
University of
Southampton

University of Southampton

Syntax and Grammar

COMP2322 Programming Language Concepts

Dr Nicholas Gibbins
(based on slides by 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

- 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 COMP2311
- 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.

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**

```
<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

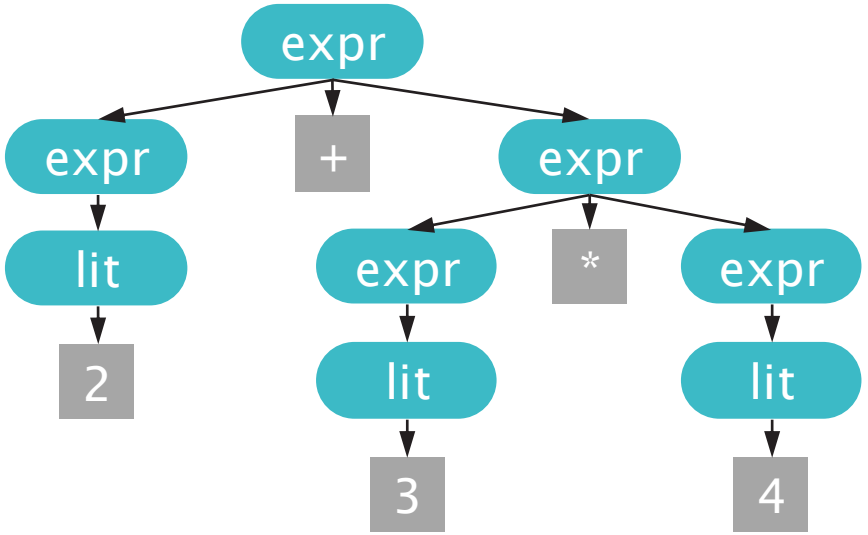
Parse Trees

- 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 into 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

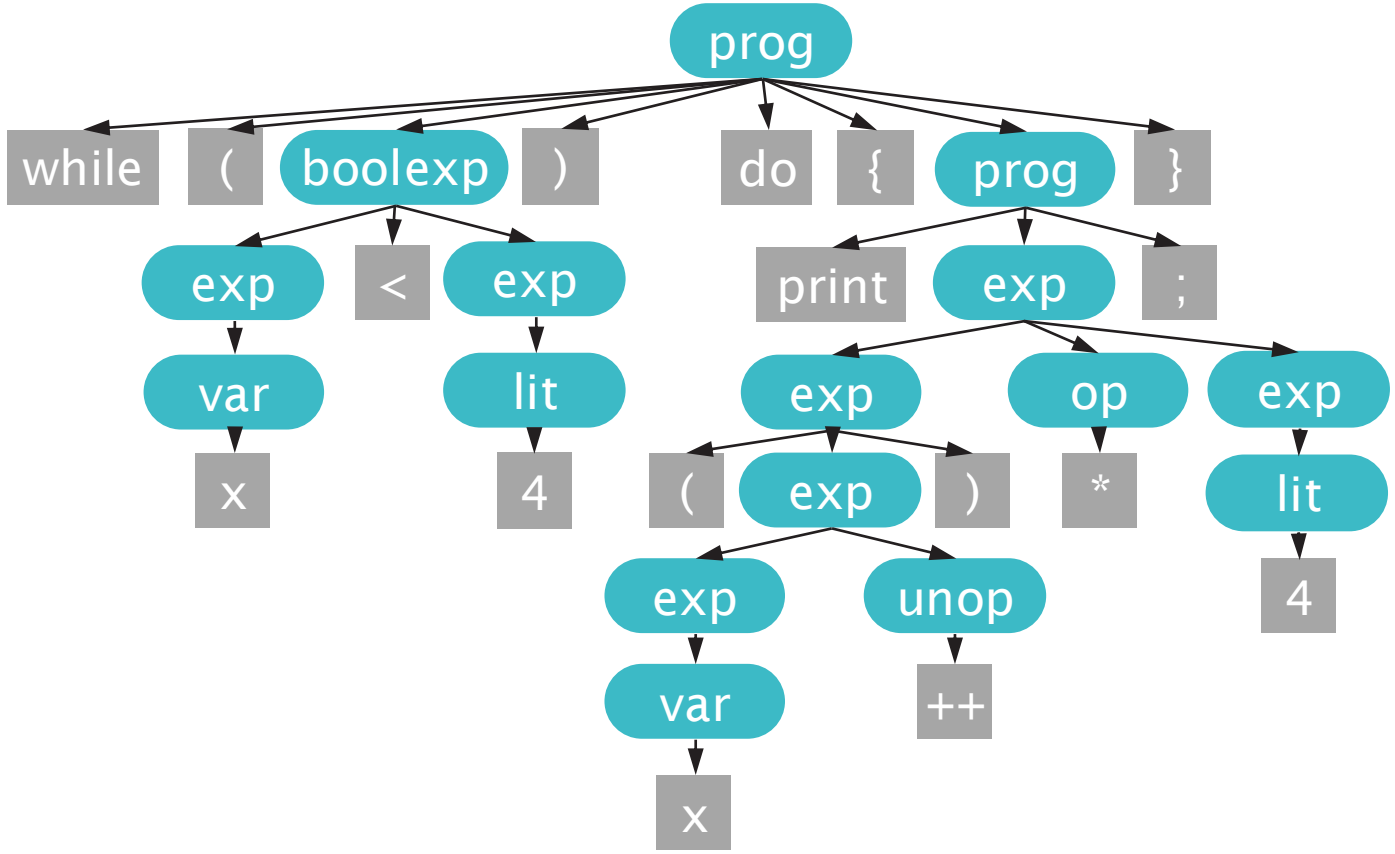
We learned in COMP2209 that abstract syntax trees are used to remove unnecessary detail regarding how a term was parsed.

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>      ::= ...
```

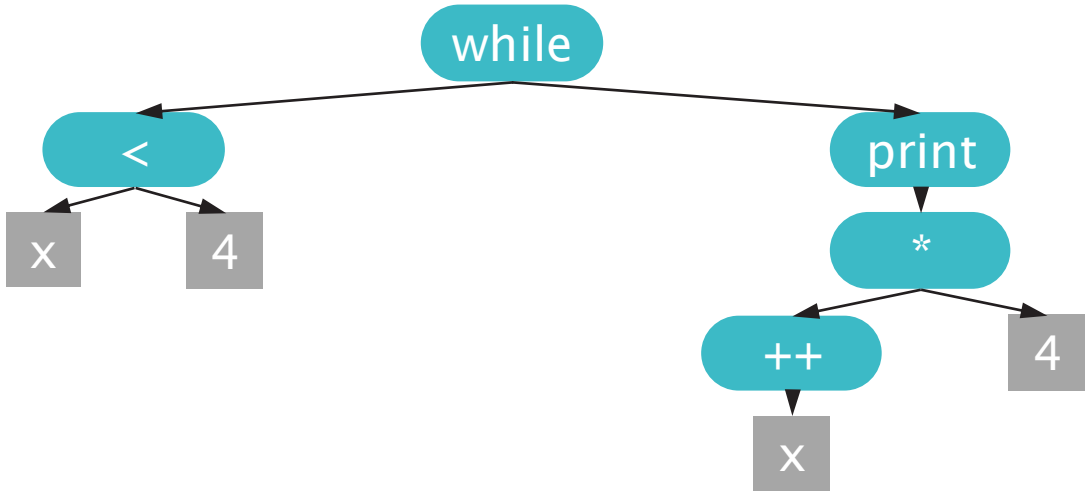
Concrete Syntax Trees (aka parse trees)

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

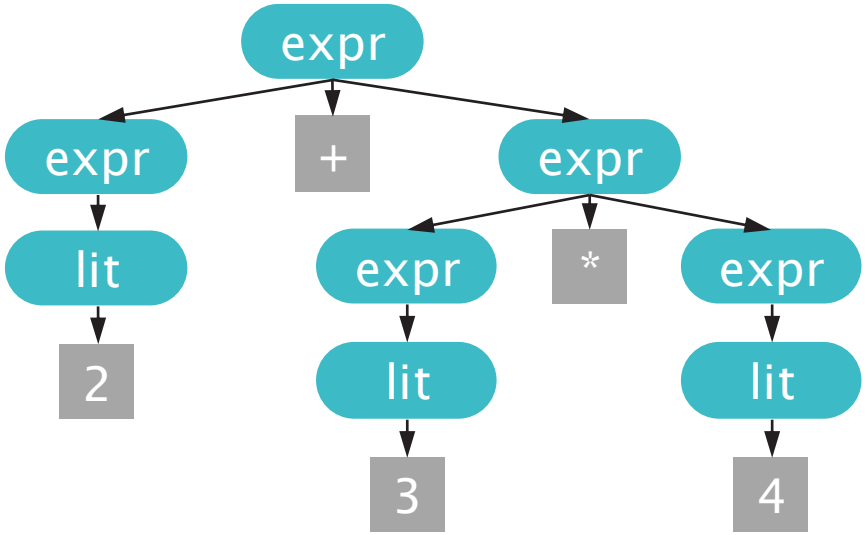
We call these Abstract Syntax Trees or ASTs

These are the structures that compilers and interpreters work with

An example parse tree

Recall this parse tree for “2 + 3 * 4”, given the following grammar:

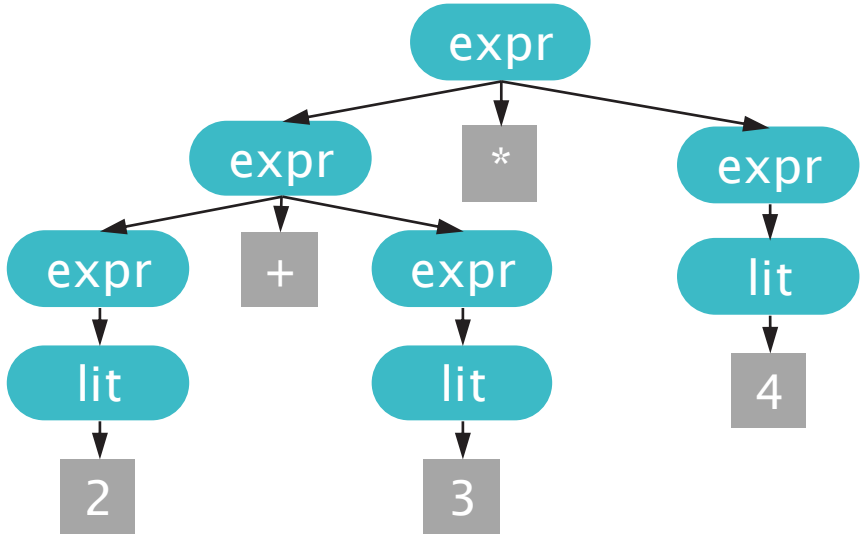
```
<expr> ::= <expr> + <expr> | <expr> * <expr> | <lit>
<lit> ::= 1 | 2 | 3 | 4 | 5 | …
```



University of Southampton

An example parse tree

We can also construct a **different** parse tree for the **same** string and grammar:



University of Southampton

Ambiguous grammars

We say that a grammar G is **ambiguous** if there exists a string s for which there exist two or more different parse trees for s using the rules of G

Ambiguity in programming language grammars is generally considered a bad thing

Two different parse trees for the same string of symbols implies two potentially different semantics for the same “program”

- e.g. what does “2 + 3 * 4” evaluate to in the above language?

University of Southampton

Resolving ambiguous grammars

How do we remove ambiguity from a grammar?

We could just put parentheses everywhere

- This is effective but impacts on readability (cf. Lisp)

We could use operator precedence:

- We can ask that one operator “binds tighter” than another operator; we say that the operator would have higher precedence
- e.g. * binds more tightly than + so * has higher precedence than +
- We understand “2 + 3 * 4” implicitly as “2 + (3 * 4)”
- n.b. higher precedence operators will appear lower in the parse tree

University of Southampton

Resolving ambiguous grammars

Consider how we might rewrite the previous grammar of + and * to resolve ambiguity:

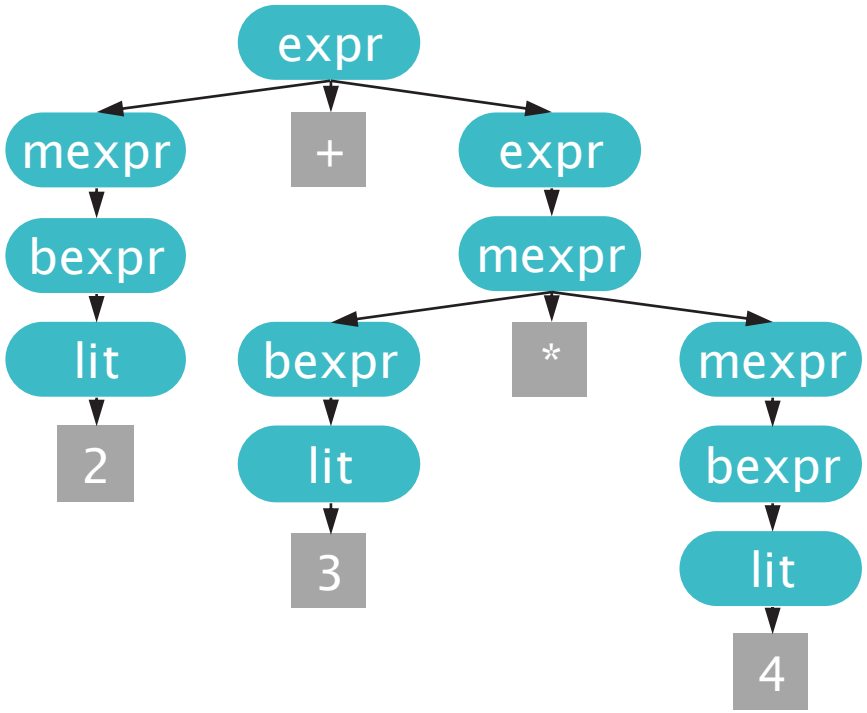
```
<expr>  ::= <mexpr> + <expr> | <mexpr>
<mexpr> ::= <bexpr> * <mexpr> | <bexpr>
<bexpr> ::= ( <expr> ) | <lit>
<lit>   ::= 1 | 2 | 3 | 4 | 5 | …
```

Here the level of the non-terminals determines precedence

Parentheses are used to “reset” precedence

Resolving ambiguous grammars

Note how the string “2 + 3 * 4” now has a unique parse tree:



Associativity

Using this, consider how we would parse the string “2 + 3 + 4”

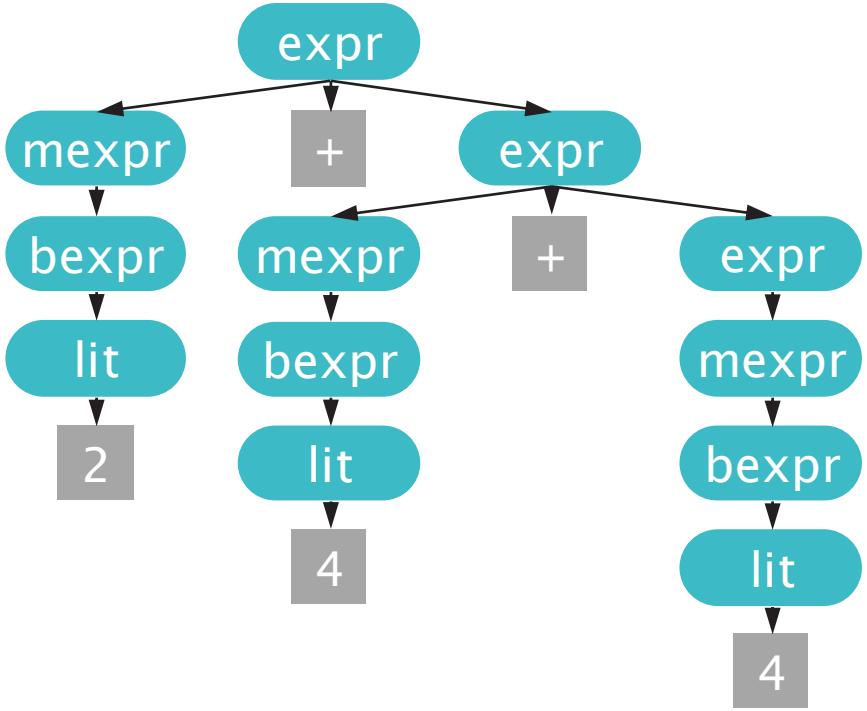
```
<expr>  ::= <mexpr> + <expr> | <mexpr>
<mexpr> ::= <bexpr> * <mexpr> | <bexpr>
<bexpr> ::= ( <expr> ) | <lit>
<lit>   ::= 1 | 2 | 3 | 4 | 5 | …
```

The operator + has the same precedence as itself so how is ambiguity resolved?

Associativity

Following the grammar, this string is implicitly derived as 2 + (3 + 4)

- This is known as being right associative



Changing associativity

Does associativity matter?

- 2 + (3 + 4) means the same as (2+3) + 4 anyway.
- But 2 - (3 - 4) is not the same as (2 - 3) - 4 !

We've seen how to guarantee right associativity, so how would we guarantee left associativity instead?

Notice the change in order!

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

Be careful with this approach - left recursive grammars don't work well with recursive descent (more on this later).

The dangling else problem

In many programming languages we can write an “if-then” statement without an “else” branch

Consider the following grammar:

```
<ifstmt> ::= if <expr> then <stmt> else <stmt> |
             if <expr> then <stmt>
```

Does the following program loop or terminate?

```
if true
then if false
     then skip
else loop
```

Which if does the else correspond to?

University of Southampton

Resolving the dangling else

The grammar for Java contains a solution for the dangling else:

Additional non-terminals are used to determine a precedence that a nested conditional in a “then” branch cannot use a single branch conditional

```
<IfThenStatement> ::=
    if ( <Expression> ) <Statement>

<IfThenElseStatement> ::=
    if ( <Expression> ) <StatementNoShortIf> else <Statement>

<IfThenElseStatementNoShortIf> ::=
    if ( <Expression> ) <StatementNoShortIf> else <StatementNoShortIf>
```

(the program on the previous slide would loop when understood as a Java program)

http://docs.oracle.com/javase/specs/

University of Southampton

Next Lecture: Lexing Concepts