Лабораторная работа №3. Разработка блока семантического анализа транслятора.

Цель: изучение основных методов организации таблиц идентификаторов в трансляторах языков программирования и методов семантического анализа с их программной реализацией.

Задачи:

1. Изучение теоретического материала об основных методах организации таблиц идентификаторов.
2. Изучение теоретического материала по организации семантического анализа языков программирования.
3. Составление формального описания программы семантического анализа.
4. Программная реализация по формальному описанию.

Ход работы:

1. Получить у преподавателя индивидуальный вариант задания, предусматривающий разработку программы семантического анализа простого языка программирования.
2. Составить формальное описание программы семантического анализа с использованием математического аппарата формальных контекстно-свободных грамматик с включением семантических процедур.
3. Произвести автоматическую генерацию исходного текста программы семантического анализа с использованием системы *bison*. Отладить полученную программу. Используемая в качестве основы грамматика не должна приводить к конфликтам и неоднозначностям. При наличии в анализируемой программе лексически, синтаксических и семантических ошибок необходимо выводить на экран соответствующие сообщения.

   При разработке программы рекомендуется отделить анализатор от вспомогательных модулей, в том числе от лексического анализатора и функций для работы с таблицами.
4. Написать отчет.

Внимание: в приложениях приведены исходные коды для вывода на экран абстрактных синтаксических деревьев разбора строк на простом языке программирования.

В нем поддерживаются конструкции для арифметических выражений, отделенных друг от друга соответствующим символом, составные операторы, замкнутые на фигурные скобки, условные операторы *if* с необязательной частью *else*. Элементами выражений являются целые и вещественные константы, а также имена переменных из одного символа, арифметические операции и операции сравнения.

Структура проекта для сборки представлена в *makefile*.

Альтернатива — http://en.wikipedia.org/wiki/GNU_bison

Ход защиты:

1. Продемонстрировать преподавателю корректную работу программы семантического анализа.

2. Пояснить работу изученных механизмов по полученному формальному описанию программы семантического анализа.

Содержание отчета:

1. Цель работы с постановкой задачи.
2. Полученная спецификация *bison*.
3. Тестовые примеры работы программы (не менее четырех).

Во время защиты лабораторной работы необходимо иметь при себе исполняемый модуль программы, исходные тексты, а также твердую копию отчета.

Краткие сведения о методах организации таблиц идентификаторов

В процессе работы транслятор хранит информацию об объектах программы (идентификаторах) в специальных таблицах символов. Как правило, информация о каждом объекте состоит из двух основных элементов: имени объекта и описания объекта. Информация об объектах программы должна быть организована таким образом, чтобы поиск ее был по возможности быстрее, а требуемая память по возможности меньше. Двумя основными операциями, выполняемыми над этими таблицами, являются добавление элементов и их поиск. При этом первая из них (добавление записи в таблицу) выполняется значительно реже второй, поскольку описания новых идентификаторов в программе встречаются гораздо реже, чем они используются.

Кроме того, со стороны языка программирования могут быть дополнительные требования к организации информации. Имена могут иметь определенную область видимости. Например, поле записи должно быть уникально в пределах структуры или уровня структуры, но может совпадать с именем объекта вне записи или другого уровня записи. В то же время имя поля может открываться оператором присоединения, и тогда может возникнуть то, что называется конфликтом имен (или, по-другому, неоднозначностью в трактовке имени). Если язык имеет блочную структуру, как, например, язык С или Паскаль, то необходимо обеспечить такой способ хранения информации, чтобы, во-первых, поддерживать блочный механизм видимости, а во-вторых – эффективно освобождать память при выходе из блока. В некоторых языках одновременно, т.е. в одном блоке могут быть видимы несколько объектов с одним именем, в других такая ситуация недопустима.

Организация таблицы в виде линейного списка

В простейшем случае таблица идентификаторов представляется собой линейный неупорядоченный список или массив, каждая ячейка которого содержит описание соответствующего элемента таблицы. Добавление элементов осуществляется в конец списка или очередную ячейку массива. Поиск в такой таблице идентификаторов выполняется путем последовательного

перебора всех ее элементов, пока нужный не будет найден. Этот поиск можно ускорить, если элементы таблицы отсортированы некоторым, заранее известным образом, например, по названию идентификатора. Эффективным алгоритмом поиска в упорядоченном списке является метод двоичного поиска.

Алгоритм достаточно прост и заключается в следующем (размер таблицы равен N): искомый элемент сравнивается с элементом в середине таблицы (N+1)/2. Если этот элемент не является искомым, то далее просматривается блок от 1 до (N+1)/2-1, или блок элементов от (N+1)/2+1 в зависимости от результата сравнения искомого элемента и элемента в середине списка. Процесс поиска и двукратного уменьшения размера блока повторяется до тех пор, пока не будет найден элемент либо не будет доказано, что такого элемента нет в таблице.

Недостаток данного алгоритма состоит в соблюдении требовании упорядоченности массива, т.е. при добавлении элемента сначала необходимо выполнить поиск подходящего места в списке либо сортировать массив при выполнении каждой операции по добавлению записи в таблицу.

Организация таблицы в виде двоичного дерева

Уменьшить время поиска элемента в таблице идентификаторов без существенного увеличения затрат на добавление элемента можно за счет отказа от использования списков и массивов и построения таблиц на основе двоичных упорядоченных деревьев. При этом каждый элемент таблицы представляет собой узел дерева, корневым узлом будет первый идентификатор встреченный транслятором при заполнении таблицы.

Алгоритм построения двоичного дерева немногим сложнее описанного выше алгоритма работы со списком. При поступлении очередного идентификатора в качестве текущего узла дерева выбирается его корень. Затем очередной идентификатор сравнивается с элементом в текущем узле. Если эти элементы равны, то поиск прекращается. Если очередной идентификатор меньше текущего узла, то происходит переход в левое поддерево, в противном случае – в правое. Добавление элемента совмещается с операцией поиска, т.е. если не удается перейти в левое поддерево (текущий элемент является листом), то создается новый узел, в который помещается информация об очередном идентификаторе. Аналогично – для правого поддерева.

Таблицы расстановки

Одним из эффективных способов организации таблицы символов является таблица расстановки (или хеш-таблица). Поиск в такой таблице может быть организован методом повторной расстановки. Суть его заключается в следующем.

Таблица символов представляет собой массив фиксированного размера N. Идентификаторы могут храниться как в самой таблице символов, так и в отдельной таблице идентификаторов.

Определим некоторую функцию h_1 (первичную функцию расстановки), определенную на множестве идентификаторов и принимающую значения от 0

до N-1 (т.е. $0 \le h_1(id) \le N$-1, где id – символьное представление идентификатора). Таким образом, функция расстановки сопоставляет идентификатору некоторый адрес в таблице символов.

Пусть мы хотим найти в таблице идентификатор id. Если элемент таблицы с номером $h_1(id)$ не заполнен, то это означает, что идентификатора в таблице нет. Если же занят, то это еще не означает, что идентификатор id в таблицу занесен, поскольку несколько идентификаторов могут иметь одно и то же значение функции расстановки. Для того чтобы определить, нашли ли мы нужный идентификатор, сравниваем id с элементом таблицы $h_1(id)$. Если они равны – идентификатор найден, если нет – надо продолжать поиск дальше.

Для этого вычисляется вторичная функция расстановки $h_2(h)$, значением которой опять таки является некоторый адрес в таблице символов. Возможны четыре варианта:

- элемент таблицы не заполнен (т.е. идентификатора в таблице нет);
- идентификатор элемента таблицы совпадает с искомым (т.е. идентификатор найден);
- адрес элемента совпадает с уже просмотренным (т.е. таблица вся просмотрена и идентификатора нет);
- предыдущие варианты не выполняются, так что необходимо продолжать поиск.

Для продолжения поиска применяется следующая функция расстановки $h_3(h_2)$, $h_4(h_3)$ и т.д. Как правило, $h_i = h_2$ для $i \ge 2$. Аргументом функции h_2 является целое в диапазоне [0, N-1], и она может быть устроена по-разному. Приведем три варианта:

1) $h_2(i) = (i + 1)\ \%\ N$.

Берется следующий (циклически) элемент массива. Этот вариант плох тем, что занятые элементы «группируются», образуют последовательные занятые участки, и в пределах этого участка поиск становится, по существу, линейным.

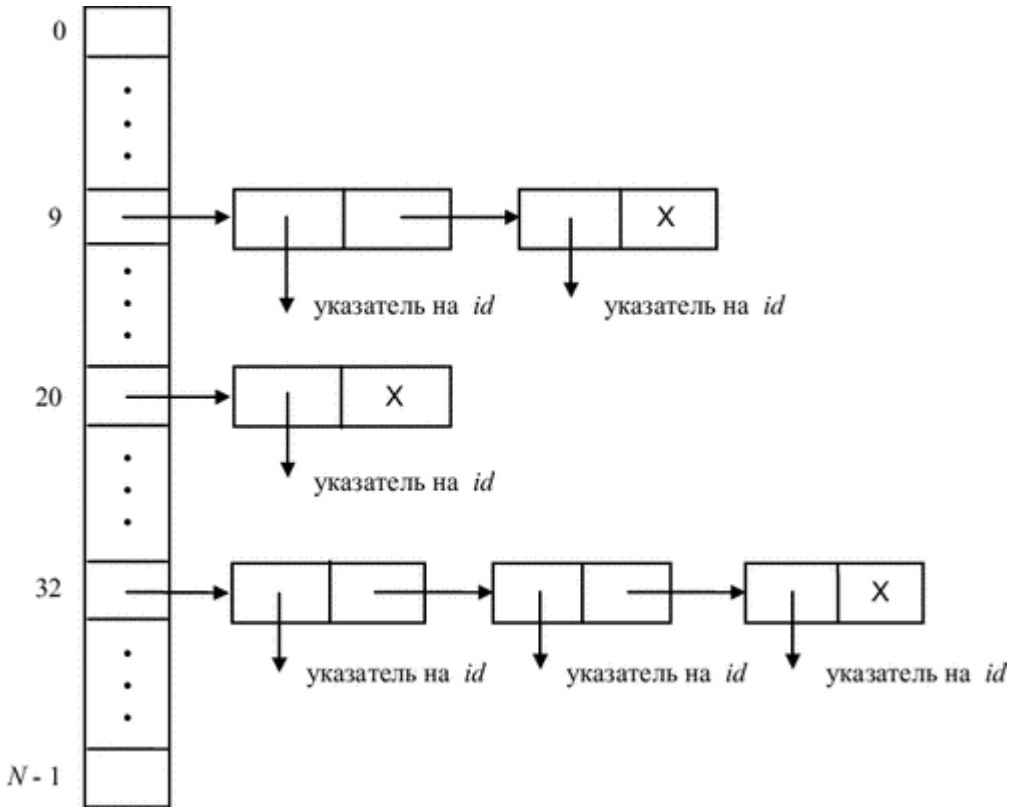
2) $h_2(i) = (i + k)\ \%\ N$, где k и N взаимно просты.

По существу, это предыдущий вариант, но элементы накапливаются не в последовательных элементах, а «разносятся».

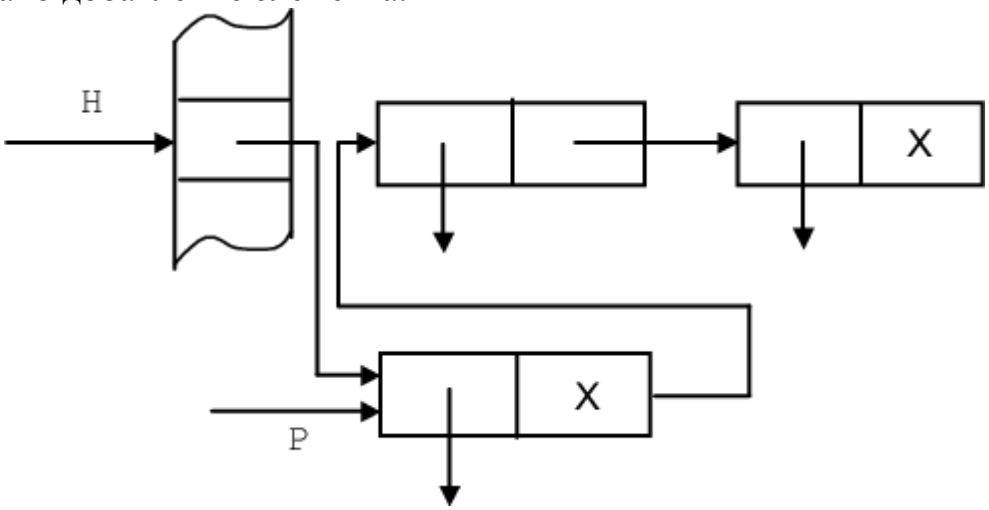
3) $h_2(i) = (a * i + c)\ \%\ N$ – «псевдослучайная последовательность».

Здесь c и N должны быть взаимно просты, $b = a$-1 кратно p для любого простого p, являющегося делителем N, b кратно 4, если N кратно 4.

Только что описанная схема страдает одним недостатком – возможностью переполнения таблицы. Рассмотрим ее модификацию, когда все элементы, имеющие одинаковое значения первичной функции расстановки, связываются в список (при этом отпадает необходимость использования функций h_i для $i \ge 2$). Таблица расстановки со списками – это массив указателей на списки элементов.



Вначале таблица расстановки пуста (все элементы имеют значение NULL). При поиске идентификатора *Id* вычисляется функция расстановки *h*(*Id*), и просматривается соответствующий линейный список. На следующем рисунке показано добавление элемента.



Много внимания исследователями было уделено тому, какой должна быть (первичная) функция расстановки. Основные требования к ней очевидны: она должна легко вычисляться и распределять равномерно. Один из возможных подходов здесь заключается в следующем.

1. По символам строки *s* определяем положительное целое *H*. Преобразование одиночных символов в целые обычно можно сделать средствами языка реализации. В Паскале для этого служит функция *ord*, в Си при выполнении арифметических операций символьные значения трактуются как целые.

2. Преобразуем H, вычисленное выше, в номер элемента, т.е. целое между 0 и N-1, где N – размер таблицы расстановки, например, взятием остатка при делении H на N.

Функции расстановки, учитывающие все символы строки, распределяют лучше, чем функции, учитывающие только несколько символов, например, в конце или середине строки. Но такие функции требуют больше вычислений.

Простейший способ вычисления H – сложение кодов символов. Перед сложением с очередным символом можно умножить старое значение H на константу q. Т.е. полагаем $H_0 = 0$, $H_i = q*H_i\text{-}1 + c_i$ для $1 \le i \le k$, k – длина строки. При $q = 1$ получаем простое сложение символов. Вместо сложения можно выполнять сложение c_i и $q*H_i$-1 по модулю 2. Переполнение при выполнении арифметических операций можно игнорировать.

Еще в одном способе функция вычисляется, начиная с $H = 0$ (предполагается, что используются 32-битовые целые числа). Для каждого символа c сдвигаем биты H на 4 позиции влево и добавляем очередной символ. Если какой-нибудь из четырех старших бит H равен 1, сдвигаем эти 4 бита на 24 разряда вправо, затем складываем по модулю 2 с H и устанавливаем в 0 каждый из четырех старших бит, равных 1.

Естественно, класс функций хеширования гораздо шире. Описание функций можно найти в соответствующей специальной литературе. Ими можно пользоваться при выполнении данной практической работы.

Краткие теоретические сведения о семантическом анализе языков программирования

Некоторые характеристики языков программирования не являются контекстно-свободными, то есть их нельзя определить с помощью КСГ. Трансляторы, построенные на основе контекстно-зависимых грамматик, возможны, однако, они очень требовательны к вычислительным ресурсам. В связи с этим трансляторы выполняют анализ программ в два этапа: сначала синтаксический анализ (далее – СА) на основе какого-либо класса КСГ, а затем семантический анализ. Входными данными для семантического анализа являются:

- результаты синтаксического разбора конструкций входного языка (например, синтаксические деревья);
- таблицы транслятора.

Семантический анализатор выполняет следующие действия:

- проверка соблюдения в программе семантических соглашений языка;
- дополнение синтаксического дерева действиями, неявно предусмотренными семантикой языка;
- проверка семантических норм, напрямую не связанных с языком.

Каждый язык имеет четко заданные семантические соглашения, которые не могут проверяться на фазе СА, их проверяет семантический анализатор. Примерами семантических соглашений являются:

- каждый идентификатор должен описываться один и только раз с учетом блочной структуры программы;
- все операнды в операциях должны иметь допустимые для данной операции типы;
- каждая метка, на которую делается ссылка, должна присутствовать в программе один и только один раз;
- число и типы фактических параметров функций должны быть согласованы с числом и типами фактических параметров.

Разумеется, конкретный перечень такого рода соглашений жестко связан с семантикой языка. Так, например, в языке Fortran можно не описывать некоторые идентификаторы. Некоторые операторы могут быть вполне корректны с синтаксической точки зрения, при этом не отвечая семантическим соглашениям. Например, в большинстве языков нельзя складывать числа со строками. Если какое-либо из семантических требований не выполняется, то транслятор выдает сообщение об ошибке и возможно прекратить процесс трансляции.

В то же время в большинстве языков программирования допускается выполнение операций присваивания или сложения между целыми и вещественными числами (хотя это, как правило, переменные разных типов). Однако, на целевой вычислительной системе, целочисленное и вещественное присваивание и сложения, выполняются разными машинными инструкциями. Как следствие, необходимо дополнить получившуюся синтаксическую конструкцию операциями преобразования типов. Для вычисления адресов при обращении к элементам сложных структур данных транслятор также должен неявно добавить несколько операций.

К большинству языков программирования могут быть применимы следующие семантические нормы:

- каждая переменная должна использоваться в программе, по крайней мере, один раз;
- результат функции определен на каждом этапе ее выполнения;
- операторы цикла должны предусматривать свое завершение;
- условные операторы должны предусматривать выполнения всех своих ветвей.

Конкретный состав семантических норм определяется семантикой языка. В принципе транслятор может и не проверять программу на соответствие этим нормам, то есть их несоблюдение не может рассматриваться как ошибка, а значит, нет необходимости останавливать процесс трансляции. Однако многие трансляторы в подобных случаях, сообщают пользователям о несоблюдении норм в виде так называемых «предупреждений».

В процессе трансляции необходимыми являются несколько таблиц:

- таблица идентификаторов;
- таблица типов;
- таблица меток;
- таблица функций.

Основная задача таблицы идентификаторов – установка соответствия между идентификатором и его типом. Организация работы с этой таблицей зависит от языка программирования и эффективности трансляции. Имя идентификатора и его тип помещаются в таблицу символов в соответствии с объявлением (на языке С *double y;*). Поиск в таблице осуществляется в соответствии с применением идентификаторы (на С – *y = 2.5;*).

Естественно, транслятор должен учитывать, что, например, указанный выше идентификатор *y* не обязан встречаться в программе единственный раз. Достаточно вспомнить, что в языках Паскаль или С внутри каждой функции или блока можно использовать одноименные переменные, если только речь идет о разных блоках программ. Чтобы их отличать, компиляторы перед помещением переменной или константы в таблицу обычно дополняют их именами блоков или функций. Похожие действия выполняются с самими функциями, если они принадлежат разным объектам. Кроме того, они (имена) изменяются в зависимости от типов аргументов.

Таблицы типов предназначены для обеспечения уникального представления каждого типа, используемого в конкретной программе. При этом необходимо принять во внимание операции, выполняемые над типами, и что многие типы высокоструктурированы и задаются рекурсивно. Обычно простые типы представляются целыми числами, а сложные – структурами. В таблице типов отображается имя на указатель на структуры конкретных типов. Таблицы функций похожи на таблицы типов, с функцией соотносятся типы аргументов и результатов. В таблице функций отображается имя на адрес конкретной функции.

Многие управляющие структуры языков программирования реализуются в терминах меток и переходов. Таким образом, таблицы меток, в которых также устанавливается соответствие между именем и адресом, будут содержать пользовательские метки и метки, добавленные транслятором.

И в случае меток, и при работе с функциями транслятор должен учитывать вложенность (области видимости), что неизбежно приведет к преобразованиям имен, подобным упомянутым выше. Нередко в трансляторах все (или почти все) указанные таблицы объединяются в одну, но это, конечно, зависит как от реализуемого языка, так и от разработчика транслятора.

В данной работе не требуется поддерживать таблицы меток, функций или типов, достаточно работы с таблицей переменных (идентификаторов). В частности необходимо добавить поле типа переменной в структуру, ее описывающую и функции, обслуживающие таблицы символов. Кроме того, в семантических процедурах для продукций, определенных во входной спецификации и реализующих некоторые операции с переменными и/или

константами, потребуется добавить код для проверки типов операндов с соответствующей диагностикой ошибок.

Для выдачи сообщений об ошибках, как правило, достаточно определить функцию *yyerror()*, после которой по умолчанию происходит завершение работы программы синтаксического анализа.

Если есть необходимость реализовать то, что выше было названо предупреждениями, то после вызова функции *yyerror()*, нужно осуществить вызов стандартного макроса *yyerrok*, не имеющего входных параметров.

Использованы следующие источники:

1. Lex & Yacc Tutorial - http://epaperpress.com/lexandyacc/index.html
2. Bison. Генератор синтаксических анализаторов, совместимый с YACC - http://www.linux.org.ru/books/GNU/bison/bison_toc.html
3. Levine, J. flex & bison / J. Levine. – Sebastopol, CA: O'Reilly Media, 2009. – 292 p.
4. Levine, J. lex & yacc, Second Edition / J. Levine, T. Mason, D. Brown. – Sebastopol, CA: O'Reilly Media, 1992. – 384 p.
5. Компиляция. 2: грамматики - http://habrahabr.ru/blogs/programming/99298/
6. Компиляция. 3: бизон - http://habrahabr.ru/blogs/programming/99366/
7. Компиляция. 4: игрушечный ЯП - http://habrahabr.ru/blogs/programming/99397/
8. Кормен, Т. Алгоритмы: построение и анализ: второе издание / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. — М.: Вильямс, 2005. — 1296 с.
9. Костельцев, А.В. Построение интерпретаторов и компиляторов / А.В. Костельцев. – СПб.: Наука и Техника, 2001. – 224 с.
10. Легалов, А.И. Трансляторы: методы разработки / А.И. Легалов - http://www.softcraft.ru/translat.shtml
11. Серебряков, В.А. Основы конструирования компиляторов / В.А. Серебряков, М.П. Галочкин - http://www.citforum.ru/programming/theory/serebryakov/
12. Хеш-таблица - http://ru.wikipedia.org/wiki/Хеш-таблица

Общая постановка задачи и варианты заданий к лабораторной работе №3

Общие требования:

1) Разработать программу, осуществляющую семантический, а также синтаксический и лексический анализ простого языка программирования. Наличие в языке конструкций, отсутствующих в задании, является ошибочным.

2) На вход семантического анализатора подается внешний файл с текстом программы на языке, описанном в конкретном варианте задания.

3) На выходе семантический анализатор должен выдавать 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.

3) Используется интерфейс командной строки, т.е. анализатор в общем случае запускается так:

```
user$: parser2 -parameters input.lng
```

здесь *user$* – подсказка командной строки; *parser2* – имя исполняемого модуля программы семантического анализа; *-parameters* – 0 или более параметров командной строки; *input.lng* – имя внешнего файла с программой на анализируемом языке.

4) 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$: parser2 –ast input.lng
```

Для отключения вывода AST на экран:

```
user$: parser2 –noast input.lng
```

5) 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$: parser2 –tree input.lng
```

Для отключения вывода дерева на экран:

```
user$: parser2 –notree input.lng
```

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

7) 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$: parser2 –lexemes input.lng
```

Для отключения вывода таблицы лексем на экран:

```
user$: parser2 –nolexemes input.lng
```

8) При выводе информации об ошибке обязательно указание на номер строки в исходном файле.

9) Диагностируются лексические, синтаксические и семантические ошибки.

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

Вариант 1. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'), операторы циклов с пред- и постусловием и операторы объявления переменных целого и вещественного типов. Арифметические выражения и циклы состоят из идентификаторов, десятичных чисел с плавающей запятой в обычном и нормализованном экспоненциальном форматах, а также целочисленных констант. Кроме того, элементами арифметических выражений являются знаки присваивания ('='), знаки операций ('+', '-', '*', '/', '%') и круглые скобки.

Вариант 2. Входной язык содержит смешанные выражения, разделенные символом точки с запятой (';'), операторы цикла с предусловием и операторы объявления переменных логического и вещественного типов. Выражения состоят из идентификаторов, нечувствительных к регистру констант *true* и *false*, десятичных чисел с плавающей точкой в обычном формате, знаков присваивания ('='), знаков операций *or*, *xor*, *and*, *not*, операций сравнения, арифметических операций и круглых скобок.

Вариант 3. Входной язык содержит операторы условий *if* с необязательной частью *else* и арифметические выражения, разделенные символом точки с запятой (';'), и операторы объявления переменных строкового и вещественного типов. Операторы условий и арифметические выражения состоят из идентификаторов, десятичных чисел с плавающей запятой в обычном и нормализованном экспоненциальном форматах, строковых литералов, знаков присваивания ('='), знаков операций сравнения (<, >, ==, !=, >=, <=), арифметических и логических операций и круглых скобок.

Вариант 4. Входной язык содержит операторы цикла с параметром, арифметические выражения, разделенные символом точки с запятой (';'), и операторы объявления переменных символьного и вещественного типов. Операторы цикла и выражения состоят из идентификаторов, десятичных чисел с плавающей запятой в обычном и нормализованном экспоненциальном форматах, символьных констант, знаков присваивания ('='), знаков операций сравнения (<, >, ==, !=, >=, <=), арифметических и логических операций и круглых скобок.

Вариант 5. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'), операторы цикла с постусловием и операторы объявления переменных целого и вещественного типов. Арифметические выражения и операторы цикла состоят из идентификаторов, римских чисел

(корректные последовательности заглавных и строчных литер C, L, X, V и I), десятичных чисел с плавающей точкой в обычном формате. Кроме того, элементами арифметических выражений являются знаки присваивания ('='), знаки арифметических операций ('+', '-', '*', '/'), операций сравнения, логических операций и круглые скобки.

Вариант 6. Входной язык содержит смешанные выражения, разделенные символом точки с запятой (';'), условные операторы *if* с необязательной частью *else* и операторы объявления переменных логического и вещественного типов. Выражения и условные операторы состоят из идентификаторов, констант 0 и 1, десятичных чисел с плавающей точкой в обычном формате, знаков присваивания ('='), знаков операций *or*, *xor*, *and*, *not*, арифметических операций, операций сравнения и круглых скобок.

Вариант 7. Входной язык содержит операторы условий *if* с необязательной частью *else*, арифметические выражения, разделенные символом точки с запятой (';'), и операторы объявления переменных целого и вещественного типов. Операторы условий и арифметические выражения состоят из идентификаторов, римских чисел (корректные последовательности заглавных и строчных литер C, L, X, V и I), десятичных чисел с плавающей точкой в обычном формате, знаков присваивания ('='), знаков операций сравнения (<, >, ==, !=, >=, <=), арифметических и логических операций и круглых скобок.

Вариант 8. Входной язык содержит операторы цикла с параметром, арифметические выражения, разделенные символом точки с запятой (';'), и операторы объявления переменных целого и вещественного типов. Операторы цикла и арифметические выражения состоят из идентификаторов, римских чисел (корректные последовательности заглавных и строчных литер C, L, X, V и I), десятичных чисел с плавающей точкой в обычном формате, знаков присваивания ('='), знаков операций сравнения (<, >, ==, !=, >=, <=), арифметических и логических операций и круглых скобок.

Вариант 9. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'), условные операторы *if* с необязательной частью *else* и операторы объявления переменных целых типов (*octal*, *decimal*, *hexadecimal*). Арифметические выражения и условные операторы состоят из идентификаторов, целочисленных констант в шестнадцатеричной, восьмеричной и десятичной системах. Кроме того, элементами арифметических выражений являются знаки присваивания ('='), знаки операций ('+', '-', '*', '/', '%'), логических операций и операций сравнения и круглые скобки.

Вариант 10. Входной язык содержит выражения, разделенные символом точки с запятой (';'), операторы цикла с предусловием и операторы объявления переменных целых типов (*octal*, *decimal*, *hexadecimal*). Выражения и циклы состоят из идентификаторов, целочисленных констант в шестнадцатеричной,

десятичной и восьмеричной системах, знаков присваивания ('='), знаков побитовых операций *or*, *xor*, *and*, *not*, арифметических операций, операций сравнения и круглых скобок.

Вариант 11. Входной язык содержит операторы условий *if* с необязательной частью *else*, арифметические выражения, разделенные символом точки с запятой (';'), и операторы объявления переменных целых типов (*octal*, *decimal*, *hexadecimal*). Операторы условий и выражения состоят из идентификаторов, целочисленных констант в шестнадцатеричной, десятичной и восьмеричной системах, знаков присваивания ('='), знаков операций сравнения (<, >, ==, !=, >=, <=), арифметических и логических операций и круглых скобок.

Вариант 12. Входной язык содержит операторы цикла с параметром типа, арифметические выражения, разделенные символом точки с запятой (';'), и операторы объявления переменных целых типов (*octal*, *decimal*, *hexadecimal*). Операторы цикла и выражения состоят из идентификаторов, целочисленных констант в шестнадцатеричной, десятичной и восьмеричной системах, знаков присваивания ('='), знаков операций сравнения (<, >, ==, !=, >=, <=), логических и арифметических операций и круглых скобок.

Вариант 13. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'), условные операторы *if* с необязательной частью *else* и операторы объявления переменных целого и комплексного типов. Арифметические выражения и условные операторы состоят из идентификаторов, комплексных чисел (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), целочисленных констант, знаков присваивания ('='), знаков операций ('+', '-', '*', '/'), операций сравнения и логических операций и круглых скобок.

Вариант 14. Входной язык содержит смешанные выражения, разделенные символом точки с запятой (';'), условные операторы *if* с необязательной частью *else* и операторы объявления переменных логического и целого типов. Выражения и условные операторы состоят из идентификаторов, нечувствительных к регистру констант *T* и *NIL*, битовых строк (последовательностей из 0 и 1, начинающихся с обязательной пары знаков 0*b* или 0*B*), знаков присваивания ('='), знаков логических операций *or*, *xor*, *and*, *not*, операций сравнения, арифметических операций и круглых скобок.

Вариант 15. Входной язык содержит операторы условий *if* с необязательной частью *else*, арифметические и строковые выражения, разделенные символом точки с запятой (';'), и операторы объявления переменных строкового и целого типов. Операторы условий и выражения состоят из идентификаторов, строковых констант (заключенная в двойные кавычки последовательность любых символов, за исключением двойных кавычек), целочисленных констант,

знаков присваивания ('='), знаков операций сравнения (<, >, ==, !=, >=, <=), логических и арифметических операций и круглых скобок.

Вариант 16. Входной язык содержит операторы цикла с параметром, арифметические выражения, разделенные символом точки с запятой (';'), и операторы объявления переменных целых типов (*binary*, *octal*, *hexadecimal*). Операторы цикла и выражения состоят из идентификаторов, целочисленных констант в восьмеричной и шестнадцатеричной системах, битовых строк (последовательностей из 0 и 1, начинающихся с обязательной пары знаков 0*b* или 0*B*), знаков присваивания ('='), операций сравнения (<, >, ==, !=, >=, <=), арифметических и логических операций и круглых скобок.

Вариант 17. Входной язык содержит выражения, разделенные символом точки с запятой (';'), операторы цикла с постусловием и операторы объявления переменных целого и вещественного типов. Выражения и циклы состоят из идентификаторов, битовых строк (последовательностей из 0 и 1, начинающихся с обязательной пары знаков 0*b* или 0*B*), десятичных чисел с плавающей точкой с обычном формате, знаков присваивания ('='), операций *or*, *xor*, *and*, *not*, операций сравнения и арифметических операций и круглых скобок.

Вариант 18. Входной язык содержит выражения с целыми числами и датами, разделенные символом точки с запятой (';'), операторы условий *if* с необязательной частью *else* и операторы объявления переменных целого типа и типа даты. Выражения и операторы условий состоят из идентификаторов, дат двух любых (на усмотрение разработчика) форматов, целочисленных констант, знаков присваивания ('='), знаков операций сравнения (<, >, ==, !=, >=, <=), арифметических и логических операций и круглых скобок.

Вариант 19. Входной язык содержит выражения с целыми числами и временными величинами, разделенные символом точки с запятой (';'), операторы условий *if* с необязательной частью *else* и операторы объявления целого и временного типов. Выражения и операторы условий состоят из идентификаторов, временных величин двух любых (на усмотрение разработчика языка) форматов, целочисленных констант, знаков присваивания ('='), знаков операций сравнения (<, >, ==, !=, >=, <=), арифметических и логических операций и круглых скобок.

Вариант 20. Входной язык содержит арифметические выражения разделенные символом точки с запятой (';'), операторы цикла с параметром, операторы объявления целого и символьного типов. Выражения и операторы цикла состоят из идентификаторов, символьных констант в стиле языка программирования C, заключенных в апострофы, целочисленных констант в десятичной и шестнадцатеричной системе, знаков присваивания ('='), знаков операций сравнения (<, >, =, !=, >=, <=), арифметических и логических операций и круглых скобок.

Вариант 21. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'), операторы цикла с предусловием и операторы объявления переменных целого и вещественного типов. Арифметические выражения и циклы состоят из идентификаторов, десятичных чисел с плавающей запятой в обычном формате, целых чисел в восьмеричной и десятичной системах. Кроме того, элементами являются знаки присваивания (':='), знаки операций ('+', '-', '*', '/', '%', модуля), операций сравнения и логических операций и круглые скобки.

Вариант 22. Входной язык содержит логические выражения, разделенные символом точки с запятой (';'), условные операторы *if* с необязательной частью *else* и операторы объявления переменных логического и целого типов. Логические выражения и операторы условий состоят из идентификаторов, нечувствительных к регистру констант *t* (истина) и *f* (ложь), битовых строк (последовательностей из 0 и 1, начинающихся с обязательной пары знаков 0*b* или 0*B*), знаков присваивания (':='), знаков операций *or*, *xor*, *and*, *not*, стрелка Пирса (пара знаков -> или ключевое слово *peirce*), операций сравнения и арифметических операций и круглых скобок.

Вариант 23. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'), операторы условий *if* с необязательно частью *else*, операторы объявления переменных вещественного и целого типов. Операторы условий состоят из идентификаторов, десятичных чисел с плавающей запятой в нормализованном экспоненциальном формате, целых чисел в восьмеричной и десятичной системах, знаков присваивания (':='), знаков операций сравнения (<, >, =, <>, >=, <=, =>, =<) и круглых скобок.

Вариант 24. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'), операторы цикла с параметром, операторы объявления переменных вещественного и целого типов. Выражения и операторы цикла состоят из идентификаторов, десятичных чисел с плавающей запятой в обычном и нормализованном экспоненциальном форматах, целочисленных констант, знаков присваивания (':='), знаков операций сравнения (<, >, =, <>, >=, <=, =>, =<), логических и арифметических операций и круглых скобок.

Вариант 25. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'), операторы цикла с постусловием и операторы объявления переменных целых типов (*decimal*, *roman*). Арифметические выражения и циклы состоят из идентификаторов, целочисленных констант, чисел из римских цифр (корректные последовательности заглавных и строчных литер C, L, X, V и I). Кроме того, элементами арифметических выражений являются знаки присваивания (':='), знаки операций ('+', '-', '*', '/', '%'), операций сравнения и логических операций и круглые скобки.

Вариант 26. Входной язык содержит логические выражения, разделенные символом точки с запятой (';'), операторы цикла с параметром и операторы объявления переменных логического и целого типов. Выражения и циклы состоят из идентификаторов, логических констант *0* и *1*, целочисленных констант в десятичной и шестнадцатеричной системах, знаков присваивания (':='), знаков операций *or*, *xor*, *and*, *not*, штрих Шеффера (знак вертикальной черты или ключевое слово *sheffer*), операций сравнения и арифметических операций и круглых скобок.

Вариант 27. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'), операторы условий *if* с необязательной частью *else* и операторы объявления переменных целых типов (*octal*, *decimal*, *roman*). Операторы условий и выражения состоят из идентификаторов, чисел из римских цифр (корректные последовательности заглавных и строчных литер C, L, X, V и I), целочисленных констант в восьмеричной и десятичной системах, знаков присваивания (':='), знаков операций сравнения (<, >, =, <>, >=, <=, =>, =<), арифметических и логических операций и круглых скобок.

Вариант 28. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'), операторы цикла с параметром, операторы объявления переменных целых типов (*octal*, *hexadecimal*, *roman*). Выражения и операторы цикла состоят из идентификаторов, чисел из римских цифр (корректные последовательности заглавных и строчных литер C, L, X, V и I), целочисленных констант в восьмеричной и шестнадцатеричной системах, знаков присваивания (':='), знаков операций сравнения (<, >, =, <>, >=, <=, =>, =<), арифметических и логических операций и круглых скобок.

Вариант 29. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'), условные операторы *if* с необязательной частью *else* и операторы объявления переменных целых типов (*binary*, *octal*, *decimal* и *hexadecimal*). Арифметические выражения и условные операторы состоят из идентификаторов, целочисленных констант в двоичной, восьмеричной, десятичной и шестнадцатеричной системах. Кроме того, элементами являются знаки присваивания (':='), знаки операций ('+', '-', '*', '/', '%', '^'), операций сравнения и логических операций и круглые скобки.

Вариант 30. Входной язык содержит выражения, разделенные символом точки с запятой (';'), операторы цикла с предусловием и операторы объявления переменных целых типов (*decimal*, *hexadecimal*). Выражения и циклы состоят из идентификаторов, целочисленных констант в шестнадцатеричной и десятичной системах, знаков присваивания (':='), знаков побитовых операций *or*, *xor*, *and*, *not*, <<, >> (сдвиг влево и вправо), операций сравнения, арифметических и логических операций и круглых скобок.

Вариант 31. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'), операторы условий *if* с необязательной частью *else*, операторы объявления переменных целых типов (*decimal*, *hexadecimal*). Операторы условий и выражения состоят из идентификаторов, целочисленных констант в десятичной и шестнадцатеричной системах, знаков присваивания (':='), знаков операций сравнения (<, >, =, ==, <>, !=, >=, <=), арифметических, логических и побитовых операций (в том числе циклический сдвиг влево и вправо) и круглых скобок.

Вариант 32. Входной язык содержит арифметические выражения, разделенные символом точки с запятой (';'), операторы цикла с параметром, операторы объявления переменных целого типа (*binary*, *hexadecimal*). Операторы цикла и выражения состоят из идентификаторов, целочисленных констант в шестнадцатеричной системе, битовых строк (последовательностей из 0 и 1, начинающихся с обязательной пары знаков 0*b* или 0*B*), знаков присваивания, инкремента, декремента (':=', '++', '—'), знаков операций сравнения (<, >, =, ==, <>, !=, >=, <=), арифметических и логических операций и круглых скобок.

Приложение А. *ast.h* – Заголовочный файл с описанием структуры атрибутов абстрактного синтаксического дерева

```
/* Based on John Levine's book "flex&bison" */

#ifndef _ABSTRACT_SYNTAX_TREE_H
#define _ABSTRACT_SYNTAX_TREE_H

#include "subexpression.h"

typedef enum
{
    typeBinaryArithOp,     /* 2-местная арифметическая операция */
    typeUnaryArithOp,      /* 1-местная арифметическая операция */
    typeAssignmentOp,      /* 1-местная арифметическая операция */
    typeComparisonOp,      /* 2-местная операция сравнения */
    typeConst,             /* константа */
    typeIdentifier,        /* переменная */
    typeIfStatement,       /* оператор If */
    typeList               /* Списко выражений или операторов */
} NodeTypeEnum;


/* Структура узла Абстрактного синтаксического дерева */
/* У кажого узла должен быть изначально заданный тип */

typedef struct TAbstractSyntaxTreeNode
{
  NodeTypeEnum nodetype;
  SubexpressionValueTypeEnum valueType;
  char* opValue;
  struct TAbstractSyntaxTreeNode* left;
  struct TAbstractSyntaxTreeNode* right;
} NodeAST;

typedef struct
{
  NodeTypeEnum nodetype;        /* Тип typeIfStatement */
  NodeAST* condition;  /* условие */
  NodeAST* trueBranch; /* операторы true-ветки */
  NodeAST* elseBranch; /* операторы необязательной false-ветки */
} TControlFlowNode;

typedef struct
{
  NodeTypeEnum nodetype;                  /* Тип K */
  SubexpressionValueTypeEnum valueType;
  union
  {
    int    iNumber;
    double dNumber;
  };
} TNumericValueNode;

#ifndef _SYMBOL_TABLE_H
#include "symtable.h"
#endif

typedef struct
{
  NodeTypeEnum nodetype;
  TSymbolTableRecord* variable;
  SubexpressionValueTypeEnum valueType;
```

```
} TSymbolTableReference;

typedef struct
{
  NodeTypeEnum nodetype;
  TSymbolTableRecord* variable;
  NodeAST* value;
} TAssignmentNode;

/* Процедуры формирования Абстрактного синтаксического дерева */
NodeAST* CreateNodeAST(NodeTypeEnum cmptype, char* opValue,
//                              SubexpressionValueTypeEnum valueType,
                                NodeAST* left, NodeAST* right
                               );
NodeAST* CreateCompareNode(NodeTypeEnum cmptype, char opValue[], NodeAST* left,
NodeAST* right);

NodeAST* CreateDoubleNumberNode(double doubleValue);
NodeAST* CreateIntegerNumberNode(int integerValue);
NodeAST* CreateControlFlowNode(NodeTypeEnum Nodetype, NodeAST* condition,
                               NodeAST* trueBranch, NodeAST* elseBranch
                              );
NodeAST* CreateReferenceNode(TSymbolTableRecord* symbol);
NodeAST* CreateAssignmentNode(TSymbolTableRecord* symbol, NodeAST *rightValue);

/* Удаление и освобождения памяти Абстрактного синтаксического дерева */
void FreeAST(NodeAST *);

/* Программный интерфейс с лексером */
extern int yylineno;       /* номер строки приходит от лексера */
void yyerror(char* s, ...);

/* Печать абстрактного синтаксического дерева */
void PrintAST(NodeAST* aTree, int level);


#endif
```

Приложение Б. *subexpression.h* – заголовочный файл с типами элементарных выражений и подвыражений

```
/* Based on John Levine's book "flex&bison" */
#ifndef _SUBEXPRESSION_H
#define _SUBEXPRESSION_H

typedef enum
{
    typeInt,          /* Целое */
    typeDouble        /* С плавающей точкой двойной точности */
} SubexpressionValueTypeEnum;

#endif
```

Приложение В. *symtable.h* – заголовочный файл со структурой таблицы символов

```
/* Based on John Levine's book "flex&bison" */

#ifndef _SYMBOL_TABLE_H
#define _SYMBOL_TABLE_H

#include "subexpression.h"

/* Структура элемента таблицы символов */
typedef struct
{
  char name;      /* Имя переменной. В текущей реализации один знак */
  int isDeclared; /* Объявлена ли переменная */
  SubexpressionValueTypeEnum valueType; /* Тип переменной или подвыражения */
  double value;   /* В текущей реализации не используется. Зарезервировано */
} TSymbolTableRecord;

#endif
```

Приложение Г. *helper.c* – Реализация функций обработки абстрактного синтаксического дерева

```
/*
 * Реализация функций для expressions
 * Based upon John Levine's book "flex&bison"
 */
#  include <stdio.h>
#  include <stdlib.h>
#  include <stdarg.h>
#  include <string.h>
#  include <math.h>
#  include "ast.h"

NodeAST*  CreateNodeAST(NodeTypeEnum  nodetype,  char*  opValue,  NodeAST*  left,
NodeAST* right)
{
  NodeAST* a = malloc(sizeof(NodeAST));

  if(!a)
  {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = nodetype;
  a->opValue = opValue;
  a->valueType = left->valueType;
  a->left = left;
  a->right = right;
  return a;
}

NodeAST* CreateCompareNode(NodeTypeEnum cmptype, char opValue[], NodeAST* left,
NodeAST* right)
{
  NodeAST* a = malloc(sizeof(NodeAST));

  if(!a)
  {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = cmptype;
  a->opValue = opValue;
  a->left = left;
  a->right = right;
  return a;
}

NodeAST* CreateDoubleNumberNode(double doubleValue)
{
  TNumericValueNode* a = malloc(sizeof(TNumericValueNode));

  if(!a)
  {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = typeConst;
  a->valueType = typeDouble;
  a->dNumber = doubleValue;
  return (NodeAST *)a;
}
```

```
NodeAST* CreateIntegerNumberNode(int integerValue)
{
  TNumericValueNode* a = malloc(sizeof(TNumericValueNode));

  if(!a)
  {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = typeConst;
  a->valueType = typeInt;
  a->iNumber = integerValue;
  return (NodeAST *)a;
}


NodeAST* CreateControlFlowNode(NodeTypeEnum nodetype, NodeAST* condition,
                               NodeAST* trueBranch, NodeAST* elseBranch
                              )
{
  TControlFlowNode* a = malloc(sizeof(TControlFlowNode));

  if(!a)
  {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = nodetype;
  a->condition = condition;
  a->trueBranch = trueBranch;
  a->elseBranch = elseBranch;
  return (NodeAST *)a;
}

NodeAST* CreateReferenceNode(TSymbolTableRecord* symbol)
{
  TSymbolTableReference* a = malloc(sizeof(TSymbolTableReference));
  if(!a)
  {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = typeIdentifier;
  a->variable = symbol;
  return (NodeAST *)a;
}

NodeAST* CreateAssignmentNode(TSymbolTableRecord* symbol, NodeAST *rightValue)
{
  TAssignmentNode* a = malloc(sizeof(TAssignmentNode));
  if(!a)
  {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = typeAssignmentOp;
  a->variable = symbol;
  a->value = rightValue;
  return (NodeAST *)a;
}

void FreeAST(NodeAST* a)
{
```

```
  if(!a)
    return;
  switch(a->nodetype)
  {
  /* два поддерева */
  case typeBinaryArithOp:
  case typeComparisonOp:
  case typeList:
    FreeAST(a->right);

    /* одно поддерево */
  case typeUnaryArithOp:
    FreeAST(a->left);

    /* терминальный узел */
  case typeConst:
  case typeIdentifier:
    break;
  case typeAssignmentOp:
      free( ((TAssignmentNode *)a)->value);
    break;
  case typeIfStatement:
    free( ((TControlFlowNode *)a)->condition);
    if( ((TControlFlowNode *)a)->trueBranch)
        FreeAST( ((TControlFlowNode *)a)->trueBranch);
    if( ((TControlFlowNode *)a)->elseBranch)
        FreeAST( ((TControlFlowNode *)a)->elseBranch);
    break;

  default: printf("internal error: free bad node %c\n", a->nodetype);
  }

  free(a); /* Сам узел тоже нужно освободить */
}

void yyerror(char *s, ...)
{
  va_list ap;
  va_start(ap, s);

  fprintf(stderr, "%d: error: ", yylineno);
  vfprintf(stderr, s, ap);
  fprintf(stderr, "\n");
}

int main (int argc, char* argv[])
{
  int yyparse();
  if (argc < 2)
  {
      printf("Too few parameters.\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;
  }
  return yyparse();
}

/* вывод содержимого Абстрактного синтаксического дерева */
```

```
void PrintAST(NodeAST* a, int level)
{
  printf("%*s", 2 * level, "");     /* indent to this level */
  ++level;

  if(!a)
  {
    printf("NULL\n");
    return;
  }

  switch(a->nodetype)
  {
    /* Константа */
  case typeConst:
      if(typeDouble == ((TNumericValueNode *)a)->valueType)
        printf("number %4.4g\n", ((TNumericValueNode *)a)->dNumber);
      else if(typeInt == ((TNumericValueNode *)a)->valueType)
      printf("number %d\n", ((TNumericValueNode *)a)->iNumber);
      else
      printf("bad constant\n");
      break;

  /* Ссылка на таблицу символов */
  case typeIdentifier:
      printf("ref %c\n", ((TSymbolTableReference *)a)->variable->name);
      break;

    /* Выражения */
  case typeList:
  case typeBinaryArithOp:
  case typeComparisonOp:
    printf("binop %s\n", a->opValue);
    PrintAST(a->left, level);
    PrintAST(a->right, level);
    return;

  /* Унарные операции */
  case typeUnaryArithOp:
    printf("unop %s\n", a->opValue);
    PrintAST(a->left, level);
    return;

    /* Присваивание */
  case typeAssignmentOp:
      printf("= %c\n", ((TSymbolTableReference *)a)->variable->name);
      PrintAST( ((TAssignmentNode *)a)->value, level);
    return;

  /* Оператор потока управления */
  case typeIfStatement:
    printf("flow\n");
    PrintAST( ((TControlFlowNode *)a)->condition, level);
    if( ((TControlFlowNode *)a)->trueBranch)
      {
        printf("%*s", 2 * level, "");
        printf("true-branch\n");
        PrintAST( ((TControlFlowNode *)a)->trueBranch, level + 1);
      }
    if( ((TControlFlowNode *)a)->elseBranch)
      {
        printf("%*s", 2 * level, "");
        printf("false-branch\n");
      PrintAST( ((TControlFlowNode *)a)->elseBranch, level + 1);
```

```
      }
    return;

  default: printf("bad %c\n", a->nodetype);
    return;
  }
}
```

Приложение Д. *expression.l* – Лексический анализатор

```
/*  Based upon John Levine's book "flex&bison" */
%option nounistd

%option noyywrap nodefault yylineno
%{
#include <ctype.h>
#include "ast.h"
#include "expression.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

%}

/* Экспонента числа с плавающей точкой */
EXP   ([Ee][-+]?[0-9]+)

%%
 /* однозначковые операции */
"+" |
"-" |
"*" |
"/" |
"=" |
";" |
"(" |
")" |
"{" |
"}"     { return yytext[0]; }

 /* операции сравнения */
">"  |
"<"  |
"!=" |
"==" |
">=" |
"<="    { strcpy(yylval.string, yytext); return CMP; }

 /* ключевые слова */

"if"    { return IF; }
"else"  { return ELSE; }

[a-zA-Z]             { yylval.i = tolower (yytext[0]) - 'a';
                       return VARIABLE;
                     }
0|[1-9][0-9]*        { yylval.i = atoi(yytext); return INTCONST; }

([0-9]*\.[0-9]+|[0-9]+\.){EXP}? | //   { yylval.d = atof(yytext); return NUMBER;
}
[0-9]+{EXP}                        { yylval.d = atof(yytext); return NUMBER; }


[ \t\n] ;  /* пропуск пробельных символов */
```

```
.      { yyerror("Magical mistery character %c\n", yytext[0]); }
%%
```

Приложение Е. *expression.y* – Семантический и синтаксический анализатор

```
%{
/* Based upon John Levine's book "flex&bison"*/
#include <stdio.h>
#include <stdlib.h>
#include "ast.h"

#define MAX_TABLE_SIZE 26

TSymbolTableRecord UserVariableTable[MAX_TABLE_SIZE];

extern int yylex();
%}

%verbose

%union
{
  NodeAST* a;
  double d;
  int i;
  char string[3];
}

/* declare tokens */
%token <d> NUMBER
%token <i> INTCONST VARIABLE

%token IF ELSE

%nonassoc IFX
%nonassoc ELSE


%nonassoc <string>CMP
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS

%type <a> exp cond_stmt stmtlist statement stmtlist_tail

%start prog

%%

/*prog : statement*/
prog : stmtlist
     {
      PrintAST($1, 0);
      FreeAST($1);
     }
     ;

statement : exp ';' { $$ = $1; }
     | VARIABLE '=' exp ';'
       {
         UserVariableTable[$1].name = $1 + 'A';
         if (0 == UserVariableTable[$1].isDeclared)
         {
           UserVariableTable[$1].isDeclared = !0;
           UserVariableTable[$1].valueType  = $3->valueType;
         }
```

```
          else if ($3->valueType != UserVariableTable[$1].valueType)
          {
            yyerror("types incompatible for assignment");
          }
          $$ = CreateAssignmentNode(&UserVariableTable[$1], $3);
        }
      | cond_stmt { $$ = $1; }
      | '{' stmtlist '}' { $$ = $2; }
      ;

cond_stmt: IF '(' exp ')' statement %prec IFX
           {
             $$ = CreateControlFlowNode(typeIfStatement, $3, $5, NULL);
           }
   | IF '(' exp ')' statement ELSE statement
           {
             $$ = CreateControlFlowNode(typeIfStatement, $3, $5, $7);
           }
   ;

stmtlist : statement stmtlist_tail
           { if ($2 == NULL)
                $$ = $1;
             else
              $$ = CreateNodeAST(typeList, "L", $1, $2);
           }
   ;

stmtlist_tail : /* пустая строка */
              { $$ = NULL; }
        | stmtlist  { $$ = $1; }
        ;

exp : exp CMP exp
      {
        char op[3] = "";

        SubexpressionValueTypeEnum typeOp1, typeOp2;
        typeOp1 = $1->valueType;
        typeOp2 = $3->valueType;
        if (typeOp1 != typeOp2)
        {
          yyerror("types incompatible");
        }
        strcpy(op, $2);
        $$ = CreateCompareNode(typeComparisonOp, op, $1, $3);
      }
   | exp '+' exp    {
                      SubexpressionValueTypeEnum typeOp1, typeOp2;
                      typeOp1 = $1->valueType;
                      typeOp2 = $3->valueType;
                      if (typeOp1 != typeOp2)
                      {
                        yyerror("types incompatible");
                      }
                      $$ = CreateNodeAST(typeBinaryArithOp, "+", $1, $3);
                    }
   | exp '-' exp    {
                      SubexpressionValueTypeEnum typeOp1, typeOp2;
                      typeOp1 = $1->valueType;
                      typeOp2 = $3->valueType;
                      if (typeOp1 != typeOp2)
                      {
                        yyerror("types incompatible");
```

```
                    }
                    $$ = CreateNodeAST(typeBinaryArithOp, "-", $1, $3);
                  }
  | exp '*' exp          {
                    SubexpressionValueTypeEnum typeOp1, typeOp2;
                    typeOp1 = $1->valueType;
                    typeOp2 = $3->valueType;
                    if (typeOp1 != typeOp2)
                    {
                      yyerror("types incompatible");
                    }
                    $$ = CreateNodeAST(typeBinaryArithOp, "*", $1, $3);
                  }
  | exp '/' exp          {
                    SubexpressionValueTypeEnum typeOp1, typeOp2;
                    typeOp1 = $1->valueType;
                    typeOp2 = $3->valueType;
                    if (typeOp1 != typeOp2)
                    {
                      yyerror("types incompatible");
                    }
                    $$ = CreateNodeAST(typeBinaryArithOp, "/", $1, $3);
                  }
  | '(' exp ')'          { $$ = $2; }
  | '-' exp %prec UMINUS
    {
      $$ = CreateNodeAST(typeUnaryArithOp, "-", $2, NULL);
    }
  | NUMBER               { $$ = CreateDoubleNumberNode($1); }
  | INTCONST             { $$ = CreateIntegerNumberNode($1); }
  | VARIABLE
    {
      if(0 == UserVariableTable[$1].isDeclared)
      {
        char errorDeclaration[32] = "";
        sprintf(errorDeclaration, "Variable %c isn't declared", $1 + 'A');
        yyerror(errorDeclaration);
      }
      $$ = CreateReferenceNode(&UserVariableTable[$1]);
    }
;


%%
```

Приложение Ё. *makefile*

```
CFLAGS=-g -Wall
YACC=bison --report=all -d -l -o expression.tab.c
LEX=flex
LFLAGS=-i -o lex.yy.c --noline

# Things that get included in our Yacc file
INCLUDED_FILES = \
      ast.h \
      subexpression.h \
      symtable.h

# The various .o files that are needed for executables.
OBJECT_FILES = expression.tab.o helper.o lex.yy.o

default: expression-ast

expression-ast: $(OBJECT_FILES)
      $(LINK.o) -o $@ $^

expression.tab.o: expression.tab.c $(INCLUDED_FILES)

expression.tab.c: expression.y
      $(YACC) $(YFLAGS) $^

lex.yy.c: expression.l
      $(LEX) $(LFLAGS) $^

clean-all:
      make clean
      -rm -f expression-ast

clean:
      -rm -f *.o
      -rm -f expression.output
      -rm -f expression.tab.*
      -rm -f lex.yy.c
```