Chapter 1: Introduction

1 Compiler

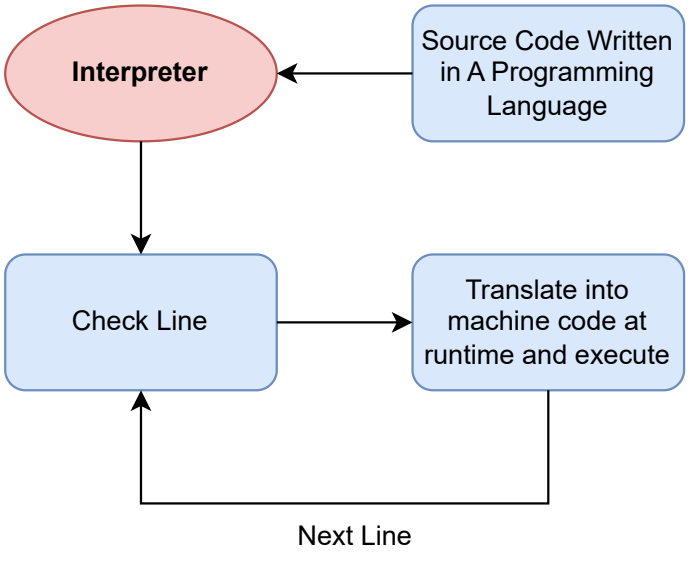
Definition

A compiler is a software program that takes a piece of code as input, analyzes it for errors, and generates output in the form of messages or logs. Additionally, it produces an executable file that can be run on a computer.

2 Interpreter

Definition

An interpreter follows the same steps as a compiler but differs in execution. Instead of analyzing the entire source code and generating an object file, it processes the code line by line then execute it immediately without producing an output file.



Note

Pros of an Interpreter:

- **Easier Debugging**: Errors are detected as the program runs, making it easier to pinpoint and fix issues.
- **Platform Independence at Source Level**: The same source code can be executed on different systems without recompilation, as long as an interpreter is available.
- **Smaller Program Size**: Interpreted programs do not require large binary files, reducing storage needs.

Cons of an Interpreter:

- **Slower Execution**: Since interpretation and execution happen simultaneously, it is slower compared to compiled programs.
- **Runtime Errors**: Errors only appear when execution reaches the faulty line, which can lead to unexpected crashes.

3 Steps of Compilation

Steps

The process of compilation is done following these steps chronologically:

1. Lexical Analysis
2. Syntactic Analysis (Parsing)
3. Semantic Analysis
    - Generation of Intermediate Code
    - Syntax-Directed Translation
4. Generation of Object Code

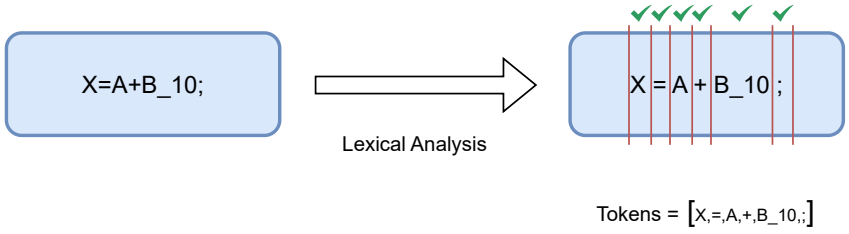
3.1 Lexical Analysis

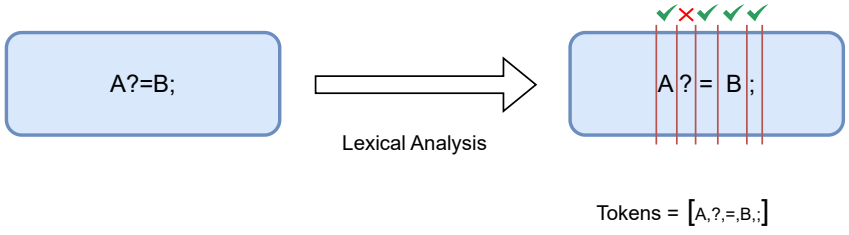
Lexical Analysis

Lexical analysis consists of dividing the entire code into lexical entities (tokens) storing them in a dictionnary and evaluating each of them independently throught automata or regex. Tokens can be divided based on:

- Spaces
- Logical operators (AND, OR, NOT, >=, >, <=, <, =)
- Arithmetic operators (+, -, *, /)
- Separators ([ ], { }, ( ))

Example





3.1.1 Types Of Token

Types

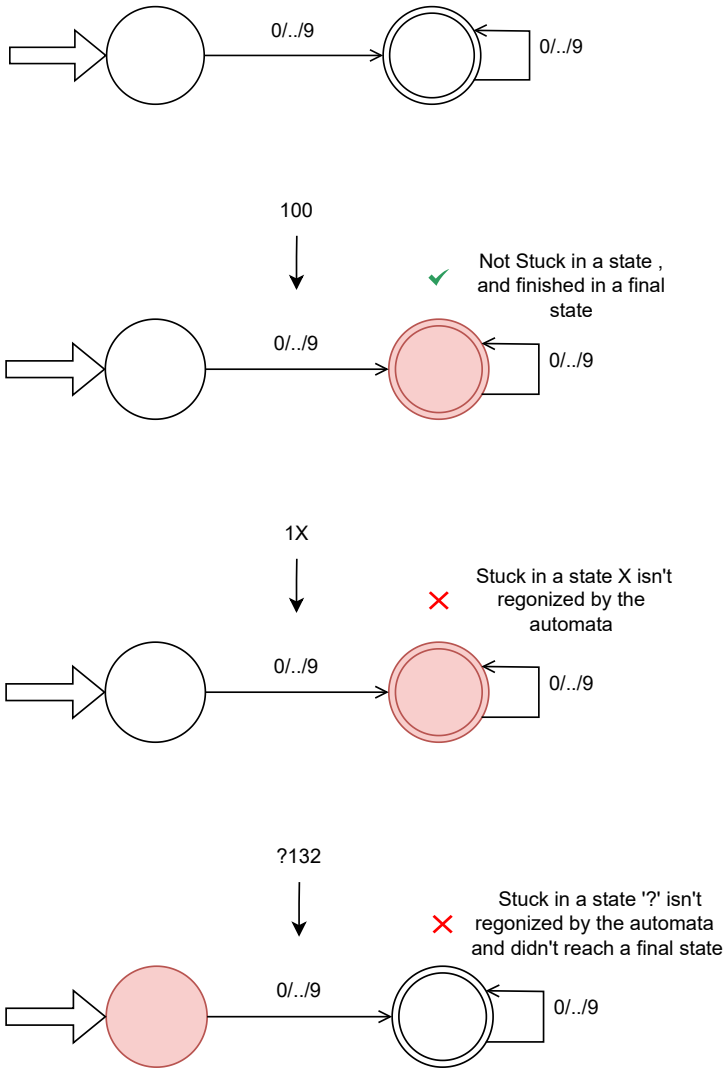
- **Constant**: Represent literal values like numbers : (3.14 , 1021 ,-120) , literal string : "hello world !" and character 'a'.
- **Keyword**: Reserved words that have a predefined meaning in the language, like 'if', 'while', and 'return'.
- **Identifier**: Names given to variables, functions, or objects.
- **Separator**: Symbols used to separate tokens (logical , arithmetic operators ...etc)

3.1.2 Automata

Automata

Lexical analysis uses a finit deterministc automata to verify wether a given token is correct or no , a token is considered correct if it reaches an end state with no blockage in the process

Example



Note

Never loop on the start state, as this can lead to undesired behavior and unexpected results. Additionally, such a loop could cause the automaton to recognize the empty word ϵ, which does not exist in practice.

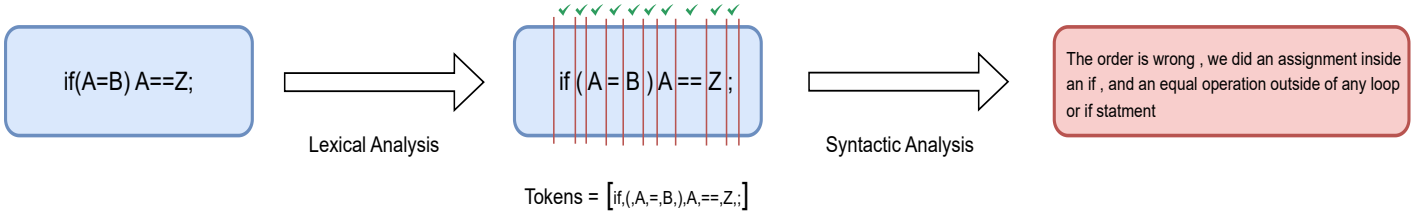
3.2 Syntactic Analysis (Parsing)

Syntactic Analysis

Syntactic analysis checks if the order of the tokens are correct and adheres to the required format. There are two methods:

- Descending methods
- Ascending methods

Example



Note

If a code is lexically correct, it does not guarantee that it will be syntactically correct. However, the opposite is true:

$$\text{Correct Syntactically} \rightarrow \text{Correct Lexically}$$

$$\text{Correct Lexically} \nrightarrow \text{Correct Syntactically}$$

3.3 Semantic Analysis

Semantic Analysis

This phase ensures the code is logically correct and adheres to the language's rules. It verifies **type mismatches, variable declarations, function calls, and more**.

- **Generation of Intermediate Code**: Produces a simplified, platform-independent representation of the program from the sentences (syntatic entities). This step is crucial for optimizing the code and converting it into machine code later , it has many types :
  - **Quadruple Form**
  - **Postfix and Prefix Forms ... etc**
- **Syntax-Directed Translation**: Generates intermediate representations, ensuring that the semantics (meaning) of the program guide the creation of subsequent steps.

3.4 Object Code Generation

Object Code Generation

The final step of compilation, where the object code is generated from the intermediate code to execute the program. This is the only step in the compilation process that we will not cover in this course.

4 Dictionary(Symboles Table)

Dictionary

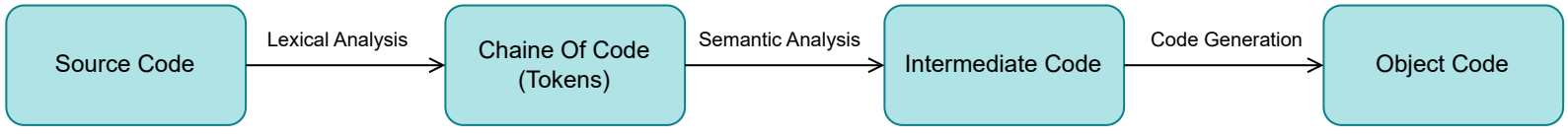
The dictionary is an array that stores tokens and information about them. It is first initialized during lexical analysis and then progressively filled as the analysis continues.

Tokens	Token Type	Semantic Type	Memory address
X	Identificator	int	10021
Y	Identificator	float	9212

Note

Memory allocation occurs after semantic analysis to prevent allocation when the source code contains errors. This ensures that resources are only allocated once the code is verified as semantically correct.

5 Compilation Steps Diagram



Chapter 2: Lexical Analysis

1 Lexical Analysis

Lexical Analysis

The first step of compilation takes the source code as input, representing a sequence of characters separated or not by spaces and new lines. It consists of:

- Fetching all lexical entities (tokens) and verifying whether they are part of the language. The tokens are then classified as constants, identifiers, keywords, or separators.
- Removing all extra spaces and comments.
- Producing a chaine of code (sequence of tokens) that will be used in syntactic analysis.
- Initializing the **symbol table** (dictionary) by inserting the retrieved tokens and their corresponding codes after verification (tokens must be unique (no repetition)). The symbol table will be updated as we progress through the next compilation steps.

2 Notes

Optimization

- In real-life implementations, the symbol table is often divided into sub-tables based on token types improving search speed and optimizes memory usage.
- We can also prefill the table of keywords and separators since they are known in advance. This simplifies the automaton, as it no longer needs states to recognize keywords and separators , it can directly compare tokens to the prefilled tables.

Reserved Words & Keywords

- **Keywords :** are special tokens in a programming language that have predefined semantic meanings, such as data types, control flow structures, or loop constructs (e.g., ‘if‘, ‘while‘, ‘int‘, ‘return‘). These words are integral to the syntax of the language.
- **Reserved words :** are tokens that cannot be used as identifiers (like variable or function names) because they are reserved for the language’s future use or for language-specific functionality.
- In most compilers, **all keywords are reserved words**. However, some non-standard or unconventional compilers may allow certain reserved words to be used as identifiers. For example, in these compilers, you might be able to write code like ‘int if = 5;‘, which is not allowed in standard compilers.

Two Schools of Thought

Imagine encountering ><:

- **Method 1**: Considers >< as a single token and checks it against the prefilled separator table. Since no match is found, it is classified as a lexical error.
- **Method 2**: Recognizes > and < as separate, valid tokens but considers their order incorrect, making it a syntactic error. This approach is more practical , and used in most compilers.

3 Implementation of Lexical Analysis

Implementation

Lexical analysis utilizes a finite deterministic automaton to break the source code into lexical entities (tokens) and verify their validity within the language. To implement the automaton, we need to:

- Choose a suitable data structure to represent the automaton in memory.
- Develop an algorithm for tokenizing the source code and validating tokens.

3.1 Representation of Automaton

Representation

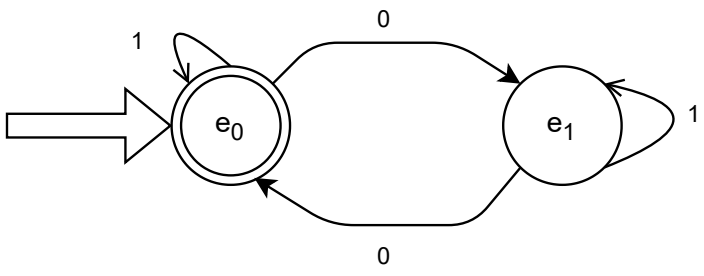
- The automaton will be represented by a transition matrix, where each row corresponds to a state, and each column represents a letter.
- An array will be used to store the final states.
- A constant will define the initial state.

Reminder

- **Deterministic:** At most one transition is possible from state e_0 to e_1 for a given letter l_0.
- **Finite:** The automaton has a limited number of states and transitions.
- **Token Acceptance:** A token is considered valid if the automaton processes it without encountering a dead end (no blockage) and reaches a final state.

Example

$L = \{ u \in \{0,1\}^* \ / \ |W|_0 \equiv 0[2]\}$



	0	1
e_0	e_1	e_0
e_1	e_0	e_1

FinalStateArray = $\{e_0\}$
Initial State Constant = e_0

Execution Of The Automata For The Word 1011

State	Current Letter	Action
e_0	1	$e_0 \rightarrow e_0$
e_0	0	$e_0 \rightarrow e_1$
e_1	1	$e_1 \rightarrow e_1$
e_1	1	$e_1 \rightarrow e_1$

1011

1011

1011

1011

1011#

Conclusion

Even though there is no blockage when the automaton reads 1011, it does not reach a final state in the end. Since $e_1 \notin$ FinalStateArray, we conclude:

$$\boxed{1011 \notin L}$$

Note

The symbol # is used to mark the end of each token.

3.2 Algorithm of Reconnaissance

Algorithm

The algorithm utilizes a finite deterministic automaton represented as a transition matrix. A token is recognized by the automaton if and only if it reaches '#' in the token's terms and a final state in the automaton.

We will first present the reconnaissance algorithm for a single token, followed by its application to an entire source code.

3.2.1 Variables & Predefined Functions

Notation

- **Entity**: Lexical entity (token).
- $\mathbf{t}_c$: Current term of the token.
- $\mathbf{t}_s$: Next term of the token.
- **F**: Array of final states.
- $\mathbf{E}_c$: Current state of the automaton.
- **M**: Transition matrix.
- **Lire(Entity)**: Reads a lexical entity from the source code, stores it in the **Entity** variable, and appends '#' at the end.
- **Saut_Blanc()**: Skips whitespace, including tabulations and newline characters and comments.

Algorithm 1 Algorithm Reconnaissance Of Single Token

```
Begin
Lire(Entity);                    -- Reads token
t_c ← Entity[0];                 -- Initializ with 1st character
E_c ← Initial State;             -- Intialize with initial state
while ( E_c != ∅ AND t_c != '#' ) do    -- Loop as long as there is no blockage and not all character of the token have been analyzed
    E_c ← M[E_c,t_c];            -- Gets next state
    t_c ← t_s;                   -- Gets next character of the token
end while
if ( E_c != ∅ ) then             -- Check if there is a blockage
    print( 'Error the lexical entity isn't regonised by the automaton because blockage happened' );
else if ( E_c ∉ F ) then         -- Check if automaton reaches a final state
    print( 'Error the lexical entity isn't regonised by the automaton because it didn't reach a final state' );
else                             -- Token is regonized
    Codify the lexical entity and put it in the symbole table.
end if
End
```

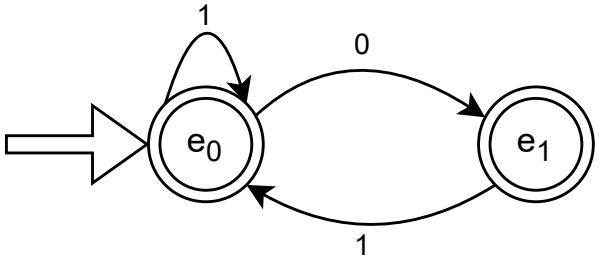
Algorithm 2 Algorithm Reconnaissance Of All Tokens

```
Begin
Saut_Blanc();                    -- Ignores spaces , newlines and comments before reading the token
while ( NOT EOF ) do             -- Loop as long as we didn't reach EOF(End Of File)
    Lire(Entity);                -- Reads token
    Rest Of The Algorithm Of Reconnaissance Of Single Token
    Saut_Blanc();
end while
End
```

Example

Language of binary code with no successive 0s:

- Create the transition matrix
- Analyze '101#'



	0	1
e_0	e_1	e_0
e_1	/	e_0

$F = \{e_0 , e_1\}$

Initial State = e_0

E_c	t_c	Action
e_0	'1'	$E_c = M[e_0,'1'] = e_0$ $t_c = t_s = '0'$
e_0	'0'	$E_c = M[e_0,'0'] = e_1$ $t_c = t_s = '1'$
e_1	'1'	$E_c = M[e_1,'1'] = e_0$ $t_c = t_s = '\#'$ Since $E_c \neq \emptyset$ and $E_c \in F$: token is regonized therefore codified and inserted in the symbole table