Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий
институт

Кафедра информатики
кафедра

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №3

Разработка блока генерации промежуточного кода транслятора простого языка программирования
тема

Преподаватель _____ А. С. Кузнецов
подпись, дата — инициалы, фамилия

Студент КИ19-04-1М, 031943329 _____ И. С. Байкалов
номер группы, зачетной книжки — подпись, дата — инициалы, фамилия

Красноярск 2020

СОДЕРЖАНИЕ

1 Цель 3
2 Задачи 3
3 Ход работы 3
3.1 Листинг кода разработанного кодогенератора 3
4 Вывод 8
ПРИЛОЖЕНИЕ А 9

1 Цель

Изучение методов генерации промежуточного кода с их программной реализацией.

2 Задачи

Изучение теоретического материала по организации генерации промежуточного кода компиляторов простых языков программирования.

2. Составление формального описания синтаксически-управляемого транслятора с действиями по генерации промежуточного кода.

3. Программная реализация компилятора в промежуточный код по формальному описанию.

3 Ход работы

В ходе работы был реализован функционал генерации промежуточного кода на основе лабораторной работы №2.

3.1 Листинг кода разработанного кодогенератора

Листинг кода полученного кодогенератора представлен в таблице 2. Данный код представляет собой функции, которые были внедрены в разработанный ранее файл парсера olmeca.y.

Таблица 2 – Листинг кода парсера.

```
int unCodegen(FILE* out_file, int operatorCode, NodeAST* operand,
NodeAST* result) {
  TSymbolTableElementPtr var;

  if (operatorCode == ASSIGN_OPERATOR) {
    var = (reinterpret_cast<TSymbolTableReference *> (result))-
>variable;
```

```
    fprintf(out_file, "\t%s\t=\t", var->table->data[var->index].name-
>c_str());
  } else {
    fprintf(out_file, "ASSIGN_%d\t=\t-", result->tmp_index);
  }

  if (operand->nodeType == typeIdentifier) {
    var = (reinterpret_cast<TSymbolTableReference *> (operand))-
>variable;

    fprintf(out_file, "%s", var->table->data[var->index].name-
>c_str());
  } else if (operand->tmp_index != -1) {
    fprintf(out_file, "$STATE_%d", operand->tmp_index);
  } else {
    TValueNode *node = reinterpret_cast<TValueNode *>(operand);

    switch (node->nodeType) {
      case typeIntConst:
        fprintf(out_file, "%d", node->iNumber);
        break;
      case typeFloatConst:
        fprintf(out_file, "%g", node->fNumber);
        break;
      case typeStringConst:
        fprintf(out_file, "%s", node->str->c_str());
        break;
      case typeCharConst:
        fprintf(out_file, "%c", node->ch);
        break;
      case typeBinaryOp:
      case typeUnaryOp:
      case typeAssignmentOp:
      case typeIdentifier:
      case typeIfStatement:
      case typeWhileStatement:
      case typeFunctionStatement:
      case typeFunctionCall:
      case typeList:
      case typeError:
        break;
    }
  }

  fprintf(out_file, "\n");
  return 1;
}

int getCompareOperator(char* comparison) {
  if (!strcmp(comparison, ">")) {
    return MORE_OPERATOR;
  }

  if (!strcmp(comparison, "<")) {
    return LESS_OPERATOR;
  }
```

```
  if (!strcmp(comparison, "==")) {
    return EQUALS_OPERATOR;
  }

  if (!strcmp(comparison, "!=")) {
    return NOT_EQUALS_OPERATOR;
  }

  return 0;
}

int getMulopOperator(char* op) {
  if (!strcmp(op, "*")) {
    return MULTY_OPERATOR;
  }

  if (!strcmp(op, "/")) {
    return DIV_OPERATOR;
  }

  return 0;
}


int binCodegen(FILE* out_file, int operatorCode, NodeAST* leftOperand,
NodeAST* rightOperand, NodeAST* result) {
  fprintf(out_file, "\tSTATE_%u\t=\t", result->tmp_index);

  if (leftOperand->nodeType == typeIdentifier) {
    TSymbolTableElementPtr var =
(reinterpret_cast<TSymbolTableReference *> (leftOperand))->variable;

    fprintf(out_file, "%s", var->table->data[var->index].name-
>c_str());
  } else if (leftOperand->tmp_index != -1) {
    fprintf(out_file, "STATE_%d", leftOperand->tmp_index);
  } else {
    TValueNode *node = reinterpret_cast<TValueNode *>(leftOperand);

    switch (node->nodeType) {
      case typeIntConst:
        fprintf(out_file, "%d", node->iNumber);
        break;
      case typeFloatConst:
        fprintf(out_file, "%g", node->fNumber);
        break;
      case typeStringConst:
        fprintf(out_file, "%s", node->str->c_str());
        break;
      case typeCharConst:
        fprintf(out_file, "%c", node->ch);
        break;
      case typeBinaryOp:
      case typeUnaryOp:
      case typeAssignmentOp:
      case typeIdentifier:
      case typeIfStatement:
```

```
      case typeWhileStatement:
      case typeFunctionStatement:
      case typeFunctionCall:
      case typeList:
      case typeError:
        break;
    }
  }

  switch (operatorCode) {
    case ADD_OPERATOR:
      fprintf(out_file, " + ");
      break;
    case SUBTRACT_OPERATOR:
      fprintf(out_file, " - ");
      break;
    case MULTY_OPERATOR:
      fprintf(out_file, " * ");
      break;
    case DIV_OPERATOR:
      fprintf(out_file, " / ");
      break;
    case MORE_OPERATOR:
      fprintf(out_file, " > ");
      break;
    case LESS_OPERATOR:
      fprintf(out_file, " < ");
      break;
    case EQUALS_OPERATOR:
      fprintf(out_file, " == ");
      break;
    case NOT_EQUALS_OPERATOR:
      fprintf(out_file, " != ");
      break;
    }

  if (rightOperand->nodeType == typeIdentifier) {
    TSymbolTableElementPtr var =
(reinterpret_cast<TSymbolTableReference *> (rightOperand))->variable;

    fprintf(out_file, "%s", var->table->data[var->index].name-
>c_str());
  } else if (rightOperand->tmp_index != -1) {
    fprintf(out_file, "STATE_%d", rightOperand->tmp_index);
  } else {
    TValueNode *node = reinterpret_cast<TValueNode *>(rightOperand);

    switch (node->nodeType) {
      case typeIntConst:
        fprintf(out_file, "%d", node->iNumber);
        break;
      case typeFloatConst:
        fprintf(out_file, "%g", node->fNumber);
        break;
      case typeStringConst:
        fprintf(out_file, "%s", node->str->c_str());
        break;
```

```
      case typeCharConst:
        fprintf(out_file, "%c", node->ch);
        break;
      case typeBinaryOp:
      case typeUnaryOp:
      case typeAssignmentOp:
      case typeIdentifier:
      case typeIfStatement:
      case typeWhileStatement:
      case typeFunctionStatement:
      case typeFunctionCall:
      case typeList:
      case typeError:
        break;
    }
  }
  fprintf(out_file, "\n");
  return 1;
}

int gotoCodegen(FILE* out_file, int operatorCode, int labelNumber,
NodeAST* optionalExpression) {
  if (operatorCode != GOTO_OPERATOR) {
    if (operatorCode == GOTO_FALSE_WAY_OPERATOR)
      fprintf(out_file, "\tIF FALSE WAY ");
    else if(operatorCode == GOTO_TRUE_WAY_OPERATOR)
      fprintf(out_file, "\tIF TRUE WAY ");
    if (optionalExpression->nodeType == typeIdentifier) {
      TSymbolTableElementPtr var =
(reinterpret_cast<TSymbolTableReference *> (optionalExpression))-
>variable;

      fprintf(out_file, "%s ", var->table->data[var->index].name-
>c_str());
    } else if (optionalExpression->tmp_index != -1) {
      fprintf(out_file, "STATE_%d ", optionalExpression->tmp_index);
    } else {
      TValueNode *node = reinterpret_cast<TValueNode
*>(optionalExpression);

      switch (node->nodeType) {
        case typeIntConst:
          fprintf(out_file, "%d ", node->iNumber);
          break;
        case typeFloatConst:
          fprintf(out_file, "%g ", node->fNumber);
          break;
        case typeStringConst:
          fprintf(out_file, "%s ", node->str->c_str());
          break;
           case typeCharConst:
             fprintf(out_file, "%c", node->ch);
          break;
        case typeBinaryOp:
        case typeUnaryOp:
        case typeAssignmentOp:
        case typeIdentifier:
```

```
        case typeIfStatement:
        case typeWhileStatement:
        case typeFunctionStatement:
        case typeFunctionCall:
        case typeList:
        case typeError:
          break;
      }
    }
  }
  fprintf(out_file, "\tGO TO LABEL_%d", labelNumber);
  fprintf(out_file, "\n");
  return 1;
}

static void PushLabelNumber(int labelNumber) {
  Labels[g_LabelStackPointer] = labelNumber;
  ++g_LabelStackPointer;
}

static int PopLabelNumber(void) {
  if (g_LabelStackPointer > 0) {
    --g_LabelStackPointer;
    return Labels[g_LabelStackPointer];
  } else {
    g_LabelStackPointer = 0;
    return -1;
  }
}

int labelCodegen(FILE* out_file, int labelNumber) {
  fprintf (out_file, "LABEL_%d:\n", labelNumber);
  return 1;
}
```

4 Вывод

В ходе практической работы был реализован блок генерации промежуточного кода транслятора. Были изучены теоретические материалы по организации генерации промежуточного кода компилятора. Результаты тестирования представлены в приложении А.

ПРИЛОЖЕНИЕ А

```
('File:', 'tests/1.olm')
Завершено!


-----     source     -----

integer a;
a = 12;

integer b = 10;

-----     codegen     -----

      a     =     12
      b     =     10

------------------
('File:', 'tests/10.olm')
Завершено!


-----     source     -----

//comment

-----     codegen     -----


------------------
('File:', 'tests/11.olm')
tests/11.olm: 1.1: WTF symbol, bro!? ^
tests/11.olm: tests/11.olm:1.1: syntax error, unexpected invalid token,
expecting end of file

-----     source     -----

^%$

-----     codegen     -----


------------------
```

```
('File:', 'tests/12.olm')
Завершено!


-----     source     -----

integer a;
a = 3;

while (a > 1) {
    if (a == 2) {
        a = a * 1;
    } else {
        a = 0;
    }
}

-----     codegen     -----

     a    =    3
LABEL_0:
     STATE_0    =    a > 1
     IF FALSE WAY STATE_0  GO TO LABEL_1
     STATE_1    =    a == 2
     IF FALSE WAY STATE_1  GO TO LABEL_2
     STATE_2    =    a * 1
     a    =    $STATE_2
     GO TO LABEL_3
LABEL_2:
     a    =    0
LABEL_3:
     GO TO LABEL_0
LABEL_1:

------------------
('File:', 'tests/2.olm')
Завершено!


-----     source     -----

//comment
integer a = 2;

-----     codegen     -----

     a    =    2

------------------
```

```
('File:', 'tests/3.olm')
Завершено!


-----     source     -----

float a;
a = 1.2;

float b = 0.123456789;

-----     codegen     -----

      a     =     1.2
      b     =     0.123457

------------------
('File:', 'tests/4.olm')
Завершено!


-----     source     -----

string a;
a = "test";

string b = "test2";

-----     codegen     -----

      a     =     "test"
      b     =     "test2"

------------------
('File:', 'tests/5.olm')
Завершено!


-----     source     -----

char a;
a = 'h';

-----     codegen     -----

      a     =     h

------------------
```

```
('File:', 'tests/6.olm')
Завершено!


-----     source     -----

integer a;
a = 3;

a = a + 2;
a = a - 2;
a = a * 2;
a = a / 2;

-----     codegen     -----

      a     =     3
      STATE_0     =     a + 2
      a     =     $STATE_0
      STATE_1     =     a - 2
      a     =     $STATE_1
      STATE_2     =     a * 2
      a     =     $STATE_2
      STATE_3     =     a / 2
      a     =     $STATE_3

------------------
('File:', 'tests/7.olm')
Завершено!


-----     source     -----

float a;
a = 3.3;

a = a + 2.1;

-----     codegen     -----

      a     =     3.3
      STATE_0     =     a + 2.1
      a     =     $STATE_0

------------------
```

```
('File:', 'tests/8.olm')
Завершено!


-----     source     -----

integer a;
a = 3;

if (a == 2) {
    a = a + 2 * 3 / 2 -1;
} else {
    a = 0;
}

-----     codegen     -----

     a     =     3
     STATE_0     =     a == 2
     IF FALSE WAY STATE_0  GO TO LABEL_0
     STATE_1     =     2 * 3
     STATE_2     =     STATE_1 / 2
     STATE_3     =     a + STATE_2
     STATE_4     =     STATE_3 - 1
     a     =     $STATE_4
     GO TO LABEL_1
LABEL_0:
     a     =     0
LABEL_1:

------------------
```

```
('File:', 'tests/9.olm')
Завершено!


-----     source     -----

integer a;
a = 3;

while (a > 1) {
    if (a == 2) {
        a = a * 1;
    } else {
        a = 0;
    }
}

-----     codegen     -----

     a    =    3
LABEL_0:
     STATE_0    =    a > 1
     IF FALSE WAY STATE_0  GO TO LABEL_1
     STATE_1    =    a == 2
     IF FALSE WAY STATE_1  GO TO LABEL_2
     STATE_2    =    a * 1
     a    =    $STATE_2
     GO TO LABEL_3
LABEL_2:
     a    =    0
LABEL_3:
     GO TO LABEL_0
LABEL_1:

------------------
```

```
('File:', 'tests/func.olm')
Завершено!


-----     source     -----

function sameName() {
    integer test = 1;
    integer test1 = 12;
    integer test12 = 12;
    integer test13 = 12;

    return 0;
}

sameName();

string qwe = "qwerty";


-----     codegen     -----

FUNCTION BEGIN sameName
     test  =    1
     test1 =    12
     test12     =    12
     test13     =    12
FUNCTION END
CALL FUNCTION sameName
     qwe   =    "qwerty"

------------------
```

```
('File:', 'tests/prog.olm')
Завершено!


-----     source     -----

// comment
string name = "Ilya";
string surname;
integer age;
char firstLetterName;

age = 23;

sameName();

function sameName() {
    while (age < 18) {
        integer r = 1;
        if (name != "Ilya") {
            surname = "Baykalov";
            age = 23;
            age= age - 2 * 3;
        } else {
            surname = "NeBaykalov";
        }

        if (age == 23) {
            firstLetterName = 'I';
        }
    }

    return age;
}

-----     codegen     -----

      name  =     "Ilya"
      age   =     23
CALL FUNCTION sameName
FUNCTION BEGIN sameName
LABEL_0:
      STATE_0     =     age < 18
      IF FALSE WAY STATE_0  GO TO LABEL_1
      r     =     1
      STATE_1     =     name != "Ilya"
      IF FALSE WAY STATE_1  GO TO LABEL_2
      surname     =     "Baykalov"
      age   =     23
      STATE_2     =     2 * 3
      STATE_3     =     age - STATE_2
      age   =     $STATE_3
      GO TO LABEL_3
LABEL_2:
      surname     =     "NeBaykalov"
LABEL_3:
      STATE_4     =     age == 23
```

```
      IF FALSE WAY STATE_4  GO TO LABEL_4
      firstLetterName =     I
      GO TO LABEL_5
LABEL_4:
LABEL_5:
      GO TO LABEL_0
LABEL_1:
FUNCTION END

------------------
```