МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»
Институт компьютерных наук и кибербезопасности
Высшая школа технологий искусственного интеллекта
Направление **02.03.01** : Математика и компьютерные науки

Лабораторная работа № 4
«Доработка компилятора языка milan»
по дисциплине «Теория алгоритмов»
Вариант 21

Обучающийся: _____ Яшнова Дарья Михайловна
группа 5130201/20002

Руководитель: _____ Востров Алексей Владимирович

«___»_____ 2025г

Санкт-Петербург, 2025

Содержание

Введение

Доработка комплятора языка MiLan согласно самостоятельно выбранному варианту.

Целый и вещественный типы. Автоматическое и явное с помощью префиксов (int) и (float) приведение типов.

1 Математическое описание

1.1 Описание языка Milan

Язык Милан — учебный язык программирования, описанный в учебнике. Программа на Милане представляет собой последовательность операторов, заключённых между ключевыми словами **Begin** и **End**. Операторы отделяются друг от друга точкой с запятой. После последнего оператора в блоке точка с запятой не ставится. Компилятор **CMilan** не учитывает регистр символов в именах переменных и ключевых словах.

В базовую версию языка Милан входят следующие конструкции: константы, идентификаторы, арифметические операции над целыми числами, операторы чтения чисел со стандартного ввода и вывода чисел на стандартный вывод, оператор присваивания, условный оператор и оператор цикла с предусловием.

Компилятор **CMilan** включает три компонента:

1. Лексический анализатор;
2. Синтаксический анализатор;
3. Генератор команд виртуальной машины Милана.

Лексический анализатор посимвольно читает из входного потока текст программы и преобразует группы символов в лексемы (терминальные символы грамматики языка Милан). При этом он отслеживает номер текущей строки, который используется синтаксическим анализатором при формировании сообщений об ошибках. Также лексический анализатор удаляет пробельные символы и комментарии.

Синтаксический анализатор читает последовательность лексем, сформированную лексическим анализатором, и проверяет её соответствие грамматике языка Милан. Для этого используется метод рекурсивного спуска. Если обнаруживается ошибка, анализатор формирует сообщение с описанием и номером строки. В процессе анализа генерируются машинные команды, соответствующие конструкциям языка.

Генератор кода формирует внешнее представление генерируемого кода. Все компоненты объединяются «драйвером», который анализирует аргументы командной строки и запускает метод `parse()`.

CMilan является однопроходным компилятором, где синтаксический анализ и генерация кода выполняются совместно.

Виртуальная машина Милан включает память для команд и данных, стек для промежуточных вычислений, поддерживает арифметические операции, условные и безусловные переходы, ввод-вывод, а также работу с целыми и вещественными числами. Выполнение программы продолжается до команды остановки или ошибки.

1.2 Грамматика языка Милан

Контекстно-свободные грамматики (КС-грамматики) описываются продукциями вида $A \to \beta$, где A — нетерминал, а β — последовательность терминалов и нетерминалов.

1.2.1 Контекстно-свободные грамматики

Контекстно-свободные грамматики (КС-грамматики) относятся ко второму типу грамматик по классификации Хомского. Эти грамматики описываются продукциями вида $A \to \beta$, где A — нетерминал, а β — последовательность терминалов и нетерминалов.

LL(k)-грамматики LL(k)-грамматики позволяют выполнять нисходящий синтаксический анализ, просматривая входную цепочку слева направо для построения канонического левого вывода. Здесь k указывает, сколько символов из непрочитанной части входной

цепочки используется для принятия решений. LL(1)-грамматики, например, используют только один символ впереди.

Функции FIRST(α) и FOLLOW(A) помогают определить:

- С каких символов могут начинаться цепочки, выводимые из α;
- Какие символы могут следовать за A в любых формах предложения.

Если множества FIRST(α) и FIRST(β) не пересекаются, можно однозначно выбрать между правилами $A \to \alpha$ и $A \to \beta$. Если FIRST(α) и FOLLOW(A) не пересекаются, можно выбрать между $A \to \alpha$ и $A \to \epsilon$.

Грамматика имеет свойство LL(k), если для любых двух цепочек левых выводов:

$$S \Rightarrow^* wAx \Rightarrow^* w\alpha x \Rightarrow^* wu$$

и

$$S \Rightarrow^* wAx \Rightarrow^* w\beta x \Rightarrow^* wv$$

из условия FIRST(u) = FIRST(v) следует $\alpha = \beta$.

1.2.2 Грамматика языка Милан

Грамматика языка Милан в расширенной форме Бэкуса-Наура:

⟨*program*⟩ ::= `'begin'` ⟨*statementList*⟩ `'end'`
⟨*statementList*⟩ ::= ⟨*statement*⟩ `';'` ⟨*statementList*⟩ | ϵ
⟨*statement*⟩ ::= ⟨*ident*⟩ `':='` ⟨*expression*⟩
| `'if'` ⟨*relation*⟩ `'then'` ⟨*statementList*⟩ `'else'` ⟨*statementList*⟩ `'fi'`
| `'while'` ⟨*relation*⟩ `'do'` ⟨*statementList*⟩ `'od'`
| `'write'` `'('` ⟨*expression*⟩ `')'`
⟨*expression*⟩ ::= ⟨*term*⟩ {⟨*addop*⟩ ⟨*term*⟩}
⟨*term*⟩ ::= ⟨*factor*⟩ {⟨*multop*⟩ ⟨*factor*⟩}
⟨*factor*⟩ ::= ⟨*ident*⟩ | ⟨*number*⟩ | `'('` ⟨*expression*⟩ `')'`
⟨*relation*⟩ ::= ⟨*expression*⟩ ⟨*cmp*⟩ ⟨*expression*⟩
⟨*addop*⟩ ::= `'+'` | `'-'`
⟨*multop*⟩ ::= `'*'` | `'/'`
⟨*cmp*⟩ ::= `'=='` | `'!='` | `'<'` | `'<='` | `'>'` | `'>='`
⟨*ident*⟩ ::= ⟨*letter*⟩ {⟨*letter*⟩ | ⟨*digit*⟩}
⟨*letter*⟩ ::= `'a'` | `'b'` | ... | `'Z'`
⟨*digit*⟩ ::= `'0'` | `'1'` | ... | `'9'`
⟨*number*⟩ ::= ⟨*digit*⟩{⟨*digit*⟩}[`'.'`{⟨*digit*⟩}]

1.2.3 Расширенная грамматика с типами данных

⟨*program*⟩ ::= 'begin' ⟨*statementList*⟩ 'end'
⟨*statementList*⟩ ::= ⟨*statement*⟩ ';' ⟨*statementList*⟩ | ϵ
⟨*statement*⟩ ::= ⟨*ident*⟩ ':=' ⟨*expression*⟩
| ⟨*type*⟩ ⟨*ident*⟩ [',' ⟨*ident*⟩]* ['=' ⟨*expression*⟩]
| 'if' ⟨*relation*⟩ 'then' ⟨*statementList*⟩ ['else' ⟨*statementList*⟩] 'fi'
| 'while' ⟨*relation*⟩ 'do' ⟨*statementList*⟩ 'od'
| 'write' '(' ⟨*expression*⟩ ')'
⟨*expression*⟩ ::= ⟨*term*⟩ {⟨*addop*⟩ ⟨*term*⟩}
⟨*term*⟩ ::= ⟨*factor*⟩ {⟨*mulop*⟩ ⟨*factor*⟩}
⟨*factor*⟩ ::= ⟨*ident*⟩
| ⟨*number*⟩
| '(' ⟨*expression*⟩ ')'
| '(' ⟨*type*⟩ ')' ⟨*ident*⟩
| '(' ⟨*type*⟩ ')' ⟨*number*⟩
| '(' ⟨*type*⟩ ')' '(' ⟨*expression*⟩ ')'
⟨*relation*⟩ ::= ⟨*expression*⟩ ⟨*cmp*⟩ ⟨*expression*⟩
⟨*addop*⟩ ::= '+' | '-'
⟨*mulop*⟩ ::= '*' | '/'
⟨*cmp*⟩ ::= '==' | '!=' | '<' | '<=' | '>' | '>='
⟨*type*⟩ ::= 'int' | 'float'
⟨*ident*⟩ ::= ⟨*letter*⟩ {⟨*letter*⟩ | ⟨*digit*⟩}
⟨*letter*⟩ ::= 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'
⟨*digit*⟩ ::= '0' | '1' | ... | '9'
⟨*number*⟩ ::= ⟨*digit*⟩{⟨*digit*⟩} ['.'{⟨*digit*⟩}]

1.2.4 Синтаксические диаграммы

На рис.1 - 3 представлены измененные синтаксические диаграммы.

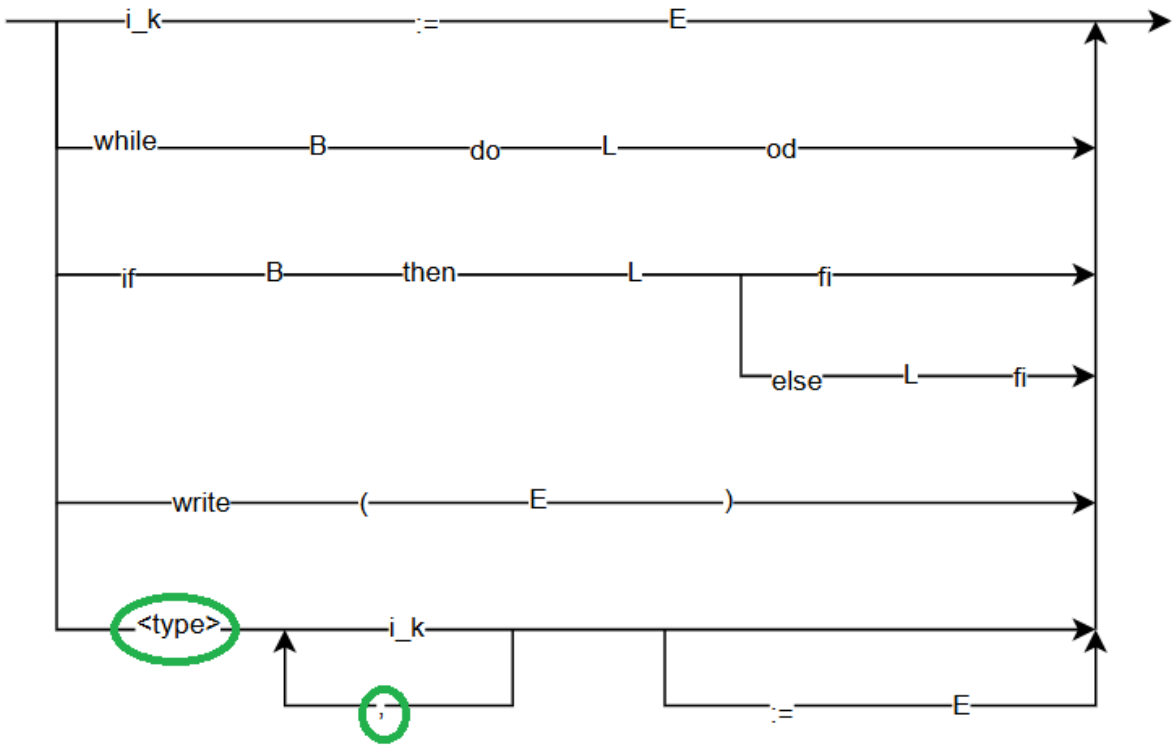

Рис. 1: Синтаксическая диаграмма языка Milan

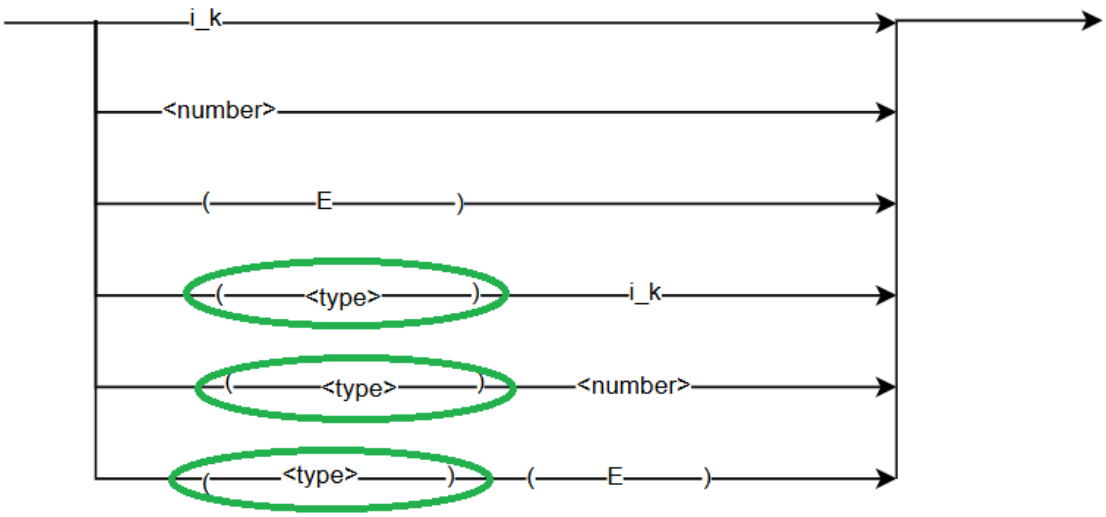

Рис. 2: Диаграмма правила <factor>

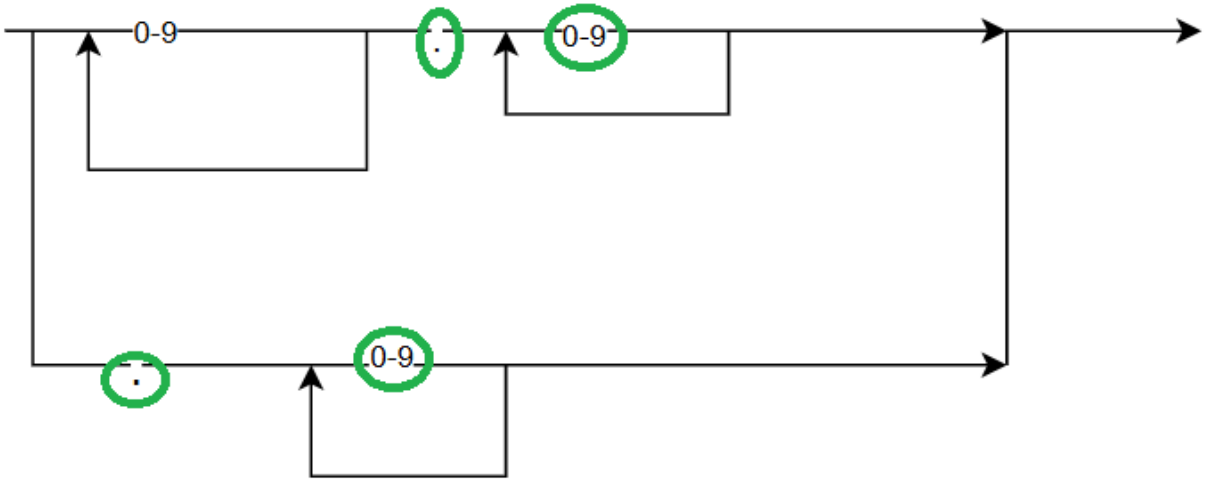

Рис. 3: Диаграмма правила <number>

2 Особеннности реализации

2.1 Изменения в виртуальной машине

В проекте виртуальной машины была добавлена новая команда `FPRINT` для вывода дробных чисел. Ниже описаны все изменения, внесённые в исходный код.

2.1.1 Изменения в заголовочном файле (vm.h)

В файл `vm.h` были внесены следующие изменения, описанные в листинге 1:

Листинг 1: Добавление операции FPRINT

```
typedef enum {
    NOP = 0,
    // ... другие операции ...
    FPRINT   // <--- ДОБАВЛЕНО
} operation;
```

2.1.2 Обновление таблицы опкодов (vm.c)

В файле `vm.c` обновлена таблица информации об опкодах (листинг 2):

Листинг 2: Обновление opcodes_table

```
opcode_info opcodes_table[] = {
    // ... другие команды ...
    {"FPRINT",   0}  // <--- ДОБАВЛЕНО
};
```

2.1.3 Реализация логики вывода дробных чисел

Была добавлена новая функция для вывода дробных чисел. **Входные данные**

- `scaled_value` (`int`) – целочисленное значение, представляющее масштабированную версию вещественного числа.

Выходные данные

- Вывод в стандартный поток (`stdout`) – вещественное число, вычисляемое по формуле:

$$\text{real\_value} = \frac{\text{scaled\_value}}{\text{SCALE\_FACTOR}}$$

  – Число выводится с 3 знаками после запятой (формат `%.3f`).

Функция vm_write_float указана на Листинге 3.

Листинг 3: Функция vm_write_float

```
void vm_write_float(int scaled_value) {
    double real_value = (double) scaled_value / SCALE_FACTOR;
    printf("%.3f\n", real_value);
    fflush(stdout);
}
```

В листинге 4 представлен обработчик команды в функции `vm_run_command`:

Листинг 4: Обработчик FPRINT

```
case FPRINT:
    vm_write_float(vm_pop());
    break;
```

2.1.4 Изменения в парсере (vmparse.y)

Обновлён список токенов и грамматика (листинг 5-6):

Листинг 5: Добавление токена FPRINT

```
%token T_FPRINT   // <--- ДОБАВЛЕНО
```

Листинг 6: Правило грамматики для FPRINT

```
| T_INT T_COLON T_FPRINT { put_command($1, FPRINT, 0); }
```

2.1.5 Правило для распознавания ключевого слова

Добавлено правило для распознавания ключевого слова (листинг 7):

Листинг 7: Правило лексера для FPRINT

```
"FPRINT"        { return T_FPRINT; }
```

После изменений необходимо выполнить команды из листинга 8:

Листинг 8: Команды генерации файлов

```
bison -d vmparse.y    # Генерация vmparse.tab.c и vmparse.tab.h
flex vmlex.l          # Генерация lex.yy.c
```

2.2 Изменения компилятора

2.2.1 Новые токены (scanner.h)

До модификации кода был всего 21 токен компилятора. Были добавлены 3 токена, представленные в листинге 9:

Листинг 9: Перечисление токенов

```
enum Token {
          ...
          T_INT,          // Ключевое слово "int"
          T_FLOAT,        // Ключевое слово "float"
          ...
          T_COMMA         //Токен запятой
};
```

2.2.2 Флаг для определения типа числа (scanner.h)

Добавлен флаг для определения типа числа (целое/нецелое). Изменения представлены в листинге 10.

Листинг 10: Флаг для определения типа числа

```
bool literalIsFloat_; // True if the last T_NUMBER was parsed from a
    float string like "1.23" or ".5"
```

Добавлен метод для проверки типа числа (листинг 11):

Листинг 11: Функция isLiteralFloat

```
bool isLiteralFloat() const // To inform parser if T_NUMBER was parsed
    from a float format
{
     return literalIsFloat_;
}
```

2.2.3 Обработка нецелых чисел (scanner.cpp)

Модифицирован метод nextToken() для обработки нецелых чисел. **Входные данные**

- Текущий символ `ch_` (тип `char`) - первый символ анализируемого токена
- Входной поток `input_` (тип `istream&`) - источник символов для сканирования
- Константы:
  - `MAX_FRACTIONAL_DIGITS` - максимальное количество цифр в дробной части
  - `SCANNER_SCALE_FACTOR` - масштабирующий коэффициент для дробных чисел

Выходные данные

- Устанавливаются следующие поля класса:
  - `token_` (тип `Token`) - тип распознанного токена (для чисел - `T_NUMBER`)
  - `intValue_` (тип `int`) - числовое значение (масштабированное для дробных чисел)
  - `literalIsFloat_` (тип `bool`) - флаг, указывающий на дробное число

Изменения функции nextToken представлены в листинге 12.

Листинг 12: Изменения в nextToken

```
1  void Scanner::nextToken()
2  {
3      skipSpace();
4      literalIsFloat_ = false; // Reset for the new token
5
6      // Обработка чисел (целых и нецелых)
7      if (isdigit(ch_) || (ch_ == '.' && input_.peek() != EOF && isdigit(
           static_cast<unsigned char>(input_.peek())))) {
8          string numStr;
9          string integerPartStr;
10         string fractionalPartStr;
11         bool dotSeen = false;
12
13         // Целая часть
14         while (isdigit(ch_)) {
15             integerPartStr += ch_;
16             nextChar();
17         }
18
19         if (ch_ == '.') {
20             dotSeen = true;
21             nextChar(); // consume '.'
22             // Дробная часть
23             while (isdigit(ch_)) {
24                 if (fractionalPartStr.length() < MAX_FRACTIONAL_DIGITS) {
25                     fractionalPartStr += ch_;
26                 }
27                 nextChar();
28             }
29         }
30
31         token_ = T_NUMBER;
32
33         long long integerValue = 0;
```

```
34          if (!integerPartStr.empty()) {
35              integerValue = stoll(integerPartStr);
36          }
37
38          if (dotSeen || !fractionalPartStr.empty()) {
39              literalIsFloat_ = true;
40              long long fractionalValue = 0;
41              if (!fractionalPartStr.empty()) {
42                  fractionalValue = stoll(fractionalPartStr);
43              }
44
45              // Масштабирование дробной части
46              long long currentFracScale = SCANNER_SCALE_FACTOR;
47              for (size_t i = 0; i < fractionalPartStr.length(); ++i) {
48                  currentFracScale /= 10;
49              }
50              if (currentFracScale == 0) currentFracScale = 1;
51
52              intValue_ = static_cast<int>(integerValue * SCANNER_SCALE_
                    FACTOR + fractionalValue * currentFracScale);
53          }
54          else { // Целое число
55              literalIsFloat_ = false;
56              intValue_ = static_cast<int>(integerValue);
57          }
58          return;
59      }
60      // ... остальной код ...
61  }
```

2.2.4 Инструкция для печати нецелых чисел

Добавлена новая инструкция для печати нецелых чисел(листинг 13):

Листинг 13: Изменения в Instruction

```
1  enum Instruction
2  {
3      // ... другие инструкции ...
4      FPRINT      // Новая инструкция: печать float с вершины стека (масшта
           бированного)
5  };
```

2.2.5 Обработка новой инструкции FPRINT

Добавлена обработка новой инструкции FPRINT. **Входные данные**

- `address` (тип `int`) - адрес команды в памяти
- `os` (тип `ostream&`) - выходной поток для записи
- Внутренние поля класса:
  - `instruction_` (тип `enum`) - код инструкции (например, `FPRINT`)

Выходные данные

- Запись в выходной поток `os` в формате:

```
<address>:    <инструкция>
```

- Пример вывода:

```
42:    FPRINT
```

Изменения функции print представлены в листинге 14.

Листинг 14: Изменения функции print

```
1  void Command::print(int address, ostream & os)
2  {
3      os << address << ":\t";
4      switch (instruction_) {
5          // ... другие case ...
6          case FPRINT:
7              os << "FPRINT";
8              break;
9      }
10     os << endl;
11 }
```

2.2.6 Функции для работы с нецелыми числами (parser.cpp)

Модифицированы функции для работы с нецелыми числами:

Метод `Parser::statement()`

Назначение Обработка оператора вывода (`WRITE`) и генерация соответствующего кода.

Входные данные

- Текущий токен от сканера (через `match(T_WRITE)`)
- Состояние парсера:
  - `error_` - флаг ошибки
  - `recovered_` - флаг восстановления после ошибки
- Результат вызова `expression()` - тип выражения для вывода

Выходные данные

- Генерация команды вывода:
  - `FPRINT` - для дробных чисел
  - `PRINT` - для целых чисел

Метод `Parser::factor()`

Назначение Обработка терминальных элементов выражений (чисел) и генерация соответствующего кода.

Входные данные

- Текущий токен от сканера (через метод `see(T_NUMBER)`)
- Доступ к сканеру через `scanner_`:
  - `getIntValue()` - получение числового значения
  - `isLiteralFloat()` - проверка типа числа
- Доступ к генератору кода через `codegen_`

Выходные данные

- Генерация команды `PUSH` с числовым значением
- Возвращаемое значение типа `VarType`:
  - `VT_FLOAT` - если число дробное
  - `VT_INT` - если число целое

В листинге 15 представлены изменения функций factor и statement.

Листинг 15: Изменения в factor и statement

```
Parser::VarType Parser::factor()
{
    if (see(T_NUMBER)) {
        int value = scanner_->getIntValue();
        bool isLitFloat = scanner_->isLiteralFloat();
        next();
        codegen_->emit(PUSH, value);
        return isLitFloat ? VT_FLOAT : VT_INT;
    }
    // ... остальной код ...
}

void Parser::statement()
{
    // ... код для обработки WRITE ...
    if (match(T_WRITE)) {
        mustBe(T_LPAREN);
        if (error_ && !recovered_) return;

        VarType exprType = expression();
        if (exprType == VT_ERROR || !recovered_) {
            recover(T_RPAREN);
            if (!recovered_) return;
        }

        mustBe(T_RPAREN);
        if (error_ && !recovered_) return;

        if (recovered_) {
            if (exprType == VT_FLOAT) {
                codegen_->emit(FPRINT);
            }
            else {
```

```
                codegen_->emit(PRINT);
            }
        }
    }
    // ... остальной код ...
}
```

2.2.7 Добавление явного приведения типов

В функцию `factor()` добавлена обработка явного приведения типов (листинг 16):

Листинг 16: Обработка явного приведения типов в factor

```
VarType Parser::factor()
{
    // ... предыдущий код ...
    else if (match(T_LPAREN)) {
        if (see(T_INT) || see(T_FLOAT)) {
            // Явное приведение типа (type casting)
            VarType targetType = VT_UNKNOWN;
            if (match(T_INT)) {
                targetType = VT_INT;
            }
            else {
                match(T_FLOAT);
                targetType = VT_FLOAT;
            }

            mustBe(T_RPAREN);
            if (!recovered_) {
                return VT_ERROR;
            }

            VarType sourceType = factor();

            if (sourceType == VT_ERROR) {
                return VT_ERROR;
            }
            if (sourceType == VT_UNKNOWN) {
                reportError("Cannot perform explicit cast from an
                    expression of unknown type.");
                recovered_ = false;
                return VT_ERROR;
            }

            // Генерация кода для приведения типов
            if (targetType == VT_INT && sourceType == VT_FLOAT) {
                codegen_->emit(PUSH, SCALE_FACTOR);
                codegen_->emit(DIV);
            }
            else if (targetType == VT_FLOAT && sourceType == VT_INT) {
                codegen_->emit(PUSH, SCALE_FACTOR);
                codegen_->emit(MULT);
            }
            else if (targetType != sourceType && targetType != VT_UNKNOWN
                && sourceType != VT_UNKNOWN) {
                reportError("Unsupported explicit cast from " +
                    varTypeToString(sourceType) +
                     " to " + varTypeToString(targetType) + ".");
                recovered_ = false;
                return VT_ERROR;
```

```
46                  }
47                  return targetType;
48              }
49              else {
50                  // Обычное выражение в скобках
51                  VarType exprType = expression();
52                  if (exprType == VT_ERROR) {
53                      return VT_ERROR;
54                  }
55                  mustBe(T_RPAREN);
56                  if (!recovered_) {
57                      return VT_ERROR;
58                  }
59                  return exprType;
60              }
61          }
62          // ... остальной код ...
63      }
```

2.2.8 Добавление неявного приведения типов

Реализовано в функциях `expression()` и `term()`.

Функция `Parser::expression()`

Назначение Обработка арифметических выражений с поддержкой операций сложения/-вычитания и неявного приведения типов.

Входные данные

- Текущее состояние парсера:
  - Токен от сканера (метод `see(T_ADDOP)`)
  - Значение арифметической операции через `scanner_->getArithmeticValue()`
- Результат вызова `term()` для левого операнда
- Глобальные константы:
  - `SCALE_FACTOR` - масштабирующий коэффициент

Выходные данные

- Возвращаемое значение типа `VarType`:
  - `VT_INT` - если оба операнда целочисленные
  - `VT_FLOAT` - если хотя бы один операнд дробный
  - `VT_ERROR` - при ошибке разбора
- Сгенерированный код операций:
  - `ADD`/`SUB` - для арифметических операций
  - `MULT` - для приведения типов
  - `STORE/LOAD` - для временного хранения значений

Функция `Parser::term()`

Назначение Обработка термов (умножений/делений) в выражениях.

Входные данные

- Текущее состояние парсера:
  - Токен от сканера (ожидается `T_MULTOP`)
  - Значение арифметической операции
- Результат вызова `factor()` для левого операнда

Выходные данные

- Возвращаемое значение типа `VarType` (аналогично `expression()`)
- Сгенерированный код операций:
  - `MULT/DIV` - для арифметических операций

В листинге 17 представлены изменения функции expression.

Листинг 17: Изменения функции expression

```
Parser::VarType Parser::expression()
{
    VarType type1 = term();
    if (type1 == VT_ERROR) return VT_ERROR;

    while (see(T_ADDOP)) {
        Arithmetic op = scanner_->getArithmeticValue();
        next();

        VarType type2 = term();
        if (type2 == VT_ERROR) return VT_ERROR;

        VarType resultType = VT_INT;

        // Неявное приведение типов для операций сложения/вычитания
        if (type1 == VT_FLOAT || type2 == VT_FLOAT) {
            resultType = VT_FLOAT;

            if (type1 == VT_INT && type2 == VT_FLOAT) {
                int tempAddr = lastVar_;
                codegen_->emit(STORE, tempAddr);
                codegen_->emit(PUSH, SCALE_FACTOR);
                codegen_->emit(MULT);
                codegen_->emit(LOAD, tempAddr);
                type1 = VT_FLOAT;
            }
            else if (type1 == VT_FLOAT && type2 == VT_INT) {
                codegen_->emit(PUSH, SCALE_FACTOR);
                codegen_->emit(MULT);
            }
        }

        if (op == A_PLUS) codegen_->emit(ADD);
        else codegen_->emit(SUB);

        type1 = resultType;
    }
    return type1;
}
```

Аналогичные изменения для операции умножения/деления (листинг 18):

Листинг 18: Изменения функции term

```
Parser::VarType Parser::term()
{
    VarType type1 = factor();
    if (type1 == VT_ERROR) return VT_ERROR;

    while (see(T_MULOP)) {
        Arithmetic op = scanner_->getArithmeticValue();
        next();

        VarType type2 = factor();
        if (type2 == VT_ERROR) return VT_ERROR;

        VarType resultType = VT_INT;

        // Неявное приведение типов для операций умножения/деления
        if (type1 == VT_FLOAT || type2 == VT_FLOAT) {
            resultType = VT_FLOAT;

            if (type1 == VT_INT && type2 == VT_FLOAT) {
                int tempAddr = lastVar_;
                codegen_->emit(STORE, tempAddr);
                codegen_->emit(PUSH, SCALE_FACTOR);
                codegen_->emit(MULT);
                codegen_->emit(LOAD, tempAddr);
                type1 = VT_FLOAT;
            }
            else if (type1 == VT_FLOAT && type2 == VT_INT) {
                codegen_->emit(PUSH, SCALE_FACTOR);
                codegen_->emit(MULT);
            }
        }

        if (op == A_MULTIPLY) {
            codegen_->emit(MULT);
            if (resultType == VT_FLOAT) {
                codegen_->emit(PUSH, SCALE_FACTOR);
                codegen_->emit(DIV);
            }
        }
        else { // A_DIVIDE
            if (resultType == VT_FLOAT) {
                int tempAddr = lastVar_;
                codegen_->emit(STORE, tempAddr);
                codegen_->emit(PUSH, SCALE_FACTOR);
                codegen_->emit(MULT);
                codegen_->emit(LOAD, tempAddr);
            }
            codegen_->emit(DIV);
        }
        type1 = resultType;
    }
    return type1;
}
```

2.2.9 Приведение типов при присваивании

Добавлено в функцию `statement()`(листинг 19):

Листинг 19: Обработка приведения типов в функции statement

```
void Parser::statement()
{
    // ... код для обработки присваивания ...
    else if (see(T_IDENTIFIER)) {
        string varName = scanner_->getStringValue();
        next();
        VarInfo varInfo = getVariable(varName);
        if (varInfo.type == VT_ERROR || !recovered_) {
            recovered_ = false;
            return;
        }

        mustBe(T_ASSIGN);
        if (error_ && !recovered_) return;

        VarType exprType = expression();
        if (exprType == VT_ERROR || !recovered_) return;

        // Приведение типов при присваивании
        bool typesOk = true;
        if (varInfo.type == VT_FLOAT && exprType == VT_INT) {
            codegen_->emit(PUSH, SCALE_FACTOR);
            codegen_->emit(MULT);
        }
        else if (varInfo.type == VT_INT && exprType == VT_FLOAT) {
            codegen_->emit(PUSH, SCALE_FACTOR);
            codegen_->emit(DIV);
        }
        else if (varInfo.type != exprType && exprType != VT_UNKNOWN &&
           varInfo.type != VT_UNKNOWN) {
            reportError("Type mismatch in assignment to '" + varName +
               "'. Variable: " +
                    varTypeToString(varInfo.type) + ", Expression: " +
                       varTypeToString(exprType));
            recovered_ = false;
            typesOk = false;
        }

        if (recovered_ && typesOk) {
            codegen_->emit(STORE, varInfo.address);
        }
    }
    // ... остальной код ...
}
```

3 Результаты работы программы

Далее представлены результаты работы программы. В программах 1-4,7 представлены результаты обработки программ, не содержащих ошибок. В программах 5-6 представлены варианты обработки программ, содержащих ошибки.

Программа №1

Текст программы	Код, сгенерированный компилятором	Вывод программы

BEGIN
  int i:=4;
  float t:=5.5;
  WRITE(i);
  WRITE(t);
END
	0: PUSH 4
	1: STORE 0
	2: PUSH 5500
	3: STORE 1
	4: LOAD 0
	5: PRINT
	6: LOAD 1
	7: FPRINT
	8: STOP
		4
		5.500

Программа №2

Текст программы

```
BEGIN
    int i:=4;
    float t:=5.5;
    WRITE(i+t);
    WRITE(t-i);
    WRITE(i*t);
    WRITE(t/i);
END
```

Код, сгенерированный компилятором

```
0: PUSH 4
1: STORE 0
2: PUSH 5500
3: STORE 1
4: LOAD 0
5: LOAD 1
6: STORE 2
7: PUSH 1000
8: MULT
9: LOAD 2
10: ADD
11: FPRINT
12: LOAD 1
13: LOAD 0
14: PUSH 1000
15: MULT
16: SUB
17: FPRINT
18: LOAD 0
19: LOAD 1
20: STORE 2
21: PUSH 1000
22: MULT
23: LOAD 2
24: MULT
25: PUSH 1000
26: DIV
27: FPRINT
28: LOAD 1
29: LOAD 0
30: PUSH 1000
31: MULT
32: STORE 2
33: PUSH 1000
34: MULT
35: LOAD 2
36: DIV
37: FPRINT
38: STOP
```

Вывод программы

```
9.500
1.500
22.000
1.375
```

Программа №3

Текст программы

```
BEGIN
    int i:= 4.8;
    int j:= (int) 6.8;
    float t:=5.5;
    float p:= (float) i;
    float s:= j;
    int y := t;
    WRITE(i);
    WRITE(j);
    WRITE(t);
    WRITE(p);
    WRITE(s);
    WRITE(y);
END
```

Код, сгенерированный компилятором

```
0: PUSH 4800
1: PUSH 1000
2: DIV
3: STORE 0
4: PUSH 6800
5: PUSH 1000
6: DIV
7: STORE 1
8: PUSH 5500
9: STORE 2
10: LOAD 0
11: PUSH 1000
12: MULT
13: STORE 3
14: LOAD 1
15: PUSH 1000
16: MULT
17: STORE 4
18: LOAD 2
19: PUSH 1000
20: DIV
21: STORE 5
22: LOAD 0
23: PRINT
24: LOAD 1
25: PRINT
26: LOAD 2
27: FPRINT
28: LOAD 3
29: FPRINT
30: LOAD 4
31: FPRINT
32: LOAD 5
33: PRINT
34: STOP
```

Вывод программы

```
4
6
5.500
4.000
6.000
5
```

Программа №4

Текст программы	Код, сгенерированный компилятором	Вывод программы

`BEGIN`
`  float pi := 3.141,`
`  r := 2.0, area;`
`  area := pi * r * r;`
`  WRITE(area);`
`END`
	`0: PUSH 3141`
	`1: STORE 0`
	`2: PUSH 2000`
	`3: STORE 1`
	`4: LOAD 0`
	`5: LOAD 1`
	`6: MULT`
	`7: PUSH 1000`
	`8: DIV`
	`9: LOAD 1`
	`10: MULT`
	`11: PUSH 1000`
	`12: DIV`
	`13: STORE 2`
	`14: LOAD 2`
	`15: FPRINT`
	`16: STOP`
		`12.564`

Программа №5

Текст программы	Код, сгенерированный компилятором	Вывод программы

`BEGIN`
`  float pi := 3.141,`
`  r := 2.0, area`
`  area := pi * r * r;`
`  WRITE(area);`
`END`
	`Line 3: identifier found while 'END' expected.`	-

Программа №6

Текст программы	Код, сгенерированный компилятором	Вывод программы

`BEGIN`
`  pi := 3.141,`
`  r := 2.0, area;`
`  area := pi * r * r;`
`  WRITE(area);`
`END`
	`Line 2: Undeclared variable 'pi'.`
	
	`Line 2: ':=' found while 'END' expected.`
		-

Программа №7

Текст программы

```
BEGIN
int i:= 4.8;
int j:=(float)(int)6.8;
float t:=5.5;
float p:= (float) i;
float s:= j;
int y := t;
WRITE(i);
WRITE(j);
WRITE(t);
WRITE(p);
WRITE(s);
WRITE(y);
END
```

Код, сгенерированный компилятором

```
0:      PUSH    4800
21:     STORE   4
22:     LOAD    2
23:     PUSH    1000
24:     DIV
25:     STORE   5
26:     LOAD    0
27:     PRINT
28:     LOAD    1
29:     PRINT
30:     LOAD    2
31:     FPRINT
32:     LOAD    3
33:     FPRINT
34:     LOAD    4
35:     FPRINT
36:     LOAD    5
37:     PRINT
38:     STOP
```

Вывод программы

```
4
6
5.500
4.000
6.000
5
```

Заключение

В ходе выполнения лабораторной работы была успешно реализована доработка компилятора языка `Milan`, включающая поддержку целого (`int`) и вещественного (`float`) типов данных, а также механизмы явного и неявного приведения типов. Реализация базировалась на следующих ключевых концепциях:

- **LL(1)-грамматики** - использованный подход рекурсивного спуска требует грамматики, допускающей предпросмотр одного символа.
- **Контекстно-свободные грамматики** - расширенная грамматика языка сохранила свойство КС-грамматики.
- Реализована система правил для явных и неявных преобразований между типами.

Достоинства реализации

- Сохранение свойства LL(1) после добавления новых правил.
- Автоматическое приведение с контролем по правилам арифметических преобразований.
- Явные приведения через каст-операторы `(int)` и `(float)` .
- Новая команда `FPRINT` вписана в систему команд VM, VM изменена минимально.
- Масштабирование значений через `SCALE_FACTOR` сохранило целочисленную природу VM.

Выявленные ограничения

:

- Потеря точности при неявных преобразованиях.
- Потеря точности за счет того, что SCALE_FACTOR = 1000. Соответственно, нецелые числа хранятся с точностью до тысячных.

Масштабирование:

- Добавление новых типов данных (double, byte).
- Улучшение точности вычислений.
- Оптимизация преобразований типов на уровне промежуточного кода.

Список литературы

[1] Электронный ресурс ВШТИИ
URL:https://tema.spbstu.ru/compiler/
(Дата обращения: 13.04.2025).

[2] Карпов, Ю. Г. Теория автоматов, Санкт-Петербург : Питер, 2003.
URL:https://djvu.online/file/eeLVKnyRZPXfl
(Дата обращения: 17.04.2025).