STATIC SEMANTICS
&
INTERMEDIATE CODE GENERATION

First Version Developed by
Amitabha Sanyal and Supratim Biswas

Revised by Supratim Biswas
January 2017

Part of Course Material
CS 324 : Compiler Design / Language Processors
CS 302 : Implementation of Programming Languages

Originally Designed under Project IMPACT
Funded by World Bank

Department of Computer Science & Engineering
Indian Institute of Technology, Bombay

BASIC CONCEPTS AND ISSUES

WHAT IS STATIC SEMANTIC ANALYSIS?

Static semantic analysis ensures that a program is correct, in that it conforms to certain extra-syntactic rules.

The analysis is called

Static: because it can be done by examining the program text,

and

Semantic: because the properties analysed are beyond syntax, i.e. they cannot be captured by context free grammars.

Examples of such analyses are:

a. type analysis,

b. name and scope analysis,

c. declaration processing

and the techniques used for static semantic analysis can also be used for

d. intermediate code generation.

BASIC CONCEPTS AND ISSUES

TYPE ANALYSIS

The type of an object should match that expected by its context.

Examples of issues involved in type analysis are:

1. Is the variable on the left hand side of an assignment statement `compatible' with the expression on the right hand side?

2. Is the number of actual parameters and their types in a procedure call compatible with the formal parameters of the procedure declaration?

3. Are the operands of a built-in operator of the right type?

If not, can they be forced (coerced) to be so?

What is the resultant operator, then, to be interpreted as?

For example, the expression **3 + 4.5** can be interpreted as:

a. Integer addition of 3 and 4 (4.5 truncated).

b. Real addition of 3.0 (3 coerced to real) and 4.5.

Note: The notion of `type compatibility' is non-trivial. For example, in the declaration

```
type ptr = ^ object;
var a : ^ object;   b : ptr;
```

are a and b type compatible?

BASIC CONCEPTS AND ISSUES

NAME AND SCOPE ANALYSIS

The issues are:

1. What names are visible at a program point? Which declarations are associated with these names?

2. Conversely, from which regions of a program is a name in a declaration visible?

EXAMPLES

1. In Pascal, the following is not allowed:

```
begin
    ...
    go to 10;
    ...
    while E do
        begin
        ...
      10: ...
        end
    ...
end;
```

Here the label 10 is the name. It is only visible in the block in which it is defined.

Example 2.

```
program S;
    var a,x:
    procedure R;
        var i:
        procedure T;
            var m,n:
        body of T;
    body of R;
    procedure E;
    body of E;
    procedure Q;
        var k,v:
        function P(y,z: ):
          var i, j:
        body of P;
    body of Q;
body of S.
```

How to find the visibility of names at various program points for languages that permit nested declaration of procedures of the kind given above?

These rules are prescribed by the language designers.

NAME AND SCOPE ANALYSIS

Static scoping rules : The names that are visible at a program point are :

1. The names declared locally in the procedure, of which this program point is a part of.
2. The names of all enclosing procedures.
3. The names declared immediately within such procedures.
4. A name that is declared in more closely nested procedures overrides the same name in an outer procedure.

- The names visible from the body of P : y, z, i, j, k, v, a, x, P, Q, E, R and S.
- The procedure P is visible from P itself and Q.

Program Point	**Locally declared names**	**Names of a enclosing Procedures**	**Declarations of names immediately within such procedures**
Body of T	m, n, T	R, S	i, a, x, E, Q
Body of R	i, T, R	S	a, x, E, Q
Body of E	E	S	a, x, R, E, Q
Body of P	i, j, y, z, P	Q, S	k, v, a, x, R, E
Body of Q	k, v, P, Q	S	a, x, R, E
Body of S	a, x, S, R, E, Q	-	-

BASIC CONCEPTS AND ISSUES

DECLARATION PROCESSING

Declaration processing involves:

1. *Uniqueness checks*: An identifier must be declared uniquely in an declaration.

2. *Symbol Table updation*: As declarations provide most of the information regarding the attributes of a name, this information must be recorded in a data structure called the symbol table. The main issues here are:

a. What information must be entered into the symbol table, and how should such information be represented?

b. What should be the organization of the symbol table itself.

3. *Address Resolution*: As declarations are processed, the addresses of variables may also be computed. This address is in the form of an offset relative to the beginning of the storage allocated for a block (refer to the module on runtime environments for more details).

BASIC CONCEPTS AND ISSUES

INTERMEDIATE CODE GENERATION

The front end translates the program into an intermediate representation, from which the back end generates the target code.

There is a choice of intermediate representations such as abstract syntax trees, postfix code and three address code, Gimple.

EXAMPLE

Three address code generated for the assignment statement x := A[y,z] < b, where A is a 10 x 20 array of integers, and b and x are reals is:

1. t1 := y * 20
2. t1 := t1 + z
3. t2 := addr(A) - 21
4. t3 := t2[t1]
5. t4 := intoreal(t3)
6. if t4 < b goto 9
7. t5 := 0
8. goto 10
9. t5 := 1
10. x:= t5

SYNTAX DIRECTED ANALYSIS

Can we always represent the properties stated above through a context free grammar? There are theoretical results to show that the answer is in the negative.

1. The language {w c w | w is a string from some alphabet Σ } can be thought of as an abstraction of languages with a single identifier declaration (the first occurrence of w), which must be declared before use (the second occurrence of w).

This language is not context free.

2. The language $\{a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1\}$ is an abstraction of languages with two procedures and their corresponding calls, which require that the number of formal parameters (represented by the a's and b's) be the same as the number of actual parameters (c's and d's).

This language is not context free.

However, we would like to do static analysis in a *syntax directed* fashion, i.e. along with the process of parsing.

SYNTAX DIRECTED ANALYSIS

SYNTAX DIRECTED DEFINITION

Associate with each terminal and non-terminal a set of values called *attributes*. The evaluation of attributes takes place during parsing, i. e., when we use a production of the form, $A \rightarrow X\ Y\ Z$, for *derivation* or *reduction*, the attributes of X, Y and Z could be used to calculate the attributes of A.

EXAMPLE: For the grammar shown below, suppose we wanted to generate intermediate code:

$S \rightarrow id := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow - E_1$

$E \rightarrow id$

It is convenient to have the following attributes:

S.code - The intermediate code associated with S.

E.code - The intermediate code associated with E.

E.place - The temporary variable which holds the value of E.

id.place - The lexeme corresponding to the token id.

SYNTAX DIRECTED ANALYSIS

SYNTAX DIRECTED DEFINITION: EXAMPLE

The grammar is augmented with the following semantic rules:

$S \rightarrow id := E$	S.code := E.code || gen(id:place ':=' E:place);
$E \rightarrow E_1 + E_2$	E.place := newtemp;
	E.code := E1.code || E2.code || gen(E.place ':=' E1.place '+' E2.place);
$E \rightarrow E_1 * E_2$	E.place := newtemp;
	E.code := E1.code || E2.code || gen(E.place ':=' E_1.place '*' E2.place);
$E \rightarrow - E_1$	E.place := newtemp;
	E.code := E1.code || gen(E.place ':=' 'uminus' E1.place);
$E \rightarrow id$	E.place := id.place; E.code := ' ';

where

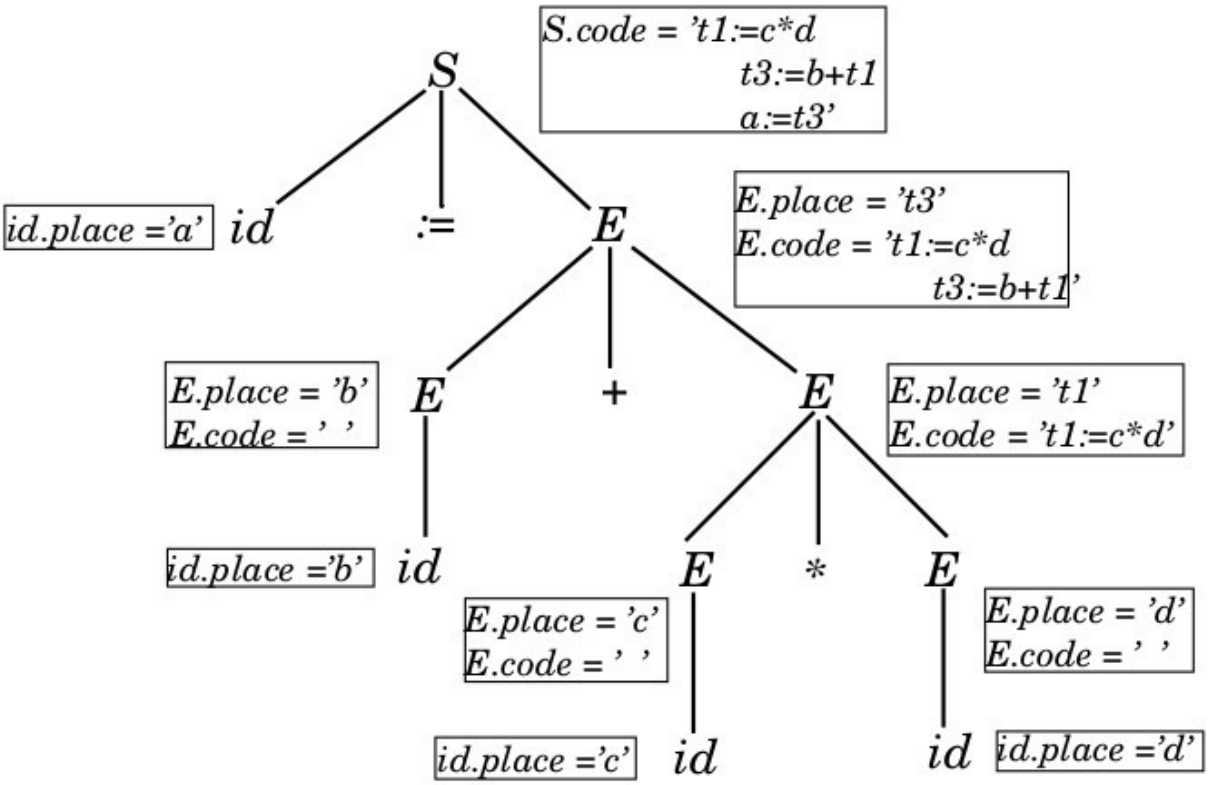
1. newtemp is a function which generates a new temporary variable name, each time it is invoked.

2. '||' is the concatenation operator.

3. gen( ) evaluates its non-quoted arguments, concatenates the arguments in order, and returns the resulting string.

SYNTAX DIRECTED ANALYSIS

The attributes of the various nonterminals for the statement a := b + c * d is



NOTE:

1. Unlike the example, the attribute evaluation should proceed along with the construction of the parse tree. This would introduce restrictions on the possible relations between attribute values.

2. Nothing has been said about the implementational details of attribute evaluation.

3. The attributes of terminals are usually supplied by the lexical analyser.

SYNTAX DIRECTED ANALYSIS

SYNTAX DIRECTED DEFINITION: FORMALIZATION

A syntax directed definition is an **augmented context free grammar**, where each production **A** → α, is associated with a set of semantic rules of the form

b := f (c_1, c_2, ..., c_k), where f is a function, and

1. b is a *synthesized attribute* of A and **c_1, c_2, ..., c_k** are attributes belonging to the grammar symbols of α , or

2. b is an *inherited attribute* of one of the grammar symbols α,

and **c_1, c_2, ..., c_k** are attributes belonging to A or α.

In either case, we say that the attribute b depends on the attributes **c_1, c_2, ..., c_k** .

In the previous example, S.code, E.code, and E.place were all synthesized attributes.

SYNTAX DIRECTED ANALYSIS

WHY INHERITED ATTRIBUTES?

1. For a variable declaration

var_dec → T : L

L → L_1 , id

L → id

T → integer

T → real

If we wanted to record the type of each variable, the following semantic rules would do the task.

var_dec → T : L	{ L.type := T.type; }
L → L_1 , id	{ L_1.type := L.type; id.type := L.type; }
L → id	{ id.type := L.type; }
T → integer	{ T.type := integer; }
T → real	{ T.type := real; }

Here L.type and id.type are inherited attributes and T.type is synthesized.

SYNTAX DIRECTED ANALYSIS

2. Consider a (simplified) procedure declaration

p_decl → p_name var_decl body

Suppose we wanted to find the list of variables accessible (environment) inside the body of a procedure. This will be the list of variables accessible just outside the procedure declaration, augmented by the variables declared locally in the procedure itself. In such a case, we can have the following attributes:

p_decl.env - environment just outside the procedure declaration.

body.env - environment in the body.

var_decl.list - list of local variables declared

Clearly, the required syntax directed definition is:

p_decl → p_name var_decl body

{ body.env := p_decl.env ∪ var_decl.list }

As is obvious, body:env is an inherited attribute.

SYNTAX DIRECTED ANALYSIS

TRANSLATION SCHEMES

A translation scheme is a refinement of a syntax directed definition in which:

(i) Semantic rules are replaced by actions.

(ii) The exact point in time when the actions are to be executed iis specified. This is done by embedding the actions within the right hand side of productions.

In the translation scheme

$$A \rightarrow X_1 \ X_2 \ . \ . \ X_i \ \{action\} \ X_{i+1} \ . \ . \ . \ X_n$$

action is executed after completion of the parse for X_i.

EXAMPLE : A translation scheme to map infix expressions into postfix expressions. Expressions involve binary *addop* operators {+, –} and token num.

For the infix expression : 10 + 7 – 3

Desired translation : 10 7 + 3 –

Using expression grammar writing rules for attributes of *addop*

$$E \rightarrow E \ addop \ T \ | \ T$$

$$T \rightarrow num$$

We however use an equivalent grammar that works for both parsing methods.

SYNTAX DIRECTED ANALYSIS

TRANSLATION SCHEMES

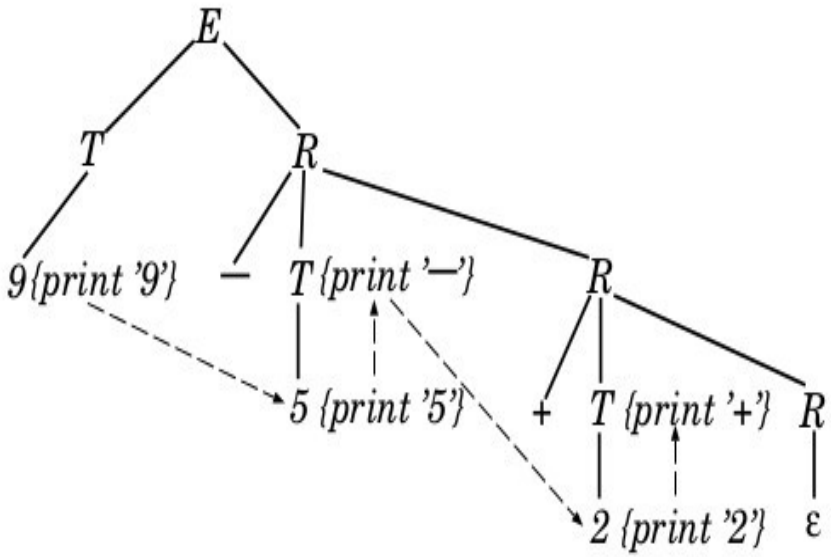
Consider the translation scheme given below.

$E \rightarrow T\ R$

$R \rightarrow$ *addop* T { print(addop.lexeme) } R_1 | ε

$T \rightarrow$ num { print(num.val) }

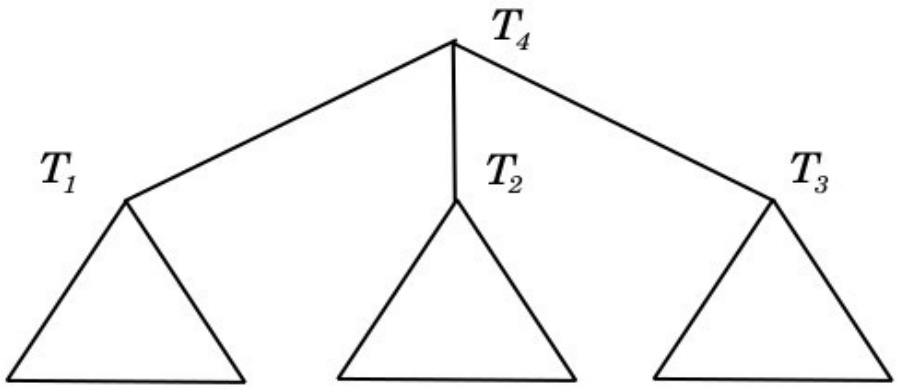
The order of execution of actions for the string 9 – 5 + 2 is shown by dotted arrows. Examine how this translation scheme works for top-down and bottom-up parsers and also whether it changes the properties of the operators.



SYNTAX DIRECTED ANALYSIS

RESTRICTIONS ON ATTRIBUTE DEPENDENCES

Certain restrictions must be placed on the dependences between attributes, so that they can be evaluated along with parsing. It is interesting to note that top-down parsers or bottom-up parsers would construct the parse tree shown in the figure in the order T1, T2, T3, T4.



Clearly an attribute of T2 cannot depend on any attribute of T3.

The extent of restriction gives rise to two classes of syntax directed definitions,

a. **S-attributed definition**

b. **L-attributed definition**

S-ATTRIBUTED DEFINITIONS

A translation scheme is *S-attributed* if:

(i) Every non-terminal has synthesized attributes only.

(ii) All actions occur on the right hand side of productions.

The syntax directed definition shown below is an example of an S-attributed definition:

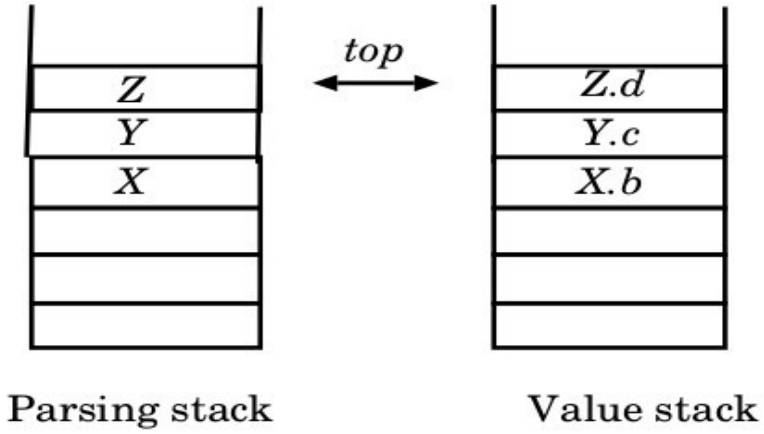
$S \rightarrow id := E$	S.code := E.code || gen(id:place ':=' E:place);
$E \rightarrow E_1 + E_2$	E.place := newtemp; E.code := E1.code || E2.code || gen(E.place ':=' E1.place '+' E2.place);
$E \rightarrow E_1 * E_2$	E.place := newtemp; E.code := E1.code || E2.code || gen(E.place ':=' E_1.place '*' E2.place);
$E \rightarrow - E_1$	E.place := newtemp; E.code := E1.code || gen(E.place ':=' 'uminus' E1.place);
$E \rightarrow id$	E.place := id.place; E.code := ' ';

S-ATTRIBUTED DEFINITIONS

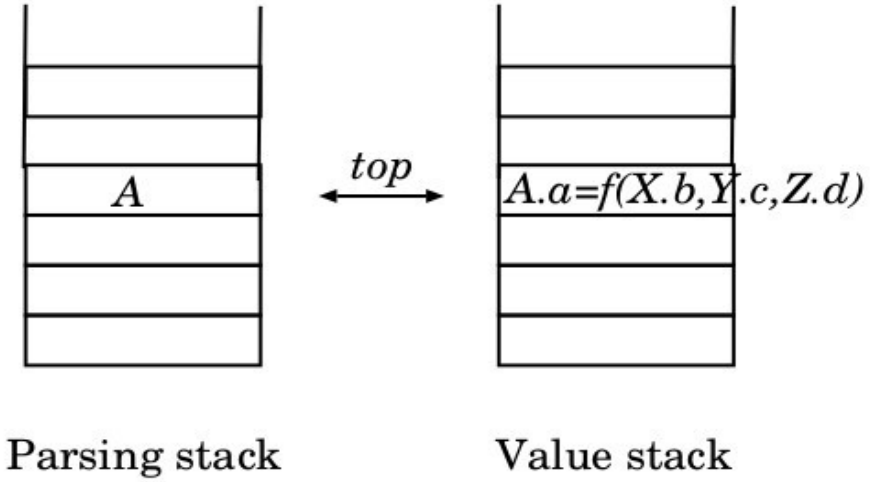
IMPLEMENTATION OF S-ATTRIBUTED DEFINITION

Assume a bottom up parsing strategy. We augment the parsing stack with a value stack (val). If location i in the parsing stack contains a grammar symbol A. then val[i] will contain all the attributes of A.

Prior to a reduction using a production A → X Y Z :



If an attribute a of A is defined as f(X.b, Y.c, Z.d), then, after the reduction:



S-ATTRIBUTED DEFINITIONS

We can now replace the semantic rules by actions which refer to the actual data structures used to hold and manipulate the attributes.

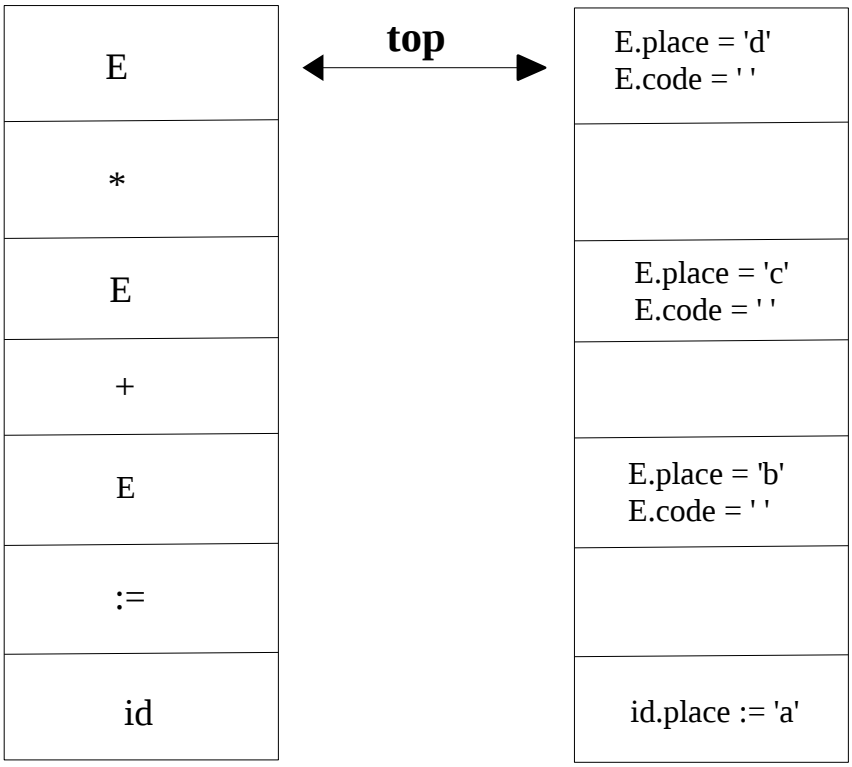
Let each element of val be a record with two fields val[i].code and val[i].place. Then the syntax directed definition is :

S → id := E	val[top−2].code := val[top].code || gen(val[top−2].place ':=' val[top].place);
E → E1 + E2	T := newtemp; val[top−2].code := val[top−2].code || val[top].code || gen(T ':=' val[top – 2].place) '+' val[top].place; val[top – 2].place := T;
E → E1 * E2	T := newtemp; val[top−2].code := val[top−2].code || val[top].code || gen(T ':=' val[top – 2].place) '*' val[top].place; val[top – 2].place := T;
E → − E_1	T := newtemp; val[top−1].code := val[top].code || gen(T ':=' 'uminus' val[top].place) '*' val[top].place; val[top – 1].place := T;
E → id	val[top].place := val[top].place; val[top].code := ' ';

S-ATTRIBUTED DEFINITIONS

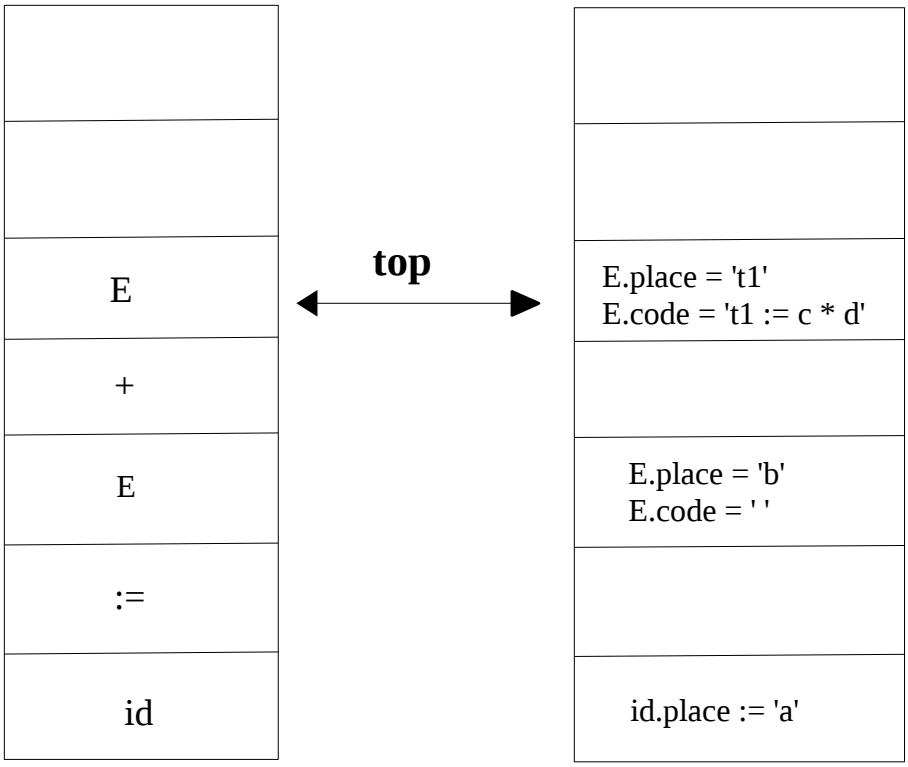
EXAMPLE : Before reduction using the production E → E1 * E2

Input string a := b + c * d



S-ATTRIBUTED DEFINITIONS

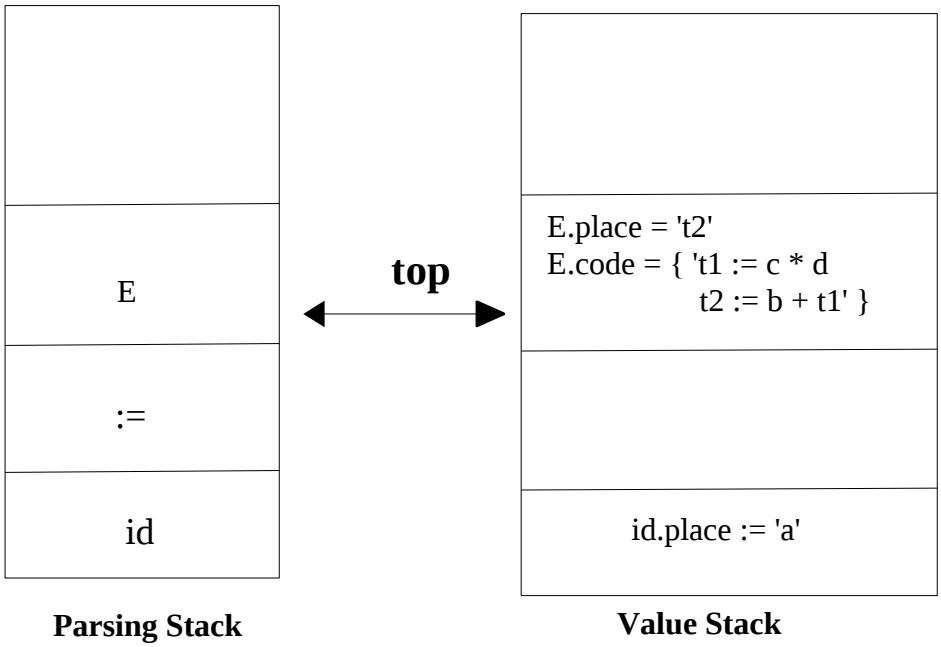
Stack contents immediately after reduction using the production E → E1 * E2



The actions specified in the S-attributed scheme are used to obtain the configuration shown above.

S-ATTRIBUTED DEFINITIONS

Stack contents immediately after reduction by $E \rightarrow E1 + E2$



METHODS FOR SEMANTIC ANALYSIS

We summarize the diiferent methods introduced inthis module for performing semantic analysis in a syntax directed manner.

Carefully distinguish between

1. *Syntax directed definition* : A context free grammar augmented with semantic rules. No assumptions are made regarding when the semantic rules are to be evaluated.

2. *Attribute grammar* : A syntax directed definition in which the functions used in the semantic rules are free of side effects.

3. *Translation scheme* : A syntax directed definition with actions instead of semantic rules. Actions are embedded within the right hand side of productions, their positions indicating the time of their evaluation during a parse.

4. *Implementation of a syntax directed definition* : A translation scheme in which the actions are described in terms of actual data structures (e.g. value stack) instead of attribute symbols.