

Simple Semantic Analysis for Structured Documents Using XML

Semantic determines the meaning of a sentence in natural language. Semantic analysis of a programming language is determined by its operations that are implemented within the language. Typically, these operations are unary or binary operations. Examples of unary operations are negation (\neg) in logical expressions or the unary minus ($-$) for arithmetic expressions. Well-known binary expressions are \wedge, \vee in logical expressions or $+, -, *, \div$ in arithmetic expressions. For example, identifiers or variables must be declared to store the results obtained by these operations applied to certain values. Hence, every identifier or variable must have a type regarding to the possibly applied operation. In the example of boolean and arithmetic expressions, these types are `boolean`, `integer` or `real` (and sometimes `float`). A type mismatch occurs, e.g. if the declaration `boolean a,b;` is used to calculate `a + b`, where the current values might be `a = 2` and `b = 3`. The type check is performed for every statement in the program. Therefore, the current declarations taken from the symbol table are proven against the applied operation.

A more extensive example In the following some productions are reviewed that define arithmetic expressions. The productions, that are given by a context-free grammar, are known. Beside the productions, the definition of semantic rules are needed. The *output(...)*-functionality is applied to write the current value (*val*) to the output. The symbol *lexval* stands for the lexical value, that is the real value (of the token `num`). Thus, the semantic rules give the bindings of values to the tokens.

	Productions	Semantic rules
S	\rightarrow E \$	output(E.val)
E	\rightarrow E + T	E.val = E.val + T.val
E	\rightarrow T * F	E.val = T.val * E.val
T	\rightarrow F	T.val = F.val
F	\rightarrow num	F.val = num.lexval

The input of the semantic analysis is, beside the semantic rules, is the parse that is produced by performing syntax analysis. A corresponding parse tree for the **input** `3*5+4` is shown in Figure 1 on the left-hand side. On the right-hand side, the so-called attributed syntax tree is given. The attributed syntax tree should be the result of the semantic analysis. The evaluation of the current labelled and type-checked nodes is also presented in this tree. The

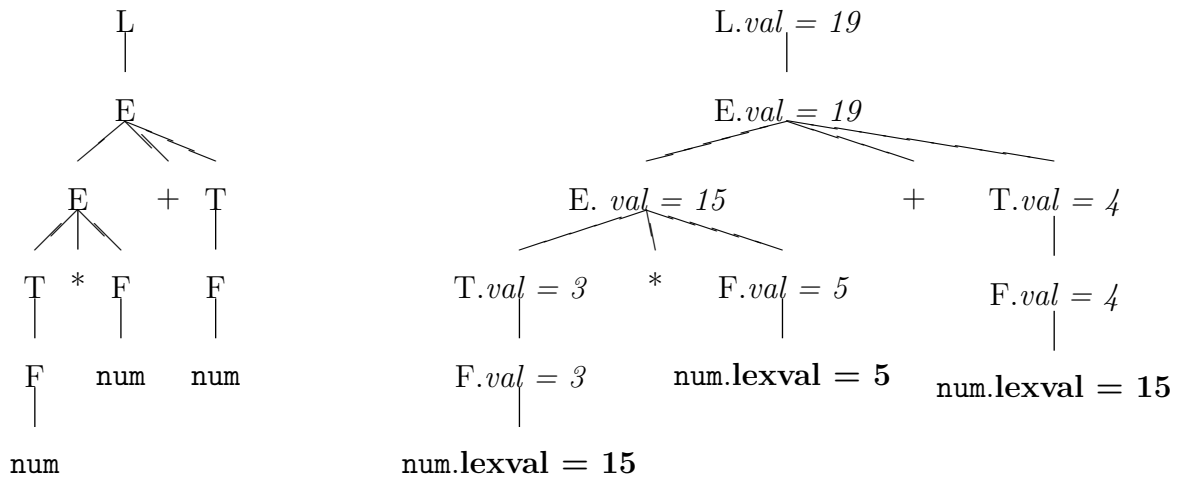


Figure 1: Syntax tree and attributed syntax tree

question is, how to compute this attributed syntax tree for implementing type checking.

Program modules performing the semantic analysis must support the so-called symbol table and/or environments, see the lecture notes on semantic analysis. The current environment must match with the program code. This means, the environments σ must be determined at first. As a consequence the tree looks as follows for the given example, Figure 2.

To evaluate, the term in correspondence to the environments, the dependencies between subterms must be known. To obtain these dependencies, a so-called dependency graph is calculated by determining a topological order for the given syntax tree respect of the operations. There might be more than one possible topological order that leads to a valid evaluation. A topological order for the given example is shown in Figure 3. The nodes are enumerated in correspondence to the numbering of the environments σ .

1. Study the given example to compute the attributed syntax for the input $3 * 5 + 4$
2. Find another dependency graph that also gives a correct evaluation of the given term.
3. Consider the input $10 + 7 * 3 * 5$
 - (a) Give the syntax tree.
 - (b) Compute the environments.
 - (c) Determine a dependency graph that delivers a correct evaluation of the given term.
 - (d) Draw the attributed syntax tree.

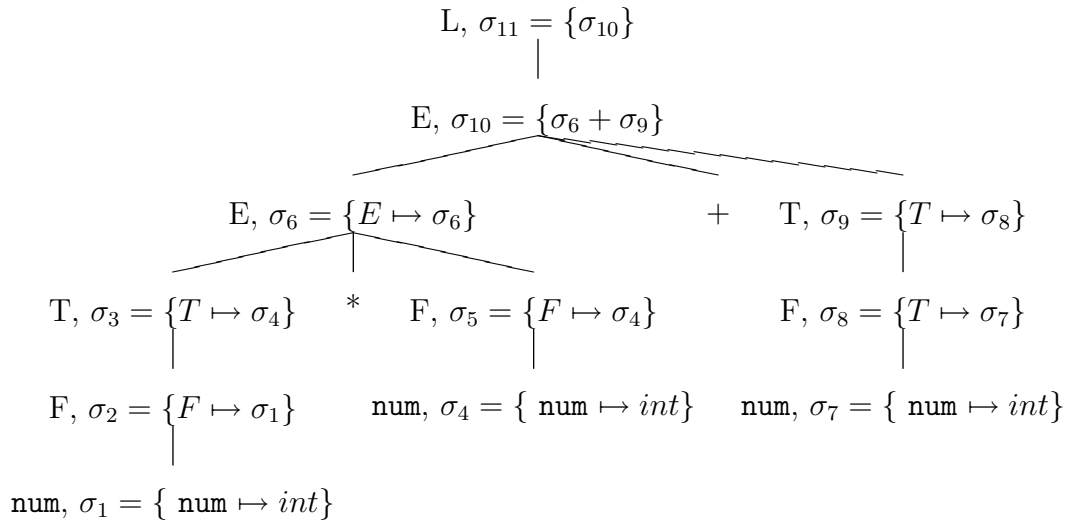


Figure 2: Syntax tree, including environments

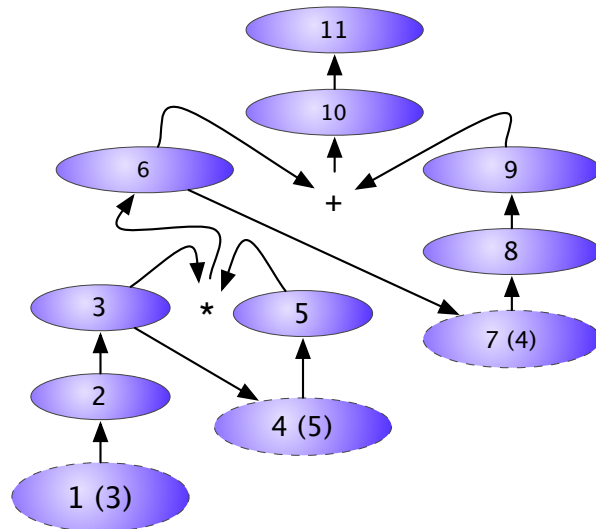


Figure 3: A dependency graph for $3 * 5 + 4$

On the implementation of the semantic analysis

The environments can be implemented, e.g. in an imperative style, see in the lecture notes. The implementation of the semantic analysis phase requests two parts within your compiler:

1. The implementation of the current environments requests the implementation of a suitable hash functionality. (The hash functionality can be reused from the lecture notes for your implementation.)
2. Methods, like `visitor` and `visit`, are applied to the current program statement that check the given data types.

Our goal is to implement a simple type checking system (that might be too simple for some cases in practical use, but it is better than nothing). For the implementation of the phase of semantic analysis we must know the types of elements within an XML document. As the consequence the open questions are: What are operations in structured documents? What are data types in XML defining structured documents then? To answer these questions perform the following tasks.

A structured document is built by a tree and its subtrees that represent certain parts of the given document. Thus, the type checking process requests two types of operations.

1. The first operation is n-ary operation ($n \geq 2$) that concatenates of already known subtrees.
2. The second operation is labeling leaves with their data types.

The data types of the leaves are given by their stored data. In XML we may distinguish between `#PCDATA`, `images` (often stored as binary large objects (blob's)), and `links` that refer to other sources on the Web. Thus, we are lucky! The operations for typing leaves are only unary operations. Combining already labeled leaves with a common root node already become a tree operation. That is too hard to prove tree operations, if we implement a type checker for the first time! *The semantic check might be restricted to check the data types of the leaves.* Therefore, the implementation is split

- to a part that serves a certain data structure of environments, and

- to the type checking process itself within the XML code represented by its parse tree, respectively its attributed syntax tree.

- Environments**
1. (Re-) Implement the hash implementation with external chaining for the application of XML denoted documents.
 2. Which symbols are suitable to be stored? (Re-) Implement the `Symbol` class for the typed leaf-elements within an XML document. *Note the restriction given above!*
 3. Implement the corresponding environments that handle the allowed types, especially applied to leaves, in an imperative style.

- Type checking**
1. Implement a suitable `Visitor` method for leaves.
 2. Finally, implement the type checker for every the statement while visiting the leaves by a suitable `visit` method. Leaves with different types must be checked in a different manner.