Содержание

Цель

Изучение составных частей, основных принципов построения и функционирования компиляторов, практическое освоение методов построения простейших компиляторов для заданного входного языка.

Задание

Написать компилятор подмножества языка Fortran 95 с незначительными модификациями и упрощениями.

Компилятор построен из следующих составных частей:

1. лексический анализатор;
2. синтаксический анализатор;
3. семантический анализатор
4. генератор результирующего кода.

Однако, 1, 3 и 4 части очень удобно реализоваить в процессе синтаксического анализа, таким образом лексический анализ, семантический анализ и генерация выходного кода проходит параллельно с синтаксическим анализом.

Компилятор должен запускаться командной строкой с двумя входными параметрами. Первый входной параметр имя входного файла, второй параметр быть имя результирующего файла.

Входной язык компилятора должен удовлетворяет следующим требованиям:

- входная программа начинается ключевым словом *program* и заканчивается *end program*;
- входная программа разбита на строки произвольным образом, все пробелы и переводы строки должны игнорироваться компилятором;
- входная программа должна представлять собой единый модуль, содержащий линейную последовательность операторов, вызовы процедур и функций не предусматриваются;
- должны быть предусмотрены следующие варианты операторов входной программы:
  - оператор присваивания;
  - условный оператор;
  - составной оператор;
  - оператор цикла *while*;
- выражения в операторах содержат следующие операции:
  - арифметические операции сложения, вычитания, умножение и деление;
  - операции сравнения – меньше, больше, равно, не равно;
  - логические операции – и, или, нет;
- операндами в выражениях могут выступать идентификаторы (переменные) и константы (двоичные);
- все идентификаторы, встречающиеся в исходной программе, должны восприниматься как переменные, имеющие тип *quad*, и длиной не более чем в 32 символа.
- не допускается присвоение значений константам;
- в условном операторе в случае истиного (ложного) условии должен присутствовать составной оператор.
- в операторе цикла, в качестве тела цикла должен присутствовать составной оператор.

Грамматика входного языка

P	program B
B	S //S// NL
S	quad I//,I//; | I = E; | **if** (E) **then** B **else** B | **while** (E) B | B
E	E1 //[== | < | > | !=] E1//
E1	T //[+ | - | ||] T//
T	F //[| / | &&] F//
F	I | N | !F | (E)
I	C | IC | IR
N	I | N
C	*a* | *b* | ... | *z* | *A* | *B* | ... |*Z*
R	*0* | *1* | *2* | ... | *9*

Замечание:

a) запись вида // // означает итерацию цепочки , т.е. в порождаемой цепочке в этом месте может находиться либо , либо , либо , либо , и т.д.
b) запись вида [|] означает, что в порождаемой цепочке в этом месте может находиться либо , либо .
c) P - цель грамматики; символ - маркер конца текста программы.

Контекстные условия

1. Любое имя, используемое в программе, должно быть описано и только один раз.
2. В условном операторе и в операторе цикла в качестве условия возможно любое выражение.
3. Операнды операции отношения должны быть целочисленными.

Таблица переменных

В данной работе использовалась таблица для хранения информации о переменных на момент компиляции, т.е. Вычисляемое компилятором значение для инициализации, флаг инициализирована переменная или нет, название переменной. Далее приведена ее структура.

```
class identifier_table
```

```
public:
     typedef std::vector<variable>::iterator iterator;
     iterator begin();
     iterator end();
     void push(variable& var);
     bool get_by_id(int id, variable& var);
     bool get_by_name(const std::string &name, variable& var);
     bool set_by_id(int id, variable &var);
     bool set_by_name(const std::string &name, variable &var);
```

```
private:
     std::vector<variable> items;
```

Функции-члены класса identifier_table

- `typedef std::vector<variable>::iterator iterator` – объявление типа итератора.
- `iterator begin()` – установить итератор в начало вектора.
- `iterator end()` – установить итератор в конец вектора.
- `void push(variable& var)` – вставить переменную в вектор.
- `bool get_by_id(int id, variable& var)` – получить переменную по id, если переменная найдена – true, если нет – false.
- `bool get_by_name(const std::string &name, variable& var)` – получить переменную по имени, если переменная найдена – true, если нет – false.
- `bool set_by_id(int id, variable &var)` – изменить значение переменной по id, если переменная найдена – true, если нет – false.
- `bool set_by_name(const std::string &name, variable &var)` – изменить переменную по имени, если переменная найдена – true, если нет – false.

Данные-члены класса identifier_table

- `std::vector<variable> items` – вектор переменных.

Лексический анализатор

Задача лексического анализа заключается в разборе входной программы. Лексический анализатор разбивает входную программу на лексемы четырех типов: ключевые слова, разделители, константы и идентификаторы. Работой объекта данного класса управляет объект класса синтаксического анализатора. С помощью лексического анализатора происходит не только обнаружение лексем, но и навигация по потоку чтения файла.

На вход лексического анализатора подается файл, далее функция read_lexma получает из файла одну лексему. Лексический анализатор построен на базе конечного автомата, что позволяет определять тип читаемых лексем во время чтения

Лексический и синтаксический анализатор взаимодействуют друг с другом на каждом этапе чтения программы, т.е. сначала синтаксический анализатор посылает запрос на чтение лексем предоставляя контейнер для нее, лексический анализатор читает лексему и если он не выявил лексических ошибок то лексема помещается в предоставленный контейнер.

```
class lexical_analyzer
```

```
private:
    struct {
        int pos;
        int line;
    } position_in_code;
    struct {
        std::ios::pos_type stream_pos;
        int pos;
        int line;
    } bookmark;
    const char* filename;
    std::fstream filestream;
    transition_table table;
    transition_table make_table();
    void backstep();
```

```
public:
```

```
    lexical_analyzer(char* filename);
    ~lexical_analyzer();
    bool read_token(token&);
    void save_statement();
    void roll_back();
    std::string get_pos();
```

Функции-члены класса lexical_analyzer

- `void backstep()` – возвращение символа в поток чтения файла.
- `bool read_token(token&)` – чтение лексемы, если она прочитана корректно то – true, если нет – false.
- `void save_statement()` – сохранение положения в потоке чтения файла, строки и позиции.
- `void roll_back()` – откатиться на сохраненную позицию.
- `std::string get_pos()` – получить строку и позицию.

Данные-члены класса lexical_analyzer

- `int pos` – позиция в строке.
- `int line` – читаемая строка.
- `position_in_code` – структура для объединения первых 2х полей.
- `std::ios::pos_type stream_pos` – сохраненная позиция потока чтения.
- `int pos` – сохраненная позиция.
- `int line` – сохраненная строка.
- `bookmark` – структура для объединения 3х полей выше.
- `const char* filename` – имя входного файла.
- `std::fstream filestream` – поток чтения.
- `transition_table table` – таблица идентификаторов.

Граф переходов

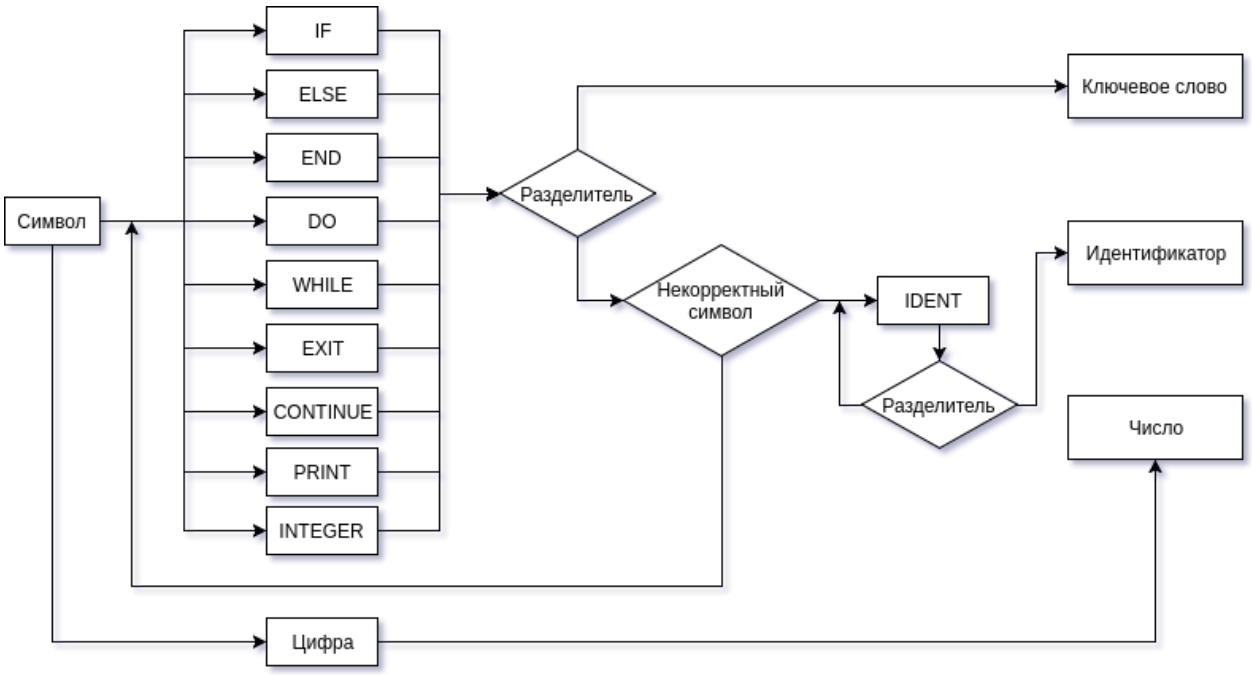


Схема 1 – граф лексического анализа

Синтаксический анализатор

Задача синтаксического анализа состоит в том, чтобы определить имеет ли цепочка лексем конструкцию, заданную синтаксисом языка. В данной работе синтаксический анализатор представляет собой набор функций, реализующий правила грамматики.

На первом этапе проверяется баланс круглых и фигурных скобок, также на этом этапе проверяется вложенность лексем, т.е. если лексема находится в блоке (внутри составного оператора), и ее вложенность на единицу больше вложенности лексем вне этого блока, вложенность *program* равна нулю, т.к. *program* находится вне главного блока программы. Проверяется баланс *if* и *else*, т.е. *else* не может встретится раньше чем *if*, и количество *else* должно совпадать с количеством *if*. Проверяется синтаксис do while, операндом которого является алгебраическое выражение. Также на первом этапе проверяется наличие program и end program.

По мере чтения код рекурсивными функциями раскладывается во внутреннее представление, проверяется семантика.

Для анализа объявления переменных, оператора присваивания, а также условий цикла *while* и условия *if* применяется специально разработанный метод анализа. В связи с тем, что необходимо анализировать лексемы по типу, по значению, по имени, а иногда несколько лексем по разным параметрам – это весьма сложно. Был разработан метод, который позволяет перейти от последовательности лексем к целочисленному массиву. Переход осуществляется по имени, типу и значению.

В объявлении переменных может происходить инициализация, не могут встечастся ключевые слова, а также разделители кроме ‘,’. Последовательность идентификаторов не может начинаться и заканчиваться ‘,’.

В операторе присваивания справа от знака ‘=’ находится арифметическое выражения, в котором могут присутствовать лишь константы и идентификаторы а также знаки арифметических операций. Арифметические операции выполняются согласно приоритету, который может быть изменен круглыми скобками.

В условии цикла *while* и условии условного оператора находится алгебраическое выражение.

Если синтксических и семантических ошибок на этапе чтения инстрекции нет, то она генерируется в выходной.

```
class syntax_analyzer
```

```
private:
     lexical_analyzer lexical;
     table<variable> *idents_table;
     table<function> funcs_table;
     table<function_call> funcs_calls_table;
     code_generator codegen;
     int if_counter;
     int do_counter;
     int cmp_counter;
     int opened_do;
     int rbp_vars_offset;
     int rbp_params_offset;
     const char* outfile;
     bool start_lexma();
     bool definition();
     bool can_read(int n);
     bool if_statement();
```

```
    bool dowhile_statement();
    bool print();
    bool on_expression(std::list<token> &_list);
    bool on_terminal(std::list<token> &_list);
    bool expression_to_tree(std::list<token> &_list, tree_node <token>
    &_root);
    int calculate_expression(std::list<token> &_poliz);
    bool lexma_is(token_type_enum type, std::list<token> &_list);
    bool lexma_is(token_type_enum type, token &t);
    bool function_declare(token t);
    bool program_declare(token t);
    bool function_statement(token &t, token result);
    bool call_function(std::list<token> params, function func);
    bool code_to_poliz(std::list<token> &poliz);
    bool list_to_poliz(std::list<token> &poliz);
    bool output(std::list<token> &_poliz);
    bool program_statement(token &t);
    bool initialization();
```

```
public:
    syntax_analyzer(char* ,const char*);
    bool parse();
```

Функции-члены класса syntax_analyzer

- `bool start_lexma()` – чтение стартовой лексемы, которая определит правила для дальнейшего разбора инструкции.
- `bool definition()` – объявление переменной.
- `bool can_read(int n)` – проверка есть ли еще лексемы.
- `bool if_statement()` – чтение ксловного оператора.
- `bool dowhile_statement()` – чтение цикла do while.
- `bool print()` – чтение оператора вывода.
- `bool On_E(std::list<token> &_list)` – чтение алгебраического выражения и заполнения списка лексем.
- `bool On_T(std::list<token> &_list)` – операнд алгебраического выражения.
- `bool ExprToTree(std::list<token> &_list, tree_node <token> &_root)` – перевод списка лексем выражения в дерево.
- `int calculate_expression(std::list<token> &_poliz)` – вычисление выражения.
- `bool lexma_is(token_type_enum type, std::list<token> &_list)` – проверяет какого типа следующая лексема и заносит ее в список лексем.
- `bool lexma_is(token_type_enum type, token &t)` – проверяет какого типа следующая лексема и заносит ее в переменную.
- `bool code_to_poliz(std::list<token> &poliz)` – перевод в ПОЛИЗ поворотом дерева.
- `bool list_to_poliz(std::list<token> &poliz)` – перевод в ПОЛИЗ поворотом дерева.
- `bool output(std::list<token> &_poliz)` – генерация кода на выходном языке для вычисления выражения.
- `bool program_statement(token &t)` – поиск начала программы и проверка ее конца.
- `bool initialization()` – объявление и инициализация переменной.
- `bool parse()` – запуск процедуры анализа.

- `bool function_declare(token t)` – разбор объявления функции.
- `bool program_declare(token t);` – разбор объявления программы.
- `bool function_statement(token &t, token result)` – разбор тела функции.
- `bool call_function(std::list<token> params, function func)` – вызов подстановки функции в код.

Данные-члены класса identifier_table

- `lexical_analyzer lexical` – лексический анализатор.
- `table<variable> *idents_table` – таблица идентификаторов.
- `table<function> funcs_table` – таблица функций.
- `table<function_call> funcs_calls_table` – таблица вызовов функций.
- `code_generator codegen` – генератор кода.
- `int if_counter` – счетчик условных операторов.
- `int do_counter` – счетчик циклов do_while.
- `int cmp_counter` – счетчик операторов сравнения.
- `int opened_do` – вложенность составного оператора
- `const char* outfile` – имя выходного файла.

Граф синтаксического анализа

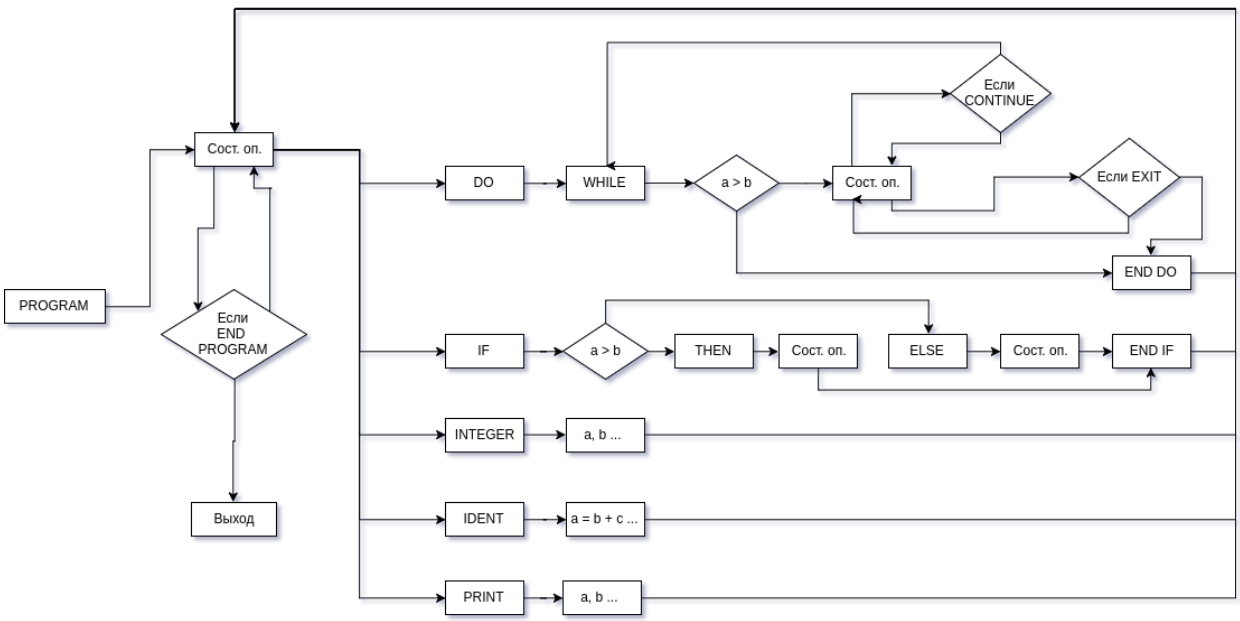


Схема 2 – граф синтаксического анализа

Семантический анализатор

Семантический анализатор представляет собой набор функций (см. функции семантического анализатора). В объявлении переменных все идентификаторы должны быть не объявлены ранее. В присваивании переменной значений выражения в правой части должно иметь целочисленное значение. В условии оператора цикла и в условии условного оператора должно быть логическое выражение.

Внутренне представление программы

Во время работы синтаксического анализатора помимо синтаксического, лексического и семантическонго анализа, происходит параллельная генерация кода, если не было выявлено каких либо ошибок. Выходной код полностью отражает содержимое входной программы.

Генерация результирующего кода

Генерация кода состоит лишь в описании входного языка выходным. В выходной файл заносится ключевое слово *program* и имя программы. Далее формируется блок объявления переменных. Далее идет анализ оставшегося кода. После условия в цикле *while* добавляется ключевое слово *do*, а после условия в операторе *if* добавляется ключевое слово *then*.

Функции генератора кода

```
class code_generator
```

```
private:
     std::string _data;
     std::string _code;
```

```
public:
     void add_line(std::string&);
     void out(std::string&);
     void declare_block(identifier_table);
     void out(const char*);
```

Функции-члены класса code_generator

- `void add_line(std::string&)` – добавление строки выходного языка.
- `void out(std::string&)` – вывод в строку.
- `void declare_block(identifier_table)` – вывод блока объявлений в _data
- `void out(const char*)` – вывод содержимого _data и _code в выходной файл.

Данные-члены класса code_generator

- `std::string _data` – блок объявлений.
- `std::string _code` – блок кода.

Выводы по работе

Результатом работы стал простейший компилятор несколько модифицированного и упрощенного языка Fortran 95.

В соответствии с целями работы, были изучены принципы построения, составные части и идеи функционирования компиляторов. Из множества способов реализации компилятора были выбраны наиболее подходящие для решения конкретной задачи

Для построения любого компилятора необходимо задать грамматику входного языка, для чего в свою очередь необходимо знание теории формальных языков и грамматик. Отсюда можно сделать вывод, что проектирование компилятора – задача в некотором смысле более теоретическая, чем чисто кодирование алгоритма. Само кодирование приложения, при соответствующем владении C++, не составляет большого труда – гораздо сложнее оказывается найти оптимальное решение задачи на теоретическом уровне.

Сообщения и ошибки

Благодаря семейству классов разработанных для оповещения о различных событиях, можно обрабатывать ошибки как исключения, выводить на экран позицию ошибки, и что именно пошло не так. Далее приведен граф наследования.

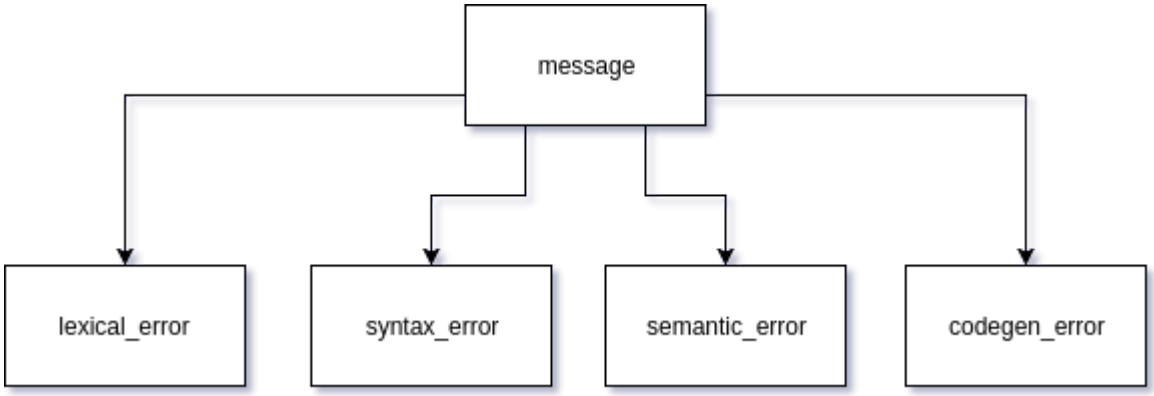
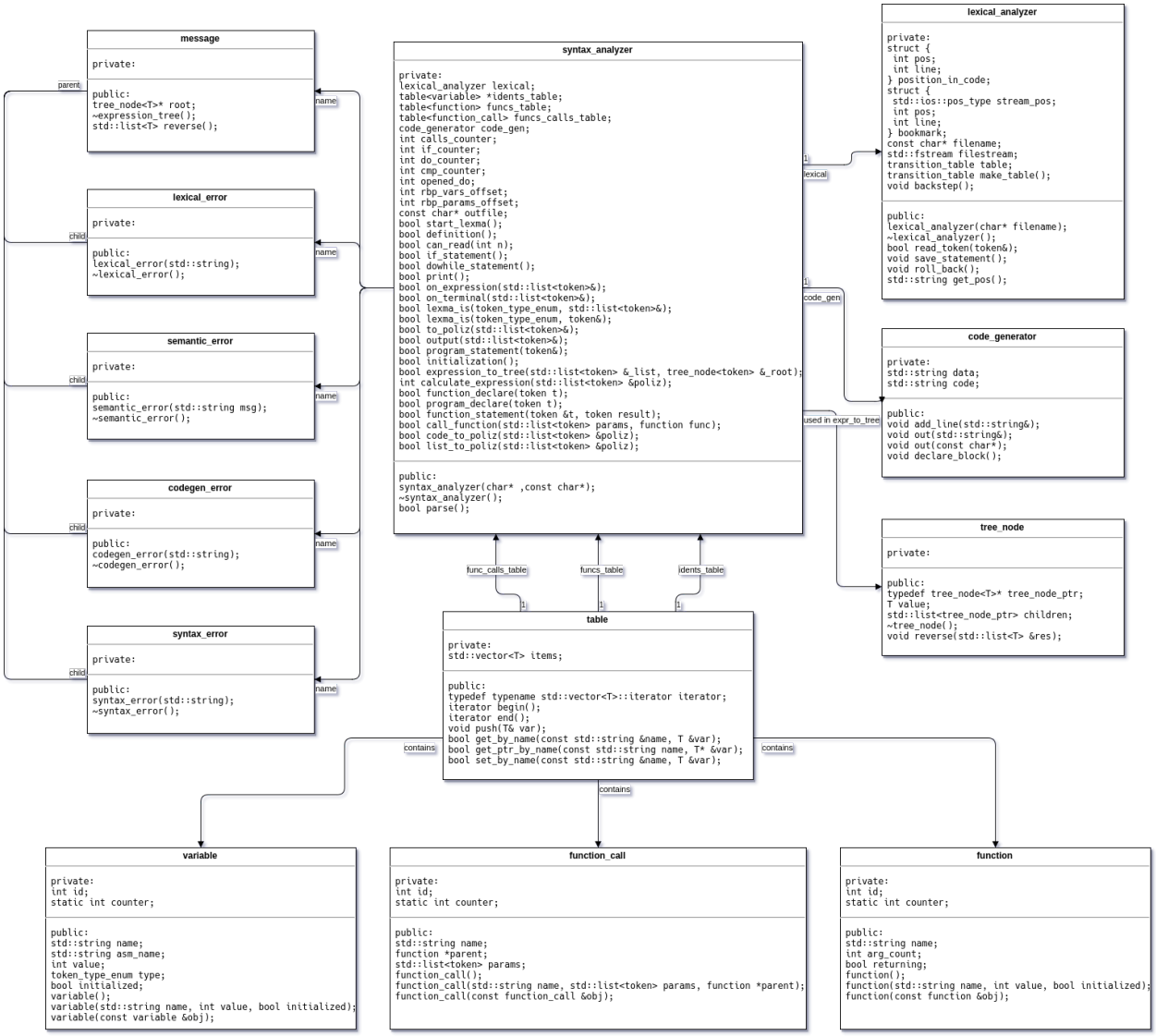


Схема 3 – наследование классов сообщений

Диаграмма классов



Тестирование программы

Программа №1

Код тестовой программы

```
program example
      integer b
      integer::a = 5
      print* a
      b = -2 * (-5)
      print* b
      do while (a<b)
            a=a+1
            if (a > 6) then
                  exit
            end if
            print* a
      end do
end program example
```

Результат работы компилятора

```
.text


.global main
main:
mov     (var_a), %rax
push    %rax
pop     %rax
lea     format(%rip), %rdi
mov     %rax, %rsi
call    printf
push    $0
push    $2
push    $0
push    $5
pop     %rbx
pop     %rax
sub     %rbx, %rax
push    %rax
pop     %rbx
pop     %rax
imul    %rbx
push    %rax
pop     %rbx
pop     %rax
sub     %rbx, %rax
push    %rax
pop     %rax
mov     %rax, (var_b)
mov     (var_b), %rax
push    %rax
pop     %rax
lea     format(%rip), %rdi
```

```
mov     %rax, %rsi
call    printf
dowhile_label_1:
mov     (var_a), %rax
push    %rax
mov     (var_b), %rax
push    %rax
pop     %rbx
pop     %rax
cmp     %rbx, %rax
jge     cmp_less_false_1
jl      cmp_less_true_1
cmp_less_true_1:
mov     $1, %rax
jmp     cmp_less_exit_1
cmp_less_false_1:
mov     $0, %rax
jmp     cmp_less_exit_1
cmp_less_exit_1:
push    %rax
pop     %rax
cmp     $0, %rax
jle     dowhile_exit_label_1
mov     (var_a), %rax
push    %rax
push    $1
pop     %rbx
pop     %rax
add     %rbx, %rax
push    %rax
pop     %rax
mov     %rax, (var_a)
mov     (var_a), %rax
push    %rax
push    $6
pop     %rbx
pop     %rax
cmp     %rbx, %rax
jg      cmp_more_true_2
jle     cmp_more_false_2
cmp_more_true_2:
mov     $1, %rax
jmp     cmp_more_exit_2
cmp_more_false_2:
mov     $0, %rax
jmp     cmp_more_exit_2
cmp_more_exit_2:
push    %rax
pop     %rax
cmp     $0, %rax
je      label_false_1
jmp     dowhile_exit_label_1
```

```
label_false_1:
if_exit_label_1:
mov     (var_a), %rax
push    %rax
pop     %rax
lea     format(%rip), %rdi
mov     %rax, %rsi
call    printf
jmp     dowhile_label_1
dowhile_exit_label_1:
exit:
mov     $1, %rax
int     $0x80

.data
format:  .asciz "%d\n"
var_b:   .quad 0
var_a:   .quad 5
```

Результат работы программы

5 10 6

Программа №2

Код тестовой программы

```
program example
       integer b
       integer::a = 5
       print* a
       b = -2 * (5)
       print* b
       do while (a>b+5)
              a=a-1
              if (a > 0) then
                     continue
              end if
              print* a
       end do
end program example
```

Результат работы компилятора

```
.text

.global main
main:
mov    (var_a), %rax
push   %rax
pop    %rax
lea    format(%rip), %rdi
mov    %rax, %rsi
call   printf
push   $0
push   $2
```

```
push   $5
pop    %rbx
pop    %rax
imul   %rbx
push   %rax
pop    %rbx
pop    %rax
sub    %rbx, %rax
push   %rax
pop    %rax
mov    %rax, (var_b)
mov    (var_b), %rax
push   %rax
pop    %rax
lea    format(%rip), %rdi
mov    %rax, %rsi
call   printf
dowhile_label_1:
mov    (var_a), %rax
push   %rax
mov    (var_b), %rax
push   %rax
push   $5
pop    %rbx
pop    %rax
add    %rbx, %rax
push   %rax
pop    %rbx
pop    %rax
cmp    %rbx, %rax
jg     cmp_more_true_1
jle    cmp_more_false_1
cmp_more_true_1:
mov    $1, %rax
jmp    cmp_more_exit_1
cmp_more_false_1:
mov    $0, %rax
jmp    cmp_more_exit_1
cmp_more_exit_1:
push   %rax
pop    %rax
cmp    $0, %rax
jle    dowhile_exit_label_1
mov    (var_a), %rax
push   %rax
push   $1
pop    %rbx
pop    %rax
sub    %rbx, %rax
push   %rax
pop    %rax
mov    %rax, (var_a)
mov    (var_a), %rax
push   %rax
push   $0
pop    %rbx
pop    %rax
cmp    %rbx, %rax
```

```
jg     cmp_more_true_2
jle    cmp_more_false_2
cmp_more_true_2:
mov    $1, %rax
jmp    cmp_more_exit_2
cmp_more_false_2:
mov    $0, %rax
jmp    cmp_more_exit_2
cmp_more_exit_2:
push   %rax
pop    %rax
cmp    $0, %rax
je     label_false_1
jmp    dowhile_label_1
label_false_1:
if_exit_label_1:
mov    (var_a), %rax
push   %rax
pop    %rax
lea    format(%rip), %rdi
mov    %rax, %rsi
call   printf
jmp    dowhile_label_1
dowhile_exit_label_1:
exit:
mov    $1, %rax
int    $0x80

.data
format:.asciz "%d\n"
var_b: .quad 0
var_a: .quad 5
```

Результат работы программы

5 -10 0 -1 -2 -3 -4 -5

Программа №3

Код тестовой программы

```
program example
        integer a,b
        a=1
        b=10
        do while (a<b)
                print* a
                a = a * (-1)
                if (a<0) then
                        a=a-1
                else
                        a=a+1
                end if
        end do
end program example
```

Результат работы компилятора

```
.text

.global main
main:
push    $1
pop     %rax
mov     %rax, (var_a)
push    $10
pop     %rax
mov     %rax, (var_b)
dowhile_label_1:
mov     (var_a), %rax
push    %rax
mov     (var_b), %rax
push    %rax
pop     %rbx
pop     %rax
cmp     %rbx, %rax
jge     cmp_less_false_1
jl      cmp_less_true_1
cmp_less_true_1:
mov     $1, %rax
jmp     cmp_less_exit_1
cmp_less_false_1:
mov     $0, %rax
jmp     cmp_less_exit_1
cmp_less_exit_1:
push    %rax
pop     %rax
cmp     $0, %rax
jle     dowhile_exit_label_1
mov     (var_a), %rax
push    %rax
pop     %rax
lea     format(%rip), %rdi
mov     %rax, %rsi
call    printf
mov     (var_a), %rax
push    %rax
push    $0
push    $1
pop     %rbx
pop     %rax
sub     %rbx, %rax
push    %rax
pop     %rbx
pop     %rax
imul    %rbx
push    %rax
pop     %rax
mov     %rax, (var_a)
mov     (var_a), %rax
push    %rax
push    $0
pop     %rbx
pop     %rax
cmp     %rbx, %rax
jge     cmp_less_false_2
jl      cmp_less_true_2
cmp_less_true_2:
```

```
mov     $1, %rax
jmp     cmp_less_exit_2
cmp_less_false_2:
mov     $0, %rax
jmp     cmp_less_exit_2
cmp_less_exit_2:
push    %rax
pop     %rax
cmp     $0, %rax
je      label_false_1
mov     (var_a), %rax
push    %rax
push    $1
pop     %rbx
pop     %rax
sub     %rbx, %rax
push    %rax
pop     %rax
mov     %rax, (var_a)
jmp     if_exit_label_1
label_false_1:
mov     (var_a), %rax
push    %rax
push    $1
pop     %rbx
pop     %rax
add     %rbx, %rax
push    %rax
pop     %rax
mov     %rax, (var_a)
if_exit_label_1:
jmp     dowhile_label_1
dowhile_exit_label_1:
exit:
mov     $1, %rax
int     $0x80

.data
format:  .asciz "%d\n"
var_a:   .quad 0
var_b:   .quad 0
```

Результат работы программы

```
1 -2 3 -4 5 -6 7 -8 9 -10
```

Программа №4

Код тестовой программы

```
function factorial (integer a, integer b) result (b)
        integer c
        a = a - 1
        if (a > 0) then
                b = b * factorial(a, b+1)
        end if
end function factorial

program example
        integer::c = 0, b=3
```

```
        c = factorial((1*2)*2+(b), 1)
        print* c
end program example
```

Результат работы компилятора

```
.text

.global main

factorial:
pushq   %rbp
movq    %rsp, %rbp
pushq   $0
movq    16(%rbp), %rax
pushq   %rax
pushq   $1
popq    %rbx
popq    %rax
subq    %rbx, %rax
pushq   %rax
popq    %rax
movq    %rax, 16(%rbp)
movq    16(%rbp), %rax
pushq   %rax
pushq   $0
popq    %rbx
popq    %rax
cmpq    %rbx, %rax
jg cmp_more_true_1
jle cmp_more_false_1
cmp_more_true_1:
movq    $1, %rax
jmp cmp_more_exit_1
cmp_more_false_1:
movq    $0, %rax
jmp cmp_more_exit_1
cmp_more_exit_1:
pushq   %rax
popq    %rax
cmpq    $0, %rax
je label_false_1
movq    24(%rbp), %rax
pushq   %rax
movq    24(%rbp), %rax
pushq   %rax
pushq   $1
popq    %rbx
popq    %rax
addq    %rbx, %rax
pushq   %rax
movq    16(%rbp), %rax
```

```
pushq   %rax
call    factorial
addq    $16, %rsp
pushq   %rax
popq    %rbx
popq    %rax
mulq    %rbx
pushq   %rax
popq    %rax
movq    %rax, 24(%rbp)
label_false_1:
if_exit_label_1:
movq    24(%rbp), %rax
movq    %rbp, %rsp
popq    %rbp
ret

program:
pushq   %rbp
movq    %rsp, %rbp
pushq   $0
pushq   $3
pushq   $1
pushq   $1
pushq   $2
popq    %rbx
popq    %rax
mulq    %rbx
pushq   %rax
pushq   $2
popq    %rbx
popq    %rax
mulq    %rbx
pushq   %rax
movq    -16(%rbp), %rax
pushq   %rax
popq    %rbx
popq    %rax
addq    %rbx, %rax
pushq   %rax
call    factorial
addq    $16, %rsp
pushq   %rax
popq    %rax
movq    %rax, -8(%rbp)
movq    -8(%rbp), %rax
pushq   %rax
popq    %rsi
movq    $format, %rdi
movq    $0, %rax
call    printf
movq    $0, %rax
```

```
movq    %rbp, %rsp
popq    %rbp
ret

main:
call    program
movq    $1, %rax
int     $0x80

.data
format:  .asciz "%d\n"
```

Результат работы программы

```
5040
```

Приложение

На CD-диске приложены исходные коды курсовой работы, данный отчет и тестовые программы.

Список литературы

1) И.А.Волкова, Т.В.Руденко. Формальные грамматики и языки. Элементы теории трансляции. Москва, 1996.

2) Ю.Г.Карпов. Основы построения трансляторов. BHV Санкт-Петербург, 2005.