Documentation:

Astra Programming Language

Principles of Programming Language

Submitted by:

Cube, Jeremy
Constantino, Bismillah
Jizmundo, Piolo Brian
Rosario, Mark Edison
Tacata, Jericho Vince

BS Computer Science 3 - 3

Submitted to:

Prof. Elias A. Austria

Polytechnic University of the Philippines
Sta. Mesa, Manila

March 2022

I. Description

Astra is a Procedural Programming Language (PPL). It captures all the key features of procedural programming. It is a general-purpose programming language used to develop simple applications.

II. Brief History

Astra was developed by a group of computer science students of the PUP-CCIS. The name astral comes from the late Latin word “astralis” which means “stars”. The developers were inspired by the simplicity of distant stars while incorporating a complicated scientific structure.

The names of developers are:

- Constantino, Bismillah
- Cube, Jeremy
- Jizmundo, Piolo Brian
- Rosario, Mark Edison
- Tacata, Jericho Vince

III. Features

- Astra implements all the key features of a Procedural Programming Language.
- It is easy to understand as it uses common English words.
- It focuses on simple syntax for efficient code reading and writing.

IV. File Types

- .AST - Program Input Source File
- .ASTD - Program Dictionary File
- .ASTL - Program Output Table File

V. Syntactic Elements

- **Character Sets**
  - ➔ Alphanumeric = {uppercase_letters, lowercase_letters, digits}

    where

    uppercase_letters = {A, B, C, D, …, Z}

    lowercase_letters = {a, b, c, d, …, z}

    digits = {0, 1, 2, 3, …, 9}
  - ➔ Symbols = {+, –, *, /, %, ^, >, =, <, !, &, !, _, ., (,), {, }, “,’, ;}
  - ➔ White spaces = {\s, \r, \n, }

- **Identifiers**
  - ➔ **Rules in Naming Identifiers**
    1. Maximum of 16 characters are allowed.
    2. Underscore (_) is the only allowed special character.
    3. It must not be a keyword or reserved word.
    4. It must only start with a letter followed by another letter, digit, or hyphen.
    5. Astra is case-sensitive. Uppercase and lowercase letters are considered distinct characters.

- **Operation Symbols**
  - ➔ **Arithmetic**: +, –, *, /, %, ^
  - ➔ **Relational**: >, >=, <, <=, ==, <>
  - ➔ **Logical**: !, &, |

- **Keywords and Reserved Words**

Category	Reserved words / Keywords
Input	scan
Output	print
Data Types	char
	int
	float

Category	Reserved words / Keywords
	bool
	string
Conditional	if
	then
	else
Repetition	for
	in
	range

- **Noise Words**

  Astra has no noise words to be used and implemented.

- **Comments**

Symbol	Definition
!*	Single line comment
!** … *!	Multiple lines comment

- **Blank/Spaces**

  Astra allows spaces for separating the character, word, expressions, and statements. Blank lines and other white spaces are ignored by the compiler.

- **Delimiters and Brackets**

Symbol	Definition
;	End of every line / separates every statement
()	Enclosing expressions
{}	Enclosing multiple statements
‘ ’	Enclosing a character literal
“ “	Enclosing a string literal

- **Free-and-Fixed Formats**

  Astra follows the fixed format of the following statements.

  Input: x = scan();

  Output: print(const);

  Astra strictly requires the use of a semicolon at the end of every statement.

- **Expressions**

Symbol
Arithmetic
X = A + B;
X = A – B;
X = A * B;
X = A / B;
X = A % B;
X = A ^ B;
Relational
X > Y
X >= 0
(X + Y) < (Y ^ X)
X <= Y * 3
X % Y == Y / X
X – Y <> 3 + Y
Logical
X > Y & X >= 0
((X + Y) < (Y ^ X)) | (X <= (Y * 3))
!(X % Y = Y / X)
Other
ident = scan();

- **Statements**

Category	Reserved words / Keywords	Syntax
Input	scan	ident = scan();
Output	print	print(const); print(exp);
Data Types	char	char a, b;
	int	int x;
	float	float m;
	bool	bool B;
	string	string str;
Relational	!	if (exp1 & exp2) then stmt/s else if(exp1 | exp3) then stmt/s else if(!(exp)) then stmt/s else stmt/s
	&	
	|	
Conditional	if	
	then	
	else	
Repetition	for	• for ident in range(ident) do stmt/s • for const in range(const) do stmt/s
	in	
	range	
	do	

Astra : Lexical Analyzer

Lexical Analyzer, the first part of the compiler for the Astra programming language. The lexical analyzer reads lexemes from an Astra Source File, given by the user, and converts it into meaningful tokens..

The lexical analyzer asks the user to provide a source file. It will then read the source file character by character and convert every lexeme into tokens with appropriate information. Every syntactical token is categorized into an appropriate token type and provided with a description.

The following list are the valid syntactical elements of Astra:

- Operator

+ - * / % ^	Arithmetic Operator
== > >= < <= <>	Relational Operator
! | &	Logical Operator
=	Assignment Operator

- Delimiter and Bracket

;	Terminator
"	Double Quotation
'	Single Quotation
,	Separator
(	Open Parenthesis
)	Close Parenthesis
{	Group Statement - Open Brace
}	Group Statement - Close Brace

- Comment

!*	Single-Line Comment
!**	Multi-Line Comment - Open
*!	Multi-Line Comment - Close
...	Comment Contents

- Constant

(0-9)*	Integer Constant Value
a-z	Character Constant Value
true + false	Boolean Constant Value
(0-9)+ . (0-9)+	Float Constant Value
(a-z + A-Z)*	String Constant Value

- Identifier

ident	Variable 'ident'

- Data Type

int	Integer Data Type
char	Character Data Type
bool	Boolean Data Type
float	Float Data Type
String	String Data Type

- Reserved / Keyword

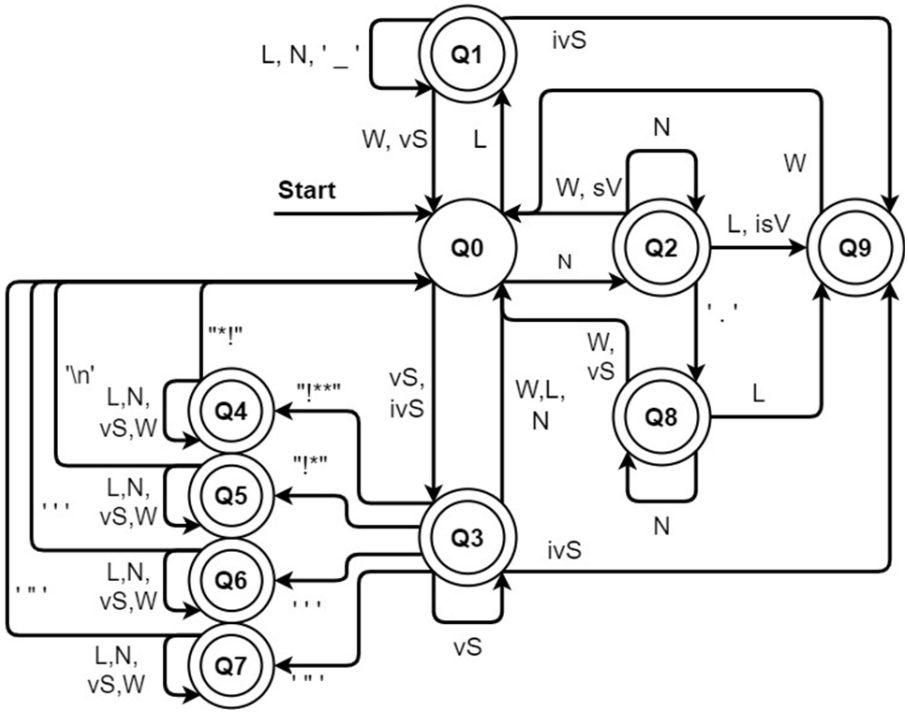
scan	Input Statement
print	Output Statement
if	Conditional Statement
then	Conditional Statement
else	Conditional Statement
for	Looping/Repetition Statement
in	Looping/Repetition Statement
range	Looping/Repetition Statement
do	Looping/Repetition Statement

After a successful conversion of lexemes to tokens, the program will create an astra table file with a file name ‘symbol table’ and it will contain all the tokens along with its information, including the line number, token type, and description.

Lexical Analyzer: Automaton Model

Tuples
M = (Q, ∑, δ, S, F)
Q = {Q0, Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8, Q9}
∑ = { L, N, W, vS, ivS }
s = {Q0}
F = {Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8, Q9}

Input	
L	a-zA-Z
N	0-9
vS	+-*/^%>=<!|&;,(){}\'"
W	whitespaces
ivS	!(vS)



The lexical analyzer is able to convert lexemes into tokens with the use of the automaton. It reads the source file character by character and changes the current state by transitioning to the next state based on the matched state within the automaton.

Astra : Syntax Analyzer

Syntax Analyzer, the second part of the compiler for the Astra programming language. The syntax analyzer uses the array of tokens generated by the Lexical Analyzer.

The syntax analyzer checks the validity of the tokens based on Astra's predefined grammar rules. Using the grammar rule, the syntax analyzer can know and understand which statements are valid. A statement would be a sequence of tokens based on the grammar rules. The syntax analyzer will then read every token and check if the grammar rules are being followed.

The Context-Free Grammar (CFG) of Astra:

G = (V, T, S, P)

where

V = { <stmt>, <stmt'>, <else>, <else>,
<o_stmt>, <dec_stmt>, <ass_stmt>,
<value>, <x>, <exp>, <exp'>, <var>,
<ident>, <const>, <data_type>, <op>
}

T = { ε, {, }, for, in, range, do, if, else, then,
print, (,), ;, =, , , scan, !, <op>, <const>,
<datatype>, <ident>
}

S = { <stmt> }

P = {

<stmt> → { <stmt> <stmt'> }
<stmt'> → ε
<stmt'> → <stmt> <stmt'>
<stmt> → for <ident> in range (<var>) do <stmt>
<stmt> → if (<exp>) then <stmt> <else>
<else> → ε

<else> → else <stmt>

<stmt> → print (<o_stmt>

<o_stmt> → <var>) ;

<o_stmt> → <exp>) ;

<stmt> → <ident> = <value> ;

<stmt> → <data_type> <dec_stmt> ;

<dec_stmt> → <ident> <ass_stmt> <x>

<ass_stmt> → ε

<ass_stmt> → = <value>

<x> → ε

<x> → , <dec_stmt>

<value> → scan ( )

<value> → <exp>

<exp> → (<exp>)

<exp> → ! <exp>

<exp> → <var> <exp'>

<exp'> → ε

<exp'> → <op> <exp>

<var> → <ident>

<var> → <const>

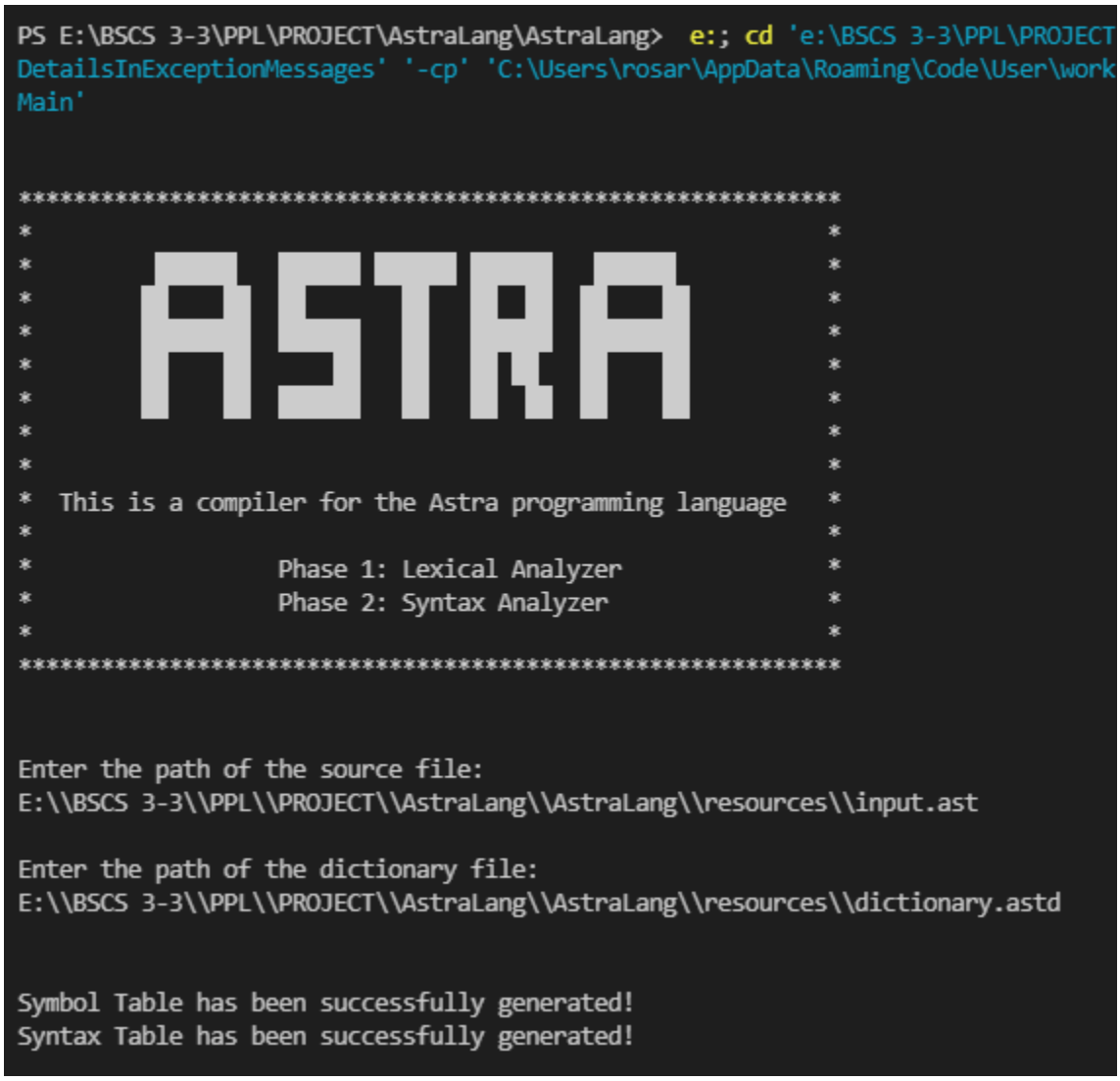
}

The syntax analyzer is able to read tokens and check the validity of the syntax with the help of the grammar rules. It reads each token and matches it with the grammar, it is able to expect a particular token should be next based on what the grammar requires. The statements will remain valid until it encounters a syntax error where it will push an error message and categorize it as an invalid syntax.

After a successful conversion of tokens into statements, the program will create an astra table file with a file name ‘syntax table’ and it will contain all the statements along with its information, including the line number, the syntax, the validity of the syntax, and the error message but only if the syntax is invalid.

Astra : Compiler Program

The image below shows a successful execution of the compiler program which includes the lexical and syntax analyzer.



Sample Astra Source File: input.ast

```
int a = 9, x , y;

String NAME= "astra", age = "21.0" + a, msg = " Hello  -->  World";

char A = 'c' ;

bool is_Even = falseD;

float z = 5.66, w = z + 1.94;

print("Astra");
print( 1 + 2 + 3 + (11 + 10));
a = b;
a = scan();

!**     This a Multi-line Comment 1
        This a Multi-line Comment 2.50

        This a Multi-line Comment true
        This a Multi-line Comment >_<
*!

a = 1 + 1;
x = a * a;
y = a - x / 11;
a = a ^ 5;

if(!hello) then
    print(" More than 5 " + b);         !* This is Single-Line Comment
else
    print(helo);
```

Sample Astra Table File: symbol_table.astl

AstraLang > symbol_table.astl

* LINE# *	* TOKENS *	* LEXEME *	* DESCRIPTION *
Line: 1	int	DATATYPE	Integer Data Type
Line: 1	a	INDENTIFIER	Variable 'a'
Line: 1	=	OPERATOR	Assignment Operator
Line: 1	9	CONSTANT	Integer Constant Value
Line: 1	,	DELIMETER_BRACKET	Separator
Line: 1	x	INDENTIFIER	Variable 'x'
Line: 1	,	DELIMETER_BRACKET	Separator
Line: 1	y	INDENTIFIER	Variable 'y'
Line: 2	;	DELIMETER_BRACKET	Terminator
Line: 3	String	DATATYPE	String Data Type
Line: 3	NAME	INDENTIFIER	Variable 'NAME'
Line: 3	=	OPERATOR	Assignment Operator
Line: 3	"	DELIMETER_BRACKET	Double Quotation
Line: 3	astra	CONSTANT	String Constant Value
Line: 3	"	DELIMETER_BRACKET	Double Quotation
Line: 3	,	DELIMETER_BRACKET	Separator

Sample Astra Table File: syntax_table.astl

AstraLang > syntax_table.astl

```
1   ********        **********                  ************            ************
2   * LINE *        * SYNTAX *                  * VALIDITY *            * MESSAGE *
3   ********        **********                  ************            ************
4
5   L01             int a = 9 , x , y ;         Valid
6   L03             String NAME = " astra " , age = " 21.0 " + a , msg = "  Hello  -->  World " ;       Valid
7   L05             char A = ' c ' ;        Valid
8   L07             bool is_Even = falseD ;         Valid
9   L09             float z = 5.66 , w = z + 1.94 ;         Valid
10  L11             prints ( " Astra " ) ;      Invalid         Unknown Syntax "prints"
11  L12             print ( 1 + 2 + 3 + ( 11 + 10 ) ) ;         Valid
12  L13             a = b ;         Valid
13  L14             a = scan ( ) ;      Valid
14  L23             a = 1 + 1 ;         Valid
15  L24             x = a * a ;         Valid
16  L25             y = a - x / 11 ;        Valid
17  L26             a = a ^ 5 ;         Valid
18  L28             if ( ! hello ) then print ( "  More than 5  " + b ) ; else print ( helo ) ;         Valid
19  L33             for i in range ( 5 ) do { x = x + 6 ; z = x + y ; z = x + y ; z = x + y ; }         Valid
20
```