# CMSC 124

## DESIGN AND IMPLEMENTATION OF PROGRAMMING LANGUAGES

CNM PERALTA

# LANGUAGE TRANSLATION ISSUES

As we saw in our earlier discussion, in order **to implement a language**, it must often be **translated**.

To translate a language, we must **understand** its **syntax** and **semantics**.

# Syntax

Form of a language's expressions, statements, and program units.

# CRITERIA FOR JUDGING LANGUAGE SYNTAX

# 1.

# Readability

Algorithm structure and program data must be apparent through program code.

Basically, the language **syntax** should **reflect** program **semantics**.

# 2.

# Writability

Syntactic structure should be **concise**.

Syntax **readability** and **writability** often **contradict** each other.

# 3.

# Ease of verifiability

Program correctness must be easy to verify.

# 4.

# Ease of translation

Programs must be easy to translate to executable form.

# Syntax must be regular.

# 5.

## Lack of ambiguity

# Ambiguity

Syntactic constructs have more than one interpretation/meaning.

It is one of the **central problems** of **program design.**

# EXAMPLE: ALGOL

```
if condition1 then if
  condition2 then
    statement1
else
    statement2
```

This **else** is dangling, i.e., it is unclear to which **if**-statement it belongs to.

# EXAMPLE: FORTRAN

`A(I, J)`

This may be a **function call** to A with parameters I and J, or **access to an array element** with indices I and J.

**Ambiguity** can **oftentimes** be **resolved**.

# EXAMPLE: ALGOL

```
if condition1 then if
  condition2 thene
    begin
      statement1
    end
else
  statement2
```

# Inherently ambiguous

Languages with that have no discernible way to become unambiguous.

# SYNTACTIC ELEMENTS

# What elements make up programming language syntax?

# 1.

# Character set

Usually, standard character sets, like ASCII and Unicode, are better because they allow I/O equipment support.

# Internationalization

has required character sets to be represented with **16 bits** (**formerly 8**), to represent **65,536 characters** (e.g., Chinese, Japanese, etc.)

# 2.

# Identifiers

User-defined names to refer to user-define data/constructs like variables and functions.

Usually, **starts with letters** and can have other **letters**, **digits**, and **special characters** (**.** or **-**) to **enhance readability.**

Identifiers usually have a **length limit**; if this limit is **too small**, it may **hamper readability**.

# 3.

# Operator symbols

Special characters to denote operations.

# Basically,

# + − * / %

Some languages (e.g., LISP) use identifiers instead:

`PLUS`, `TIMES`, etc.

# 4.

# Keywords and reserved words

# Keywords

Words used as a fixed part of the syntax of a statement.

Examples: `if`, `for`, `while`, etc.

# Reserved words

Keywords that cannot be used as a programmer-chosen identifier, used to accommodate extensions/updates of a language.

# 5.

# Noise words

Optional words that are inserted into statements to improve readability.

# EXAMPLE: COBOL

`GO TO label`

The **TO** is optional but is an improvement over `GO label`.

# 6.

# Comments

Programmer-inserted statements that are ignored by the compiler.

Comments are important for documentation.

# 7.

## Blanks and spaces

Whitespaces, new lines, tabs, etc.

Some languages use whitespaces as **delimiters** or **separators**.

# 8.

## Delimiters and brackets

# Delimiters

Syntactic elements used to **mark** the **beginning** and **end** of a syntactic unit.

# Brackets

Paired delimiters.

Delimiters and brackets assist in **improving readability** and **removing ambiguity**.

# 9.

# Fixed- and free-field formats

# Fixed-field syntax

Positioning of input lines convey information about that line.

Example: COBOL

# Free-field syntax

Program statements can occur at any point on the line.

Example: Almost all of the languages you know.

# 10.

# Expressions

Functions that access data objects in a program and return some value.

# 11.

# Statements

Made up of (possibly many) expressions, they are the most prominent syntactic component of imperative languages.

Languages may have a single basic statement format for syntax regularity, like in functional languages:

```
(+ 5 3 2)
(set! x 5.0)
(number? x)
```

Or many statement formats for different constructs for syntax readability like almost every other PL that is not functional.

Statements can be **simple** or **nested/embedded**.

# SYNTACTIC ORGANIZATION OF MAIN AND SUBPROGRAMS

The aforementioned syntactic elements are organized in different ways by the different PLs.

# 1.

# Separate subprogram definitions

Each subprogram (function) definition is a separate syntactic unit.

This allows separate compilation of subprograms and subsequent linking of subprograms at load time.

# EXAMPLE: C

```c
double max(double a, double b) {
//code
}
double getSum(double a, double b) {
//code
}
int main() {
//code
}
```

FORTRAN also uses this syntactic organization.

# 2.

# Separate data definitions

All operations that operate on a data object are grouped together, usually using classes.

# EXAMPLE: JAVA

```
public class Student {
  private String name;
  private int age;
  public String getName() {return name;}
  public int getAge {return age;}
  public void setName(String name)
    {this.name = name;}
  public void setAge(int age) {this.age = age;}
}
```

# 3.

# Nested subprogram definitions

Subprograms appear as declarations within the main program.

Furthermore, subprograms may also be nested within other subprograms.

# EXAMPLE: PASCAL

```
function E(x: real): real;
    function F(y: real): real;
      begin
        F:= x+y
      end
begin
  E:=F(3)+F(4)
end;
```

# 4.

# Separate interface definitions

Subprograms appear as declarations within the main program.

Hybrid approach between 1 (FORTRAN) and 3 (Pascal).

# EXAMPLE: C

Use of header files (`.h`) containing interface definitions implemented in `.c` files.

# 5.

# Data descriptions separated from executable statements

All data are global (no local variables)

Allows independence between data formats and algorithms.

# EXAMPLE: COBOL

```
DATA DIVISION.
  *variable declarations
PROCEDURE DIVISION.
  *program statements
```

# 6.

# Unseparated subprogram definitions

No syntactic distinction between main and subprograms.

Any statement may be part of the main program as well as any number of subprograms.

Examples are SNOBOL4 and BASIC.

# STAGES OF TRANSLATION

Translation is more complex the farther the source program is from executable program form.

# TWO MAJOR PARTS

# 1.

Analysis of the input source program.

# 2.

## Synthesis of executable object program

Compilers commonly need **up to three passes** of the **source program**.

**One-pass compilers** construct object code as source code is analyzed, emphasizing **compilation speed**.

Two-pass compilers (most common) use each pass to perform the two major parts of translation.

Three-pass compilers add an optimization pass between the two major parts of translation.

# ANALYSIS OF THE SOURCE PROGRAM

There are three steps in source program analysis.

# 1.

# Lexical analysis

Recognizes small-scale language constructs, i.e., names, numeric literals, etc.

# 2.

# Syntax analysis

Recognizes large-scale language constructs, i.e., expressions, statements, program units, etc.

# 3.

# Semantic analysis

Analysis of the **meaning of the syntactic constructs** recognized by the syntax analyzer.

There are three reasons why lexical analyzers are separated from syntax analyzers.

# 1.

Lexical analysis techniques are simpler than syntax analysis techniques.

The integration of the simpler lexical analysis with complex syntax analysis can make both processes more complicated.

# 2.

If is **more efficient** to **optimize** the **lexical analyzer** than the syntax analyzer.

Putting them together disallows the optimization of the lexical analyzer.

# 3.

**Syntax analyzers** are **more portable** than lexical analyzers because it is the latter that reads input program files.

# LEXICAL ANALYSIS

Lexical analysis is done by the ***lexical analyzer*** or ***scanner***.

Lexical analysis recognizes elementary syntactic constructs via **pattern matching**.

Group sequences of characters to elementary PL constituents.

These resulting **groupings** are called

***lexemes.***

The categories/classifications for these lexemes are called

**tokens.**

# EXAMPLE: BEFORE

```
int main() {
int x;
printf(“Enter a number: ”);
scanf(“%d”, &x);
return 0;
}
```

# EXAMPLE: AFTER

```
int main() {
int x;
printf(“Enter a number: ”);
scanf(“%d”, &x);
return 0;
}
```

# EXAMPLE: AFTER

| Lexeme | Token Tag | Lexeme | Token Tag |
|---|---|---|---|
| `int` | Data type keyword | `;` | Delimiter |
| `main` | Function identifier | `scanf` | Function identifier |
| `(` | Delimiter | `(` | Delimiter |
| `)` | Delimiter | “ | Delimiter |
| `{` | Delimiter | `%d` | |
| `int` | Data type keyword | ” | Delimiter |
| `x` | Variable identifier | , | Delimiter |
| `;` | Delimiter | `&` | Reference Operator |
| `printf` | Function identifier | `x` | Variable Identifier |
| `(` | Delimiter | `)` | Delimiter |
| “ | Delimiter | `;` | Delimiter |
| `Enter a number` | String literal | `return` | Keyword |
| ” | Delimiter | `0` | Number literal |
| `)` | Delimiter | `;` | Delimiter |
| | | `}` | Delimiter |

During lexical analysis, a number of steps are also taken to ensure that the source program will be understood by the succeeding translation processes.

# 1.

Numbers are converted to the translator's internal representation, e.g., binary, fixed- or floating-point form, etc.

# 2.

Detected identifiers are stored in the symbol table.

# 3.

Skipping meaningless blanks and comments.

# 4.

Detecting syntax errors; these errors may terminate the translation process.

In general, lexical analysis **takes the most time** out of all the translation steps because it is **done one character at a time**.

The output of lexical analyzers is the list of tokens and lexemes, which is then used by the syntax analyzer.

# TWO GENERAL WAYS TO MATCH PATTERNS

# 1.

# Regular Expressions
# aka
# Regex

Regular expressions are used to recognize strings that belong to a regular language.

Most PLs are simple enough so that **correct syntax** can be **recognized** by **regular expressions**.

Regular expressions express patterns using a variety of symbols, the most commonly used of which are...

# REGEX SYMBOLS

<table>
<tr><th>Symbol</th><th>Meaning</th></tr>
<tr><td>^</td><td>Starts with; nothing precedes</td></tr>
<tr><td>$</td><td>Ends with; nothing follows</td></tr>
<tr><td>+</td><td>One or more occurrences of the previous regex</td></tr>
<tr><td>*</td><td>Zero or more occurrences of the previous regex</td></tr>
<tr><td>?</td><td>Optional; zero of one occurrence of the previous regex</td></tr>
<tr><td>{3}</td><td>Exactly three occurrences of the previous regex; 3 may be any number</td></tr>
<tr><td>{3, 6}</td><td>Three to six occurrences of the previous regex; 3 and 6 may be any number</td></tr>
<tr><td>{3, }</td><td>Three of more occurrences of the previous regex; 3 may be any number</td></tr>
<tr><td>[0123456789]</td><td>Exactly one character from the list; may be a range, e.g., A-Z, a-z, 0-9</td></tr>
<tr><td>[^0123456789]</td><td>Exactly one character that is not in the list; may also be a range.</td></tr>
<tr><td>(true|false|0|1)</td><td>Exactly one out of the possibilities separated by |.</td></tr>
</table>

Some escape characters mean groups of characters.

# REGEX SYMBOLS

| Symbol | Meaning |
|---|---|
| `.` | Any character |
| `\d` | Any digit (0-9) |
| `\D` | Any non-digit |
| `\s` | White space character (space, tab, newline, etc...) |
| `\S` | Non-whitespace character |
| `\w` | Word character (a-z, A-Z, 0-9, _) |
| `\W` | Non-word character |

# The following characters need to be escaped:

[ \ ^ $ . | ? * + ( _

The use of `[ ]` specify **character classes**. Inside these, the following characters must be escaped:

`^ - ] \`

The list is by no means exhaustive; the list of regex symbols is too vast to be explained all in one sitting.

Regular expressions take a **string of symbols** as **input** and attempts to **match** it to a **pattern**.

# Strings either **match** or do not **match**.

# EXAMPLE

The following regex recognizes numbers (both integers and floating-points):

```
(-|\+)?\d+\.?\d*
```

# EXAMPLE: REGEX

```
(-|\+)?\d+\.?\d*
```

Optional – or +

One or more digits

Optional . character

Zero or more digits

# EXAMPLE: STRINGS

| String | Matches (Y/N)? |
| --- | --- |
| 1.25 | |
| -55.55 | |
| 216 | |
| +132.5 | |
| FFFFFF | |

# EXAMPLE: REGEX

What strings does this regex recognize?

```
(0[1-9]|[1-4][0-9])\s+
[a-zA-Z][a-zA-Z0-9\-_]*
\s+PIC\s+(9+(V9+)?|X+)\.
```

# EXAMPLE: REGEX

What strings does this regex recognize?

```
[a-zA-Z][a-zA-Z0-9\-_]\s+=\s+
{\d+(,\d+)*};
```

Not that the **regular expressions** we have discussed are **similar, but different** from the **theoretical regular expressions** discussed in CMSC 141.

# 2.

# State-transition diagrams

The class of state-transition diagrams that we will discuss today is called

**finite automata (FA).**

# FAs are made up **states** and **transitions**.

# States

are represented by **circles** in the diagram and are usually **named** with a **number** or a **concise description**.

There is exactly one

# start state,

denoted by a triangle on its side beside the state.

There can be any number of

# final states,

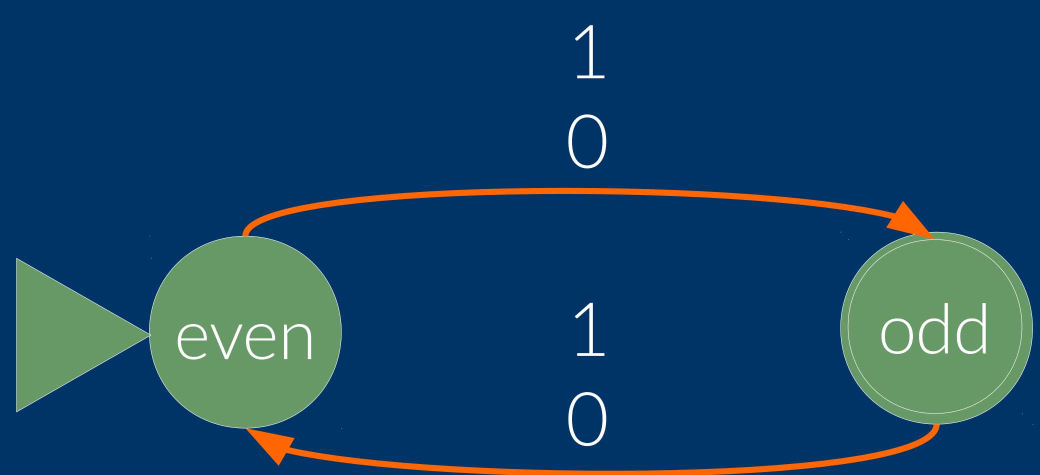
denoted by two concentric circles.

Arrows represent

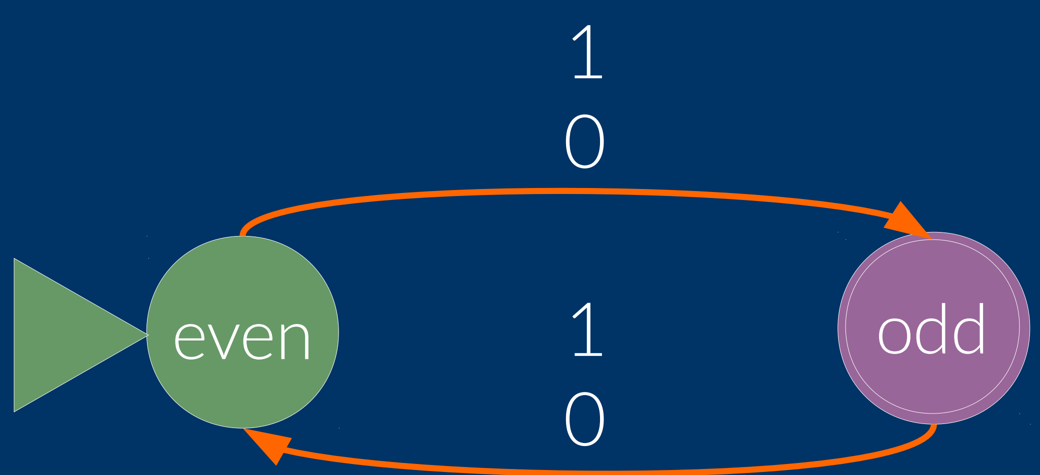
# transitions,

from one state to another.

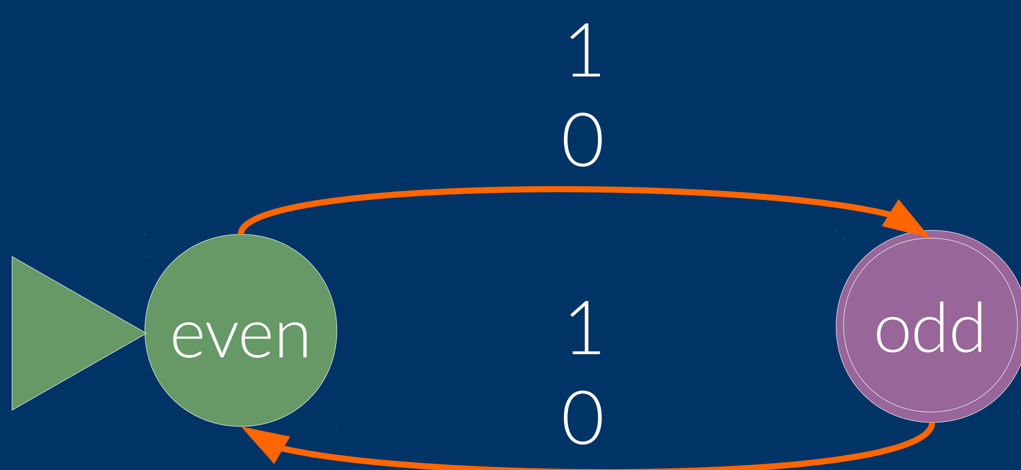
Transitions are associated with a symbol that is required for the transition to be traversed.



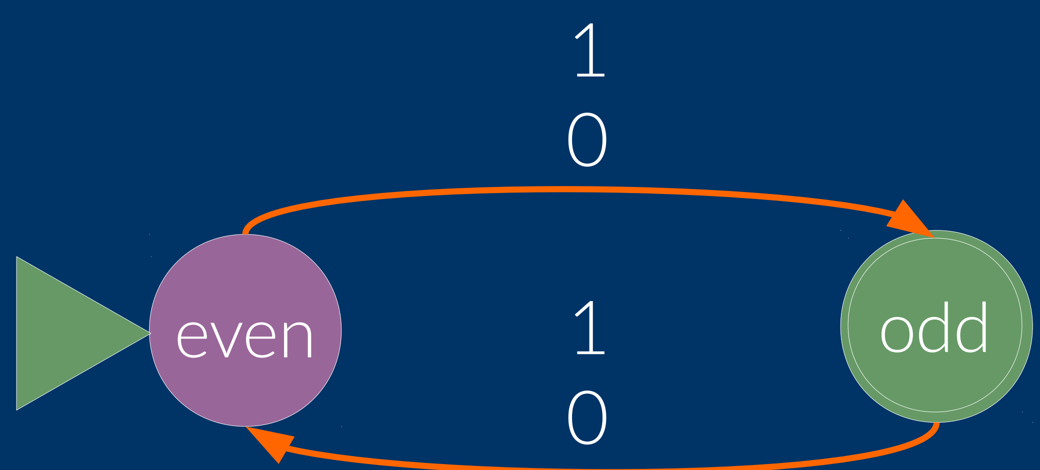
Strings are either accepted (input string ends at accept state) or rejected (otherwise).



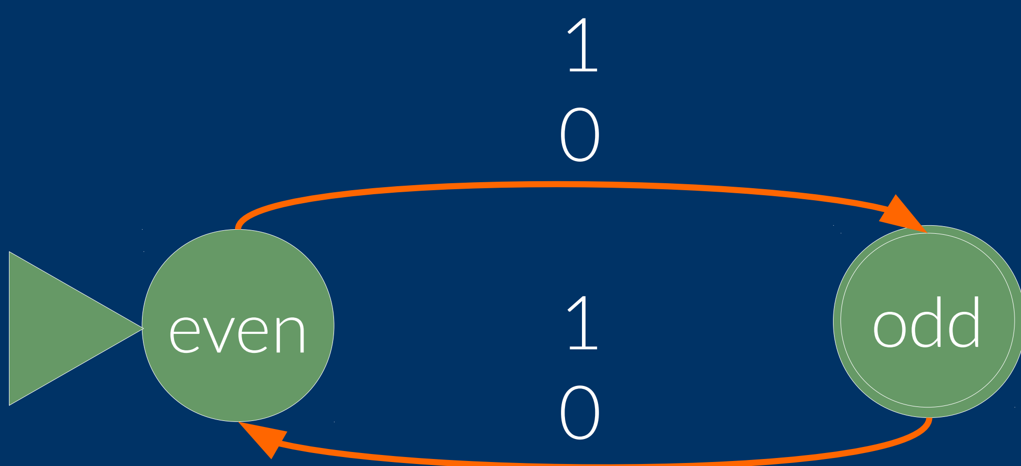
# Strings are either accepted, when the input string ends at accept state, or...



# ...rejected, when the input string does not end at an accept state.

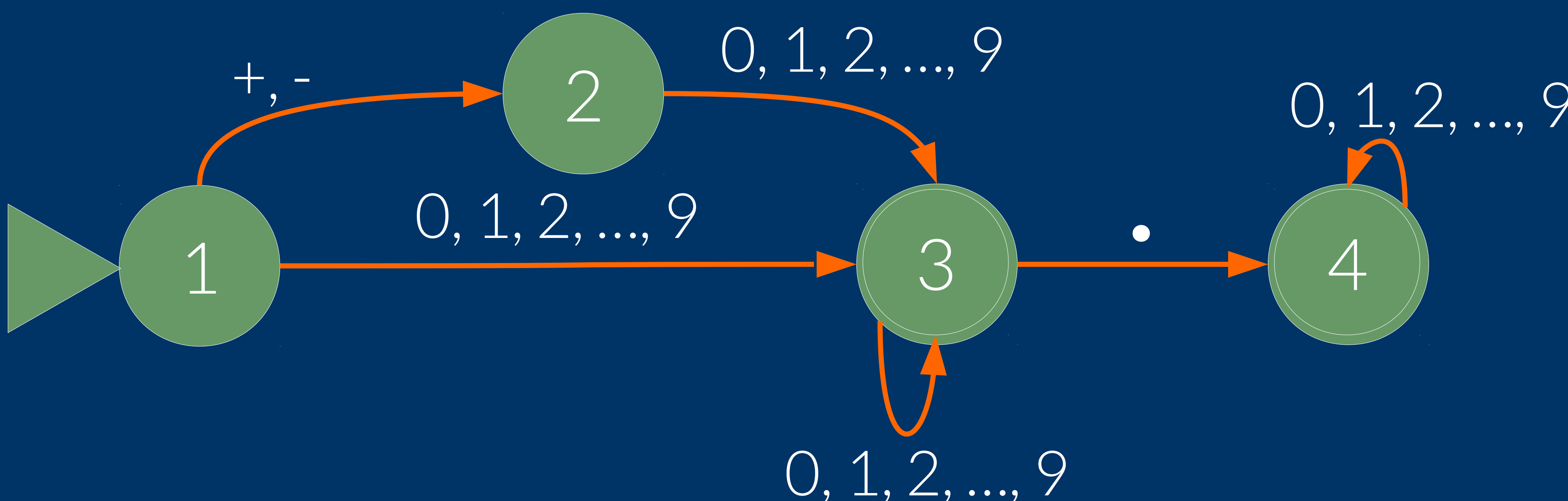


# What does this finite automaton do?



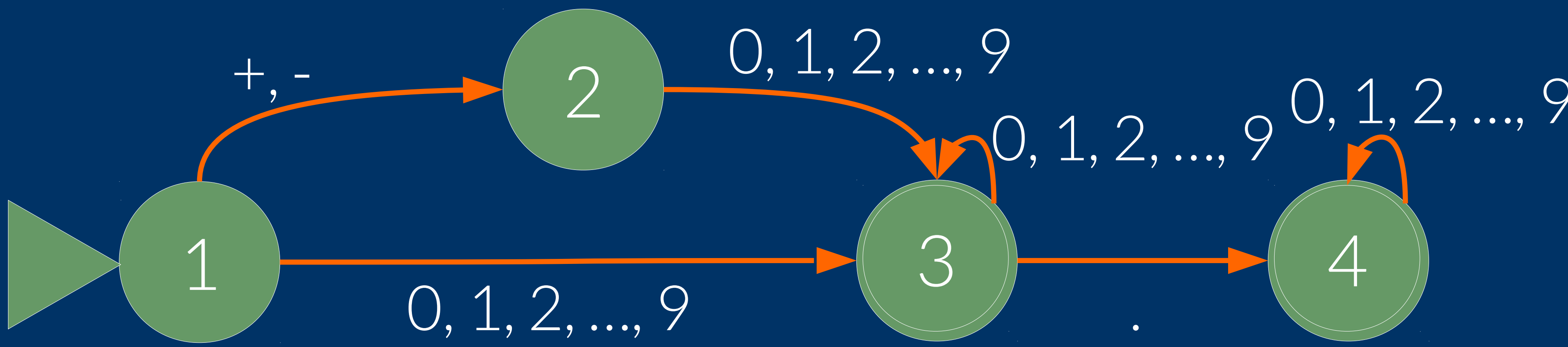
# EXAMPLE

What does this finite automaton recognize?



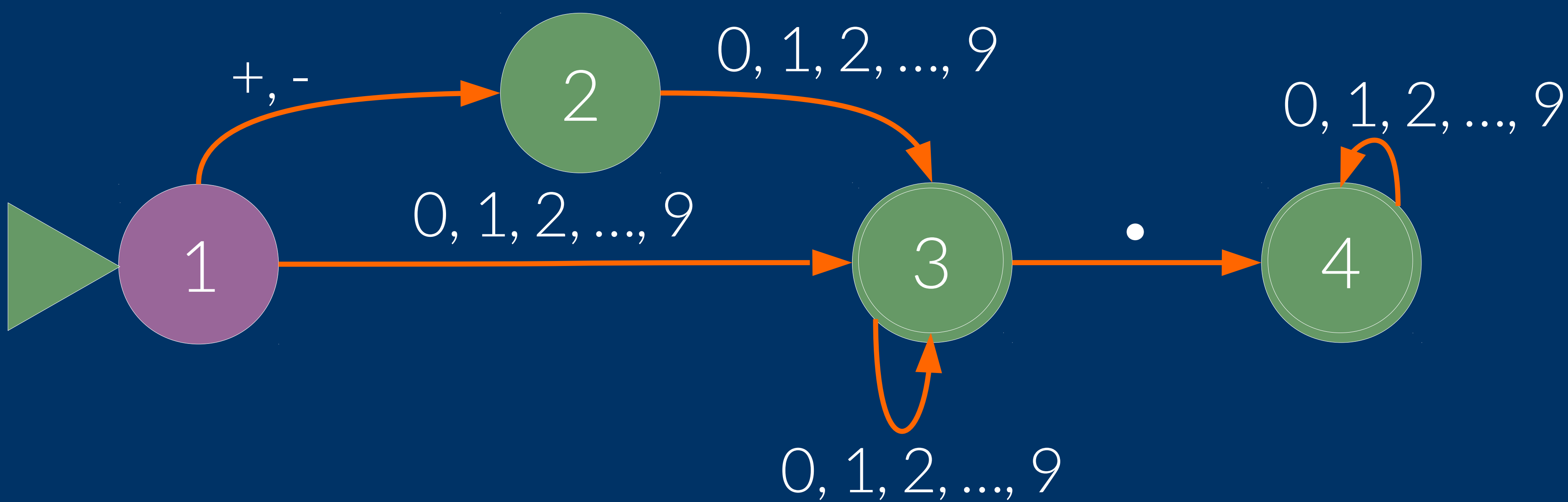
# EXAMPLE

Given the string -12.54...



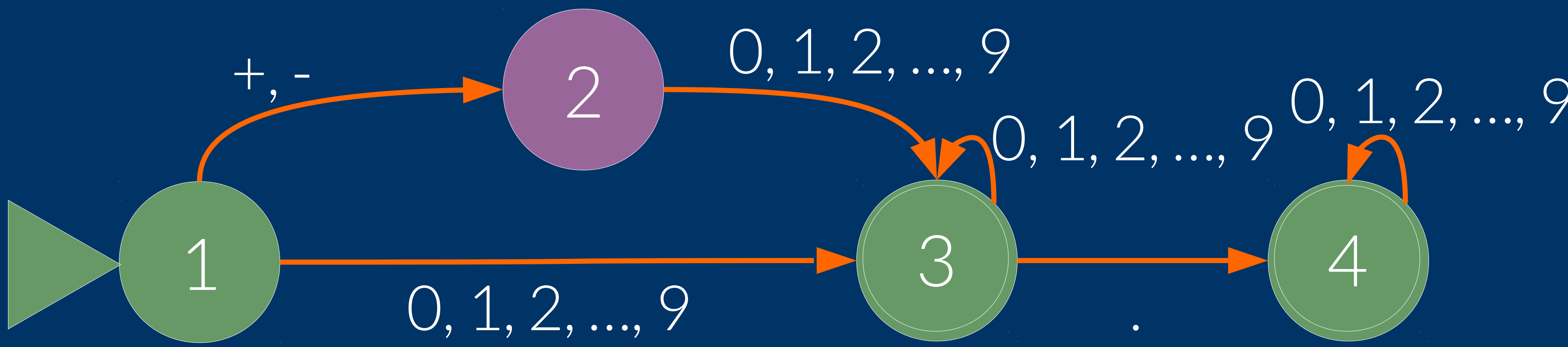
# EXAMPLE

Given the string -12.54...



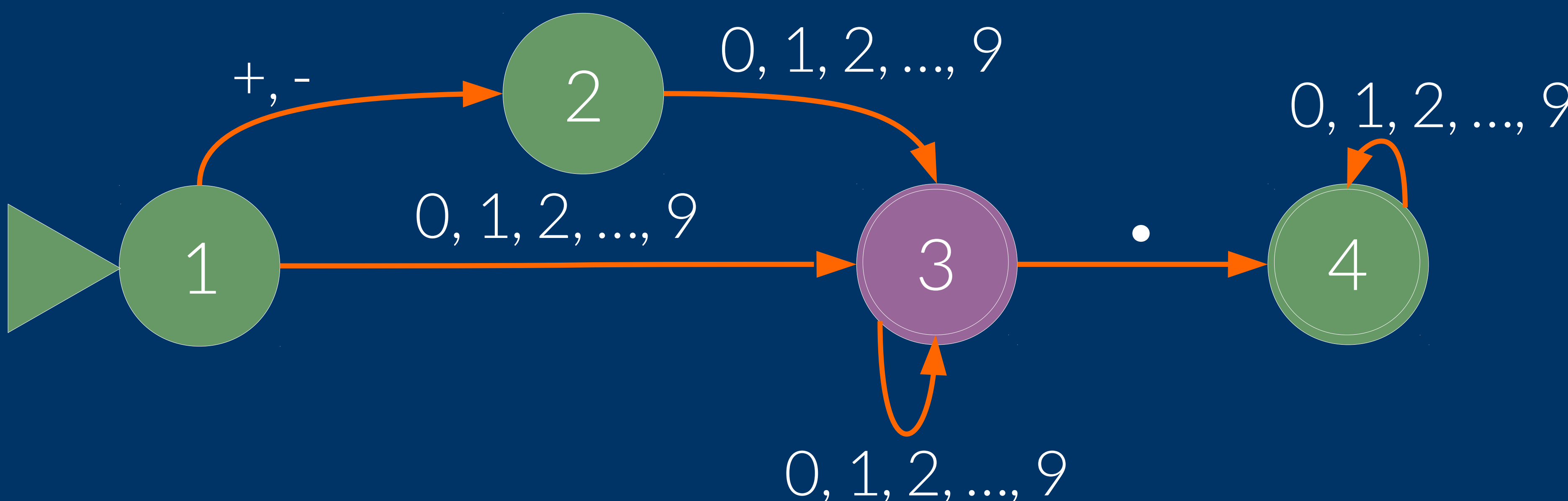
# EXAMPLE

Given the string -12.54...



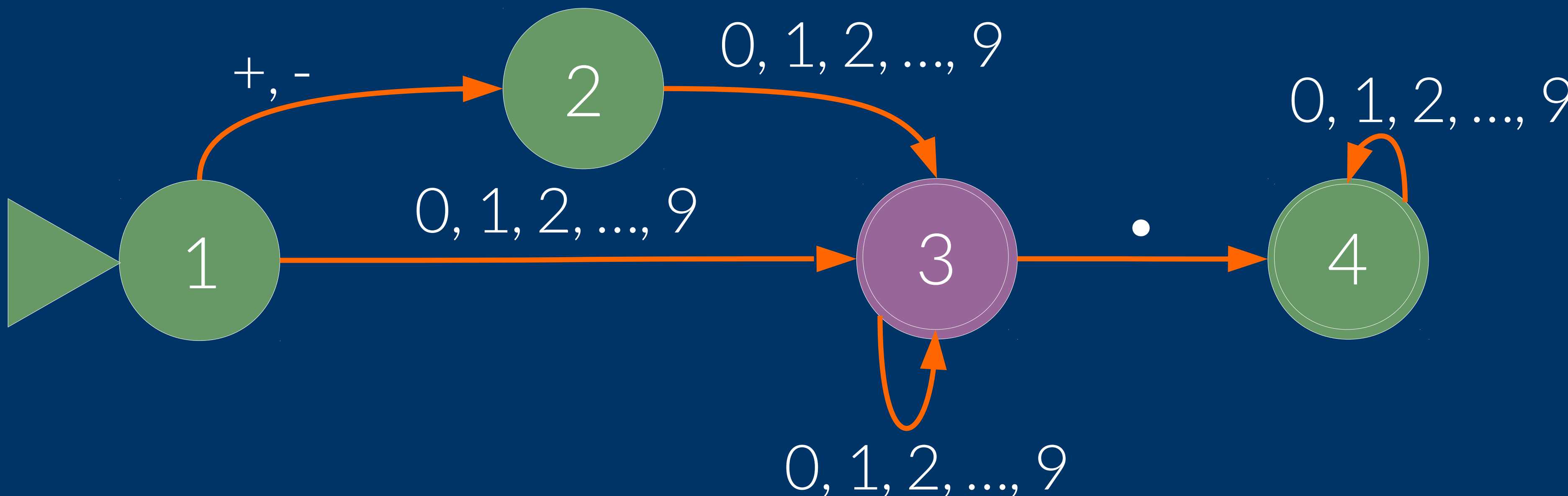
# EXAMPLE

Given the string -12.54...



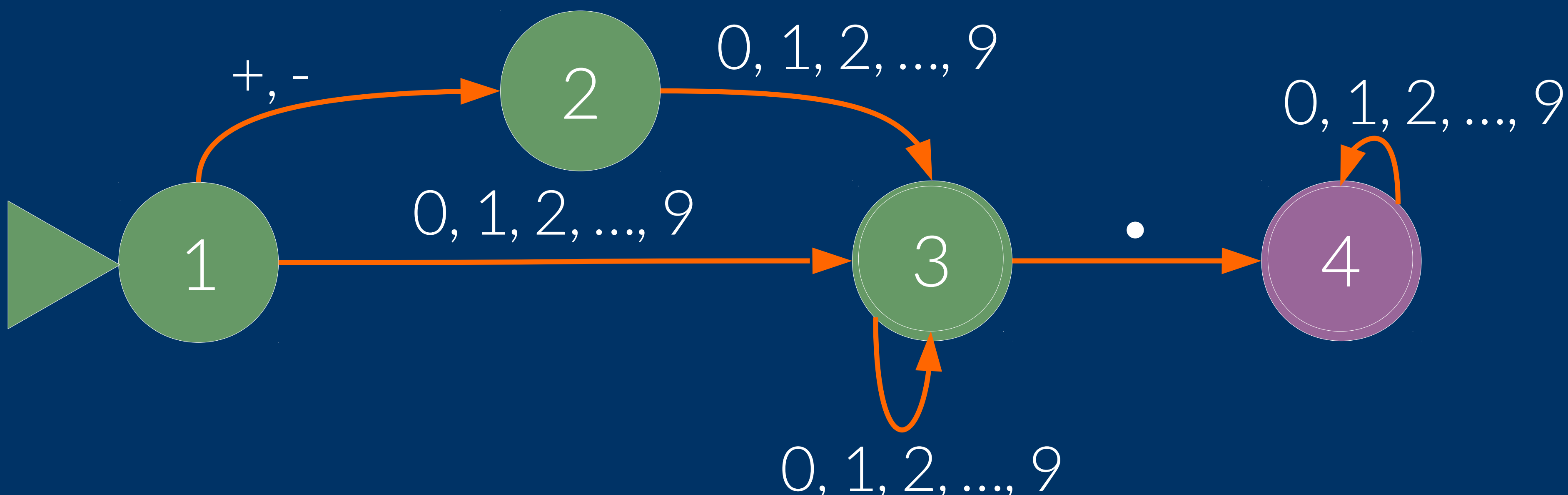
# EXAMPLE

Given the string -12.54...



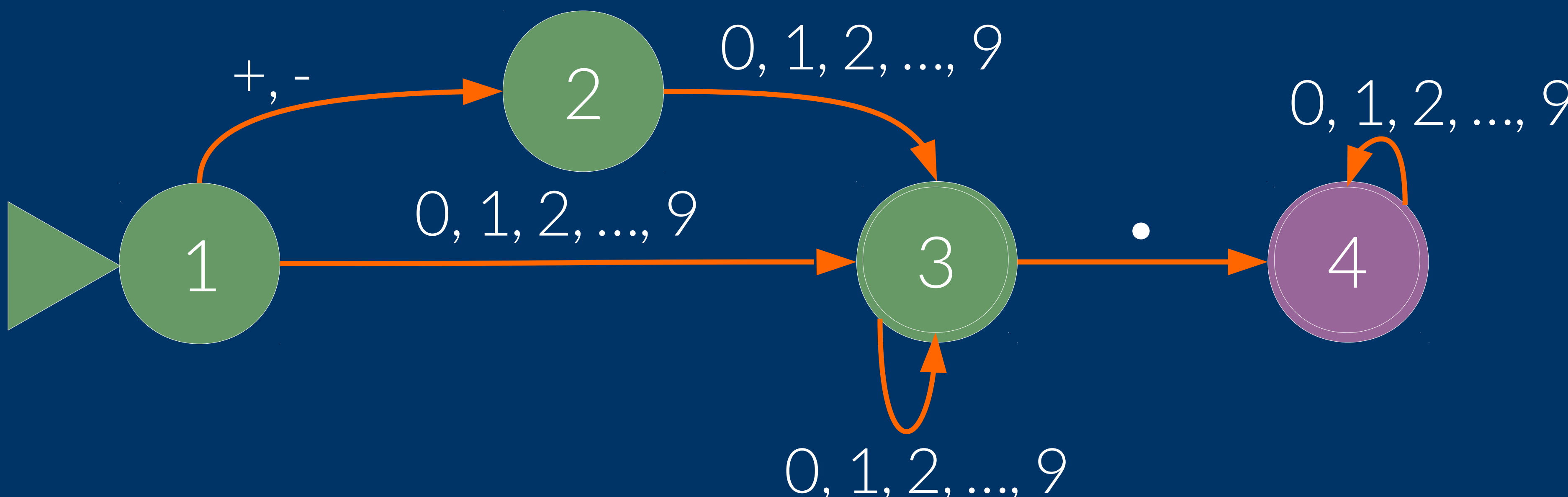
# EXAMPLE

Given the string -12.54...



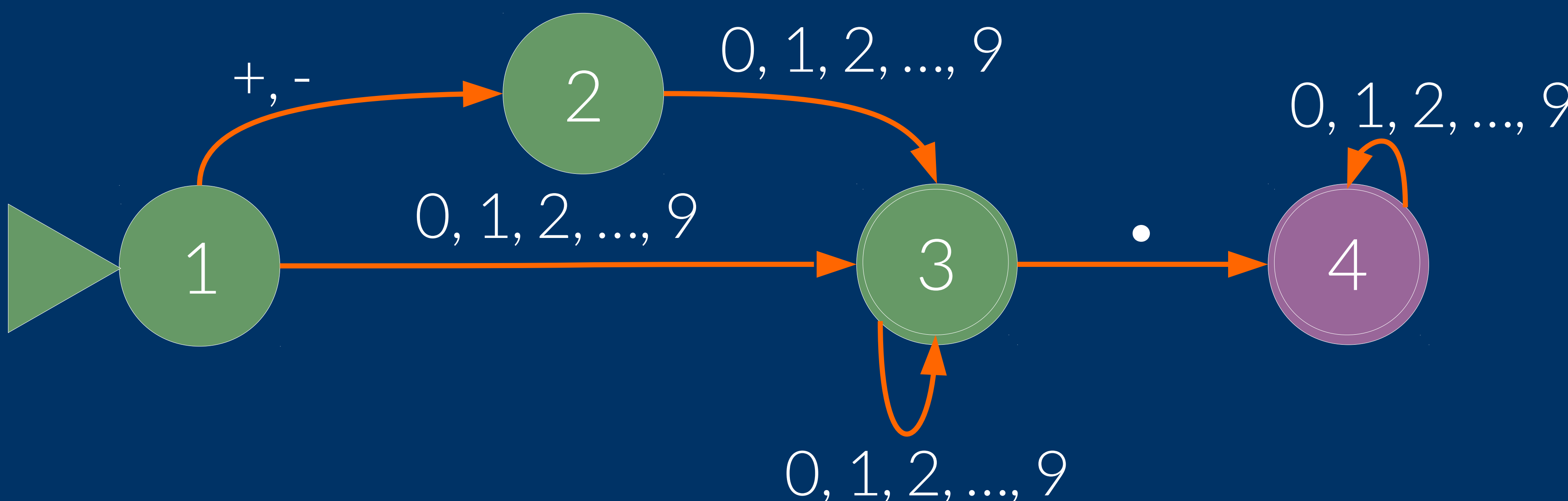
# EXAMPLE

Given the string -12.54...



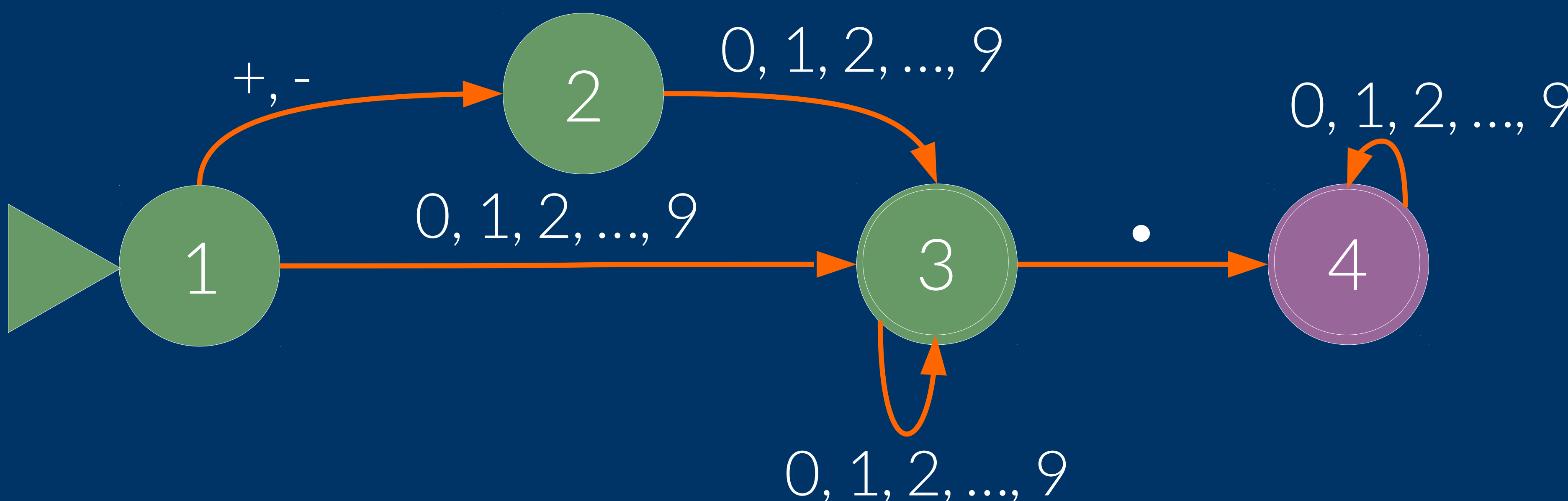
# EXAMPLE

Given the string -12.54...



# EXAMPLE

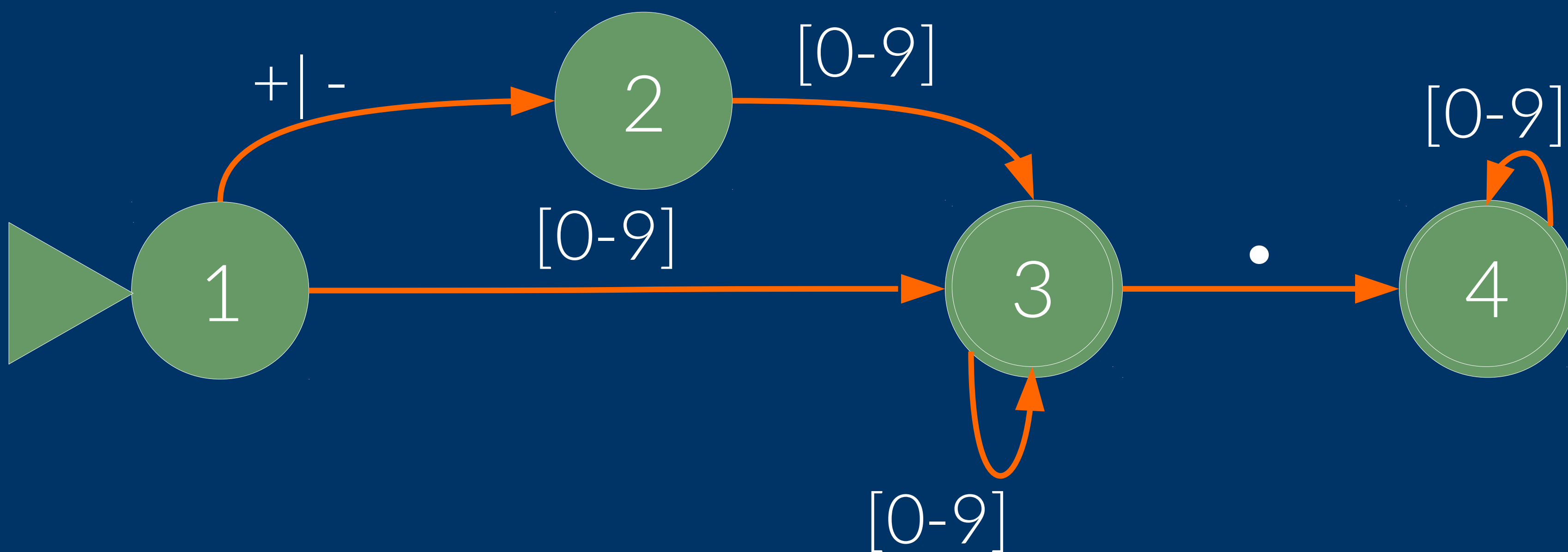
-12.54 ended up in an accept state (4); therefore, the automaton accepts it.



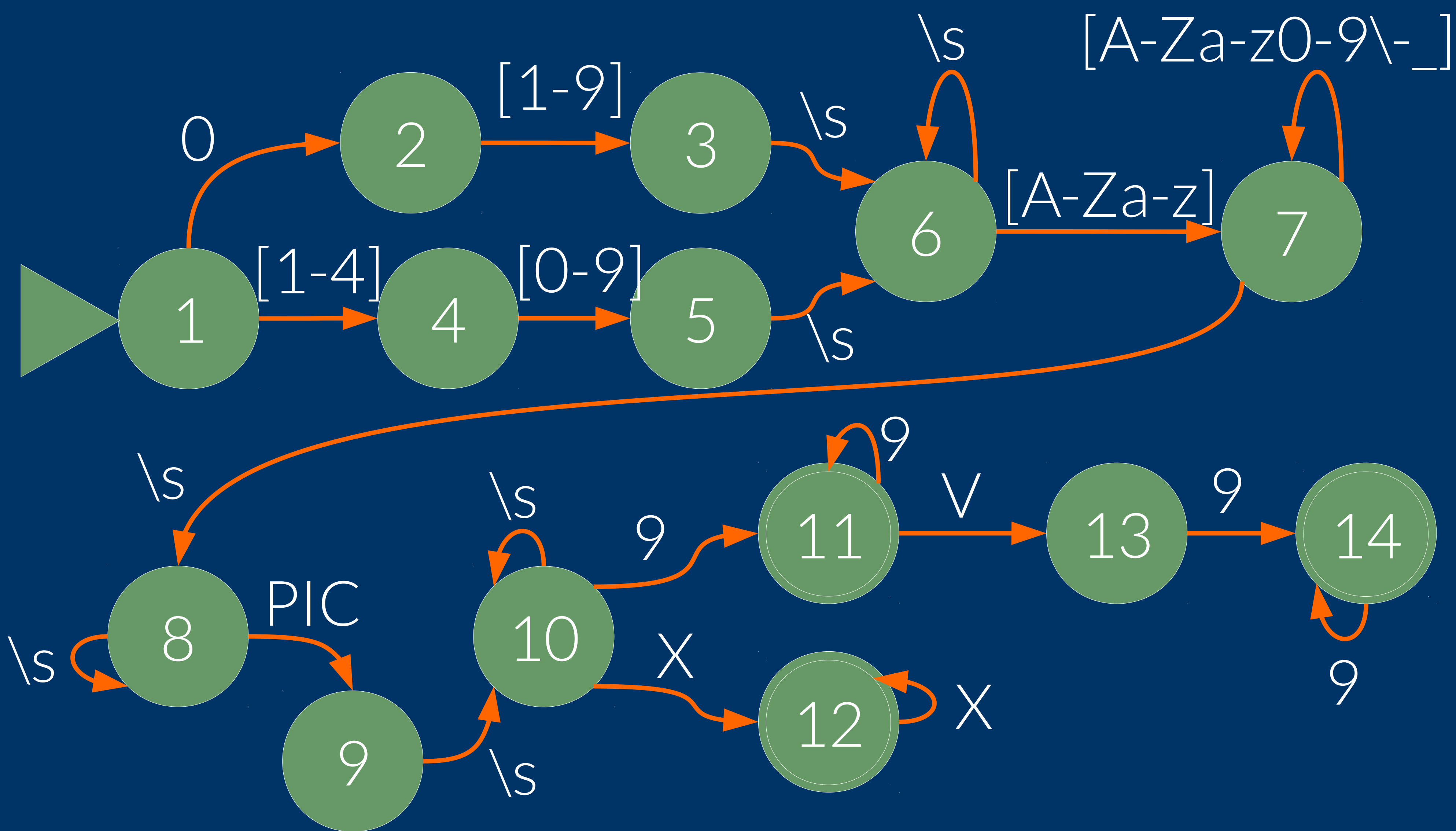
To make **transitions** more **concise**, we can use **regular expressions as transitions**.

Disclaimer: Do not try this in CMSC 141.

# EXAMPLE



# EXAMPLE



\s
[A-Za-z0-9\-_]
[1-9]
0
2
3
\s
6
[A-Za-z]
7
1
[1-4]
4
[0-9]
5
\s
9
\s
11
V
13
9
14
\s
8
PIC
10
9
X
12
X
\s
9
\s
9

Finite automata are equivalent to regular expressions in recognizing regular languages.

Thus, for every regular expression there is an equivalent finite automaton.

# SYNTAX ANALYSIS

The second stage of translation is called

**syntax analysis** or **parsing**.

Larger program structures are recognized by constructing parse trees from the lexemes produced by the lexical analyzer.

The **main task** of syntax analysis is to generate a **complete parse tree** for the **entire source program**.

However, in order to do that, we need to discuss parse trees and the theory behind them.

We have already discussed regular expressions and state-transition diagrams, both of which are **language recognition mechanisms**.

This time, we will look at **grammars**.

# Grammars

are language generation mechanisms used to describe syntax.

The most commonly used grammars for PLs are **Backus-Naur Form** and **Context-Free Grammars**.

# Backus-Naur Form

also known as Backus Normal Form and often abbreviated to BNF is just a notation for context-free grammars.

BNF and CFG were developed by John Backus and Noam Chomsky independently in the 1950s.

# GRAMMARS

Grammars are made up of **rules** or **productions**.

Each rule has a left-hand side (LHS) and right-hand side (RHS) separated by an arrow (CFG) or ::= (BNF).

$$LHS \rightarrow RHS$$

LHS ::= RHS

# Left-hand sides

are abstractions (single, non-terminal variables) that are defined by their corresponding right-hand sides.

# Right-hand sides

can contain both non-terminal symbols/variables and terminal symbols.

Multiple RHS definitions are separated by pipes (|).

Grammars always start with a

**start symbol/variable.**

$$S \rightarrow RHS$$

It is usually the variable on the LHS of the first rule.

**Derivation** or **generation**

is used to yield sentences by applying the rules of a grammar repeatedly.

# EXAMPLE

<assign> → <var> = <expr>;

<var> → A | B | C

<expr> → <operand> <op> <operand>

<op> → + | - | * | /

<operand> → <var> | <digit>

<digit> → 0 | 1 | 2 | … | 9 | <digit><digit>

To **derive strings** from a grammar, repeatedly **replace** either the **leftmost non-terminal always xor**[1] the **rightmost non-terminal always**, **starting** from the **start symbol**.

[1] Either leftmost or rightmost, **but not both**.

# EXAMPLE

<assign>

<var> = <expr>;

A = <expr>;

A = <operand> <op> <operand>;

A = <digit> <op> <operand>;

A = 4 <op> <operand>;

A = 4 * <operand>;

A = 4 * <var>;

A = 4 * C;

# EXAMPLE

<assign>

<var> = <expr>;

<var> = <operand> <op> <operand>;

<var> = <operand> <op> <var>;

<var> = <operand> <op> C;

<var> = <operand> * C;

<var>= <digit> * C;

<var> = 4 * C;

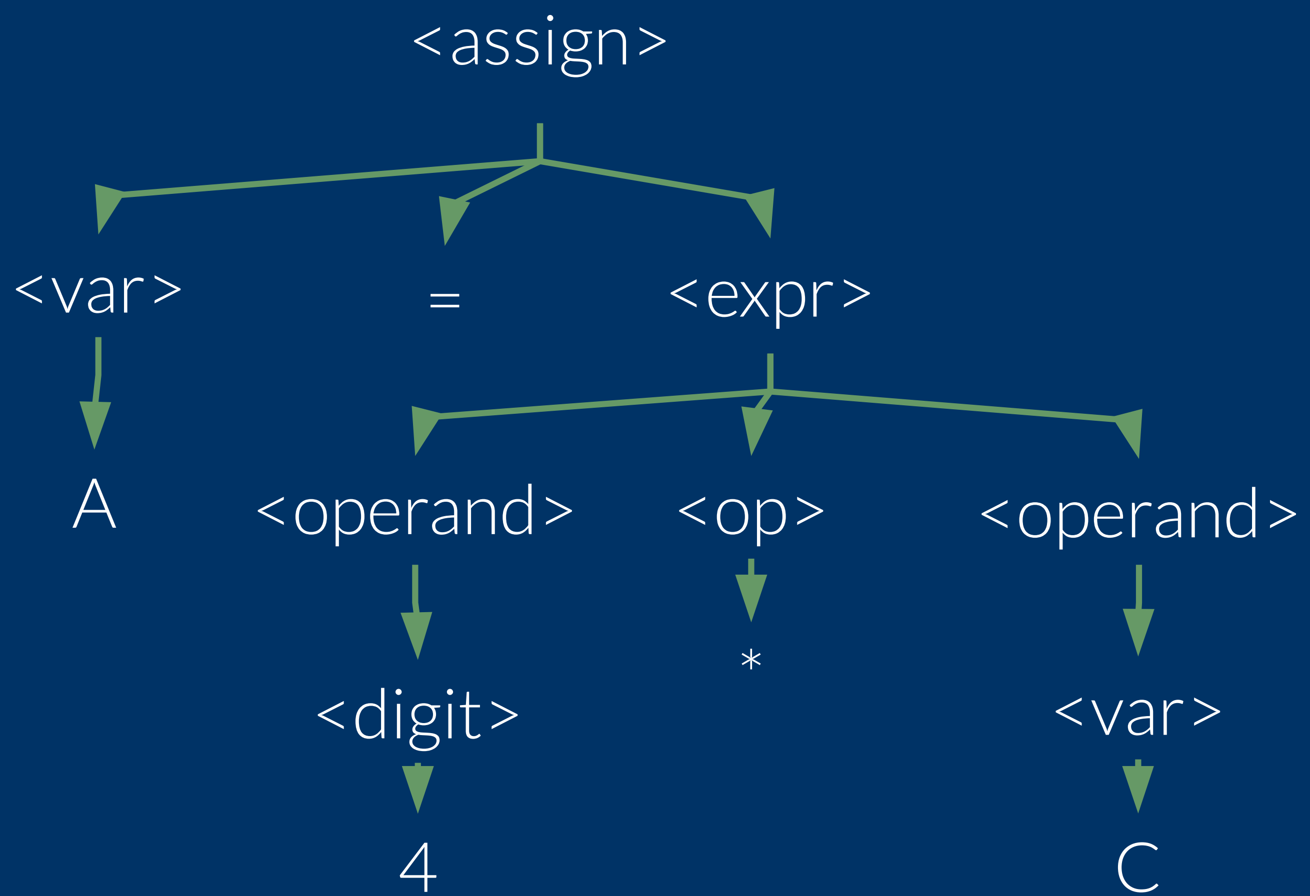
A = 4 * C;

Derivation order has no effect on the language generated by the grammar.

Each **derivation** has a **corresponding parse tree.**

Whether leftmost or rightmost derivation is used, the parse tree should be the same.

# EXAMPLE



We have already introduced ambiguity in a previous topic, but we will now define it a bit more formally.

# Ambiguity

means that a **grammar generates** the **same string** with **two or more distinct parse trees.**

# EXAMPLE

The grammar for the dangling-else problem introduced before is:

<stmt> → if <cond> then <stmt> |

if <cond> then <stmt> else <stmt> |

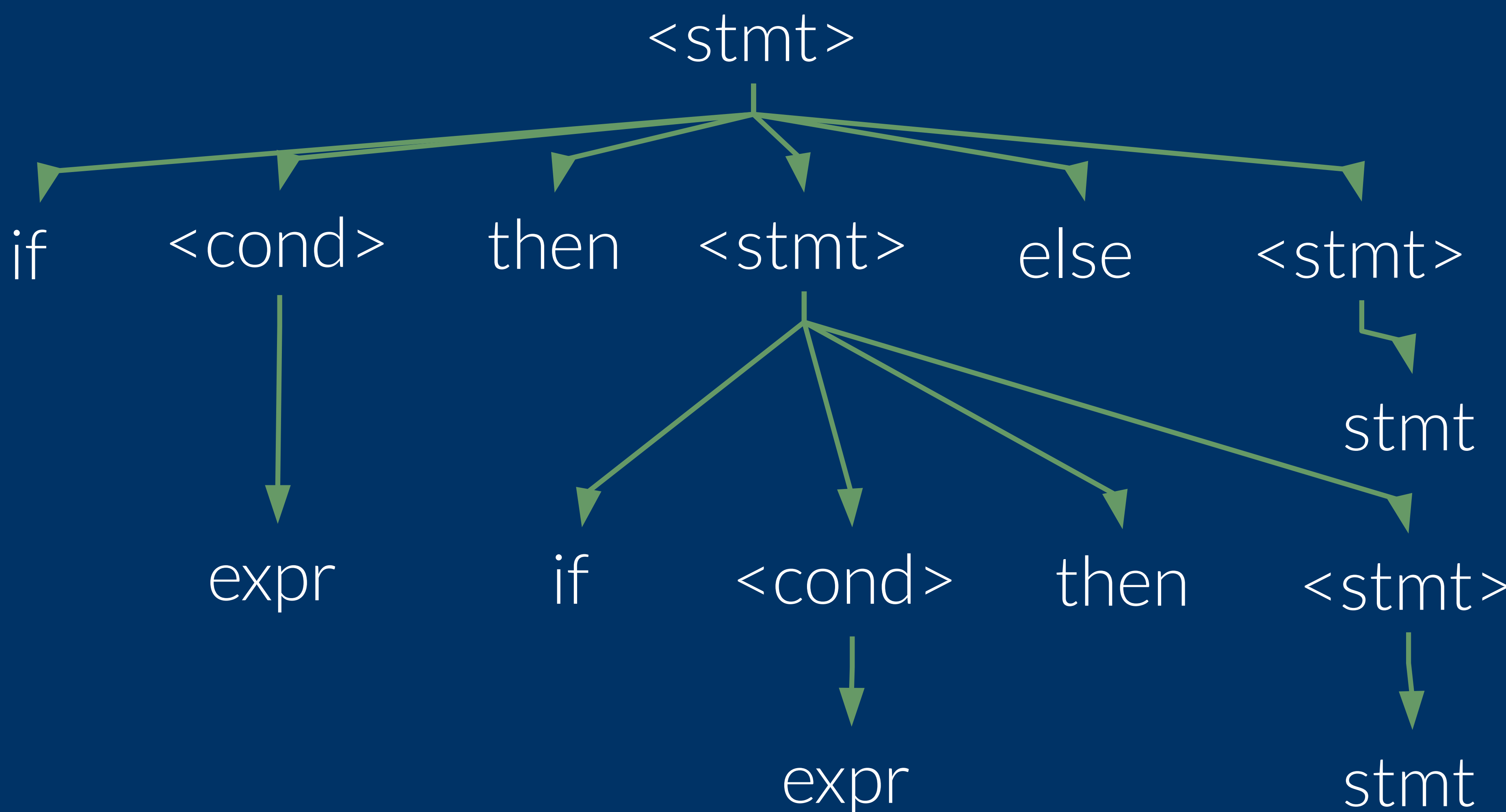
stmt

<cond> → expr

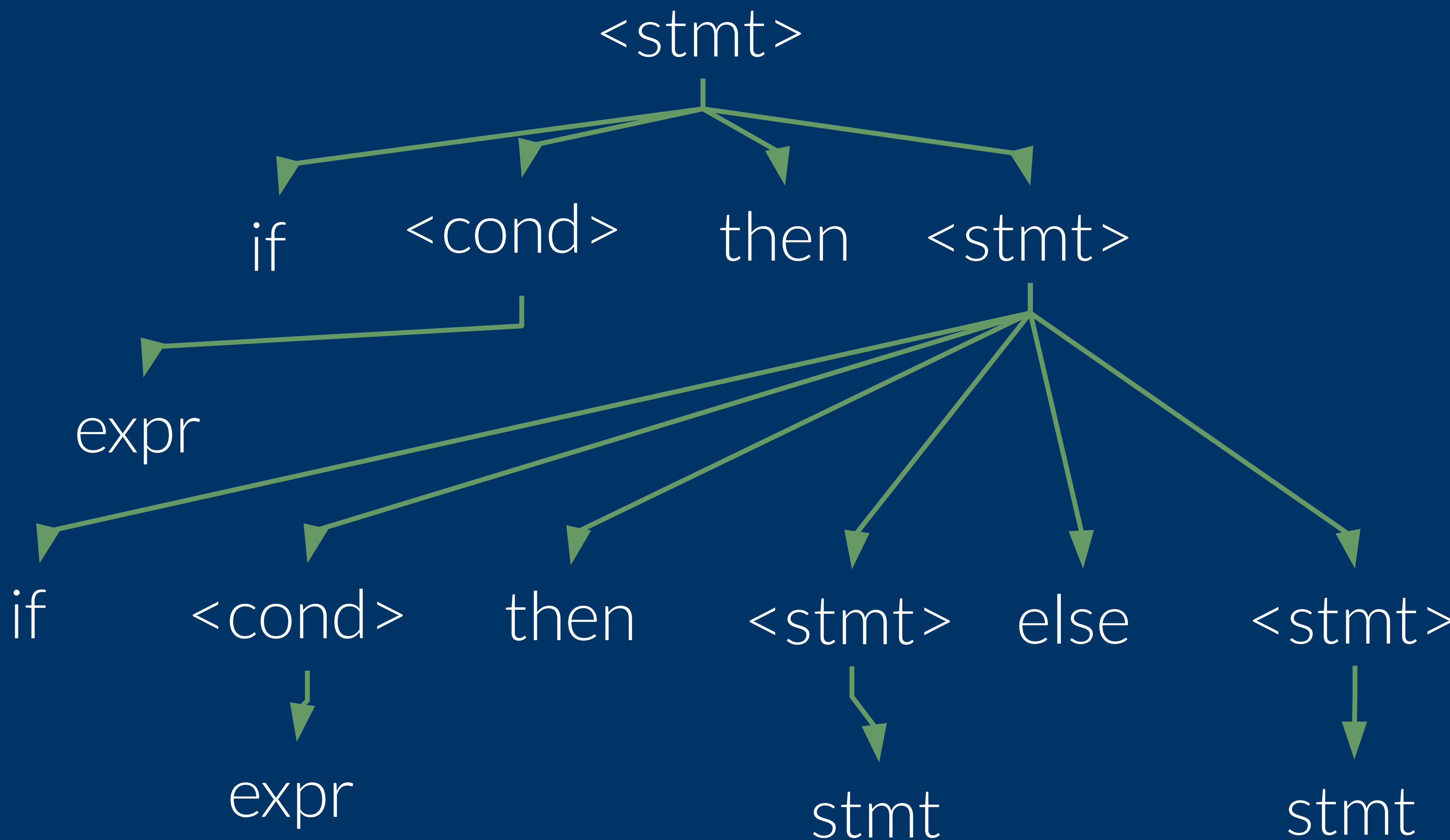
# EXAMPLE

```
if expr(1) then if
  expr(2) then
    stmt(1)
else
    stmt(2)
```

# EXAMPLE


<stmt>
if
<cond>
then
<stmt>
else
<stmt>
expr
if
<cond>
then
<stmt>
stmt
expr
stmt

# EXAMPLE



<stmt>
if
<cond>
then
<stmt>
expr
if
<cond>
then
<stmt>
else
<stmt>
expr
stmt
stmt

# EXAMPLE

There are **no specific rules to remove ambiguity**; ambiguous grammars are simply **rewritten** to attempt to remove their ambiguity.

# EXAMPLE

<stmt> → if <cond> then begin <stmt> end |

if <cond> then begin<stmt> end

else begin<stmt> end |

stmt

<cond> → expr

# EXAMPLE

<stmt> → if (<cond>) then { <stmt> } |

if (<cond>) then {<stmt> }

else {<stmt> } |

stmt

<cond> → expr

# QUIZ

Given the grammar:

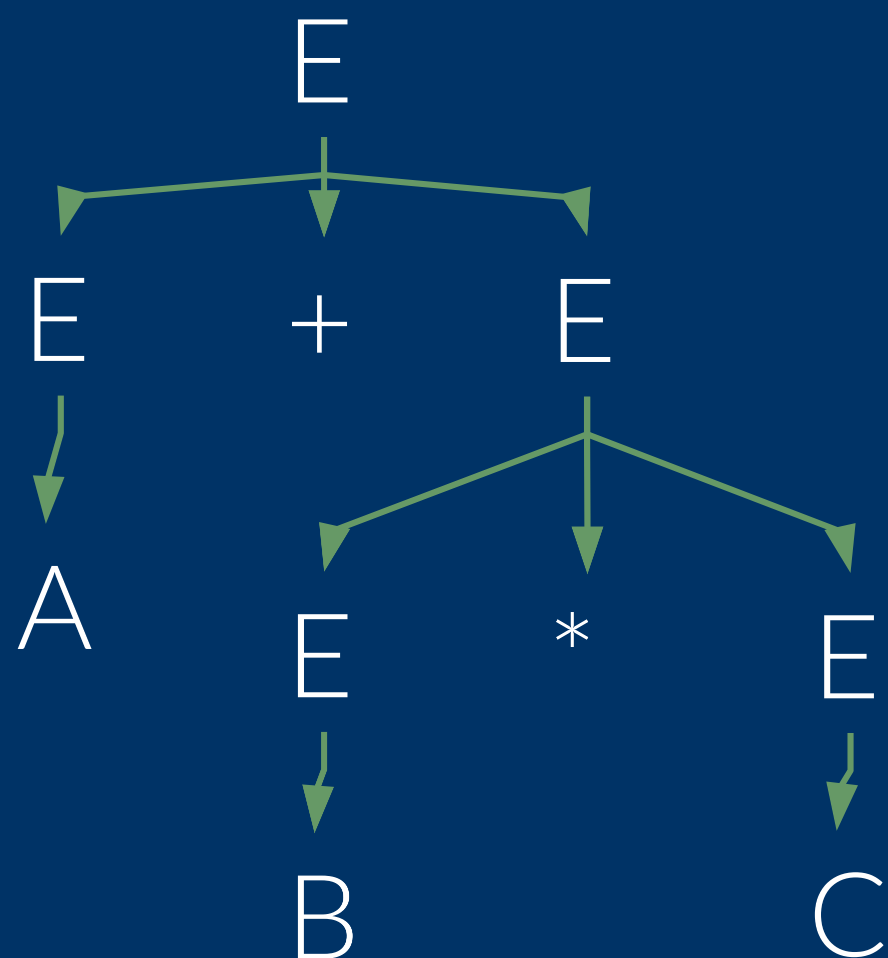
$E \rightarrow E + E \mid E * E \mid (E) \mid A \mid B \mid C$

(1) Give all possible distinct derivations of the string A + B * C by replacing non-terminal symbols with their corresponding RHS.

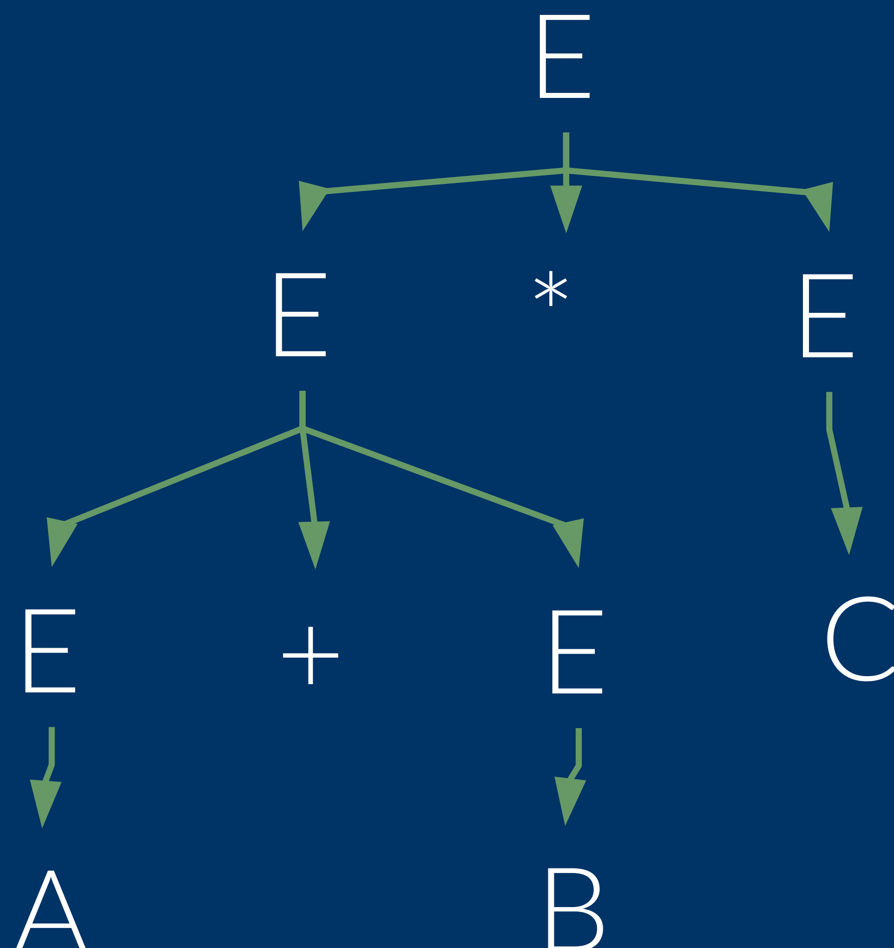
(2) Give each derivation's parse tree.

| (A) | (B) |
|---|---|
| E | E |
| E+E | E*E |
| E+E*E | E+E*E |
| E+E*C | A+E*E |
| E+B*C | A+B*E |
| A+B*C | A+B*C |

# ANSWERS (A)



# ANSWERS (B)



In grammars concerning mathematical operations we can apply concepts of operator precedence and associativity to remove ambiguity.

# Operator precedence

specifies the order in which operations should be executed. It is usually remembered by the mnemonic PEMDAS.

The **expression** generated **lower** in the parse tree is **evaluated first** and thus have **higher precedence**.

# EXAMPLE

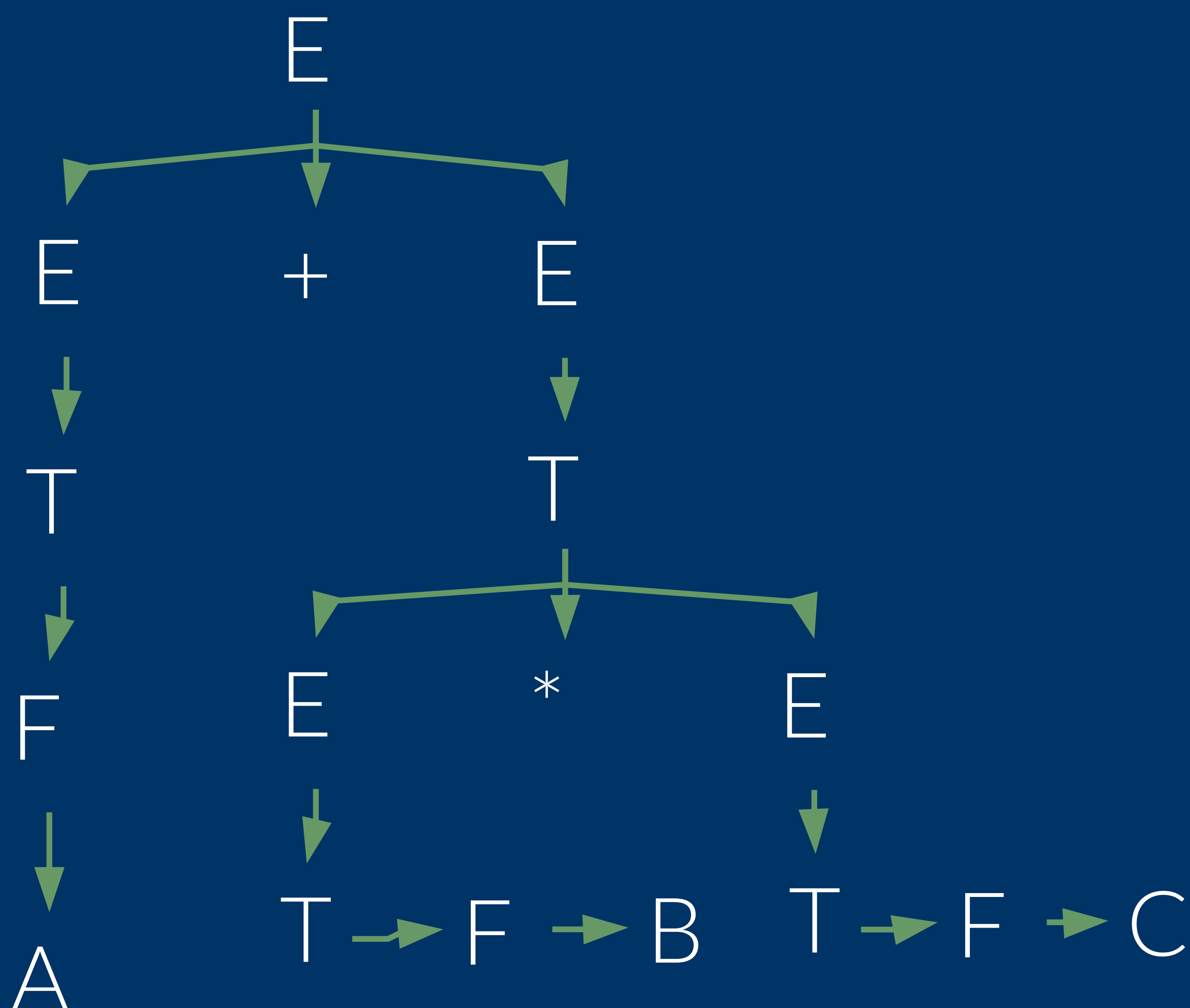
$E \rightarrow E+E|T$

$T \rightarrow T*T \mid F$

$F \rightarrow (E) \mid id$

$id \rightarrow A \mid B \mid C$

# EXAMPLE: A+B*C



# Operator associativity

determines which operation among operations of equal precedence must be evaluated first.

# EXAMPLE

The addition operation is associative:

$$A + B + C = (A + B) + C = A + (B + C)$$

Left-associative

Right-associative

# Left associativity

means operations of equal precedence are evaluated left to right; grammars must be left-recursive.

# Left-recursive rules

have non-terminal symbols occur at the beginning of the RHS.

Examples of left-associative operations are **addition**, **subtraction**, **multiplication**, and **division**.

# Right associativity

means operations of **equal precedence** are evaluated **right to left**; grammars must be **right-recursive**.

# Right-recursive rules

have non-terminal symbols occur at the end of the RHS.

An examples of a right-associative operation is exponentiation.

We can force associativity by rewriting our grammar.

# EXAMPLE

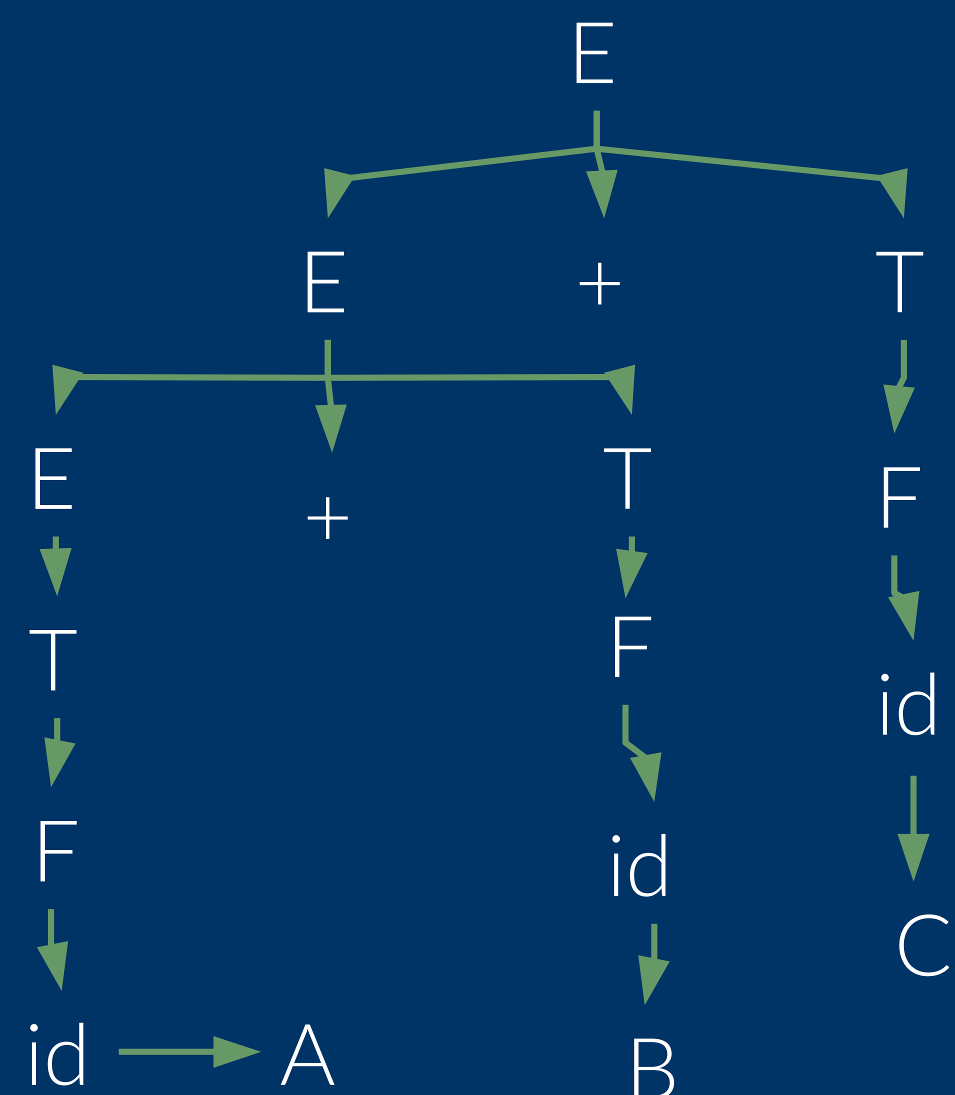
$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

$id \rightarrow A \mid B \mid C$

# EXAMPLE: A+B+C



# EXAMPLE

$E \rightarrow \boxed{E + T \mid E - T} \mid T$

$T \rightarrow \boxed{T * F \mid T / F} \mid F$

$F \rightarrow \boxed{G \wedge F} \mid G$

$G \rightarrow (E) \mid id$

$id \rightarrow A \mid B \mid C$

Left-recursive rules (LHS occurs at start of rules)

Right-recursive rule (LHS occurs at the end of the rule)

# There are two kinds of parsers.

# 1.

# Top-down parsers

build parse trees from **root to leaves** (start symbol $\rightarrow^*$ string) using **preorder traversal**.

Top-down parsers use **leftmost derivations** to arrive at the target string.

The **most common** top-down parsers are called

**recursive-descent parsers.**

# 2.

# Bottom-up parsers

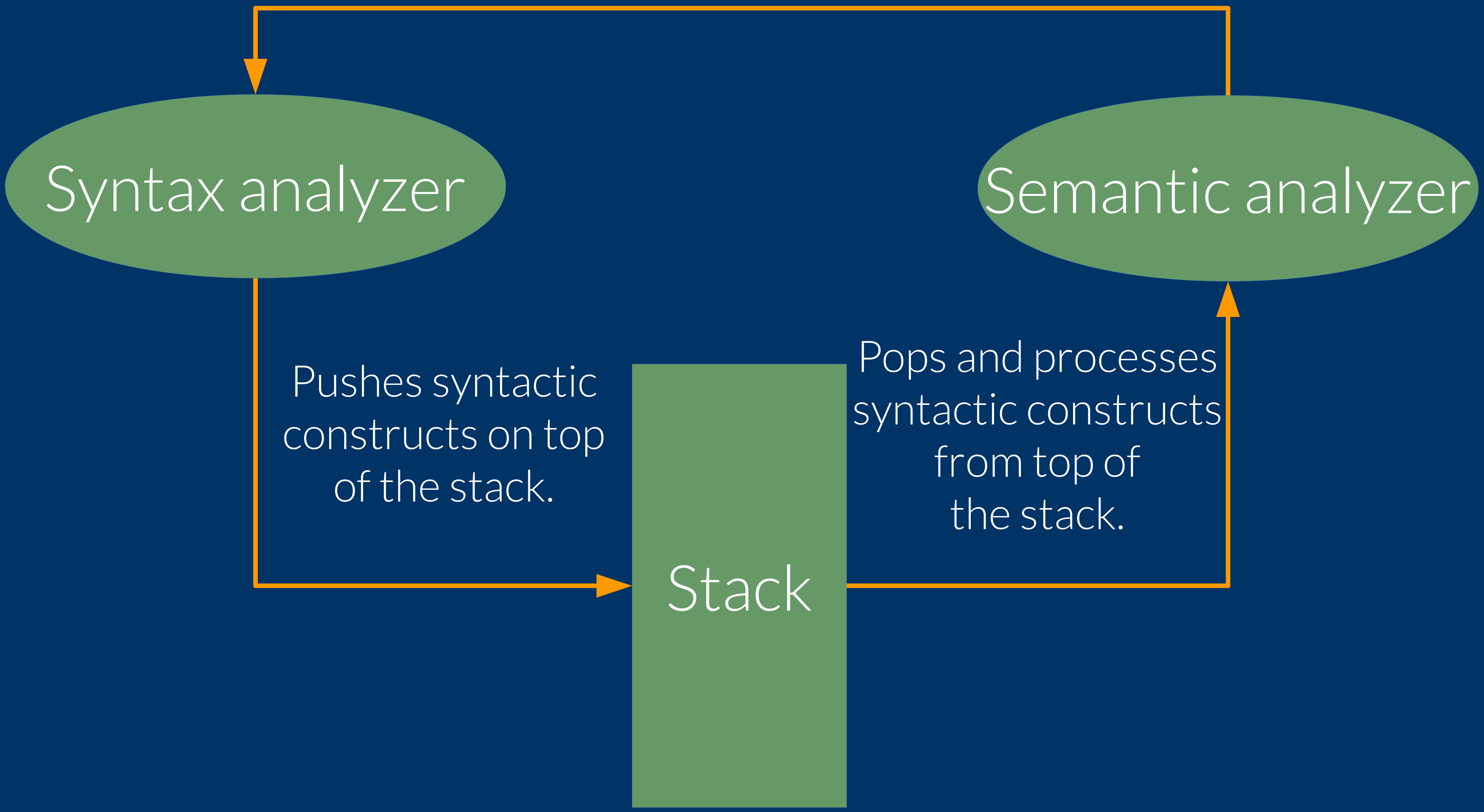
build parse trees from **leaves to root** (string $\rightarrow^*$ start symbol).

Bottom-up parsers use **reverse rightmost derivations**.

The **most common** bottom-up parsers are called

# *shift-reduce parsers.*

Syntactic analyzers alternate execution with semantic analyzers.


Syntax analyzer
Semantic analyzer
Pushes syntactic constructs on top of the stack.
Stack
Pops and processes syntactic constructs from top of the stack.

# SEMANTIC ANALYSIS

Semantic analysis is considered the

**central phase**

of translation.

Syntactic constructs recognized during syntactic analysis are processed.

**Semantic analysis** serves as the **bridge between** the **analysis** of the source program and the **synthesis** of the object program.

Semantic analyzers are broken down per syntactic construct.

# EXAMPLE

## Semantic Analyzer

- SemA for if-else
- SemA for for-loops
- SemA for variable declarations
- SemA for while-loops
- SemA for switch
- SemA for user-defined types

# The common functions of semantic analysis are:

# 1.

# Symbol table maintenance

# Symbol tables

are the **central data structure** during translation.

During syntax analysis, each different identifier encountered is entered into the symbol table.

During semantic analysis, the values of the identifiers are updated in the symbol table as they are used in the source program.

Information about each identifier may also be included in the symbol table, some of which may be...

# 1.1.

## Type of identifier

Simple variable, array, subprogram, user-defined data type, etc.

# 1.2.

## Type of value

Integer, real, other data types

# 1.3.

# Referencing environment

Global, local, etc.

Symbol tables are usually discarded after translation, except when identifiers can be defined during run-time. (e.g., LISP, Prolog, etc.)

# 2.

# Insertion of implicit information

# EXAMPLE

Some languages initialize static variables to 0 if no value is explicitly given.

# EXAMPLE

```
static int x; //the value 0 is
              //implicitly
              //assigned to x
```

# 3.

# Error detection

Semantic analyzers must recognize errors and be able to continue in spite of them.

# 4.

# Macro processing and compile-time operations

# Macros

are separately-defined program text that is inserted into the program during translation when a macro call is encountered in the source program.

# EXAMPLE

```
#define L 50-5
switch(num) {
  case L: //stmts
    break;
  case L*2: //stmts
    break;
  case L*3: //stmts
    break;
}
```

```
#define L 50-5
switch(num) {
  case 50-5: //stmts
    break;
  case 50-5*2: //stmts
    break;
  case 50-5*3: //stmts
    break;
}
```

# Compile-time operations

are operations performed during translation to **control the translation** of the source program.

# EXAMPLE

```
#define pc
...
#ifdef pc
system(cls);
#else
system(clear);
#endif
```

Depending on the value in the `#define` macro, translation chooses one of the two system calls. The version not chosen is discarded.

# SYNTHESIS OF THE OBJECT PROGRAM

# Object program synthesis is done in three steps.

# 1.

# Optimization

Semantic analyzers usually output poor code, which can be improved by optimization.

# WHY?

Semantic analysis focuses on **one syntactic unit at a time**, **disregarding context** (surrounding code).

# Some improvements that optimization can do are:

# 1.1.

Computing common sub-expressions only once.

# EXAMPLE

```
if(a < 5 == 0) {
  b = 0;
  b = a * 2;
} else {
  b = 0;
  b = a / 2;
}
```

```
b = 0;
if(a < 5 == 0) {
  b = a * 2;
} else {
  b = a / 2;
}
```

# 1.2.

Removing constant operations from loops.

# EXAMPLE

```
for(i = 0; i < n;
    i++) {
  a[i]=abs(rand())
       %50;
  x = 0;
}
```

```
x = 0;
for(i = 0; i < n;
    i++) {
  a[i]=abs(rand())
       %50;
}
```

# 1.3.

## Optimizing the use of registers

# 1.4.

Optimizing the calculation of array-accessing formulas.

# 2.

# Code generation

Derives object program code from the output of semantic analysis/optimization.

Almost final executable form.

# 3.

# Linking and loading

Required when subprograms are separately translated.

Object program code may contain references to external data or subprograms.

# Loader tables

contain the locations of external references in the object program code.

# Linking loaders/link editors

read the loader tables and fill in subprogram addresses as needed in the specified locations.