Национальный исследовательский университет ИТМО
Мегафакультет компьютерных технологий и управления
Факультет программной инженерии и компьютерной техники

Отчет по практическому заданию №3

по курсу «Языки программирования»

Вариант: 2

Выполнил студент группы P4119 _____ Д.Л. Симоновский
(подпись)

Руководитель _____ Ю. Д. Кореньков
(подпись)

15 октября 2025 г.

Санкт-Петербург
2025

Оглавление

1. Цель работы

Реализовать формирование линейного кода в терминах некоторого набора инструкций посредством анализа графа потока управления для набора подпрограмм. Полученный линейный код вывести в мнемонической форме в выходной текстовый файл.

Вариант работы по 3 заданию представлен ниже:

Таблица 1.1. Исходные данные для 3 задания.

Вариант	Типизация	Формат инструкций ВМ	Банки памяти
22	Статическая	Стековый	Code and constants, data

2. План работы

1. Составить описание виртуальной машины с набором инструкций и моделью памяти по варианту.
2. Реализовать модуль, формирующий образ программы в линейном коде для данного набора подпрограмм.
3. Доработать тестовую программу, разработанную в задании 2 для демонстрации работоспособности созданного модуля.

3. Ход работы

3.1. Описание виртуальной машины

Первым пунктом задания необходимо выполнить разработку описания виртуальной машины.

Первым делом необходимо определить регистры для проекта:

Листинг 3.1. Описание регистров виртуальной машины.

```
registers:

    storage ip [32];   // instruction pointer (byte address)
    storage sp [32];   // stack pointer
    storage bp [32];   // base/frame pointer for stack frames
    storage hp [32];   // heap base pointer

    // volatile temporaries used by instructions; not preserved across ops
    storage t0 [32];
    storage t1 [32];
```

```
    storage t2 [32];

    storage rin [8];   // input port
    storage rout [8];  // output port
```

Первые четыре регистра – служебные, пользователь может на них влиять лишь косвенно, через специальные команды:

1. ip – регистр, который хранит адрес текущей выполняемой команды.
2. sp – поскольку архитектура стековая, этот регистр хранит адрес вершины стека.
3. bp – регистр, который хранит указатель, использующийся для обращения к локальным переменным, а также передаваемым значениям в функцию.
4. hp – регистр, в котором находится указатель на вершину кучи.

Далее идут регистры, используемые для временного хранения вычислений в рамках одной инструкции, пользователь не имеет к ним доступа.

Последние два регистра нужны для системы ввода-вывода.

Далее задаются банки памяти, по заданию их два:

Листинг 3.2. Память виртуальной машины.

```
memory:

    // code + embedded constants (logical 64K bank)
    range code [0x0000 .. 0xffff] {
        cell = 8;
        endianess = little-endian;
        granularity = 2;
    }

    // data bank; stack lives here and grows down from top of the bank
    range dmem [0x0000 .. 0xffff] {
        cell = 8;
        endianess = little-endian;
        granularity = 2;
    }
```

Стоит обратить внимание, что регистры 32 битные, а память адресуется 16 битами.

Также важно отметить, что мнемоники в отчете приводится не будут, они полностью дублируют названия инструкций, подробнее можно ознакомиться в исходных файлах.

Для начала опишем поля инструкций, их два вида – 32 битные для данных и 16 битные для адресов:

Листинг 3.3. Объявление полей.

```
instructions:

    encode imm16 field = immediate [16];
    encode imm32 field = immediate [32];
```

Первым делом рассмотрим инструкции переходов, условные и безусловные:

Листинг 3.4. Инструкции переходов.

```
    // unconditional jump (absolute)
    instruction jmp = { 0001 0000, imm16 as target } {
        ip = target;
    };

    // jump if top-of-stack == 0; pops 1 word
    instruction jz = { 0001 0001, imm16 as target } {
        t0 = dmem:4[sp];   // use temp
        sp = sp + 4;
        if t0 != 0 then
            ip = ip + 3;   // non-zero -> no jump
        else
            ip = target;   // zero -> jump
    };

    // jump if top-of-stack != 0; pops 1 word
    instruction jnz = { 0001 0010, imm16 as target } {
        t0 = dmem:4[sp];   // use temp
        sp = sp + 4;
        if t0 != 0 then
            ip = target;   // non-zero -> jump
        else
            ip = ip + 3;   // zero -> skip jump
    };
```

Эти инструкции позволяют переходить к переданному адресу, есть возможность условного перехода, чтоб он совершался, если на вершине стека ноль или не ноль, в зависимости от инструкции. Значение с вершины стека удаляется.

Также создадим инструкцию для окончания выполнения программы:

Листинг 3.5. Инструкция для окончания выполнения программы.

```
    // stop execution
    instruction hlt = { 0001 0101 } {
    };
```

Следующие две команды – служебные и не генерируются от кода пользователя, они необходимы для стартовой инициализации стека и кучи:

Листинг 3.6. Инициализация стека и кучи.

```
// set stack pointer to immediate (use to initialize stack)
instruction ldsp = { 0001 0110, imm16 as value } {
    sp = value;
    ip = ip + 3;
};

// set heap base pointer to immediate (use to initialize heap head)
instruction ldhp = { 0010 1000, imm16 as value } {
    hp = value;
    ip = ip + 3;
};
```

Несомненно, самыми важными инструкциями для стековой архитектуры являются операции над стеком:

Листинг 3.7. Инструкции для работы со стеком.

```
// push immediate 32-bit value
instruction push = { 0001 1000, imm32 as value } {
    sp = sp - 4;
    dmem:4[sp] = value;
    ip = ip + 5;
};

// drop top of stack
instruction drop = { 0001 1001 } {
    sp = sp + 4;
    ip = ip + 1;
};

// load 8-bit: pop address, push zero-extended byte at [address]
instruction load1 = { 0011 0000 } {…};

// load 16-bit: pop address, push zero-extended word at [address]
instruction load2 = { 0011 0001 } {};

// load 32-bit: pop address, push dword at [address]
instruction load = { 0011 0010 } {
    t0 = dmem:4[sp];              // addr
    dmem:4[sp] = dmem:4[t0];      // overwrite with loaded dword
    ip = ip + 1;
};

// store 8-bit: pop value and address
instruction store1 = { 0011 0011 } {…};

// store 16-bit: pop value and address (
instruction store2 = { 0011 0100 } {…};

// store 32-bit: pop value and address
instruction store = { 0011 0101 } {
    t0 = dmem:4[sp + 4];          // addr
    t1 = dmem:4[sp];              // value
    dmem:4[t0] = t1;
```

```
            sp = sp + 8;
            ip = ip + 1;
        };
```

Здесь представлены операции добавления на стек константы, также добавление значения разной размерности (1, 2, 4 байта – для работы с массивами (для 1 и 2 байт команды были опущены в целях удобства, они аналогичны 4 байтной версии за исключением размерности запрашиваемых данных)), удаление значения со стека, сохранение значения со стека.

Стоит отметить, что стек растет вниз.

Также были реализованы инструкции для вывода и ввода:

Листинг 3.8. I/O инструкции.

```
        // read byte from input port, push zero-extended to 32-bit
        instruction inb = { 0001 1100 } {
            sp = sp - 4;
            dmem:4[sp] = rin;
            ip = ip + 1;
        };

        // pop 32-bit, write low byte to output port
        instruction outb = { 0001 1101 } {
            rout = dmem:4[sp];
            sp = sp + 4;
            ip = ip + 1;
        };
```

Еще были реализованы инструкции сравнение, бинарные и унарные инструкции:

Листинг 3.9. Унарные и бинарные инструкции.

```
        // add: pop b, pop a, push (a + b)
        instruction add = { 0001 1110 } {...};

        // shift left: pop count, pop value, push (value << (count & 31)) or
(value >> (-count & 31)) if count < 0
        instruction shl = { 0001 1111 } {...};

        // shift right logical: pop count, pop value, push (value >> (count &
31)) or (value << (-count & 31)) if count < 0
        instruction shr = { 0010 0000 } {...};

        // subtract: pop b, pop a, push (a - b)
        instruction sub = { 0100 0000 } {...};

        // multiply: pop b, pop a, push (a * b)
        instruction mul = { 0100 0001 } {...};

        // divide: pop b, pop a, push (a / b) (no div-by-zero handling)
        instruction div = { 0100 0010 } {...};

        // modulo: pop b, pop a, push (a % b)
```

```
instruction mod = { 0100 0011 } {...};

// bitwise or: pop b, pop a, push (a | b)
instruction bor = { 0100 0100 } {...};

// bitwise and: pop b, pop a, push (a & b)
instruction band = { 0100 0101 } {...};

// bitwise xor: pop b, pop a, push (a ^ b)
instruction bxor = { 0100 0110 } {...};

// logical or: pop b, pop a, push (a!=0 || b!=0 ? 1 : 0)
instruction lor = { 0100 0111 } {...};

// logical and: pop b, pop a, push (a!=0 && b!=0 ? 1 : 0)
instruction land = { 0100 1000 } {...};

// logical not (unary): push (!a) where result is 1/0
instruction not_u = { 0100 1001 } {...};
// bitwise not (unary)
instruction bnot_u = { 0100 1010 } {...};

// equal: pop b, pop a, push (a == b ? 1 : 0)
instruction eq = { 0100 1011 } {...};

// not equal: pop b, pop a, push (a != b ? 1 : 0)
instruction ne = { 0100 1100 } {...};

// greater than: pop b, pop a, push (a > b ? 1 : 0)
instruction gt = { 0100 1101 } {...};

// less than: pop b, pop a, push (a < b ? 1 : 0)
instruction lt = { 0100 1110 } {...};

// greater than or equal: pop b, pop a, push (a >= b ? 1 : 0)
instruction ge = { 0100 1111 } {...};

// less than or equal: pop b, pop a, push (a <= b ? 1 : 0)
instruction le = { 0101 0000 } {...};
```

Их описание опущено так как достаточно интуитивно, но занимает много места. Для подробного ознакомления можно обратиться к исходным файлам.

Далее рассмотрим операции работы с кучей:

Листинг 3.10. Инструкции для кучи.

```
// hp = hp + top_of_stack; pops value
instruction addhp = { 0010 1110 } {
    hp = hp + dmem:4[sp];
    sp = sp + 4;
    ip = ip + 1;
};

// push current hp
instruction pushhp = { 0010 1010 } {
    sp = sp - 4;
    dmem:4[sp] = hp;
    ip = ip + 1;
};
```

Стоит отметить, что куча растет вверх, а также никто её не очищает, поэтому в теории она может быстро заполниться и залезать на стек. Данный процесс не рассматривается в рамках лабораторной работы.

Последнее, что необходимо реализовать – инструкции для работы с переменными, текущим состоянием функции, вызовами функций и всем что с этим связано.

Было принято решение придерживаться следующих договоренностей:

1. Перед вызовом функции необходимо на стек положить все передающиеся значения в порядке передачи, то есть первый аргумент функции необходимо первым положить на стек.
2. Вызов функции кладет на стек адрес возврата после чего добавляет адрес bp вызывающей функции и устанавливает этот адрес новым адресом bp, который будет использовать вызываемая функция.
3. Далее вызываемая функция кладет на стек все локальные переменные, таким образом инициализируя их нулями.
4. При выходе из функции вызываемая кладет на место последнего своего аргумента возвращаемо значение (если оно есть), если аргументов у функции нет, то вызывающий должен был положить на стек 0.
5. Далее восстанавливается bp, а стек указывает на последний передаваемый аргумент

Однако выяснилось, что такие договоренности отлично работают при условии наличия отдельного слова под return, однако в языке оно отсутствует, из-за этого появилась необходимость несколько модифицировать алгоритм:

1. Как и в прошлом алгоритме необходимо положить на стек все аргументы в порядке передачи.
2. Аналогично прошлому алгоритму на стек кладется адрес возврата, потом bp.
3. Далее также вызываемая функция создает локальные переменные, путем добавления их на стек, но главное отличие в том, что, если

функция имеет возвращаемое значение, она создает дополнительную переменную с именем функции.

4. Пользователь в процессе работы может изменять переменную, с именем функции, если она есть.
5. При выходе из функции вызываемая функция кладет на место последнего своего аргумента возвращаемо значение (если оно есть) из локальной переменной, если аргументов у функции нет, то вызывающий должен положить на стек 0, если функция должна что-то вернуть.
6. Далее восстанавливается bp, а стек указывает на последний передаваемый аргумент

Путем таких договоренностей к передаваемым и локальным переменным всегда есть доступ в функции. Реализация вызова и возврата приведена ниже:

Листинг 3.11. Инструкции вызова и возврата из функции.

```
        // call absolute; pushes return address (next instruction) on stack
        instruction call = { 0001 0011, imm16 as target } {
            // caller already pushed args;
            // prologue saves return + bp, then sets new bp
            sp = sp - 4;
            dmem:4[sp] = ip + 3;  // return address
            sp = sp - 4;
            dmem:4[sp] = bp;      // save caller bp
            bp = sp;              // establish frame pointer (bp -> saved bp)
            ip = target;
        };

        // return; pops ip from stack
        instruction ret = { 0001 0100 } {
            // frame layout at bp:
            // [bp+0] saved bp, [bp+4] return, [bp+8...] caller args
            t0 = dmem:4[bp];        // old bp
            ip = dmem:4[bp + 4];    // return address
            // convention: callee stores return value at [bp+8] before ret
            sp = bp + 8;            // drop frame, leave args
            bp = t0;
        };
```

Таким образом целевая виртуальная машина была описана.

3.2. Реализация транслятора

Из прошлого задания main функция выглядит следующим образом:

Листинг 3.12. Main функция из прошлой практической работы.

```
def main():
```

```
    grammar_dir, lang_name, file_paths, out_dir, lib_path = parse_cli()
    result = analyze_files(file_paths, lib_path, lang_name, grammar_dir,
             out_dir=out_dir)

    out_dir_path = Path(out_dir)
    out_dir_path.mkdir(parents=True, exist_ok=True)
    call_graph_base = out_dir_path / "call_graph"
    render_call_graph(result, filename=str(call_graph_base), fmt="svg")

    errors_report_path = call_graph_base.with_suffix(".errors.txt")
    ready_assemble = write_errors_report(result,
                     filename=str(errors_report_path))

    if ready_assemble:
        return
```

Здесь идет получение аргументов, далее формируется граф потока правления и отрисовывается. Если есть ошибки, то они записываются в файл и процесс трансляции не начинается.

Основная же часть данной практической идет далее и состоит из двух основных этапов:

Листинг 3.13. Продолжение main.

```
    # Проверка типов: если есть ошибки, останавливаем трансляцию
    has_errors, typed_data, funcs_returns = handle_type_check(result,
                                                              out_dir_path)
    if has_errors:
        return

    # Проверка наличия main без аргументов и без возвращаемого значения
    if not check_main_function(result, errors_report_path):
        return

    # Проверяем, что typed_data не None перед использованием
    if typed_data is None:
        print("Ошибка: typed_data равен None")
        return

    asm_file = out_dir_path / "result.asm"

    # Передаем typed_data (с обновленными vars) и funcs_returns в generate_asm
    generate_asm(typed_data, str(asm_file), funcs_returns)
    print(f"Ассемблерный код сохранен в {asm_file}")

    return
```

Сначала происходит определение типов данных, затем непосредственно генерация ассемблер кода. По факту в handle_type_check происходит обход всех деревьев операций для добавления нового поля каждому блоку – их типа.

В рамках данной практической работы был выбран подход без неявного преобразования типов друг к другу.

Перейдем к рассмотрению самой функции:

Листинг 3.14. Функция для распределения типов.

```
def handle_type_check(result, out_dir_path: Path) -> tuple[bool, dict, dict]:
    # возвращает typed_data - та же структура, но с типами в node.type
    # и funcs_returns - информацию о возвращаемых типах функций
    typed_data, errors, funcs_returns = process_type(result)

    if errors:
        write_type_errors(errors, out_dir_path)
        return True, None, None  # Ошибки найдены, останавливаем трансляцию

    if typed_data:
        render_typed_graphs(typed_data, out_dir_path)

    return False, typed_data, funcs_returns
```

Здесь вызывается очередная функция для расставления типов, после чего проверяется наличие ошибок типизации, а потом отрисовывается граф потока управления с заданными типами, это нужно для отладки.

typed_data это словарь графов потока управления с размеченными типами, где ключи – названия функций.

funcs_returns это словарь возвращаемых функциями значений, где ключи – названия функций.

Дальнейшее описание будет без приведения кода так как он достаточно объемный, а лишь с описанием того, что делают функции. Для более детального ознакомления можно обратиться к исходным файлам.

process_type – основная функция процесса. Сначала собирает информацию о типах: для каждой функции извлекает возвращаемые типы через *get_func_returns_type*, типы аргументов через *get_args_dict* и типы локальных переменных через *get_dict_var*. Затем проверяет конфликты имен со встроенными функциями и конфликты между параметрами и локальными переменными. Добавляет в словари встроенные функции (*read_byte*, *send_byte*, функции конвертации типов, конструкторы массивов). После этого вызывает *check_all_functions* для проверки типов во всех функциях.

check_all_functions создает экземпляр *TypeChecker* с собранными типами функций, переменных и аргументов, затем для каждой функции вызывает *check_types_in_cfg*.

check_types_in_cfg устанавливает контекст текущей функции через *set_context* и для каждого блока CFG вызывает *assign_types* на корневом узле дерева.

assign_types рекурсивно выводит и присваивает типы узлам. Сначала вызывает *infer_type* для вывода типа узла, затем записывает результат в *node.type*. После этого пропагирует типы к дочерним узлам и рекурсивно обрабатывает всех потомков.

infer_type определяет тип узла по его структуре: для констант извлекает тип литерала, для переменных (*load*) получает тип из контекста функции, для вызовов функций (*call*) проверяет соответствие аргументов и возвращает тип функции, для присваиваний (*store*) проверяет совместимость типов, для бинарных операций использует специализированные методы проверки (*_check_binary_arithmetic, _check_binary_bitwise, _check_binary_logical, _check_comparison*), для унарных операций — соответствующие методы проверки, для индексации проверяет, что база — массив, а индекс — целое число.

Таким образом на выходе мы получаем граф потока управления с размеченными типами для каждого элемента дерева в нем.

Далее рассмотрим непосредственно алгоритм для преобразования графа потока управления в код под разработанную ранее виртуальную машину.

Листинг 3.15. Код функции для трансляции графа потока управления.

```
def generate_asm(typed_blocks, out_file, funcs_returns=None):
    # Записываем подготовительные инструкции
    generate_preparation(out_file)

    # Записываем встроенные функции
    generate_builtin_func(out_file)

    # Обрабатываем пользовательские функции
    for f_name, (_, _, cfg, tree, params, vars) in typed_blocks.items():
        # Пропускаем псевдо-узлы файлов
        if f_name.startswith('<file:'):
            continue
        # Пропускаем функции без CFG
        if cfg is None:
            continue
        process_func(f_name, cfg, tree, params, out_file, vars, funcs_returns)
```

Все начинается с записи в файл подготовительных инструкций:

Листинг 3.16. Подготовительные инструкции.

```
[section code, code]
ldsp 0xFFFC       ; even-aligned stack top
ldhp 0x0000       ; init heap base
setbp             ; establish caller bp before first call
call main
hlt
```

Они задают вершину стека, начало кучи, а также задают bp на текущую вершину стека. После этого вызывается main, а когда из него будет возврат, программа завершается.

Далее идет запись следующих встроенных функций:

Листинг 3.17. Встроенные функции.

```
read_byte:      ; Builtin function: read_byte()
    inb         ; read in byte
    stbp 8      ; store at bp + 8
    ret

send_byte:      ; Builtin function: send_byte(b: byte)
    ldbp 8      ; load bp + 8
    outb        ; send byte
    ret

alloc:
     ; [bp] saved bp, [bp+4] return, [bp+8] size_shift, [bp+12] len
     ldbp 12       ; push len
     ldbp 8        ; push size_shift
     shl           ; total_bytes = len << size_shift
     pushhp        ; old hp -> will be return value
     stbp 8        ; store return value at [bp+8], pop old hp
     addhp         ; hp = hp + total_bytes, pop total
     ret

bool:                   ; Builtin constructor: bool(size) -> array[] of bool
     ldbp 8             ; load size of array from [bp+8] and push to stack
     push 0             ; push size_shift to stack
     call alloc         ; alloc(size_shift, len) -> result on TOS after ret
     stbp 8             ; store result at [bp+8] and pop
     ret
```

read_byte и *send_byte* используются для чтения и отправки байт, *alloc* служебная функция, она создана для выделения памяти для массивов, она принимает длину, размер одного элемента и возвращает указатель на кучу. После неё идут встроенные функции для инициализации массивов разных типов, для примера приведен массив из bool’ов.

Важно отметить, что здесь не соблюдаются оговоренные ранее правила для вызываемой функции, так как никакой локальной переменной она не задает, а

работает напрямую с последним переданным значением. Причина в том, что это несколько бы усложнило код этих функций, а так как они постоянно вызываются это бы замедлило исполнение. Главное, что соблюдаются правила аргументов и возвращаемого значения.

Также есть ряд встроенных функций для переводов между типами, но они достаточно очевидны, для просмотра конкретной реализации можно обратиться к исходным файлам

Далее в функции для обработки графа потока управления (листинг 4.4) вызывается для каждой функции её парсер. В ней список переменных и список параметров обретает индексы смещения:

Листинг 3.18. Определение смещения для получения переменной.

```
def process_func(f_name, f_cfg, f_tree, f_params, out_file, vars,
                 funcs_returns=None):
    # Преобразуем vars из списка кортежей в словарь с смещениями
    vars_dict = {}
    for index, (var_name, var_type) in enumerate(vars):
        offset = (index + 1) * -4
        vars_dict[var_name] = (var_type, offset)

    # Преобразуем f_params из списка кортежей в словарь с смещениями
    params_dict = {}
    for index, (param_name, param_type) in enumerate(f_params):
        offset = 4 * (len(f_params) - index + 1)
        params_dict[param_name] = (param_type, offset)

    # Проверяем, возвращает ли функция значение
    has_return = False
    if funcs_returns:
        func_type = funcs_returns.get(f_name)
        if func_type and func_type[0] is not None:
            has_return = True
```

Далее идет запись вступления функции, после чего вызывается обработка содержимого и создается её завершение:

Листинг 3.19. Продолжение обработки.

```
    with open(out_file, 'a', encoding='utf-8') as f:
        f.write(f'\n{f_name}:\n')

        for var_name, var_type in vars:
            f.write(f'    push 0    ; {var_name}\n')

        process_cfg(f_name, f_cfg, f_tree, params_dict, f, vars_dict,
                    funcs_returns)
        f.write(f'.out:\n')

        # Если функция возвращает значение
```

```
        if has_return and f_name in vars_dict:
            var_offset = vars_dict[f_name][1]
            f.write(f'    ldbp {var_offset}  ; load return from {f_name}\n')
            f.write(f'    stbp 8  ; store return value at bp + 8\n')

        f.write('    ret\n')
```

process_cfg для каждого узла графа вызывает код для обработки:

Листинг 3.20. Функция process_cfg.

```
def process_cfg(f_name, f_cfg, f_tree, params_dict, out_file, vars_dict,
                funcs_returns=None):
    for _, block in f_cfg.blocks.items():
        process_block(f_name, block, params_dict, out_file, vars_dict,
                      funcs_returns)
```

Сам *process_block* приведен ниже:

Листинг 3.21. Функция обработки узла графа.

```
def process_block(f_name, block, params_dict, f, vars_dict,
funcs_returns=None):
    f.write(f'.id{block.id}:\n')

    process_ast(block.tree, params_dict, f, vars_dict, funcs_returns)

    if len(block.succs) == 0:
        f.write(f'    jmp .out\n')
    elif len(block.succs) == 1:
        succ_id, _ = block.succs[0]
        f.write(f'    jmp .id{succ_id}\n')
    else:
        # Два перехода: True и False
        true_succ = next(succ_id for succ_id, label in block.succs if
                         label.lower() == "true")
        false_succ = next(succ_id for succ_id, label in block.succs if
                          label.lower() == "false")
        f.write(f'    jnz .id{true_succ}\n')
        f.write(f'    jmp .id{false_succ}\n')
```

Здесь создается метка для перехода, вызывается функция для обработки дерева, а после записываются инструкции переходов к следующему узлу.

Сам *process_ast* просто рекурсивно вызывает обработчик для каждого узла вершины дерева, для подробного ознакомления рекомендую обратиться в исходные файлы так как в рамках отчета это не рассмотреть.

В результате выполнения кода мы получаем транслированный код на описанную ранее виртуальную машину.

3.3. Тестирование

Проведем последовательное тестирование разработанной программы. Создадим файл только с main функцией без содержимого:

Листинг 3.22. Код первого теста.

```
method main() begin

end;
```

Полученный ассемблерный код приведен ниже:

Листинг 3.23. Транслированный код первого теста.

```
[section code, code]
ldsp 0xFFFC       ; even-aligned stack top
ldhp 0x0000       ; init heap base
setbp             ; establish caller bp before first call
call main
hlt

...

main:
.id0:
    jmp .id1
.id1:
    jmp .out
.out:
    ret
```

Здесь и далее объявление встроенных функций опущено так как сильно нагружает листинг и не дает полезной информации.

Как можно заметить, код транслировался корректно, а *begin* → *end* был преобразован в корректную последовательность блоков.

Выполним запуск этого теста на виртуальной машине, используем для этого самописный скрипт, с ним можно подробно ознакомиться в исходных файлах, в отчете он приведен не будет. Он выполняет сборку бинарного файла и последующий интерактивный запуск, данные берет из файла и результат пишет в файл:

```
Task ExecuteBinaryWithInput started with id 295f57d9-82b1-4d50-8b45-
5e8581cbfc44
Waiting for completion...
```

Файл успешно собрался и запустился, в выводе ничего нет так как программа ничего и не выводит.

Далее добавим локальную переменную, сохраним в неё значение из потока ввода и выведем его в консоль.

Листинг 3.24. Код второго теста.

```
method main()
var readed_byte : byte;
begin
    readed_byte := read_byte();
    send_byte(readed_byte);
end;
```

Транслируем его:

Листинг 3.25. Транслированный код второго теста.

```
main:
    push 0    ; readed_byte
.id0:
    jmp .id2
.id1:
    jmp .out
.id2:
    push 0  ; for return value
    call read_byte
    stbp -4
    jmp .id3
.id3:
    ldbp -4
    call send_byte
    drop
    jmp .id1
.out:
    ret
```

Здесь видно создание переменной *readed_byte*, далее вызывается функция *read_byte*, поскольку у неё нет аргументов, но есть возвращаемое значение, создается буфер перед вызовом. После вызова указатель указывает на возвращенное значение, мы его сразу сохраняем в переменную *readed_byte*.

После этого байт загружается из памяти, как аргумент для *send_byte*. По завершению вызова выполняется *drop*, чтоб убрать лишнее со стека (передаваемое значение), а далее программа завершается.

Запишем в передаваемый файл цифру 3 и запустим код. Результат приведен ниже:

```
Task ExecuteBinaryWithInput started with id 84500060-c2e7-45fb-a96e-
02e94236eadc
Waiting for completion...
Here is task output interpreted with UTF8:
3
```

Видим, что все выполнилось корректно, цифра 4 была считана из файла и записана в поток вывода.

Усложним задачу, попробуем выполнить комплексную математическую операцию:

Листинг 3.26. Код третьего теста.

```
method main()
var a: uint;
    b: uint;
begin
    b := 200;
    a := (
        (
            ((((5000 + 3000) * 4 - b * 10) / 10) % 97)
            - (b * 5 / 25)
        )
        +
        (
            (((b * 6 / 3) - b * 2) * (7 - 7))
            + ((99 % 9) * (123 + 1))
            + (((500 / 10) - 50) * b)
        ) / 1
    );
    send_byte(int_to_byte(uint_to_int(a)));
end;
```

Результатом выполнения этой операции должно быть 50, то есть цифра 2 в ASCII. Выполним трансляцию, результат гораздо более большой, чем в предыдущих тестах, ознакомиться можно в исходных файлах проекта.

Выполним сборку и получим следующий результат:

```
Task ExecuteBinaryWithInput started with id de812f01-e790-4e9a-bf5c-
4e9d8352e6be
Waiting for completion...
Here is task output interpreted with UTF8:
2
```

Видим, что все работает корректно.

Продолжим тестирование различных стейтментов, на очереди if:

Листинг 3.27. Код четвертого теста.

```
method test_if(a: int) begin
    if a = 3 then
        send_byte(51);
    else
        if a = 4 then
            send_byte(52);
        else
            send_byte(48);
end;

method main()
```

```
begin
    test_if(3);
    test_if(4);
    test_if(2);
    test_if(5);
end;
```

Здесь помимо тестирования if, также используется самописная функция. Она выводит в консоль 3, если получает 3, 4, если получает 4 и 0, если получает что-то другое. Выполним трансляцию, компиляцию и запуск:

```
Task ExecuteBinaryWithInput started with id fdbfb80f-d0a1-4b5e-a254-
945ef517afd9
Waiting for completion...
Here is task output interpreted with UTF8:
3400
```

Все было выполнено корректно, в итоговом выводе как раз и видим переданные в функцию 3, 4, а затем 0 и 0 так как оба if попали в else.

Продолжим тестирование, заменив вывод на возвращение значения, а вывод сделаем уже в main:

Листинг 3.28. Код пятого теста.

```
method test_if(a: int): byte begin
    if a = 3 then
        test_if := 51;
    else
        if a = 4 then
            test_if := 52;
        else
            test_if := 48;
end;

method main()
begin
    send_byte(test_if(3));
    send_byte(test_if(4));
    send_byte(test_if(2));
    send_byte(test_if(5));
end;
```

Выполним запуск:

```
Task ExecuteBinaryWithInput started with id 0d707de3-7571-4e78-97d3-
96cc37b7edc9
Waiting for completion...
Here is task output interpreted with UTF8:
3400
```

Вывод аналогичен, что и ожидалось.

Далее выполним проверку других стейтментов, например while:

Листинг 3.29. Код шестого теста.

```
method test_while(a: int) begin
```

```
    while a != 0 do begin
        send_byte(int_to_byte(a + 48));
        a := a - 1;
    end;
end;

method main()
begin
    test_while(3);
    send_byte(0x0A);
    test_while(4);
    send_byte(0x0A);
    test_while(2);
    send_byte(0x0A);
    test_while(0);
end;
```

Функция *test_while* будет выводить в выходной поток значения от переданного до единицы включительно. Строки *send_byte(0x0A)* посылают в выходной поток перенос строки, чтоб было удобнее читать вывод.

Выполним сборку и запустим:

```
Task ExecuteBinaryWithInput started with id 0cbb9bbb-c319-453a-89da-
4190151150fa
Waiting for completion...
Here is task output interpreted with UTF8:
321
4321
21
```

В результате запуска получили результат, соответствующий ожиданиям.

Далее выполним проверку do while и do until:

Листинг 3.30. Код седьмого теста.

```
method test_while(a: int)
var a_temp: int;
begin
    a_temp:=a;
    repeat begin
        send_byte(int_to_byte(a_temp + 48));
        a_temp := a_temp - 1;
    end; while a_temp > 0;
    send_byte(0x0A);
    repeat begin
        send_byte(int_to_byte(a + 48));
        a := a - 1;
    end; until a < 0;
end;

method main()
begin
    test_while(3);
    send_byte(0x0A);
    test_while(4);
    send_byte(0x0A);
    test_while(2);
```

```
    send_byte(0x0A);
    test_while(0);
end;
```

Здесь сначала выводятся цифры, пока a не достигнет нуля, потом, пока не опустится ниже нуля. Результат приведен ниже:

```
Task ExecuteBinaryWithInput started with id 03cb8b8e-2280-44dc-b82e-
75dcd87c3994
Waiting for completion...
Here is task output interpreted with UTF8:
321
3210
4321
43210
21
210
0
0
```

Как можно заметить, вывод соответствует ожиданиям.

Выполним тестирование этой же функции, но с типом данных *long*, чтоб проверить, что данный алгоритм работает с любым типом:

Листинг 3.31. Код восьмого теста.

```
method test_while(a: long)
var a_temp: long;
begin
    a_temp:=a;
    repeat begin
        send_byte(int_to_byte(long_to_int(a_temp) + 48));
        a_temp := a_temp - 1;
    end; while a_temp > 0;
    send_byte(0x0A);
    repeat begin
        send_byte(int_to_byte(long_to_int(a) + 48));
        a := a - 1;
    end; until a < 0;
end;

method main()
begin
    test_while(3);
    send_byte(0x0A);
    test_while(4);
    send_byte(0x0A);
    test_while(2);
    send_byte(0x0A);
    test_while(0);
end;
```

Тест аналогичен седьмому, но с измененным типом в функции. Результат запуска аналогичен:

```
Task ExecuteBinaryWithInput started with id f8a5021e-c7d4-4a4e-bb47-
bf77204cae8a
Waiting for completion...
Here is task output interpreted with UTF8:
```

```
321
3210
4321
43210
21
210
0
0
```

В девятом тесте проверим работу всех арифметических операций:

Листинг 3.32. Код девятого теста.

```
method helper_value(): int begin
    helper_value := 20;
end;

method all_operations_demo(): int begin
    all_operations_demo := (
        (helper_value() * 2) +
        (helper_value() / 2) +
        ((helper_value() % 10) - (helper_value() % 10)) +
        ((helper_value() << 1) - (helper_value() << 1)) +
        ((helper_value() >> 1) - (helper_value() >> 1)) +
        (((helper_value() | 0) & 15) - ((helper_value() & 15))) +
        (((helper_value() ^ helper_value()) | 1) + (~0 & 1) + 2)
    );
end;

method main()
begin
    send_byte(int_to_byte(all_operations_demo()));
end;
```

Результатом будет значение 54, которое в ASCII кодирует цифру 6, проверим результат путем запуска:

```
Task ExecuteBinaryWithInput started with id f3869150-3beb-4231-8d96-
5f2b0a768bf3
Waiting for completion...
Here is task output interpreted with UTF8:
6
```

Как можно заметить, ожидания совпали с реальностью, что свидетельствует о корректности работы.

Далее проверим логические операции:

Листинг 3.33. Код десятого теста.

```
method test_bool_ops()
var a: bool; b: bool; c: bool; result: bool;
begin
    a := true;
    b := false;
    c := true;
    result := !a;
    send_byte(bool_to_byte(result) + 48);
    result := a && b;
    send_byte(bool_to_byte(result) + 48);
```

```
    result := a || b;
    send_byte(bool_to_byte(result) + 48);
    result := (a && b) || c;
    send_byte(bool_to_byte(result) + 48);
    result := !(a && b);
    send_byte(bool_to_byte(result) + 48);
    result := a || (!b);
    send_byte(bool_to_byte(result) + 48);
end;

method main()
begin
    test_bool_ops();
end;
```

Результатом запуска будет следующий файл:

```
Task ExecuteBinaryWithInput started with id 1bac768e-779c-43cd-bd12-
c0732cad65eb
Waiting for completion...
Here is task output interpreted with UTF8:
001111
```

Именно такой ответ и ожидался от программы.

Далее проверим работу break:

Листинг 3.34. Код одиннадцатого теста.

```
method test_break(n: int)
    var i: int;
begin
    i := 0;
    while i < 10 do begin
        send_byte(int_to_byte(i + 48));
        i := i + 1;
        if i = n then
            break;
    end;
end;

method main()
begin
    test_break(3);
    send_byte(0x0A);
    test_break(5);
    send_byte(0x0A);
    test_break(10);
end;
```

Вывод приведен ниже:

```
Task ExecuteBinaryWithInput started with id d3b5c2e2-50b3-43ec-b4f9-
77d40e10a1dd
Waiting for completion...
Here is task output interpreted with UTF8:
012
01234
0123456789
```

Как и было задано, программа останавливалась на заданном значении.

Следующим тестом будет рекурсия:

Листинг 3.35. Код двенадцатого теста.

```
method factorial(n: int): int begin
    if n <= 1 then
        factorial := 1;
    else
        factorial := n * factorial(n - 1);
end;

method main()
begin
    send_byte(int_to_byte(factorial(1) + 48));
    send_byte(0x0A);
    send_byte(int_to_byte(factorial(2) + 48));
    send_byte(0x0A);
    send_byte(int_to_byte(factorial(3) + 48));
end;
```

Результат вывода приведен ниже:

```
Task ExecuteBinaryWithInput started with id 638959d2-1ce6-429f-aabe-
494016255a1d
Waiting for completion...
Here is task output interpreted with UTF8:
1
2
6
```

Все работает исправно в соответствии с ожиданиями.

В заключение необходимо выполнить тестирование массивов:

Листинг 3.36. Код тринадцатого теста.

```
method test_arrays()
var byte_arr: array[] of byte;
    int_arr: array[] of int;
    long_arr: array[] of long;
    i: int;
begin
    byte_arr := byte(3);
    int_arr := int(3);
    long_arr := long(3);

    byte_arr[0] := 1;
    byte_arr[1] := 2;
    byte_arr[2] := 3;

    int_arr[0] := 100;
    int_arr[1] := 200;
    int_arr[2] := 300;

    long_arr[0] := 1000;
    long_arr[1] := 2000;
    long_arr[2] := 3000;

    i := 0;
    while i < 3 do begin
        send_byte(byte_arr[i] + 48);
        i := i + 1;
    end;
    send_byte(0x0A);
```

```
    i := 0;
    while i < 3 do begin
        send_byte(int_to_byte((int_arr[i] / 100) + 48));
        i := i + 1;
    end;
    send_byte(0x0A);

    i := 0;
    while i < 3 do begin
        send_byte(int_to_byte((long_to_int(long_arr[i]) / 1000) + 48));
        i := i + 1;
    end;
end;

method main()
begin
    test_arrays();
end;
```

Выполним запуск и проверим результаты:

```
Task ExecuteBinaryWithInput started with id 36388fb4-8393-4321-8044-
ded14b1af5bd
Waiting for completion...
Here is task output interpreted with UTF8:
123
123
123
```

Как можно заметить, все тесты прошли успешно, что свидетельствует о правильности выполненной работы.

4. Вывод

В ходе выполнения практического задания №3 была разработана стековая виртуальная машина по варианту и реализован модуль трансляции, формирующий линейный код (в мнемонической форме) на основе графов потока управления, полученных на предыдущем этапе. Проведённое тестирование на наборе программ подтвердило корректность генерации кода для основных конструкций языка (арифметика, логика, ветвления, циклы, break, вызовы функций и рекурсия), а также работоспособность полученного результата при запуске.

5. Исходные файлы

Исходные файлы проекта: репозиторий [Электронный ресурс]. — GitHub. — Режим доступа: https://github.com/DafterT/parsing_grammar (дата обращения: 15.12.2025).