
University of
Southampton

University of Southampton

Ambiguous Grammars

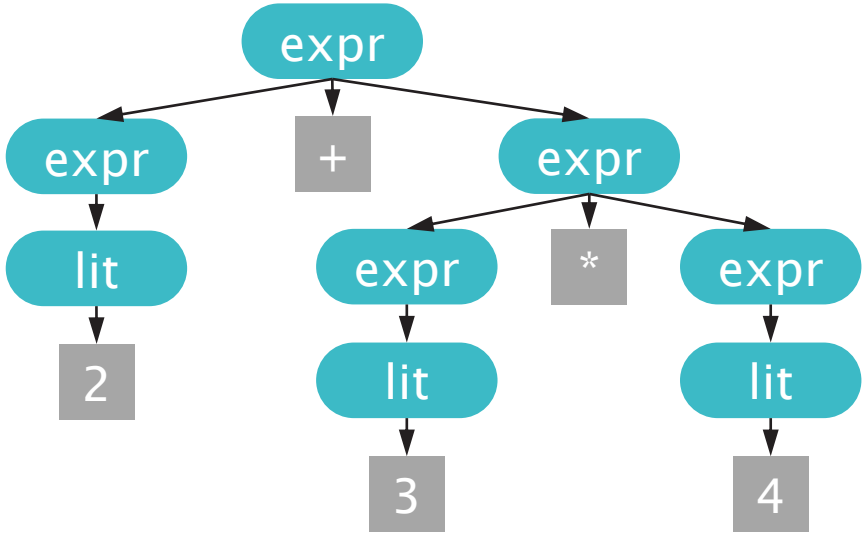
COMP2212 Programming Language Concepts

Dr Julian Rathke

An example parse tree

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

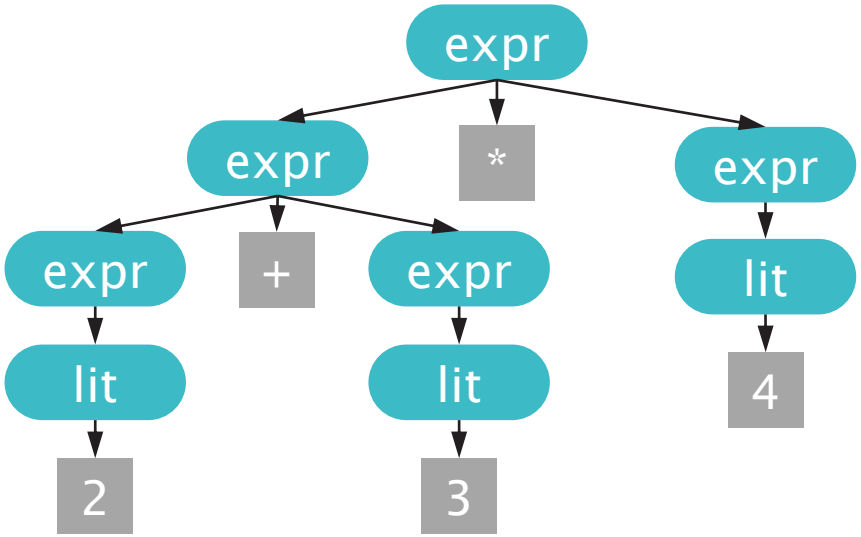
```
<expr> ::= <expr> + <expr> | <expr> * <expr> | <lit>
<lit> ::= 1 | 2 | 3 | 4 | 5 | …
```



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

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?

Following the above 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/