Opis

Język natywnie wspiera typ całkowitoliczbowy (int), zmiennoprzecinkowy (float), i ciąg znaków (string), który posiada metodę budującą co umożliwia użytkowników korzystanie z dowolnych znaków. Umożliwia także deklarację funkcji, zmiennych globalnych, zdania warunkowe, pętlę “while”, rekurencję, przypisywanie wartości do wcześniej zdefiniowanych zmiennych jak i “przykrywanie” zmiennych globalnych przez zmienne lokalne. W celu wypisywanie zmiennych na ekran używanie jest funkcja “print”. W razie występowania błędów wypisywany jest komunikat o błędzie i jego lokacja w kodzie. Wspierane operacje na typach liczbowych to: dodawanie, odejmowanie, mnożenie i dzielenie. Dla typu znakowego możliwa jest konkatenacja za pomocą znaku dodawania. Język wspiera operacje logiczne które zwracają wartości całkowite, gdzie 0 oznacza fałsz, a liczba różna od zera oznacz prawdę. Konstruowanie własnych komentarzy jest dozwolone zaczynających się od “//“ i kończących się wraz z końcem linii. Funkcje nie muszą zwracać wartości ale słowo kluczowe “return” sprawia że reszta ciała funkcji nie jest wykonywana. Tworzenie zmiennych bez wartości domyślnej sprawia że są one inicjalizowaine jako zero, lub puste. Wszystkie zmienne mają swój zakres widoczności, zmienne zadeklarowane w danej funkcji są niewidoczne w innych funkcjach.

Budowa programu

Interpreter jest podzielony na cztery główne części
- Analizator leksykalny
- Analizator składniowy
- Analizator semantyczny
- Interpreter drzewa AST

Analizator leksykalny

Jako wejście przyjmuje kod źródłowy programu i przetwarza go na tokeny zdefiniowane jako gramatyka języka. Tokeny są jednoznacznie powiązane z wartościami, słowa kluczowe i specjalne symbole przedstawiane są przez zdefiniowane do tego tokeny, a stałe liczbowe i identyfikatory przechowują w sobie odpowiednie informacje. Dodatkowo tokeny przechowują w sobie informację o swoim położeniu w kodzie źródłowym programu. Ze względu na charakterystykę swojego zadania analizator informuje o błędach w sytuacji gdy:
- Zdefiniowana liczba zaczyna się zerami
- Został przekroczony rozmiar wartości lub nazwy identyfikatora

---

Analizator składniowy

Przyjmuje na wejściu strumień tokenów wyprodukowanych przez analizator leksykalny. Jego zadaniem jest wyprodukowanie drzewa rozbioru składniowego. Na podstawie tokenów, w sposób rekursywnie zstępujący oczekuje na tokeny określonego typu. Obsługa błędów jest zrealizowany w sposób informowania o napotkanym błędzie za pomocą wyjątku, który zawiera informacje o położeniu

błędnego wyrażenia w kodzie źródłowym. Jest najbardziej rygorystycznym elementem interpretera, oczekuje na konkretne tokeny przy analizie wyrażeń.

Analizator semantyczny

Operuje on na drzewie AST, które produkuje analizator składniowy. Na celu ma ma analizę “sensowności” kodu (np. sprawdzenie czy w kodzie występuje funkcja “main”). Sprawdza odwołania do zmiennych, czy zgadza się typ, czy zostały użyte w poprawnym kontekście, czy istnieją etc. Jeżeli trafi na nieścisłość informuje o tym wypisując lokalizację kodu, komunikat o błędzie, dodatkowo jeśli naruszenie semantyki nastąpiło w kontekście zmiennych lub funkcji wypisuje identyfikator.

Interpreter drzewa AST

Gdy analiza semantyczna zakończy się powodzeniem, do pracy przystępuje interpreter drzewa AST. Ma on na celu nadanie wartości zmiennym globalnym i wykonaniem odpowiednich instrukcji zdefiniowanych przez funkcje.

Dodatkowo każdy z modułów programu korzysta z ze specjalnie stworzonej klasy wyjątków “LangException” która wypisuje komunikat o napotkanym błędzie, pozycji w której wystąpił błąd w kodzie źródłowym i pozycji względem zaimplementowanych modułów.

Implementacja

W celu implementacji analizatora leksykalnego zostałe zaimplementowane:
- Enumeracja “TokenType” które identyfikuje typ tokenu
- Klasa pomocnicza “Position” identyfikuje jednoznacznie położenie tokenów w kodzie
- Klasa “Token” reprezentujące indywidualne tokeny ich wartość i położenie

W celu reprezentacji strumienia znaków została wykorzystana klasa bazowa “std::istream”. Implementacja analizatora składniowego jest reprezentowana przez abstrakcyjną klasę bazową, która określa węzeł w drzewie. Każdy węzeł jest obiektem pochodnym klasy “Node”. Analizator semantyczny został zaimplementowany za pomocą wzorca projektowego “wizytator”, wymagało to dodanie implementacji publicznej metody dla każdego węzła. Interpreter tak jak analizator semantyczny został zaimplementowany przy użyciu wzorca “wizytator”. Metody służące odwiedzaniu wymuszają odwiedzenie węzłów potomnych - co sprawia że odwiedzając blok warunkowy odwiedzony zostanie jedynie “aktywny” blok.

Testy

Moduły posiadają testy zrealizowane za pomocą biblioteki “google test”. Dla każdego modułu z wyjątkiem analizatora leksykalnego (który posiada testy jednostkowe), są to testy integracyjne z poprzednikami ze względu na problematykę projektu.

Analizator leksykalny

Sprawdzane jest czy dla danego ciągu znaków zwrócony zostanie ciąg określonych enumeracji “TokenType”

Analizator składniowy

Testy polegają na sprawdzeniu czy dla danego kodu źródłowego zostanie wygenerowane zostanie odpowiednie drzewo AST które reprezentuje ciągi odwiedzeń konkretnych węzłów.

Interpreter

W celu przetestowania poprawności działania interpretera jest realizowane przy użyciu funkcji “print” sprawdzając czy wypisywany wynik działania programu jest zgodny z oczekiwaniami.

Gramatyka

Część składniowa

<program> ::= <declaration>
| <function>
| <comment>

<comment> ::= “//“ <character> “\n”

<declaration> ::= <type> <identifier> [“=“ <arithmetic_expression>]

<function> ::= <type> <identifier> ”(” {<type> <identifier> ”)” <block>

<block> ::= ”{” <statement_list> ”}”

<statement_list> ::= “{“, <statement>, “}”

<statement> ::= <function_call>, “;”
| <if_statement>
| <while_statement>
| <declaration>, “;”
| <assignment>, “;”
| <return_statement>, “;”
| <statement_list>

<assignment> ::= <identifier>, “=“, {<arithmetic_expression> |
<string_expression> } ”;”

<assignment_expression> ::= <identifier> ”=” <logical_expression>

<if_statement> ::= “if”, “(“, <expression>, “)” <statement_list>, [“else”, <statement_list]

<while_statement> ::= “while”, “(“, <expression>, “)”, “<statement_list>

<return_statement> ::= “return”, <arithmetic_expression>

<expression> ::= <arithmetic_expression>, [<relational_operator>, <arithmetic_expression>]

<function_call> ::= <identifier>, “(“, [<argument_list>], “)”

<argument_list> ::= <arithmetic_expression>, “{“, <arithmetic_expression>, “}”

<arithmetic_expression> ::= <term>, “+”, <term> | [<term>], “-“ | <term>

<string_expression> ::= <character>, “+”, <character>

<term> ::= <factor>, “*”, <factor>
| <factor>, “/“, <factor>
| <factor>

<factor> ::= “(“, <arithmetic_expression>, “)”
| <number>
| <identifier>
| <function_call>

<identifier> ::= <letter>, { letter | digit}

<character> ::= <letter>
| <digit>
| <special_character>

Część leksykalna

<type> ::= ”int”
| ”float”
| ”string”

<return_statement> ::= ”return” <logical_expression> ;

<literal> ::= <integer_literal>
| <float_literal>
| <string_literal>

<integer_literal> ::= <non_zero_digit>, {<digit>}

<float_literal> ::= <integer_literal>, ”.”, {<integer_literal>}

<char_literal> ::= ”‘” <character> ”‘”

<string> ::= ‘”‘ {<character>} ‘”‘

<letter> ::= ”A” | ”B” | … | ”Z” | ”a” | … | ”z” |

<digit> ::= ”0” | <non_zero_digit>

<non_zero_digit> ::= “1” | “2” | … | “9”

<character> ::= <letter> | <digit> | <special_character>

<special_character> ::= ” ” | ”!” | ”$” | … | ”~”

Dokumentacja online

Dodatkowo dostępna jest dokumentacja online