M1.I = {101}
E3.loc = b, E3.type = int
N2.NL = {101}
M2.I = {102}
E4.loc = c, E4.type = int
E5.loc = t0, E5.type = int
I = {103}
I = I U {105} = {103,105}
E2.type = bool
E2.FL = {106}
E2.TL = {107}
```




```
100: goto 106 // BP ( N1.NL, 106 )
101: goto 104 // BP ( N2.NL, 104 )
102: t0 = c
103: goto 108 // BP ( I, 108 )
104: t0 = b
105: goto 108 // BP ( I, 108 )
106: if a == 0 goto 102 // BP ( E2.FL, M2.I )
107: goto 101 // BP ( E2.TL, M1.I )
108: d = t0
```

Name	Type	Size	Offset
a	int	4	0
b	int	4	4
c	int	4	8
d	int	4	12
t0	int	4	16

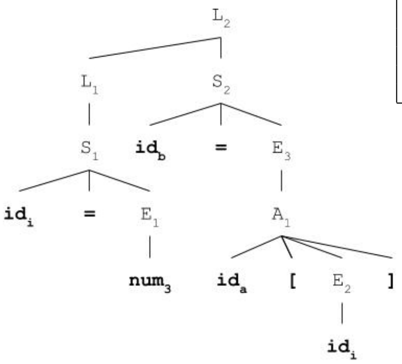
Problem: Array Expression: 1

Using the grammar for expressions with arrays and semantic actions for translation as discussed in the lectures (Module 5), translate the following code snippets. Illustrate with the parse trees, attribute updates, symbol table/s, type expressions, and generated code in every case.

1.
```
int a[5], i, b;
...
i = 3;
b = a[i];
```
2.
```
int a[5], i;
...
i = 0;
a[i + 1] = a[i];
```
3.
```
int a[5];
...
a[0] = 0;
a[1] = a[a[0]] + 1;
```
4.
```
int a[5], b[3][4], c, i, j, k;
...
a[k] = b[i][j] + c;
```

Solution: Array Expression: 1 (1)

```
E1.loc = t0, E1.type = int;
E2.loc = i, E2.type = int;
A1.array = ST[00];
A1.type = T1.elem = int;
A1.loc = t1, A1.loc.type = int;
E3.loc = t2, E3.type = int;
```




```
100: t0 = 3
101: i = t0
102: t1 = i * 4
103: t2 = a[t1]
104: b = t2
```

S.No.	Name	Type	Size	Offset
00	a	T1	20	0
01	i	int	4	20
02	b	int	4	24
03	t0	int	4	28
04	t1	int	4	32
05	t2	int	4	36

T1 = array (5, int)
T1.width = 5 * int.width = 5*4 = 20

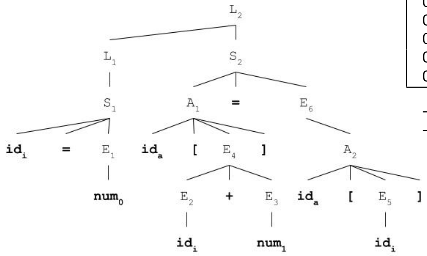
Solution: Array Expression: 1 (2)

```
E1.loc = t0, E1.type = int;
E2.loc = i, E2.type = int;
E3.loc = t1, E3.type = int;
E4.loc = t2, E4.type = int;
A1.array = ST[00];
A1.type = T1.elem = int;
A1.loc = t3, A1.loc.type = int;
E5.loc = i, E5.type = int;
A2.array = ST[00];
A2.type = T1.elem = int;
A2.loc = t4, A2.loc.type = int;
E6.loc = t5, E6.type = int;
```

```
100: t0 = 0
101: i = t0
102: t1 = 1
103: t2 = i + t1
104: t3 = t2 * 4
105: t4 = i * 4
106: t5 = a[t4]
107: a[t3] = t5
```

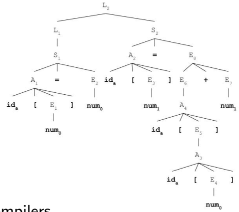
S.No.	Name	Type	Size	Offset
00	a	T1	20	0
01	i	int	4	20
03	t0	int	4	24
04	t1	int	4	28
05	t2	int	4	32
06	t3	int	4	36
07	t4	int	4	40
08	t5	int	4	44

T1 = array (5, int)
T1.width = 5 * int.width = 5*4 = 20



Solution: Array Expression: 1 (3)


```
E1.loc = t0, E1.type = int;
A1.array = ST[00];
A1.type = T1.elem = int;
A1.loc = t1, A1.loc.type = int;
E2.loc = t2, E2.type = int;
E3.loc = t3, E3.type = int;
A2.array = ST[00];
A2.type = T1.elem = int;
A2.loc = t4, A2.loc.type = int;
E4.loc = t5, E4.type = int;
A3.array = ST[00];
A3.type = T1.elem = int;
A3.loc = t6, A3.loc.type = int;
E5.loc = t7, E5.type = int;
A4.array = ST[00];
A4.type = T1.elem = int;
A4.loc = t8, A4.loc.type = int;
E6.loc = t9, E6.type = int;
E7.loc = t10, E7.type = int;
E8.loc = t11, E8.type = int;
```



```
100: t0 = 0
101: t1 = t0 * 4
102: t2 = 0
103: a[t1] = t2
104: t3 = 1
105: t4 = t3 * 4
106: t5 = 0
107: t6 = t5 * 4
108: t7 = a[t6]
109: t8 = t7 * 4
110: t9 = a[t8]
111: t10 = 1
112: t11 = t9 + t10
113: a[t4] = t11
```

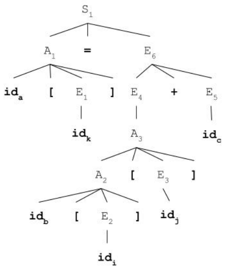
S.No.	Name	Type	Size	Offset
00	a	T1	20	0
01	t0	int	4	20
02	t1	int	4	24
03	t2	int	4	28
04	t3	int	4	32
05	t4	int	4	36
06	t5	int	4	40
07	t6	int	4	44
08	t7	int	4	48
09	t8	int	4	52
10	t9	int	4	56
11	t10	int	4	60
12	t11	int	4	64

T1 = array (5, int)

T1.width = 5 * int.width = 5*4 = 20

Solution: Array Expression: 1 (4)

```
E1.loc = k, E1.type = int;
A1.array = ST[00];
A1.type = T1.elem = int;
A1.loc = t0, A1.loc.type = int;
E2.loc = i, E2.type = int;
A2.array = ST[01];
A2.type = T2.elem = T2';
A2.loc = t1, A2.loc.type = int;
E3.loc = j, E3.type = int;
A3.array = A2.array = ST[01];
A3.type = A2.type.elem = int;
A3.loc = t3, A3.loc.type = int;
E4.loc = t4, E4.type = int;
E5.loc = c, E5.type = int;
E6.loc = t5, E6.type = int;
```




```
100: t0 = k * 4
101: t1 = i * 16
102: t2 = j * 4
103: t3 = t1 + t2
104: t4 = b[t3]
105: t5 = t4 + c
106: a[t0] = t5
```

S.No.	Name	Type	Size	Offset
00	a	T1	20	0
01	b	T2	48	20
02	c	int	4	68
03	i	int	4	72
04	j	int	4	76
05	k	int	4	80
06	t0	int	4	84
07	t1	int	4	88
08	t2	int	4	92
09	t3	int	4	96
10	t4	int	4	100
11	t5	int	4	104

T1 = array (5, int)
T1.width = 5 * int.width = 5*4 = 20
T2 = array (3, array (4,int))
T2' = array (4,int)
T2'.width = 4 * int.width = 4*4 = 16
T2 = array (3, T2')
T2.width = 3 * T2'.width = 3*16 = 48

Problem: Function Declaration & Invocation: 1

Using the function declaration & invocation grammars and semantic actions for translation as discussed in the lectures (Module 5), translate the following code snippets. Illustrate with the parse trees, attribute updates, symbol table/s and generated code in every case.

1.
```
int func1();
...
int a;
...
a = func1();
```
2.
```
int func2(double x, double y, int z);
...
int a;
double p, q, r;
...
a = func2(p + q, q + r, a);
```
3.
```
int f(int a, int b);
int g(int a);
...
int a;
...
a = f(a, g(a));
```

Solution: Function Declaration & Invocation: 1 (1)

Declaration

```
T1.type = int;
F_opt.ST = ST(func1);
```

ST(global)

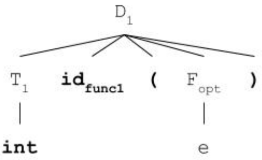
Name	Type	Size	Offset	Nested Table
func1	void → *int*	0	...	ST(func1)

ST(func1)

Name	Type	Size	Offset	Nested Table
_retval	int	4	0	null

ST(?)

Name	Type	Size	Offset	Nested Table
a	int	4	0	null
t0	int	4	4	null

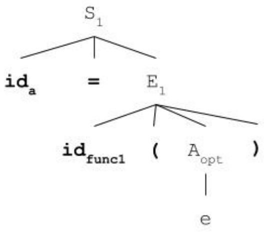


Invocation

```
A_opt.list = 0;
E1.loc = t0, E1.type = int;
```

TAC

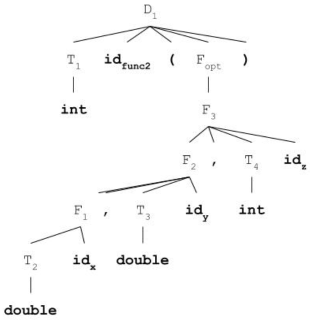
```
100: t0 = call func1, 0;
101: a = t0
```



Solution: Function Declaration: 1 (2)

Declaration

```
T1.type = int;
T2.type = double;
F1.ST = ST(func2);
T3.type = double;
F2.ST = F1.ST = ST(func2);
T4.type = int;
F3.ST = F2.ST = ST(func2);
F_opt.ST = F3.ST = ST(func2);
```



ST(global)

Name	Type	Size	Offset	Nested Table
func2	double * double * int → *int*	0	...	ST(func2)

ST(func2)

Name	Type	Size	Offset	Nested Table
x	double	8	0	null
y	double	8	8	null
z	int	4	16	null
_retval	int	4	20	null

Solution: Function Invocation: 1 (2)

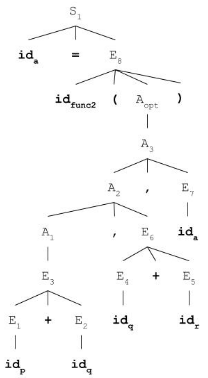
Invocation

```
E1.loc = p; E1.type = double;
E2.loc = q; E2.type = double;
E3.loc = t0; E3.type = double;
A1.list = { (t0,double) };
E4.loc = q; E4.type = double;
E5.loc = r; E5.type = double;
E6.loc = t1; E6.type = double;
A2.list = { (t0,double),(t1,double) }
E7.loc = a; E7.type = int;
A3.list = { (t0,double),(t1,double),(a,int) }
A_opt.list = A3.list;
E8.loc = t2; E8.type = int;
```

ST(?)

Name	Type	Size	Offset	Nested Table
a	int	4	0	null
p	double	8	4	null
q	double	8	12	null
r	double	8	20	null
t0	double	8	28	null
t1	double	8	36	null
t2	int	4	44	null

```
100: t0 = p + q
101: t1 = q + r
102: param t0
103: param t1
104: param a
105: t2 = call func2, 3
106: a = t2
```

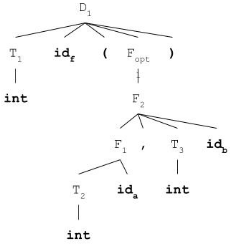


Solution: Function Declaration: 1 (3)

Declaration

```
T1.type = int;
T2.type = int;
F1.ST = ST(f);
T3.type = int;
F2.ST = F1.ST = ST(f);
F_opt1.ST = F2.ST = ST(f);

T4.type = int;
T5.type = int;
F3.ST = ST(g);
F_opt2.ST = F3.ST = ST(g);
```



ST(global)

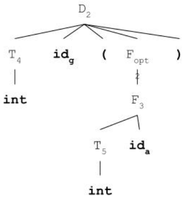
Name	Type	Size	Offset	Nested Table
f	int * int $\rightarrow$ *int*	0		ST(f)
g	int $\rightarrow$ *int*	0		ST(g)

ST(f)

Name	Type	Size	Offset	Nested Table
a	int	4	0	null
b	int	4	4	null
_retval	int	4	8	null

ST(g)

Name	Type	Size	Offset	Nested Table
a	int	4	0	null
_retval	int	4	4	null



Solution: Function Invocation: 1 (3)

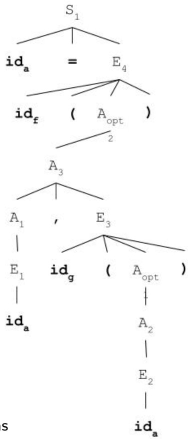
Invocation

```
E1.loc = a; E1.type = int;
A1.list = { (a,int) }
E2.loc = a; E2.type = int;
A2.list = { (a,int) }
A_opt1.list = A2.list;
E3.loc = t0, E3.type = int;
A3.list = { (a,int), (t0,int) }
A_opt2.list = A3.list;
E4.loc = t1, E4.type = int;
```

```
100: param a
101: t0 = call g, 1
102: param a
103: param t0
104: t1 = call f, 2
105: a = t1
```

ST(?)

Name	Type	Size	Offset	Nested Table
a	int	4	0	null
t0	int	4	4	null
t1	int	4	8	null



Practice Problems

Using the appropriate grammars and semantic actions for translation as discussed in the lectures (Module 5), translate the following code snippets. Illustrate with the parse trees, attribute updates, symbol table/s and generated code in every case.

```
int a[50];

int sum(int n) {
    int i, t;

    t = 0;
    for(i = 0; i < n; i = i + 1)
        t = t + a[i];

    return t;
}

int main() {
    int n, i, s;

    n = 10;
    for(i = n - 1; i >= 0; i = i - 1)
        a[i] = n - i;

    s = sum(n);

    return 0;
}
```