Binary Tree Applications

Chapter 6.6

Parse Trees

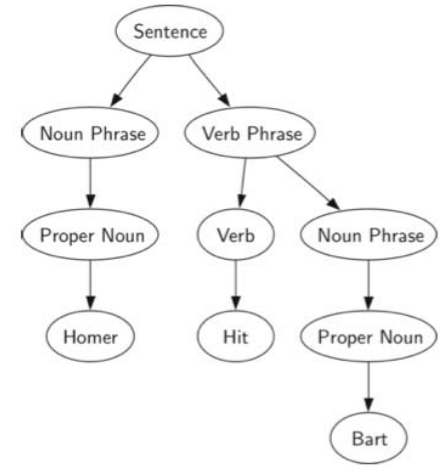


Figure 6.13: A Parse Tree for a Simple Sentence

- What is parsing?
  - Originally from language study
  - The breaking up of sentences into component parts e.g. noun phrase
- In computing compilers and interpreters parse programming languages.
- One aspect is parsing expressions.

Expression Trees

- The leaves are values and the other nodes are operators.
- We can use them to represent and evaluate the expression.
    - We work up from the bottom evaluating subtrees.
- Compilers can use this to generate efficient code - e.g. how many registers are needed to calculate this expression.

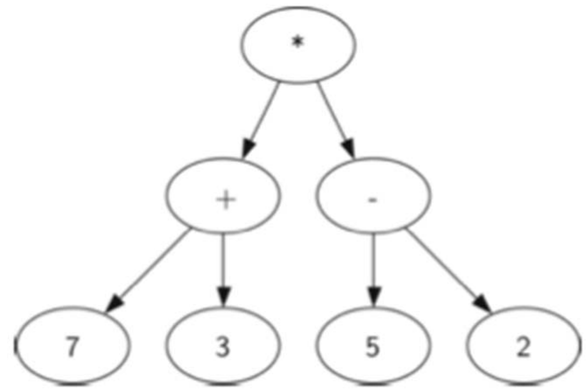


Figure 6.14: Parse Tree for $((7 + 3) * (5 - 2))$

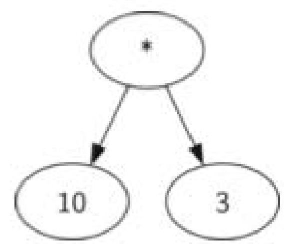


Figure 6.15: A Simplified Parse Tree for $((7 + 3) * (5 - 2))$

Tokens

- Parsing starts with recognising tokens.
- A token is a symbol made up of one or more characters (commonly separated by white space).
    - e.g. a variable name or a number or an operator “+”.
- For an expression tree the tokens are numbers, operators and parentheses.

Parsing Rules

- As we identify tokens we can apply rules to what we should do.
  - If the expression is fully parenthesised
    - a left parenthesis "(" starts a subtree
    - a right parenthesis ")" finishes a subtree

4 Rules

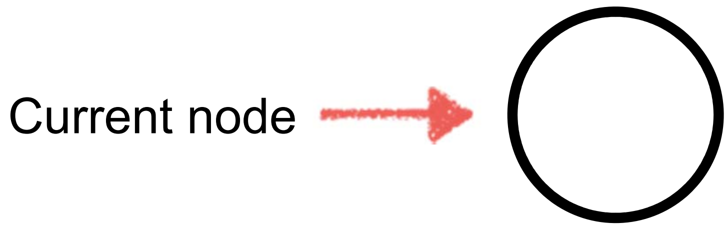
1. If the current token is a ‘(’, add a new node as the left child of the current node, and descend to the left child.

2. If the current token is in the list [‘+’,’−’,’*’,‘/’], set the root value of the current node to the operator represented by the current token. Add a new node as the right child of the current node and descend to the right child.

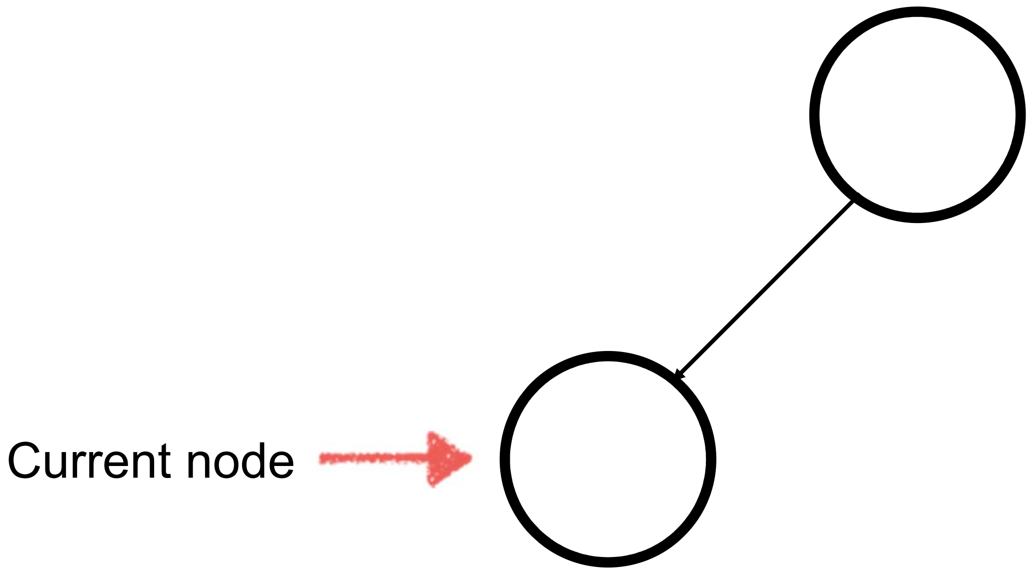
3. If the current token is a number, set the root value of the current node to the number and return to the parent.

4. If the current token is a ‘)’, go to the parent of the current node.

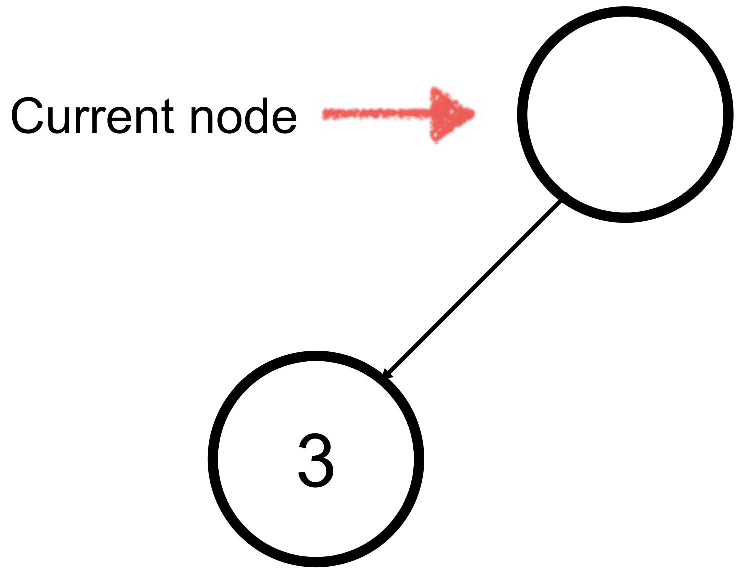
$$(3 + (4 * 5))$$



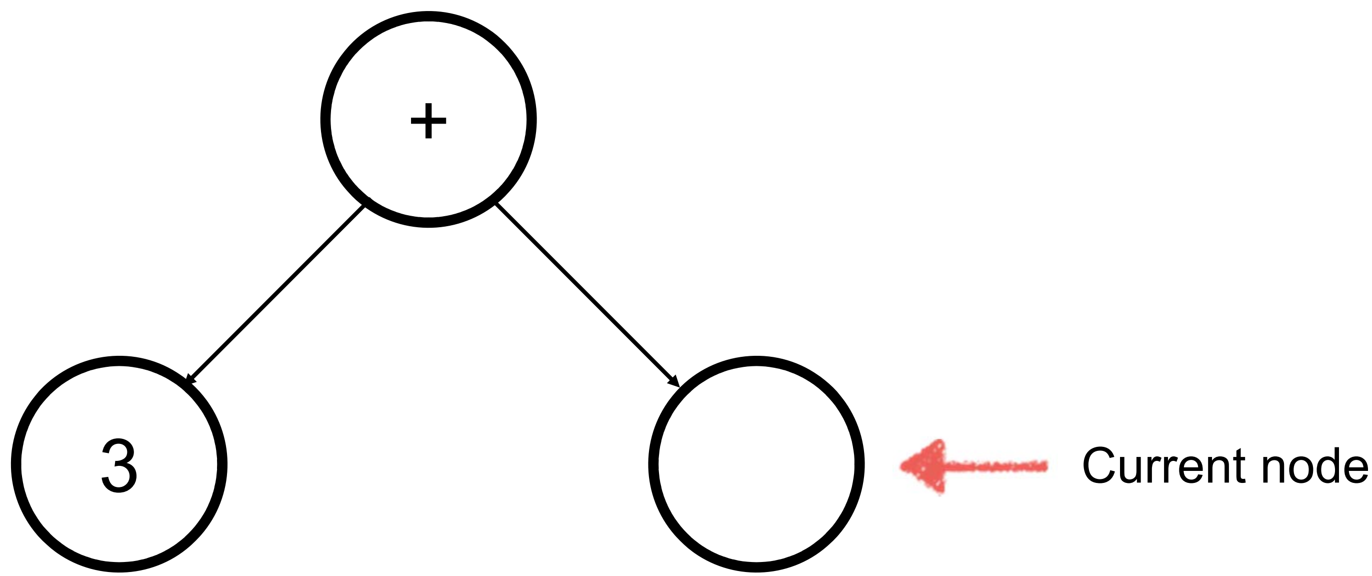
(3 + (4 * 5))



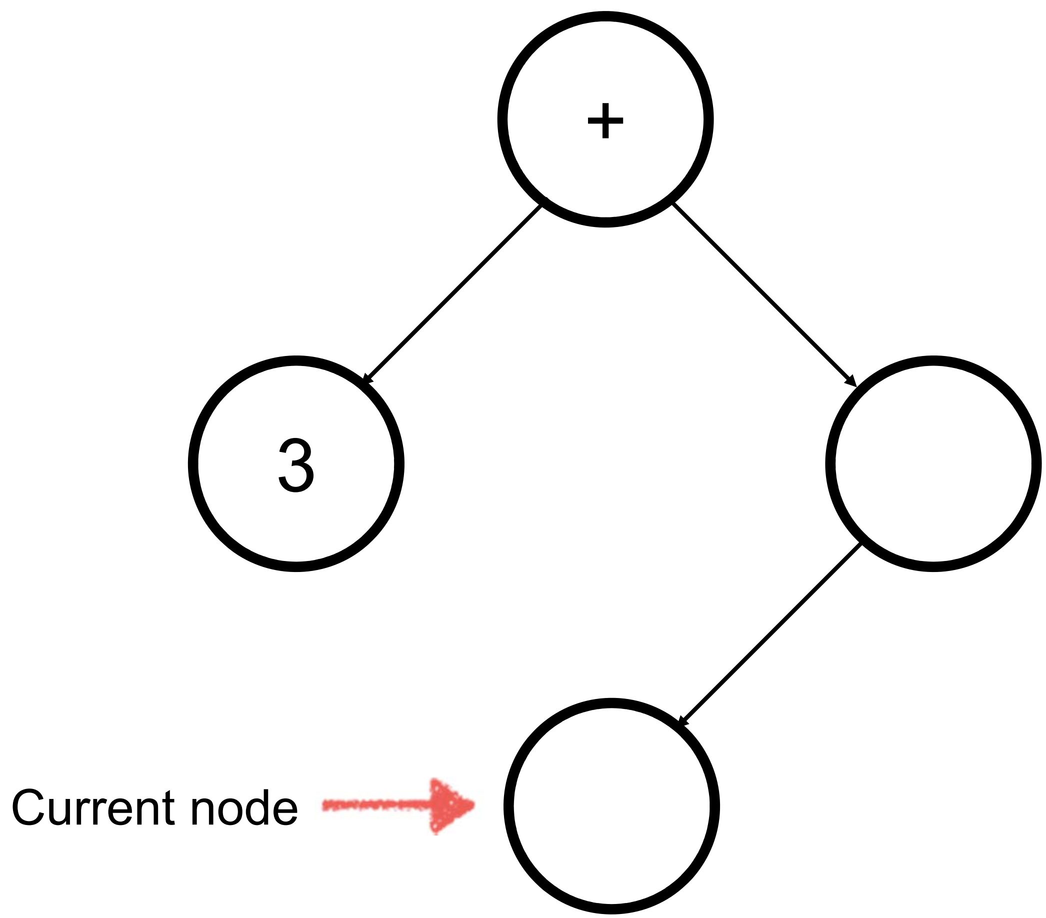
(3 + (4 * 5))



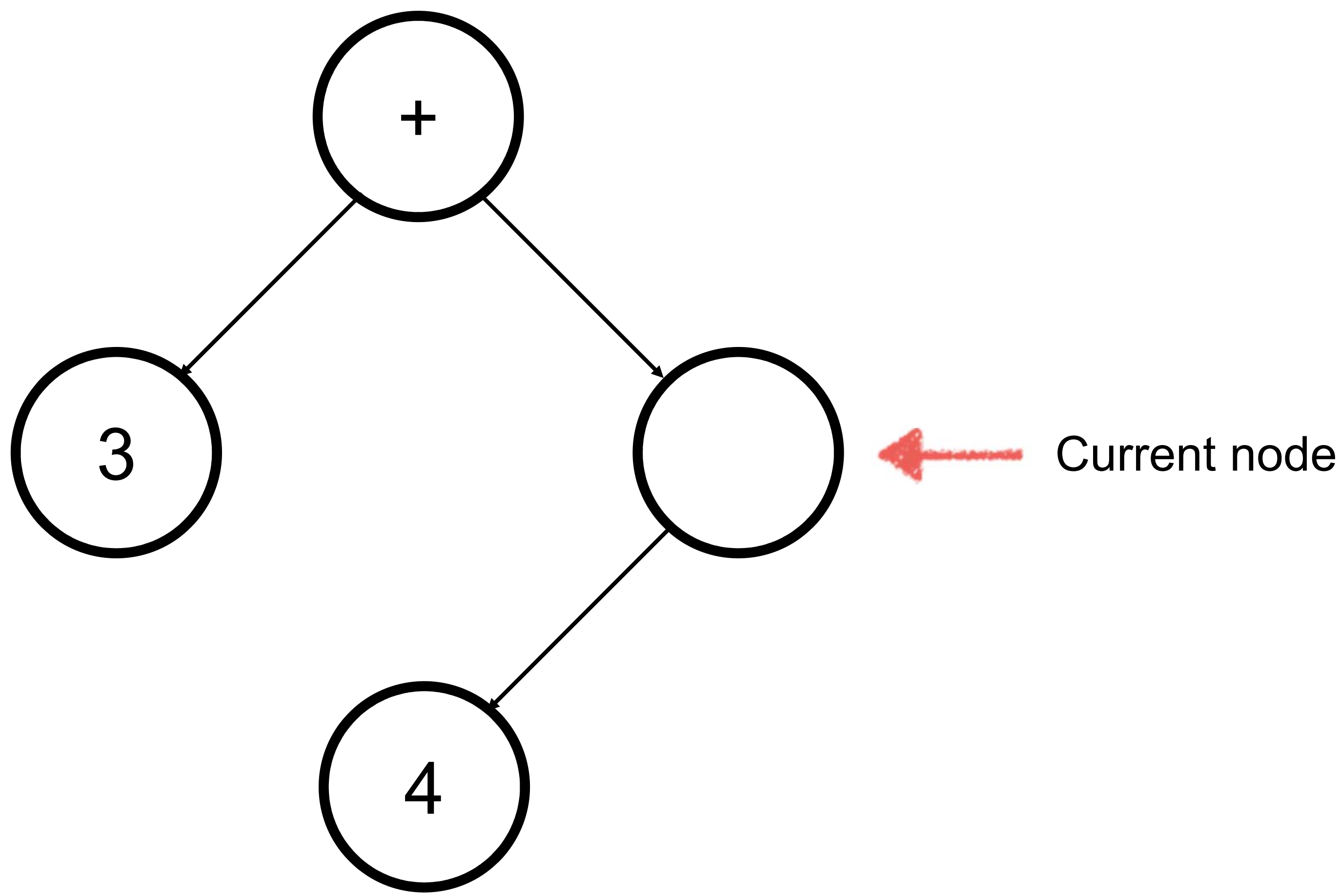
(3 **+** (4 * 5))



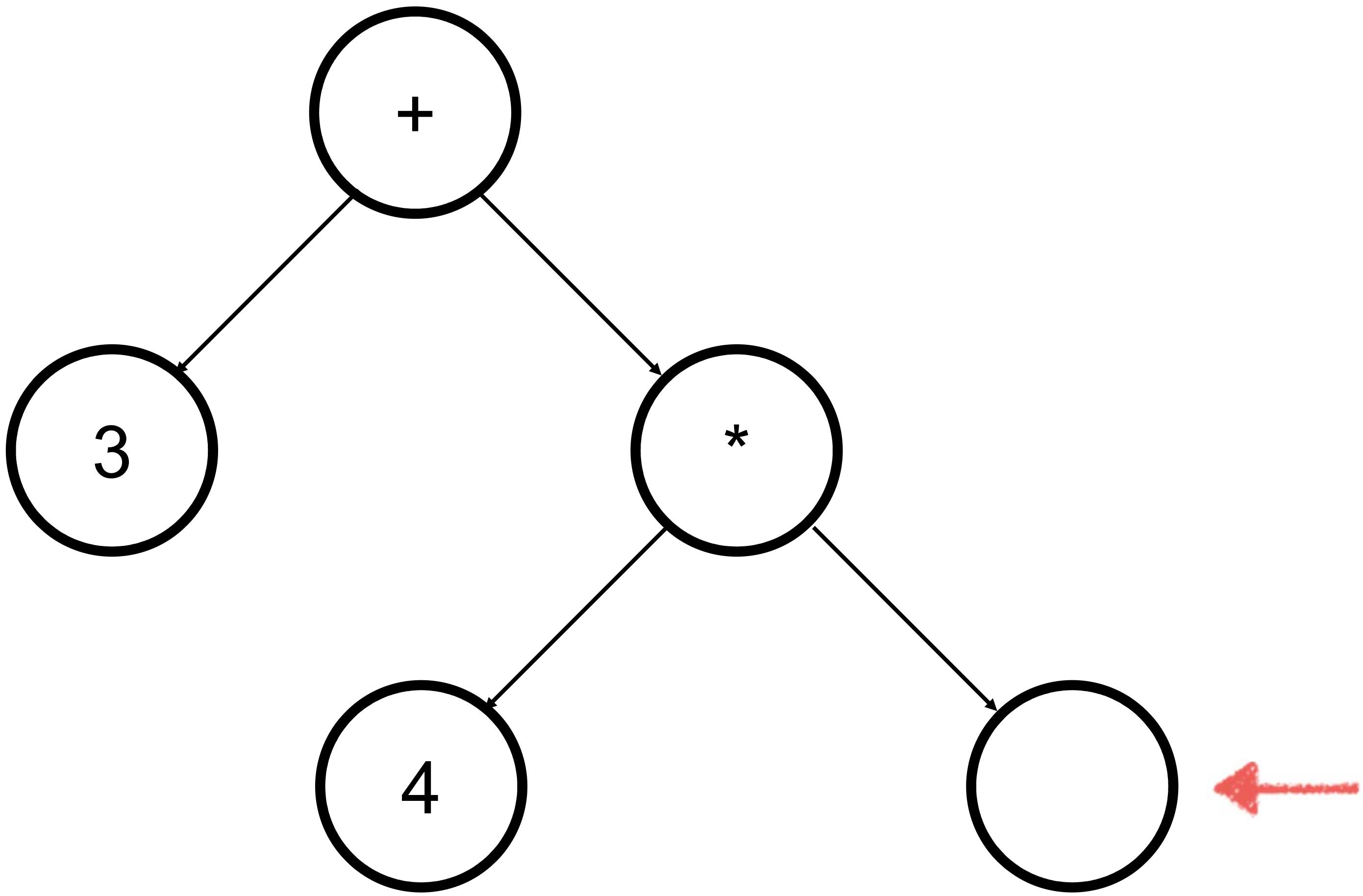
(3 + (4 * 5))



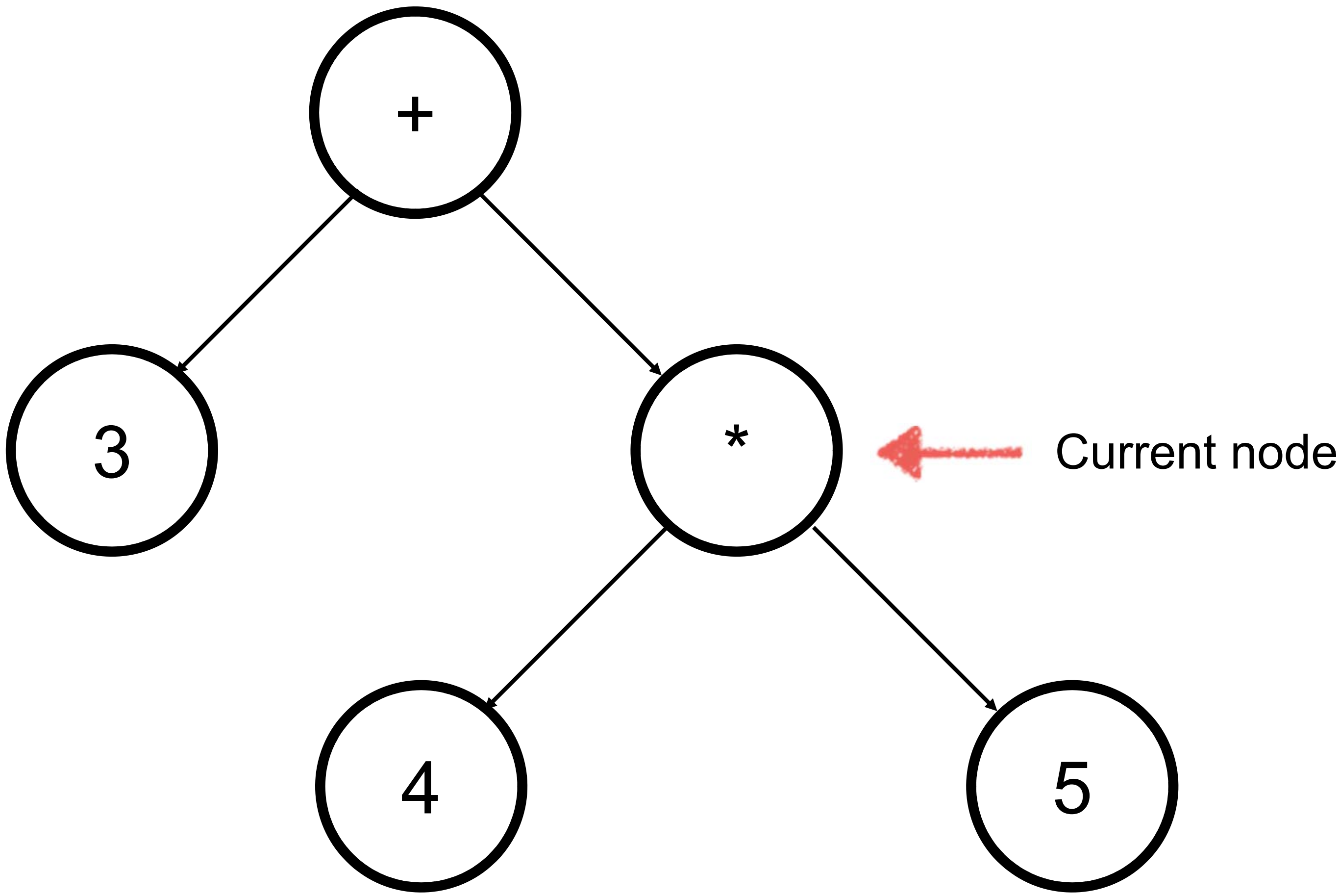
(3 + (4 * 5))



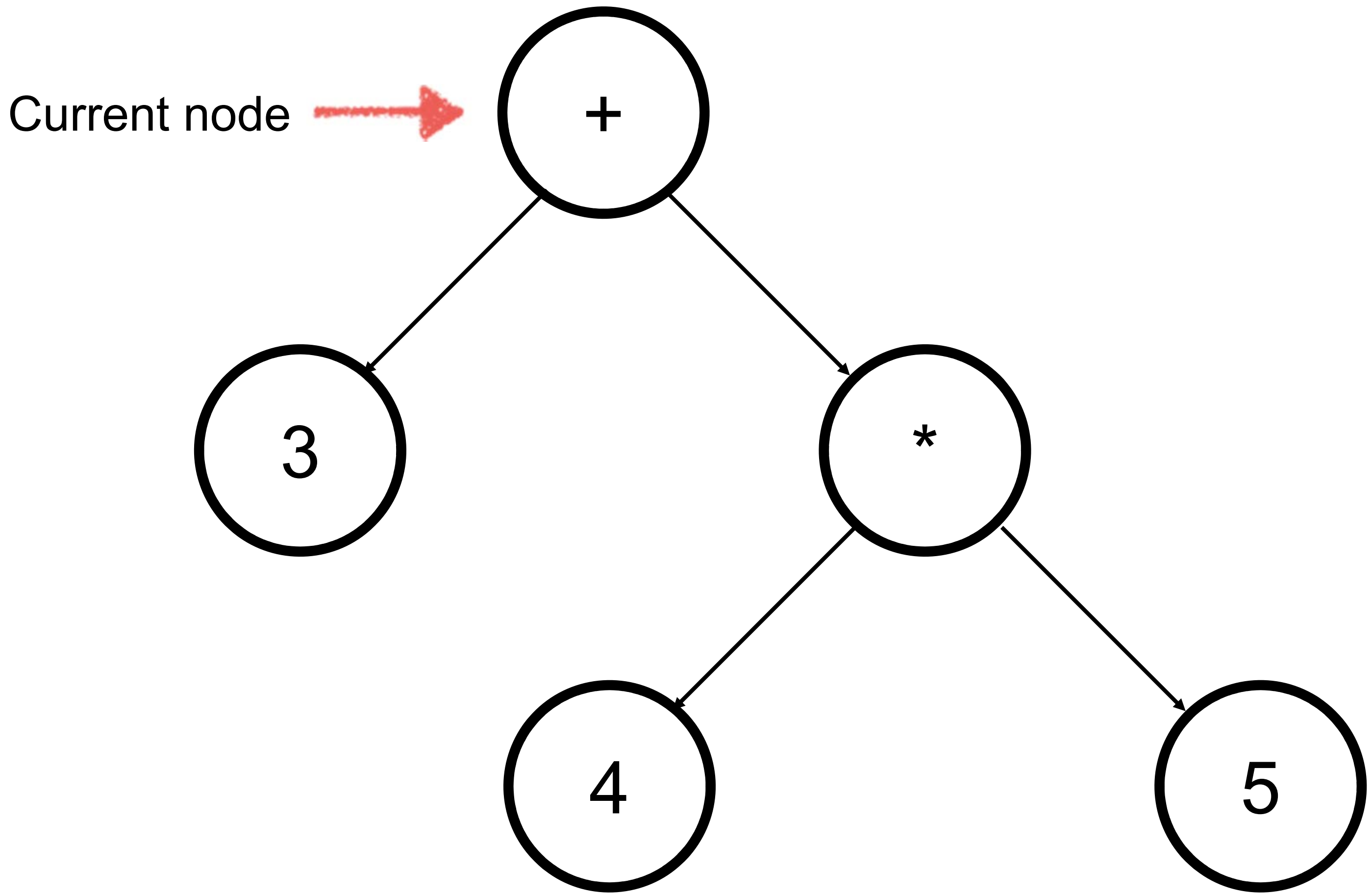
(3 + (4 * 5))



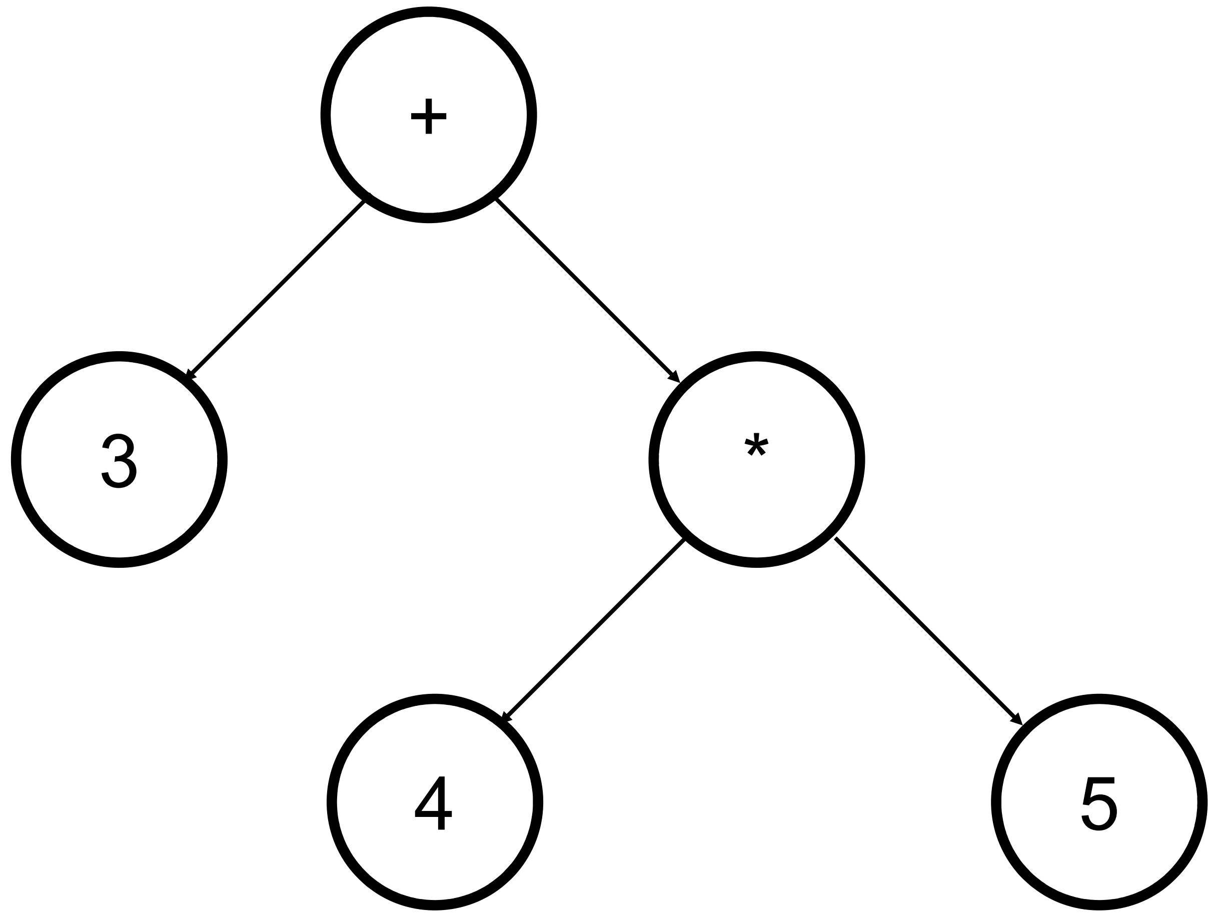
(3 + (4 * 5))



(3 + (4 * 5))



(3 + (4 * 5))



Your turn

- Generate the expression tree for

((2 * ((3 - 4) + 6)) + 2)

Keeping Track of the Parent

- We need to be able to move back up the tree.
- So we need to keep track of the parent of the current working node.
- We could do this with links from each child node back to its parent.
- Or we could store the tree in a list and use the 2 x n trick (if the tree is not complete - most won't be) then there will be lots of empty space in this list.
- Or we could push the parent node onto a stack as we move down the tree and pop parent nodes off the stack when we move back up.

Build the tree code

set up

```
def build_expression_tree(parenthesized_expression):
    """Builds an expression parse tree.

    parenthesized_expression -- a fully parenthesized expression
    with spaces between tokens
    """
    token_list = parenthesized_expression.split()
    parent_stack = Stack()
    expression_tree = BinaryTree('')
    parent_stack.push(expression_tree)
    current_tree = expression_tree
```

Implementing the rules

1. If the current token is a ‘(’, add a new node as the left child of the current node, and descend to the left child.

```
for token in token_list:
    if token == '(':
        current_tree.insert_left('')
        parent_stack.push(current_tree)
        current_tree = current_tree.get_left_child()
```

Implementing the rules

2. If the current token is in the list [‘+’,‘−’,‘*’,‘/’], set the root value of the current node to the operator represented by the current token. Add a new node as the right child of the current node and descend to the right child.

```
elif token in ['+', '-', '*', '/']:
    current_tree.set_value(token)
    current_tree.insert_right('')
    parent_stack.push(current_tree)
    current_tree = current_tree.get_right_child()
```

Implementing the rules

3. If the current token is a number, set the root value of the current node to the number and return to the parent.

```
elif is_number(token):
    current_tree.set_value(float(token))
    current_tree = parent_stack.pop()
```

```
def is_number(token):
    """Check if the token is a number."""
    try:
        float(token)
    except:
        return False
    else:
        return True
```

Implementing the rules

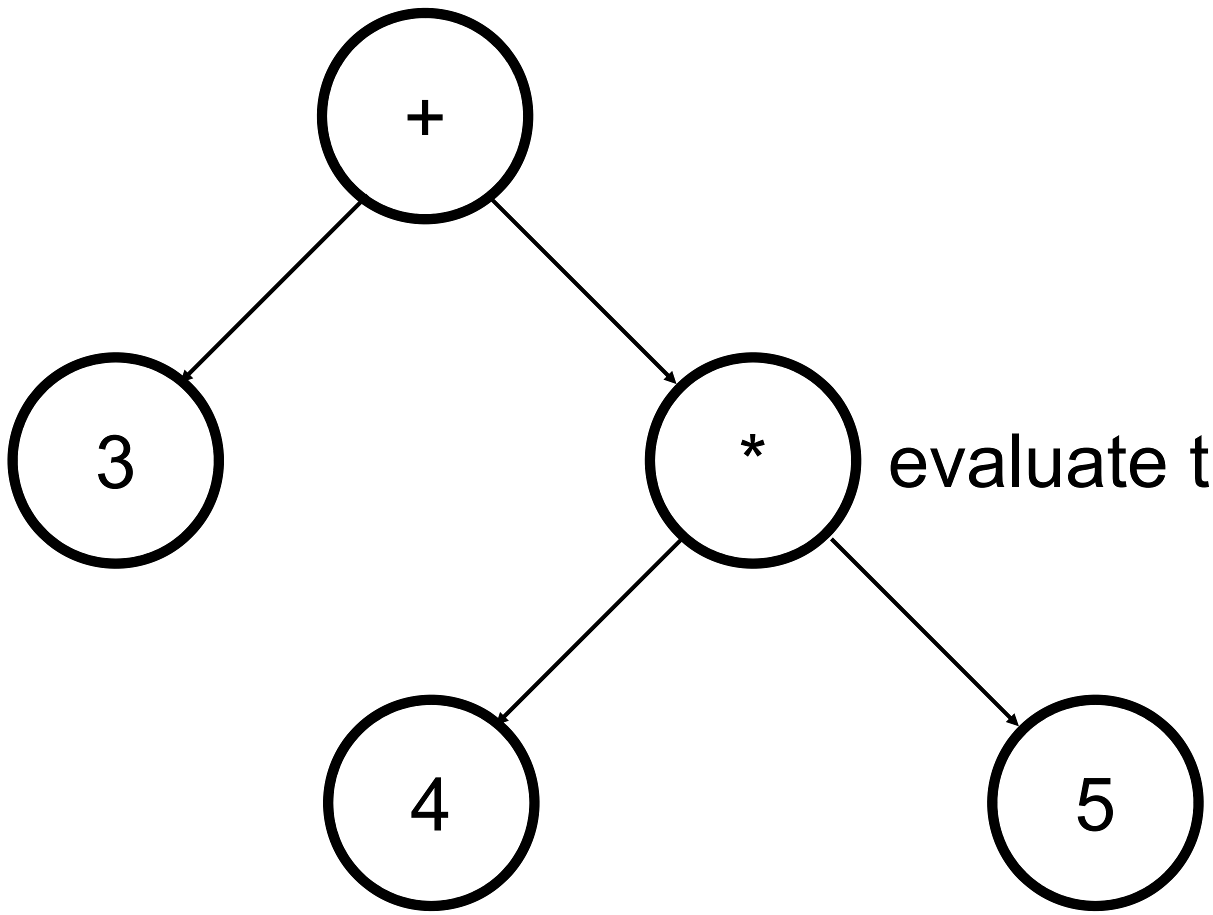
4. If the current token is a ‘)’, go to the parent of the current node.

```
elif token == ')':
    current_tree = parent_stack.pop()
else:
    raise ValueError
```

Evaluating the expression

- Once we have generated the expression tree we can easily evaluate the expression.
- In a compiler the expression would contain variables which we wouldn't know the value of until the program ran, so the evaluation would be done at run time.

How would you evaluate?



evaluate this subtree

Algorithm

- To evaluate the subtree under a node
  - if the node has children
    - the node holds an operator
    - return the result of applying the operator on the left and right subtrees
  - else the node held a number
    - return the number

Evaluation Code

```
import operator
def evaluate(expression_tree):
    """Return the result of evaluating the expression."""
    token = expression_tree.get_value()

    operations = {'+':operator.add, '-':operator.sub,
                  '*':operator.mul, '/':operator.truediv}

    left = expression_tree.get_left_child()
    right = expression_tree.get_right_child()
    if left and right:
        return operations[token](evaluate(left), evaluate(right))
    else:
        return token
```

What is that operator stuff?

- The operator module provides functions to add, subtract etc.
- We use a dictionary “operations” to connect the tokens “+”, “-”, “*” and “/” with the corresponding function.
- The line

```
operations[token](evaluate(left), evaluate(right))
```

evokes the function on its parameters.

Tree Traversals

Text book Section 6.7

- With a binary tree we can recursively travel through all of the nodes (or traverse) in three standard ways.
- We can deal with the node first then deal with the left subtree, then the right subtree.
  - This is a preorder traversal.
- We can deal with the left subtree, then with the node, then with the right subtree.
  - This is an inorder traversal (and as we will see this keeps things in order).
- We can deal with the left subtree, then the right subtree and lastly the node itself.
  - This is a postorder traversal (we used this to evaluate expression trees).

Code for printing tree traversals

```
def print_preorder(tree):
    """Print the preorder traversal of the tree."""
    if tree:
        print(tree.get_value(), end=' ')
        print_preorder(tree.get_left_child())
        print_preorder(tree.get_right_child())

def print_postorder(tree):
    """Print the postorder traversal of the tree."""
    if tree:
        print_postorder(tree.get_left_child())
        print_postorder(tree.get_right_child())
        print(tree.get_value(), end=' ')

def print_inorder(tree):
    """Print the inorder traversal of the tree."""
    if tree:
        print_inorder(tree.get_left_child())
        print(tree.get_value(), end=' ')
        print_inorder(tree.get_right_child())
```