

[Next](#)
[Up](#)
[Previous](#)
[Contents](#)

Next: [4.1 Building Abstract Syntax](#)
Up: [CSE 5317/4305: Design and](#)
Previous: [3.4 Case Study: The Contents](#)

4 Abstract Syntax

A grammar for a language specifies a recognizer for this language: if the input satisfies the grammar rules, it is accepted, otherwise it is rejected. But we usually want to perform semantic actions against some pieces of input recognized by the grammar. For example, if we are building a compiler, we want to generate machine code. The semantic actions are specified as pieces of code attached to production rules. For example, the CUP productions:

```

E ::= E:e1 + E:e2      { : RESULT = e1 + e2; : }
   | E:e1 - E:e2      { : RESULT = e1 - e2; : }
   | num:n             { : RESULT = n; : }
;

```

contain pieces of Java code (enclosed by `{ : : }`) to be executed at a particular point of time. Here, each expression `E` is associated with an integer. The goal is to calculate the value of `E` stored in the variable `RESULT`. For example, the first rule indicates that we are reading one expression `E` and call its result `e1`, then we read a `+`, then we read another `E` and call its result `e2`, and then we are executing the Java code `{ : RESULT = e1 + e2; : }`. The code can appear at any point of the rhs of a rule (even at the beginning of the rhs of the rule) and we may have more than one (or maybe zero) pieces of code in a rule. The code is executed only when all the symbols at the left of the code in the rhs of the rule have been processed.

It is very uncommon to build a full compiler by adding the appropriate actions to productions. It is highly preferable to build the compiler in stages. So a parser usually builds an Abstract Syntax Tree (AST) and then, at a later stage of the compiler, these ASTs are compiled into the target code. There is a fine distinction between parse trees and ASTs. A parse tree has a leaf for each terminal and an internal node for each nonterminal. For each production rule used, we create a node whose name is the lhs of the rule and whose children are the symbols of the rhs of the rule. An AST on the other hand is a compact data structure to represent the same syntax regardless of the form of the production rules used in building this AST.

Subsections

- [4.1 Building Abstract Syntax Trees in Java](#)
- [4.2 Building Abstract Syntax Trees in C](#)
- [4.3 Gen: A Java Package for Constructing and Manipulating Abstract Syntax Trees](#)

[Next](#)
[Up](#)
[Previous](#)
[Contents](#)

Next: [4.1 Building Abstract Syntax](#) **Up:** [CSE 5317/4305: Design and](#) **Previous:** [3.4 Case Study: The Contents](#)

fegaras 2012-01-10