Московский государственный университет
имени М. В. Ломоносова

Факультет вычислительной математики и кибернетики

Язык программирования Yuma

Юрий Фетисов
Максим Кóдрян,
Антон Гнатенко,

Москва, 2016.

Содержание

Общая информация

Язык программирования **Yuma** – интерпретируемый процедурный язык программирования общего назначения, испытавший влияние таких языков, как **C**, **Pascal** и **Python**.

Интерпретатор языка **Yuma** написан на **C++** и содержит 1750 строк кода. При написании использовались возможности объектно-ориентированного программирования, поэтому интерпретатор можно условно разделить на несколько блоков, которые легко было реализовывать по отдельности:

1) Лексический анализатор,

2) Синтаксический анализатор,

3) Блок выполнения.

В целях координации действий использовался **GitHub** – крупнейший веб-сервис для совместной разработки и поддержки IT-проектов, основанный на популярной системе контроля версий **Git**.

Помимо стандартных требований к модельному языку, были реализованы также следующие дополнительные возможности:

1) Динамическое изменение типа переменной,

2) Массивы,

3) Лямбда-исчисление.

Подробную информацию об этих возможностях и различных аспектах языка можно найти в соответствующем разделе отчёта.

Устройство языка

Лексика

При написании программ на языке **Yuma** используются идентификаторы, операторы, строковые и числовые константы и ключевые слова.

Идентификатор в языке **Yuma** — это любая последовательность букв и цифр, начинающаяся с буквы, не совпадающая с оператором или ключевым словом.

Операторами являются следующие последовательности символов:

`==`	равно	`/`	деление	`;`	точка с запятой
`!=`	не равно	`=`	присваивание	`,`	запятая
`&&`	логическое И	`(`	откр. скобка	`<`	меньше
`||`	логическое ИЛИ	`)`	закр. скобка	`>`	больше
`!`	логическое НЕ	`{`	от. ф. скобка	`<=`	меньше или равно
`+`	плюс	`}`	з. ф. скобка	`>=`	больше или равно
`-`	минус	`[`	от. кв. скобка	`lire`	ввод
`*`	умножение	`]`	з. кв. скобка	`ecrire`	вывод

Числовая константа — это последовательность цифр, содержащая, возможно, одну точку как внутренний символ.

Строковая константа — это любая последовательность символов, заключённая в кавычки.

Ключевыми словами являются следующие последовательности символов:

```
while     do        if        then      else      array
```

Синтаксис

Для описания синтаксиса языка и проведения синтаксического анализа использовались инструменты теории формальных грамматик. Была построена соответствующая грамматика (см. ниже), на её основе была построена система функций, реализующая синтаксический анализ методом рекурсивного спуска.

В грамматике символы **{**, **}**, **[** и **]** понимаются в традиционном для грамматик смысле, а символы {, }, [и] являются символами языка.

```
prog -> { body }
body -> com; { com;}
com -> id = [expr | array] | id[expr] = expr |
         if (expr) then { body } [; | else { body };] |
         while (expr) do { body }; |
         lire id; |
         ecrire expr {, expr } ;|
         id = func { body };
expr -> sum [ == | < | > | <= | >= | != ] sum | sum
sum -> prod { [+ | - | || ] prod }
prod -> oprnd { [ * | / | && ] oprnd }
oprnd -> id | id[expr] | num | !oprnd | (expr)
id -> char | id_char | id_digit
digit_line -> digit_digit_line
num -> digit_line[.digit_line]
char -> a | b | ... | z | A | B | ... |Z
digit -> 0 | 1 | ... | 9
```

Важно помнить, что любая команда должна заканчиваться точкой с запятой. Это касается и команд, заканчивающихся фигурной скобкой:

```
if (expr) then {body} ;

if (expr) then {body} else {body} ;

while (expr) then {body} ;
```

Семантика простых типов и выражений

В языке **Yuma** предусмотрены простые и сложные типы данных. К простым типам относятся числовой и строковый типы, к сложным – массивы и функции. Константы могут быть только простого типа. Тип переменной может динамически меняться во время выполнения программы, это происходит при присваивании переменной значения определённого типа либо объявлении переменной как функции или массива (см. соответствующие разделы отчёта).

Числовой тип хранится в переменных типа **double** языка **C++** и объединяет в себе целочисленный и вещественный типы. В случаях, когда это необходимо (например, при индексировании массивов), он приводится к **int** или **unsigned int**. Для числового типа определены все стандартные операции.

Строковый тип хранится в объектах класса **string** языка **C++**. Для строк определены операции присваивания, сложения (конкатенация) и лексико-графического сравнения.

Логические выражения, используемые в вычислениях, условном операторе и операторе цикла всегда вычисляются полностью, слева направо.

Одной из главных особенностей языка является то, что типы переменных определяются динамически и могут меняться по ходу выполнения программы. Это справедливо в том числе для массивов и функций, то есть функция, например, может изменить внутри себя свой собственный тип. Из-за этого проверка семантики возможна только на этапе выполнения программы.

Каждая переменная при описании должна быть инициализированна каким-то значением для определения типа.

Оператор **lire** может быть применён только к переменной известного типа.

Массивы

Массив в **Yuma** – это вектор, элементами которого являются переменные простых типов Допускаются массивы, у которых типы элементов различаются.

Для доступа к элементу используется оператор **[]**, внутри квадратных

скобок может стоять любое выражение, тип значения которого – числовой. Допускаются нецелые индексы, в этом случае их значение приводится к **`unsigned int`** с отбрасыванием дробной части. Не допускаются двойные, тройные и т.д. индексы:

```
a[1][2] = expr;  #error
```

Программистам предлагается организовывать многомерные массивы с помощью одномерных.

При объявлении массива его элементы получают неопределённый тип. Таким образом, для считывания элементов массива (например, в цикле), требуется до этого (в цикле) определить тип этих элементов. Например, для создание массива чисел достаточно сначала занести число `1` в каждый элемент.

Функции и лямбда-исчисление

Функция – это переменная особого типа, значением которого является алгоритм, который эта функция реализует. Таким образом, переменная любого типа может быть в произвольном месте программы переведена в функцию, а функция – в переменную любого типа. Частным случаем этого правила является возможность динамического изменения тела функции.

В языке **`Yuma`** одна общая область видимости, поэтому идентификаторы, используемые внутри тела функции не должны совпадать с идентификаторами, используемыми в основной программе и других функциях. Если тело функции было выполнено, все переменные, объявленные в ней, остаюся доступными из других частей программы.

Общность области видимости накладывает также важное ограничение: у функций в языке **`Yuma`** нет параметров. Рекомендуется передавать параметры с помощью введения дополнительных переменных (по аналогии с передачей параметров через регистры в языке ассемблера).

Обработка ошибок

Если обнаружена ошибка, выполнение текущего этапа интерпретации прекращается, и на экран выводится сообщение, с описанием ошибки и информацией, на каком этапе она была обнаружена.

Лексические ошибки:

```
Lexic error:

Bad identifier        12df = expr;
Bad number            a = 12.34.56;
Not paired commas     a = «abc;
```

Синтаксические ошибки:

```
Syntax error:

Sign { expected at the begining of the program
Sign } expected at the end of program
Sign } expected at the end of IF
Sign { expected at the begining of THEN
Sign { expected at the begining of ELSE
Sign } expected at the end of ELSE
THEN expected after IF
Sign { expected at the beginning of WHILE
Sign } expected at the end of WHILE
DO expected after WHILE
Wrong <func>-command syntax: expected identifier
Array index must be a number
Sign ] expected after array index
Identifier must follow LIRE
-array index must be a calculateble expression-
multidimensional arrays are strictly prohibited
```

```
sign = expected
Wrong <func()>-command syntax: ')' expected
Sign = or () expected
Sign ; expected at the end of command
Closing bracket expected at the end of expression
Operand expected
```

Ошибки времени выполнения:

```
Runtime error:
```

`undefined identifier` не определён тип одной из переменных.
`Types mismatch in <operator>` типы операндов в операторе `<operator>` не совпадают

`Bad <operand> arguments` неподходящие типы операндов в операторе `<operator>`
`<internal error>` ошибка в коде интерпретатора. Обратитесь в службу поддержки

```
Logic expression expected in if or while condition
```

`Tried to run not a function` оператор вызова функции применён к переменной другого типа

```
array index must be a number
```

`only an array may have nice [] after it` оператор индексирования массива применён к переменной другого типа

`we do not <lire> arrays, functions or undefiend values` попытка считывания переменной неподходящего типа.
`array border violation` выход за границы массива

Примеры программ

Вычисление квадрата числа:

```
{
    i = 0;
    lire i;
    i = i * i;
    ecrire i;
}
```

Вычисление N-ого числа Фибоначчи:

```
{
    i = 0;
    j = 2;
    prnum = 1;
    num = 1;
    buf = 1;
    lire i;
    if ((i == 1) || (i == 2)) then
    {
        ecrire 1;
    }
    else
    {
        while (j < i) do
        {
            buf = num;
            num = num + prnum;
            prnum = buf;
            j = j + 1;
        };
        ecrire num;
    };
}
```

Поиск максимума в массиве из 10 элементов:

```
{
    a = array;
    k = 0;
    N = 10;
    while (k < N) do
    {
        a[i] = 0; # определение элементов типом number
    };
    k = 0;
    while (k < N) do
    {
        lire a[i];
    };
    k = 1;
    max = a[0];
    while (k < N) do
    {
        if (max < a[k]) then
        {
            max = a[k];
        };
    };
    ecrire “Max is ”, max;
}
```

Вызов функции с переопределением:

```
{
    f = func {ecrire 1; f = func {ecrire 2}; ecrire 3;};
    f();     # будет выведено 1 3
    f();     # будет выведено 2
}
```

Вопрос к коллоквиуму по языку Yuma. Что будет напечатано в результате выполнения следующей программы?

```
{
        a = func {
                    ecrire "before";
                    a = func {
                                ecrire "new one";
                             };
                    ecrire "after";
                    a();
                    ecrire "after rec";
                };
        a();
        a();
}
```