Dokumentacja końcowa

Paweł Bęza

Temat projektu

Interpreter prostego języka z typami specjalnymi reprezentującymi wektory 2d i 3d

Opis funkcjonalny

Język *SimPL* jest dynamicznie typowanym, interpretowanym językiem wysokiego poziomu umożliwiającym programowanie proceduralne.

Język wspiera następujące konstrukcje językowe:

Typy danych

typ	rozmiar w bajtach	opis
int	4	32 bitowa liczba całkowita
vec2d	8	wektor 2d składający z się z 2 liczb typu int
vec3d	12	wektor 3d składający się z 3 liczb typu int

Operatory

typ	arytmetyczne	przypisania	porównania
int	“+”, “-”, “*”, “/”, “%”	“=”, “:=”	“==”, “!=”, “<”, “<=”, “>”, “>=”
vec2d, vec3d	“+”, “-”, “*”, “x”	“=”, “:=”	“==”, “!=”

W przypadku wektorów operacja:

- “*” oznacza mnożenie skalarne
- “x” oznacza mnożenie wektorowe

Zmienne

typ	inicjalizacja
int	a := 0;
vec2d	vec2d := [1,2]
vec3d	vec3d := [1,2,3]

Zakres widoczności zmiennych

Widoczność zmiennej jest ograniczona do *bloku*, w którym została zadeklarowana, gdzie:
blok - sekcja kodu zamknięta między nawiasy klamrowe {}

Instrukcje warunkowe

keywords	przykład użycia

if oraz else	i := 0;
	if (i == 1) {
	  i = i + 1;
	} else {
	  i = i / 2;
	}

Instrukcje pętli

keyword	przykład użycia

while	i := 10;
	while (i > 0) {
	  i = i - 1;
	}

Definicje funkcji

keyword	przykłady użycia

func	func slow_pow(base, exp) {
	  result := 1;
	  while (exp > 0) {
	  result = result * base;
	  exp = exp - 1;
	  }
	  return result;
	}
	
	func factorial(n) {
	  if (n <= 1) {
	  return 1;
	  }
	  return factorial(n - 1) * n;
	}

Argumenty funkcji są przekazywane przez wartość.

Zaktualizowany opis gramatyki

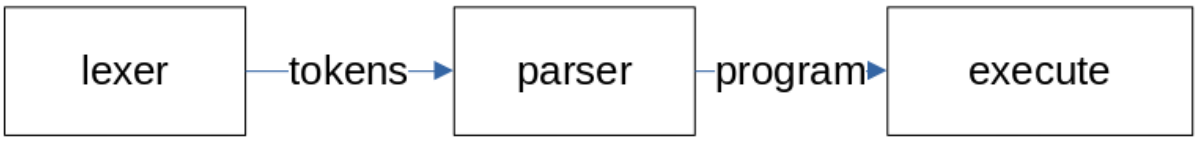
```
<program> ::= <function_def>+
<function_def> ::= "func" <whitespace> <id> "(" <parameters> ")" <block>
<parameters> ::= (<id> ("," <id>)*)?
<block> ::= "{" ((((<function_call> | <function_return> | <var_definition> |
<var_assignment> | <print>) ";") | <while> | <conditional>)* | <block>) "}"

<while> ::= "while" "(" <logic_expression> ")" <block>
<function_call> ::= <id> "(" <function_args> ")"
<function_return> ::= "return" <whitespace> <logic_expression>
<function_args> ::= (<logic_expression> ("," <logic_expression>)*)?
<conditional> ::= "if" "(" <logic_expression> ")" <block> ("else" <block>)?
<var_definition> ::= <id> ":=" <logic_expression>
<var_assignment> ::= <id> "=" <logic_expression>
<print> ::= "print" "(" (<string> | <logic_expression>) ("," (<string> |
<logic_expression>))* ")"

<logic_expression> ::= <logic_and_term> (<or_op> <logic_and_term>)*
<logic_and_term> ::= <logic_rel_term> (<and_op> <logic_rel_term>)*
<logic_rel_term> ::= <logic_factor> (<relational_op> <logic_factor>)*
<logic_factor> ::= <math_expression> | <logic_unary_op> <math_epxression>

<math_expression> ::= <math_term> (<additive_op> <math_term>)*
<math_term> ::= <math_factor> (<multiplicative_op> <math_factor>)*
<math_factor> ::= <value> | “-” <math_factor>  | "(" <logic_expression> ")"

<additive_op> ::= "+" | "-"
<multiplicative_op> ::= "*" | "x" | "/" | "%"
<relational_op> ::= "==" | "!=" | "<" | "<=" | ">" | ">="
<logic_unary_op> ::= "!"
<and_op> ::= "&&"
<or_op> ::= "||"

<string> ::= "\"" <printable_char> "\""
<value> ::= <id> | <int> | <vec> | <function_call>
<id> ::= <letter> (<letter> | <digit> | <special_char>)*
<int> ::= "0" | [1-9] <digit>*
<vec> ::= "[" <int> "," <int> ("," <int>)? "]"

<letter> ::= [a-z] | [A-Z]
<printable_char> ::= [:print:]
<whitespace> ::= " "
<special_char> ::= "_"
<digit> ::= [0-9]
```

Opis techniczny realizacji

Komponenty

Program został podzielony na 3 komponenty:

- lexer - wyodrębnia tokeny z ciągu znaków
- parser - z utworzonych tokenów tworzy *AST* i zwraca korzeń drzewa składniowego
- execute - uruchomienie interpretacji drzewa AST



Narzędzia

język	C++
biblioteki	STL, Google Test
inne	CMake

Kompilacja interpretera

```
mkdir build && cd build && cmake .. && make
```

Uruchamianie interpretera

Wymaga argumentu w postaci ścieżki do interpretowanego pliku

```
./simpl_demo <path_to_interpreted_file>
```

Uruchomienie testów

```
./build/test/simpl_lib_test
```

Przykłady programów w języku SimPL

Plik wejściowy nr 1

```
func print_tree(base_tree) {
    print("While loop tree\n");
    i := 0;
    while (i < base_tree) {
        j := 0;
        while (j < base_tree - i - 1) {
            print(" ");
            j = j + 1;
        }

        j = 0;
        while (j < 2 * i + 1) {
            print("*");
            j = j + 1;
        }

        i = i + 1;

        print("\n");
    }
}

func main() {
    print_tree(6);
}
```

Wyjście

```
While loop tree
     *
    ***
   *****
  *******
 *********
***********
```

Plik wejściowy nr 2

```
func fast_pow(base, exp) {
    if (exp == 0) {
        return 1;
    }

    half_pow := fast_pow(base, exp / 2);
    if (exp % 2 == 0) {
        return half_pow * half_pow;
    }
    return half_pow * half_pow * base;
}
```

```
func main() {
    fast_pow(2, 10);
}
```

Wyjście

```
[1024]
```

Plik wejściowy nr 3

```
func add_vec(a, b) {
    return a + b;
}

func main() {
    print("Vector ops:\n\n");

    a := [1,2,3];
    b := [3,2,1];
    vec_sum := add_vec(a, b);
    print("adding vectors: ", a, "+", b, "=", vec_sum, "\n");

    print("vector product of vectors: ", a, "*", b, "=", a * b, "\n");
    print("scalar product of vectors: ", a, "x", b, "=", a x b, "\n");

    print("vector difference: ", [1,2], "-", [2,1], "=", [1,2] - [2,1], "\n");
    print("vector difference: ", [1,2], "/", [2,1], "=", [1,2] / [2,1], "\n");

    print("vector ops mix: ", [1,2], "*", [2,1], "+", [1,1,1], "*", [1,1,1],
"=", [1,2] * [2,1] + [1,1,1] * [1,1,1], "\n");
}
```

Wyjście

```
Vector ops:

adding vectors: [1, 2, 3]+[3, 2, 1]=[4, 4, 4]
vector product of vectors: [1, 2, 3]*[3, 2, 1]=[10]
scalar product of vectors: [1, 2, 3]x[3, 2, 1]=[-4, 8, -4]
vector difference: [1, 2]-[2, 1]=[-1, 1]
vector difference: [1, 2]/[2, 1]=[0, 2]
vector ops mix: [1, 2]*[2, 1]+[1, 1, 1]*[1, 1, 1]=[7]
```

Plik wejściowy nr 4

```
func ugly_factorial(n) {
    if (n <= 1) {
        return 1;
    }
    return ugly_factorial(n - 1) * n;
}
```

```
func main() {
    ugly_factorial(5);
}
```

Wyjście

```
[120]
```