SER-502

KIDDO-LANG

Github:https://github.com/unallami/SER502-KiddoLang-Team2

Team 2

- Uma Maheswar Reddy Nallamilli
- Shreya Prakash
- Venkata Srinivas Babu Oguri
- Parth Rakesh Patel

Why Kiddo-lang ?

“WHY CAN’T PROGRAMMING BE FUN AND FRIENDLY?”

Kiddo-Lang was built to make programming feel approachable, especially for young learners and first-time coders. It focuses on clarity, creativity, and confidence.

- Friendly Keywords: Uses natural, child-friendly words like say, set, yes, and no.
- Minimal Syntax: No scary symbols —just expressive code.
- Learning-Focused: Encourages experimentation and logical thinking.
- Welcoming for All: Designed for kids, but delightful for anyone starting out.

Key-Highlights

Beginner-Friendly Syntax:
Uses plain English-like commands (`set`, `say`, `when`) to ease the learning curve.

Essential Programming Building Blocks:

- Variables with intuitive syntax
- Conditionals via `when`, `otherwise`, and ternary expressions
- Loops with readable patterns like `repeat until` and `count from`
- Built-in support for math and logic operations

Designed for Fast Learning:

Ideal for kids, beginners, or anyone new to coding, easy to write, easier to understand.

Grammar

```
program        : statement+ ;
statement
    : assignment
    | printStatement
    | ifStatement
    | loopStatement
    | ternaryExpr ';'
    ;
assignment     : SET ID TO expr SEMI ;
printStatement  : SAY expr SEMI ;
ifStatement     : WHEN LPAREN expr RPAREN block (OTHERWISE block)? ;
loopStatement   : forLoop | whileLoop ;
forLoop         : COUNT FROM expr TO expr block ;
whileLoop       : REPEAT UNTIL LPAREN expr RPAREN block ;
ternaryExpr     : expr QMARK expr COLON expr ;
block           : LBRACE statement+ RBRACE ;
```

```
expr
    : expr MULT expr
    | expr DIV expr
    | expr PLUS expr
    | expr MINUS expr
    | expr GT expr
    | expr LT expr
    | expr EQ expr
    | expr AND expr
    | expr OR expr
    | NOT expr
    | LPAREN expr RPAREN
    | ID
    | INT
    | FLOAT
    | YES
    | NO
    | STRING
    ;
```

●●●●

```
//Keywords
SET            : 'set' ;
TO             : 'to' ;
SAY            : 'say' ;
WHEN           : 'when' ;
OTHERWISE      : 'otherwise' ;
COUNT          : 'count' ;
FROM           : 'from' ;
REPEAT         : 'repeat' ;
UNTIL          : 'until' ;
AND            : 'and' ;
OR             : 'or' ;
NOT            : 'not' ;
YES            : 'yes' ;
NO             : 'no' ;
```

```
// --- Operators & Punctuation ---
PLUS           : '+' ;
MINUS          : '-' ;
MULT           : '*' ;
DIV            : '/' ;
GT             : '>' ;
LT             : '<' ;
EQ             : '==' ;
QMARK          : '?' ;
COLON          : ':' ;
LPAREN         : '(' ;
RPAREN         : ')' ;
LBRACE         : '{' ;
RBRACE         : '}' ;
SEMI           : ';' ;
```

```
// --- Identifiers & Literals ---
ID             : [a-zA-Z_][a-zA-Z0-9_]* ;
INT            : [0-9]+ ;
FLOAT          : [0-9]+ '.' [0-9]+ ;
STRING         : '"' (~["\\] | '\\' .)* '"' ;

// --- Whitespace & Comments ---
WS             : [ \t\r\n]+ -> skip ;
COMMENT        : '//' ~[\r\n]* -> skip ;
```

Features

- Basic Data Types
  - Integer,Float,String,Boolean
- Operations
  - Addition,Subtraction,Multiplication,Division
  - Relational: >,<,==
  - Logical: and,or,not
- Control Structures
  - Ternary,if-else.
  - Loop: for,while

Syntax

Program Structure

- Every statement must end with a semicolon ”;” except for blocks(conditional statements).
  Example: `set x to 10; say x;`
- Code blocks (like those in loops or conditionals) are enclosed in `{ ... }`.

Variables

- Variable names must start with a letter and may include letters, digits, and underscores.
- Declared using: `set <name> to <value>;` Example: `set score to 100;`

Conditionals

- Format: `when (<condition>) { ... } otherwise { ... }`
- The `otherwise` block is optional.

Loops

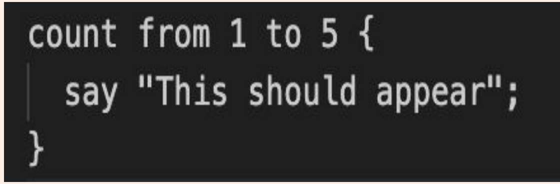
- For loop: `count from <start> to <end> { ... }`
- While loop: `repeat until (<condition>) { ... }`

Output

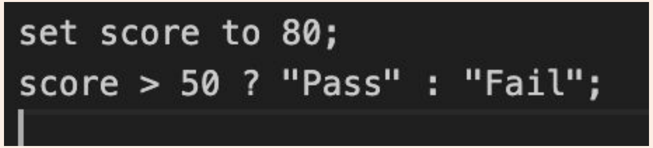
- Use `say <expression>;` to print output to console.

Literals & Booleans

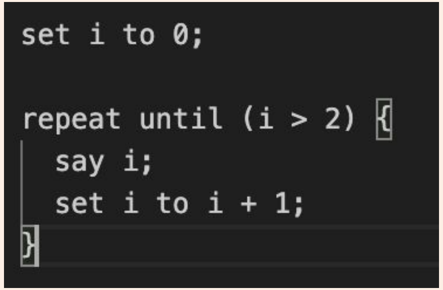
- Strings: `"Hello"`
- Integers & Floats: `5`, `3.14`
- Booleans: `yes`, `no`



```
count from 1 to 5 {
  say "This should appear";
}
```



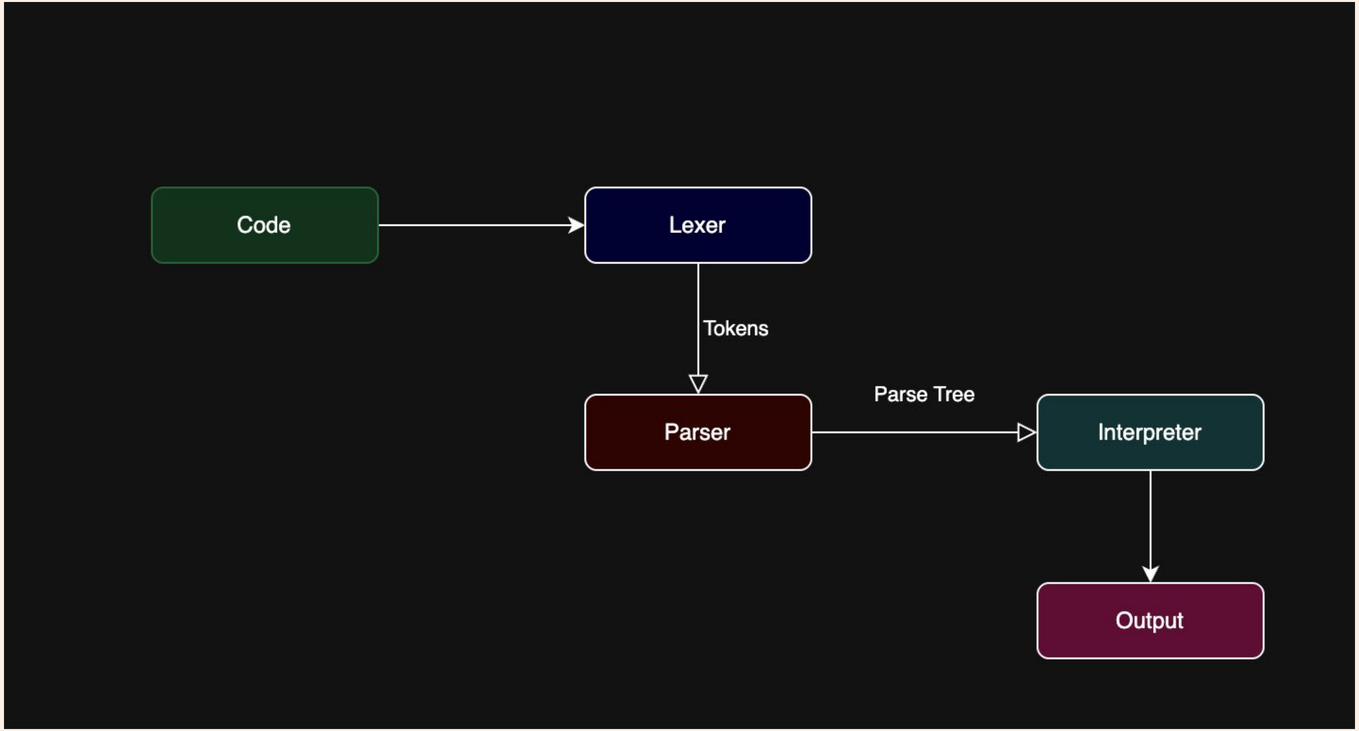
```
set score to 80;
score > 50 ? "Pass" : "Fail";
```



```
set i to 0;

repeat until (i > 2) {
  say i;
  set i to i + 1;
}
```

Flow



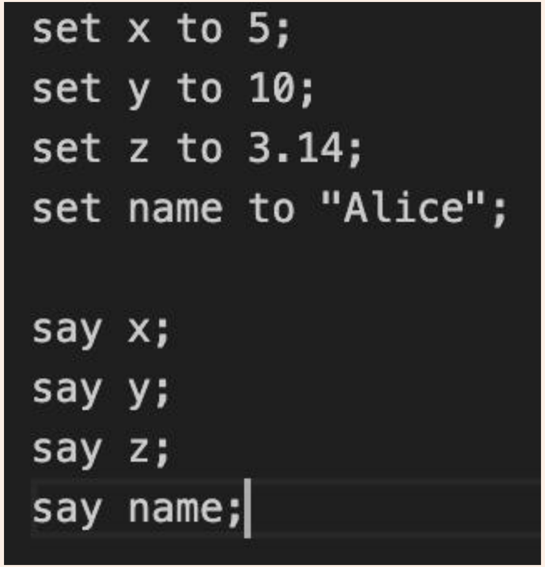
Compilation Flow

1. Lexical Analyzer

Purpose: Scans the KiddoLang source code and breaks it into tokens.

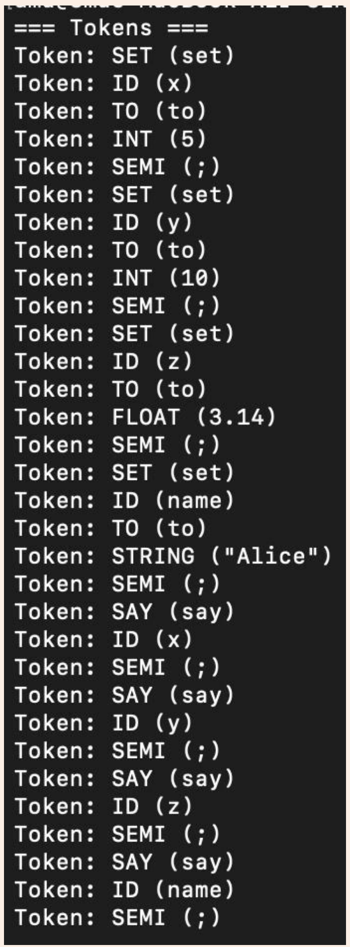
Output: A series of tokens representing keywords, identifiers, operators, etc.

Sample Input:



```
set x to 5;
set y to 10;
set z to 3.14;
set name to "Alice";

say x;
say y;
say z;
say name;
```

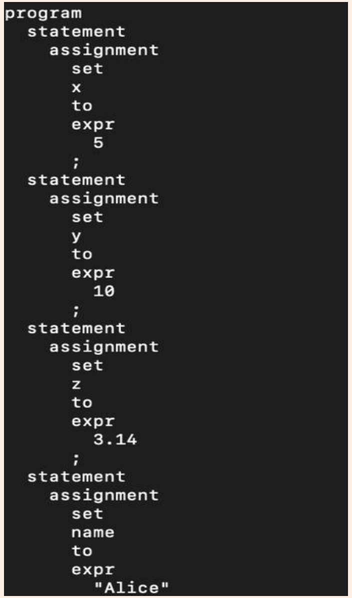


```
=== Tokens ===
Token: SET (set)
Token: ID (x)
Token: TO (to)
Token: INT (5)
Token: SEMI (;)
Token: SET (set)
Token: ID (y)
Token: TO (to)
Token: INT (10)
Token: SEMI (;)
Token: SET (set)
Token: ID (z)
Token: TO (to)
Token: FLOAT (3.14)
Token: SEMI (;)
Token: SET (set)
Token: ID (name)
Token: TO (to)
Token: STRING ("Alice")
Token: SEMI (;)
Token: SAY (say)
Token: ID (x)
Token: SEMI (;)
Token: SAY (say)
Token: ID (y)
Token: SEMI (;)
Token: SAY (say)
Token: ID (z)
Token: SEMI (;)
Token: SAY (say)
Token: ID (name)
Token: SEMI (;)
```

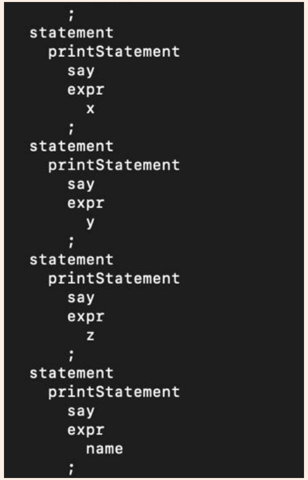
2. Parser

Purpose: Analyzes the sequence of tokens to ensure the syntax is correct.

Output: A parse tree representing the program’s structure.



```
program
  statement
    assignment
      set
      x
      to
      expr
        5
      ;
  statement
    assignment
      set
      y
      to
      expr
        10
      ;
  statement
    assignment
      set
      z
      to
      expr
        3.14
      ;
  statement
    assignment
      set
      name
      to
      expr
        "Alice"
```

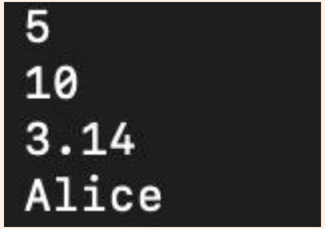


```
      ;
  statement
    printStatement
      say
      expr
        x
      ;
  statement
    printStatement
      say
      expr
        y
      ;
  statement
    printStatement
      say
      expr
        z
      ;
  statement
    printStatement
      say
      expr
        name
      ;
```

3. Interpreter

Purpose: Defines the logical meaning of each construct in the language.

Output: Ensures program behavior matches expected outcomes.



```
5
10
3.14
Alice
```

Sample Code

ADDITION:

```
set x to 8;

set y to x + 2;

say y;
```

FOR LOOP:

```
count from 2 to 10 {
        say “Hi”;
}
```

NESTED FOR LOOP:

```
count from 2 to 10 {
    count from 1 to 3 {
        say j;
    }
}
```

Sample Code

WHILE LOOP:

```
set a to 1;
repeat until (a < 10) {
    say "You can do this !!!";
    set a to a + 1;
}
```

IF-ELSE:

```
set a to 10;
set b to 20;
when (a > b) {
    say "a is greater";
}
otherwise {
    say "b is greater";
}
```

TERNARY EXPRESSION:

```
set a to 5;
set b to 10;
(a > b) ? a : b;
```

Future Work

- **Type Checking & Error Recovery**
  Introduce a robust type system and better runtime error handling to provide clear, beginner-friendly messages.

- **Function Definitions**
  Enable users to define and reuse functions to encourage modular thinking and code reuse.

- **Lists & Collections**
  Add support for list structures to allow iteration over multiple values and enhance expressiveness.

- **File I/O Capabilities**
  Allow reading from and writing to files for basic data processing.

THANK YOU