Compilers

3rd year
Spring term

Alfonso Ortega: alfonso.ortega@uam.es
Enrique Alfonseca: enrique.alfonseca@uam.es

Lesson 5: semantic analysis

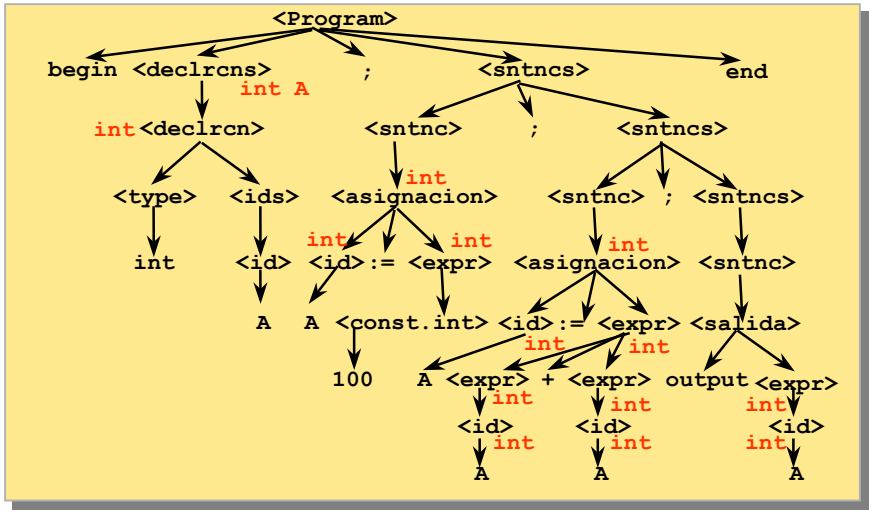
Semantic analysis

Introduction: motivation

- Set of independent routines used by the morphological and syntactic analyser.
- The input is the parse tree, either fully built, or in construction, and these routines add semantic error checking:
  - They test type restrictions in expressions and assignations.
  - They check other semantic limitations, e.g. declaration and initialisation of identifiers before their use, type correspondence, number of arguments in function calls, etc.
- We can image that, when the semantic analysis ends, the syntactic tree will have some annotations about its semantics.
- These annotations serve to:
  - Determine the **semantic correctness** of the program.
  - Prepare the next step: the **code generation**

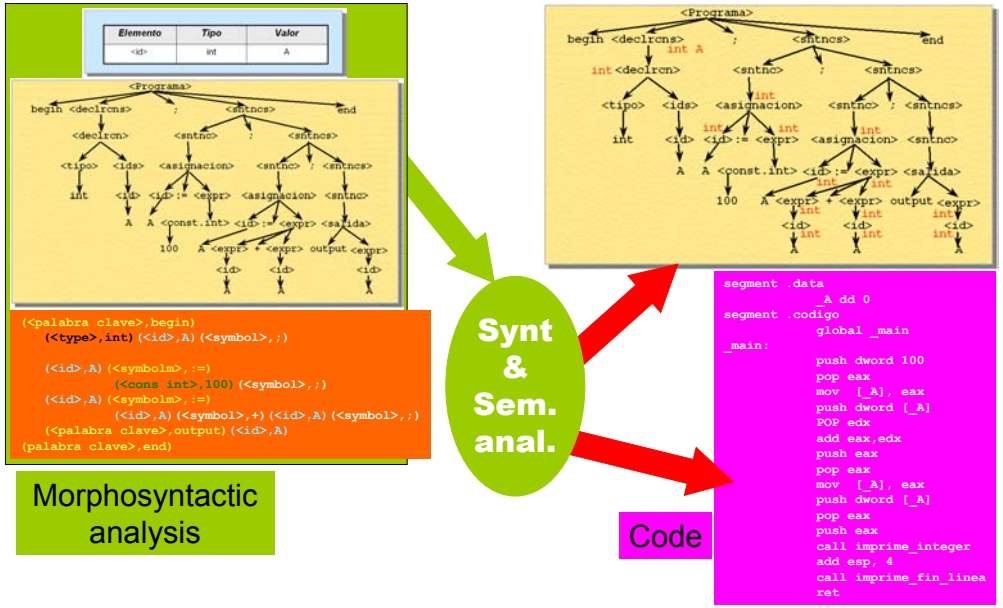
Semantic analysis

Introduction: motivation



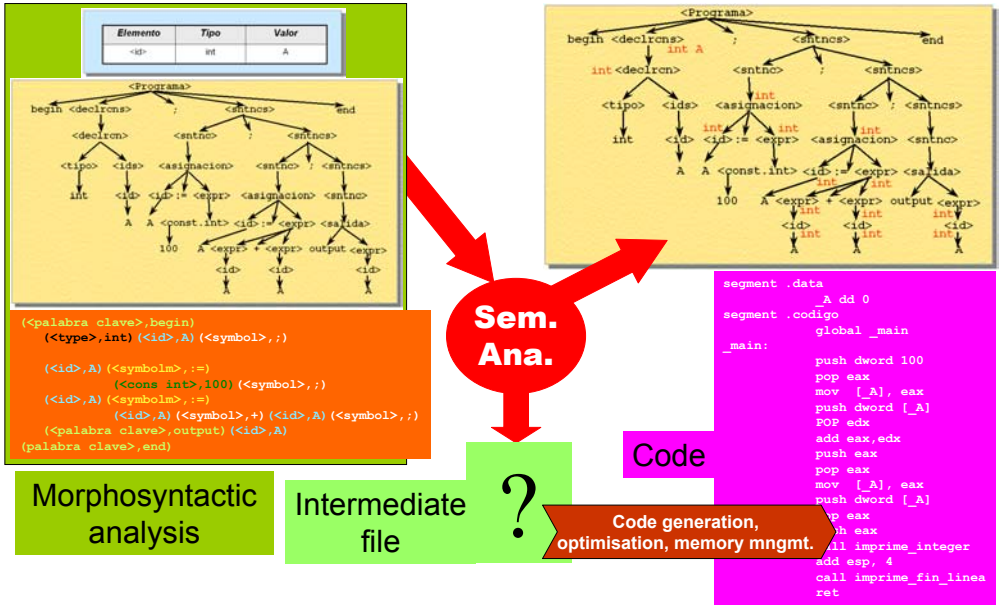
Semantic analysis

Introduction: Semantic analysis in 1-pass compilers



Semantic analysis

Introduction: Semantic analysis in compilers with 2 o more passes



Semantic analysis

Introduction: use of the stack

- The syntactic analyser is an “enriched variation” of a pushdown automata that recognises the language studied.
- We have seen in detail how a stack is used in most kinds of analysers: LR(0), SLR(1), LALR(1), LR(1), LL(1) and with precedence grammars.
- The semantic analyser will also use a stack, called **semantic stack**, to store the “semantic annotations” for each of the syntactic elements analysed.
  - The semantic stack can be the same as the syntactic stack.

Semantic analysis

Introduction: attributes

- The procedure which is most widely used consists of adding, to each node in the parse tree, the information about the semantics of the program that we are going to need.
- This information is usually called **attributes**.
- Along this lesson, we are going to formalise the idea of extending the grammar of the language:
  - Adding attributes to the symbols in the grammar.
  - Providing methods for calculating the value of the attributes of a symbol, in function of the attributes of other symbols.

Attributes: informal description

Examples

- Consider the subset of the rules that describe the arithmetic expressions in the high-level programming languages.
- To treat correctly these expressions, most programming languages have ways to specify:
  - The type of the result of the expressions (integer, real, etc.)
  - The resulting value.
- In this way, each symbol can have a type and a value associated.
- The following are some illustrative examples.

Attributes: informal description

Examples

- Consider the following grammar for arithmetic expressions. Note that it is ambiguous, so there may be conflicts in the design of some of the analysers, which might be solved by assigning priorities to the operators.

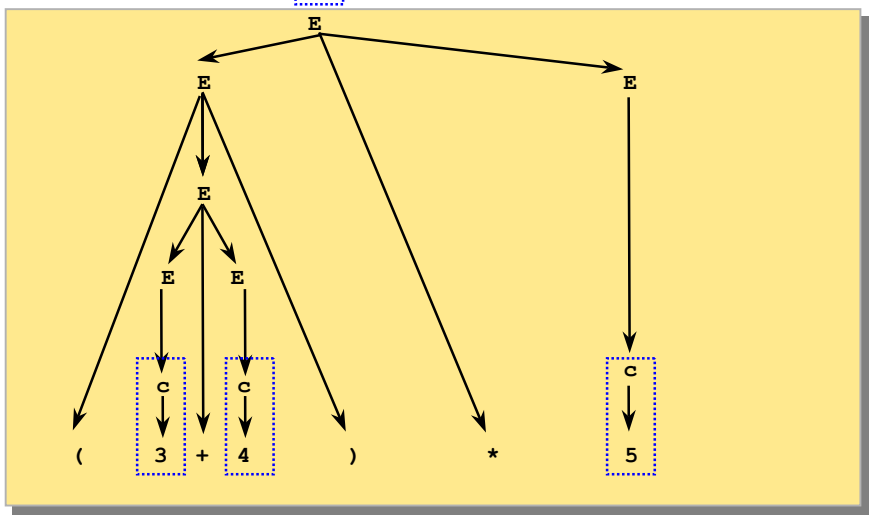
G_E={{+,*,(,),c,i}, {E},
{
E→E+E,
E→E-E,
E→-E,
E→E*E,
E→E/E,
E→E^E,
E→(E),
E→i,
E→c,
},
E}

- We shall assume that:
  - `i` refers to any identifier, and `c` refers to any numeric constant.

Attributes: informal description

Examples

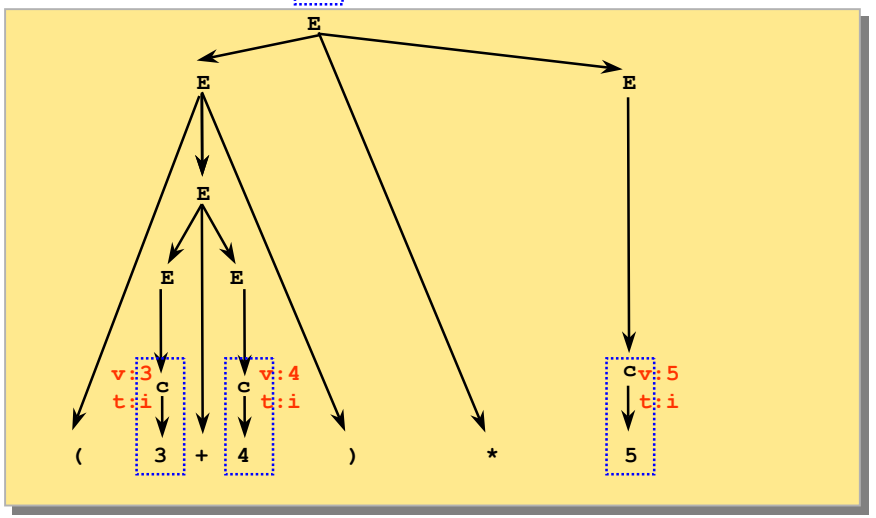
- Expression: (3+4)*5 indicates the work performed by the lexical analyser



Attributes: informal description

Examples

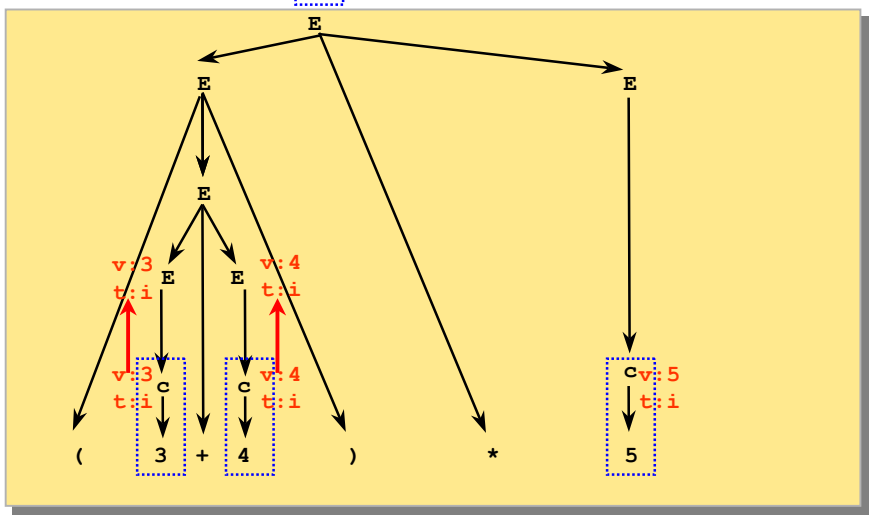
- Expression: (3+4)*5 indicates the work performed by the lexical analyser



Attributes: informal description

Examples

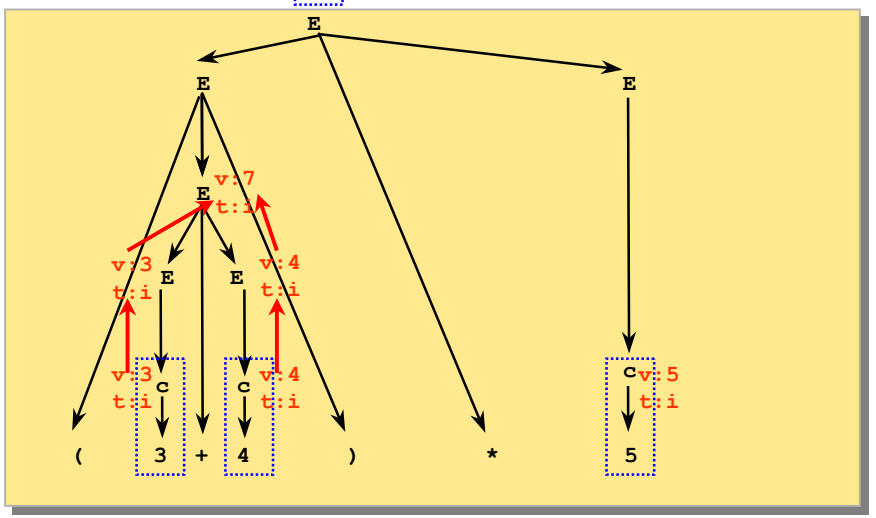
- Expression: (3+4)*5 indicates the work performed by the lexical analyser



Attributes: informal description

Examples

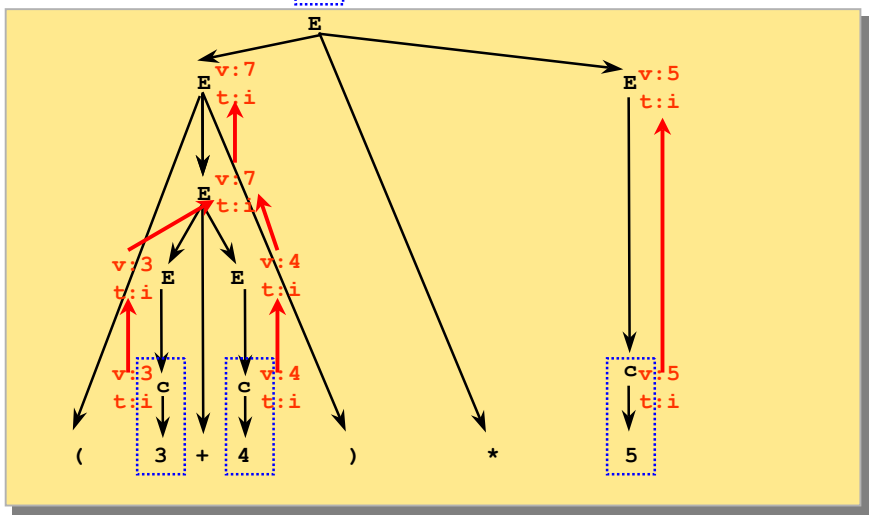
- Expression: (3+4)*5 indicates the work performed by the lexical analyser



Attributes: informal description

Examples

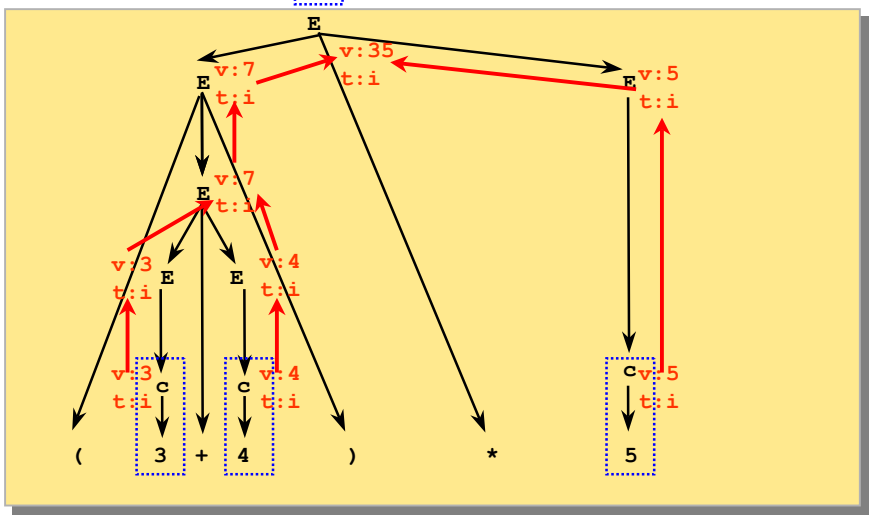
- Expression: (3+4)*5 indicates the work performed by the lexical analyser



Attributes: informal description

Examples

- Expression: (3+4)*5 indicates the work performed by the lexical analyser



Attributes: informal description

Example 1: Conclusions

- Observe that, in this case, if we want to obtain the correct value of the whole expression, the "value" attribute should be propagated from the bottom up.
- From the point of view of the rules,
  - The information associated to the left-hand side of the rule is calculated from the information associated to the symbols in the right-hand side of the rule.
- For instance, in the following rule:

  E→E+E
- The semantics would be described from the following expression:

  E→E+E and **E.value = E.value+E.value**
- In order to distinguish between the three symbols E in that rule, we can use sub-indices **i** and **d**, as follows:

  $E \rightarrow E_i + E_d$ and $E.value = E_i.value + E_d.value$

Attributes: informal description

Examples

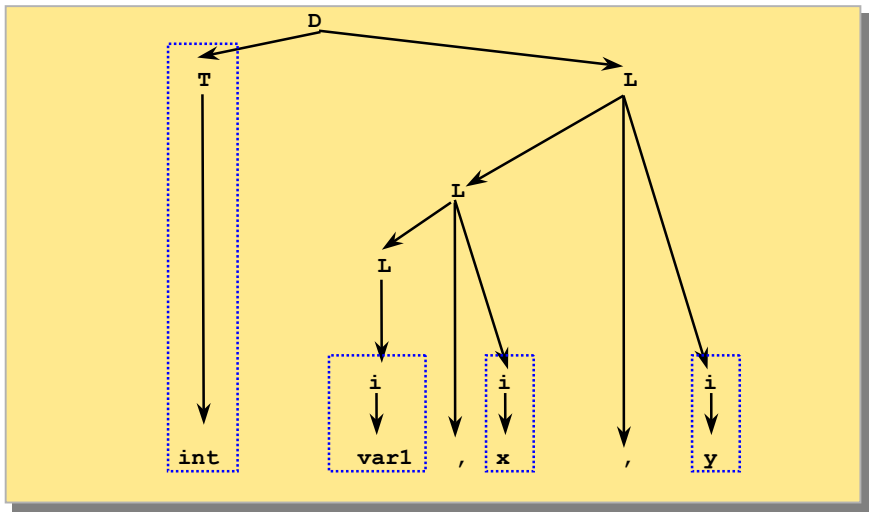
- Consider the rules for declaring variables:

  G_E={{int, real, i}, {D,T,L},
  {
  D→TL
  T→int
  T→real
  L→L,i
  L→i
  },
  D}
- We shall assume that:
  - **i** refers to identifiers.
  - **D** represents a declarative sentence (a declaration).
  - **T** indicates the type of the variables.
  - **L** represents a list of comma-separated identifiers (,)

Attributes: informal description

Examples

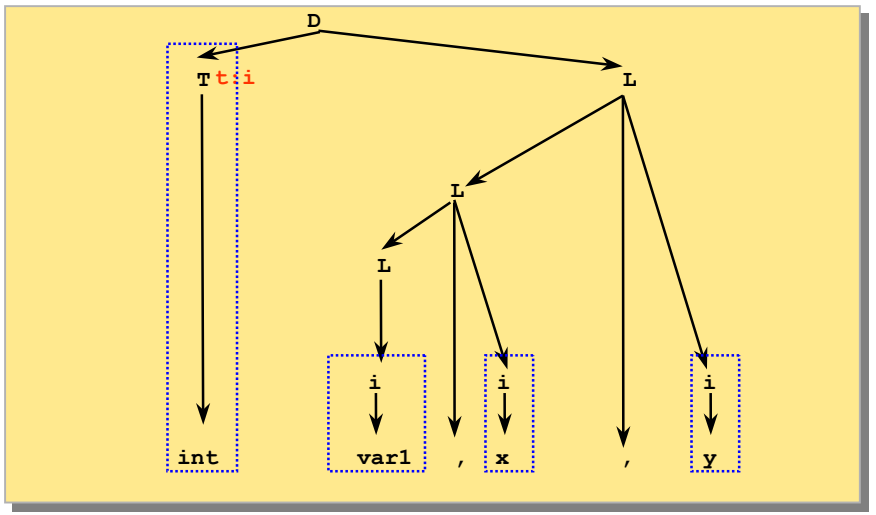
- In the following declaration: **int var1, x, y**



Attributes: informal description

Examples

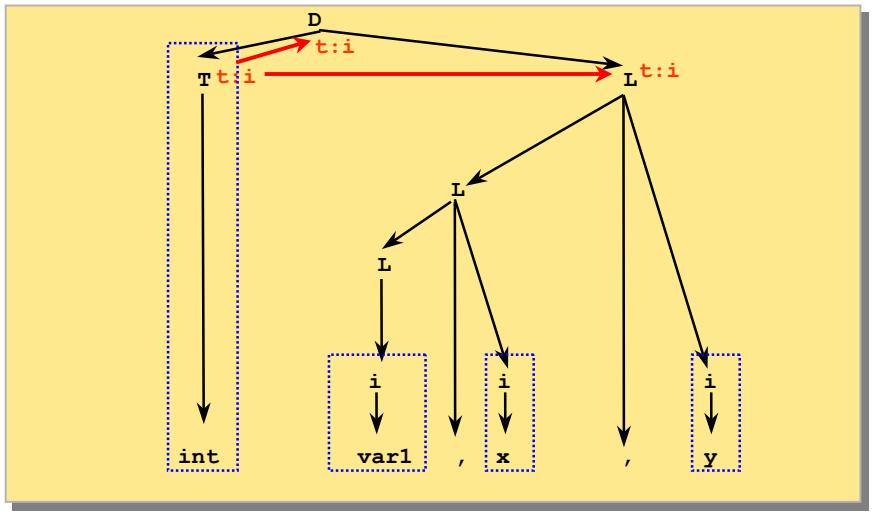
- In the following declaration: **int var1, x, y**



Attributes: informal description

Examples

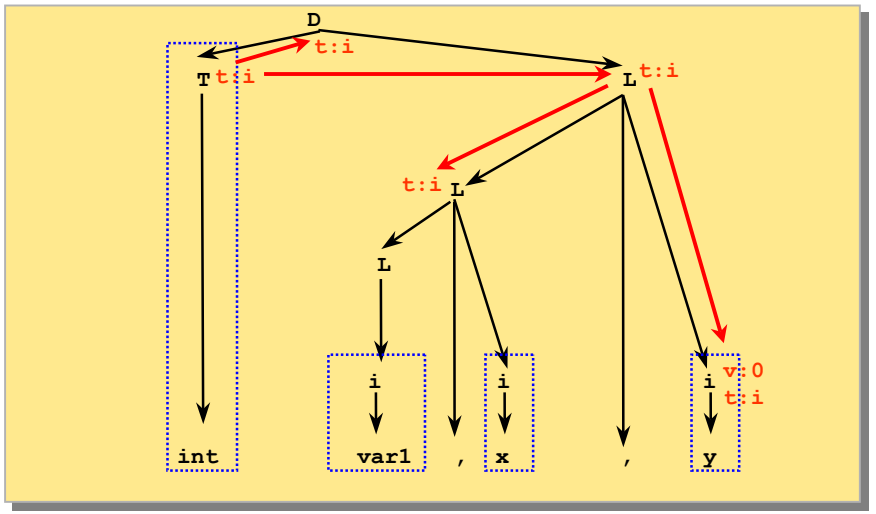
- In the following declaration: `int var1, x, y`



Attributes: informal description

Examples

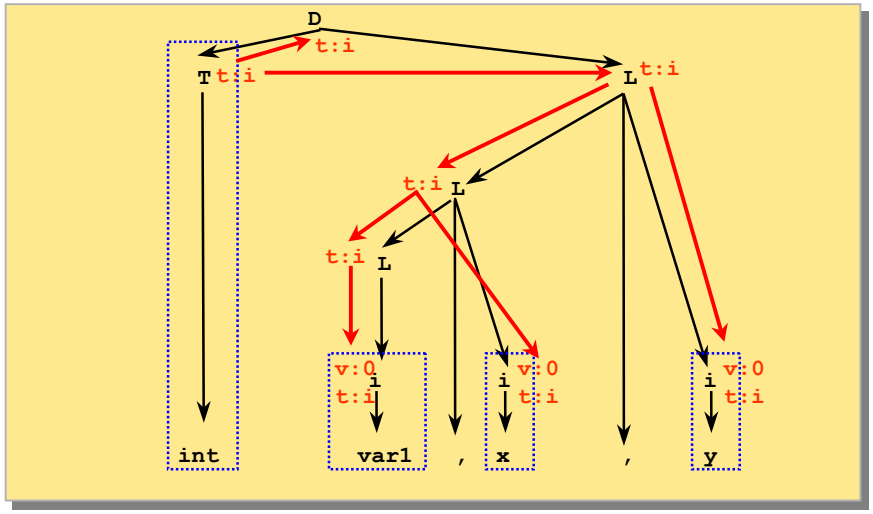
- In the following declaration: `int var1, x, y`



Attributes: informal description

Examples

- In the following declaration: `int var1, x, y`



Attributes: informal description

Example 2: conclusions

- In this case, it seems reasonable that, when the parser finds the type (the reserved word `int`) then the type should propagate to all the identifiers in the list, and to the complete declarative sentence.

- We can express this process in the following way:

`D→TL` and `D.type = T.type; and L.type = D.type`

Attributes: informal description

Kinds of attributes, and attribute propagation

- The previous examples have illustrated two kinds of attributes:
  - **Synthesised attributes**:
    - Those used in the cases in which the attributes of the left-hand side are calculated in function of the attributes in the right-hand side.
    - An example is the **value** attribute in the example or arithmetic expressions:
      - $E \rightarrow E_i + E_d$ y `E.value = Ei.value + Ed.value`
  - **Inherited attributes**:
    - Those used in the cases in which the attributes of a symbol have to be calculated from the attributes of a sibling in the parse tree.
    - An example is the **type** attribute, in the second example:
      - `D→TL y D.type = T.type; y L.type = D.type`
- We shall call propagation to the process of calculating the value of the attributes of a symbol in function of the values of the attributes of other symbols in the grammar.

Syntax-directed Definitions

Informal idea

- "**Syntax-directed definitions**" can be defined as:

An **extension of context-independent grammars**, with the following additions

- The symbols (terminals and non-terminals) are extended with:
  - **Attributes**, which will be used by the modules for semantic analyses and code generation.
  - Examples of these attributes:
    - The identifiers can be extended with their name, type and value.
    - Constants and expressions can have a type and value associated.
- The rules are extended with:
  - **Actions for manipulating those attributes.**
  - For instance, the rule that describes an expression can have the semantic action that checks the correctness of the types of the sub-expressions, and the resulting type of the operator, and the calculation of the resulting value.

Syntax-directed Definitions

Informal idea

- In other words,
  - Each symbol has a list of attributes associated. These will be the semantic values for the symbol.
    - In the example of arithmetic expressions, the following would be the extended alphabets with attributes.

`{+,*,(,),c{value,type},i{value,type}}, {E{value,type}}`

    - In the example of the declaration of variables, the alphabets of terminals and non-terminals are:

`{{int{type}, real{type}, i{type}}, {D{type},T{type},L{type}}`

    - We are going to assume that the terminal symbols only have the attributes calculated by the morphological analyser.
      - These attributes are usually called synthesised attributes.

Syntax-directed Definitions

Informal idea

- Each rule has a set of semantic actions that indicate how we can calculate the semantic values of the non-terminal symbols.
  - Each semantic action can be represented as an "assignation statement", in which the attribute of the non-terminal symbol at the left-hand side of the rule is assigned the result of an expression (or a function call), whose arguments are the attributes of the symbols at the right-hand side of the rule.
  - We can represent that each symbol has associated a structure with different fields, like a struct in C. So, we can refer to the attribute"attr" of the symbol "A" as "A.attr".

Syntax-directed Definitions

Informal idea

- The following is the example with arithmetic expressions:

G_E={{+,*,(,),c{value,type}(1),i{value,type}(1)}, {E{value,type}},
{
E→E_i+E_d {E.value:= E_i.value+E_d.value, E.type:= E_i.type(2)},
E→E_i-E_d {E.value:= E_i.value-E_d.value, E.type:= E_i.type(2)},
E→-E_d, {E.value:= -E_d.value, E.type:= E_d.type},
E→E*E {E.value:= E_i.value*E_d.value, E.type:= E_i.type(2)},
E→E_i/E_d, {E.value:= E_i.value/E_d.value, E.type:= E_i.type(3)}
E→E_i^E_d {E.value:= E_i.value^E_d.value, E.type:= E_i.type(3)},
E→(E_d) {E.value:= E_d.value, E.type:= E_i.type},
E→i {E.value:= i.value, E.type:= i.type},
E→c {E.value:= c.value, E.type:= c.type}
},
E}

- (1) These values are provided by the morphological analyser
- (2) We can place here the type checks.
- (3) We can place here type checks, and additional error detections, such as making sure that the second argument of the division is not 0.

Syntax-directed Definitions

Informal idea

- The following is the example with the declarations of variables:

G_E={ {{int{type}, real{type}, i{type}}(1),
{D{type},T{type},L{type}},
{
D→TL {L.type:=T.type},
T→int {T.type:=integer},
T→real {T.type:=real},
L→L_d,i {L_d.type:=L.type},
L→i
},
D}

- (1) Values provided by the morphological analyser

Syntax-directed Definitions

Informal idea

- We can also add semantic actions with a different effect to that of calculating the values of other semantic attributes.
- Whenever this is needed, it can be implemented as creating a new attribute (fictitious or nameless) for a symbol, which will not be assigned any value by the function associated.
  - In this last example, we might add a new action to modify the symbols table when we are determining the type of an identifier.

G_E={ {{int{type}, real{type}, i{type}}(1),
{D{type},T{type},L{type}},
{
D→TL {L.type:=T.type},
T→int {T.type:=integer},
T→real {T.type:=real},
L→L_d,i {L_d.type:=L.type},
L→i
},
D}

Syntax-directed Definitions

Informal idea

- Imagine that the action of inserting an identifier in the symbols table is performed by the following function:

`add_type(SymbolsTable* pT, Identifier i, Type type)`

- Where:
  - `pT` is a pointer to the symbols table.
  - `i` is the token returned by the morphological analyser.

G_E={ {{int{type}, real{type}, i{type}}(1),
{D{type},T{type},L{type}},
{
D→TL {L.type:=T.type},
T→int {T.type:=integer},
T→real {T.type:=real},
L→L_d,i {L_d.type:=L.type,add_type(&Table,i,L.type)},
L→i {add_type(&Table,i,L.type)}
},
D}

Syntax-directed Definitions

Examples

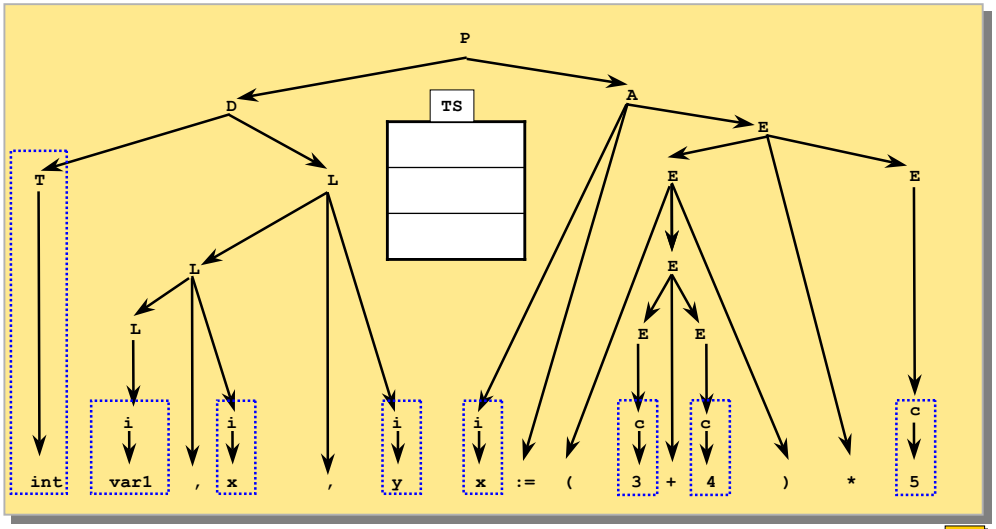
- Let us consider now a slightly more complicated grammar, which includes the ones that we have studied until now. This grammar will allow us to declare identifiers, and to assign them the result of expressions:

G_E={{int{type}, real{type}, i{type,value}, +, *, (,)},
{D{type},T{type},L{type},E{value,type},A{type},P},
{ P→DA, A→i:=E {"i∈SymTab"∧E.type=i.type⇒A.type:=i.type}
D→TL {L.type:=T.type},
T→int {T.type:=integer},
T→real {T.type:=real},
L→L_d,i {L_d.type:=L.type,add_type(&T,i,L.type)},
L→i {add_type(&T,i,L.type)},
E→E_i+E_d {E.value:= E_i.value+E_d.value, E.type:= E_i.type},
E→E_i-E_d {E.value:= E_i.value-E_d.value, E.type:= E_i.type},
E→-E_d, {E.value:= -E_d.value, E.type:= E_d.type},
E→E*E {E.value:= E_i.value*E_d.value, E.type:= E_i.type},
E→E_i/E_d, {E.value:= E_i.value/E_d.value, E.type:= E_i.type}
E→E_i^E_d {E.value:= E_i.value^E_d.value, E.type:= E_i.type},
E→(E_d) {E.value:= E_d.value, E.type:= E_i.type},
E→i {E.value:= i.value, E.type:= i.type},
E→c {E.value:= c.value, E.type:= c.type}},
P}

Syntax-directed Definitions

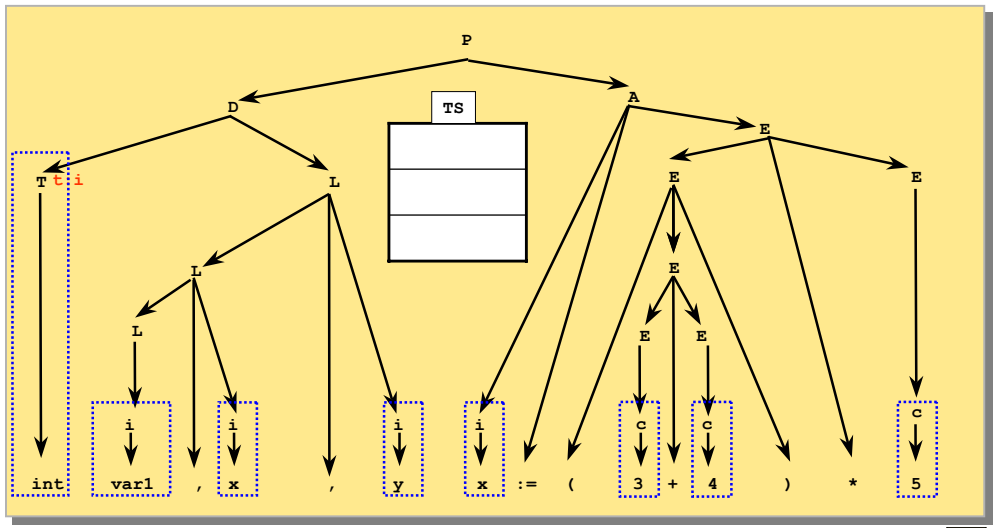
Examples

- The following is the analyses of `int var1, x, y  x:=(3+4)*5`



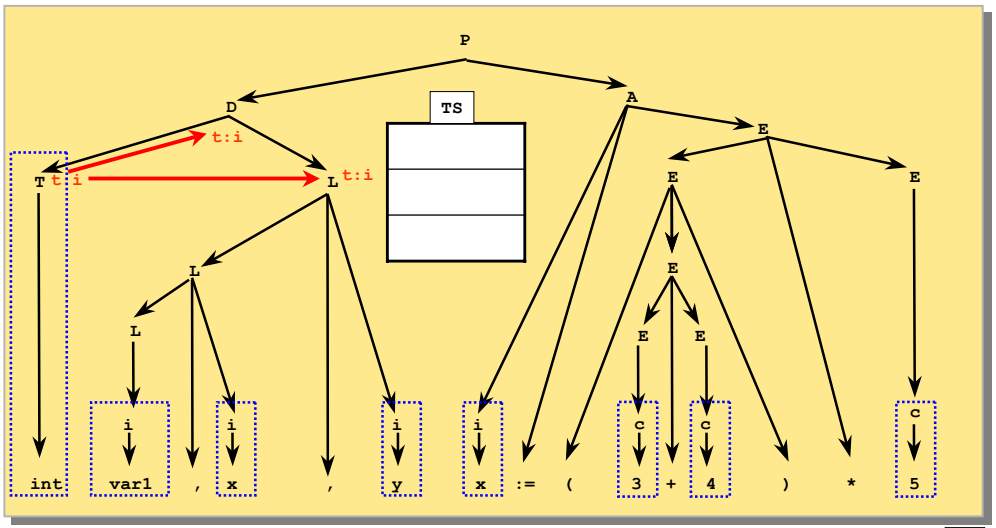
Syntax-directed Definitions

Examples



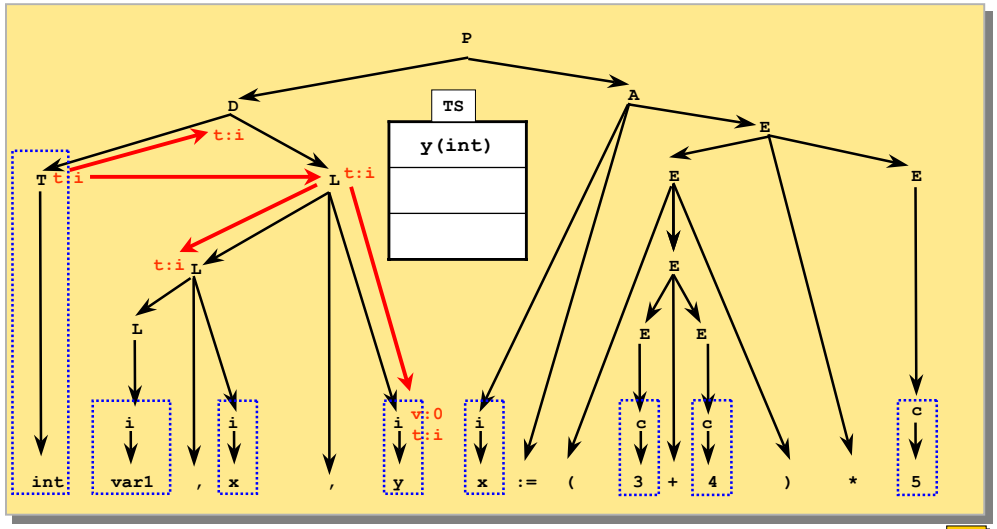
Syntax-directed Definitions

Examples



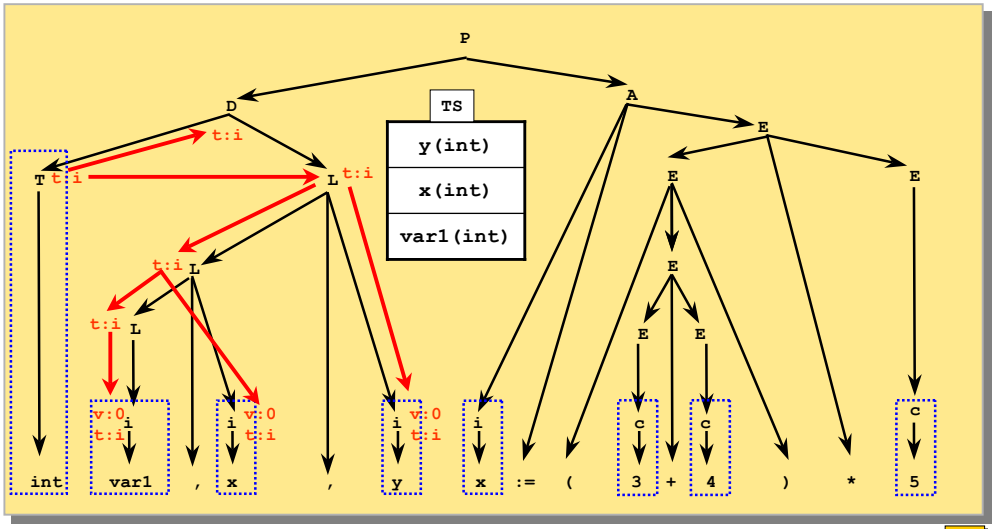
Syntax-directed Definitions

Examples



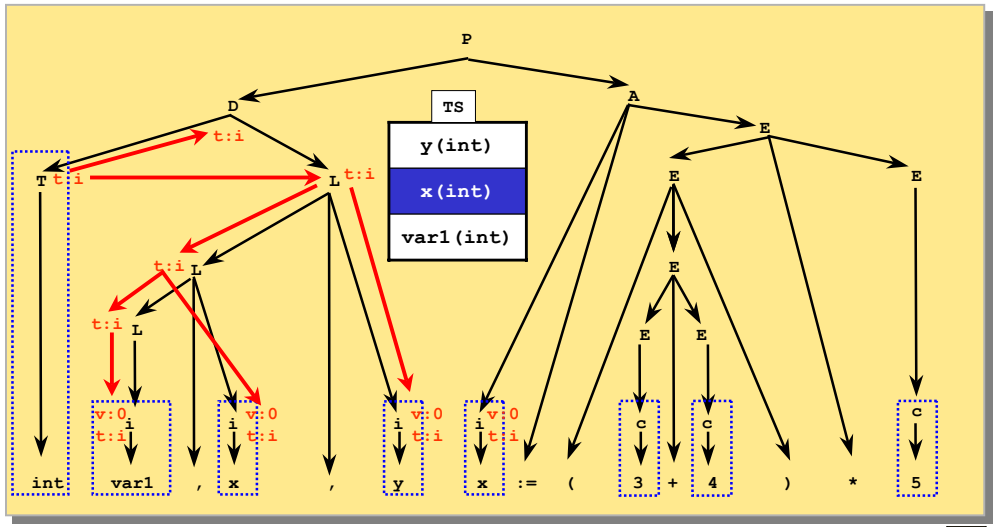
Syntax-directed Definitions

Examples



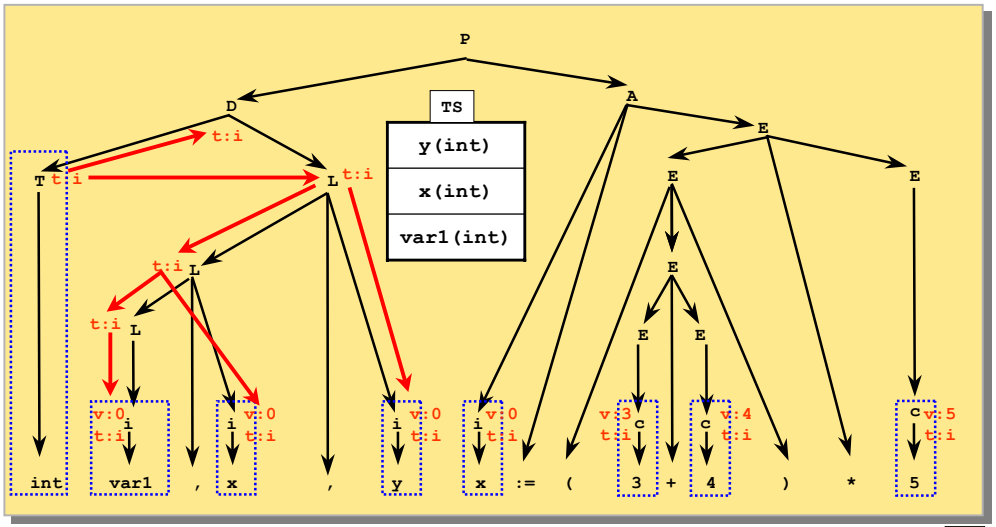
Syntax-directed Definitions

Examples



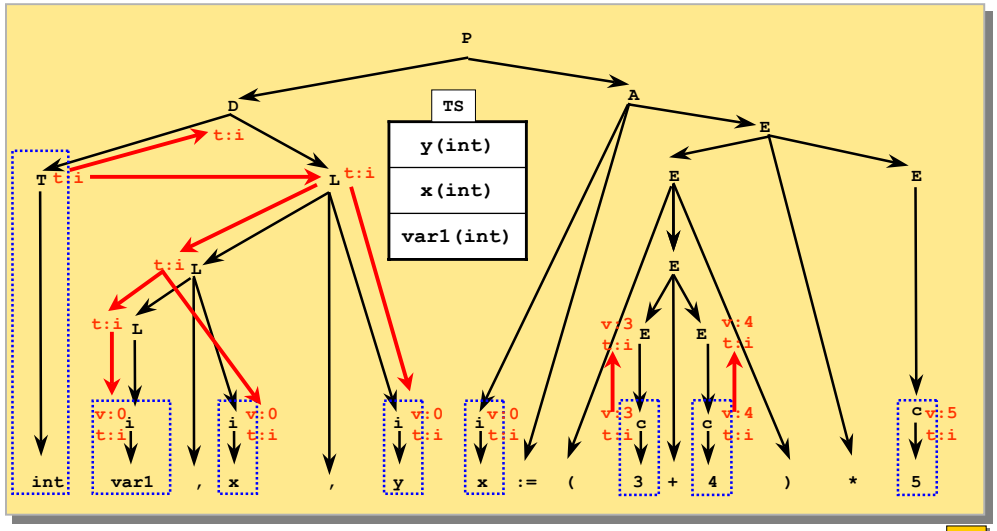
Syntax-directed Definitions

Examples



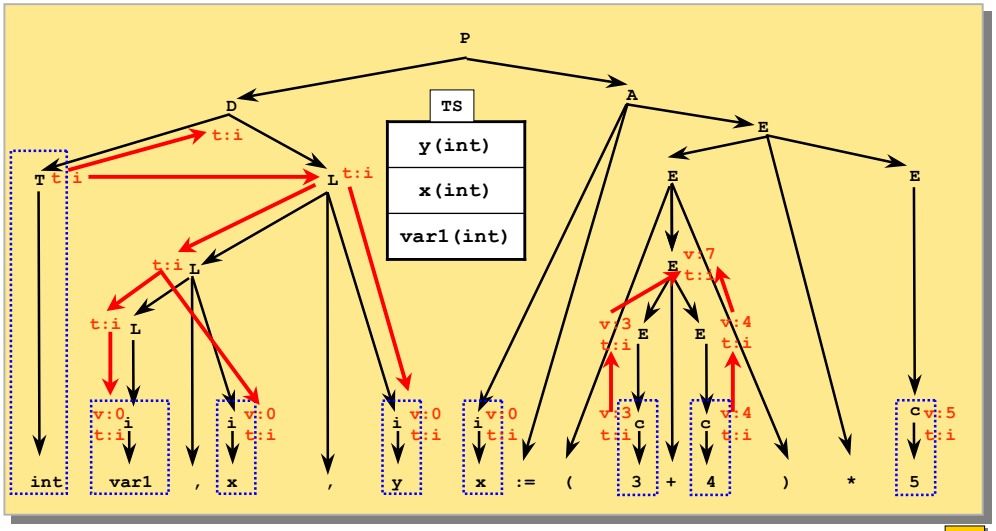
Syntax-directed Definitions

Examples



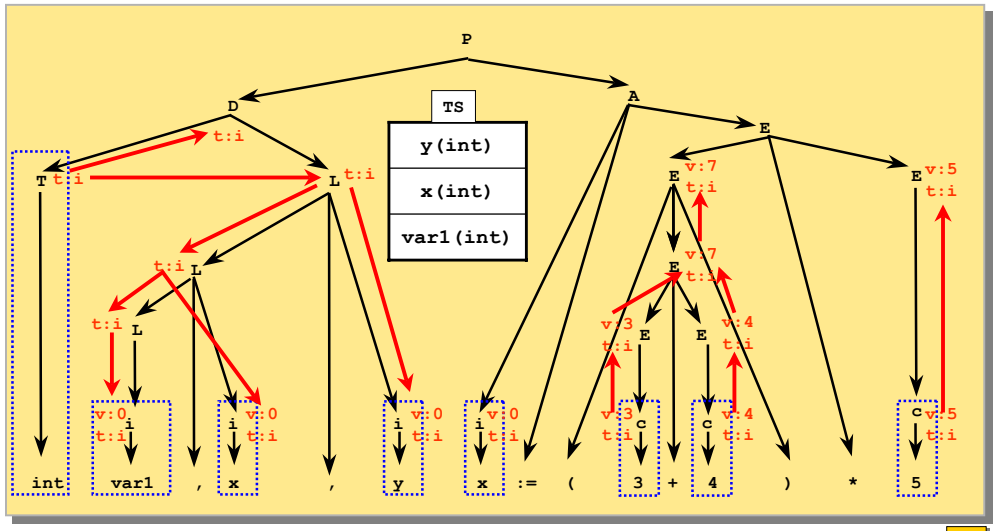
Syntax-directed Definitions

Examples



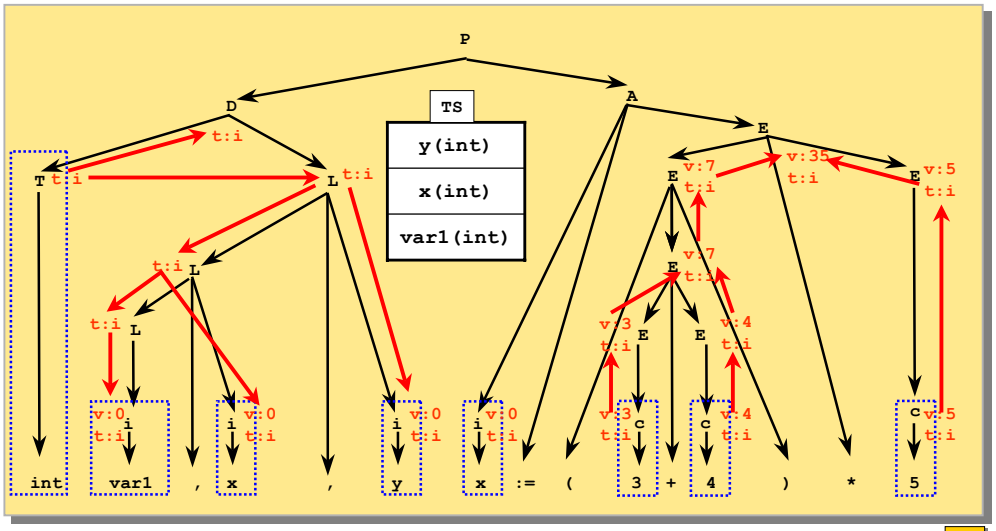
Syntax-directed Definitions

Examples



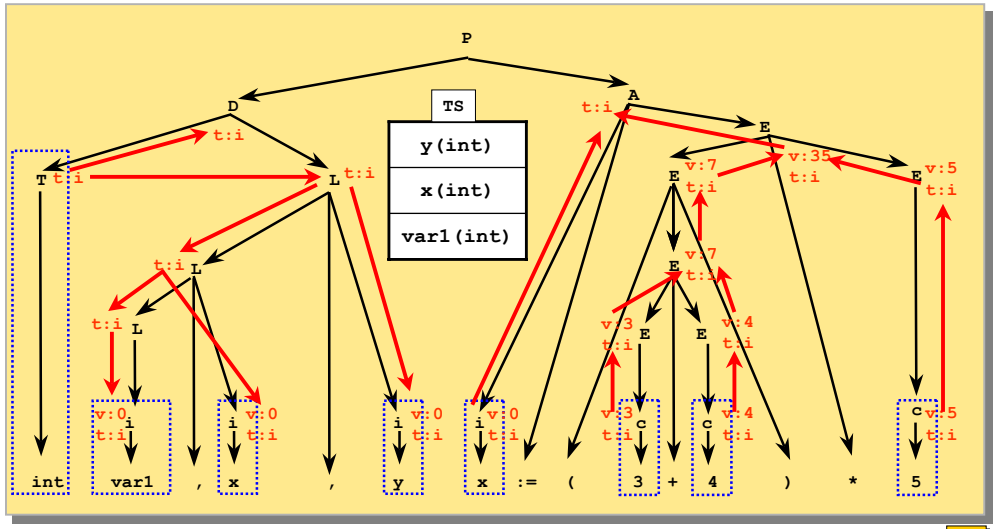
Syntax-directed Definitions

Examples



Syntax-directed Definitions

Examples



Syntax-directed Definitions

Conclusions

- The propagation of attributes is like adding annotations to the nodes in the syntactic parse tree.
- This formalism makes it easy to determine the order in which we need to perform the semantic actions to obtain the correct result.
- However, in real situations, we may not have the parse tree, and it is not always adequate to generate it completely before starting annotating it.
- We need to decide, therefore:
  - **How to determine the order in which the tree nodes will be visited** so we can calculate all the semantic values correctly.
  - **How to perform the semantic actions in the nodes** during the syntactic analysis, to obtain an adequate compiler with respect to power and efficiency.
    - We shall see that some attributes are more suitable for top-down analysis, and others for bottom-up analysis.

Attribute Grammars

Formal definition

- An **attributes grammar** is a syntax-directed definition, in which the semantic actions do not have side effects. They can only be used to calculate the values of attributes.
  - Actions such as introducing an identifier in the symbols table would not be allowed.
- The sides effects could be treated by creating fictitious attributes.
  - From now, we are going to use, indistinctly, the terms **attributes grammar**, and **syntax-directed definitions**.

Attribute Grammars

Introduction to the most general technique

- The most general techniques used for semantic analyses with attributes grammar is the following:
  1. To determine the dependences between some attributes and the others, by studying the semantic rules.
  2. To build the parse tree.
  3. To determine an order between the nodes (by annotating the tree with a numeric sequence). It will be the correct ordering for calculating the semantic attributes.
  4. To establish a path along the tree that respects the previous ordering, and to apply it.
  5. The result of the evaluation of the semantic actions in the previous ordering will be the correct semantic analysis of the program.

Attribute Grammars

Introduction to the most general technique

1. To determine the dependences between some attributes and the others, by studying the semantic rules.
   - The following approach will be followed:
     1. For every semantic action,
     2. If the action is of the following type

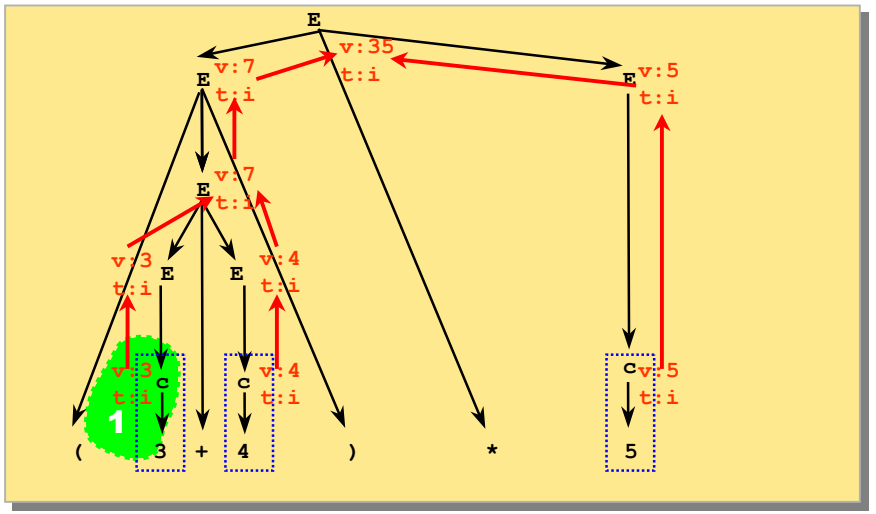
        $b := f(c_1, \ldots, c_n)$

        We shall say that b depends on $c_1, \ldots, c_n$
   - At the end, we shall have all the semantic dependences calculated, between all the attributes.
2. Next, the parse tree is built.
3. The following are possible orderings for the parse trees seen in the previous examples.

Attribute Grammars

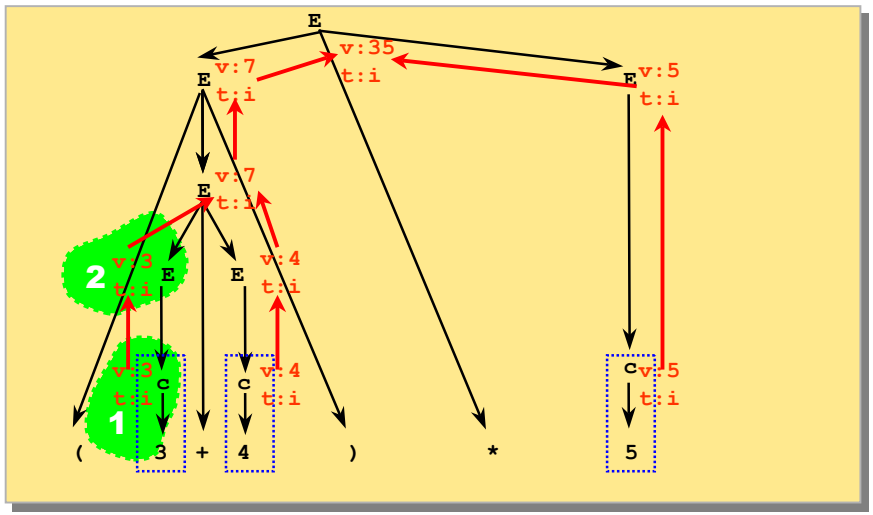
Examples: steps 2 and 3

- We can annotate a path in the parse tree:



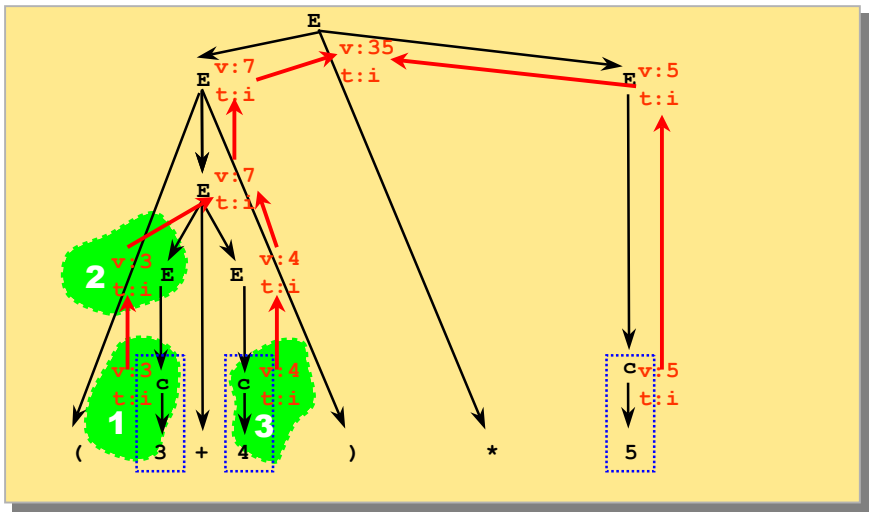
Attribute Grammars

Examples: steps 2 and 3



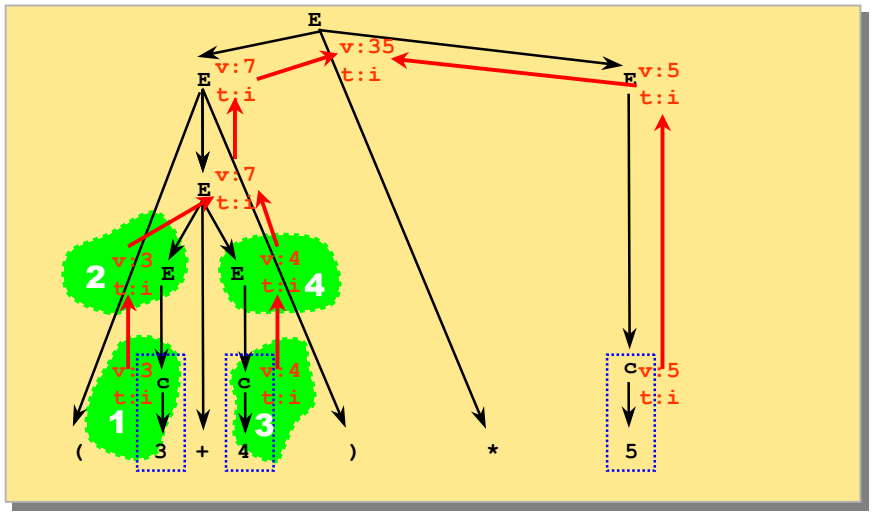
Attribute Grammars

Examples: steps 2 and 3



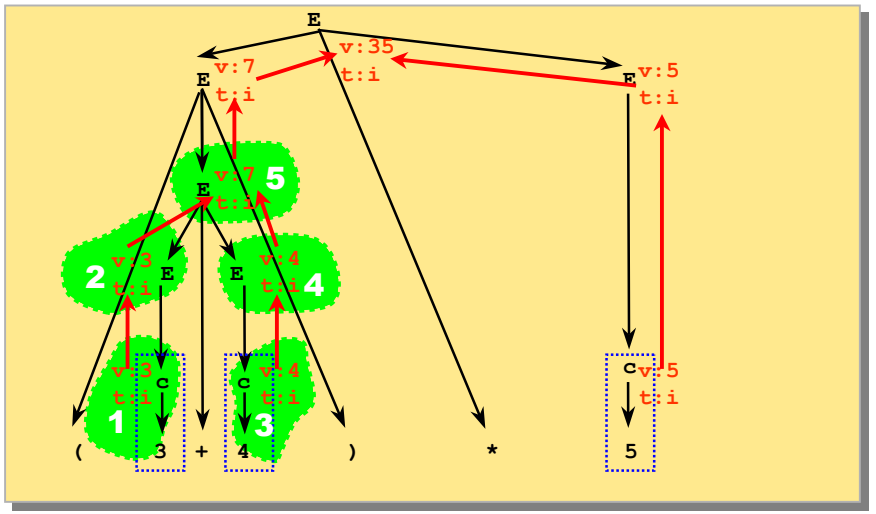
Attribute Grammars

Examples: steps 2 and 3



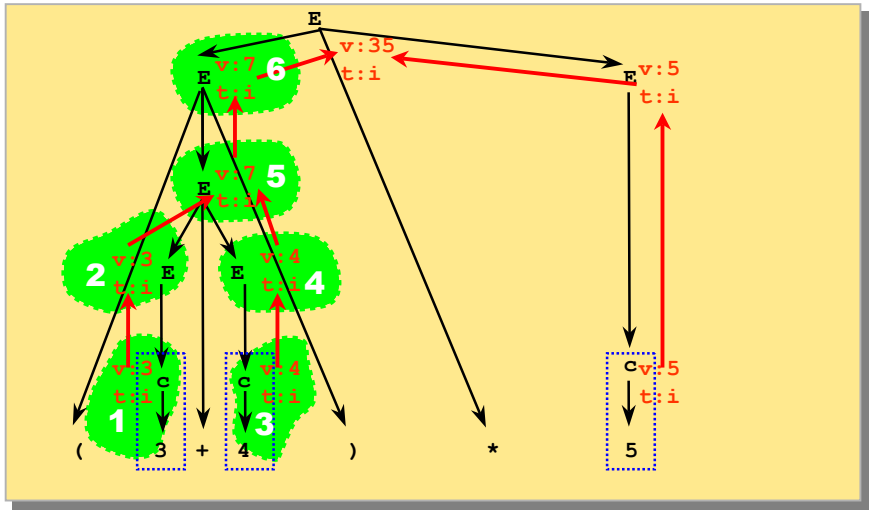
Attribute Grammars

Examples: steps 2 and 3



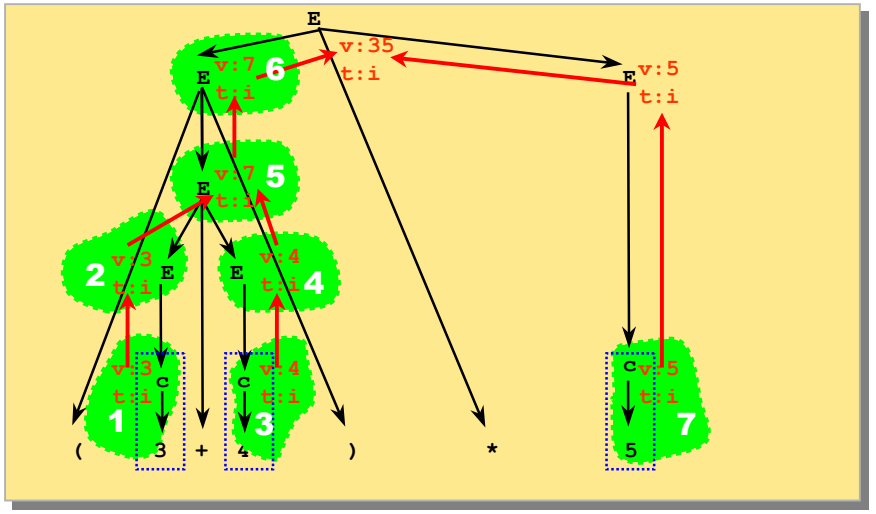
Attribute Grammars

Examples: steps 2 and 3



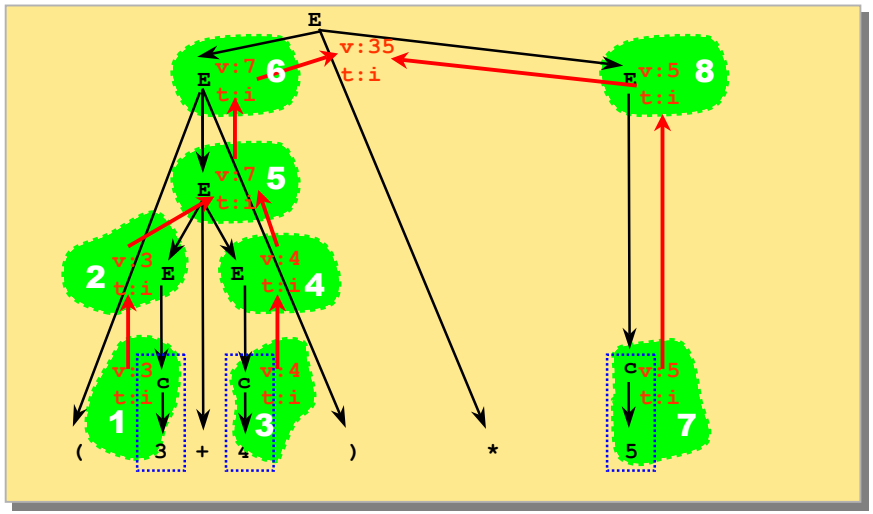
Attribute Grammars

Examples: steps 2 and 3



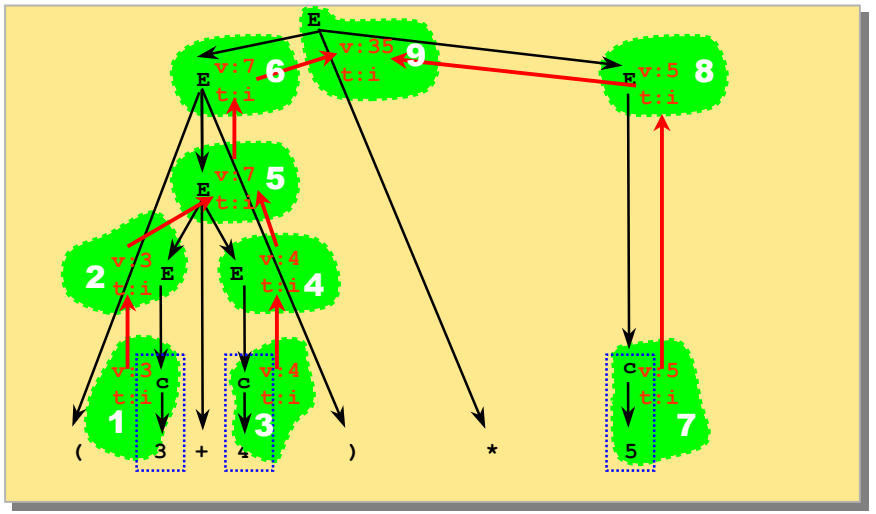
Attribute Grammars

Examples: steps 2 and 3



Attribute Grammars

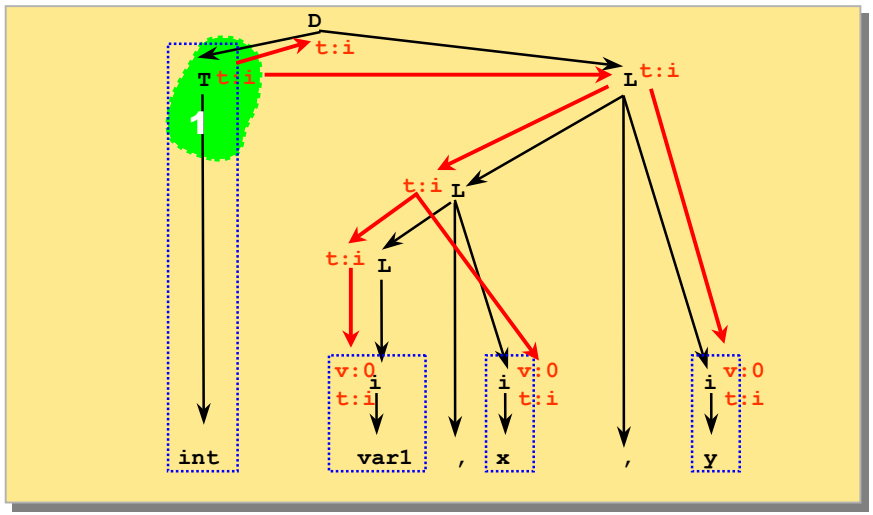
Examples: steps 2 and 3



Attribute Grammars

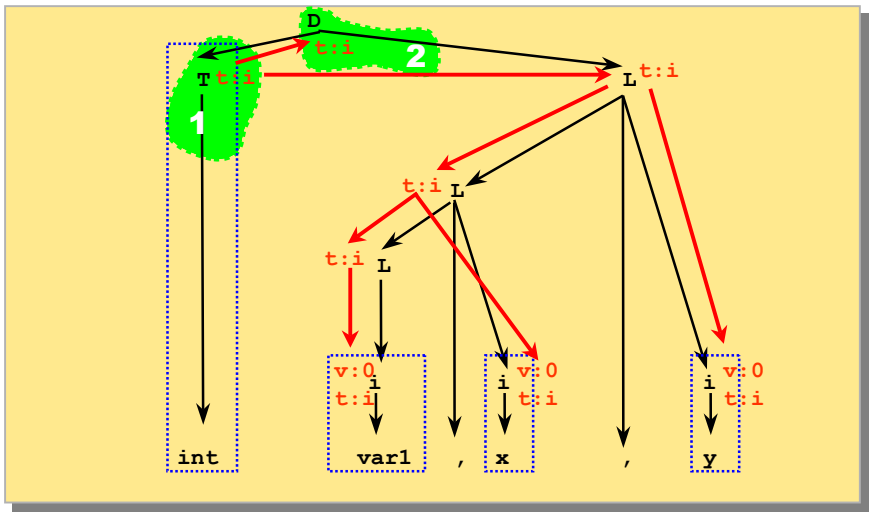
Examples: steps 2 and 3

- The same for the declaration:



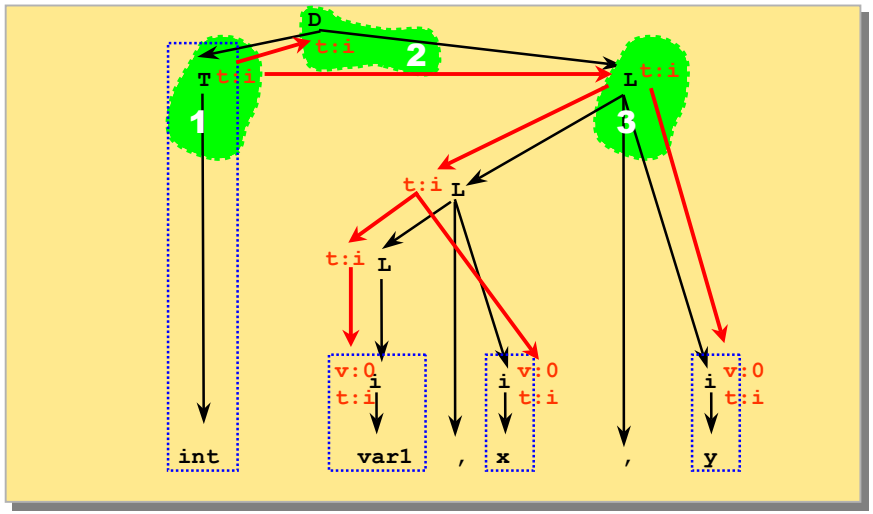
Attribute Grammars

Examples: steps 2 and 3



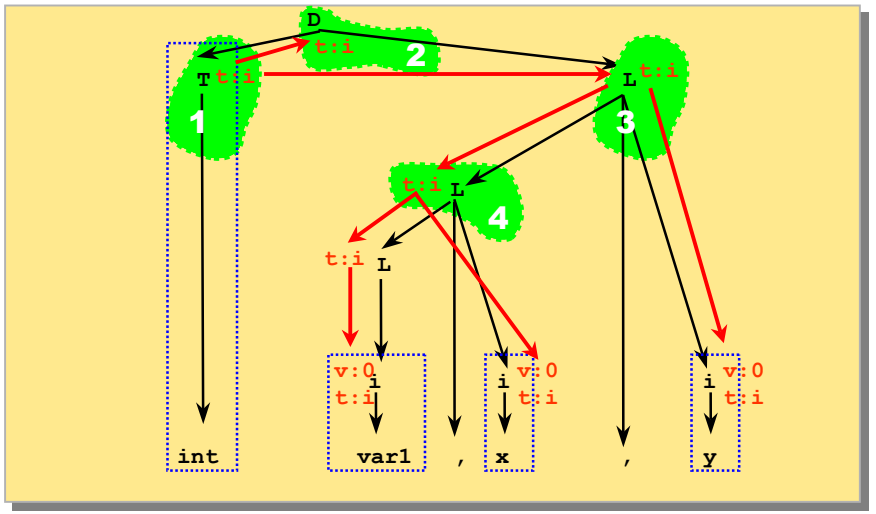
Attribute Grammars

Examples: steps 2 and 3



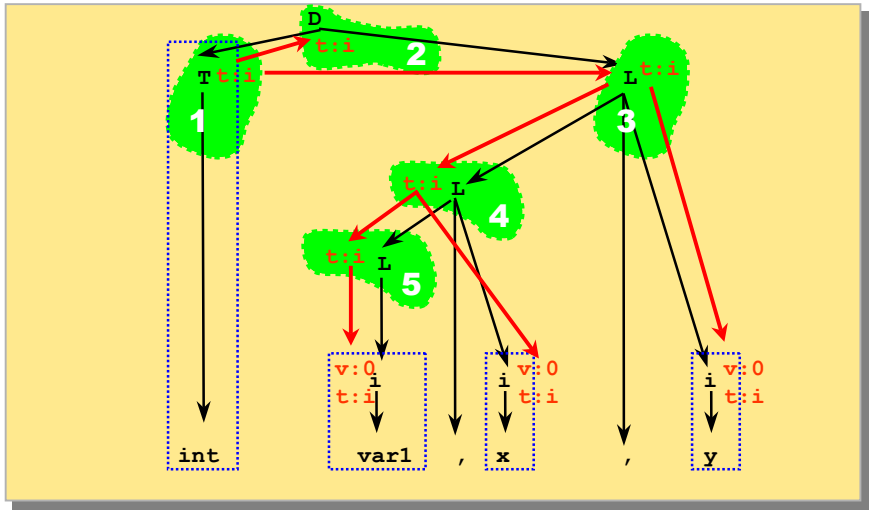
Attribute Grammars

Examples: steps 2 and 3



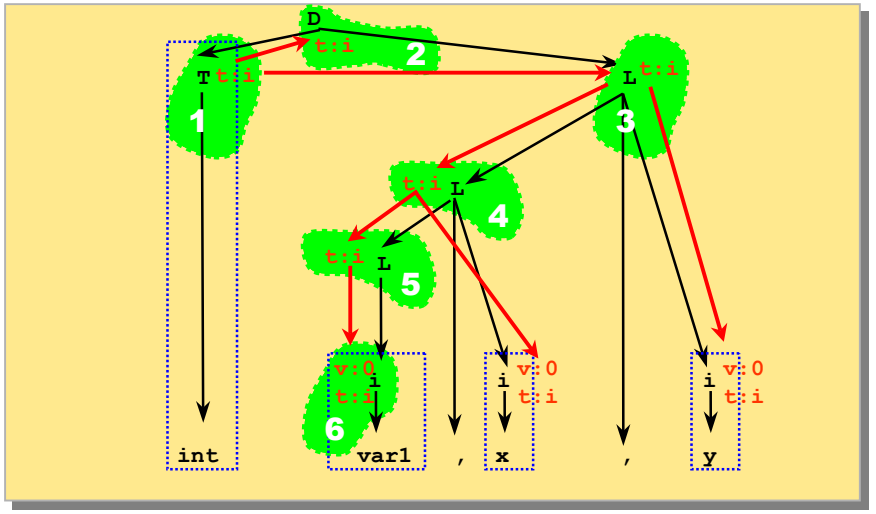
Attribute Grammars

Examples: steps 2 and 3



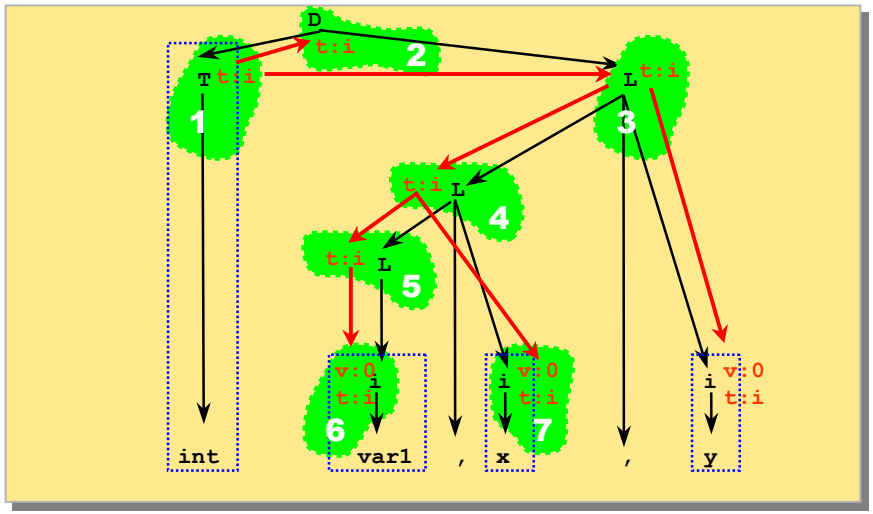
Attribute Grammars

Examples: steps 2 and 3



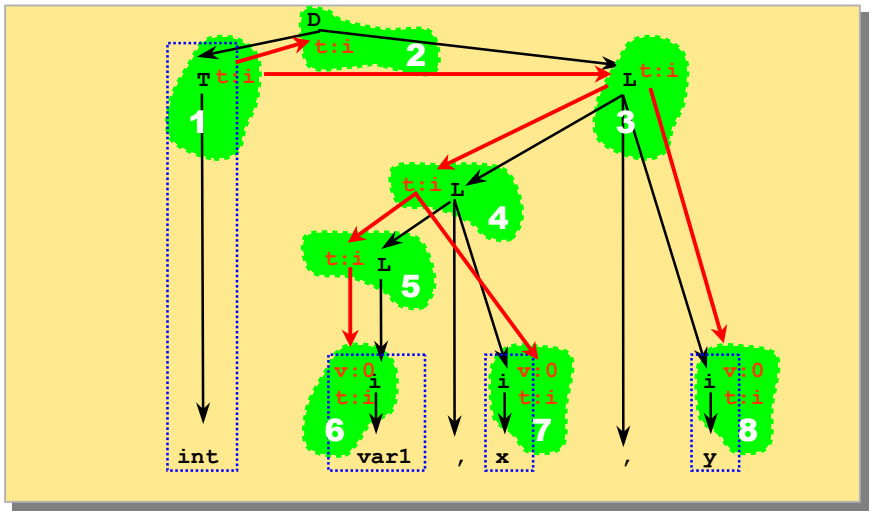
Attribute Grammars

Examples: steps 2 and 3



Attribute Grammars

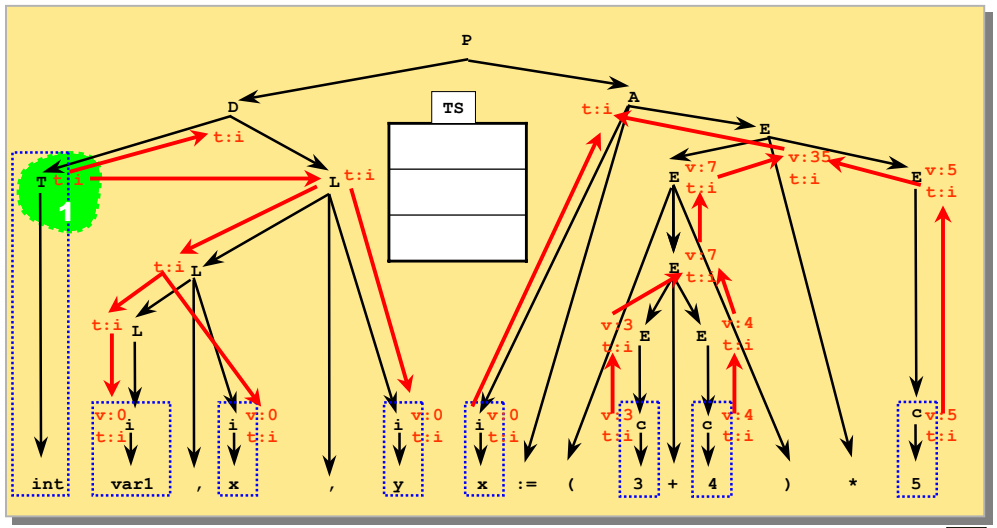
Examples: steps 2 and 3



Attribute Grammars

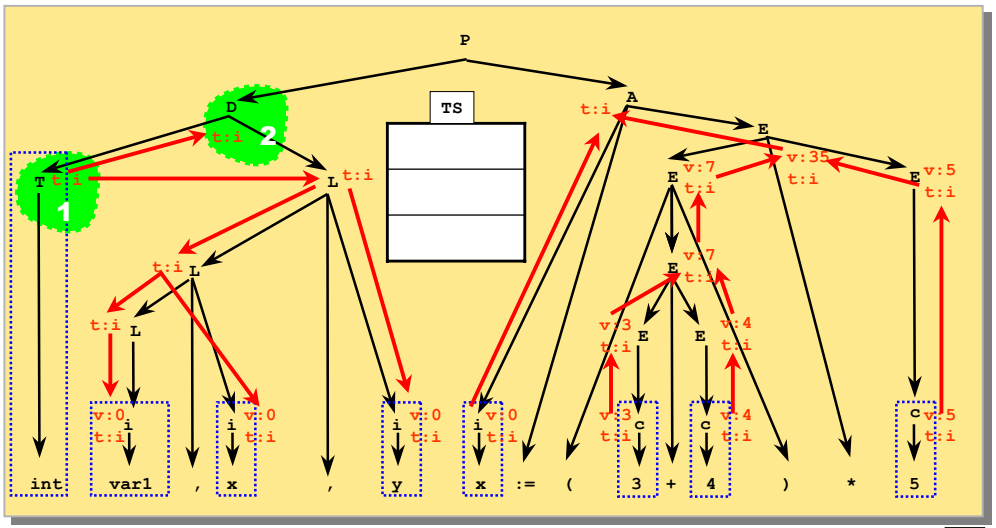
Examples: steps 2 and 3

- And for the final example:



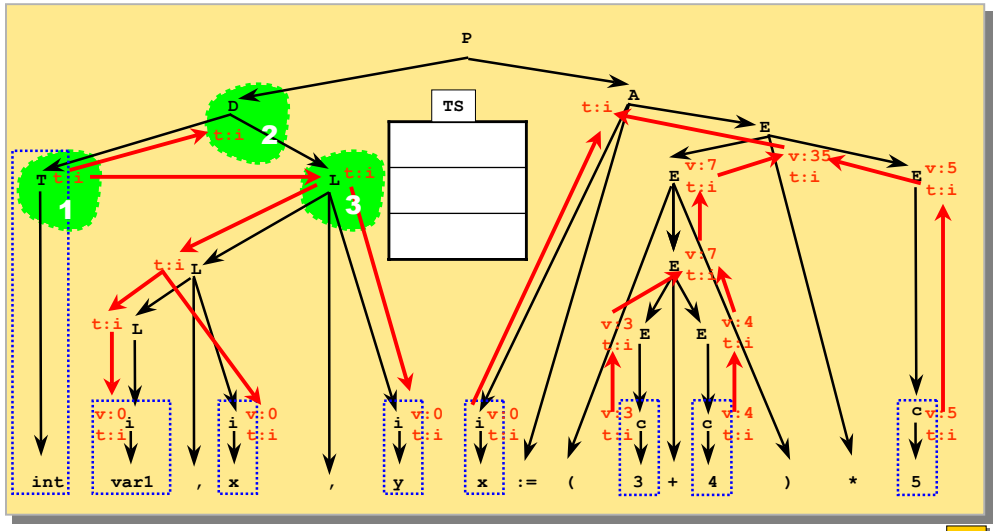
Attribute Grammars

Examples: steps 2 and 3



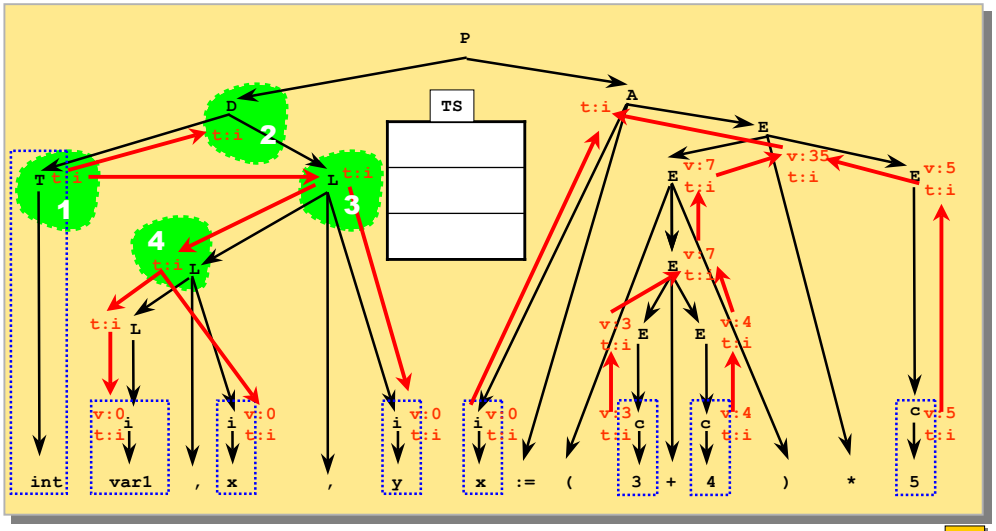
Attribute Grammars

Examples: steps 2 and 3



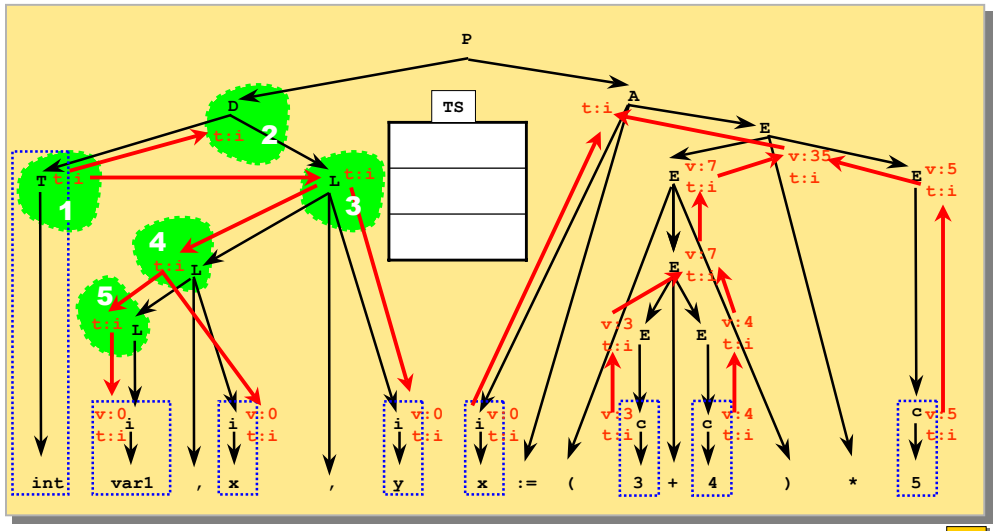
Attribute Grammars

Examples: steps 2 and 3



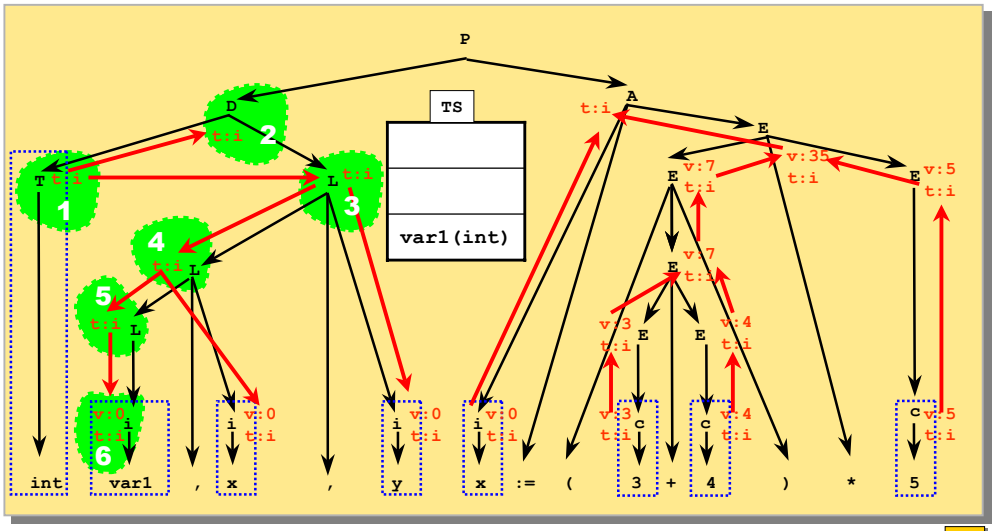
Attribute Grammars

Examples: steps 2 and 3



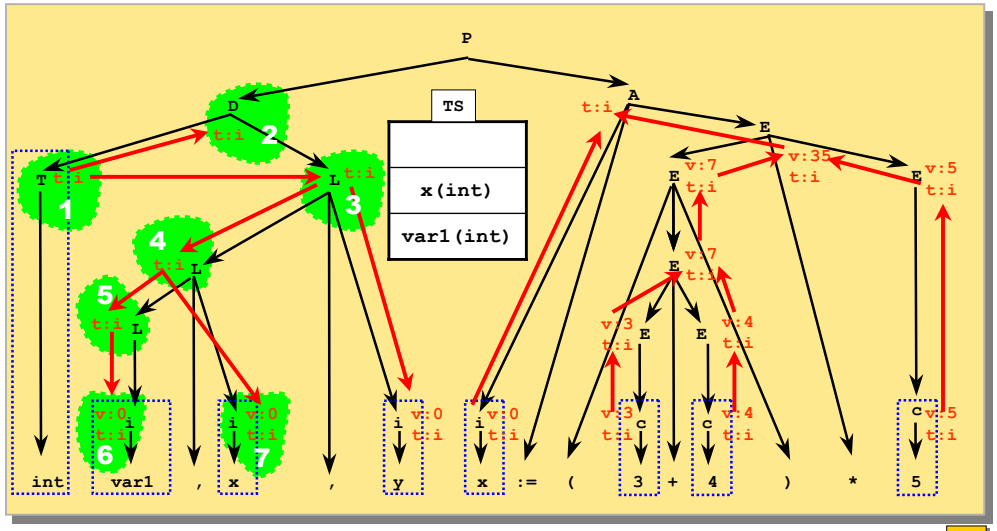
Attribute Grammars

Examples: steps 2 and 3



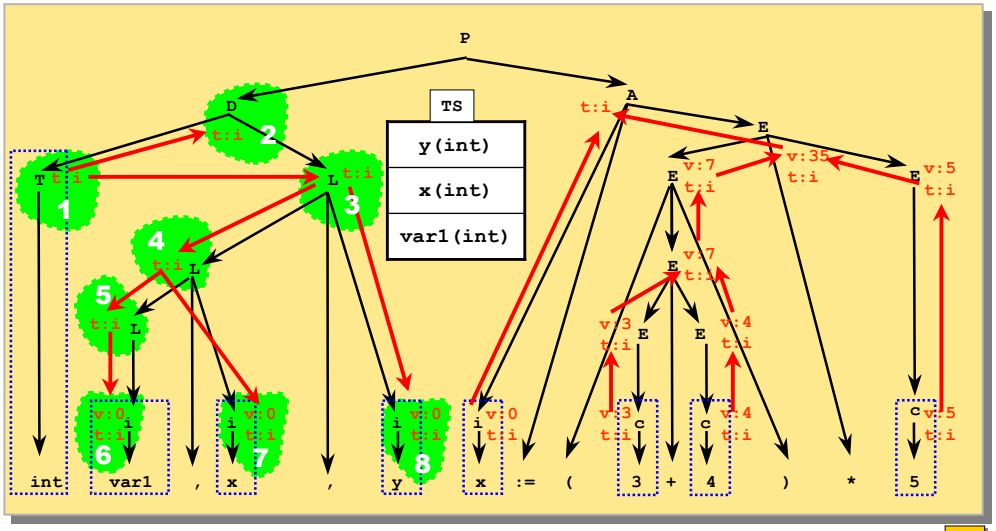
Attribute Grammars

Examples: steps 2 and 3



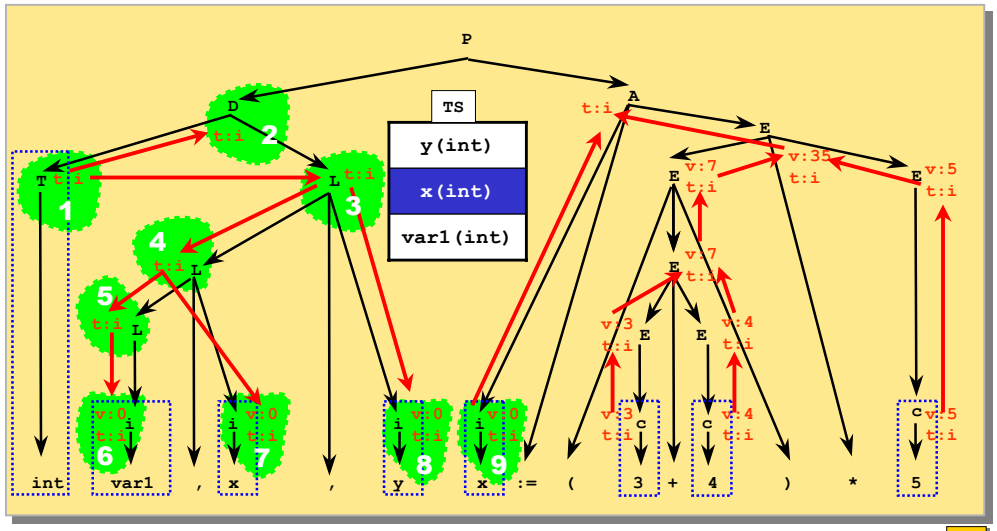
Attribute Grammars

Examples: steps 2 and 3



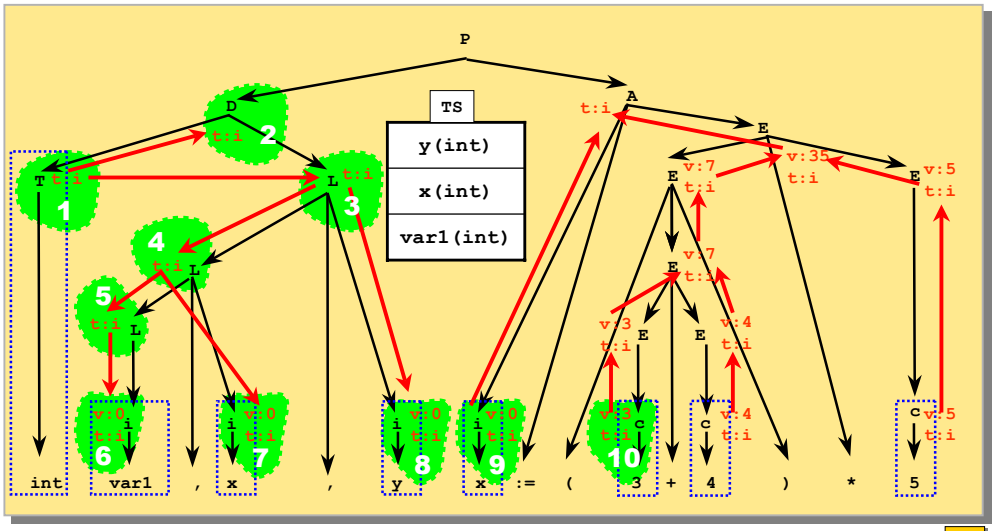
Attribute Grammars

Examples: steps 2 and 3



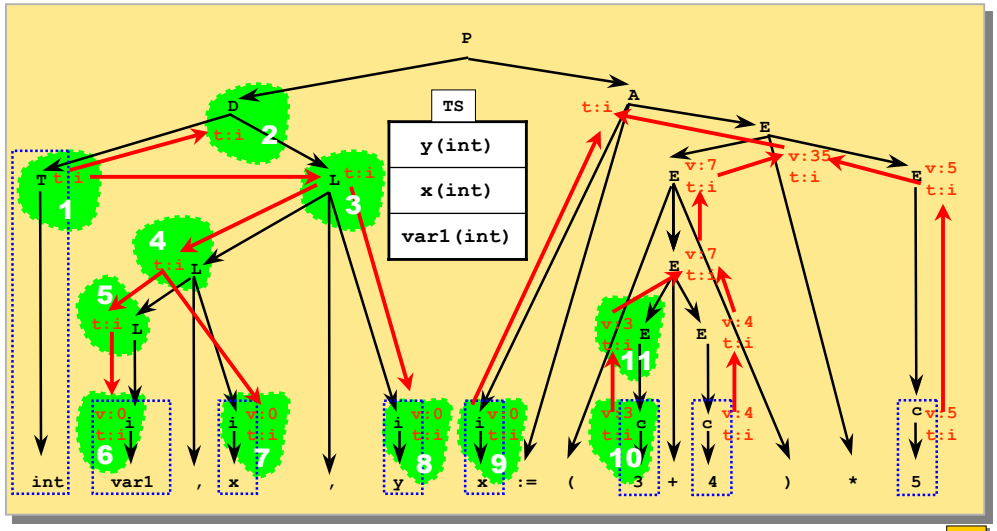
Attribute Grammars

Examples: steps 2 and 3



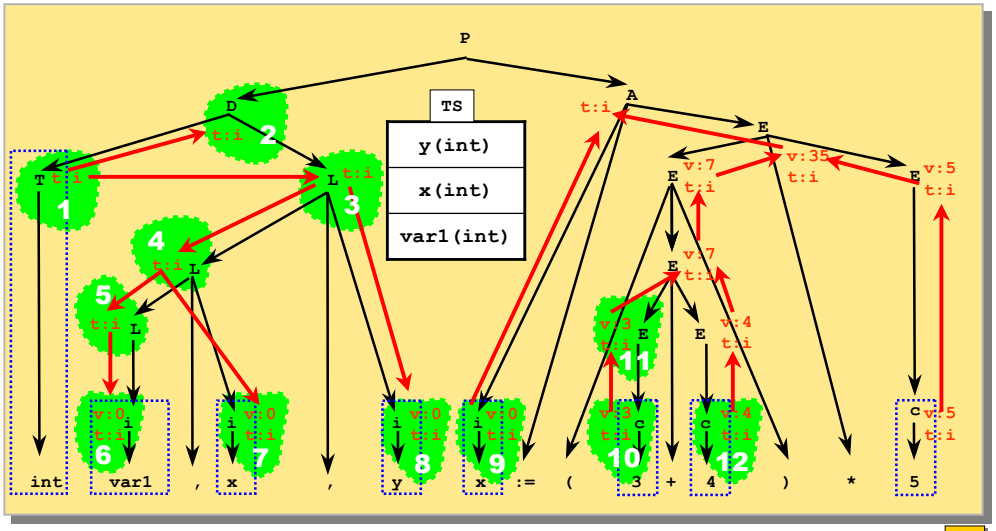
Attribute Grammars

Examples: steps 2 and 3



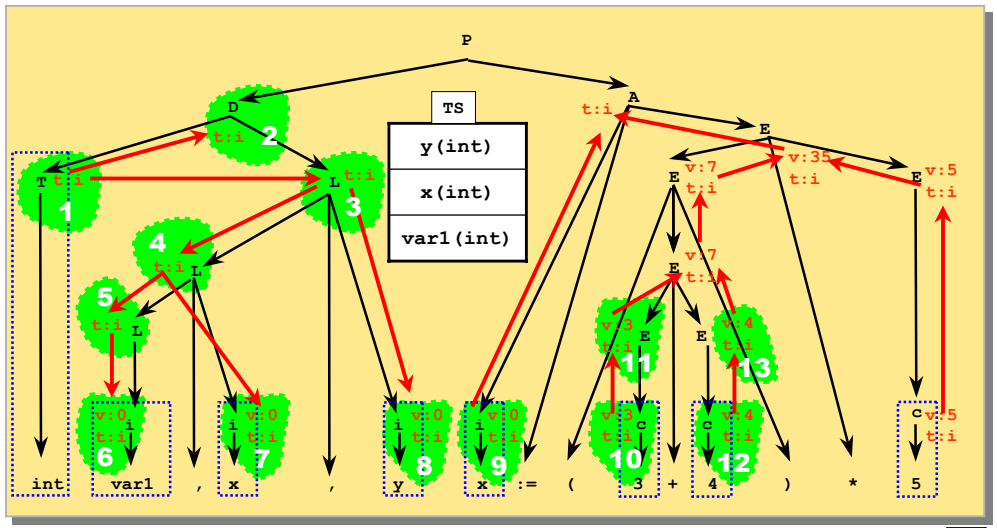
Attribute Grammars

Examples: steps 2 and 3



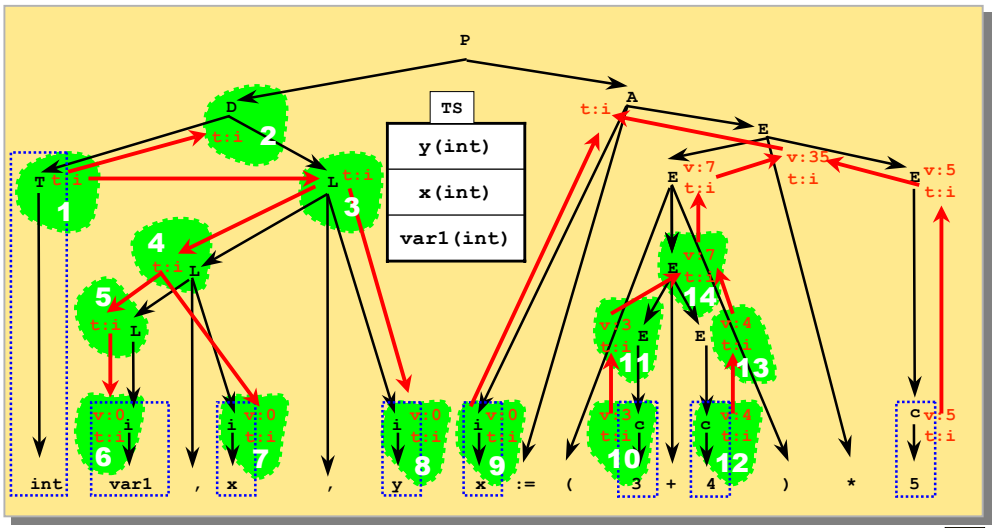
Attribute Grammars

Examples: steps 2 and 3



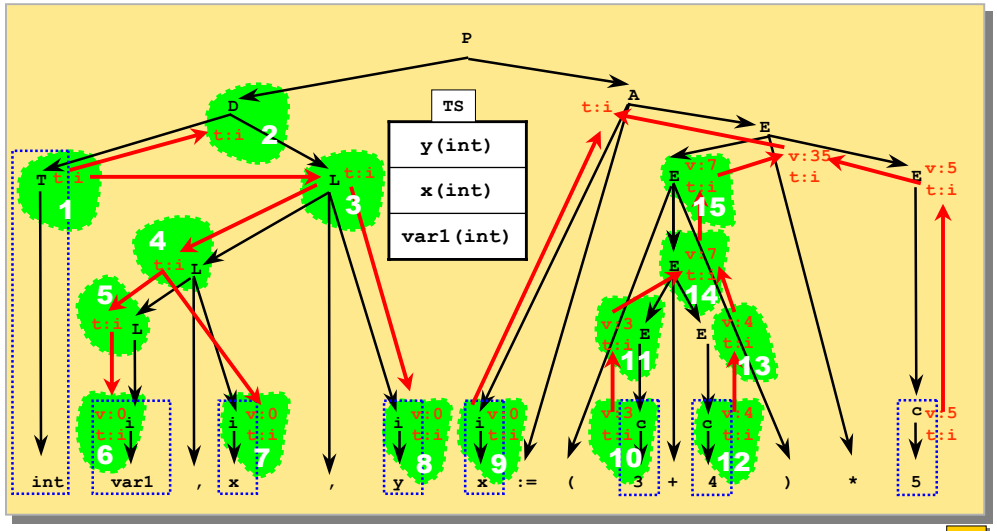
Attribute Grammars

Examples: steps 2 and 3



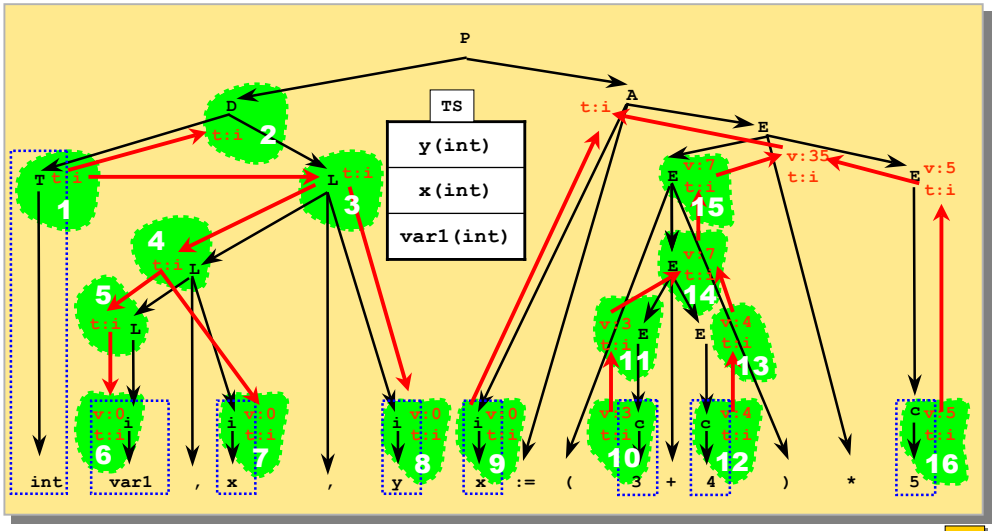
Attribute Grammars

Examples: steps 2 and 3



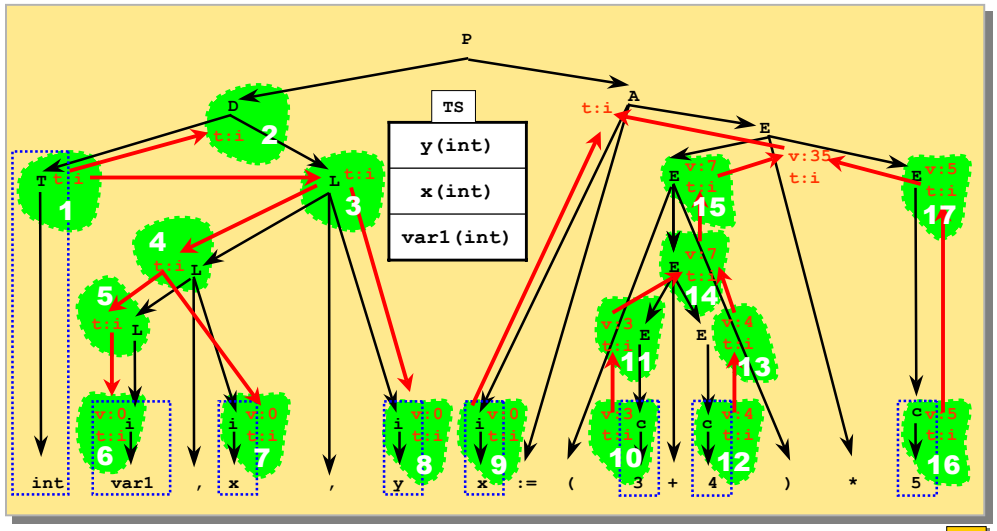
Attribute Grammars

Examples: steps 2 and 3



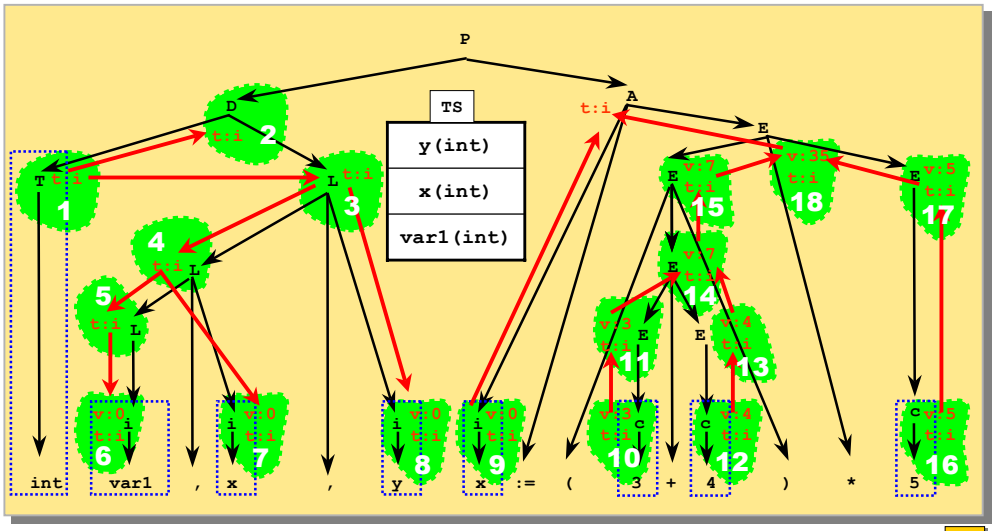
Attribute Grammars

Examples: steps 2 and 3



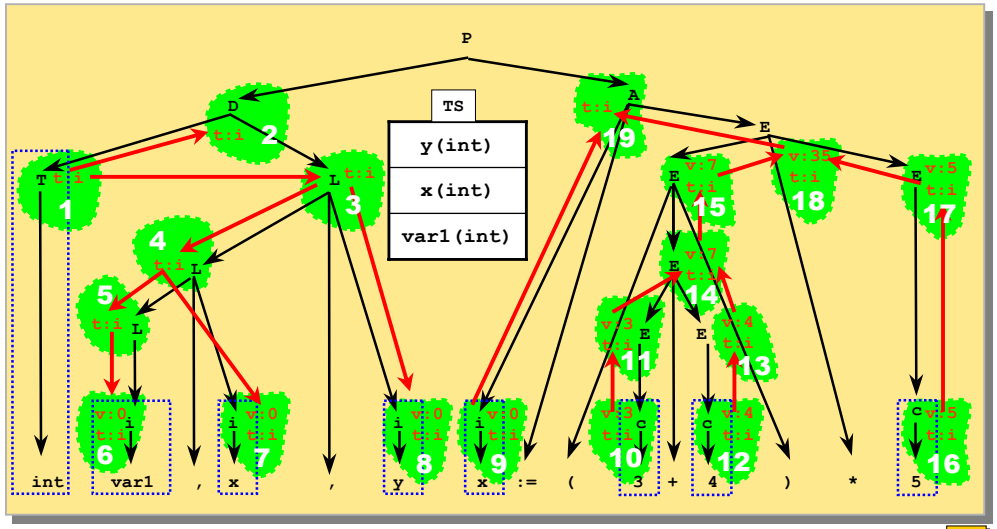
Attribute Grammars

Examples: steps 2 and 3



Attribute Grammars

Examples: steps 2 and 3



Attribute Grammars

Introduction to the more general technique

3. The third step, that we have seen with some examples, can be formalised:
   1. For each semantic action that is a side effect:

      $g(a_1,\ldots,a_m)$

      1. We create a new attribute `a'`
      2. We assume that the semantic action is really:

         $a' := g(a_1,\ldots,a_m)$
   2. We construct a **dependency graph** with the following **algorithm:**

```
graph BuildDependencyGraph
        (tree as, attributes_grammar ga){
   node n; semantic_attribute a; semantic_action acc;
   graph gd = empty_graph;
    "For each node (n) in the parse tree as"
       "For each attribute (a) of symbol n"
              AddNode( new_node(a), gd);
    "For each node (n) in the parse_tree as"
       "For each action (acc=b:=f(c1,...,ck)) of n"
              "For i from 1 to k" AddEdge(new_edge(ci,b),gd);
```


Attribute Grammars

Introduction to the more general technique

3. (cont.) It is very important to mention the possibility of having cycles in the dependency graph. In this case, we shall say that the **syntax-directed definition** is circular.
   - **Circularity** is a problem in the design of the grammar: some attributes depend on others, which themselves depend on the first ones.
   - They provoke a very small performance, so it is advisable to re-design the grammar if it has this problem.
4. To establish a path along the tree that respects the previous ordering, and to apply it.
   - In the examples, the order marked in green would determine the path to follow.
5. The result of the evaluation of the semantic actions in the previous ordering will be the correct semantic analysis of the program.
   - It is possible to check, in the examples, that the interpretation obtained is correct.

Attribute Grammars

Examples: Conclusions

- We can observe, that, depending on the semantic rules, the type of path along the tree can be very different.
- Remember the order of appearance of the nodes in the tree using the different methods that we have studied, both top-down and bottom-up.
- The combination of the syntactic and the semantic analysis will be easier if we can express the semantic information using a set of attributes and rules which is compatible with the ordering in which the syntactic analyser builds the tree.

- The following are two possibilities:
  - **Syntax-directed definitions with synthesised attributes.**
  - **Syntax-directed definitions with left attributes.**

Syntax-Directed Definitions with Synthesised Attributes

Introduction and example

- **These are compatible with bottom-up analysis**:
  - The utility of these kinds of definitions is their compatibility with the order in which the nodes are created, in the parse tree, in bottom-up analysis.
- The **syntax-directed definitions with synthesised attributes:**
  - are those that do not use inherited attributes, i.e. all the attributes are calculated for the parent nodes from their children.
- The example of arithmetic expressions is an example of this kind of grammar.
  - Remember that the evaluation ordering of the attributes is exactly the same as the order in which the rules are reduced in a bottom-up analyser.

Syntax-Directed Definitions with Left Attributes

Introduction

- **Depth-first analysis**:
  - Many techniques of syntactic analysis perform a depth-first analysis.
  - It has the following characteristics:
    - The children of a node are visited from left to right.
    - When all the children of a node have been processed, the parent node is processed.
- Syntax-directed definitions with left attributes:
  - Are compatible with depth-first analysis.
  - We can consider them as the most general class of syntax-directed definitions.
  - They allow us to perform the semantic analyses before creating explicitly the whole parse tree.
  - They include all the definitions directed by LL(1) grammars.
  - They can also be used with bottom-up parsers.

Syntax-Directed Definitions with Left Attributes

Definition

- Syntax-directed definitions with left attributes:
  - They are syntax-directed definitions, which establish the following conditions for the attributes inherited by a non-terminal symbol, X_j, at the right-hand side of a rule:

$$A \rightarrow X_1 X_2 \ldots X_n$$

    - These attributes can only depend on:
      - The attributes of the left siblings.

$$X_1, X_2, \ldots, X_{j-1}$$

      - The attributes of the parent node A

Syntax-Directed Definitions with Left Attributes

Observation: attributes synthesized from the left

- Because syntax-directed definitions with left-attributes only establish conditions on the inherited attributes, we can say that
  - Any syntax-directed definition with synthesised attributes is included in the set of definitions allowed here.

Attributes Propagation and Syntax Analysis

Introduction

- We have seen the most general method, which requires that we have the whole parse tree already built before we attempt the semantic analysis.
- We are also interested in performing the semantic analysis at the same time as the syntactic.
- The following are some possibilities.

Attributes Propagation and Syntax Analysis

With bottom-up analysis

- As already mentioned, the semantic actions in grammars with synthesised attributes are compatible with bottom-up analysis.
- When a rule is reduced, we shall have to execute the semantic actions associated to that rule.

Attributes Propagation and Bottom-Up Syntax Analysis

Example: evaluation of arithmetic expressions

- In this grammar, the only semantic value is **v** which represents the value of the expression:

G_E={{+,*,-,/,(,),c{v},i{v}}, {E{v}, T{v}, F{v}},

{

E→E_d+T {E.v=E_d.v+T.v}

E→E_d-T {E.v=E_d.v-T.v}

E→T {E.v=T.v}

T→T_d*F {T.v=T_d.v*F.v}

T→T_d/F {T.v=T_d.v/F.v}

T→F {T.v=F.v}

F→i {F.v=i.v}

F→c {F.v=c.v}

F→(E) {F.v=E.v}

F→-F_d {F.v=-F_d.v}

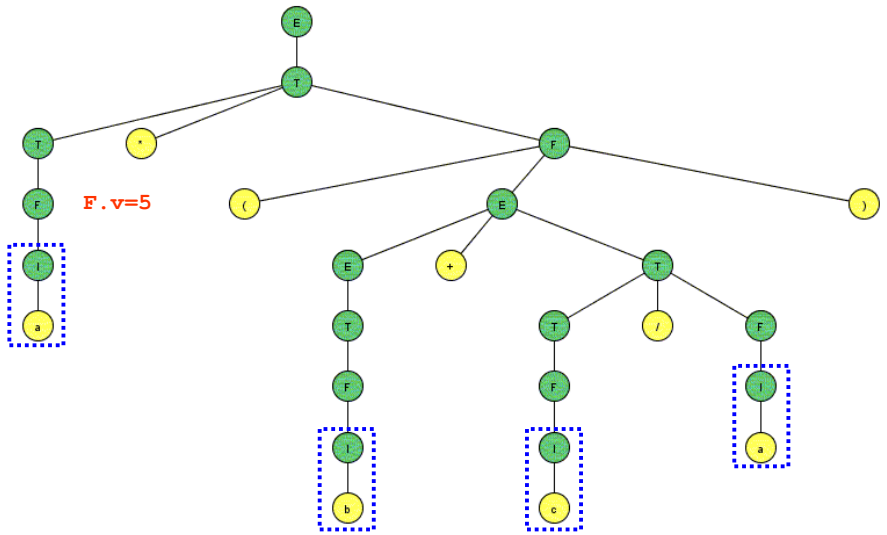
},}

- Imagine that, before this expression **a*(b+c/a)**, **we have the following code:**

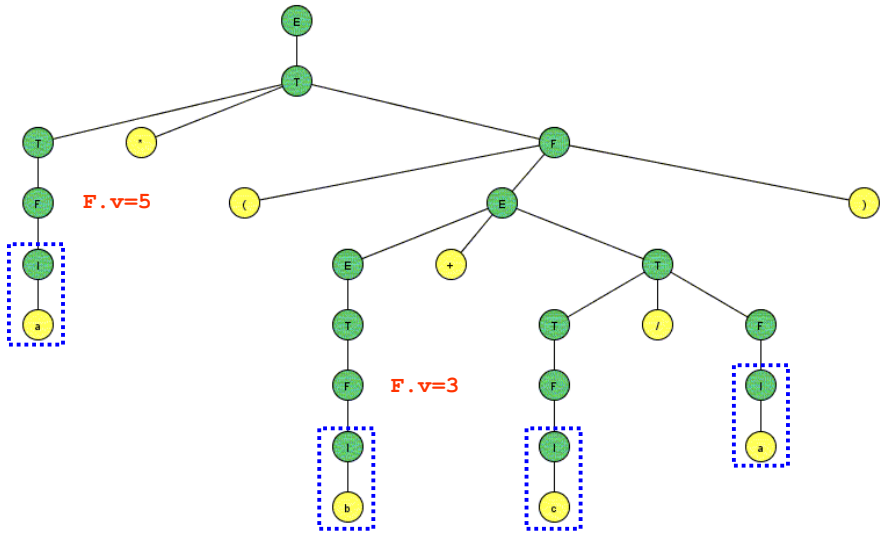
```
a:=5;
b:=3;
c:=10;
```

- We can evaluate the expression and calculate the correct value (25)by propagating the attributes simultaneously with the parsing **a*(b+c/a)**

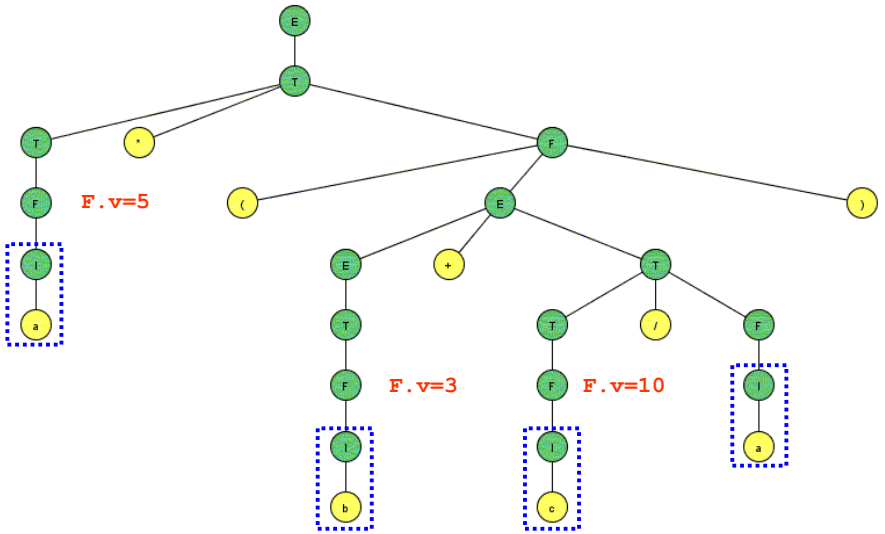
Attributes Propagation and Bottom-Up Syntax Analysis



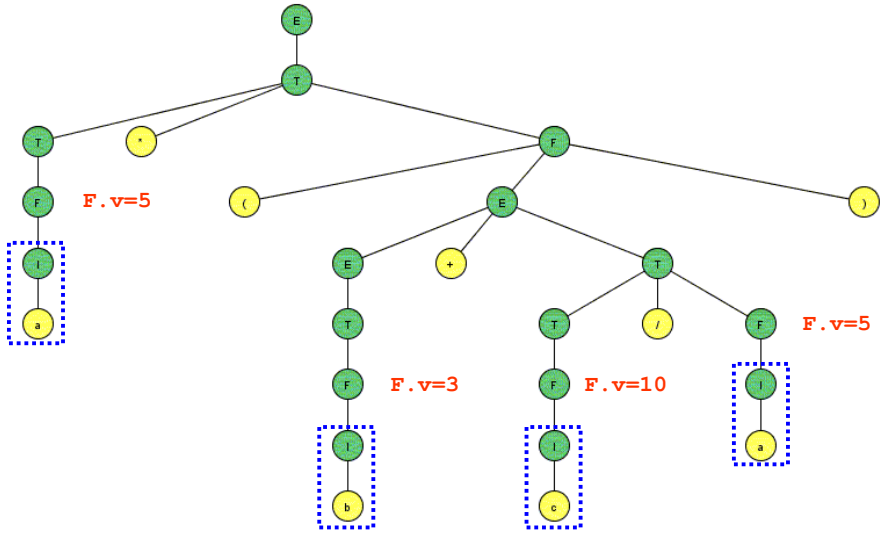
Attributes Propagation and Bottom-Up Syntax Analysis



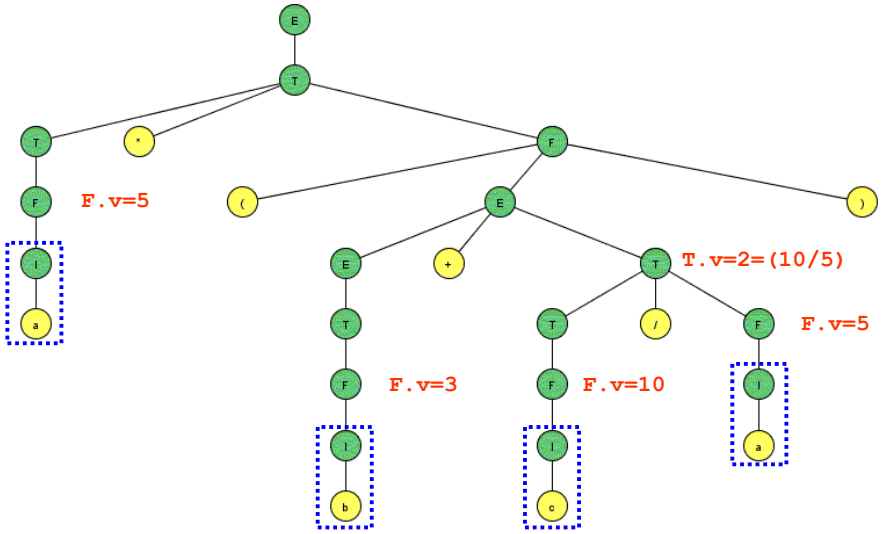
Attributes Propagation and Bottom-Up Syntax Analysis



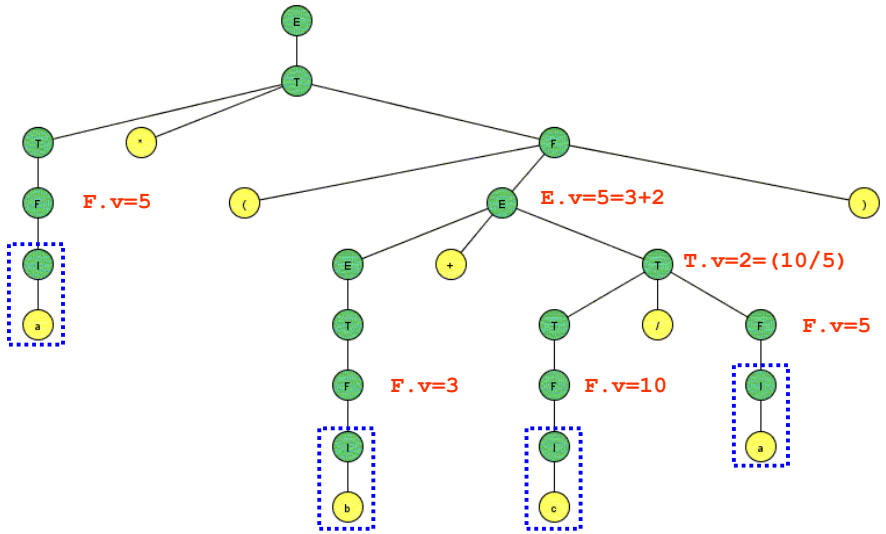
Attributes Propagation and Bottom-Up Syntax Analysis



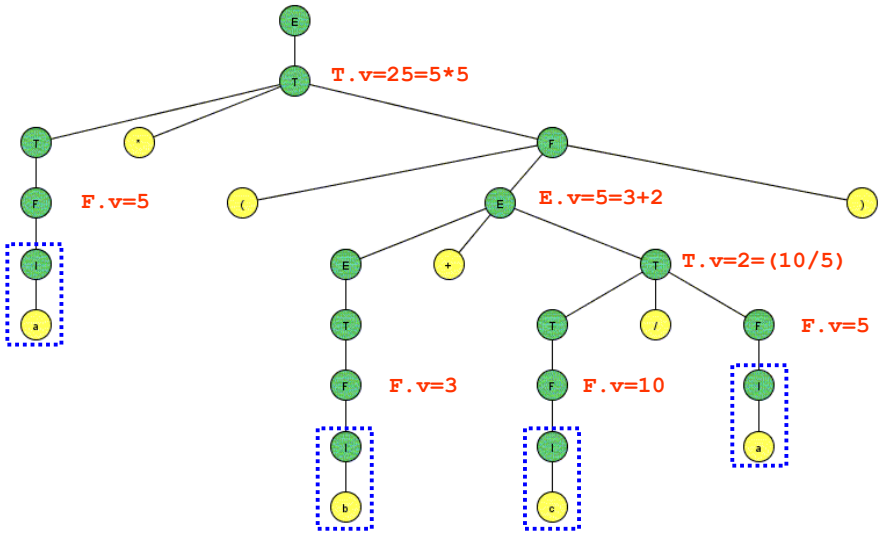
Attributes Propagation and Bottom-Up Syntax Analysis



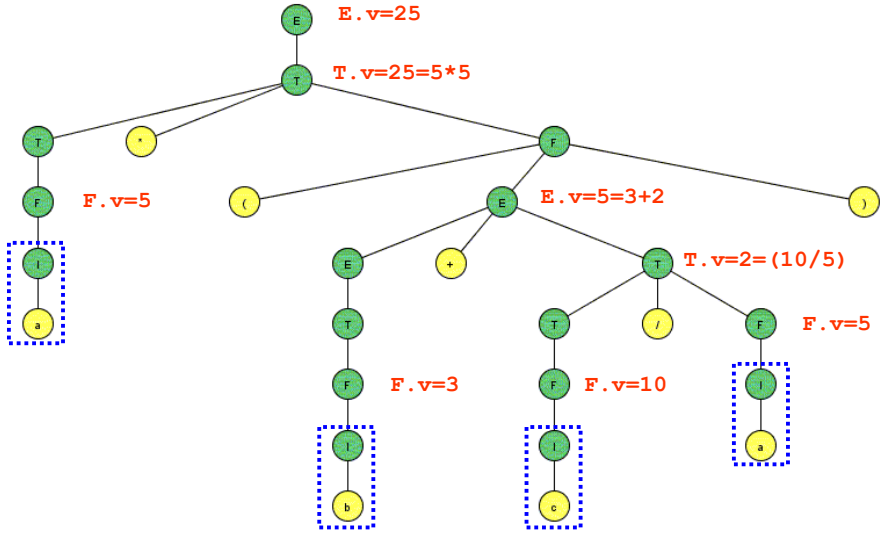
Attributes Propagation and Bottom-Up Syntax Analysis



Attributes Propagation and Bottom-Up Syntax Analysis



Attributes Propagation and Bottom-Up Syntax Analysis



Attributes Propagation and Bottom-Up Syntax Analysis

Extensions: propagating some inherited attributes

- If we use a stack for the semantic values, or we store them in the stack used by the syntactic analyser, then, when we propagate an inherited value, we can have in the stack, below it, the values on which it depends.
- In these cases, by accessing the stack, it will be possible to propagate inherited values (e.g. sibling's attributes).
- Other possibility will be to modify the grammar, so this information will be synthesised rather than inherited.
- The detailed description of these options is outside the scope of this course.

Attributes Propagation and Bottom-Up Syntax Analysis

Extensions: actions between the symbols at the right-hand side

- Grammar to assign expressions to identifiers, to calculate the value and check the types.

```
G_A={{:=,+,c{v,t},i{v,t}}, {A{v,t}, E{v,t}, F{v}},
   {
      A→i{"Check that i has been declared"}
         := <Exp>
         {"Type compatibility: (i.t) and (E.t)";
          i.v = E.v }
      E→E'+T
         {"Type compatibility: (E'.t) and (T.t)";
          E.t=type_sum(E'.t,T.t);(*)E.v=E'.v+T.v}
      E→T {E.t=T.t; E.v=T.v}
      T→i {"Check that i has been declared";
           T.t=i.t; T.v=i.v}
      T→c {T.t=c.t; T.v=c.v}
   },}
```


Attributes Propagation and Bottom-Up Syntax Analysis

Extensions: actions between the symbols at the right-hand side

- Grammar for conditional statements:

```
G_A={{if, then, else}, {E, Instr},
   {
      Instr→if E {Action2} then Instr {Action1}
      Instr→if E {Action2} then Instr else {Action3} Instr
        {Action1}
   },}
```


Attributes Propagation and Bottom-Up Syntax Analysis

Extensions: actions between the symbols at the right-hand side

- In bottom-up analysers, the semantic actions are only executed when the rule is reduced.
- Therefore, the actions should all be at the end of the right-hand sides of the rules.
- A possibility is to add markers for semantic actions, which are symbols which just generate the empty word, (*M*→λ) but that execute an action before disappearing.
- This technique is applied in the first example above, in the first rule. The actions that are not problematic are abbreviated as {})

```
G_A={{:=,+,c{v,t},i{v,t}}, {A{v,t}, E{v,t}, F{v}, M},
   {
      A→i M := <Exp> {}
      M→λ {"Check that the identifier was declared"}
      E→E'+T {}
      E→T {}
      T→i {}
      T→c {}
   },}
```


Attributes Propagation and Bottom-Up Syntax Analysis

Extensions: actions between the symbols at the right-hand side

- The previous technique is used by some automatic generators of parsers.
- The designer of the grammar can use the same procedure.
- In the second example:

```
G_A={{if, then, else}, {E, E1, Else, Instr},
   {
      Instr→if E1 then Instr {Action1}
      Instr→if E1 then Instr Else Instr {Action1}
      E1→E {Action2}
      Else→else {Action3}
   },}
```

Attributes Propagation and Bottom-Up Syntax Analysis

Extensions: actions between the symbols at the right-hand side

- It is easy to see that the new grammars are equivalent to the previous ones.
- However, take care with this technique, because too many rules to the empty word may start provoking conflicts in the analysis matrix.

Attributes Propagation and Top-Down Syntax Analysis

With top-down analysers

- We have already mentioned that **the semantic actions in left-attribute grammars** are compatible with LL(1) grammars.
- In the case of top-down analyses, it is not relevant the position of the semantic actions in the right-hand sides of the rules.
- The semantic actions can be considered as **additional symbols.**
- If it is possible to transform a grammar into LL(1) form, together with the semantic actions, then the generation of a syntax/semantic analyser is very easy.
- This lesson will show later a variation of the recursive top-down LL(1) analyser with some semantic actions added.