Cairo University - Faculty of Engineering
Computer Engineering Department
Spring 2024

Languages & Compilers Project Report

Team #12

Team members:

Name	Section	BN
Donia Gameel	1	24
Shaza Mohammed	1	32
Heba Ashraf Raslan	2	32

Project Overview

The project aims to develop a simple programming language compiler using Lex and Yacc. The compiler is designed to support basic programming constructs such as variable declarations, assignments, conditional statements, loops, and print statements. It also includes support for constants and basic arithmetic operations.

Tools and Technologies Used

- Lex: For lexical analysis.
- Yacc: For syntax analysis and parser generation.
- C Programming Language: For implementing the compiler.

Tokens:

Token	Description	Example	Regex
IF	Keyword for if statement	if (condition) { ... }	if
ELSE	Keyword for else statement	if (condition) { ... } else { ... }	else
WHILE	Keyword for while loop	while (condition) { ... }	while
DO	Keyword for do-while loop	do { ... } while (condition);	do
REPEAT	Keyword for repeat-until loop	repeat { ... } until (condition);	repeat
UNTIL	Keyword for repeat-until loop	repeat { ... } until (condition);	until
FOR	Keyword for for loop	for (initialization; condition; update) { ... }	for

SWITCH	Keyword for switch statement	switch (expression) { ... }	switch
CASE	Keyword for case statement	case value: ...	case
DEFAULT	Keyword for default case in switch	default: ...	default
INTEGER	Keyword for integer data type	int variable;	int
FLOAT	Keyword for float data type	float variable;	float
CHAR	Keyword for char data type	char variable;	char
STRING	Keyword for string data type	string variable;	string
BOOL	Keyword for boolean data type	bool variable;	bool
VOID	Keyword for void data type	void function();	void
CONTINUE	Keyword for continue statement	continue;	continue
BREAK	Keyword for break statement	break;	break
CONST	Keyword for const keyword	const int MAX_SIZE = 10;	const
PRINT	Keyword for print statement	print("Hello, world!");	print
ENUM	Keyword for enum declaration	enum Days { MON, TUE, WED };	enum
EQUALS	Operator for equality comparison	if (x == y) { ... }	==
NOT_EQUALS	Operator for inequality comparison	if (x != y) { ... }	!=

LESS_THAN	Operator for less than comparison	if (x < y) { ... }	<

LESS_THAN_OR_EQUALS	Operator for less than or equal comparison	if (x <= y) { ... }	<=
GREATER_THAN	Operator for greater than comparison	if (x > y) { ... }	>
GREATER_THAN_OR_EQUALS	Operator for greater than or equal comparison	if (x >= y) { ... }	>=
LOGICAL_AND	Operator for logical AND	if (x && y) { ... }	&&
LOGICAL_OR	Operator for logical OR	`if (x	
LOGICAL_NOT	Operator for logical NOT	if (!x) { ... }	!
INCR	Operator for increment	x++;	++
DECR	Operator for decrement	x--;	--
;	Semicolon token	statement;	;
,	Comma token	func(arg1, arg2);	,
?	Question mark token	condition ? true : false;	?
:	Colon token	case value: ...	:
=	Assignment operator token	variable = value;	=
(	Left parenthesis token	(...)	(
)	Right parenthesis token	(...)	)
{	Left brace token	{ ... }	{
}	Right brace token	{ ... }	}
[	Left bracket token	[...]	[
]	Right bracket token	[...]	]
+	Addition operator token	x + y;	+
-	Subtraction operator token	x - y;	-
UMINUS	Unary minus operator	-x	handeled in parser
*****	Multiplication operator token	x * y;	*
/	Division operator token	x / y;	/

^	Exponentiation operator token	x ^ y;	^
%	Modulo operator token	x % y;	%
FLOAT_VAL	Float literal token	3.14	[0-9]*\.[0-9]+
INTEGER_VAL	Integer literal token	42	[1-9][0-9]*
CHAR_VAL	Char literal token	'a'	'[^\\']'
STRING_VAL	String literal token	"Hello, world!"	`"([^"\;]
TRUE_VAL	Boolean true literal token	true	true
FALSE_VAL	Boolean false literal token	false	false
IDENTIFIER	Identifier token	variableName	[a-zA-Z_][a-zA-Z0-9_]*

Quadruples:

Quadruple	Description
ADD	Adds two operands and stores the result in a temporary variable.
SUB	Subtracts the second operand from the first operand and stores the result.
MUL	Multiplies two operands and stores the result.
DIV	Divides the first operand by the second operand and stores the result.
POW	Raises the first operand to the power of the second operand and stores the result.
MOD	Computes the remainder of the division of the first operand by the second operand.
LOGICAL_AND	Performs logical AND operation between two operands and stores the result.
LOGICAL_OR	Performs logical OR operation between two operands and stores the result.
EQ	Checks if two operands are equal and stores the result as a boolean value.

NEQ	Checks if two operands are not equal and stores the result as a boolean value.
LT	Checks if the first operand is less than the second operand and stores the result.
LTQ	Checks if the first operand is less than or equal to the second operand.
GT	Checks if the first operand is greater than the second operand and stores the result.
GTE	Checks if the first operand is greater than or equal to the second operand.
NEG	Negates the operand (unary minus operation).
PRE_INCR	Increments the operand before using its value.
POST_INCR	Increments the operand after using its value.
PRE_DEC	Decrements the operand before using its value.
POST_DECR	Decrements the operand after using its value.
PUSH	Pushes a value onto the stack.
POP	Pops a value from the stack into a specified variable.
LABEL:	Marks a position in the code with a label for jumps and calls.
END LABEL	Marks the end of a labeled block or function.
CALL LABEL	Calls a labeled block or function.
JMP LABEL	Unconditionally jumps to the specified label.
JZ LABEL	Jumps to the specified label if the top of the stack is zero (false).