Compiler Design - Lexical Analysis

Tutorial One – tutorials point

Special Symbols

A typical high-level language contains the following symbols:-

Arithmetic Symbols	Addition(+), Subtraction(-), Modulo(%), Multiplication(*), Division(/)
Punctuation	Comma(,), Semicolon(;), Dot(.), Arrow(->)
Assignment	=
Special Assignment	+=, /=, *=, -=
Comparison	==, !=, <, <=, >, >=
Preprocessor	#
Location Specifier	&
Logical	&, &&, |, ||, !
Shift Operator	>>, >>>, <<, <<<

Longest Match Rule

When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.

For example:

```
int intvalue;
```

While scanning both lexemes till 'int', the lexical analyzer cannot determine whether it is a keyword *int* or the initials of identifier int value.

The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.

The lexical analyzer also follows **rule priority** where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

Tutorial 2 : https://www.geeksforgeeks.org/compiler-lexical-analysis/

Compiler Design | Lexical Analysis

Lexical Analysis is the first phase of compiler also known as scanner. It converts the High level input program into a sequence of **Tokens**.

- Lexical Analysis can be implemented with the Deterministic finite Automata.
- The output is a sequence of tokens that is sent to the parser for syntax analysis

Lexeme: The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. eg- "float", "abs_zero_Kelvin", "=", "-", "273", ";" .

How Lexical Analyzer functions

1. Tokenization .i.e Dividing the program into valid tokens.
2. Remove white space characters.
3. Remove comments.
4. It also provides help in generating error message by providing row number and column number.

he lexical analyzer identifies the error with the help of automation machine and the grammar of the given language on which it is based like C , C++ and gives row number and column number of the error.

Exercise 1:
Count number of tokens :

```
int main()
{
  int a = 10, b = 20;
  printf("sum is :%d",a+b);
  return 0;
}
Answer: Total number of token: 27.
```

Tutorial 3 Medium https://hackernoon.com/lexical-analysis-861b8bfe4cb0

3 Tutorials
T1 : Let's Build a Programming Language
https://hackernoon.com/lets-build-a-programming-language-2612349105c6

Ring :

1- Data types `Int`, `Double`, `String` **and** `Bool`.
2- Comments // and /* */
3- Declaring variables int m = 10 ;
4- Declaring multiple var at once int a =10, b =15, c= …….. ;

5- **Conditions**

```
if (<condition>) {
    <do something>
} else {
    <do something>
}
```

The condition must evaluate to a value of type `Bool`

6- Looping :
A -**While**

```
while (<condition>) {
    <do something>
}
```

b – for loop

7-Defining functions
What I want is
Func returnType Name(type par1, type par2,……){
Expressions;
Return var ;
}

Functions are defined using the `func` keyword.

```
func sum(a: Int, b: Int): Int = {
    a + b
}
```

Function parameters in Blink are separated by commas `,` and enclosed in parenthesis `(a: Int, b: Int)`. Each parameter is defined by its name followed by a colon `:`, followed by its type `a: Int`.

7- Return Type

a- Return type. The return type is defined by adding a colon `:` followed by the type of the value returned by the function after the parameter list. If the function does not return a value, the return type must be `Unit`.

b-
```
func greet(): Unit = {
    Console.println("Hello, Blink!")
}
```

c- Function body. After the return type, comes the equal `=` operator and then the body of the function enclosed in curly braces. There is **no return keyword**in Blink, the value of the last *expression* in the body of the function **is** the value returned by the function.

d- If the body is made of only one *expression*, the curly braces can be omitted. Our first example could be rewritten like this:

e-
```
func sum(a: Int, b: Int) = a + b
```

#NOTE we can desing classes but we will ignore OOP

8- Defining classes :

a. Classes are defined using the `class` keyword.

b.
```
class Person {
}
```

c. This defines a simple (albeit useless) class `Person`. Objects of the class `Person` can now be created using the `new` keyword.

d.
```
let p = new Person() in { }
```

e.Properties:

A class can have one or more properties declared with the varkeyword.

```
class Person {
    var firstname: String = "Klaus"
    var lastname: String
    var age: Int
}
```

A property can be optionally initialized at its declaration. If a property is initialized, the initialization *expression* will be evaluated when the object is being created.

Properties in Blink are private. *They can not be made public.* Getters and setters (which are just functions) can be created to access properties outside of the class.

f-Functions

A class can have functions

```
class Person {
   var firstname: String = "Klaus"
   var lastname: String
   var age: Int
   func firstname(): String = {
       firstname
   }
   func setFirstname(name: String) = {
       firstname = name
   }
}
```

Functions are public by default in Blink. However, it's possible to make a function private by adding the private modifier before the func keyword.

```
    private func age(): Int = // ...
```

9-Constructor

A class in Blink has *one and only one* constructor. By default, a class in Blink has a default constructor which takes no parameter. An explicit custom constructor can be defined by adding a list of parameters enclosed in parenthesis to the name of the class.

```
class Person(firstname: String, lastname: String) {
    func firstname(): String = firstname
    func setFirstname(name: String) = firstname = name
    func lastname(): String = lastname
    func setLastname(name: String) = lastname = name
}
```

10-Inheritance

Inheritance in Blink is expressed with the `extends` keyword.

```
class Employee extends Person {
}
```

If the class being inherited (the superclass) has an explicit constructor, the inheriting class must honor the superclass's constructor by passing the arguments necessary to create an object of the superclass.

```
class Employee(firstname: String, lastname: String,
company: String) extends Person(firstname, lastname) {
}
```

11-**Overriding functions**

The `override` modifier is used to override a function in the superclass.

```
class Person(firstname: String, lastname: String) {
    override func toString(): String = firstname + " " +
lastname
}
```

in the above example, we're overriding the `toString` available in the `Object`class. All classes inherit from `Object` in Blink

12- *Defining operators*

The difference between an *expression* and a statement is that an *expression* always evaluates to a *value* while a statement just performs some action.

NOTE :

How to handle print statements

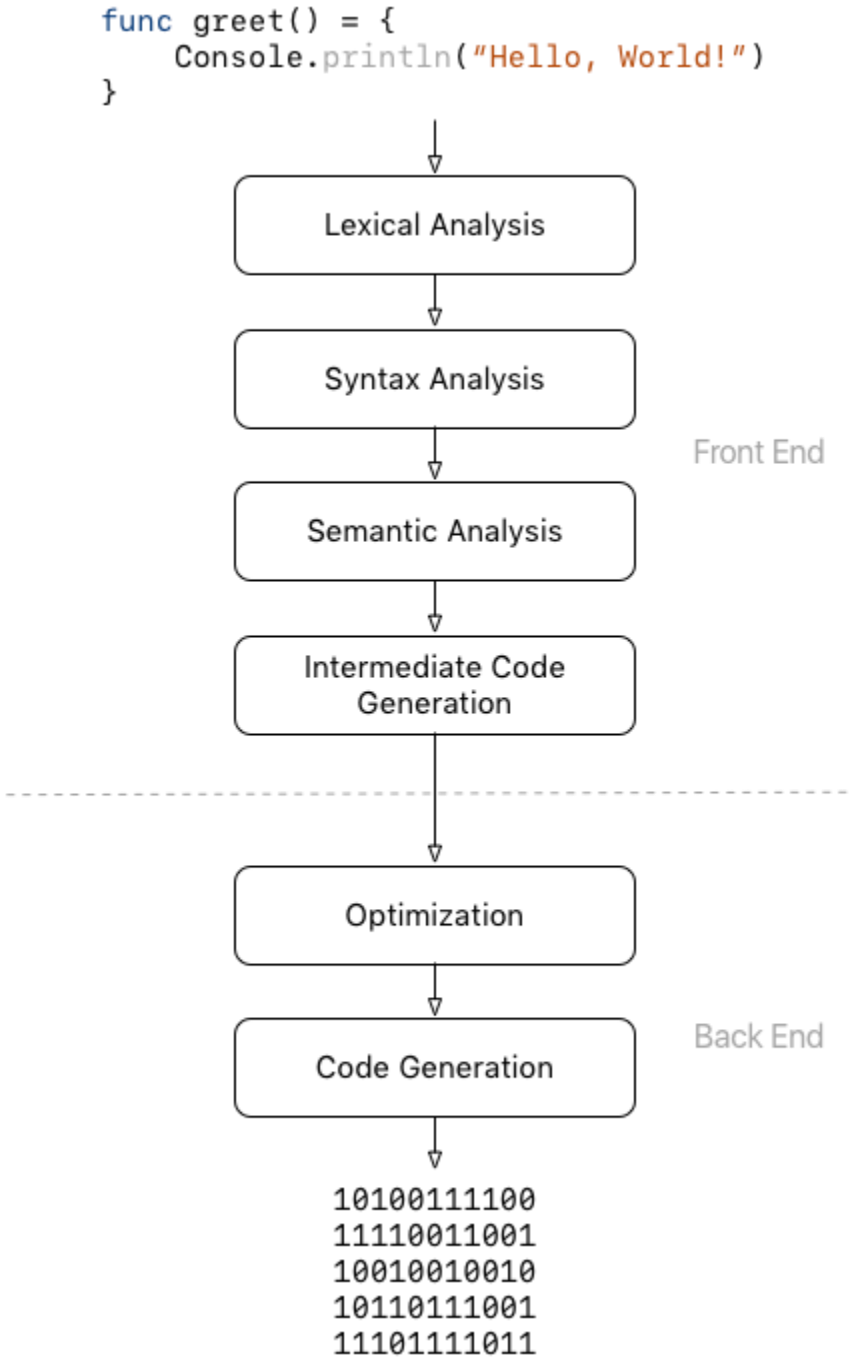
Tutorial 2 : Compilers and Interpreters

https://hackernoon.com/compilers-and-interpreters-3e354a2e41cf

A compiler can be divided into 2 parts.

- The first one generally called **the front end** scans the submitted source code for syntax errors, checks (and infers if necessary) the type of each declared variable and ensures that each variable is declared before use. If there is any error, it provides informative error messages to the user. It also maintains a data structure called **symbol table** which contains information about all the *symbols* found in the source code. Finally, if no error is detected, another data structure, an *intermediate representation* of the code, is built from the source code and passed as input to the second part.
- The second part, the **back end** uses the *intermediate representation* and the *symbol table* built by the *front end* to generate low-level code.

The phases of the front end generally include **lexical analysis**, **syntax analysis**, **semantic analysis** and **intermediate code generation** while the back end includes **optimization** and **code generation**.

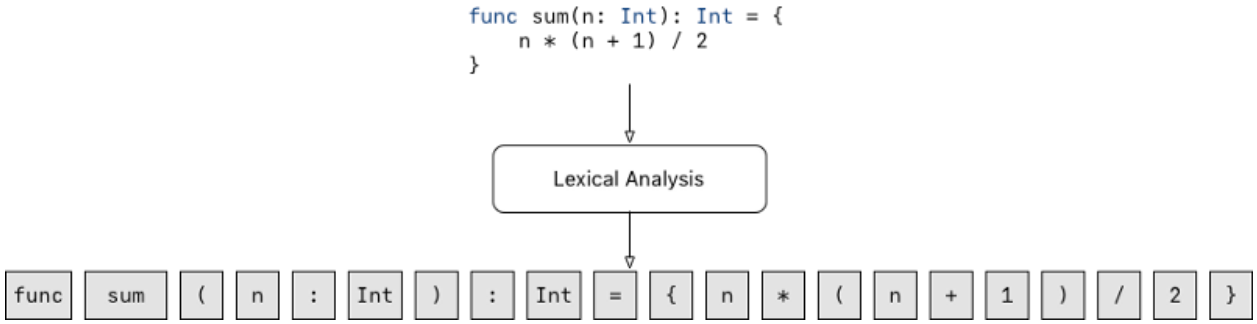


1-Lexical Analysis

The first phase of the compiler is the *lexical analysis*. In this phase, the compiler breaks the submitted source code into meaningful elements called **lexemes** and generates a sequence of **tokens** from the lexemes

A *lexeme* can be thought of as a uniquely identifiable string of characters in the source programming language, for example, *keywords* such as `if`, `while`or `func`, *identifiers*, *strings*, *numbers*, *operators* or *single characters* like `(`, `)`, `.` or `:`.

A *token* is an object describing a *lexeme*. Along with the value of the *lexeme*(the actual string of characters of the lexeme), it contains information such as its type (*is it a keyword? an identifier? an operator?* ...) and the position (line and/or column number) in the source code where it appears.



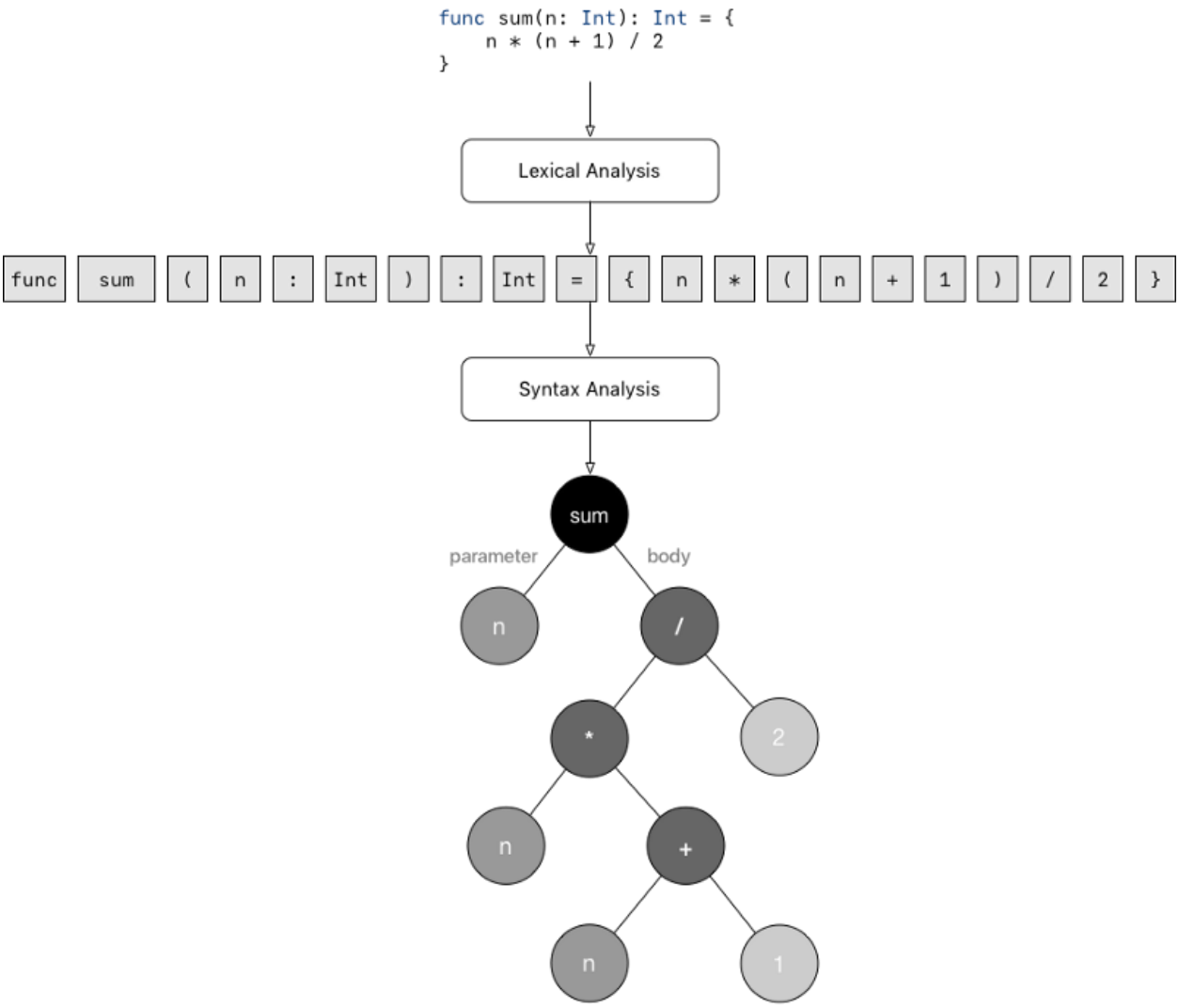
Sequence of lexemes generated during lexical analysis

If the compiler encounters a string of characters for which it cannot create a *token*, it will stop its execution by throwing an

error; for example, if it encounters a malformed string or number or an invalid character

2-Syntax Analysis

During syntax analysis, the compiler uses the sequence of *tokens* generated during the lexical analysis to generate a tree-like data structure called **Abstract Syntax Tree**, **AST** for short. The *AST* reflects the syntactic and logical structure of the program.



Syntax analysis is also the phase where eventual syntax errors are detected and reported to the user in the form of informative messages. For instance, in the example above, if we forget the closing brace `}` after the definition of the `sum` function, the compiler should return an error stating that there is a missing `}` and the error should point to the line and column where the `}` is missing.

If no error is found during this phase, the compiler moves to the *semantic analysis* phase.

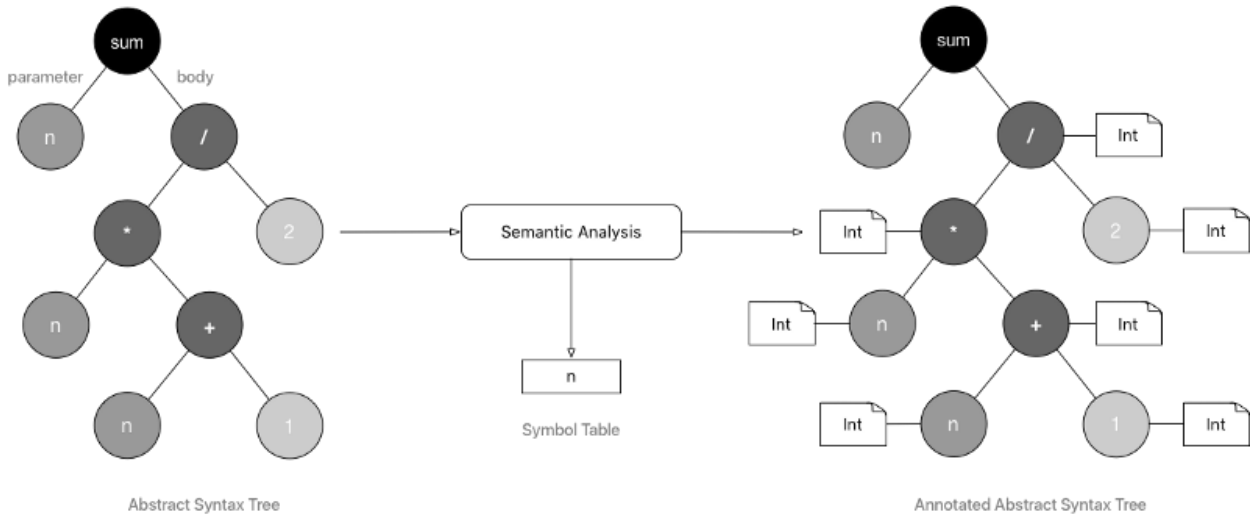
3-Symantic Analysis

During semantic analysis, the compiler uses the *AST* generated during syntax analysis to check if the program is consistent with all the rules of the source programming language. Semantic analysis encompasses

- **Type inference**. If the programming language supports type inference, the compiler will try to infer the type of all untyped expressions in the program. If a type is successfully inferred, the compiler will **annotate** the corresponding node in the *AST* with the inferred type information.
- **Type checking**. Here, the compiler checks that all values being assigned to variables and all arguments involved in an operation have the correct type. For example, the compiler makes sure that no variable of type `String` is being assigned a `Double` value or that a value of type `Bool` is not passed to a function accepting a parameter of type `Double` or again that we're not trying to divide a `String` by an `Int`, `"Hello" / 2` (unless the language definition allows it).
- **Symbol management**. Along with performing type inference and type checking, the compiler maintains a data structure

called **symbol table**which contains information about all the symbols (or names) encountered in the program. The compiler uses the *symbol table* to answer questions such as *Is this variable declared before use?*, *Are there 2 variables with the same name in the same scope? What is the type of this variable? Is this variable available in the current scope?* and many more.

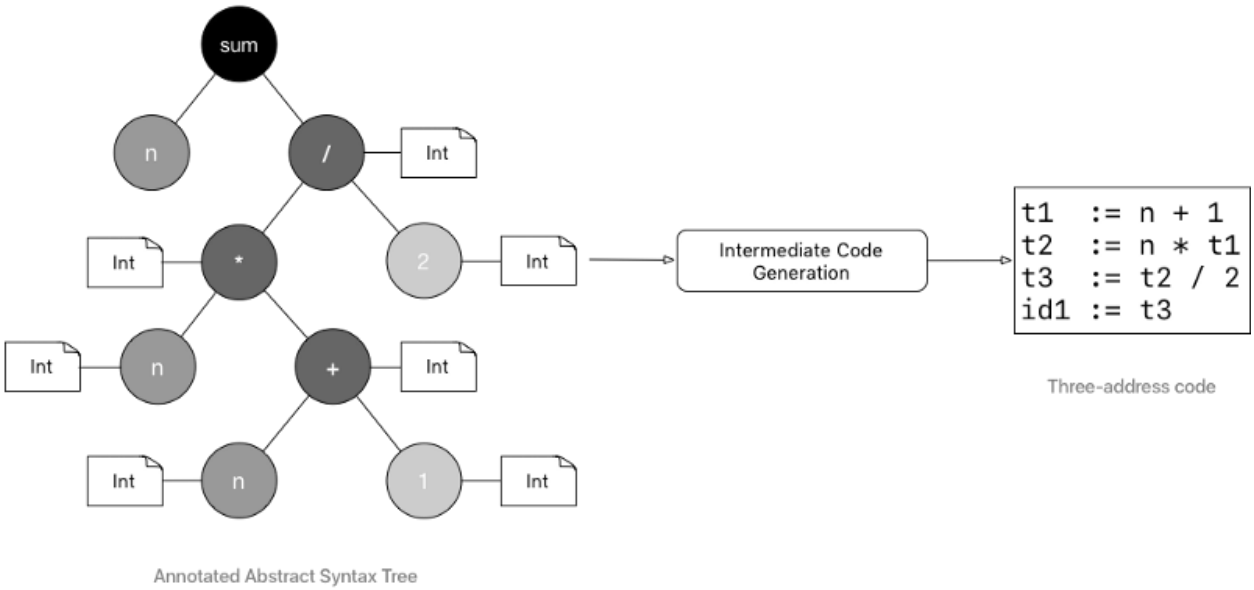
The output of the semantic analysis phase is an **annotated AST** and the **symbol table**.



4-Intermediate Code Generation

After the semantic analysis phase, the compiler uses the *annotated AST* to generate an intermediate and machine-independent low-level code. One such intermediate representation is the three-address code**.**

The *three-address code* (*3AC*), in its simplest form, is a language in which an instruction is an assignment and has at most 3 operands.
Most instructions in *3AC* are of the form `a := b <operator> c` or `a := b`.



The image represent the code below :

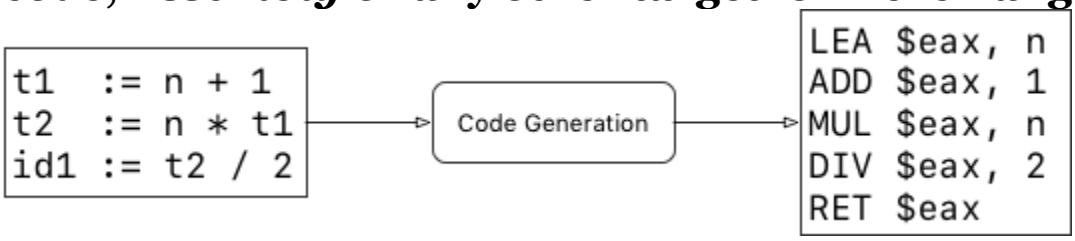
```
func sum(n: Int): Int = {
    n * (n + 1) / 2
}
```

5- Optimization :

In the optimization phase, the first phase of the *back end*, the compiler uses different optimization techniques to improve on the intermediate code generated by making the code faster or shorter for example.
For example, a very simple optimization on the *3AC* code in the previous example would be to eliminate the temporary assignment `t3 := t2 / 2` and directly assign to `id1` the value `t2 / 2`.

6- Code Generation

In this last phase, the compiler translates the optimized intermediate code into machine-dependent code, *Assembly* or any other target low-level language.



7- Compiler vs. Interpreter

Interpreters and compilers are very similar in structure. The main difference is that an interpreter directly executes the instructions in the source programming language while a compiler translates those instructions into efficient machine code.

An interpreter will typically generate an efficient intermediate representation and immediately evaluate it. Depending on the interpreter, the intermediate representation can be an *AST*, an *annotated AST* or a machine-independent low-level representation such as the *three-address code*.

Not obvious difference

Tutorial 3 : Lexical Analysis

https://hackernoon.com/lexical-analysis-861b8bfe4cb0

The lexer**, also called** lexical analyzer **or** tokenizer**, is a program that breaks down the input source code into a sequence of lexemes. It reads the input source code character by character, recognizes the lexemes and outputs a sequence of tokens describing the lexemes**

What's a lexeme?

A **lexeme** is a single identifiable sequence of characters, for example, keywords (such as `class`, `func`, `var`, and `while`), literals (such as numbers and strings), identifiers, operators, or punctuation characters (such as `{`, `(`, and `.`)

What's a token?

A **token** is an object describing the *lexeme*. A token has a **type** (e.g. Keyword, Identifier, Number, or Operator) and a **value** (the actual characters of the described lexeme*)*. A token can also contain other information such as the line and column numbers where the lexeme was encountered in the source code

See code Example

The lexer in code

A **lexer** can be implemented as a class, whose constructor takes an input string in parameter (representing the source code to perform lexical analysis on). It exposes a method to recognize and return the next token in the input.

```
Class
Lexer
{
        constructor(input) {
            this.input = input;
        }

        // Returns the next recognized 'Token' in the input.
        nextToken() {
            // ...
        }
    }
```

How tokens are recognized

All possible lexemes that can appear in code written in a programming language, are described in the specification of that programming language as a set of rules called **lexical grammar**. Rules in the lexical grammar are often transformed into automata called **finite state machines** (*FSM*). The lexer then simulates the finite state machines to recognize the tokens.

Lexical Grammar

The lexical grammar of a programming language is a set of formal rules that govern how valid lexemes in that programming language are constructed. For example, the rules can state that a string is any sequence of characters enclosed in double-quotes or that an identifier may not start with a digit. The rules in the lexical grammar are often expressed with a set of **regular definitions**.

A regular definition is of the form `<element_name> = <production_rule>`where `<element_name>` is the name given to a symbol or a lexeme that can be encountered in the programming language and `<production_rule>` is a **regular expression** describing that symbol or lexeme.

```
letter = [a-zA-Z]
```

For example, the regular definition above defines a *letter* as any lowercase or uppercase alphabet character.

A regular definition can make use, in its regular expression, of any element name defined in the same lexical grammar.

```
letter        = [a-zA-Z]
digit         = [0-9]
identifier    = (letter | _) (letter | digit | _)*
```

Finite State Machines

.....

Lexical Grammar and FSMs

To recognize a token described by a regular definition, the regular expression in the definition is often transformed into a FSM. The resulting FSM has a **finite number of states** comprising an **initial state** and a **set** of **accepting states**.

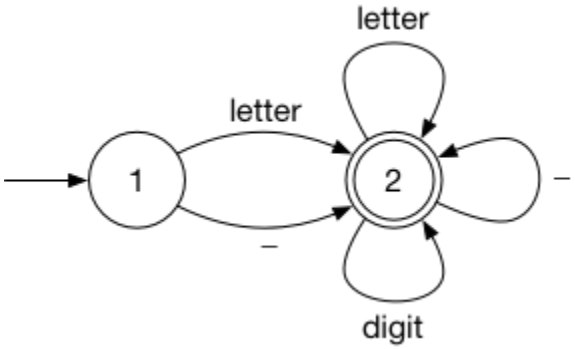
From regular expression to FSM

We can transform any regular expression into a FSM by building on top of the following three basic rules.

1- A|B
2- AB
3- A*

Any other regular expression can be transformed into a *FSM* by reusing one or any combination of the basic rules above

final FSM for *identifier*



The FSM in code

A FSM is a combination of

- the set of all the possible **states** the FSM can be in
- the **initial state** the FSM is in
- a set of **accepting states**
- and the set of all **transitions**.

```
class
FSM {
        constructor(states, initialState, acceptingStates) {
            this.states = states;
            this.initialState = initialState;
            this.acceptingStates = acceptingStates;
        }
    }
```

The transitions of a FSM can be modeled with a function that takes a state `currentState` and a character or symbol `input` in parameters and returns the state the FSM will be in after consuming `input` while in state `currentState`. We can call that function the **transition function** and name it `nextState()`

Most often, the transition function will be a switch statement on the parameter `currentState`, with each case returning the next state according to the parameter `input`

To assist in the implementation of the transition function, the FSM can first be converted into a transition table. The transition table maps each state S and input I to a state S', where S' is the state the FSM will be in when the input I is consumed from the state S.

Now, to complete the constructor of our FSM class, let's add a parameter `nextState` of type `Function` representing the transition function to it.

```
class FSM {
   constructor(states, initialState, acceptingStates, nextState) {
      this.states = states;
      this.initialState = initialState;
      this.acceptingStates = acceptingStates;
      this.nextState = nextState; // The transition function.
   }
}
```

```
nextState(currentState, input) {
   switch (currentState) {
      case 1:
         if (input === 'a' || input === 'b') {
           return 2;
         }
      break;

      case 2:
         if (input === 'c') {
```

```
            return 3;
          }
        break;

      default:
        break;
    }

  return NoNextState; // 'NoNextState' here is a constant to specify that there is no next state for the provided parameters.
}
```

The next step in the implementation of our FSM is to add a function allowing to run**,** simulate **or** execute **the FSM on an input string. The function will return a boolean specifying whether the input string (or a subset of the input string) matches the regular expression corresponding to the FSM.**

```
/// Runs this FSM on the specified 'input' string.
    /// Returns 'true' if 'input' or a subset of 'input' matches
    /// the regular expression corresponding to this FSM.
    run(input) {
        // ...
```

```
}
```

The implementation of the `run` function is straightforward. The function will read the input character by character while keeping track of the current state the FSM is in. For each character read, it updates the current state with the next state the FSM will be in, by calling the transition function `nextState()`. At the end of the execution of the loop, if the current state is one of the accepting states of the FSM, then the input string (or a subset of the input string) matches the regular expression corresponding to the FSM.

```
run(input) {
    let currentState = this.initialState;

    for (let i = 0, length = input.length; i < length; ++i) {
      let character = input.charAt(i);
      let nextState = this.nextState(currentState,
character);

      // If the next state is one of the accepting states,
      // we return 'true' early.
      if (this.acceptingStates.has(nextState)) {
        return true;
      }
```

```
        if (nextState === NoNextState) {
            break;
        }

        currentState = nextState;
    }

    return this.acceptingStates.has(currentState);
  }
}
```

Usage of a FSM

To conclude this part on FSMs, let's see how we can use our newly implemented `FSM` class to recognize identifiers.

Let's reuse the following regular definitions.

```
let
fsm =
FSM();
      fsm.states = new Set([1, 2]);
      fsm.initialState = 1;
      fsm.acceptingStates = new Set([2]);
      fsm.nextState = (currentState, character) => {
          switch (currentState) {
              case 1:
```

```
            // Assuming 'CharUtils' is a class with static helper
            // methods allowing us to answer simple questions about
            // characters. For example 'CharUtils.isLetter(c)'
            // determines if the character 'c' is a letter.
            if (CharUtils.isLetter(character) || character === '_') {
                return 2;
            }
            break;

        case 2:
            if (CharUtils.isLetter(character) || CharUtils.isDigit(character) ||
character === '_') {
                return 2;
            }
            break;
    }

    return NoNextState;
}
```

Our FSM instance can then be used to recognize identifiers.

fsm.run("camelCaseIdentifier"); // => true

fsm.run("snake_case_identifier"); // => true

fsm.run("_identifierStartingWithUnderscore"); // => true

fsm.run("1dentifier_starting_with_digit"); // => false

fsm.run("ident1f1er_cont4ining_d1g1ts"); // => true

Putting it all together

Armed with all the necessary tools (lexical grammar, regular expressions, FSMs, transition tables, etc.), we can now have a look at how they all come together in the implementation of a lexer.

Let's consider a simple language for performing mathematical operations. The language supports the four basic arithmetic operators (+, -, * and /), comparison operators (>, ≥, <, ≤ and ==), and grouping using parenthesis. It also has basic support for variables and assignment using the = symbol.

Below are examples of valid instructions in our mini language.

```
1 + 2

(5 - 4) + 7

12.34 * 5e-9

pi = 22 / 7

radius = 5

circle_circumference = 2 * pi * radius
```

The valid lexemes for this language, identifiers**,** numbers**,** parenthesis**, and** operators **are described with the following regular definitions.**

```
letter      = [a-zA-Z]
digit       = [0-9]
digits      = digit digit*
identifier  = letter | (letter | digit | _)*
fraction    = . digits | ε
exponent    = ((E | e) (+ | - | ε)) digits) | ε
number      = digits fraction exponent
operator    = + | - | * | / | > | >= | < | <= | = | ==
parenthesis = ( | )
```

LEXEME

TokenType

The first step in implementing a lexer is to add a token type for each valid lexeme.

```
let TokenType = {
    Identifier: 'identifier',
    Number: 'number',
    Operator: 'operator',
    Parenthesis: 'parenthesis'
};
```

Because there is a finite number of operators and parenthesis, we will gain in clarity and granularity by adding a specific token for each type of operator and parenthesis. It will also be helpful to add a special token `EndOfInput` to be returned when all the characters in the input have been read.

```
let TokenType = {
    /// Identifiers and literals
```

```
Identifier:        'identifier',
Number:            'number',

/// Arithmetic operators
Plus:              'plus',               // +
Minus:             'minus',              // -
Times:             'times',              // *
Div:               'div',                // /

/// Comparison operators
GreaterThan:       'greater than',       // >
GreaterThanOrEqual: 'greater than or equal', // >=
LessThan:          'less than',          // <
LessThanOrEqual:   'less than or equal',   // <=
Equal:             'equal',              // ==

/// Assignment operator
Assign:            'assign',             // =

/// Parenthesis
LeftParenthesis:   'left parenthesis',     // (
RightParenthesis:  'right parenthesis',    // )
```

```
  /// Special tokens
  EndOfInput:       'end of input'
};
```

The next step is to complete the implementation of the `Lexer` class.

The Lexer class

Let's start by adding properties to the `Lexer` class to keep track of the current position in the input, as long as the current line and column.

`nextToken()` method.

A strategy we could use for `nextToken()` is to read the character at the current position. If the character matches the starting character in the production rule of one of the lexemes, we delegate the recognition of the lexeme to a helper method corresponding to that production rule

---

```
class Lexer {
  constructor(input) {
    this.input = input;
    this.position = 0;
    this.line = 0;
    this.column = 0;
  }
```

```
/// Returns the next recognized 'Token' in the input.
nextToken() {
    if (this.position >= this.input.length) {
        return new Token(TokenType.EndOfInput);
    }

    let character = this.input.charAt(this.position);

    if (CharUtils.isLetter(character)) {
        return this.recognizeIdentifier();
    }

    if (CharUtils.isDigit(character)) {
        return this.recognizeNumber();
    }

    if (CharUtils.isOperator(character)) {
        return this.recognizeOperator();
    }

    if (CharUtils.isParenthesis(character)) {
        return this.recognizeParenthesis();
```

```
    }

    // ...
}

/// Recognizes and returns an identifier token.
recognizeIdentifier() {
    // ...
}

/// Recognizes and returns a number token.
recognizeNumber() {
    // ...
}

/// Recognizes and returns an operator token.
recognizeOperator() {
    // ...
}

/// Recognizes and returns a parenthesis token.
recognizeParenthesis() {
```

```
        // ...
    }
}
```

Examining the helper methods

```
/// Recognizes and returns a parenthesis token.
recognizeParenthesis() {
    let position = this.position;
    let line = this.line;
    let column = this.column;
    let character = this.input.charAt(position);

    this.position += 1;
    this.column += 1;

    if (character === '(') {
        return new Token(TokenType.LeftParenthesis, '(', line, column);
    }

    return new Token(TokenType.RightParenthesis, ')', line, column);
}
```

In the definition for the `operator` lexeme, we can notice that we basically have 2 types of
operator, *arithmetic* and *comparison* operators (technically we have a third one, the assignment operator `=` but to simplify the implementation, we'll add it to the comparison operators group here).

For readability, we can delegate the recognition of each type of operator to a specific helper function.

See code on the automata project folder

Recognizing identifiers

We could build an FSM for this and use it to recognize identifiers but these rules are simple enough to be implemented with a loop. We just have to keep reading characters in the input until we encounter a character that is not a letter, a digit or an underscore
See code

Recognizing Numbers

See pdf