Syntax Analysis and Code Generation: Expressions and Assignment Statements

Prof. Naga Kandasamy
ECE Department, Drexel University

A typical compilation process consists of two major steps: syntax analysis and code generation. We discuss the minimal set of concepts necessary to develop a syntax analyzer for the compiler front-end: scanners, context-free grammars, parse trees, and recursive-descent algorithms to construct them.

We start with problem of analyzing the syntax of a classic expression grammar. The complete code for the expression evaluator is available in *evaluator.py* along with a text file containing a number of test cases. The code is adapted from an excellent series of blog articles by Ruslan Spivak called "Let's Build A Simple Interpreter" (`ruslanspivak.com/lsbasi-part1/`).

Scanners

Scanning is the first in a multi-step process that the compiler uses to understand the input program. The *scanner* — also called lexical analyzer or *lexer* for short — reads a stream of characters and produces a stream of words or tokens. Consider the following expression:

```
3 + (10 - 4)/2
```

The process of decomposing the above input string into tokens is called *lexical analysis* where *token* is a string that is associated with a syntactic category along with a value. A *lexeme* is a sequence of characters that form a token. For example, the above string will be tokenized, after eliminating white spaces, as

```
(INTEGER, `3')
{PLUS, `+'}
(LPAREN, `(')
(INTEGER, `10')
(MINUS, `-')
(INTEGER, '4')
(RPAREN, `)')
(DIV, `/')
(INTEGER, `2')
```

Scanning is the only compiler pass which manipulates every character of the input program. Since scanners perform a relatively simple task, they lend themselves to very fast implementations. Tools for automated scanner generation are common; for example, Flex (Fast Lexical Analyzer Generator). The *Scanner* class in the supplied *evaluator.py* file is an example of a hand-crafted scanner that recognizes various tokens occurring in an expression.

Parsers

Parsing is the second stage of the compiler's front end. It works with the program as transformed by the scanner — as a stream of tokens where each token is annotated with a corresponding syntactic category. The parser derives a syntactic structure for the program — called a *parse tree* — fitting the tokens into a grammatical model of the source programming language. The parser is responsible for recognizing syntax –

that is, determining if the program being compiled is a valid sentence in the syntactic model of the programming language. The traditional solution to describe programming-language syntax is to use a *context-free grammar* (*CFG*) — a set of recursive rules that can be used to generate strings or sentences. The collection of sentences that can be derived from the underlying CFG is called a *language defined by that grammar.* Grammars offer the following advantages to a compiler writer:

- Provides a precise and easy-to-understand syntactic specification of a programming language.
- Efficient parser implementations can be automatically constructed for certain classes of grammar to determine if the source program is syntactically well-formed.
- New constructs can be added to a language more easily when there is an existing implementation based on a grammatical description of the language.

Consider a grammar, which we will call *CV*, with the following two rules or *productions*:

Rule			
0	*Goal*	$\rightarrow$	*CatVoice*
1	*CatVoice*	$\rightarrow$	`meow` *CatVoice*
2		$\vert$	`meow`

Rule 1 reads "*CatVoice* can derive the word `meow` followed by more *CatVoice*" whereas rule 2 reads "*CatVoice* can also derive the string `meow`." Here *CatVoice* is a syntactic variable representing the set of strings that can be derived from the grammar. We call such a variable a *nonterminal* symbol. Each word in the language defined by the grammar is a *terminal* symbol. To derive a sentence, we start with a prototype string containing just the goal symbol, which in our case is *CatVoice.* We pick a nonterminal symbol, α, in the prototype string, choose a grammar rule, $\alpha \rightarrow \beta$, and rewrite α with β. We repeat this rewriting process until the prototype string contains no more nonterminals, at which point it consists entirely of words or terminal symbols, and is a sentence in the language.

To derive a sentence using our *CV* grammar, we start with a string that consists of the goal symbol, *CatVoice.* We can rewrite *CatVoice* with either rule 1 or rule 2. If we rewrite *CatVoice* with rule 2, the sentence becomes `meow` and has no further opportunities for rewriting. This is a valid sentence in the language. The other option, rewriting the initial string with rule 1, leads to a string with two symbols: `meow` *CatVoice.* This string has one remaining nonterminal; rewriting it with rule 2 leads to the sentence `meow meow` which is also a valid sentence in the language. Rule 1 lengthens the string whereas rule 2 eliminates the nonterminal *CatVoice.* Note that the string can never contain more than one instance of *CatVoice.* All valid strings in *CV* are derived by zero or more applications of rule 1 followed by rule 2. Applying rule 1 k times followed by rule 2 generates a string with $k + 1$ `meow`s.

Now consider a more complex example of a classic expression grammar:

Rule					
0	*Goal*	$\rightarrow$	*Expr*		
1	*Expr*	$\rightarrow$	*Term*	$+$	*Expr*
2		$\vert$	*Term*	$-$	*Expr*
3		$\vert$	*Term*		
4	*Term*	$\rightarrow$	*Factor*	$\times$	*Term*
5		$\vert$	*Factor*	/	*Term*
6		$\vert$	*Factor*		
7	*Factor*	$\rightarrow$	(*Expr*)		
8		$\vert$	`number`		

This grammar can derive expressions such as those shown below.

```
1 + 2 + 3 + 4 + 5
10/2 + 5 * 2 - 3
1 + 2 * (3 + 4) + 5
```

The *parse tree* generated by processing the input statement `1+2*(3+4)+5` while using the foregoing grammar is shown in Fig. 1. This tree represents the syntactic structure of a language construct as per the grammar. The parse tree records the sequence of rules applied by the parser to recognize the input string — in other words, how the start symbol of our grammar, which is *Expr*, derives a certain string in the programming language. The root is labeled with the start symbol of the grammar; each interior or non-terminal node represents the application of a grammar rule, such as $Expr$, $Term$, or $Factor$; and each leaf node represents a token (shown in boldface). The call stack of the parser implicitly represents a parse tree, which is automatically built in memory by the parser as it tries to recognize a certain language construct.

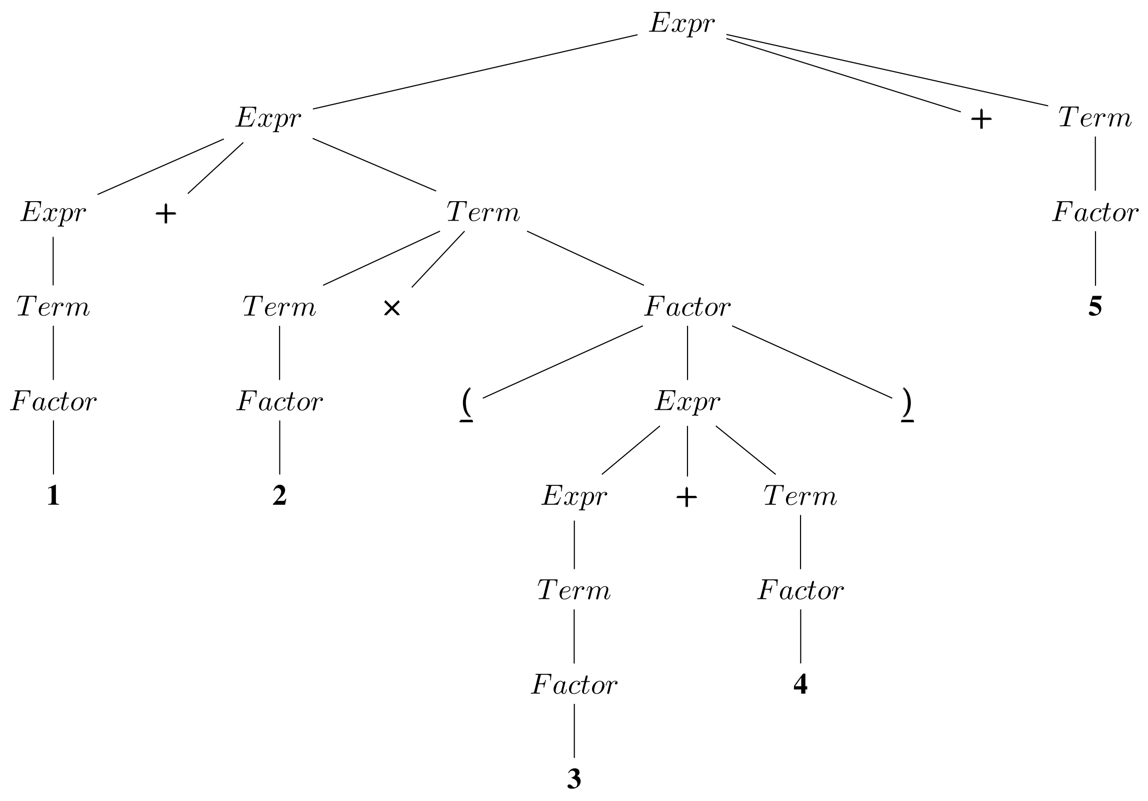

Fig. 1: A parse tree for the expression `1+2×(3+4)+5`.

An *abstract syntax tree* (*AST*) differs from a parse tree in that it is a simplified syntactic representation of source code. It does not show the whole syntactic clutter, but represents the parsed string in a structured way, discarding all information that may be important for parsing the string, but isn't needed for analyzing it. The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural or content-related details. For instance, grouping parentheses are implicit in the tree structure and do not need to appear within an AST.

Figure 2 shows the AST generated while parsing the expression `1+2*(3+4)+5`. Here, each interior node and the root node represents an operator, and the node's children represent the operands associated with that operator. Main differences between ASTs and parse trees are:

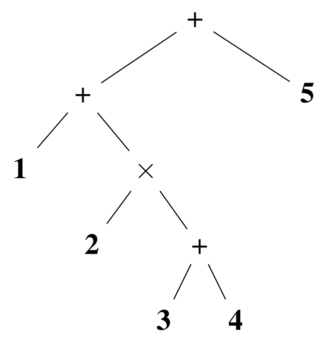

Fig. 2: An abstract syntax tree for the expression `1+2×(3+4)+5.` Operators with higher precedence are placed lower in the tree.

- An AST uses operators as the root and interior nodes, and operands as the children of these nodes.
- ASTs do not represent every detail from the actual syntax. For example, they do not contain rule nodes or parentheses.
- ASTs are more compact data structures compared to parse trees for the same language construct.

Operator precedence is encoded within an AST as follows. Parenthesized expressions are evaluated first. Multiplication and division operations have higher precedence than addition and subtraction, and if operators have equal precedence, they associate from left to right. Since the AST in Fig. 2 is the only possible one, given the grammar used, this means that associativity and precedence are actually side effects of the grammatical structure. *In particular, subtrees that are lower down on the AST are always processed first, and positioning a production rule further down in the grammar guarantees a lower relative position in the AST.* Higher-precedence operators should then be placed lower down in the grammar, as is the case in the grammar used here. Revisiting our grammar, the only way to get to a *term* is through an *expr* and so multiplication is always further down the tree than addition. Parenthesis, since they are at the bottom of the grammar, have higher precedence than any other operator. Associativity is determined by the left recursion process used to construct the tree.

Parser Implementation

Given an expression, the corresponding AST can be generated using a *recursive-descent parser*, which is a top-down parser that uses a set of recursive functions to process the input string. The parser begins by constructing the top node of the AST and then gradually constructs lower nodes. We will now discuss such an implementation.

The code snippet below shows the various classes needed to implement the AST data structure. They inherit from the dummy *AST* base class. The *BinaryOperator* class represents binary operators that form the AST's interior nodes — *left* points to the node representing the left operand whereas *right* points to the right operand, and *op* holds a token for the operator itself; for example, `(PLUS, '+')` for the plus operator, `(MINUS, '-')` for the minus operator, and so on.

```
class AST(object):
    pass

class BinaryOperator(AST):
    def __init__(self, left, op, right):
        self.left = left
        self.token = self.op = op
        self.right = right

class Number(AST):
    def __init__(self, token):
        self.token = token
        self.value = token.value
```

The parser implementation closely follows the classic expression grammar shown earlier in the document. We initialize the parser object with the previously instantiated scanner object. The *expr()* method aims to obtain a production involving *term* — thus the call to *term()* in line number 36. In turn, the *term()* method seeks to obtain a production involving *factor* (line 21). The *factor()* method either returns a terminal node (a *Number* object containing an integer value) or if it encounters a nested expression at that level, calls *expr()* in recursive fashion. Operators with higher precedence end up occurring lower in the tree. The *consume()* method compare the current token type with the passed token type and if the types match, consumes the token and assigns the next token to *current_token*. *It's important to point out again that the call stack of the parser implicitly represents the AST, which is automatically built in memory by the parser as it tries to recognize the underlying grammar.*

```
1   class Parser(object):
2       def __init__(self, scanner):
3           self.scanner = scanner
4           self.curr_token = self.scanner.get_next_token()
5
6       def consume(self, token_type):
7           self.curr_token = self.scanner.get_next_token()
8
9       def factor(self):
10          token = self.curr_token
11          if token.type == INTEGER:
12              self.consume(INTEGER)
13              return Number(token)
14          elif token.type == LPAREN:
15              self.consume(LPAREN)
16              node = self.expr()
17              self.consume(RPAREN)
18              return node
19
20      def term(self):
21          node = self.factor()
22          while self.curr_token.type in (MUL, DIV):
23              token = self.curr_token
24              if token.type == MUL:
25                  self.consume(MUL)
26              elif token.type == DIV:
27                  self.consume(DIV)
28
29              node = BinaryOperator(left = node, op = token,
30                                    right = self.factor())
31
32          return node
33
34
```

```
35     def expr(self):
36         node = self.term()
37         while self.curr_token.type in (PLUS, MINUS):
38             token = self.curr_token
39             if token.type == PLUS:
40                 self.consume(PLUS)
41             elif token.type == MINUS:
42                 self.consume(MINUS)
43 
44             node = BinaryOperator(left = node, op = token,
45                                   right = self.term())
46 
47         return node
48 
49     def parse(self):
50         return self.expr()
```

Figures 3 and 4 illustrate how the AST is built for the expressions `1+2+3+4+5` and `1+2+3*(4+5)+6`, respectively, in step-by-step fashion. The parser code builds the tree such that each *BinaryOperator* node adopts the current value of the node variable as its left child and the result of a call to either *term()* or *factor()* as its right child — see lines 42 and 29 in the *Parser* code. Therefore, it effectively pushes nodes that occur on the left side progressively down the tree as the tree is being constructed.

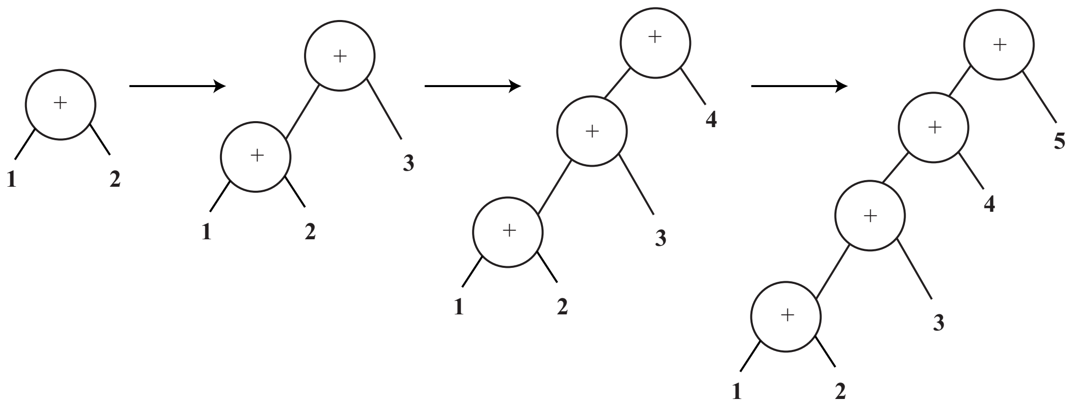

Fig. 3: Example of how the parser builds an AST for the expression `1+2+3+4+5`.

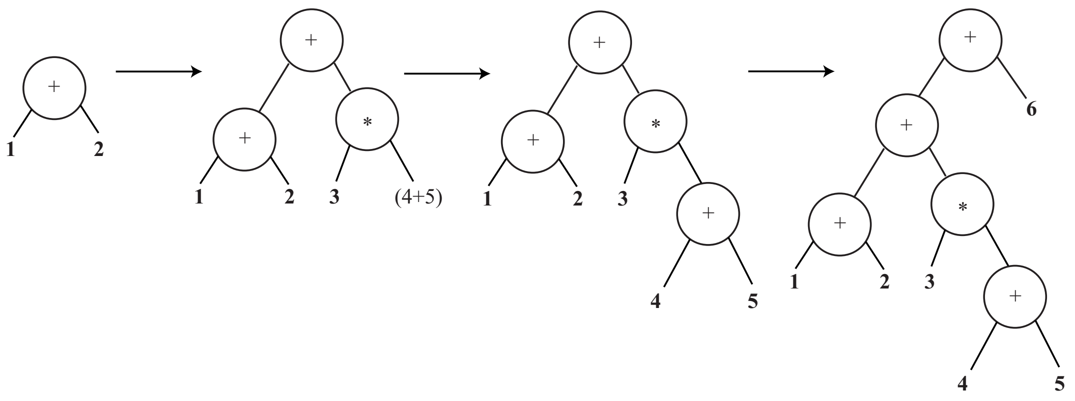

Fig. 4: Example of how the parser builds an AST for the expression `1+2+3*(4+5)+6`.

Code Generation

An AST is the final output of the parser and this is the *intermediate representation* (*IR*) that we will use to generate code for our stack-based VM. We must traverse the AST in *postorder fashion* of the AST — special case of depth-first search — which starts at the root node and recursively visits the children of each node from left to right, is shown below. The reasons for using a postorder traversal are twofold: (1) we must first evaluate interior nodes lower in the tree since they represent operators with higher precedence; and (2) we must evaluate operands associated with an operator before applying that operator. Postorder actions include addition, subtraction, multiplication, or division for an interior node and a simpler action such as returning the integer value associated with a terminal node.

Let's assume that the AST for expression `1+2*(3+4)+5` has been generated. We then perform a postorder depth-first traversal of the tree to obtain the postfix form as `1234+*+5+`. The VM commands to evaluate the expression are generated by processing the postfix form from left to right as follows:

```
// Evaluate 1+2*(3+4)+5. Result is placed on top of the working stack
set sp 256
push constant 1
push constant 2
push constant 3
push constant 4
call add 2
call mult 2
call add 2
push constant 5
call add 2
```

Referring to the above code snippet, note that deeper nodes are always processed first. The depth-first traversal forces the `3 + 4` to be done first (since that subtree is the deepest in the tree); then, moving up the tree, result of the previous addition is multiplied by 2; then 1 is added to the accumulated subexpression; and, finally the `5` is added into the total.

There are two basic approaches to tree walking. Either we embed methods within each node class which define what to do when visiting that class or we encapsulate these methods within an external visitor. Such a visitor design pattern is advantageous since it allows us to alter tree-walking behavior without modifying the node classes themselves. Furthermore, rather than build external visitors manually, we can automate visitor construction. The code snippet below implements the *Evaluator* class that evaluates the expression embedded within an AST. The implementation follows the visitor design pattern.

```python
class NodeVisitor(object):
    def visit(self, node):
        method_name = 'visit_' + type(node).__name__
        visitor = getattr(self, method_name, self.generic_visit)
        return visitor(node)

    def generic_visit(self, node):
        raise Exception('No visit_{} method'.format(type(node).__name__))

class Evaluator(NodeVisitor):
    def __init__(self, parser):
        self.parser = parser

    def visit_BinaryOperator(self, node):
```

```
15         if node.op.type == PLUS:
16             return self.visit(node.left) + self.visit(node.right)
17         if node.op.type == MINUS:
18             return self.visit(node.left) - self.visit(node.right)
19         if node.op.type == MUL:
20             return self.visit(node.left) * self.visit(node.right)
21         if node.op.type == DIV:
22             return self.visit(node.left) / self.visit(node.right)
23
24     def visit_Number(self, node):
25         return node.value
26
27     def evaluate(self):
28         tree = self.parser.parse()
29         return self.visit(tree)
30
31 if __name__ == "__main__":
32     evaluator = Evaluator(parser)
33     result = evaluator.evaluate()
34     print(result)
```

The *Evaluator* object inherits from the *NodeVisitor* object, which constructs the parse tree in line 28. The *NodeVisitor* specializes the *visit()* method based on the type of node visited — see lines 3 and 4. For example, if an interior node of type *BinaryOperator* is encountered, the corresponding *visit()* method is specialized as *visit_BinaryOperator()* based on the object type, which explore nodes under it from left to right (lines 15–22). Similarly, the *visit()* method is specialized as *visit_Number()* when a terminal node is encountered and returns integer value associated with that node (lines 24–25).

Extending the Expression Grammar with Unary Operators

A unary operator operates on a single operand with the following set of rules:

- The unary minus ($-$) operator produces the negation of its numeric operand.
- The unary plus ($+$) operator yields its numeric operand without change.
- The unary operators have higher precedence than the binary operators $+$, $-$, $*$, and $/$.

The updated expression grammar that includes unary plus and minus operators is shown below.

$$
\begin{array}{lcl}
\textit{Expr} & \rightarrow & \textit{Term}\ \text{PLUS}\ \textit{Expr} \\
 & | & \textit{Term}\ \text{MINUS}\ \textit{Expr} \\
 & | & \textit{Term} \\
\textit{Term} & \rightarrow & \textit{Factor}\ \text{MULTIPLY}\ \textit{Term} \\
 & | & \textit{Factor}\ \text{DIVIDE}\ \textit{Term} \\
 & | & \textit{Factor} \\
\textit{Factor} & \rightarrow & \text{PLUS}\ \textit{Factor} \\
 & | & \text{MINUS}\ \textit{Factor} \\
 & | & \text{LPAREN}\ \textit{Expr}\ \text{RPAREN} \\
 & | & \texttt{number}
\end{array}
$$

The unary operators are added to the *Factor* rule since these operators have higher precedence than binary operators. The *Factor* rule references itself, allowing us to derive valid expressions such as `--+-3` that contain many unary operators. The AST generated for the expression `-10--5` is shown in Fig. 5.

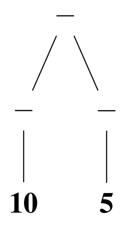

Fig. 5: An abstract syntax tree for the expression `-10--5`.

We define a class for the unary operator as follows.

```python
class UnaryOperator(AST):
    def __init__(self, op, expr):
        self.token = self.op = op
        self.expr = expr
```

The *Factor* method within the *Parser* object is updated to reflect the new expression grammar that includes the unary $+$ and unary $-$ operators.

```python
def factor(self):

    token = self.curr_token

    if token.type == PLUS:
        self.consume(PLUS)
        node = UnaryOperator(token, self.factor())
        return node

    if token.type == MINUS:
        self.consume(MINUS)
        node = UnaryOperator(token, self.factor())
        return node

    if token.type == INTEGER:
        self.consume(INTEGER)
        return Number(token)
    elif token.type == LPAREN:
        self.consume(LPAREN)
        node = self.expr()
        self.consume(RPAREN)
        return node
```

Finally, the *Evaluator* class is updated to include the *visit()* function tailored to the unary operator node.

```
class Evaluator(NodeVisitor):
    def __init__(self, parser):
        self.parser = parser

    def visit_BinaryOperator(self, node):
        if node.op.type == PLUS:
            return self.visit(node.left) + self.visit(node.right)
        if node.op.type == MINUS:
            return self.visit(node.left) - self.visit(node.right)
        if node.op.type == MUL:
            return self.visit(node.left) * self.visit(node.right)
        if node.op.type == DIV:
            return self.visit(node.left) / self.visit(node.right)

    def visit_UnaryOperator(self, node):
        if node.op.type == PLUS:
            return +self.visit(node.expr)
        elif node.op.type == MINUS:
            return -self.visit(node.expr)

    def visit_Number(self, node):
        return node.value

    def evaluate(self):
        tree = self.parser.parse()
        return self.visit(tree)
```

Assignment Statements

Consider the following code snippet.

```
a = 3;
b = 10;
c = a + (b - 4)/2;
```

The following grammar will correctly parse a program comprising entirely of assignment statements, of the form shown above. We add a production rule involving `identifier` to the list of rules under *Factor* to accommodate symbols occurring within expressions on the right-hand side of the assignment statement.

Program	→	*StatementList* EOF
statementList	→	*Statement*
	|	*Statement StatementList*
Statement	→	*AssignmentStmt*
AssignmentStmt	→	`identifier` ASSIGN *Expr* SEMI
Expr	→	*Term* PLUS *Expr*
	|	*Term* MINUS *Expr*
	|	*Term*
Term	→	*Factor* MULTIPLY *Term*
	|	*Factor* DIVIDE *Term*
	|	*Factor*
Factor	→	PLUS *Factor*
	|	MINUS *Factor*
	|	LPAREN *Expr* RPAREN
	|	`number`
	|	`identifier`

Tokens appearing in the foregoing grammar are:

```
(PLUS, `+'}
(MINUS, `-')
(MUL, `*'}
(DIV, `/')
(LPAREN, `(')
(RPAREN, `)')
(SEMI, `;')
(ASSIGN, `=')
```

The `EOF` symbol denotes end of file.

The abstract syntax tree (AST) generated by the parser for our code snippet is shown below.

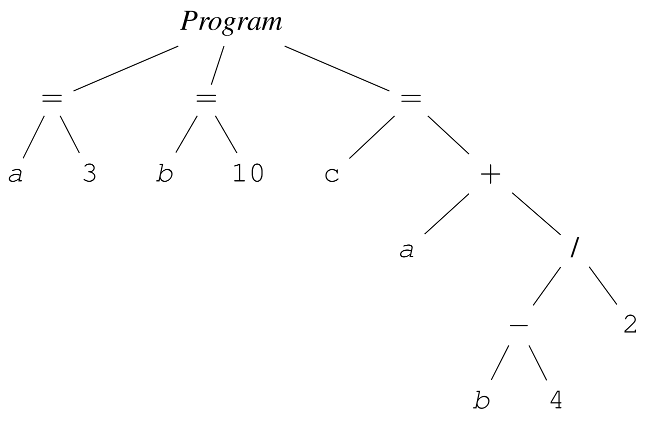

Fig. 6: The abstract syntax tree corresponding to the code snippet shown earlier.

In addition to the binary and unary operator nodes introduced in the earlier lecture, we define new nodes within the AST to represent the program, assignment statement, and the identifier.

```python
class Program(AST):
"""Node represents a program. It contains a list of assignment-statement
nodes as its children.
"""
    def __init__(self):
        self.children = []       # List of statements

class Assign(AST):
"""Node represents an assignment statement. The left edge stores a
variable node and right edge stores a node returned by the expr() method.
"""
    def __init__(self, left, op, right):
        self.left = left
        self.token = self.op = op
        self.right = right

class Identifier(AST):
"""Node represents an identifier."""
    def __init__(self, token):
        self.token = token
        self.value = token.value
```

Code Generation: Assume our example code is stored in a text file called *foo.txt*.

```
// foo.txt
a = 3;
b = 10;
c = a + (b - 4)/2;
```

Generating code from the AST is straightforward. We use a *symbol table* to track various variables within the source code. Visiting the children of the *Program* node from left to right, upon arriving at an assignment node, the variable occurring on the left-hand side of the assignment operator is added to the symbol table, if it has not previously, along with the corresponding RAM address location. Variables are stored in the local segment. Contents of the symbol table after the three nodes have been visited are as follows.

Variable name	Base	Offset
a	LCL	0
b	LCL	1
c	LCL	2

When a variable is encountered on the right-hand side of the assignment-operator node, we perform a lookup into the symbol table to locate the corresponding address. If the variable does not occur in the symbol table, it is undefined, leading to an error.

VM commands for the program can be generated by vising children of the *Program* node from left to right. When visiting the assignment node, code for the right-hand side (the expression) is generated first, followed by assignment to the identifier.

```
// foo.vm
// Initialize SP and LCL
set SP 256
set LCL 16
```

```
// a = 3
push constant 3
pop local 0

// b = 10
push constant 10
pop local 1

// c = a b 4 - 2 / +
push local 0
push local 1
push constant 4
sub
push constant 2
call div 2
add
pop local 2
```

In the above, we assume the availability of a function called *div* which performs integer division on two operands.