Techniki kompilacji 2024Z

Jan Górski

Informacje podstawowe

Tematem projektu jest napisanie interpretera własnego języka programowania.

- Język wykorzystany w projekcie: C++
- Wykorzystane biblioteki:
  - Boost.

Opis zakładanej funkcjonalności

- kod programu napisane w języku ewaluowany linijka po linijce - brak punku wejściowego, jak np. funkcja `main`.
- wartości są domyślnie stałe: wartość zmiennej zadeklarowanej bez użycia słowa kluczowego `mut` nie może być zmieniona;
- typowanie dynamiczne - zmienna może w czasie działania programu zmienić swój typ,
- typowanie silne - zmienna w każdej chwili działania programu ma przypisany typ,
- zmienne są przekazywanie przez referencję,
- pętla `while`,
- instrukcje warunkowe `if`, `else`,
- wyrażenia warunkowe zwracają wartość,
- całkowitoliczbowe typy danych, typ logiczny `bool`,
- funkcja może być przekazywana jako parametr innej funkcji,
- operator `pipe` oraz

Przykłady obrazujące dopuszczalne konstrukcje językowe i semantykę

- w pliku `example.txt`.

Formalna specyfikacja i składnia EBNF realizowanego języka

```
program = features ;

features = { feature } ;

feature = {
        | one_line_comment
        | multiline_comment
        | statement
        } ;

one_line_comment = "//" , { character - new_line } , new_line ;
```

```
multiline_comment = "/*" , { ( character character ) - "*/" } , "*/" ;

statement = variable_declaration
          | while_loop
          | return_statement
          | expression_statement ;

variable_declaration = "let" [ "mut"] identifier [ "=" expession ] ";" ;

while_loop = "while" "(" equality_expression ")" "{"  { statement } "}" ;

return_statement = "return" [ expression ] ";" ;

expression_statement = [ expression ] ";" ;

expression = function_declaration
           | if_expression
           | literal
           | identifier
           | assignement_expression
           | function_call ;

if_expression = if_clause { else_if_clause } [ else_clause ]

else_if_clause = "else" if_clause ;

else_clause = "else" "{" statements  [ expression ] "}" ;

if_clause = "if" "(" relational_expression ")" "{" statements [ expression ]

assignement_expression = equlity_expression
                       | primary_expression "=" equality_expression ;

equality_expression = relational_expression
                    | equality_expression ( "==" | "!=" ) relational_expression

relational_expression = additive_expression
                      | relational_expression ( "<" | ">" | "<=" | ">=" ) additive_expre

additive_expression = multiplicative_expression
                    | additive_expression ( "+" | "-" ) multiplicative_expression ;

multiplicative_expression = cast_expression
                          | multiplicative_expression ( "+" | "-" ) cast_expression ;

cast_expression = primary_expression
```

```
                      | cast_expression "<-" type_name ;

     primary_expression = identifier
                      | literal
                      | "(" expression ")" ;

             identifier = letter { letter | digit | "_" } ;

                literal = bool_literal
                        | float_literal
                        | integer_literal
                        | string_literal ;

            bool_literal = "true" | "false" ;

           float_literal = integer_literal "." { digit } ;

         integer_literal = ( non_zero_digit { digit } ) | "0" ;

          string_literal = """ {
                           special_string_character | normal_string_character
                           } """ ;

           function_call = identifier "(" { expression } ")" ;

   function_declaration = "fn" , identifier , function_parameter_list function_body ;

 function_parameter_list = "(" , [
                           [ "mut" ] identifier , { "," , [ "mut" ] identifier }
                           ] ")" ;

           function_body = "{"
                           { statement - return_statment }
                           [ return_statement | expression ]
                           "}" ;

 normal_string_character = character - ( """ | "\" ) ;

special_string_character = "\"" | "\n" | "\\" | "\b";

               character = alphanumeric_character
                         | special_character
                         | white_character ;

         white_character = "\n" | "\t" | " " ;
```

```
alphanumeric_character = letter | digit ;

                letter = "A" | "B" | "C" | "D" | "E" | "F" | "G"
                       | "H" | "I" | "J" | "K" | "L" | "M" | "N"
                       | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
                       | "V" | "W" | "X" | "Y" | "Z" | "a" | "b"
                       | "c" | "d" | "e" | "f" | "g" | "h" | "i"
                       | "j" | "k" | "l" | "m" | "n" | "o" | "p"
                       | "q" | "r" | "s" | "t" | "u" | "v" | "w"
                       | "x" | "y" | "z" ;

                 digit = "0" | non_zero_digit ;

          non_zero_digit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

       special_character = "~" | "!" | "@" | "#" | "$" | "%" | "^" | "&" | "*"
                         | "(" | ")" | "-" | "_" | "+" | "=" | "{" | "}" | "["
                         | "]" | "|" | "\" | ":" | ";" | """ | "'" | "<" | ">"
                         | "," | "." | "?" | "/"

               new_line = ( "\n" | "\r" | "\r\n" ) ;
```

Obsługa błędów, przykłady komunikatów

Na każdym etapie przetwarzania danych znajdowała się będzie struktura, do której będą zapisywane komunikaty dotyczące wykrytych błędów. Na sam koniec procesu interpetacji lub gdy wykryty błąd będzie niemożliwy do pominięcia użytkownik zostanie powiadomiony o tym przez wyjście standardowe, a proces interpretacji zakończy się błędem i nie zostanie ukończony.

Sposób uruchomienia, wej/wyj.

```
$ ./my-language --help
Usage: ./my-language [OPTIONS] INPUT
Options:
  -h [ --help ]                         produce help message
  -l [ --lexer ]                        create a file with lexer output (with
                                        .l extention)
  -p [ --parser ]                       create a file with result of syntax
                                        analysis (with .p extention)
  -s [ --semantic ]                     create a file with result of semantic
                                        analysis (with .s extention)
  -i [ --intermediate_representation ]  create a file with result of
                                        intermediate representation(with .ir
                                        extention)
```

Analiza wymagań funkcjonalnych i niefunkcjonalnych

Zwięzły opis modułów, obiektów i interfejsów, lista tokenów, realizacja przetwarzania w poszczególnych komponentach

Lista tokenów:

- END_OF_FILE
- LITERAL
- IDENTIFIER
- COMMA
- DOT
- LPARENT
- RPARENT
- LBRACE
- RBRACE
- SEMICOLON
- COLON
- ASSIGN
- AT
- PLUS
- MINUS
- STAR
- SLASH
- PERCENT
- LEQ
- LESS
- GEQ
- GREATER
- EQ
- NEQ
- RARROW
- PIPE
- ONELINECOMMENT

- MULTILINECOMMENT
-
- FN
- IF
- ELSE
- STRUCT
- GLOBAL
- MUT
- RETURN

Lista słów kluczowych:

- fn
- if
- else
- while
- struct
- global
- mut
- return
- as
- true
- false

Wykorzystane struktury danych

- tablica dynamiczna,
- graf acykliczny (drzewo),
- tablica symboli,

Przykład wykorzystania struktury danych

- tablica dynamiczna: przechowywanie lexemów, przechowywanie informacji o błędach,
- graf acykliczny (drzewo): struktura w analizatorze składniowym.
- tablica symboli: dla każdego `zasięgu` (bloku kodu) oddzielna tablica mająca postać hash-mapy. Same zaś zasięgi na stosie zasięgów.

Formy pośrednie

1. Kod źródłowy -> analizator leksykalny
   - analizator leksykalny pobiera wejściowe dane ze strumienia (`std::istream`)
2. analizator leksykalny -> analizator składniowy
   - analizator składniowy pobiera obiekty typu `Leksem` (`tokeny`) z analizatora leksykalnego
3. analizator składniowy -> analizator semantyczny
   - analizator składniowy generuje drzewo `AST`

Opis sposobu testowania Proces testowania będzie opierał się na stopniowym kolejkowym testowaniu - zawsze wejściem testu będzie strumień znaków. W zależności który etap.

- Analizator leksykalny:
  - wejście - ciąg znaków do przeprowadzenia analizy leksykalnej
  - wyjście - tablica leksemów + tablica błędów
  - test - porównywanie poszczególnych wartości leksemów i błędów wynikowych z oczekiwanymi
- Analizator składniowy:
  - wejście - ciąg znaków do przeprowadzenia analizy leksykalnej
  - wyjście - postać drzewa wyjścia parsera sprowadzona do postaci wyświetlalnej (jako `string`)
  - test - porównanie wyjścia z oczekiwanym wynikiem