JPP - Interpreter

Opis języka

Karol Soczewica (ks394468)

Opis

Język, którego interpreter będę implementował to język Latte z pewnymi dodatkami/zmianami (Latte++). Wykonywanie programu będzie zaczynało się od funkcji **int main() {}**, która w programie musi wystąpić.
W Latte++ dodatkowym typem będzie jednowymiarowa tablica.
Dodatkami do instrukcji Latte są instrukcje **for**, **foreach** (tak jak w Javie), **break**, **continue** oraz **print**. Oprócz tego język Latte++ będzie miał słowo kluczowe **final**, dzięki któremu będzie można mieć zmienne tylko do odczytu. Intrukcja **if** zostanie rozszerzona o możliwość dodawania bloków **else if**.
Zmianą w składni w stosunku do Latte jest to, że instrukcje w pętlach oraz w **if** muszą zawsze znajdować się w środku bloku, czyli nielegalne będzie napisanie **if (i < 5) print(i);**, zamiast tego trzeba będzie napisać **if (i < 5) { print(i); }**.
Każda zmienna musi być zadeklarowana przed użyciem. Jeśli zmienna nie jest zainicjalizowana, to przyjmuje domyślną dla swojego typu wartość: **int** → 0, **bool** → false, **string** → "".
Zmienne deklarowane wewnątrz bloków przesłaniają zmienne o tych samych nazwach spoza bloku.
Parametry funkcji przekazywane są przez wartość.

Gramatyka (LBNF)

```
-- Programs ------------------------------------------------------------

entrypoints Program ;

Program. Program ::= [FunDef] ;

-- Types ---------------------------------------------------------------

Int. Type ::= "int" ;

Str. Type ::= "string" ;

Bool. Type ::= "bool" ;

Void. Type ::= "void" ;

Array. Type ::= "Array" "<" Type ">" ;

-- Statements ----------------------------------------------------------

Block. Block ::= "{" [Stmt] "}" ;

separator Stmt "" ;

BStmt. Stmt ::= Block ;

Empty. Stmt ::= ";" ;

FunDef. FunDef ::= Type Ident "(" [Arg] ")" Block ;

separator nonempty FunDef "" ;
```

```
Arg. Arg ::= Type Ident ;

separator Arg "," ;

FStmt. Stmt ::= FunDef ;

ArrDecl. Stmt ::= "Array" "<" Type ">" Ident "=" Expr "**" "[" Expr "]" ";" ;

ArrAss. Stmt ::= Ident "[" Expr "]" "=" Expr ";" ;

DStmt. Stmt ::= Decl ;

NormalDecl. Decl ::= Type [Item] ";" ;

FinalDecl. Decl ::= "final" Type [Item] ";" ;

NoInit. Item ::= Ident ;

Init. Item ::= Ident "=" Expr ;

separator nonempty Item "," ;

Ass. Stmt ::= Ident "=" Expr ";" ;

Inc. Stmt ::= Ident "++" ";" ;

Dec. Stmt ::= Ident "--" ";" ;

Ret. Stmt ::= "return" Expr ";" ;

RetV. Stmt ::= "return" ";" ;

Break. Stmt ::= "break" ";" ;

Continue. Stmt ::= "continue" ";" ;

Cond. Stmt ::= "if" "(" Expr ")" Block [ElseIf] ;

CondElse. Stmt ::= "if" "(" Expr ")" Block [ElseIf] "else" Block ;

ElseIf. ElseIf ::= "else if" "(" Expr ")" Block ;

separator ElseIf "" ;

While. Stmt ::= "while" "(" Expr ")" Block ;

For. Stmt ::= "for" "(" ForInit [Expr] ";" [Expr] ")" Block ;

ForInitExpr. ForInit ::= [Expr] ";" ;

ForInitVar. ForInit ::= Decl ;

ForIn. Stmt ::= "for" "(" Ident ":" Ident ")" Block ;

EStmt. Stmt ::= Expr ";" ;

Print. Stmt ::= "print" "(" Expr ")" ";" ;
```

```
-- Expressions ---------------------------------------------------------

Evar. Expr6 ::= Ident ;

ELitInt. Expr6 ::= Integer ;

ELitTrue. Expr6 ::= "true" ;

ELitFalse. Expr6 ::= "false" ;

EApp. Expr6 ::= Ident "(" [Expr] ")" ;

ArrRead. Expr6 ::= Ident "[" Expr "]" ;

EString. Expr6 ::= String ;

Neg. Expr5 ::= "-" Expr6 ;

Not. Expr5 ::= "!" Expr6 ;

EMul. Expr4 ::= Expr4 MulOp Expr5 ;

EAdd. Expr3 ::= Expr3 AddOp Expr4 ;

EInc. Expr3 ::= Expr3 "++" ;

EDec. Expr3 ::= Expr3 "--" ;

ERel. Expr2 ::= Expr2 RelOp Expr3 ;

EAnd. Expr1 ::= Expr1 "&&" Expr2 ;

EOr. Expr ::= Expr "||" Expr1 ;

coercions Expr 6 ;

separator Expr "," ;

-- Operators ----------------------------------------------------------

Plus. AddOp ::= "+" ;

Minus. AddOp ::= "-" ;

Times. MulOp ::= "*" ;

Div. MulOp ::= "/" ;

Mod. MulOp ::= "%" ;

Lt. RelOp ::= "<" ;

Leq. RelOp ::= "<=" ;

Gt. RelOp ::= ">" ;

Geq. RelOp ::= ">=" ;
```

```
Eq. RelOp ::= "==" ;

Neq. RelOp ::= "!=" ;

-- Comments ---------------------------------------------------------------

comment "#" ;

comment "//" ;

comment "/*" "*/" ;
```

Przykłady

```
// PrintArrayElements.lpp (wypisywanie wartosci z tablicy na trzy sposoby)
int main() {
    Array<int> xs = 5 ** [1]; // utworzenie tablicy postaci [1, 1, 1, 1, 1]
    for (x : xs) {
        print(x);
    }

    for (int i = 0; i < 5; i++) {
        print(xs[i]);
    }

    int i = 0;
    while (i < 5) {
        print(xs[i]);
    }
    return 0;
}
```

```
// PrintEvenNumbers.lpp
int main() {
    bool isEven(int x) {
        if (x % 2 == 0) {
            return true;
        } else {
            return false;
        }
    }

    Array<int> numbers = 10 ** [0];
    for (int i = 1; i <= 10; i++) {
        numbers[i - 1] = i;
    }

    for (x : numbers) {
        if (isEven(x)) {
            print(x);
        }
    }
    return 0;
}
```

```
// Fib.lpp (obliczanie n-tej liczby Fibonacciego na trzy sposoby)
int fib_rec(int n) {
    if (n <= 1) {
        return n;
    }
    return fib_rec(n - 1) + fib_rec(n - 2);
}

int main() {
    int n = 10;
    print(fib_rec(n));
    print(fib_arr(n));
    print(fib_opt(n));
    return 0;
}

int fib_arr(int n) {
    Array<int> f = (n + 2) ** [0];
    int i = 2;

    f[1] = 1;
    while (i <= n) {
        f[i] = f[i - 1] + f[i - 2];
        i++;
    }

    return f[n];
}

int fib_opt(int n) {
    int f1 = 0, f2 = 1, res;
    if (n == 0) {
        return f1;
    }

    for (int i = 2; i <= n; i++) {
        res = f1 + f2;
        f1 = f2;
        f2 = res;
    }

    return f2;
}
```

Tabelka funkcjonalności

Na 15 punktów

01 (trzy typy)	+
02 (literały, arytmetyka, porównania)	+
03 (zmienne, przypisanie)	+
04 (print)	+
05 (while, if)	+
06 (funkcje lub procedury, rekurencja)	+
07 (przez zmienną / przez wartość / in/out)	+
08 (zmienne read-only i pętla for)	+

Na 20 punktów

09 (przesłanianie i statyczne wiązanie)	+
10 (obsługa błędów wykonania)	+
11 (funkcje zwracające wartość)	+

Na 30 punktów

12 (4) (statyczne typowanie)	+
13 (2) (funkcje zagnieżdżone ze statycznym wiązaniem)	+
14 (1) (rekordy/tablice/listy)	+
15 (2) (krotki z przypisaniem)	
16 (1) (break, continue)	+
17 (4) (funkcje wyższego rzędu, anonimowe, domknięcia)	
18 (3) (generatory)	

Razem: $20 + 4 + 2 + 1 + 1 = 28$