Matreshka lang
Руководство разработчика

Version 1.0.0

Работа Бабакова Ильи Андреевича
и Запорожца Артёма Сергеевича 11К
(babakovi679@gmail.com, zarchy888@gmail.com)

10 июля 2024 г.

Содержание

1 Введение

Данное руководство предназначено для тех смелых, решившихся программировать на нашем языке. В нем будут покрыты основные концепты нашего языка для лучшего понимания его работы "под капотом".

2 Структура языка. BNF

2.1 Базовые концепты языка

Matreshka является языком с процедурной парадигмой. Все выражения в нем выполняются в круглых скобках. Для вычисления выражений используется польская нотация. Такое решение было вызвано желанием приблизить визуально язык к его идейному вдохновителю - языку lisp. Грамматику можно найти в разделе Дополнения.

2.2 Типы, представленые в языке

Matreshka слабо типизируема, что означает, что нам ничего не мешает переопределять тип переменных. Все выражения представляются в виде списков, которые затем просто представлять как бинарные деревья. Язык Matreshka предлагает пользователю на выбор 5 типов данных:

Таблица 1: Типы данных языка

Boolean	хранит логические Правда и Ложь.
String	строковый тип данных, C-style null-terminated
Char	буквенный литерал по стандарту ASCII
Integer	целочисленный тип данных
Floating point	тип данных с плавающей точкой
NIL	несуществующий объект, пустой список

2.3 Этапы трансляции

Лексический анализатор работает на основе ДКА. После его работы мы получаем массив лексем. Затем Синтаксический анализатор проверяет правильность порядка лексем. Для разбора выражений мы используем метод рекурсивного спуска. После чего осуществляется контроль контекстных условий. Внутреннее представление представляет из себя ПОЛИЗ, который потом используется для интерпертации.

2.3.1 Лексический анализ

Лексический анализ — процесс аналитического разбора исходный код на одном с целью получения на выходе последовательности символов, называемых лексемами (Последовательность символов в исходном коде, которые соответствуют заданным предопределенным языковым правилам для каждой лексемы, которая должна быть указана в качестве допустимого токена). Каждой лексеме присваивается уровень и они объединяются в токен.

```
struct Token {
    std::string token;
    int level;

    Token(){}
    Token(std::string token, int level): token(token), level(level){}
};
```

2.3.2 Синтаксический анализ

Синтаксический анализатор работает по принципу рекурсивного спуска. Все токены приходят на вход синтаксическому анализатору в порядке их нахождения в коде. У токенов есть уровни, которые используются для упрощения разбора выражений и соблюдения приоритета операций.

2.3.3 Семантический анализ

Семантический анализ – процесс проверяющий правильность текста исходной программы с точки зрения семантики входного языка. Кроме непосредственно проверки, семантический анализ должен выполнять преобразования текста, требуемые семантикой входного языка (проверка правильной последовательности токенов).

Используется структура TID для проверки уникальности/наличия переменных

```
struct TID
{
    std::vector<std::string> name_;
    std::vector<std::string> type_;

    // Adding an identifier to the TID
    void push_name(std::string s, std::string t);

    // Checking for the presence of an ID in the TID
    std::string check_name(std::string s);

    // Replacing the ID type
    void replace(std::string s, std::string t);
```

```
    std::vector<std::string> get_names() { return name_; }
    std::vector<std::string> get_types() { return type_; }
};
```

Эти структуры объединены в дерево для проверки локальности переменной

```
class TID_tree
{
public:
    TID tid;
    TID_tree *next = nullptr, *prev = nullptr;

    // Adding a new TID to TID_tree and changing the current TID (cur_ID)
    void create_TID();

    void create_TID(TID tid);

    // Deleting a current TID from TID_tree and changing the current TID (cur_ID
       )
    void del_TID();

    // Checking for the presence of an ID in the TID_tree
    std::string check_name(std::string s);

    // Checking for the presence of an identifier in TID_tree (without throwing
       an error)
    bool find(std::string s);

    // Replacing the ID type in TID_tree
    void replace(std::string s, std::string t);
};
```

Далее представлен класс, содержащий TID для функций

```
struct func_TID
{
    std::vector<std::string> name_;
    std::vector<std::string> type_;
    std::vector<int> num_of_lex;
    std::vector<TID> tid_;

    // Adding an identifier of function to the TID
    void push_name(std::string s, int n, std::vector<std::string> p_name);

    // Checking for the presence of an ID in the TID
    std::string func_type(std::string s);
```

```
    // Checking for the presence of an identifier in func_TID (without throwing
       an error)
    bool find(std::string s);

    void set_type(std::string s, std::string t);

    void set_params_type(std::string s, std::vector<std::string> p_type);

    int get_idx(std::string s);

    int get_num_of_lex(int i);

    TID get_tid_params(int i);
};
```

3 Примеры кода

Этот код переводит введеное число в двоичную систему счисления:

```
(tovarisch x 0)
(sprosi x)
(napishi x " binary: ")
(tovarisch res "")
(zhivi
(tovarisch res (+ (v_stroku (mod x 2)) res))
(tovarisch x (/ x 2))
(umri_kogda (= x 0)))
(napishi res)
```

4 Лексемы языка

4.1 Токены

Токены являются структурными единицами трансляции и представляют собой пары < Лексема , уровень >

4.2 Операторы

4.2.1 Унарные операторы

- ! — Возвращает произведение первых n целых чисел
- ne — логическое отрицание (НЕ, !)

4.2.2 Бинарные операторы

- mod — Возвращает математически верный остаток от деления по модулю натурального числа.
- takzhe — логическая конъюнкция (И, &)
- libo — логическая дизъюнкция (ИЛИ, ||)
- = — логическая эквивалентность
- != — логическая неэквивалентность
- > — больше
- < — меньше
- >= — больше или равно
- <= — меньше или равно
- pribav — увеличивает первый операнд на значение второго(если второй не указан, то используется 1)
- ubav — уменьшает первый операнд на значение второго(если второй не указан, то используется 1)

4.2.3 Небинарные операторы

Принимают неограниченное количество аргументов и выполняют операции последовательно .

- \+ — Возвращает сумму всех операндов: $a_1 + a_2 + a_3 + ... + a_n$
- \- — Возвращает разность первого операнда и суммы всех последующих: $a_1 - (a_2 + a_3 + ... + a_n)$
- * — Возвращает произведение всех операндов: $a_1 * a_2 * a_3 * ... * a_n$
- / — Возвращает частное первого операнда и произведения всех последующих: $a_1/(a_2 * a_3 * ... * a_n)$
- max — Возвращает максимальный значение среди операндов.
- min — Возвращает минимальное значение среди операндов.

4.2.4 "Сложные"операторы

Операторы с уникальной структурой и своими требованиями.

loop_for

```
<loop_for> ::= "idi_poka <var> " ne_stanet " <num> (<oper>)*
```

Оператор цикла, который выполняет находящиеся в нем операторы(<oper>) пока переданная ему переменная(<var>) не достигнет значения численного литерала(<num>). Это аналог цикла for из языка C++, но с неизменяемым шагом +1.

loop

```
<loop> ::= "zhivi " (<oper>)* "(umri_kogda (" <cond_oper> "))"
```

Оператор цикла, который выполняет находящиеся в нем операторы(<oper>) и после каждой итерации проверяет истинность логического выражение(<cond_oper>), если оно истинно управление передается началу цикла, иначе заканчивается выполнение цикла.

if

```
<if> ::= "esli (" <cond_oper> ") (" (<oper>)* ") (" (<oper>)* ")"
```

Условный оператор if - самый простой оператор для реализации ветвления. Он используется для определения того, будет ли выполняться определенный оператор или блок операторов, т.е. если определенное условие истинно, тогда выполняется первый блок операторов иначе второй.

write

```
<write> ::= "napishi " (<arg>)+
```

Оператор для вывода информации. По умолчанию выводит в стандартный поток вывода.

read

```
<read> ::= "sprosi " (<var>)+
```

Оператор для приема входных данных со стандартного потока ввода.

setf

```
<setf> ::= "tovarisch " <var> " " <arg>
```

Объявление переменной.

to_string

```
<to_str> ::= "v_stroku " <arg>
```

Перевод типа данных в string.

4.3 Функции

Объявление функции

```
<func> ::= "func " <name> " (" <params> ") (" (<oper>)* ") (" <return> ")"
<params> ::= <var> <params_extr> | <empty>
<params_extr> ::= " " <var> <params_extr> | <empty>
<return> ::= "verni " <arg>
<empty> ::= " "
```

Вызов функции

```
<func_call> ::= <var> <params>
<params> ::= <var> <params_extr> | <empty>
<params_extr> ::= " " <var> <params_extr> | <empty>
```

4.4 Литералы

4.4.1 Численные литералы

Численные литералы представлены множеством действительных десятичных чисел. Ограничений на длину нет (в разумных пределах).

4.4.2 Буквенные литералы

Все ASCII символы, представимые компьютером.

4.4.3 Остальные литералы

Логические литералы — PRAVDA(true) и LOZH(false)

5 Дополнения

5.1 Грамматика языка Matreshka

<s_exp> ::= "(" <oper> ") " <s_exp> | <empty>
<arg> ::= <var> | <liter> | "(" <oper> ")"
<var> ::= <name>
<name> ::= <let> | <name_extr>
<name_extr> ::= <let> <name_extr> | <num> <name_extr> | <empty>
<let> ::= [a-z]
<num> ::= [0-9]
<liter> ::= <sign> <nums> | "\"" <str> "\"" | "'" <char> "'" | "PRAVDA" | "LOZH"
<char> ::= [a-z] | [0-9]
<sign> ::= <empty> | "+" | "-"
<nums> ::= <num> <nums_extr>
<nums_extr> ::= <num> <num_extr> | "." <d_num> | <empty>
<d_num> ::= <num> <d_num_extr>
<d_num_extr> ::= <num> <d_num_extr> | <empty>
<str> ::= <char> <str_extr>
<str_extr> ::= <char> <str_extr> | <empty>
<oper> ::= <easy_oper> | <hard_oper> | <func_call> | <empty>
<easy_oper> ::= <simple_oper> | <cond_oper>

<argf> ::= <var> | <sign> <num> | "(" <oper> ")"
<simple_oper> ::= <simple> (" " <argf>)+
<cond_oper> ::= <cond> (" " <arg>)+
<simple> ::= "+" | "-" | "*" | "/" | "max" | "min"
<cond> ::= "takzhe" | "libo" | "=" | "!=" | ">" | "<" | ">=" | "<="
<hard_oper> ::= <loop_for> | <loop> | <if> | <write> | <read> | <mod> |
<not> | <incf> | <decf> | <fact> | <setf> | <func_call>
<loop_for> ::= "idi_poka " <var> " ne_stanet " <num> (<oper>)*
<loop> ::= "zhivi " (<oper>)* "(umri_kogda (" <cond_oper> "))"
<if> ::= "esli (" <cond_oper> ") (" (<oper>)* ") (" (<oper>)* ")"
<func> ::= "func " <name> " (" <params> ") (" (<oper>)* ") (" <return> ")"
<params> ::= <var> <params_extr> | <empty>
<params_extr> ::= " " <var> <params_extr> | <empty>
<write> ::= "napishi " <arg>
<read> ::= "sprosi " <var>
<mod> ::= "mod " <argf> " " <argf>
<not> ::= "ne " <arg>
<fact> ::= "! " <argf>
<incf> ::= "pribav " <argf> (" " <argf>)?
<decf> ::= "ubav " <argf> (" " <argf>)?
<return> ::= "verni " <arg>
<setf> ::= "tovarisch " <var> " " <arg>
<func_call> ::= <var> <params>
<empty> ::= " "

5.2 Уровни лексем

Таблица 2: Уровни лексем

1	Сложные операторы
2	Переменные
3	Буквенные и численные литералы
4	Простые операторы
5	Скобки
404	Ошибка. Служебный уровень