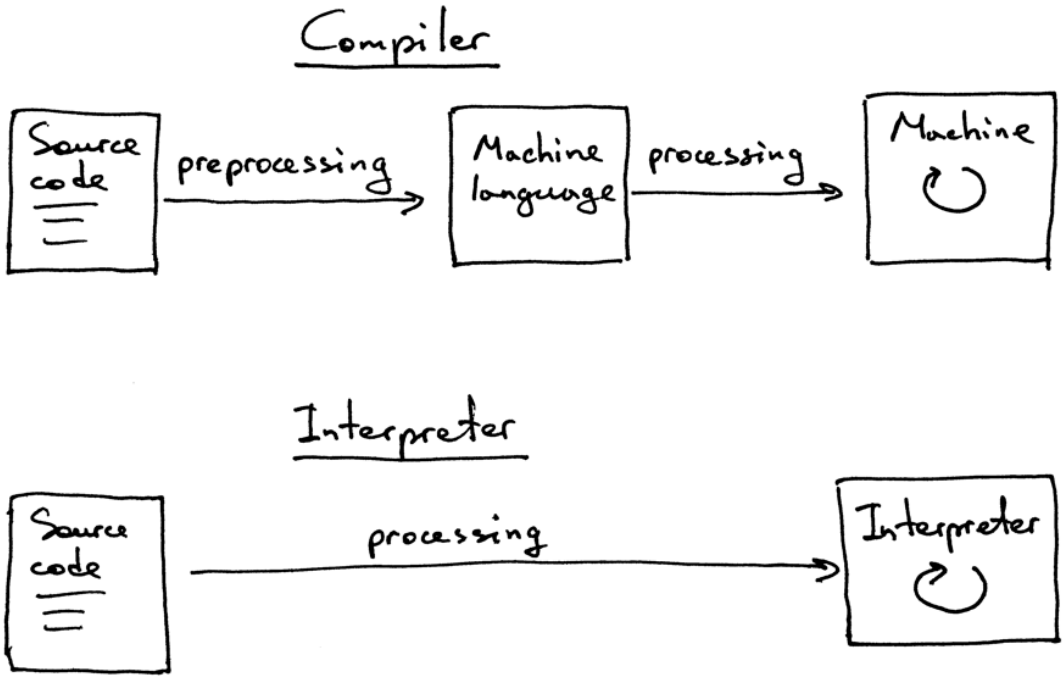
Interpreter Notes

Part-1:



In the compiler, through each stage we send the entire code. But In Interpreter each line we send all stages and then go to the next line.

A **token** is an object that has a type and a value. For example, for the string *"3"* the type of the token will be INTEGER and the corresponding value will be integer *3*.The process of breaking the input string into tokens is called **lexical analysis**.

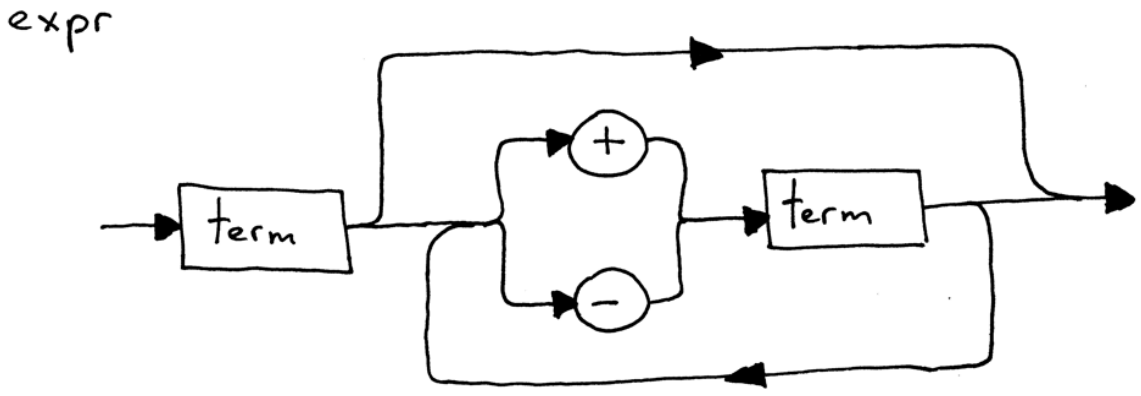
So, the first step your interpreter needs to do is read the input of characters and convert it into a stream of tokens. The part of the interpreter that does it is called a **lexical analyzer**, or **lexer** for short. You might also encounter other names for the same component, like **scanner** or **tokenizer**. They all mean the same: the part of your interpreter or compiler that turns the input of characters

into a stream of tokens.The method *get_next_token* of the *Interpreter* class is your lexical analyzer. Every time you call it, you get the next token created from the input of characters passed to the interpreter.

Part-2:

A **lexeme** is a sequence of characters that form a token.
The process of finding the structure in the stream of tokens, or put differently, the process of recognizing a phrase in the stream of tokens is called **parsing**. The part of an interpreter or compiler that performs that job is called a **parser**. So now you know that the *expr* method is the part of your interpreter where both **parsing** and **interpreting** happens - the *expr* method first tries to recognize (**parse**) the INTEGER -> PLUS -> INTEGER or the INTEGER -> MINUS -> INTEGER phrase in the stream of tokens and after it has successfully recognized (**parsed**) one of those phrases, the method interprets it and returns the result of either addition or subtraction of two integers to the caller.

Part-3:



A **syntax diagram** is a graphical representation of a programming language's syntax rules. Basically, a syntax diagram visually shows you which statements are allowed in your programming language and which are not.Syntax diagrams serve two main purposes:

- They graphically represent the specification (grammar) of a programming language.
- They can be used to help you write your parser - you can map a diagram to code by following simple rules.

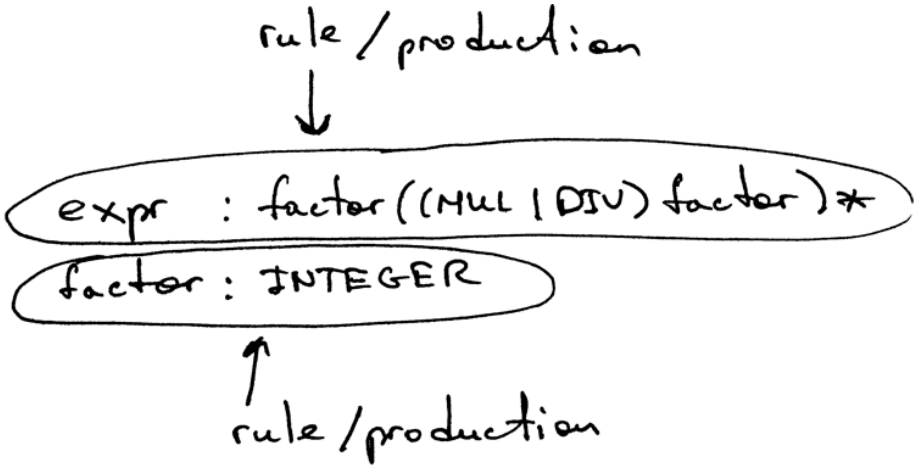
Parsing is also called **syntax analysis**, and the parser is also aptly called, you guessed it right, a **syntax analyzer**.

Part-4:

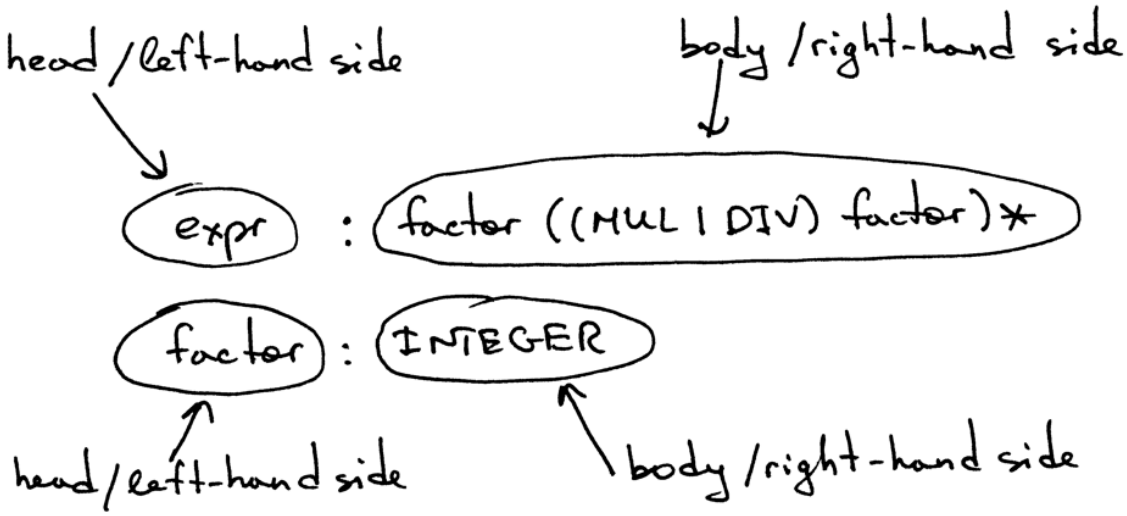
Here are a couple of reasons to use grammars:

1. A grammar specifies the syntax of a programming language in a concise manner. Unlike syntax diagrams, grammars are very compact. You will see me using grammars more and more in future articles.
2. A grammar can serve as great documentation.
3. A grammar is a good starting point even if you manually write your parser from scratch. Quite often you can just convert the grammar to code by following a set of simple rules.
4. There is a set of tools, called *parser generators*, which accept a grammar as an input and automatically generate a parser for you based on that grammar.

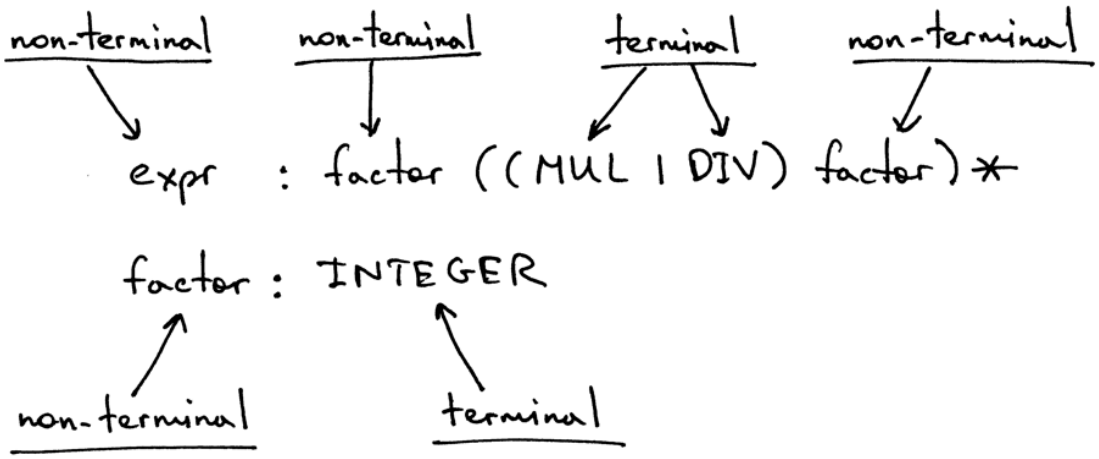
A grammar consists of a sequence of *rules*, also known as *productions*. There are two rules in our grammar:



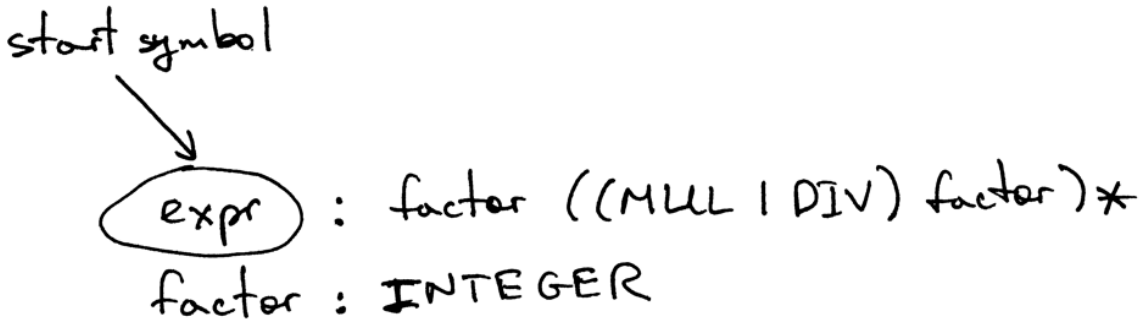
A rule consists of a *non-terminal*, called the ***head*** or ***left-hand side*** of the production, a colon, and a sequence of terminals and/or non-terminals, called the ***body*** or ***right-hand side*** of the production:



In the grammar I showed above, tokens like MUL, DIV, and INTEGER are called ***terminals*** and variables like *expr* and *factor* are called ***non-terminals***. Non-terminals usually consist of a sequence of terminals and/or non-terminals:



The non-terminal symbol on the left side of the first rule is called the ***start symbol***. In the case of our grammar, the start symbol is *expr*:



You can read the rule *expr* as “An *expr* can be a *factor* optionally followed by a *multiplication* or *division* operator followed by another *factor*, which in turn is optionally followed by a *multiplication* or *division* operator followed by another *factor* and so on and so forth.”

What is a *factor*? For the purpose of this article a *factor* is just an integer.

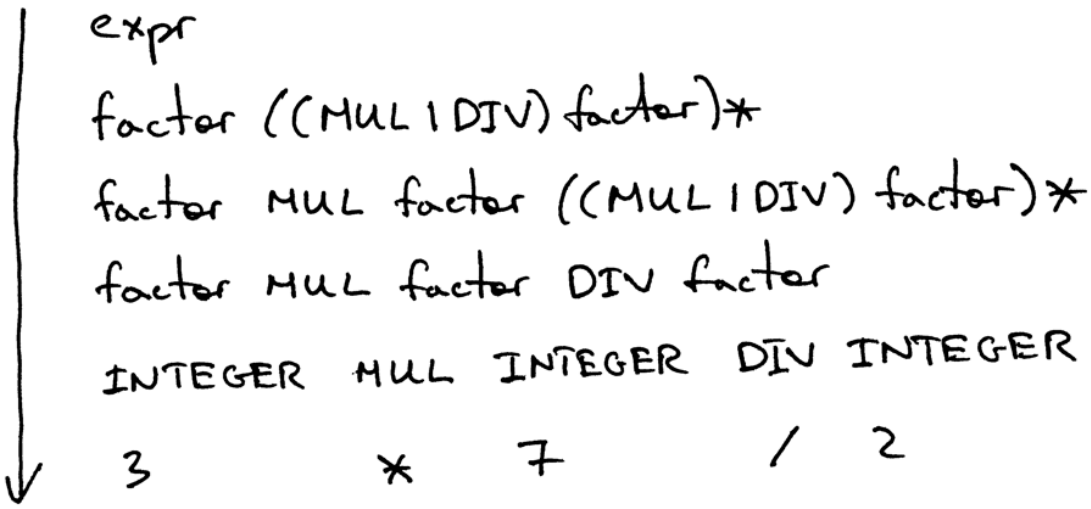
Let’s quickly go over the symbols used in the grammar and their meaning.

- **|** - Alternatives. A bar means “or”. So (MUL | DIV) means either MUL or DIV.
- **(…)** - An open and closing parentheses mean grouping of terminals and/or non-terminals as in (MUL | DIV).
- **(…)*** - Match contents within the group zero or more times.

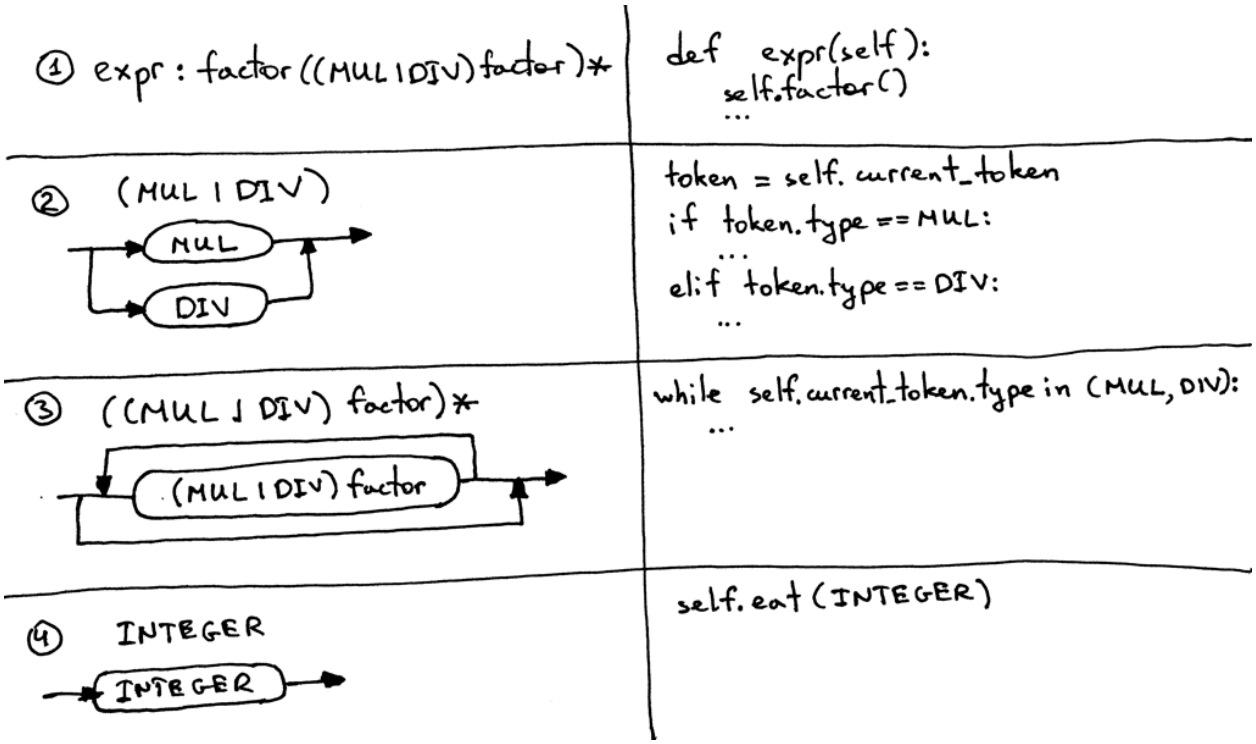
A grammar defines a *language* by explaining what sentences it can form. This is how you can *derive* an arithmetic expression using the grammar: first you begin with the start symbol *expr* and then repeatedly replace a non-terminal by the body of a rule for that non-terminal until you have generated a sentence consisting solely of terminals. Those sentences form a *language* defined by the grammar.

If the grammar cannot derive a certain arithmetic expression, then it doesn’t support that expression and the parser will generate a syntax error when it tries to recognize the expression.

this is how the grammar derives the expression *3 * 7 / 2*:

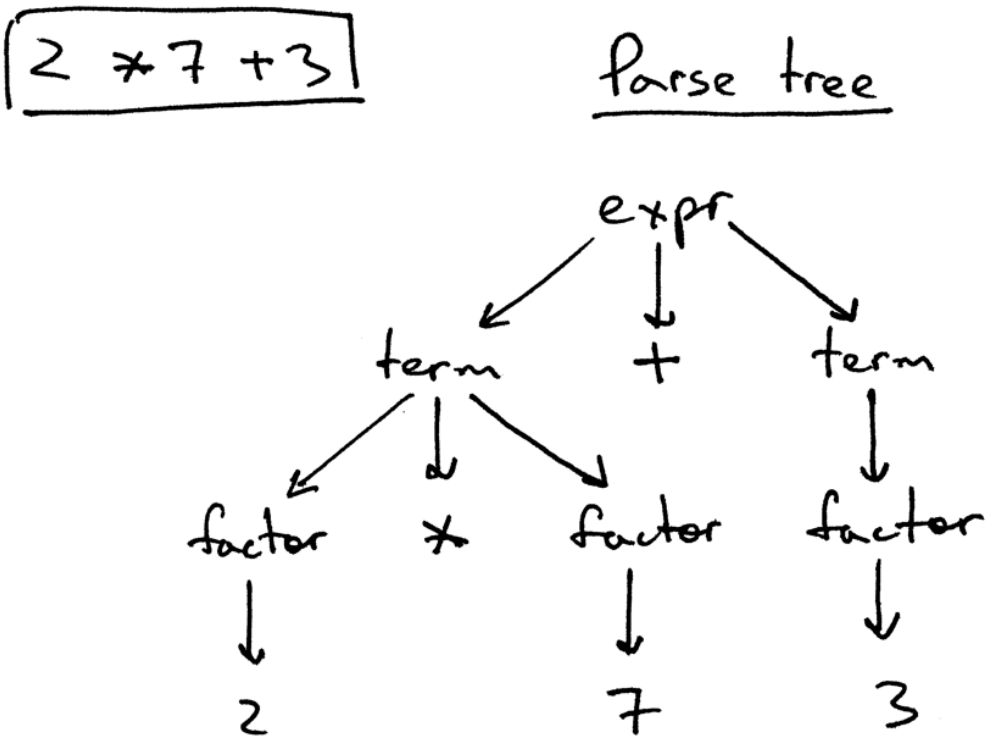


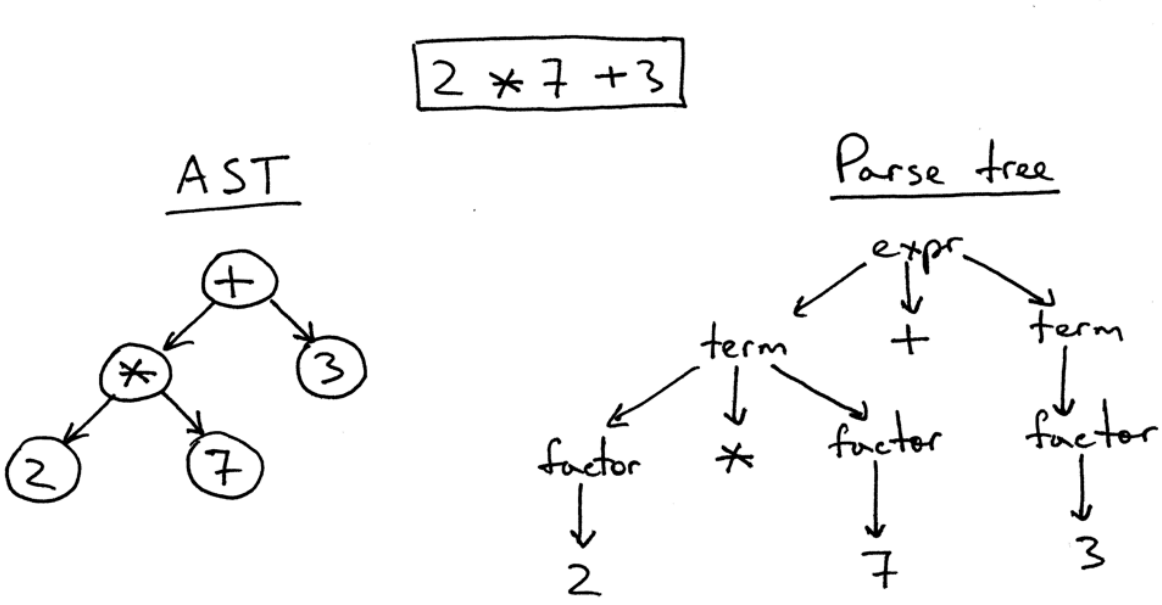
Visually the guidelines look like this:



Part-5:

Up until now, we had our interpreter and parser code mixed together and the interpreter would evaluate an expression as soon as the parser recognized a certain language construct like addition, subtraction, multiplication, or division. Such interpreters are called *syntax-directed interpreters*.
In order to analyze more complex programming language constructs, we need to build an *intermediate representation* (*IR*). Our parser will be responsible for building an *IR* and our interpreter will use it to interpret the input represented as the *IR*.It turns out that a tree is a very suitable data structure for an IR.
The IR we'll use throughout the series is called an *abstract-syntax tree* (*AST*).
We are not using parse trees, but we will study them.
A *parse-tree* (sometimes called a *concrete syntax tree*) is a tree that represents the syntactic structure of a language construct according to our grammar definition. It basically shows how your parser recognizes the language construct or, in other words, it shows how the start symbol of your grammar derives a certain string in the programming language.



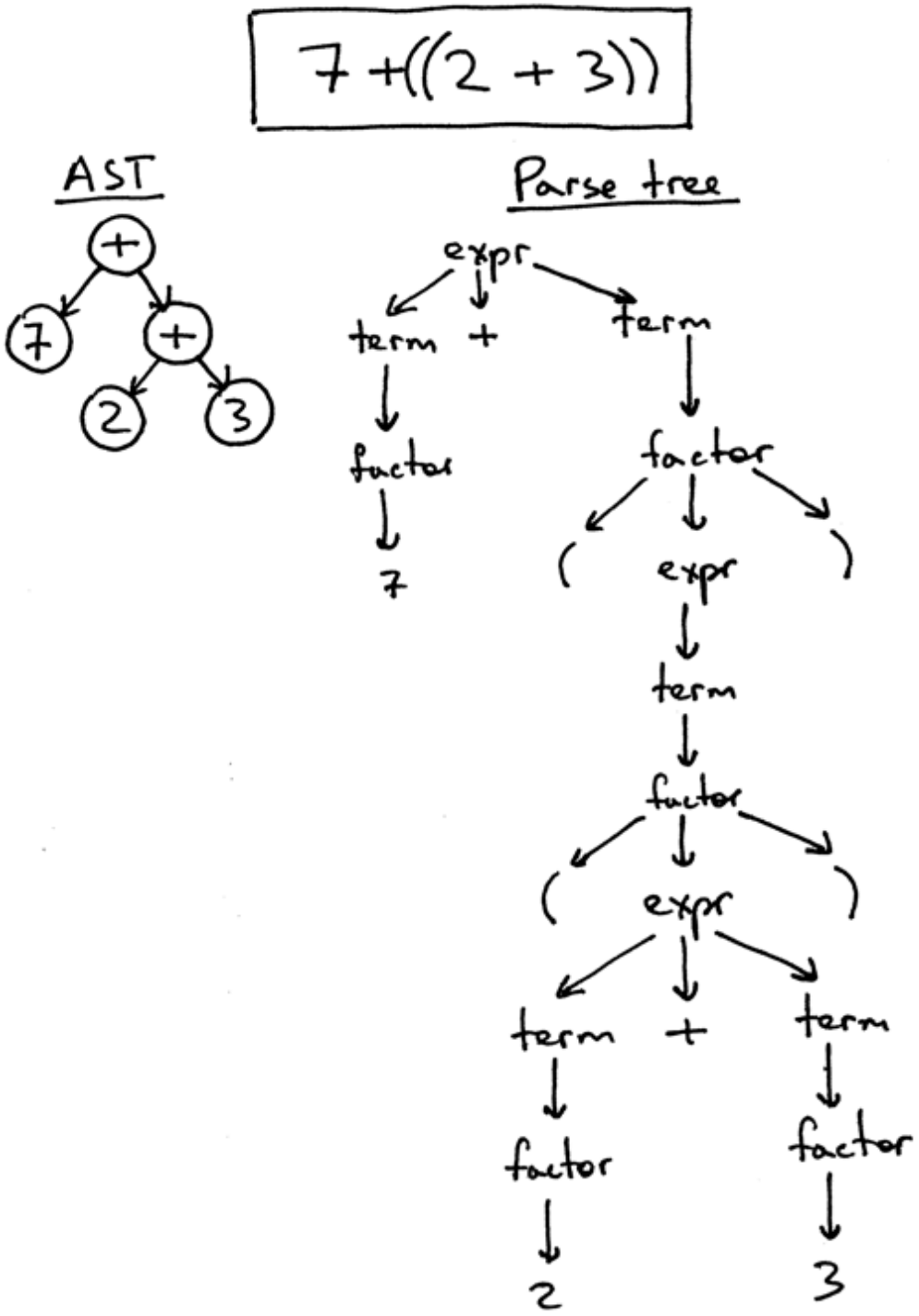


As you can see from the picture above, the AST captures the essence of the input while being smaller.

Here are the main differences between ASTs and Parse trees:

- ASTs uses operators/operations as root and interior nodes and it uses operands as their children.
- ASTs do not use interior nodes to represent a grammar rule, unlike the parse tree does.
- ASTs don't represent every detail from the real syntax (that's why they're called *abstract*) - no rule nodes and no parentheses, for example.
- ASTs are dense compared to a parse tree for the same language construct.

An *abstract syntax tree* (*AST*) is a tree that represents the abstract syntactic structure of a language construct where each interior node and the root node represents an operator, and the children of the node represent the operands of that operator.
In order to encode the operator precedence in AST, that is, to represent that "X happens before Y" you just need to put X lower in the tree than Y.



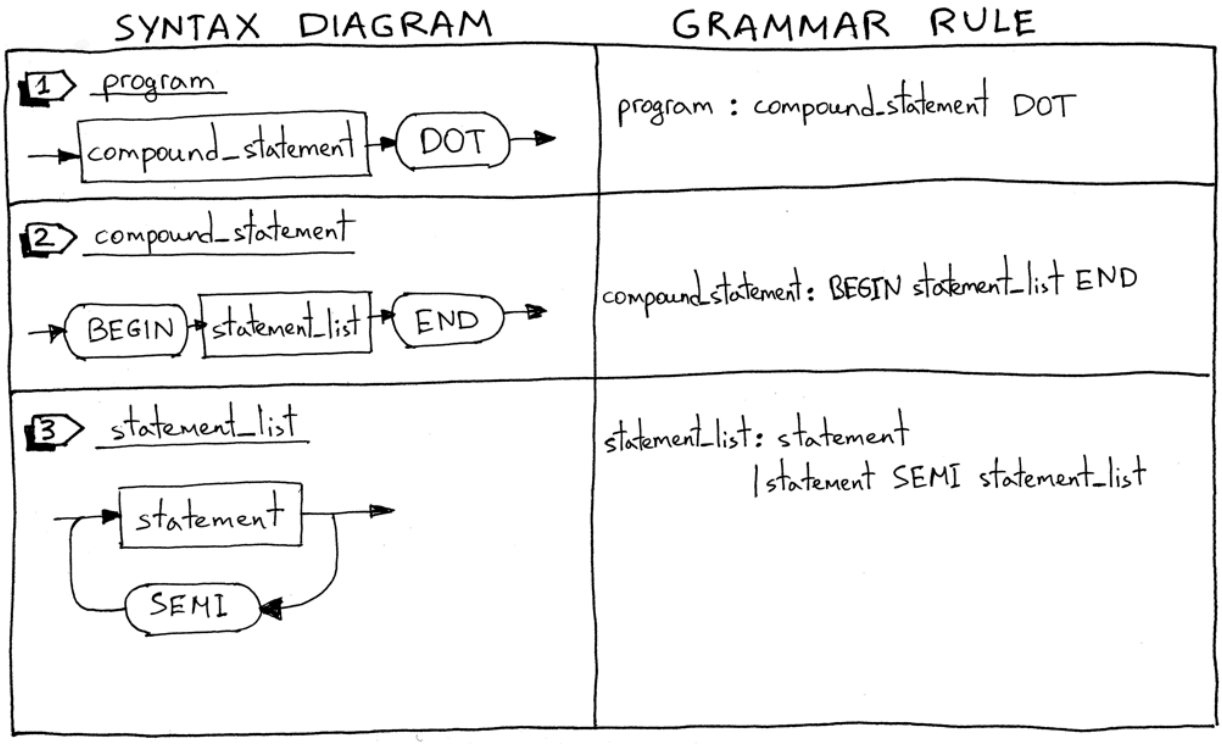
The current interface between the lexer, parser, and the interpreter now looks like this:You can read that as “The parser gets tokens from the lexer and then returns the generated AST for the interpreter to traverse and interpret the input.”

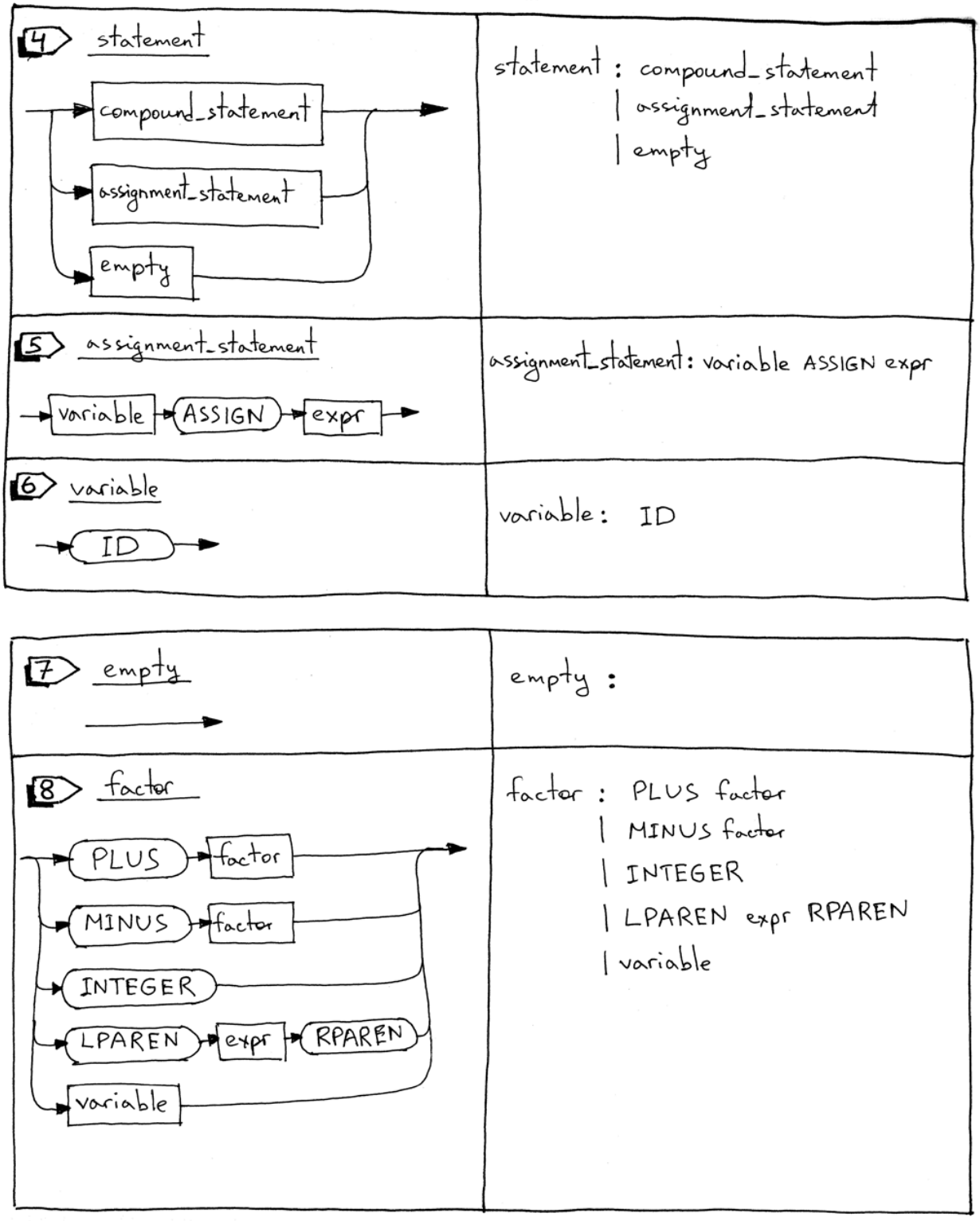
A *unary operator* is an operator that operates on one *operand* only.

Here are the rules for unary plus and unary minus operators:

- 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 /

Part-6:



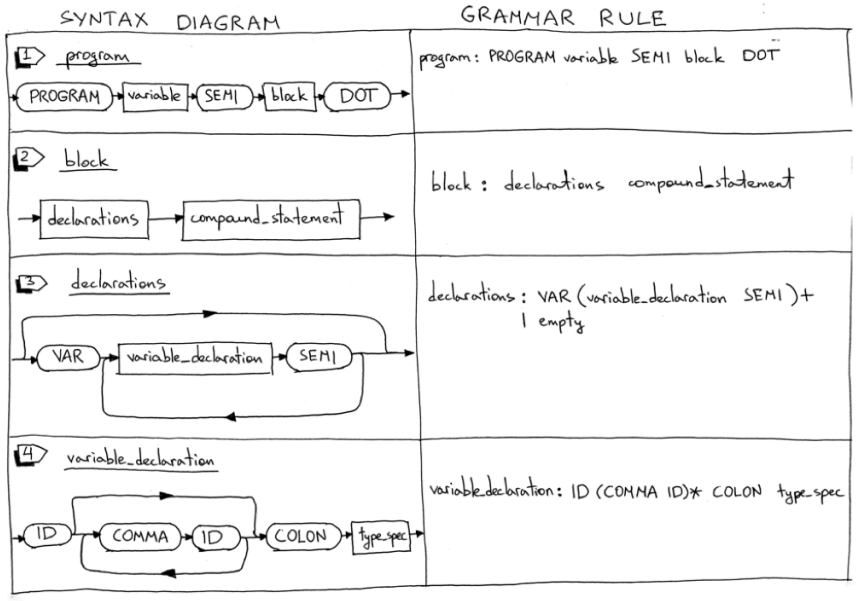


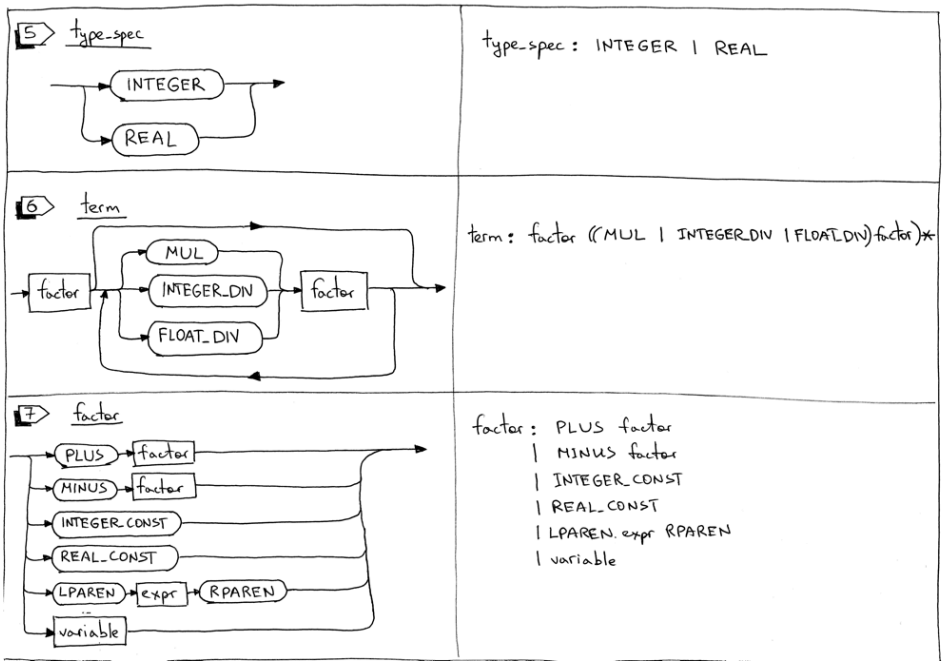
A Pascal ***program*** consists of a *compound statement* that ends with a dot. A ***compound statement*** is a block marked with BEGIN and END that can contain a list (possibly empty) of statements including other compound

statements. Every statement inside the compound statement, except for the last one, must terminate with a semicolon. The last statement in the block may or may not have a terminating semicolon. A ***statement list*** is a list of zero or more statements inside a compound statement.A ***statement*** can be a *compound statement*, an *assignment statement*, or it can be an *empty* statement.An ***assignment statement*** is a variable followed by an ASSIGN token (two characters, ':' and '=') followed by an expression.A ***variable*** is an identifier. We'll use the ID token for variables. The value of the token will be a variable's name like 'a', 'number', and so on.An ***empty*** statement represents a grammar rule with no further productions. We use the *empty_statement* grammar rule to indicate the end of the *statement_list* in the parser and also to allow for empty compound statements as in 'BEGIN END'.The ***factor*** rule is updated to handle variables.

A ***symbol table*** is an abstract data type (**ADT**) for tracking various symbols in source code. The only symbol category we have right now is variables and we use the Python dictionary to implement the symbol table ADT. For now I'll just say that the way the symbol table is used in this article is pretty "hacky": it's not a separate class with special methods but a simple Python dictionary and it also does double duty as a memory space.

Part-7:



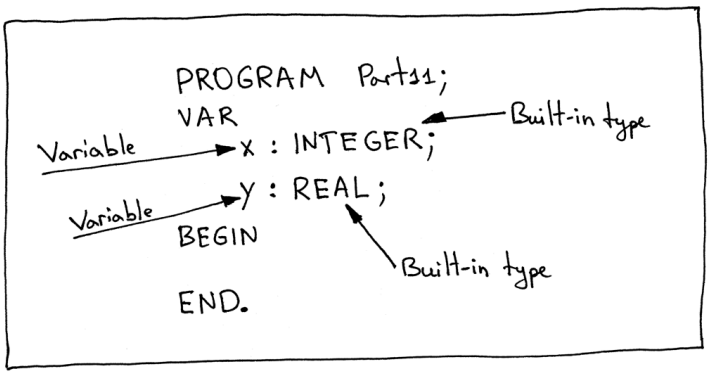


1. The ***program*** definition grammar rule is updated to include the ***PROGRAM*** reserved keyword, the program name, and a block that ends with a dot.
2. The ***block*** rule combines a *declarations* rule and a *compound_statement* rule. We'll also use the rule later in the series when we add procedure declarations.
3. Pascal declarations have several parts and each part is optional. In this article, we'll cover the variable declaration part only. The ***declarations*** rule has either a variable declaration sub-rule or it's empty.
4. Pascal is a statically typed language, which means that every variable needs a variable declaration that explicitly specifies its type. In Pascal, variables must be declared before they are used. This is achieved by declaring variables in the program variable declaration section using the ***VAR*** reserved keyword.
5. The ***type_spec*** rule is for handling *INTEGER* and *REAL* types and is used in variable declarations.
6. The ***term*** rule is updated to use the ***DIV*** keyword for integer division and a forward slash / for float division.
7. The ***factor*** rule is updated to handle both integer and real (float) constants. I also removed the INTEGER sub-rule because the constants will be represented by ***INTEGER_CONST*** and ***REAL_CONST*** tokens and the ***INTEGER*** token will be used to represent the integer type.

Part-8:

For our purposes, we'll informally define a symbol as an identifier of some program entity like a variable, subroutine, or built-in type. For symbols to be useful they need to have at least the following information about the program entities they identify:

- Name (for example, 'x', 'y', 'number')
- Category (Is it a variable, subroutine, or built-in type?)
- Type (INTEGER, REAL)



Here are some of the reasons:

- To make sure that when we assign a value to a variable the types are correct (type checking)
- To make sure that a variable is declared before it is used

A ***symbol table*** is an abstract data type (***ADT***) for tracking various symbols in source code.

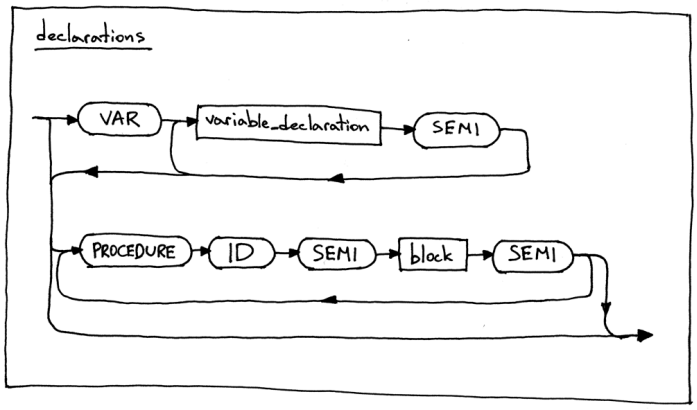
Symbol Table

- key -	- value -
INTEGER	BuiltinTypeSymbol instance
x	VarSymbol instance <x:INTEGER>
REAL	BuiltinTypeSymbol instance
y	VarSymbol instance <y:REAL>

What is a ***procedure declaration***? A ***procedure declaration*** is a language construct that defines an identifier (a procedure name) and associates it with a block of Pascal code.

```
declarations : VAR (variable_declaration SEMI)+
             | (PROCEDURE ID SEMI block SEMI)*
             | empty
```

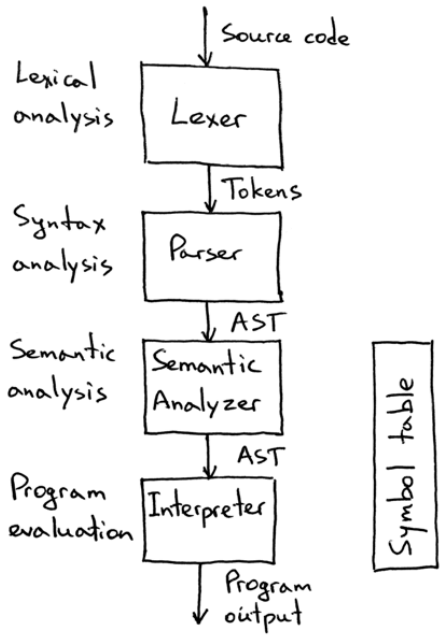
The procedure declaration sub-rule consists of the reserved keyword **PROCEDURE** followed by an identifier (a procedure name), followed by a semicolon, which in turn is followed by a *block* rule, which is terminated by a semicolon.Here is the updated syntax diagram for the *declarations* rule:



Part-9:

While our Pascal program can be grammatically correct and the parser can successfully build an *abstract syntax tree*, the program still can contain some pretty serious errors. To catch those errors we need to use the *abstract syntax tree* and the information from the *symbol table*.When the parser has finished building the AST, we know that the program is grammatically correct; that is, that its syntax is correct according to our grammar rules and now we can

separately focus on checking for errors that require additional context and information that the parser did not have at the time of building the AST.



- **Semantic analysis** checks whether a program *makes sense*, after syntax is already verified by the parser.
- It uses the **AST and symbol table** to detect errors that grammar alone cannot catch.
- Main static checks done before execution:
  - Variables must be declared before use
  - No duplicate declarations
  - Type correctness (handled separately later)
- **Symbols** represent program entities (variables, types, procedures) with name, category, and type.
- **BuiltinTypeSymbol** represents built-in types like `INTEGER`, `REAL`.
- **VarSymbol** represents declared variables and stores their type.
- **Symbol Table** is a dictionary mapping names → symbol objects.
- Semantic analyzer **walks the AST (visitor pattern)** to:
  - Insert symbols on declarations
  - Lookup symbols on variable use (name resolution)
- **Name resolution** = mapping a variable reference to its declaration in the symbol table.
- If lookup fails → **undeclared variable semantic error**.

- If duplicate found during insertion → **duplicate identifier semantic error**.
- Built-in types are inserted into symbol table **before analysis starts**.
- Semantic analysis happens **before interpretation**, and interpreter runs only if semantics are valid.

Part-10:

A scope is a textual region of a program in which a name such as a variable, procedure, or type can be used and is visible. In Pascal, scopes are introduced by keywords like `PROGRAM` and `PROCEDURE`. The global scope begins at the `PROGRAM` keyword and ends at the final `END.` of the program. Every procedure declaration introduces a new scope nested inside the scope in which the procedure is declared. The scope of a declaration is the region of the program where that declared name can be legally referenced.

Pascal is a lexically or statically scoped language, meaning that name resolution depends only on the program's textual structure and not on the order of execution. This allows the compiler or interpreter to determine which declaration a name refers to simply by analyzing the source code. Lexical scoping also allows programmers to reuse variable names in different scopes, because each scope provides its own isolated namespace. An inner scope may redeclare a name that exists in an outer scope, which causes the outer declaration to be hidden within the inner scope.

To implement scopes in a compiler or interpreter, each scope is represented by a scoped symbol table. A scoped symbol table stores mappings from names to symbols, where symbols represent program entities such as variables, procedures, and built-in types. In addition to storing symbols, each scoped symbol table also keeps track of its scope name, scope level, and a reference to its enclosing scope. The enclosing scope link is used to represent nesting relationships between scopes, forming a structure known as a scope tree.

Nested scopes are implemented by chaining scoped symbol tables together using the enclosing scope reference. When a new scope is entered, such as

when processing a program or procedure declaration, a new scoped symbol table is created and its enclosing scope is set to the previously active scope. The current scope pointer is then updated to point to this new table. When leaving the scope, the current scope pointer is restored to the enclosing scope. This behavior is equivalent to pushing and popping scopes on a stack and is essential for building the correct scope tree when multiple nested procedures exist.

Semantic analysis is the compiler phase that checks whether a program is meaningful according to the language rules, beyond just being grammatically correct. During semantic analysis, the compiler walks the Abstract Syntax Tree and builds scoped symbol tables while performing checks such as verifying that variables are declared before use and that no duplicate declarations exist within the same scope. Built-in types such as INTEGER and REAL are inserted into the global scope before analyzing user declarations so that type references can be resolved correctly.

Name resolution is the process of matching a variable reference to its corresponding declaration. In the presence of nested scopes, name resolution follows the most closely nested scope rule. The semantic analyzer first searches for the name in the current scope. If the name is not found, the search continues in the enclosing scope, and this process repeats until the name is found or the global scope is reached. If no declaration is found, a semantic error is raised. This lookup mechanism is implemented by recursively or iteratively following the enclosing scope links of scoped symbol tables.

Procedure declarations require additional semantic handling because they introduce both a new scope and a new symbol. When a procedure is declared, a procedure symbol is inserted into the enclosing scope, and a new scoped symbol table is created for the procedure body. Formal parameters of the procedure are treated as variable symbols and are inserted into the procedure’s local scope. These parameter symbols are also stored inside the procedure symbol for reference. Local variables declared inside the procedure are added to the same procedure scope.

To correctly detect duplicate identifiers, semantic analysis must ensure that a new declaration does not already exist in the current scope. Therefore, symbol lookup for duplicate checking must be limited to the current scope only and must not search enclosing scopes. This allows valid shadowing of outer variables while still preventing multiple declarations of the same name in a single scope.

A source-to-source compiler is a translator that converts a program into another version of the same language, usually with additional annotations. In the context of learning scopes and name resolution, a source-to-source compiler can rewrite programs by adding scope-level subscripts to variable names and explicit type information to variable references. This makes scope boundaries and name bindings visually clear and helps in understanding how nested scopes and name resolution operate. Such a compiler can be implemented by extending the semantic analyzer to generate annotated source code while traversing the AST and using symbol table information.

Overall, implementing scopes using chained scoped symbol tables allows a compiler or interpreter to correctly model nested program structures, perform accurate name resolution, and enforce semantic correctness before execution. This design mirrors how real-world compilers manage environments and is fundamental to understanding programming language implementation.

Part-11:

A function is a named block of code that takes zero or more parameters as input, executes a sequence of statements, and returns a single value of a specified type. The syntax of a function declaration in Pascal is `FUNCTION name(parameters) : return_type;`. Unlike procedures, functions return a value and can be used directly inside expressions. The return value of a function is produced by assigning a value to the function's own name inside the function body.

Formal parameters are the parameters declared in the function definition and are listed in the function header along with their types. Multiple parameters of

the same type can be grouped together, such as `(a, b, c : INTEGER)`, while parameters of different types are separated by semicolons, for example `(x : INTEGER; y : REAL)`. Actual parameters are the arguments provided when the function is called. These are evaluated as expressions before the function execution begins, and their number must exactly match the number of formal parameters. Parameters are typically passed by value, meaning that the function receives a copy of the argument values, not references to the original variables.

Each function creates a new scope, usually at a deeper scope level than the enclosing scope. The function name itself is defined as a variable inside its own scope so that it can store the return value. Formal parameters become local variables within the function scope, and additional local variables can be declared using the `VAR` section inside the function. A function can access variables from enclosing scopes according to lexical scoping rules, but variables declared in the inner scope hide variables with the same name in outer scopes.

Function calls can appear inside expressions, for example `result := Add(10, 20) + 5`. The function call is evaluated to its return value, which then participates in the expression like any other value. Function calls can also be nested, such as `Func1(Func2(x))`. The execution of a function call follows a specific sequence: first, actual parameter expressions are evaluated from left to right; then, their values are bound to the corresponding formal parameters; next, the function body is executed; after that, the return value is obtained from the function name; finally, the function's local scope is discarded and the return value replaces the function call in the original expression.

During semantic analysis, several checks are performed related to functions. The analyzer verifies that functions are declared before they are used, that no duplicate function names exist in the same scope, and that parameter names are not duplicated within a function. It also checks that the number of arguments in a function call matches the number of formal parameters and that the function's return type is properly declared and valid. These checks ensure that function usage is semantically correct before execution begins.

In the symbol table, each function is represented using a FunctionSymbol, which stores the function's name, its list of parameters, and its return type. The function symbol is inserted into the enclosing scope at the time of declaration so that it can be resolved during function calls. When a function call is encountered, the semantic analyzer looks up the corresponding FunctionSymbol to validate the call and bind arguments to parameters correctly.

Part-12:

The **call stack** is a runtime data structure used to manage the execution of function and procedure calls. It keeps track of active subprograms, their local variables, parameters, and return addresses. Every time a function or procedure is called, a new block of memory is created and pushed onto the call stack, and when the function finishes execution, that block is removed from the stack. This mechanism allows programs to support nested and recursive function calls.

An **activation record** (also called a stack frame) is the data structure that represents one function or procedure call on the call stack. Each activation record stores all the information needed to execute and later return from that call. In other words, the call stack is a stack of activation records, where each record corresponds to one active function or procedure invocation.

An activation record typically contains several components. It stores the **name of the subprogram** being executed and the **return address**, which indicates where execution should continue after the function finishes. It also stores the **actual parameter values** passed to the function and the **local variables** declared inside the function. In many implementations, it also contains a reference to the **enclosing scope or dynamic link**, which helps with accessing non-local variables in nested scopes. Additionally, it stores space for the **return value** in the case of functions.

When a function is called, the program follows a specific sequence. First, actual arguments are evaluated. Then a new activation record is created

containing parameter values, local variables, and bookkeeping information. This activation record is pushed onto the call stack, and control transfers to the function body. While the function executes, all variable accesses refer to the data stored in its activation record. When the function finishes, its activation record is popped from the stack, and control returns to the caller using the stored return address. The return value is then passed back to the calling expression.

The call stack naturally supports **nested and recursive calls**. In recursion, each recursive call creates a new activation record even though the same function is called again. Each record has its own separate copy of parameters and local variables, which prevents interference between different recursive levels. When recursive calls return, activation records are popped one by one in reverse order of calls, following the Last-In-First-Out (LIFO) property of stacks.

The call stack is different from the symbol table. Symbol tables are used during semantic analysis and compilation to resolve names and scopes, while activation records are used during execution to store actual runtime values. Symbol tables exist at compile-time or analysis-time, whereas activation records exist only at runtime. However, symbol table information is used to decide what variables and parameters should be stored in each activation record.

In interpreters, the call stack is often explicitly implemented as a Python list or stack structure that holds activation record objects. Each activation record is usually implemented as a dictionary mapping variable names to their current values, along with metadata such as scope name and nesting level. This makes it easy for the interpreter to manage variable environments and return values during execution.

Overall, the call stack and activation records together provide the mechanism that enables structured program execution, proper handling of function calls, local variables, recursion, and correct return behavior in programming languages.

Part-13:

Conditionals are control flow structures that enable programs to make decisions and execute different code paths based on conditions evaluated at runtime. They are fundamental to programming because they allow software to react dynamically to data. Without conditionals, programs would be limited to executing instructions in a fixed, linear order with no ability to branch or adapt.

At the core of conditional logic are boolean expressions, which evaluate to either true or false. The implementation supports three main categories of boolean operations. Comparison operators such as =, <>, <, >, <=, and >= compare two arithmetic values and return a boolean result based on their relationship. Logical operators combine boolean values: AND returns true only when both operands are true, OR returns true when at least one operand is true, and NOT negates a boolean value. Operator precedence plays a critical role in correct evaluation. Comparison operators bind more tightly than logical operators, AND has higher precedence than OR, and NOT has the highest precedence among boolean operators. This precedence allows expressions like `a < b AND c = d OR e > f` to be interpreted as `((a < b) AND (c = d)) OR (e > f)`.

The IF statement follows the structure `IF condition THEN statement [ELSE statement] END`, where the ELSE clause is optional. This allows for both simple conditional execution and full if-else branching. The statements within the THEN and ELSE branches can be single assignments, compound BEGIN–END blocks, or even nested IF statements, giving flexibility in how programs are structured.

Implementing conditionals required careful grammar design to ensure boolean operator precedence was handled correctly. A hierarchy of parsing functions was introduced: `boolean_expression` handles OR operators with the lowest precedence, `boolean_term` handles AND operators with higher precedence, and `boolean_factor` handles NOT operations and comparisons with the highest precedence. Comparisons themselves are

handled by a dedicated rule that delegates evaluation of operands to arithmetic expressions. A subtle challenge involved managing parentheses, since arithmetic expressions already use parentheses for grouping. The solution was to let arithmetic parentheses be handled entirely by the arithmetic grammar rules, avoiding conflicts with boolean expression grouping.

When the interpreter encounters an IF statement, it first evaluates the boolean expression to determine a true or false result. If the condition is true, it executes the THEN branch and skips the ELSE branch. If the condition is false and an ELSE branch exists, it executes the ELSE branch. If the condition is false and no ELSE branch exists, it skips both branches and continues execution. This behavior is implemented using the visitor pattern, where the `visit_IfStatement` method evaluates the condition node and then conditionally visits either the then-branch or else-branch node.

Comparison operations are handled by evaluating both operands as arithmetic expressions and applying Python's comparison operators to produce a boolean result. Boolean operations evaluate their operands as boolean expressions and apply logical operations using Python's `and`, `or`, and `not` operators.

From a semantic analysis perspective, several validations are required. Variables used in conditions must be declared, and function calls within conditions must have correct signatures. For recursive functions, special care is needed to distinguish between the function symbol itself and the variable representing its return value, which may share the same name within the function's scope. This issue was resolved by modifying function call handling to search up the scope chain specifically for function symbols while skipping variable symbols with the same name.

Conditionals are essential for enabling recursion. Without IF statements, recursive functions would have no termination condition and would recurse infinitely. With conditionals, base cases can be defined to stop recursion. This makes implementations like factorial possible, where a condition such as `n <= 1` returns a fixed value, and recursive calls occur only when the condition is false.

Adding conditionals had a significant impact on the grammar. New tokens were introduced, including IF, THEN, ELSE, AND, OR, NOT, and the comparison operators. A new statement type was added to the statement rule, along with three new production rules to enforce boolean expression precedence. This illustrates how introducing control flow structures increases language complexity while greatly enhancing expressiveness.

Part-14: