Санкт-Петербургский Политехнический Университет Петра Великого

Институт компьютерных наук и технологий

**Курсовая работа**

**Расширение языка Milan для построения транслятора**

по дисциплине “Теория автоматов и формальных языков”

Выполнила студентка   
группы 3530904/00103: Бабушкин Е. А.

Преподаватель: Тышкевич А. И.

Санкт-Петербург, 2023

**Содержание**

[Постановка задачи 3](#_Toc134050507)

[Изменения в грамматике 4](#_Toc134050508)

[Семантика новой конструкции 5](#_Toc134050509)

[Программная реализация 6](#_Toc134050510)

[Примеры 10](#_Toc134050511)

[Заключение 14](#_Toc134050512)

# Постановка задачи

Внести изменения в компилятор согласно заданию:

**Вариант №7**

Оператор выбора:

***switch***(<выражение>)

***begin***

***case*** <выражение> : <список операторов>***break***;

***case*** <выражение>:<список операторов> ***break***;

…

***default:*** <список операторов>

***endcase***

# Изменения в грамматике

Для выполнения поставленной задачи в грамматику языка были внесены следующий изменения:

1. Добавлена новая КС-грамматика для ⟨case⟩ и ⟨caseList⟩:

⟨case⟩ ::= ‘case’⟨expression⟩ : ⟨statementList⟩’break’;

⟨caseList⟩ ::= ⟨case⟩ {⟨case⟩}

1. Дополнена КС-грамматика ⟨statement⟩:

⟨statement⟩ ::= ⟨ident⟩ ‘:=’ ⟨expression⟩

| ‘switch’ (⟨expression⟩)

‘begin’

⟨caseList⟩

‘default:’ ⟨statementList⟩

‘endcase’

| ‘if’ ⟨relation⟩ ‘then’ ⟨statementList⟩ [‘else’ ⟨statementList⟩] ‘fi’

| ‘while’ ⟨relation⟩ ‘do’ ⟨statementList⟩ ‘od’

| ‘write’ ‘(’ ⟨expression⟩ ‘)’

Остальная грамматика осталась без изменений.

# 

# Семантика новой конструкции

Семантика новой конструкции фактически заменяет собой огромные конструкции if…else if…else if…else fi fi fi.

Новая конструкция полностью аналогична конструкции:

‘if’ ⟨relation⟩ ‘then’ ⟨statementList⟩

[‘else’ ‘if’ ⟨relation⟩ ‘then’ ⟨statementList⟩ [‘else’ ⟨statementList⟩] ‘fi’] ‘fi’

Количество ‘if’-ов соответствует количеству меток ‘case’, а ‘else’ последнего ‘if’ будет соответствовать метке ‘default’.

⟨relation⟩ в ‘if’ же представляет из себя сравнение значения выражения, которое в случае реализованного ‘switch’ находилось бы в скобках после данного ключевого слова и значений выражений, которые бы находились после меток ‘case’.

|  |  |
| --- | --- |
| Оператор SWITCH | Аналогичная конструкция if-then-else |
| switch (*выражение1*)  begin  case *выражение2*: *инструкция1*  break;  case *выражение3*: *инструкция2*  break;  case *выражение4*: *инструкция3*  break;  …  default: *инструкцияN*  endcase | if(*выражение1 = выражение2*) then *инструкция1*  else if(*выражение1 = выражение3*) then *инструкция2*  else if(*выражение1 = выражениe4*) then *инструкция3*  …  else *инструкцияN* fi  … fi fi |

# Программная реализация

Для реализации оператора выбора *switch* было добавлено:

Ключевые слово ‘SWITCH’, ‘CASE’, ‘BREAK’, ‘DEFAULT’, ‘ENDCASE’ и ‘:’ в перечисление *Token*:

enum Token {

…

T\_SWITCH, // Ключевое слово "switch"

T\_CASE, // Ключевое слово "case"

T\_BREAK, // Ключевое слово "break"

T\_DEFAULT, // Ключевое слово "default"

T\_ENDCASE, // Ключевое слово "endcase"

T\_COLON // Ключевое слово ':'

};

К двумерному массиву символов *tokenNames\_* добавлены те же ключевые слова:

static const char \* tokenNames\_[] = {

…

"'switch'",

"'case'",

"'break'",

"'default'",

"'endcase'",

"':'"

};

К инициализации ассоциативного массива *keywords\_* в конструкторе лексического анализатора *Scanner* также добавлен перевод цепочки символов для ключевых слов:

explicit Scanner(const string& fileName, istream& input)

: fileName\_(fileName), lineNumber\_(1), input\_(input)

{

…

keywords\_["switch"] = T\_SWITCH;

keywords\_["case"] = T\_CASE;

keywords\_["break"] = T\_BREAK;

keywords\_["default"] = T\_DEFAULT;

keywords\_["endcase"] = T\_ENDCASE;

nextChar();

}

Были внесены изменения в метод statement() класса *Parser.* При встрече лексемы ‘switch’ транслятор проверяет наличие следующих лексем в порядке: проверка на наличие открывающейся скобки, выражения, наличие закрывающей скобки, лексемы ‘begin’, любого количества следующих лексем в порядке: лексема ‘case’, выражение, лексема ‘:’, список операторов и лексемы ‘break’ и ‘;’. Далее, в конце оператора выбора должны находиться: лексемы ‘default’, ‘:’, список операторов и лексема ‘endcase’, завершающая оператор ‘switch’.

Алгоритм работы синтаксического анализатора:

Если очередная лексема равна T\_SWITCH, синтаксический анализатор имеет дело с оператором выбора SWITCH. Затем анализатор проверяет, что следующая лексема совпадает с T\_LPAREN. После знака ‘(’ в программе должно следовать арифметическое выражение, поэтому анализатор вызывает метод expression(). Этот метод проверяет правильность арифметического выражения и генерирует последовательность команд для его вычисления. В результате выполнения этой последовательности на вершине стека будет находиться значение выражения. Далее мы должны сохранить результат в новую переменную с названием “switchValue compilation variable” в память командой STORE. Далее анализатор проверяет, что следующая лексема – T\_RPAREN, а за ней T\_BEGIN.

Вводим флаг-переменную (DF), в которой будет храниться ‘0’, если ни одна метка case не была выполнена, или значение отличное от нуля, если метка была выполнена. Мы добавляем в стек значение ‘0’ командой PUSH, далее значение сохраняется командой STORE.

Далее необходимо обработать все метки ‘case’, которых может быть любое количество. Для этого используется цикл с постусловием, для окончания которого компилятор должен встретить лексему T\_DEFAULT. Опишем список операторов внутри цикла.

Сначала анализатор проверяет наличие лексемы T\_CASE. Далее, чтобы избежать выполнения сразу нескольких ‘case’ в одном операторе ‘switch’, мы загружаем из память флаг DF и если он равен 1, то переходим сразу к следующему ‘case’. Если DF = 0 - вызывается метод expression(), который кладёт на вершину стека результат выражения. Далее в стек загружается значение SVCV и выполняется команда COMPARE. Есть 2 случая: SVCV не равно выражению после метки ‘case’ или совпадает с ним. Для обработки данных случаев добавляется команда JUMP\_NO после сравнения, которая выполнит переход к следующему case в случае, если COMPARE положит 0 в верхушку стека. В ситуации совпадения анализатор проверяет, что далее следует лексема T\_COLON (‘:’) и далее вызывается метод statementList(), который выполняет код до лексемы T\_BREAK.  
После завершения statementList() выполняется проверка на соответствующую лексему и далее проверка на T\_SEMICOLON. Также в стек командой PUSH загружается ‘1’ и командой STORE загружается в флаг DF.

В последнем ‘case’ перед лексемой T\_DEFAULT в конце будет находиться команда JUMP\_YES (прыжок в конец оператора SWITCH), перед выполнением которой в стек командой LOAD загружается результат сравнения из начала цикла с предусловием.

После обработки лексемы T\_DEFAULT и выхода из цикла анализатор убеждается, что далее следует лексема T\_COLON (‘:’). Далее в стек загружается флаг DF и команда JUMP\_YES, которая перейдет за конец метки ‘default’.

В случае если DF = 0 вызывается метод statementList().

Структура кода оператора SWITCH:

PUSH 0

STORE DFadress  
switchLabel: вычисление выражения внутри скобок (SVCV)

STORE SVCVadress  
…  
caseLabel2:

STORE DFadress

JUMP\_YES caseLabelN

вычисление выражения

LOAD SVCVadress

STORE

COMPARE 0

JUMP\_NO caseLabel3

Операторы тела case2

PUSH 1

STORE DFadress  
  
caseLabelN:

…

LOAD DFadress

JUMP\_YES endcaseLabel

defaultLabel:

операторы тела default

endcaseLabel:  
  
В конце выполняется проверка лексемы T\_ENDCASE.  
  
Код расширенной функции statement() в parser.cpp:

else if (match(T\_SWITCH)) {

mustBe(T\_LPAREN);

expression();

//Создаем переменную для хранения в памяти результата выражения в switch

int switchVariableAdress = findOrAddVariable("switchValue compilation variable");

codegen\_->emit(STORE, switchVariableAdress);

//Индикатор, показывающий, было ли выполнено хотя бы одно условие case (то есть должен ли выполниться default)

int isDefaultNotNeed = findOrAddVariable("indicator compilation variable");

//Изначально значение 0

codegen\_->emit(PUSH, 0);

codegen\_->emit(STORE, isDefaultNotNeed);

mustBe(T\_RPAREN);

// Проверяем, что следующее ключевое слово - 'begin'

mustBe(T\_BEGIN);

int jumpYesAddress;

// Обрабатываем все встречающиеся 'case'

bool isJumpYesInitialized = false; //Переменная чтобы мы могли проверять,

//инициализирована ли предыдущая переменная

//Это возможно когда первый case синтаксически

//не верен и был выполнен break из while.

do

{

if (!match(T\_CASE)) break;

//Проверяем, был ли уже выполнен какой-либо case, если да - пропускаем

codegen\_->emit(LOAD, isDefaultNotNeed);

int jumpToNextCase = codegen\_->reserve();

// Считаем выражение и сравниваем со значением внутри switch

expression();

codegen\_->emit(LOAD, switchVariableAdress);

codegen\_->emit(COMPARE, 0);

int compareResult = findOrAddVariable("compareResult compilation variable");

codegen\_->emit(STORE, compareResult);

codegen\_->emit(LOAD, compareResult);

int skipCaseAddress = codegen\_->reserve();

//Обрабатываем набор команд внутри case

if (!match(T\_COLON)) break;

statementList();

if (!match(T\_BREAK)) break;

if (!match(T\_SEMICOLON)) break;

//Записываем в индикатор, что один из case подошёл

codegen\_->emit(PUSH, 1);

codegen\_->emit(STORE, isDefaultNotNeed);

isJumpYesInitialized = true;

codegen\_->emitAt(skipCaseAddress, JUMP\_NO, codegen\_->getCurrentAddress());

codegen\_->emitAt(jumpToNextCase, JUMP\_YES, codegen\_->getCurrentAddress());

} while (!match(T\_DEFAULT));

mustBe(T\_COLON);

// Пропускаем default, если сработал хотя бы один case

codegen\_->emit(LOAD, isDefaultNotNeed);

int skipDefaultAddress = codegen\_->reserve();

statementList();

mustBe(T\_ENDCASE);

codegen\_->emitAt(skipDefaultAddress, JUMP\_YES, codegen\_->getCurrentAddress());

}

# Примеры

1. Код программы:

begin

n := 2;

switch(n)

begin

case 1+2: write(1) break;

case 3-2: write(2) break;

case 4+7: write(3) break;

default: write(4)

endcase

end

Код стековой машины:

0: PUSH 2

1: STORE 0

2: LOAD 0

3: STORE 1

4: PUSH 0

5: STORE 2

6: LOAD 2

7: JUMP\_YES 20

8: PUSH 1

9: PUSH 2

10: ADD

11: LOAD 1

12: COMPARE 0

13: STORE 3

14: LOAD 3

15: JUMP\_NO 20

16: PUSH 1

17: PRINT

18: PUSH 1

19: STORE 2

20: LOAD 2

21: JUMP\_YES 34

22: PUSH 3

23: PUSH 2

24: SUB

25: LOAD 1

26: COMPARE 0

27: STORE 3

28: LOAD 3

29: JUMP\_NO 34

30: PUSH 2

31: PRINT

32: PUSH 1

33: STORE 2

34: LOAD 2

35: JUMP\_YES 48

36: PUSH 4

37: PUSH 7

38: ADD

39: LOAD 1

40: COMPARE 0

41: STORE 3

42: LOAD 3

43: JUMP\_NO 48

44: PUSH 3

45: PRINT

46: PUSH 1

47: STORE 2

48: LOAD 2

49: JUMP\_YES 52

50: PUSH 4

51: PRINT

52: STOP

Результат программы:

Изображение выглядит как Шрифт, текст, белый, каллиграфия

Автоматически созданное описание

1. Код программы:

begin

n := 2;

switch(n

case 1+2: write(1) break;

cas 3-2: write(2) break;

case 4+7: write(3) break;

default write(4)

endswitch

end

Результат программы:

Изображение выглядит как текст

Автоматически созданное описание

Пояснение:

1) После ‘switch’ компилятор ожидает увидеть выражение в скобках, если после выражения он видит не ‘)’ – выводится ошибка (в данном примере он видит ‘c’).

2) Компилятор после закрывающей скобки ожидал ‘begin’, но получил case – он вывел ошибку, пропустил его, вместо ‘begin’ увидел ключевое слово ‘break’ и вывел соответствующую ошибку.  
  
3), 4) if (!match(T\_CASE)) break; - как мы видим в коде компилятора, если не было обнаружено ключевое слово ‘case’ – обработка сразу переходит к проверке ‘default’ и дальнейшего кода. Он видит пропущенный символ ‘:’ после ‘default’. Далее он проверяет statement. В функции было расширено условие выхода: if(see(T\_END) || see(T\_OD) || see(T\_ELSE) || see(T\_FI) || see(T\_ENDCASE)) {return;}  
Но так как ключевое слово ‘endcase’ написано некорректно, то функция не может корректно обработать код и выводит ошибку.  
  
5), 6) ‘endcase’ написано неверно и компилятор выдает соответствующую ошибку. Далее компилятор ожидает ‘END’, но компилятор его пропускает из-за предыдущих ошибок и получает токен T\_EOF, который не является T\_END и выводит соответствующую ошибку.

Семантическая диаграмма:

Изображение выглядит как диаграмма

Автоматически созданное описание

# Заключение

Транслятор языка Milan был дополнен новым оператором ‘SWITCH’. Было проведено соответствующее тестирование для проверки правильности работы синтаксического анализатора и всего транслятора языка Milan в код стековой машины. На виртуальной стековой машине также был проведен тест, показавший правильность трансляции входной строки. Были освоены основы работы компилятора, транслятора и синтаксического анализатора, на основе которого и проводилось дополнение языка Milan.