Лабораторная работа №4. Реализация блока генерации промежуточного кода транслятора простого языка программирования.

Цель: изучение методов генерации промежуточного кода с их программной реализацией.

Задачи:

1. Изучение теоретического материала по организации генерации промежуточного кода компиляторов простых языков программирования.
2. Составление формального описания программы синтаксического анализа с включениями действий по генерации промежуточного трехадресного кода .
3. Программная реализация по формальному описанию компилятора в промежуточный код.

Ход работы:

1. Получить у преподавателя индивидуальный вариант задания, который должен предусматривать конструкции требуемого языка программирования, их распознавание и диагностику, некоторые семантические проверки и генерацию промежуточного кода (на примере трехадресного кода).
2. Составить формальное описание программы компиляции простого языка программирования в трехадресный код с использованием математического аппарата формальных контекстно-свободных грамматик.
3. Произвести автоматическую генерацию исходного текста компилятора простого языка программирования в трехадресный код с использованием систем *flex* и *bison*. Отладить полученный компилятор.
4. Написать отчет.

Внимание: в приложениях приведены исходные коды компилятора в трехадресный код простого языка программирования с частичным использованием абстрактных синтаксических деревьев.

Ход защиты:

1. Продемонстрировать преподавателю корректную работу компилятора в трехадресный код.
2. Пояснить работу изученных механизмов по полученному формальному описанию программы компиляции в трехадресный код.

Содержание отчета:

1. Цель работы с постановкой задачи.
2. Полученные спецификации *flex* и *bison*, а также всех вспомогательных модулей.
3. Тестовые примеры работы программы (не менее четырех).

Во время защиты лабораторной работы необходимо иметь при себе исполняемые модули программ, исходные тексты, а также твердую копию отчета.

Краткие теоретические сведения о генерации промежуточного кода

В процессе компиляции нередко используется промежуточное представление программ (промежуточный код, промежуточный язык), которое предназначено в первую очередь для удобства генерации кода и проведения его оптимизации. Промежуточный код близок к реализуемому языку или к машинному языку целевой вычислительной системы. Обычно инструкция промежуточного кода соответствует одной или нескольким машинным команд, а при их генерации из промежуточных инструкций может потребоваться обращение к разным таблицам компилятора. Популярными формами промежуточного кода являются:

- абстрактное синтаксическое дерево;
- ассемблерный код;
- трехадресный код;
- P-код, на котором основано и на который ориентировано большинство реализаций языка Паскаль;
- байт-код, на который ориентированы реализации языка Java.

В данной работе используется трехадресный код.

Трехадресный код

Примером трехадресного кода является цепочка

```
x := y op z
```

В этой цепочке *y* и *z* – операнды, *op* – любая операция, в том числе сравнения, *x* – результат выполнения операции. Так, выражение

```
a*(b+c)*d
```

представляется следующими инструкциями 3-адресного кода.

```
$t1 := b + c
$t2 := a * $t1
$t3 := $t2 * d
```

Здесь переменные $\$t_k$ – создаваемые компилятором временные переменные. Также в этом примере есть лишь бинарные операции, однако нет никаких проблем с использованием унарных операций. Так, унарное отрицание *!e* в языке C реализуется единственной трехадресной инструкцией, но использоваться будут только два адреса.

```
$t1 := not e
```

Трехадресный код применяется также для представления других элементов языков программирования типа обращений к элементам массивов, присваивания, условных и безусловных переходов, вызовов и возвратов из

процедур, вычислений параметров. Их объединяет максимальное количество используемых в операторе адресов – **три**.

3-адресное присваивание выглядит очень просто:

```
$t1 := e
```

Безусловный переход

```
goto $L
```

После этой инструкции будет выполнена инструкция, которая помечена меткой *$L*.

Условный переход типа

```
iftrue x goto $L1
iffalse y goto $L2
```

Эта инструкция применяет проверяет истинность или ложность заданного значения к *x* и *y*, и следующей выполняется инструкция с меткой $*L*, если сравнение *x* и *y* истинно и ложно соответственно. В противном случае выполняется следующая за условным переходом инструкция.

Инструкции

```
param x
```

и

```
call p, n
z := call p, n
```

для вызова процедур и функций и

```
return y
return
```

для возврата из них, где *y* обозначает возвращаемое значение. Обычно они используются в виде следующей последовательности трехадресных инструкций

```
param x1
param x2
...
param xn
call p, n
```

Данная последовательность генерируется в качестве части вызова процедуры $p\ (x_1, x_2, \ldots, x_n)$. В инструкции *call p, n* целое число *n*, указывающее количество действительных параметров, не является излишним в силу того, что вызовы могут быть вложенными.

Индексированные присвоения типа

```
x := y[i]
```

и

```
x[i] := y
```

Первая инструкция присваивает *x* значение, находящееся в *i*-й ячейке

памяти по отношению к *y*. Инструкция *x[i] := y* заносит в *i*-ю ячейку памяти по отношению к *x* значение *y*. В обеих инструкциях *x*, *y* и *i* ссылаются на объекты данных.

Присвоение адресов и указателей вида

```
x := &y
x := *y
```

и

```
*x := y
```

Первая инструкция устанавливает значение *x* равным местонахождению *y* в памяти. Предположительно, *y* представляет собой имя, возможно временное, обозначающее выражение с *l*-значением типа *A[i][j]*, а *x* — имя указателя или временное имя. Таким образом, *r*-значение *x* представляет собой *l*-значение некоторого объекта. Во второй инструкции под *y* подразумевается указатель или временная переменная, *r*-значение которой представляет собой местоположение ячейки памяти. В результате *r*-значение *x* становится равным содержимому этой ячейки. И наконец, третья инструкция **x := y* устанавливает *r*-значение объекта, указываемого *x*, равным *r*-значению *y*.

Выбор приемлемых операторов представляет собой важный вопрос в создании промежуточного представления. Очевидно, что множество операторов должно быть достаточно богатым, чтобы позволить реализовать все операции исходного языка. Небольшое множество операторов легче реализуется на новой целевой машине, однако ограниченное множество инструкций может привести к генерации длинных последовательностей инструкций промежуточного представления для некоторых конструкций исходного языка и добавить работы оптимизатору и генератору целевого кода.

Циклы и условные операторы сводятся к проверакам условий и переходам. Например, оператор *if (выражение) оператор* можно реализовать в 3-адресном коде следующим образом.

```
    код для вычисления выражения
    $t1 := выражение
    iffalse $t1 goto $L1
    код оператора
$L1:
```

Еще один пример иллюстрирует 3-адресную реализацию оператора *do-while* в языке С – *do { оператор } while (выражение);*.

```
$L1: код оператора
     $t1 := выражение
     if $t1 goto $L1
```

В обоих примерах $\$L_k$ – метка, номер которой назначается компилятором. Этот кажущийся легким процесс очень непрост и может потребовать наличия дополнительного стека для меток.

Использованы и рекомендованы следующие источники:

1. CIL (C Intermediate Language) - http://cil.sourceforge.net/
2. http://epaperpress.com/lexandyacc/index.html
3. http://www.linux.org.ru/books/GNU/bison/bison_toc.html
4. Levine, J. flex & bison / J. Levine. – Sebastopol, CA: O'Reilly Media, 2009. – 292 p.
5. Levine, J. lex & yacc, Second Edition / J. Levine, T. Mason, D. Brown. – Sebastopol, CA: O'Reilly Media, 1992. – 384 p.
6. Parrot — Википедия. - http://ru.wikipedia.org/wiki/Parrot
7. p-code - http://ru.wikipedia.org/wiki/UCSD_p-System
8. The LLVM Compiler Infrastructure - http://llvm.org
9. Three address code - http://en.wikipedia.org/wiki/Three_address_code
10. Байт-код — Википедия - http://ru.wikipedia.org/wiki/Байткод
11. Виртуальная машина Parrot - http://ru.wikibooks.org/wiki/Виртуальная_машина_Parrot
12. Компиляция. 3: бизон - http://habrahabr.ru/blogs/programming/99366/
13. Компиляция. 4: игрушечный ЯП - http://habrahabr.ru/blogs/programming/99397/
14. Компиляция. 6: промежуточный код – http://habrahabr.ru/blogs/programming/99592/
15. Костельцев, А.В. Построение интерпретаторов и компиляторов / А.В. Костельцев. – СПб.: Наука и Техника, 2001. – 224 с.

Общая постановка задачи и варианты заданий к лабораторной работе №4

Общие требования:

1) Разработать программу, осуществляющую анализ простого языка программирования и генерацию промежуточного кода в случае их отсутствия. Наличие в языке конструкций, отсутствующих как в общей части задания, так и в индивидуальных вариантах, является ошибочным.

2) На вход компилятора подается внешний файл с текстом программы на языке, описанном в конкретном варианте задания, и имя выходного внешнего файла, в котором при отсутствии ошибок сохраняются трехадресные инструкции. Трехадресный код может создаваться путем обхода явно или неявно создаваемых дерева разбора либо абстрактного синтаксического дерева (AST).

3) На выходе компилятор должен выдавать трехадресный код и/или дерево разбора программы или абстрактное синтаксическое дерево и/или информацию об ошибках и/или таблицу всех лексем с важнейшими характеристиками: тип/класс лексемы; ее литеральное написание; порядковый номер строки, содержащей лексему.

4) Используется интерфейс командной строки, т.е. компилятор в общем случае запускается так:

```
user$: compiler3ac -parameters input.lng output.3ac
```

здесь *user$* – подсказка командной строки; *compiler3ac* – имя исполняемого модуля компилятора в 3-адресный код; *-parameters* – 0 или более параметров командной строки; *input.lng* – пример имени внешнего файла с программой на анализируемом языке; *output.3ac* – пример имени внешнего файла для генерации промежуточного кода при отсутствии в программе ошибок.

5) С помощью параметра командной строки можно отключать и включать функцию получения трехадресного кода. По умолчанию эта функция должна быть включена. Для генерации промежуточного кода:

```
user$: compiler3ac –il input.lng output.3ac
```

Для отключения генерации промежуточного кода:

```
user$: compiler3ac –noil input.lng
```

6) С помощью параметра командной строки можно отключать и включать функцию вывода абстрактного синтаксического дерева (AST). По умолчанию эта функция должна быть отключена. Для вывода AST на экран:

```
user$: compiler3ac –ast input.lng output.3ac
```

Для отключения вывода AST на экран:

```
user$: compiler3ac –noast input.lng output.3ac
```

7) С помощью параметра командной строки можно явно отключать и включать функцию вывода дерева разбора. По умолчанию эта функция должна быть отключена. Для вывода дерева на экран:

```
user$: compiler3ac –tree input.lng output.3ac
```

Для отключения вывода дерева на экран:

```
user$: compiler3ac –notree input.lng output.3ac
```

8) Функции вывода дерева разбора и абстрактного синтаксического дерева являются взаимоисключающими.

9) С помощью параметра командной строки можно явно отключать и включать функцию вывода таблицы лексем. По умолчанию эта функция должна быть отключена. Для вывода таблицы лексем на экран:

```
user$: compiler3ac –lexemes input.lng output.3ac
```

Для отключения вывода таблицы лексем на экран:

```
user$: compiler3ac –nolexemes input.lng output.3ac
```

10) При выводе информации об ошибке обязательны указание на номер строки в исходном файле и как можно более точный характер ошибки. Сообщение вида “*syntax error*” должно выводить только в крайнем случае.

11) Диагностируются лексические, синтаксические и семантические ошибки.

12) Входной язык имеет блочную структуру, то есть могут использоваться простые и составные операторы с соответствующими скобками для обособления групп операторов.

13) Входной язык должен поддерживать массивы элементов заданных типов.

14) Входной язык должен поддерживать простые конструкции для ввода значений переменных, включая элементы массивов, а также для вывода значений переменных, констант и выражений, в том числе элементов массивов.

15) Во входном языке должны быть предусмотрены конструкции досрочного завершения выполнения оператора (например, *break*) или досрочного перехода на начало циклического оператора (например, *continue*).

Вариант 1. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'), операторы циклов с пред- и постусловием, циклов с параметром, операторы условий *if* с необязательной частью *else*, операторы объявления переменных целого и вещественного типов и их массивов. В операторах могут использоваться идентификаторы, десятичные числа с плавающей запятой в обычном и нормализованном экспоненциальном форматах, а также целочисленные константы, элементы массивов, знаки присваивания ('=' и другие), знаки операций и круглые скобки.

Операции: сложение; вычитание; умножение; деление; вычисление остатка; сравнения (<, >, ==, !=, <=, >=); обращение к элементам массивов по индексу; логические (И, ИЛИ, НЕ и другие).

Вариант 2. Входной язык содержит смешанные выражения, разделенные символом точки с запятой (';'), операторы циклов с пред- и постусловием, циклов с параметром, операторы условий *if* с необязательной частью *else*, операторы объявления переменных логического и вещественного типов и их массивов. В операторах могут использоваться идентификаторы, нечувствительные к регистру константы *true* и *false*, десятичные числа с плавающей точкой в обычном формате, элементы массивов, знаки присваивания ('=' и другие), знаки операций и круглые скобки.

Операции: сложение; вычитание; умножение; деление; вычисление остатка; смена знака; сравнения (<, >, ==, !=, <=, >=); обращение к элементам массивов по индексу; логические (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ).

Вариант 3. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы циклов с пред- и постусловием, циклов с параметром, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных строкового и вещественного типов и их массивов. В операторах могут использоваться идентификаторы, десятичные числа с плавающей запятой в обычном и нормализованном экспоненциальном форматах, строковые литералы, знаки присваивания ('=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: сравнение (<, >, ==, !=, >=, <=); арифметика для чисел и строк (сложение, вычитание, умножение, деление, смена знака); логика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 4. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных символьного и вещественного типов и их массивов. В операторах могут использоваться идентификаторы, десятичные числа с плавающей запятой в обычном и нормализованном экспоненциальном форматах, символьные константы в стиле языка программирования C, знаки присваивания ('=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: сравнения (<, >, ==, !=, >=, <=); арифметические (сложение, вычитание, умножение, деление, смена знака); логические (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 5. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных целого и вещественного типов и их массивов. В операторах могут использоваться идентификаторы, римские числа (корректные последовательности заглавных и строчных литер *D*, *C*, *L*, *X*, *V* и *I*), десятичные числа с плавающей точкой в обычном формате, знаки присваивания ('=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: арифметика ('+', '-', '*', '/', унарный '-'); сравнения; логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 6. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, смешанные выражения, разделенные символом точки с запятой (';'), операторы объявления переменных логического, целого и вещественного типов и их массивов. В операторах могут использоваться идентификаторы, логические константы 0 и 1, целые числа, десятичные числа с плавающей

точкой в обычном формате, знаки присваивания ('=' и другие), элементы массивов, знаки оперпий и круглые скобки.

Операции: логическая арифметика (*or*, *xor*, *and*, *not*); численная арифметика (сложение, вычитание, умножение, деление, смена знака); сравнения; обращение к элементам массивов по индексу.

Вариант 7. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных целого и вещественного типов и их массивов. В операторах могут использоваться идентификаторы, римские числа (корректные последовательности заглавных и строчных литер *D*, *C*, *L*, *X*, *V* и *I*), десятичные числа с плавающей точкой в обычном формате, знаки присваивания ('=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: сравнения (<, >, ==, !=, >=, <=); числовая арифметика (сложение, вычитание, умножение, деление, смена знака); логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 8. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных целого и вещественного типов и их массивов. В операторах могут использоваться идентификаторы, римские числа (корректные последовательности заглавных и строчных литер *D*, *C*, *L*, *X*, *V* и *I*), десятичные числа с плавающей точкой в обычном формате, знаки присваивания ('=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: сравнения (<, >, =, !=, >=, <=); числовая арифметика (сложение, вычитание, умножение, деление, смена знака); логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 9. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных целого (*octal*, *decimal*, *hexadecimal*) и вещественного типов и их массивов. В операторах могут использоваться идентификаторы, целочисленные константы в шестнадцатеричной, восьмеричной и десятичной системах, десятичные числа с плавающей точкой, знаки присваивания ('=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: числовая арифметика ('+', '-', '*', '/', '%', унарный '-'), логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); сравнения (<, >, =, !=, >=,

<=); обращение к элементам массивов по индексу.

Вариант 10. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных целых типов (*octal*, *decimal*, *hexadecimal*) и их массивов. В операторах могут использоваться идентификаторы, целочисленные константы в шестнадцатеричной, десятичной и восьмеричной системах, знаки присваивания ('=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: побитовая арифметика (*or*, *xor*, *and*, *not*), числовая арифметика (сложение, вычитание, умножение, деление, смена знака); сравнения (<, >, =, !=, >=, <=); логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 11. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных целых (*octal*, *decimal*, *hexadecimal*) и вещественного типов и их массивов. В операторах могут использоваться идентификаторы, целочисленные константы в шестнадцатеричной, десятичной и восьмеричной системах, десятичные числа с плавающей точкой, знаки присваивания ('=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: сравнения (<, >, ==, !=, >=, <=); числовая арифметика (сложение, вычитание, умножение, деление, смена знака); логическая арфиметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 12. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных целых (*octal*, *decimal*, *hexadecimal*) и строкового типов и их массивов. В операторах могут использоваться идентификаторы, целочисленные константы в шестнадцатеричной, десятичной и восьмеричной системах, символьные константы (один или несколько символов, заключенных в апострофы), знаки присваивания ('=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: сравнения (<, >, =, !=, >=, <=); логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); числовая арифметика (сложение, вычитание, умножение, деление, смена знака); обращение к элементам массивов по индексу.

Вариант 13. Входной язык содержит операторы условий *if* с необязательной

частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных целого и комплексного типов и их массивов. В операторах могут использоваться идентификаторы, комплексных числа (действительная и мнимая части отделяются нечувствительными к регистру символами *I* или *J*, и представляют собой два целых и/или десятичных числа с плавающей точкой в обычном формате), целочисленные константы, знаки присваивания ('=' и другие), знаки операций, элементы массивов и круглые скобки

Операции: числовая арифметика ('+', '-', '*', '/', унарный '-'); операции сравнения; логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 14. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, смешанные выражения, разделенные символом точки с запятой (';'), операторы объявления переменных логического и целого типов и их массивов. В операторах могут использоваться идентификаторы, нечувствительные к регистру константы *T* и *NIL*, битовые строки (последовательности из 0 и 1, начинающиеся с обязательной пары знаков 0*b* или 0*B*), целочисленные константы, знаки присваивания ('=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: логическая арифметика (*or*, *xor*, *and*, *not*), побитовая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); операции сравнения; числовая арфиметика (сложение, вычитание, умножение, деление, смена знака); обращение к элементам массивов по индексу.

Вариант 15. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические и строковые выражения, разделенные символом точки с запятой (';'), операторы объявления переменных строкового и целого типов и их массивов. В операторах могут использоваться идентификаторы, строковые константы (заключенная в двойные кавычки последовательность любых символов, за исключением двойных кавычек), целочисленные константы, знаки присваивания ('=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: сравнения (<, >, ==, !=, >=, <=); логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); численная и строковая арифметика (сложение, вычитание, умножение, деление, обращение); обращение к элементам массивов по индексу.

Вариант 16. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические и строковые выражения, разделенные символом точки с запятой (';'), операторы объявления переменных целых (*binary*, *octal*,

hexadecimal) и строкового типов и их массивов. В операторах могут использоваться идентификаторы, целочисленные константы в восьмеричной и шестнадцатеричной системах, битовые строки (начинающиеся с обязательной пары знаков 0*b* или 0*B* последовательности из 0 и 1), строковые литералы (заключенная в двойные кавычки последовательность любых символов, за исключением двойных кавычек), знаки присваивания ('=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: сравнения (<, >, =, !=, >=, <=); числовая и строковая арифметика (сложение, вычитание, умножение, деление, обращение); логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 17. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных целого и вещественного типов и их массивов. В операторах могут использоваться идентификаторы, битовые строки (начинающиеся с обязательной пары знаков 0*b* или 0*B* последовательности из 0 и 1), целочисленные константы, десятичные числа с плавающей точкой в обычном формате, знаки присваивания ('=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ), сравнения; числовая арифметика (сложение, вычитание, умножение, деление, вычисление остатка, смена знака); побитовая арифметика (*or*, *xor*, *and*, *not*); обращение к элементам массивов по индексу.

Вариант 18. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, смешанные выражения с целыми числами и датами, разделенные символом точки с запятой (';'), операторы объявления переменных целого типа и типа даты и их массивов. В операторах могут использоваться идентификаторы, даты двух любых (на усмотрение разработчика) форматов, целочисленные константы, знаки присваивания ('=' и другие), элементы массивов, знаки и круглых скобок.

Операции: сравнения (<, >, ==, !=, >=, <=), арифметика над числами и датами (сложение, вычитание, умножение, деление, смена знака), логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 19. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, смешанные выражения с целыми числами и временны̆ми величинами, разделенные символом точки с запятой (';'), операторы объявления переменных целого и временно̆го типов и их массивов. В операторах могут использоваться идентификаторы, временны̆е величины двух любых (на

усмотрение разработчика языка) форматов, целочисленные константы, знаки присваивания ('=' и другие), элементы массивов, знаки операций и круглые скобки

Операции: сравнения (<, >, ==, !=, >=, <=); арифметика над числами и временными величинами(сложение, вычитание, умножение, деление, смена знака), логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 20. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения разделенные символом точки с запятой (';'), операторы объявления переменных целого и символьного типов и их массивов. В операторах могут использоваться идентификаторы, символьные константы в стиле языка программирования C, заключенные в апострофы, целочисленные константы в десятичной и шестнадцатеричной системе, знаки присваивания ('=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: сравнения (<, >, =, !=, >=, <=); целочисленная и символьная арифметика (сложение, вычитания, умножение, деление, смена знака); логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 21. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных целого и вещественного типов и их массивов. В операторах могут использоваться идентификаторы, десятичные числа с плавающей запятой в обычном формате, целые числа в восьмеричной и десятичной системах, знаки присваивания (':=' и другие), элементы массивов знаки операций и круглые скобки.

Операции: численная арифметика ('+', '-', '*', '/', '%', модуль); сравнения (<, >, =, !=, >=, <=); логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 22. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, логические и арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных логического и целого типов и их массивов. В операторах могут использоваться идентификаторы, нечувствительные к регистру константы *t* (истина) и *f* (ложь), битовые строки (начинающиеся с обязательной пары знаков 0*b* или 0*B* последовательности из 0 и 1), целочисленные константы, знаки присваивания (':=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: побитовая арифметика (*or*, *xor*, *and*, *not*, стрелка Пирса (пара знаков -> или ключевое слово *peirce*)); сравнения (<, >, =, !=, >=, <=); числовая

арифметика (сложение, вычитание, умножение, деление, смена знака); логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 23. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных вещественного и целого типов и их массивов. В операторах могут использоваться идентификаторы, десятичные числа с плавающей запятой в нормализованном экспоненциальном формате, целые числа в восьмеричной и десятичной системах, знаки присваивания (':='), элементы массивов, знаки операций и круглые скобки.

Операции: сравнения (<, >, =, <>, >=, <=, =>, =<); числовая арифметика (сложение, вычитание, умножение, деление, смена знака); логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 24. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных вещественного и целого типов и их массивов. В операторах могут использоваться идентификаторы, десятичные числа с плавающей запятой в обычном и нормализованном экспоненциальном форматах, целочисленные константы, знаки присваивания (':=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: сравнения (<, >, =, <>, >=, <=, =>, =<); логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); числовая арифметика (сложение, вычитание, умножение, деление, смена знака); обращение к элементам массивов по индексу.

Вариант №25. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных целых (*decimal*, *roman*) и вещественного типов и их массивов. В операторах могут использоваться идентификаторы, целочисленные константы, римские числа (корректные последовательности заглавных и строчных литер *D*, *C*, *L*, *X*, *V* и *I*), десятичные числа с плавающей точкой, знаки присваивания (':=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: численная арифметика ('+', '-', '*', '/', '%'); сравнения (<, >, =, <>, >=, <=, =>, =<); логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 26. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и

постусловием, логические и арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных логического и целого типов и их массивов. В операторах могут использоваться идентификаторы, логические константы 0 и 1, целочисленные константы в десятичной и шестнадцатеричной системах, знаки присваивания (':=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: побитовая арифметика (*or*, *xor*, *and*, *not*, штрих Шеффера (знак вертикальной черты или ключевое слово *sheffer*)); сравнения (<, >, =, <>, >=, <=); численная арифметика ('+', '-', '*', '/', '%'); логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 27. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных целых (*octal*, *decimal*, *roman*) и вещественного типов и их массивов. В операторах могут использоваться идентификаторы, римские числа (корректные последовательности заглавных и строчных литер *D*, *C*, *L*, *X*, *V* и *I*), целочисленные константы в восьмеричной и десятичной системах, десятичные числа с плавающей точкой, знаки присваивания (':=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: сравнения (<, >, =, <>, >=, <=, =>, =<); численная арифметика ('+', '-', '*', '/', '%'); логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 28. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных целых (*octal*, *hexadecimal*, *roman*) и вещественного типов и их массивов. В операторах могут использоваться идентификаторы, римские числа (корректные последовательности заглавных и строчных литер *D*, *C*, *L*, *X*, *V* и *I*), целочисленные константы в восьмеричной и шестнадцатеричной системах, десятичные числа с плавающей точкой, знаки присваивания (':=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: сравнения (<, >, =, <>, >=, <=, =>, =<), численная арифметика ('+', '-', '*', '/', '%'); логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 29. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных целых типов (*binary*, *octal*, *decimal* и *hexadecimal*) и их массивов. В операторах могут использоваться идентификаторы, целочисленные константы в двоичной, восьмеричной,

десятичной и шестнадцатеричной системах, знаки присваивания (':=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: численная арифметика ('+', '-', '*', '/', '%', '^'); сравнения (<, >, =, <>, >=, <=); логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); обращение к элементам массивов по индексу.

Вариант 30. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, выражения, разделенные символом точки с запятой (';'), операторы объявления переменных целых (*decimal*, *hexadecimal*) и вещественного типов и их массивов. В операторах могут использоваться идентификаторы, целочисленные константы в шестнадцатеричной и десятичной системах счисления, десятичные числа с плавающей точкой, знаки присваивания (':=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: побитовая арифметика (*or*, *xor*, *and*, *not*, << (сдвиг влево), >> и (сдвиг вправо)); сравнения (<, >, ==, !=, >=, <=); логическая арифметика (И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ); численная арифметика ('+', '-', '*', '/', '%'); обращение к элементам массивов по индексу.

Вариант 31. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных целых (*decimal*, *hexadecimal*) и вещественного типов и их массивов. В операторах могут использоваться идентификаторы, целочисленные константы в десятичной и шестнадцатеричной системах счисления, десятичные числа с плавающей точкой, знаки присваивания (':=' и другие), элементы массивов, знаки операций и круглые скобки.

Операции: сравнения (<, >, =, ==, <>, !=, >=, <=); численная, логическая и побитовая (включая циклический сдвиг влево и вправо); обращение к элементам массивов по индексу.

Вариант 32. Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных целого (*binary*, *hexadecimal*) и вещественного типов и их массивов. В операторах могут использоваться идентификаторы, целочисленные константы в шестнадцатеричной системе, битовые строки (начинающиеся с обязательной пары знаков 0*b* или 0*B* последовательности из 0 и 1), десятичные числа с плавающей точкой, знаки присваивания(':=' и другие), инкремента (‘++’), декремента (‘--’), элементы массивов, знаки операций и круглые скобки.

Операции: сравнения (<, >, =, ==, <>, !=, >=, <=), численная, логическая и побитовая арифметика; обращение к элементам массивов по индексу.

Приложение А. Компилятор в трехадресный промежуточный код. Лексический анализатор реализован в виде входной спецификации *flex* (см. приложение Б). Обязательное условие запуск генератора *bison* с опцией *-d*, чтобы отделить определение лексем от реализации.

bison -d <имя файла для следующей спецификации>

```
/* Входная спецификация bison для компилятора в трехадресный промежуточный код
     простых арифметических выражений,
     условных операторов if с обязательной частью else,
     операторов присваивания, вывода значений выражений.
   Пустой оператор недопустим.
   Разделитель операторов - символ новой строки.
   Имя переменной - нечувствительная к регистру латинская литера.

   Распространяется под лицензией zlib,
     см. http://www.gzip.org/zlib/zlib_license.html

   Разработчик - Александр Кузнецов.
   Проект начат 15.10.2008, модифицирован 17.10.2008, 23.03.2013, 03.04.2013

   Обязательна генерация заголовочного файла синтаксического анализатора.
   Пример:
   $ bison -d language.y -o language.tab.c
*/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include "calc.h"     /* описание структуры синтаксического дерева */

int g_tmpvar_upper_index = 0;

/* размер временной строки не изменяется*/
#define TMP_STRING_MAX_SIZE 64

#define ADDITION_OPERATOR       1
#define SUBTRACTION_OPERATOR    2
#define MULTIPLICATION_OPERATOR 3
#define DIVISION_OPERATOR       4

#define NEGATION_OPERATOR       5
#define UNPLUS_OPERATOR         6

#define IF_FALSE_GOTO_OPERATOR  7
#define IF_TRUE_GOTO_OPERATOR   8
#define GOTO_OPERATOR           9
#define SET_LABEL_OPERATOR     10

#define OUTPUT_OPERATOR        11

#define ASSIGN_OPERATOR        12

extern int lineno;

/* размер таблицы символов (только uppercase-литеры) */
#define TABLE_SIZE 26
```

```
/* описание таблицы символов,
   в текущей реализации не используется поле dval
*/
typedef struct
{
    double dval;        // вещественное значение, зарезервировано на будущее
    int   is_defined;   // объявлена ли переменная
} TVariableTableRecord;

TVariableTableRecord table[TABLE_SIZE];


/* ПРОТОТИПЫ ФУНКЦИЙ: */

/* Лексический анализатор */
extern int yylex ();

/* Обработка синтаксического дерева */
void freenode (nodeType* p);
nodeType* constants(double value);
nodeType* idents (int index);
nodeType* tmpvars (int tmp_index);

/* Генерация трехадресного кода */
int codegenBinary(FILE* outputFile, int operatorCode,
            nodeType* leftOperand, nodeType* rightOperand, nodeType* result
           );
int codegenUnary(FILE* outputFile, int operatorCode,
            nodeType* operand, nodeType* result
           );
int codegenGoto(FILE* outputFile, int operatorCode,
            int labelNumber, nodeType* optionalExpression
           );
int codegenLabel(FILE* outputFile, int labelNumber);


/* Вывод сообщений об ошибках */
int my_yyerror(char* error_message);
int yyerror(char* errormessage);

int g_ErrorStatus = 0; /* Состояние ошибки при анализе либо кодогенерации */


FILE* outfile;           /* Внешний выходной файл */
char g_outFileName[256]; /* Имя выходного файла */

/* Обработка таблицы меток. Используется стековая организация */
static int g_LastLabelNumber = 0;
static int g_LabelStackPointer = 0;

static int Labels[256];
static void PushLabelNumber(int);
static int  PopLabelNumber(void);
static void EmptyLabels(void);
%}

%union
{
    double    dval;            // числовое значение лексемы
    int       index_in_table;  // индекс в таблице символов
```

```
    char      other;           // другой символ
    nodeType* nptr;            // узел или поддерево синтаксического дерева
}

%token <index_in_table>VARIABLE
%token <dval>NUMBER
%token <other>EOFILE IF_KEYWORD ELSE_KEYWORD

%left '+' '-'
%left '*' '/'
%right UMINUS UPLUS

%type <nptr>expr expr_in_pars
%type <other>error

%start program

%%

program : lines;

lines : stmt '\n'
      | stmt EOFILE  { YYACCEPT; }
      | lines stmt '\n'
      | lines stmt EOFILE { YYACCEPT; }
      | EOFILE { YYACCEPT; }
      ;

stmt :  VARIABLE '=' expr
                    {
                        table[$1].is_defined = !0;
                        codegenUnary(outfile, ASSIGN_OPERATOR, $3, idents($1));

                        g_tmpvar_upper_index = 0;
                        freenode($3);
                    }
     | expr     {   // в случае ошибки в выражении возможен
                        // вывод некорректного значения
                        codegenUnary(outfile, OUTPUT_OPERATOR, $1, NULL);

                        g_tmpvar_upper_index = 0;
                        freenode ($1);
                    }
     | conditional_statement
     | error        {
                        // В этом случае считаем, что пришедшая лексема
                        // здесь совершенно не уместна.
                        // Формируем сообщение об ошибке и выводим на экран
                        char tmp[TMP_STRING_MAX_SIZE] = "\0";
                        int ch;
                        strcat(tmp,"Unexpected token ");
                        if (isgraph($1))
                        {
                            tmp[strlen(tmp)] = $1;
                        }
                        else
                        {
                            char tmp1[32] = "\0";
                            strcat(tmp, "with code ");
                            itoa((int)$1, tmp1, 10);
                            strcat(tmp, tmp1);
                        }
```

```
                        strcat(tmp," in expression");
                        my_yyerror(tmp);

                        yyclearin; /* discard lookahead */
                        yyerrok;
                    }
    ;

conditional_statement :
       conditional_begin true_branch false_branch
    ;

conditional_begin :
       IF_KEYWORD '(' expr ')'
       {
         codegenGoto(outfile, IF_FALSE_GOTO_OPERATOR
                     , g_LastLabelNumber, $3
                    );
         PushLabelNumber(g_LastLabelNumber);
         ++g_LastLabelNumber;
       }
    |
       IF_KEYWORD '(' expr error
       {
         my_yyerror("Unbalanced parentheses");
       }
    ;

true_branch :
       stmt
       {

         codegenGoto(outfile, GOTO_OPERATOR
                     , g_LastLabelNumber, NULL

                    );
         codegenLabel(outfile, PopLabelNumber());
         PushLabelNumber(g_LastLabelNumber);
         ++g_LastLabelNumber;
       }
    ;

false_branch :
       ELSE_KEYWORD stmt
       {
         codegenLabel(outfile, PopLabelNumber());
       }
    |
      error
       {
         my_yyerror("Else keyword expected");
       }
    ;

expr : expr '+' expr
                    {
                        $$ = tmpvars(g_tmpvar_upper_index);
                        $$->left = $1;
                        $$->right = $3;

                        ++g_tmpvar_upper_index;
```

```
                        codegenBinary(outfile, ADDITION_OPERATOR, $1, $3, $$);
                    }
    | expr '-' expr
                    {
                        $$ = tmpvars(g_tmpvar_upper_index);
                        $$->left = $1;
                        $$->right = $3;
                        ++g_tmpvar_upper_index;
                        codegenBinary(outfile,  SUBTRACTION_OPERATOR,  $1,  $3,
$$);
                    }
    | expr '*' expr
                    {
                        $$ = tmpvars(g_tmpvar_upper_index);
                        $$->left = $1;
                        $$->right = $3;

                        ++g_tmpvar_upper_index;

                        codegenBinary(outfile, MULTIPLICATION_OPERATOR, $1, $3,
$$);
                    }
    | expr '/' expr
                    {
                        $$ = tmpvars(g_tmpvar_upper_index);
                        $$->left = $1;
                        $$->right = $3;

                        ++g_tmpvar_upper_index;

                        codegenBinary(outfile, DIVISION_OPERATOR, $1, $3, $$);
                    }
    | '-' expr %prec UMINUS
                    {
                        $$ = tmpvars(g_tmpvar_upper_index);
                        $$->right = $2;
                        ++g_tmpvar_upper_index;

                        codegenUnary(outfile, NEGATION_OPERATOR, $2, $$);

                    }
    | '+' expr %prec UPLUS
                    {
                        $$ = tmpvars (g_tmpvar_upper_index);
                        $$->right = $2;
                        ++g_tmpvar_upper_index;

                        codegenUnary(outfile, UNPLUS_OPERATOR, $2, $$);
                    }
    | expr_in_pars
                    {
                        $$ = $1;
                        $$->place = $1->place;
                        $$->type = $1->type;
                        $$->left = $1->left;
                        $$->right = $1->right;
                    }
    | VARIABLE
                    {
                        // если нет такой переменной в таблице
                        // выводим сообщение об ошибке
                        if (table[$1].is_defined == 0)
```

```
                        {
                            // формируем сообщение об ошибке и выводим на экран
                            char tmp[TMP_STRING_MAX_SIZE] = "\0";
                            strcat(tmp,"Undefined variable ");
                            tmp[strlen(tmp)] = $1 + 'A';
                            my_yyerror(tmp);

                            // восстановление после ошибки
                            yyerrok;
                            yyclearin;

                            // поскольку переменная не объявлена,
                            // то считаем, что этот элемент выражения равен 0
                            $$ = constants(0.0);
                        }
                        else
                        {
                            // если такая переменная есть в таблице,
                            // выводим сообщение об ошибке
                            // то считаем, что этот элемент выражения равен
                            // значению, взятому из таблицы
                            $$ = idents($1);
                        }
                    }
    | NUMBER        {
                        $$ = constants($1);
                    }
    ;

expr_in_pars : '(' expr ')'
                    {
                        $$ = $2;
                        $$->place = $2->place;
                        $$->type = $2->type;
                        $$->left = $2->left;
                        $$->right = $2->right;
                    }
             | '(' expr error
                    {
                        // если не закрыта круглая скобка, выводим сообщение,
                        // но не читаем до синхронизирующей лексемы,
                        // так как дальше может быть все в порядке
                        my_yyerror("Right parentheses expected.\n");
                        $$ = $2;
                        $$->place = $2->place;
                        $$->left = $2->left;
                        $$->right = $2->right;
                    }
             ;
%%
// функция вывода сообщения об ошибке.
// Входной параметр - error_message
int my_yyerror(char* error_message)
{
    fprintf(stderr, "Line %d: %s.\n", lineno, error_message);
    g_ErrorStatus = !0;
    return !0;
}

// Yacc-функция вывода сообщения об ошибке.
// Входной параметр - error_message
int yyerror(char* errormessage)
{
```

```
    // Подавляем встроенное сообщение об ошибке
    // Вместо нее используется my_yyerror (см.выше)
    return !0;
}

nodeType* constants(double value)
{
    nodeType* p;
    size_t    nodeSize;

    /* выделить память для узла */
    nodeSize = sizeof(nodeType);
    if (NULL == (p = (nodeType*)malloc(nodeSize)))
       my_yyerror("out of memory");

    /* установить значения полей */
    p->type = typeConst;
    p->place = 0;       // константа размещается в самом узле
    p->constant.value = value;

    p->right = p->left = NULL;

    return p;
}

nodeType* idents(int index)
{
    nodeType* p;
    size_t    nodeSize;

    /* выделить память для узла */
    nodeSize = sizeof(nodeType);
    if (NULL == (p = (nodeType*)malloc(nodeSize)))
       my_yyerror("out of memory");

    /* установить значения полей */
    p->type = typeIdentifier;
    p->place = index;          // ссылка на таблицу идентификаторов
    p->right = p->left = NULL;

    return p;
}

nodeType* tmpvars (int tmp_index)
{
    nodeType* p;
    size_t    nodeSize;

    /* выделить память для узла */
    nodeSize = sizeof(nodeType);
    if (NULL == (p = (nodeType*)malloc(nodeSize)))
       my_yyerror("out of memory");

    /* установить значения полей */
    p->type = typeTmpvar;
    p->place = tmp_index;   // сохранение до последующего использования
    p->right = p->left = NULL;

    return p;
}
```

```
// освобождение занятой пямяти
void freenode(nodeType* p)
{
    if (!p) return;
    freenode(p->left);
    freenode(p->right);
    free(p);
    return;
}


int codegenBinary(FILE* outputFile, int operatorCode,
            nodeType* leftOperand, nodeType* rightOperand, nodeType* result
           )
{
    fprintf(outputFile, "\t$t%u\t:=\t", result->place);
    switch (leftOperand->type)
    {
    case typeIdentifier:
        fprintf(outputFile, "%c", leftOperand->place + 'A');
        break;
    case typeTmpvar:
        fprintf(outputFile, "$t%d", leftOperand->place);
        break;
    case typeConst:
        fprintf(outputFile, "%g", leftOperand->constant.value);
        break;
    }

    switch (operatorCode)
    {
    case ADDITION_OPERATOR:
        fprintf(outputFile, "+");
        break;
    case SUBTRACTION_OPERATOR:
        fprintf(outputFile, "-");
        break;
    case MULTIPLICATION_OPERATOR:
        fprintf(outputFile, "*");
        break;
    case DIVISION_OPERATOR:
        fprintf(outputFile, "/");
        break;
    }

    switch (rightOperand->type)
    {
    case typeIdentifier:
        fprintf(outputFile, "%c", rightOperand->place + 'A');
        break;
    case typeTmpvar:
        fprintf(outputFile, "$t%d", rightOperand->place);
        break;
    case typeConst:
        fprintf(outputFile, "%g", rightOperand->constant.value);
        break;
    }

    fprintf(outputFile, "\n");
}

int codegenUnary(FILE* outputFile, int operatorCode,
            nodeType* operand, nodeType* result
```

```
            )
{
    if (operatorCode == OUTPUT_OPERATOR)
    {
        fprintf (outputFile, "\toutput\t");
    }

    else if (operatorCode == ASSIGN_OPERATOR)
    {
        fprintf(outputFile, "\t%c\t:=\t", result->place + 'A');
    }
    else
    {
        fprintf(outputFile, "\t$t%u\t:=\t", result->place);
        switch (operatorCode)
        {
        case UNPLUS_OPERATOR:
            fprintf(outputFile, "+");
            break;
        case NEGATION_OPERATOR:
            fprintf(outputFile, "-");
        }
    }

    switch (operand->type)
    {
    case typeIdentifier:
        fprintf(outputFile, "%c", operand->place + 'A');
        break;
    case typeTmpvar:
        fprintf(outputFile, "$t%d", operand->place);
        break;
    case typeConst:
        fprintf(outputFile, "%g", operand->constant.value);
        break;
    }
    fprintf(outputFile, "\n");
}

int codegenGoto(FILE* outputFile, int operatorCode,
            int labelNumber, nodeType* optionalExpression
           )
{
    if (operatorCode != GOTO_OPERATOR)
    {
        if(operatorCode == IF_FALSE_GOTO_OPERATOR)
            fprintf(outputFile, "\tiffalse\t");
        else if(operatorCode == IF_TRUE_GOTO_OPERATOR)
            fprintf(outputFile, "\tiftrue\t");

        switch (optionalExpression->type)
        {
        case typeIdentifier:
            fprintf(outputFile, "%c", optionalExpression->place + 'A');
            break;
        case typeTmpvar:
            fprintf(outputFile, "$t%d", optionalExpression->place);
            break;
        case typeConst:
            fprintf(outputFile, "%g", optionalExpression->constant.value);
            break;
        }
    }
```

```
    fprintf(outputFile, "\tgoto\t$L%d", labelNumber);
    fprintf(outputFile, "\n");

}

int codegenLabel(FILE* outputFile, int labelNumber)
{
    fprintf (outputFile, "$L%d:", labelNumber);

}


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;
   }
}

static void EmptyLabels(void)
{
   g_LabelStackPointer = 0;
}

int main (int argc, char* argv[])
{
    int yyparse();

    // инициализируем таблицу идентификаторов
    int i;
    for (i = 0; i < TABLE_SIZE; ++i)
    {
        table[i].is_defined = 0;
    }

    if (argc < 3)
    {
        printf("Too few paremeters.\n");
        system("PAUSE");  // not for *NIXes
        return EXIT_FAILURE;
    }
    if (NULL == freopen (argv[1], "r", stdin))
    {
        printf("Cannot open input file %s.\n", argv[1]);
        system("PAUSE");   // not for *NIXes
        return EXIT_FAILURE;
    }
    outfile = fopen(argv[2], "w");
```

```
    if (NULL == outfile)
    {
        printf("Cannot open output file %s.\n", argv[2]);
        system("PAUSE");   // not for *NIXes
        return EXIT_FAILURE;
    }

    strcpy(g_outFileName, argv[2]);

    yyparse();

    fclose(outfile);
    if (0 != g_ErrorStatus)
    {
        printf("Target code isn't generated.\n");
        unlink(g_outFileName);
    }
}
```

Приложение Б. Входная спецификация *flex* для компилятора в трехадресный код (см. приложение А)

```
%{
/* Входная спецификация flex лексического анализатора для компилятора
   в трехадресный промежуточный код простых арифметических выражений,
   условных операторов if с обязательной частью else,
   операторов присваивания, вывода значений выражений.
   Разделитель операторов - символ новой строки.
   Имя переменной - нечувствительная к регистру латинская литера.

   Разработчик - Александр Кузнецов.
   Проект начат 15.10.2008, модифицирован 17.10.2008, 23.03.2013, 03.04.2013
   Распространяется под лицензией zlib,
     см. http://www.gzip.org/zlib/zlib_license.html

   Для корректной компиляции исходного кода лексического анализатора
     необходима генерация заголовочного файла синтаксического анализатора,
     например:
   $ bison -d language.y -o language.tab.c

     с последующим его включением соответствующими директивами.
   Пример (см. также далее по тексту):
   #include "language.tab.h"

*/

#include <stdio.h>
#include <ctype.h>
#include "calc.h"
#include "language.tab.h"

#if defined _WIN32
#include <io.h>              // Для isatty
#elif defined _WIN64
#include <io.h>              // Для isatty
#endif

#ifdef MSVC
#define  isatty _isatty       // В VC isatty назван _isatty
#endif

int yylex();

int lineno = 1;
%}

%option nounistd nodefault

first_part   [0-9]+(\.[0-9]*)?
second_part  [0-9]*\.[0-9]+
%%
if                  {
                        return IF_KEYWORD;
                    }
else                {
                        return ELSE_KEYWORD;
                    }
[a-zA-Z]            {
                        yylval.index_in_table = tolower (yytext[0]) - 'a';
                        return VARIABLE;
```

```
                        }
{first_part}|{second_part}        {
                            yylval.dval = atof (yytext);
                            return NUMBER;
                        }
[ \t]+                  ;
<<EOF>>                 {
                            yylval.other = yytext[0];
                            return EOFILE;
                        }
\n                      {
                            ++lineno;
                            return yylval.other = yytext[0];
                        }
.                       {
                            return yylval.other = yytext[0];
                        }
%%
int yywrap ()
{
    return 1;
}
```

Приложение В. Заголовочный файл *calc.h* для компилятора арифметических выражений в трехадресный промежуточный код (см. приложение А)

```
/* Описание структур данных синтаксического дерева
     для компилятора в трехадресный промежуточный код
     простых арифметических выражений,
     условных операторов if с обязательной частью else,
     операторов присваивания, вывода значений выражений.
   Пустой оператор недопустим.
   Разделитель операторов - символ новой строки.
   Имя переменной - нечувствительная к регистру латинская литера.

   Распространяется под лицензией zlib,
     см. http://www.gzip.org/zlib/zlib_license.html

   Разработчик - Александр Кузнецов.
   Проект начат 15.10.2008, модифицирован 17.10.2008, 23.03.2013, 03.04.2013

   Обязательна включение этого заголовочного файла
     в исходный текст синтаксического анализатора.
   Пример:
   #include "calc.h"
*/


// тип узла синтаксического дерева
typedef enum
{
    typeConst,          // константа
    typeIdentifier,     // идентификатор
    typeTmpvar          // временная переменная
} nodeEnum;

// константы
typedef struct
{
    double value;       // значение константы
} constNodeType;

typedef struct nodeTypeTag
{
    struct nodeTypeTag* left;
    struct nodeTypeTag* right;
    nodeEnum     type;   // тип узла дерева

    // размещение результата для идентификатора или временной переменной
    unsigned int place;

    // следующее объединение должно идти последней в структуре
    // из-за возможных проблем с выравниванием разделов памяти
    union               // вычисляемое значение атрибута узла
    {
        constNodeType constant;  // константа
        unsigned int  id;        // идентификатор
        unsigned int  number;    // номер временной переменной
    };

} nodeType;
```