CS421 PA3
Sulfur Language Parser

Adam Omundsen

Quick Syntax Review

Prime Finder Program

```
AnumNV2

Wnum<100Y
Ais_primeBVT
AdivisorNV2

Wdivisor<numY

Inum%divisor=2Y
Ais_primeBVU
J
Z
```

```

AdivisorNVdivisor+1
Z

Iis_primeY
P(num)
P("\n")
Z
AnumNVnum+1
Z
```

Quick Syntax Review (cont.)

Functions Program

```
M# Defines 2 simple functions, runs them, and displays the results #M

M# Add function returns an integer(N) and takes two integer arguments #M
AaddFNV(Na,Nb)Y
Ra+b
Z

P("4+2: ")
P(add(4,2))
P('\n')

M# Factorial function is recursive, takes an int as an argument and returns a long #M
AfacFLV(Nx)Y
Ix=0|x=1Y
R1
ZEY
Rx*fac(x-1)
Z
Z

P("Factorial of 7: ")
P(fac(7))
P('\n')
```

```
program       → block

block         → statement+

statement     → assignment | while | if | function_call | print | 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)* ")"

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 | "( cast_type )" | "(" expression ")" ;
```

Operator Precedence

- Borrowed from the internet

Precedence	Operator	Type	Associativity
15	() [] .	Parentheses Array subscript Member selection	Left to Right
14	++ --	Unary post-increment Unary post-decrement	Right to left
13	++ -- + - ! ~ (*type*)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast	Right to left
12	* / %	Multiplication Division Modulus	Left to right
11	+ -	Addition Subtraction	Left to right
10	<< >> >>>	Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension	Left to right
9	< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)	Left to right
8	== !=	Relational is equal to Relational is not equal to	Left to right
7	&	Bitwise AND	Left to right
6	^	Bitwise exclusive OR	Left to right
5	|	Bitwise inclusive OR	Left to right
4	&&	Logical AND	Left to right
3	||	Logical OR	Left to right
2	? :	Ternary conditional	Right to left
1	= += -= *= /= %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to left

Parser Generator

```
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)) {
            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;
}
```

Live Demo