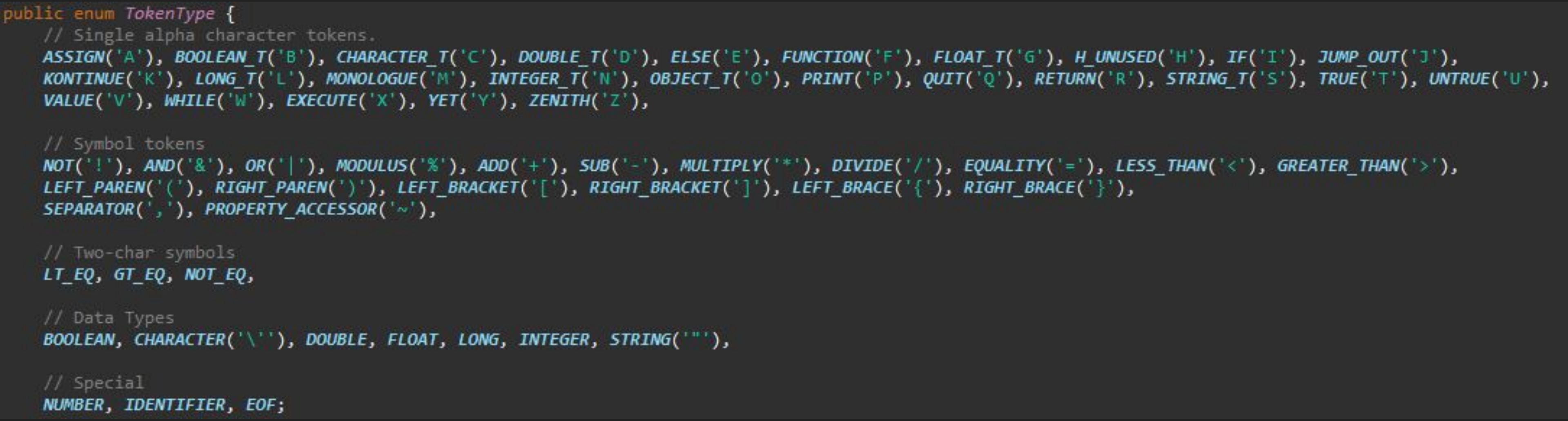
Sulfur Language

Eliminates whitespace and shortens your code length

Adam Omundsen

Lexer

Each capital letter has its own meaning and usage. Symbols usage is relatively similar to Java. Variables must be lowercase and can contain underscores and numbers. The lexer uses a combination of a hashmap and regular expressions.



```java
public enum TokenType {
    // Single alpha character tokens.
    ASSIGN('A'), BOOLEAN_T('B'), CHARACTER_T('C'), DOUBLE_T('D'), ELSE('E'), FUNCTION('F'), FLOAT_T('G'), H_UNUSED('H'), IF('I'), JUMP_OUT('J'),
    KONTINUE('K'), LONG_T('L'), MONOLOGUE('M'), INTEGER_T('N'), OBJECT_T('O'), PRINT('P'), QUIT('Q'), RETURN('R'), STRING_T('S'), TRUE('T'), UNTRUE('U'),
    VALUE('V'), WHILE('W'), EXECUTE('X'), YET('Y'), ZENITH('Z'),

    // Symbol tokens
    NOT('!'), AND('&'), OR('|'), MODULUS('%'), ADD('+'), SUB('-'), MULTIPLY('*'), DIVIDE('/'), EQUALITY('='), LESS_THAN('<'), GREATER_THAN('>'),
    LEFT_PAREN('('), RIGHT_PAREN(')'), LEFT_BRACKET('['), RIGHT_BRACKET(']'), LEFT_BRACE('{'), RIGHT_BRACE('}'),
    SEPARATOR(','), PROPERTY_ACCESSOR('~'),

    // Two-char symbols
    LT_EQ, GT_EQ, NOT_EQ,

    // Data Types
    BOOLEAN, CHARACTER('\''), DOUBLE, FLOAT, LONG, INTEGER, STRING('"'),

    // Special
    NUMBER, IDENTIFIER, EOF;
```

Symbols used by the language

Data Type Tokens:

B: Boolean

C: Character

D: Double

G: Float

L: Long

N: Integer

O: Object (WIP)

S: String

T: True / U: Untrue

Control Flow Tokens:

I: If

E: Else

W: While

J: Jump out (break)

K: Kontinue

F: Function

R: Return

Y: Yield ({)

Z Zenith (})

Q: Quit

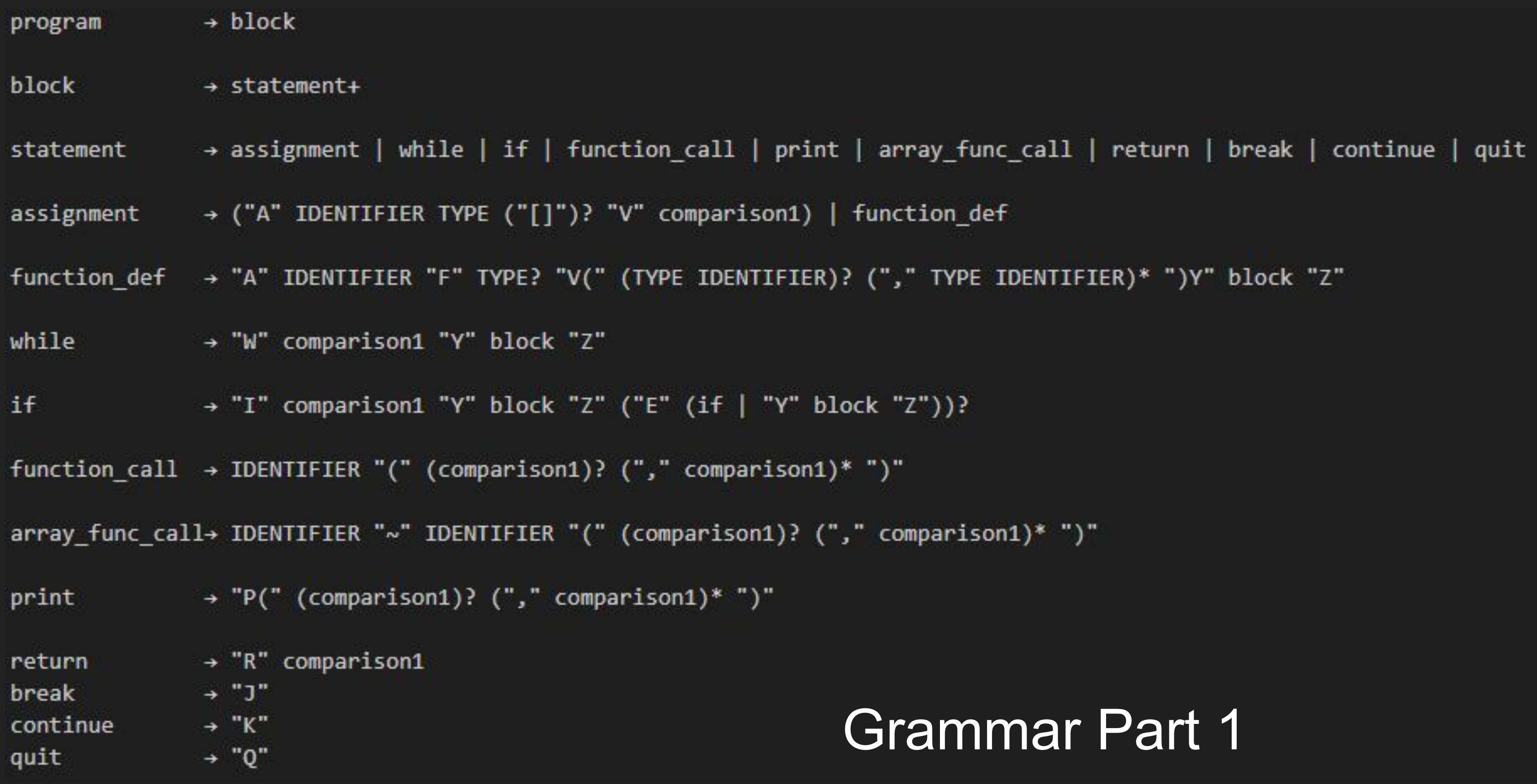
Other Tokens:

A Assign (=)

M Monologue (comment)

P Print

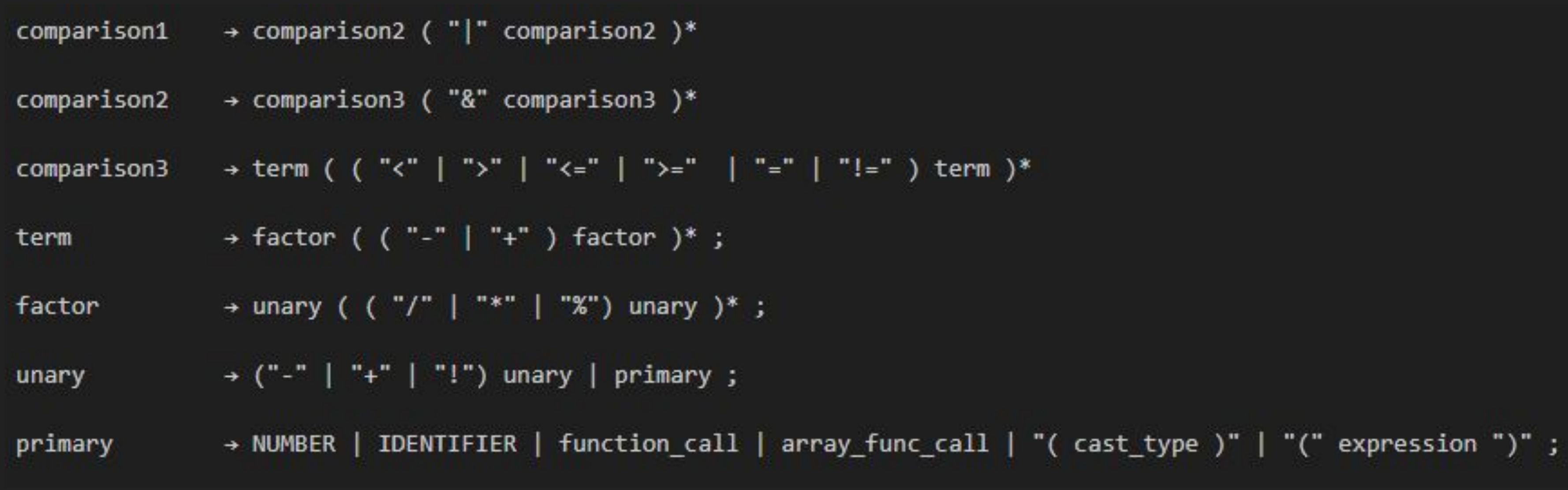
V Value (used with assignments)



```
program        → block

block          → statement+

statement      → assignment | while | if | function_call | print | array_func_call | return | break | continue | quit

assignment     → ("A" IDENTIFIER TYPE ("[]")? "V" comparison1) | function_def

function_def   → "A" IDENTIFIER "F" TYPE? "V(" (TYPE IDENTIFIER)? ("," TYPE IDENTIFIER)* ")Y" block "Z"

while          → "W" comparison1 "Y" block "Z"

if             → "I" comparison1 "Y" block "Z" ("E" (if | "Y" block "Z"))?

function_call  → IDENTIFIER "(" (comparison1)? ("," comparison1)* ")"

array_func_call→ IDENTIFIER "~" IDENTIFIER "(" (comparison1)? ("," comparison1)* ")"

print          → "P(" (comparison1)? ("," comparison1)* ")"

return         → "R" comparison1
break          → "J"
continue       → "K"
quit           → "Q"
```

Grammar Part 1

Grammar Part 2

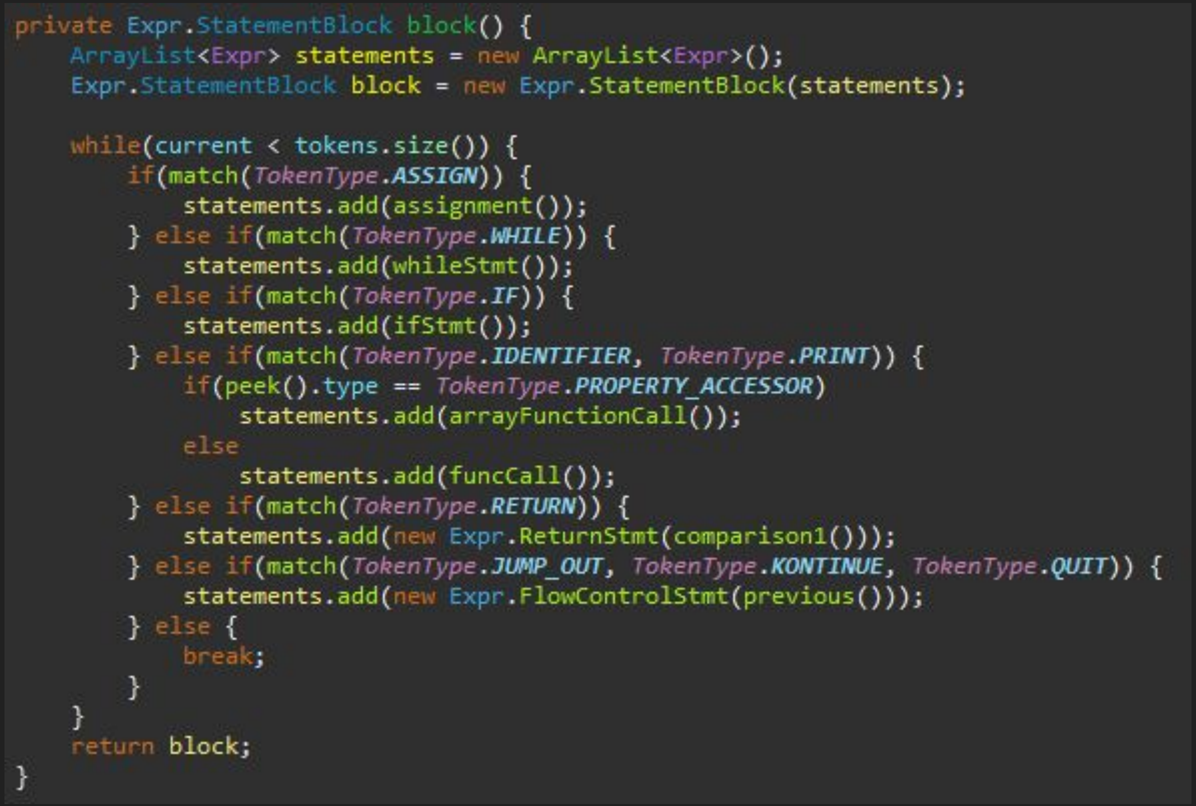


```
comparison1    → comparison2 ( "|" comparison2 )*

comparison2    → comparison3 ( "&" comparison3 )*

comparison3    → term ( ( "<" | ">" | "<=" | ">="  | "=" | "!=" ) term )*

term           → factor ( ( "-" | "+" ) factor )* ;

factor         → unary ( ( "/" | "*" | "%") unary )* ;

unary          → ("-" | "+" | "!") unary | primary ;

primary        → NUMBER | IDENTIFIER | function_call | array_func_call | "( cast_type )" | "(" expression ")" ;
```

Examples

Sulfur Code	Meaning
A num NV 42 * 2	Assign value (42*2) to integer variable ‘num’
A div_2 FNV (N x) Y R x/2 Z A div_2 FNV (N x) Y R x/2 Z Adiv_2FNV(Nx)YRx/2Z	Assign the function variable ‘div_2’ the value: a function with integer parameter ‘x’ that returns an integer, ‘x/2’
W num>5 Y A num NV div_2(num) Z	While num>5 divide it by 2
I num=3 Y P(“tree”) Z El num=2 Y P(“dos”) Z E Y P(num) Z	If num is 3, print “tree”, else if num is 2, print “dos”, otherwise print num’s value
A list D[]V {42.0, 3.14}	Create an arraylist with 2 initial double values
P(list~get(1))	Use the property accessor (~) to call an arraylist function

Parser



```
private Expr.StatementBlock block() {
    ArrayList<Expr> statements = new ArrayList<Expr>();
    Expr.StatementBlock block = new Expr.StatementBlock(statements);

    while(current < tokens.size()) {
        if(match(TokenType.ASSIGN)) {
            statements.add(assignment());
        } else if(match(TokenType.WHILE)) {
            statements.add(whileStmt());
        } else if(match(TokenType.IF)) {
            statements.add(ifStmt());
        } else if(match(TokenType.IDENTIFIER, TokenType.PRINT)) {
            if(peek().type == TokenType.PROPERTY_ACCESSOR)
                statements.add(arrayFunctionCall());
            else
                statements.add(funcCall());
        } else if(match(TokenType.RETURN)) {
            statements.add(new Expr.ReturnStmt(comparison1()));
        } else if(match(TokenType.JUMP_OUT, TokenType.KONTINUE, TokenType.QUIT)) {
            statements.add(new Expr.FlowControlStmt(previous()));
        } else {
            break;
        }
    }
    return block;
}
```

Interpreter

- Like always, written in Java
- Starts at root expression and evaluates all sub trees
- Each expression type has its own evaluation function
- Can throw errors due to wrong types, undefined variables, misused break/continue statements etc.

Live Demo

Questions?