
University of
Southampton

University of Southampton

Syntax and Grammar

COMP2212 Programming Language Concepts

Dr Julian Rathke

University of Southampton

Syntax vs Semantics

Syntax

- This refers to the **structure** of statements in a program.
- It typically follows a grammar based upon certain lexical symbols (e.g. keywords in a language)

Semantics logic

- This refers to the **meaning** of programs and how programs execute.

The role of an interpreter or compiler of a language is to transform syntax into semantics.

University of Southampton

Language Syntax as Grammars

- You have learned about regular languages and context free languages in COMP2210
- Most programming languages are context free languages expressible using a Context Free Grammar (CFG)
- You have learned about Backus-Naur Form (BNF) for describing Context-Free Grammars
- BNF is the de facto standard for defining the grammar of a programming language.

University of Southampton

Non-Terminals versus Terminals

BNF acts as a **metalanguage** for defining languages

- It is a convenient meta-syntax for defining the grammar of a language.

Non-terminals represent different states of determining whether a string is accepted by the grammar

- In programming language terms these refers to the different kinds of expressions one may have in the language
- e.g. class level declarations, method declarations, statements, expressions

Terminals represent the actual symbols that appear in the strings accepted by the grammar

- These are sometimes called tokens or lexemes and refer to the reserved words, variable names and literals of our programs.

e.g. 7 and 10 are -- integer constants --> literals

University of Southampton

Can be in exam!!!

Example BNF grammar

The following is an example of a BNF grammar for a simple language of assignments

Non-terminals are written as <follows> and the terminals are written in **bold**

is a token/lexeme

begin, end, ;, skip, X, Y, Z, +, -, =

Anything that is not bolded in / part of the program

e.g. <program> and <stmt_list> are not in the program, but begin and ; are

```
<program>   ::= begin <stmt_list> end
<stmt_list> ::= <stmt> | <stmt> ; <stmt_list>
<stmt>      ::= skip | <assgn>
<assgn>     ::= <var> = <expr>
<var>       ::= X | Y | Z
<expr>      ::= <var> + <var> | <var> - <var> | <var>
```

```
begin
X = Y + Z ;
skip ;
Y = Z ;
Z = Z - X
end
```

University of Southampton

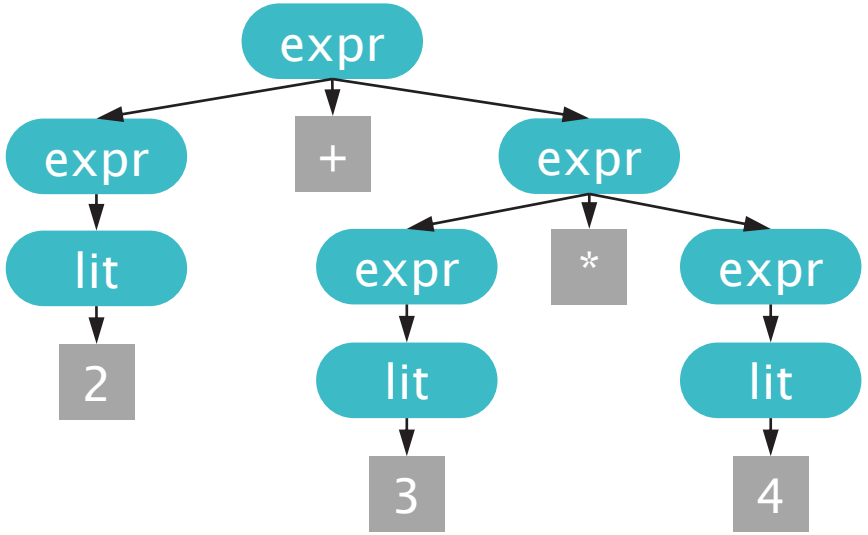
Parse Trees

- The legal programs of a language are those strings for which there is a derivation in the BNF grammar for the language.
- A derivation of a string in a BNF grammar can be represented as a tree
- At each node, the tree represents which rule of the grammar has been used to continue deriving the string
  - The child nodes represent the matches of the substrings according to the grammar
- We call such trees **parse trees**

An example parse tree

Given the following grammar, we can construct the following parse tree for “2 + 3 * 4”

```
<expr> ::= <expr> + <expr> | <expr> * <expr> | <lit>
<lit> ::= 1 | 2 | 3 | 4 | 5 | …
```



University of Southampton

Syntax to Execution

We can understand programs as a string of text, parsed as a tree according to some grammar, usually expressed in BNF.

How do we find the derivation of a string in a grammar?

Step 1 - Lexing:

- Involves translating the particular symbols or characters in the string that make up the terminals of the grammar in to tokens.

Step 2 - Parsing:

- Involves translating the sequence of tokens that make up the input string in to a tree. The parser must follow the rules of the grammar and build a tree representing the derivation.

In this latter step we often move away from parse trees and work with **Abstract Syntax Trees (AST)**

Abstract Syntax Trees

We learned in COMP2209 that abstract syntax trees are used to remove unnecessary detail regarding how a term was parsed.

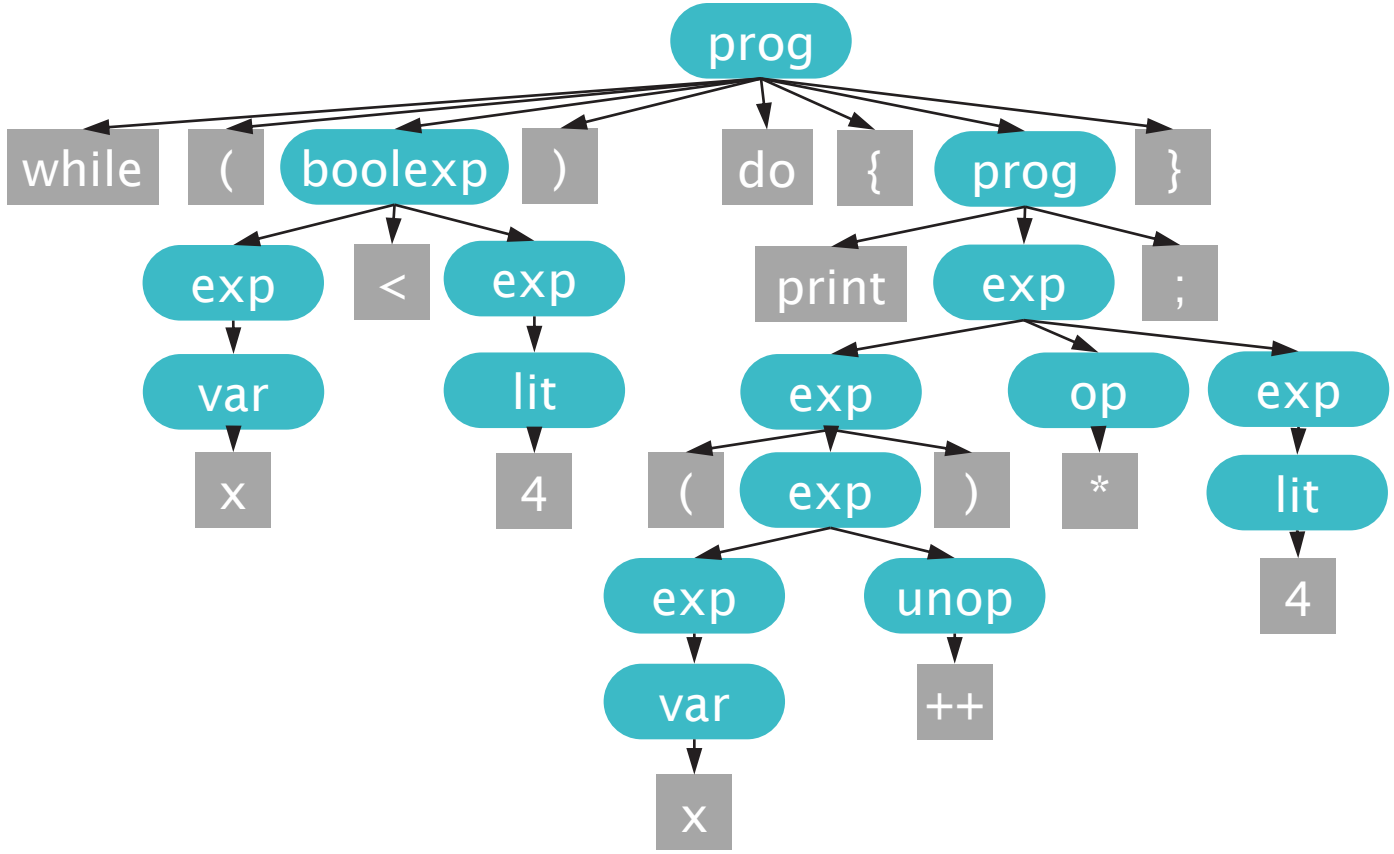
Let's consider a modified version of the grammar from those lectures and compare concrete (parse) trees and abstract syntax trees again to remind ourselves:

```
<prog>     ::= <exp> ; |
               while <boolexp> do { <prog> } |
               print <exp> ;
<exp>      ::= <var> | <lit> | <exp> <op> <exp> |
               <exp> <unop> | ( <exp> )
<boolexp>  ::=  <exp> < <exp> | <exp> == <exp> | ...
<op>       ::= + | - | * | /
<unop>     ::= ++ | --
<var>      ::= ...
```

University of Southampton

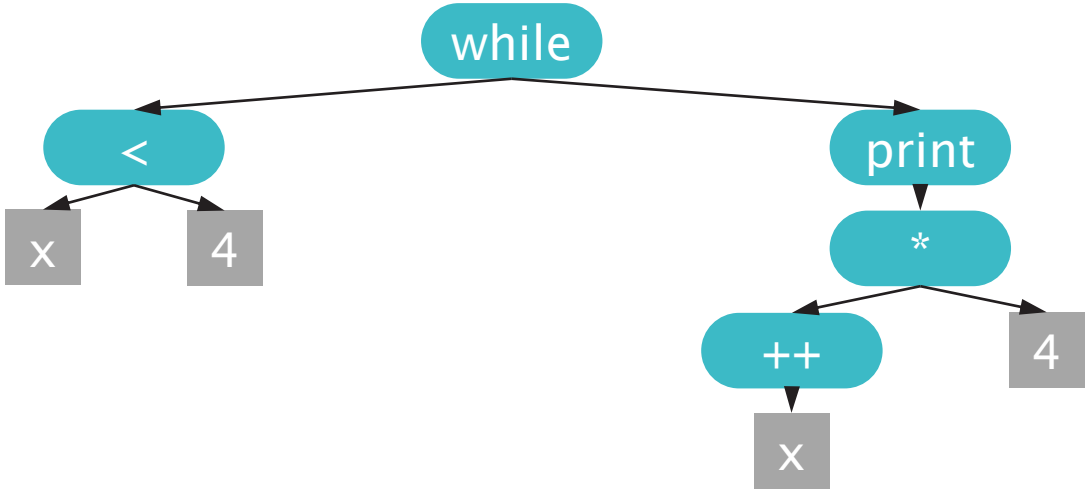
Concrete Syntax Trees (aka parse trees)

Consider the following program: `while (x < 4) do { print (x++) * 4 ; }`



Many of these nodes are unnecessary – let's remove them

Abstract Syntax Tree



This tree retains the structure of the code, but abstracts away the syntax that's used only to shape the tree

We call these Abstract Syntax Trees or ASTs

These are the structures that compilers and interpreters work with

University of Southampton

Next Lecture: Ambiguous Grammars