С. П. Кунцевич

Дополнительные главы информатики. Системное программное обеспечение

Практикум по дисциплине «Дополнительные главы информатики. Системное программное обеспечение» для студентов 5 курса специальности Прикладная математика (1-31 03 03)

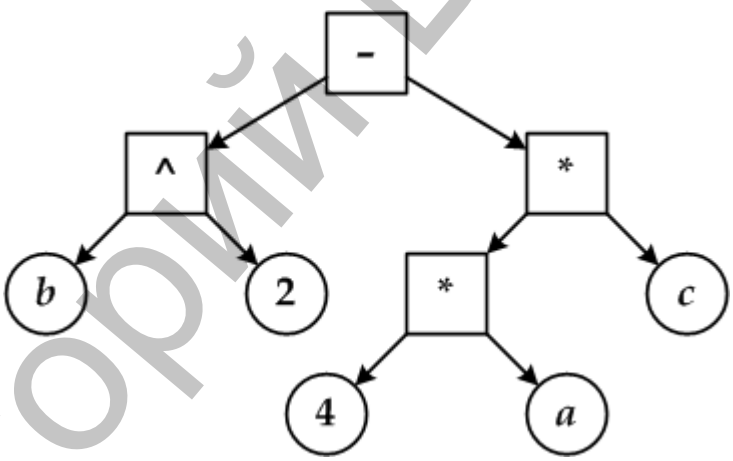
2008

Лабораторная работа №1

Вычисление арифметических выражений по ПОЛИЗ

Одной из причин появления языков программирования высокого уровня явились вычислительные задачи, требующие больших объёмов рутинных вычислений. Поэтому изначально при разработке языков программирования для них разрабатывались формы записи выражений, максимально приближенные к традиционному математическому языку. В этой связи одной из главных задач системного программирования стало исследование способов представления и вычисления выражений.

Наиболее наглядной формой представления выражений является синтаксическое дерево – ориентированный ациклический граф. На рисунке изображено синтаксическое дерево для выражения *b^2–4*a*c*. В таком дереве листьями являются операнды, узлами – операторы, корнем – оператор, выполняемый последним.



Для вычисления значения выражения необходимо реализовать алгоритм полного обхода дерева. Более простым оказывается алгоритм вычисления арифметического выражения, основанный на линеаризованном представлении такого графа, называемом *польской инверсной записью* (ПОЛИЗ) или *постфиксной записью*.

Постфиксная запись арифметического выражения представляет собой список вершин синтаксического графа, полученных при обходе снизу-вверх слева-направо. Для графа, изображенного на рисунке, постфиксная запись будет иметь вид: *b 2 ^ 4 a * c * –*.

Таким образом, в постфиксной записи, в отличие от традиционной для математики *инфиксной*, операторы располагаются *после* операндов. Часто используется и *префиксная запись*, в которой операторы располагаются *перед* операндами, например: *разность(степень(b,2),произведение(произведение(4,a),c))*.

Основными достоинствами постфиксной записи является отсутствие скобок и достаточно простой алгоритм вычисления выражений.

Алгоритм вычисления выражений по ПОЛИЗ.

В алгоритме для хранения чисел используется стек.

1. Поэлементно читается входное выражение.

2. Если прочитанный элемент – операнд, то его нужно положить в стек.

3. Если прочитанный элемент – оператор, то необходимо извлечь два операнда из стека, выполнить с ними операцию, соответствующую данному оператору, и числовой результат положить обратно в стек.

4. После того как прочитан и обработан последний элемент исходного выражения, единственное оставшееся в стеке число является значением выражения.

Алгоритм преобразования из постфиксной записи в инфиксную.

В алгоритме для хранения строк используется стек.

1. Поэлементно читается входное выражение.

2. Если прочитанный элемент – операнд, то его нужно положить в стек.

3. Если прочитанный элемент – оператор, то необходимо извлечь два операнда из стека, провести над ними действие данным оператором и результат, заключенный в скобки, положить обратно в стек. Далее это выражение будет рассматриваться как единый операнд.

4. После того как прочитан и обработан последний элемент исходной последовательности, единственная оставшаяся в стеке строка будет содержать инфиксную запись исходного выражения.

Задание 1. Разработайте программу для вычисления значения числового арифметического выражения, представленного в постфиксной форме.

Задание 2. Разработайте программу для преобразования числового арифметического выражения из постфиксной в инфиксную форму.

Указание. Реализуйте простейшие проверки на корректность входного выражения.

Лабораторная работа №2

Перевод арифметических выражений в ПОЛИЗ

Существует несколько алгоритмов перевода арифметических выражений в постфиксную форму. Один из эффективных алгоритмов, основанный на использовании *стека с приоритетами*, был предложен в 1960 г. голландским ученым Эдсгером Дейкстрой. Простейший вариант этого метода применим только к арифметическим и логическим выражениям, содержащим простые переменные, знаки арифметических и логических операций, знаки операций отношения и круглые скобки.

В соответствии с алгоритмом, каждому знаку арифметической операции сопоставляется числовой *приоритет*. Более высокий приоритет получают операции, выполняемые в первую очередь. Скобкам также удобно сопоставить приоритет, меньший приоритета любой операции:

Знак	*Приоритет*
(	0
)	1
+, –	2
*, /	3

Алгоритм перевода выражений в обратную польскую запись

1. Входная строка считывается слева направо.

2. Операнды по мере их считывания помещаются в выходную строку.

3. Прочитанные знаки операций заносятся в стек по следующим правилам:

а) если стек пуст, или приоритет текущего знака больше приоритета знака на вершине стека, то новый знак просто добавляется в стек;

б) если стек не пуст и приоритет текущего знака меньше или равен приоритету знака на вершине стека, из стека извлекаются и переносятся в результирующую строку все знаки с приоритетами большими или равными приоритету текущего знака. После этого текущий знак заносится в стек.

4. Прочитанные скобки обрабатываются следующим образом:

а) открывающая скобка всегда помещается в стек, впоследствии, благодаря низкому приоритету, её не может вытолкнуть ни один знак, кроме закрывающей скобки;

б) закрывающая скобка, поскольку её приоритет меньше приоритета любой операции, приводит к извлечению из стека в результирующую строку всех

знаков операций до ближайшей открывающей скобки. Открывающая скобка также извлекается из стека, но в выходную строку не записывается, сама закрывающая скобка также никуда не записывается.

5. После просмотра всех символов входной строки из стека извлекаются и переносятся в выходную строку все оставшиеся знаки.

Описанный алгоритм может быть распространен и на выражения, содержащие функции, переменные с индексами, а также на структурные операторы языка программирования.

Задание. Разработайте программу для преобразования арифметического выражения из инфиксной в постфиксную форму записи.

Указание. При реализации алгоритма для определения приоритета знаков операций удобно использовать отдельную функцию.

Лабораторная работа №3

Организация таблиц идентификаторов

На различных этапах компиляции для хранения различных характеристик элементов исходной программы используются *таблицы идентификаторов*. Таблица идентификаторов состоит из записей, каждая из которых соответствует единственному элементу – переменной, константе, функции и т.п. Набор характеристик, соответствующий каждому элементу, зависит от типа этого элемента, от его смысла и роли, которую он исполняет в программе.

Занесение нового имени в таблицу идентификаторов происходит при обработке его *определяющего вхождения*, т.е. объявления. При трансляции *использующего вхождения*, т.е. операторов, содержащих тот или иной идентификатор, происходит обращение к таблице для определения его атрибутов.

Самой часто выполняемой операцией является *поиск* идентификатора в таблице. Поэтому при проектировании таблиц идентификаторов большое внимание уделяется скорости поиска.

Одним из эффективных способов является организация таблиц идентификаторов в виде *бинарного дерева*.

Алгоритм добавления идентификатора в бинарное дерево.

0. Если дерево пустое, то создать новый узел, поместить в него имя идентификатора, сделать новый узел корневым и завершить алгоритм (возвратить ссылку на созданный узел), иначе – перейти к шагу 1.

1. Сделать текущим узлом корневой узел дерева.

2. Сравнить имя очередного идентификатора с именем идентификатора, содержащегося в текущем узле.

3. Если имя очередного идентификатора меньше, то перейти к шагу 4, если равно – прекратить выполнение алгоритма с ошибкой (двух одинаковых идентификаторов быть не должно!), иначе – перейти к шагу 6.

4. Если у текущего узла существует левый потомок, то сделать его текущим узлом и вернуться к шагу 2, иначе – перейти к шагу 5.

5. Создать новый узел, поместить в него имя идентификатора, сделать новый узел левым потомком текущего узла и завершить алгоритм (возвратить ссылку на созданный узел).

6. Если у текущего узла существует правый потомок, то сделать его текущим узлом и вернуться к шагу 2, иначе – перейти к шагу 7.

7. Создать новый узел, поместить в него имя идентификатора, сделать новый узел правым потомком текущего узла и завершить алгоритм (возвратить ссылку на созданный узел).

Алгоритм поиска идентификатора в бинарном дереве.

0. Если дерево пустое, то завершить алгоритм с ошибкой, иначе – перейти к шагу 1.

1. Сделать текущим узлом корневой узел дерева.

2. Сравнить имя искомого идентификатора с именем идентификатора, содержащимся в текущем узле дерева.

3. Если имена совпадают, то искомый идентификатор найден, алгоритм завершается (возвратить ссылку на найденную вершину), иначе перейти к шагу 4.

4. Если имя очередного идентификатора меньше, то перейти к шагу 5, иначе – перейти к шагу 6.

5. Если у текущего узла существует левый потомок, то сделать его текущим узлом и вернуться к шагу 2, иначе – искомый идентификатор не найден, алгоритм завершается с ошибкой.

6. Если у текущего узла существует правый потомок, то сделать его текущим узлом и вернуться к шагу 2, иначе – искомый идентификатор не найден, алгоритм завершается с ошибкой.

Если последовательность определяющих вхождений идентификаторов в исходной программе является статистически неупорядоченной, то описанные алгоритмы будут иметь среднее время работы порядка $O(\log_2 N)$.

Задание. Разработайте программу для вычисления значения введенного пользователем арифметического выражения с переменными. Значения переменных должны запрашиваться у пользователя. Для этого:

1. Реализуйте алгоритмы добавления и поиска идентификатора в бинарном дереве.

2. Модифицируйте программу из лаб. раб. №2 таким образом, чтобы найденные в арифметическом выражении идентификаторы заносились в таблицу идентификаторов. При этом в таблицу заносите и значения переменных, введенные с клавиатуры.

3. Модифицируйте программу из лаб. раб. №1 таким образом, чтобы при вычислении значения арифметического выражения использовались значения переменных, хранящиеся в таблице идентификаторов.

Лабораторная работа №4

Программирование лексического анализатора. Конечные автоматы

Напомним, что на фазе ***лексического анализа*** входная программа, представляющая собой поток литер, разбивается транслятором на лексемы – терминальные символы в соответствии с алфавитом языка.

Лексический анализатор может работать в двух основных режимах: либо как отдельная подпрограмма, вызываемая синтаксическим анализатором для получения очередной лексемы, либо выполняя полный проход, результатом которого является файл лексем.

В процессе выделения лексем лексический анализатор может как самостоятельно строить таблицы объектов (идентификаторов, строк, чисел и т.д.), так и выдавать значения для каждой лексемы при очередном к нему обращении.

Также на этапе лексического анализа обнаруживаются некоторые (простейшие) ошибки (недопустимые символы, неправильная запись чисел, идентификаторов и др.).

Краткие сведения из теории конечных автоматов

Конечным автоматом (КА) называют пятерку следующего вида:

$$M(\mathbf{Q}, \mathbf{V}, f, q_0, \mathbf{F}),$$

где

Q – конечное множество состояний автомата;

V – конечное множество допустимых входных символов (алфавит автомата);

f – функция переходов, отображающая декартово произведение множеств $\mathbf{V}\times\mathbf{Q}$ во множество подмножеств **Q**:

$$f(a, q)=\mathbf{R},\ a\in\mathbf{V},\ q\in\mathbf{Q},\ \mathbf{R}\subseteq\mathbf{Q};$$

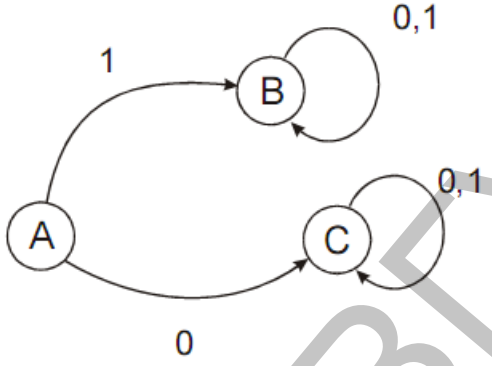
q_0 — начальное состояние автомата **Q**, $q_0\in\mathbf{Q}$;

F — непустое множество конечных состояний автомата, $\mathbf{F}\subseteq\mathbf{Q}$, $\mathbf{Q}\neq\varnothing$.

Графически КА автоматы принято изображать с помощью диаграмм (графов) переходов. *Диаграмма переходов КА* – это направленный граф, вершины которого помечены символами состояний КА, и содержащий помеченные дуги, описывающие допустимые переходы, т.е. на графе изображается дуга (p, q), помеченная символом $a\in\mathbf{V}$, если $q \in f(p, a)$.

Например, на рисунке приведена диаграмма переходов КА с множеством состояний **Q**={A,B,C}, алфавитом **V**={0,1}, начальным состоянием q_0=A и функцией переходов, заданной следующим образом:

f(A,1)={B},
f(A,0)={C},
f(B,0)=f(B,1)={B},
f(C,0)=f(C,1)={C}.



Обычно начальное и конечные состояния на графе помечаются особым образом.

Работа конечного автомата представляет собой последовательность шагов (или тактов). На каждом шаге автомат находится в одном из своих состояний q (в текущем состоянии), в начале работы автомат всегда находится в начальном состоянии q_0. На следующем шаге он может перейти в другое состояние или остаться в текущем. То, в какое состояние автомат перейдет на следующем шаге работы, определяет функция переходов f. Она зависит не только от текущего состояния, но и от того, какой символа из алфавита **V** был подан на вход автомата. Когда функция перехода допускает несколько следующих состояний автомата, то КА может перейти в любое из этих состояний. Работа КА продолжается до тех пор, пока на его вход поступают символы.

КА называют *полностью определенным*, если в каждом его состоянии существует функция перехода для всех возможных входных символов:

$$\forall a \in \mathbf{V} \;\; \forall q \in \mathbf{Q} \;\; \exists\, f(a, q) = \mathbf{R}, \;\; \mathbf{R} \subseteq \mathbf{Q}.$$

КА называют *детерминированным*, если для любой допустимой комбинации входного символа и текущего состояния значение функции переходов содержит не более одного следующего состояния. В противном случае КА называют *недетерминированным*.

Конечные автоматы часто используются для проверки принадлежности строки символов некоторому языку. Программная реализация любого детерминированного КА может быть представлена с помощью подпрограммы. Для отображения текущего состояния КА достаточно одной переменной, имеющей пе-

речисляемое множество значений. Переходы из одного состояния КА в другое представляются как изменение значения этой переменной и могут реализованы, например, с помощью операторов выбора (switch).

Работа подпрограммы-КА должна продолжаться до тех пор, пока не будет достигнут конец анализируемой строки символов. Для вычисления результата необходимо проверить текущее состояние КА. Если это одно из конечных состояний, то работа автомата завершается успешно и входная строка считается принадлежащей заданному языку, в противном случае – нет. Кроме того, при выполнении лексического анализа по состоянию конечного автомата можно судить о типе, который имеет анализируемая лексема.

Программная реализация конечного автомата

1. Установить начальное состояние КА.
2. Для каждого символа анализируемой строки выполнить шаг 3.
3. В зависимости от текущего состояния КА и очередного символа установить новое состояние КА.
4. Проверить текущее состояние КА и вернуть результат.

Задание 1. Составьте перечень состояний, в котором может находится КА для анализа лексем, составляющих буквенно-числовые арифметические выражения:

– идентификаторы;

– дробные числа (десятичный разделитель – точка);

– арифметические операторы +, –, *, /;

– скобки (и).

Постройте диаграмму переходов КА.

Указание. В перечень состояний КА удобно добавить хотя бы одно *ошибочное* состояние.

Задание 2. Разработайте программную реализацию полученного КА для проверки корректности записи введенных с клавиатуры лексем и определения их типа.

Лабораторная работа №5

Программирование лексического анализатора. Таблицы лексем

Результатом работы лексического анализатора является перечень всех найденных в тексте исходной программы лексем с учетом характеристик каждой лексемы. Этот перечень лексем можно представить в виде таблицы, называемой *таблицей лексем*. Таблица лексем фактически содержит весь текст исходной программы, обработанный лексическим анализатором.

Лексемы в таблице лексем обязательно располагаются в том же порядке, что и в исходной программе (порядок лексем в ней не меняется). Каждой лексеме в таблице лексем соответствует некий уникальный условный код, зависящий от типа лексемы, и дополнительная служебная информация: информацию о виде лексемы, ее типе и, возможно, значении. Обычно структуры данных, служащие для организации такой таблицы, имеют два поля: первое — тип лексемы, второе — указатель на информацию о лексеме.

Пример. Таблица лексем для выражения ***for i:=1 to 10 do***

Лексема	**Тип лексемы**	**Атрибут**
for	KEY_FOR	
i	IDENT	i
:=	OP_ASSIGN	
1	CONST	1
to	KEY_TO	
10	CONST	10
do	KEY_DO	

В общем случае алгоритм работы лексического анализатора можно описать так:

1. Из входного потока извлечь очередной символ.

2. Если это пробельный символ, то проигнорировать его и перейти к шагу 1.

3. Обратиться к распознавателю лексем (КА). Запущенный распознаватель просматривает входной поток символов, выделяя символы, входящие в требуемую лексему, до обнаружения очередного символа, который может ограничивать лексему, либо до обнаружения ошибочного символа.

4. При успешном распознавании информация о выделенной лексеме заносится в таблицу лексем, алгоритм возвращается к *шагу 2* и продолжает анализировать входной поток с того символа, на котором остановился распознаватель лексем.

5. При неуспешном распознавании выдать сообщение об ошибке.

Задание. Разработайте лексический анализатор, который на основании программы на языке С формирует текстовый файл, содержащий таблицу лексем.

Лексический анализатор должен воспринимать полный набор ключевых слов языка С, десятичные целочисленные и дробные константы, ограниченный набор пунктуаторов (символов-разделителей):

знаки пунктуации: **, ; :**

скобки: **( ) [ ] { }**

арифметические операторы: **= + – * / %**

Указание 1. Помните, что символы-разделители не только ограничивают числовые константы и идентификаторы, но и являются самостоятельными лексемами.

Указание 2. При реализации конечного автомата удобно заранее не различать идентификаторы и ключевые слова, и только после завершения процесса распознавания попытаться найти идентификатор в списке ключевых слов.

Указание 3. При выводе сообщения об ошибке указывайте строку и позицию ошибочного символа.

Справочник 1. Полный список ключевых слов языка С в соответствии со стандартом ISO/IEC 9899:1990 “Programming Languages – C”:

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

Лабораторная работа №6

Программирование лексического анализатора. Подсветка синтаксиса

Одной из самых наглядных форм использования лексического анализатора является *подсветка синтаксиса* языков программирования, реализованная в специализированных текстовых редакторах и интегрированных средах программирования.

```
int main(int argc, char * argv[]) {
  puts("Hello, World!");
  return 0;
}
```

Задание. Используя результаты предыдущей лабораторной работы, разработайте программу-конвертор, формирующую гипертекстовую страницу, отображающую подсветку синтаксиса для исходной программы на языке С.

Предусмотрите отдельное оформление для следующих элементов:

– ключевые слова;

– идентификаторы;

– константы.

Указание. При реализации подсветки синтаксиса в гипертекстовых страницах удобно использовать *таблицы стилей*. В простейшем случае, содержание страницы будет соответствовать таблице лексем.

```
<HTML>
<HEAD>
<TITLE>HelloWorld.cpp</TITLE>
<STYLE TYPE="text/css">
span.keyword {font-weight: bold;}
span.ident {color: blue;}
span.const {color: green;}
span.string {color: red;}
</STYLE>
</HEAD>
<BODY>
<CODE>
<span class="keyword">int</span>
<span class="ident">main</span>
(
<span class="keyword">int</span>
<span class="ident">argc</span>
,
<span class="keyword">char</span>
*
<span class="ident">argv</span>
[
]
{
<BR>
<span class="ident">puts</span>
(
<span class="string">"Hello, World"</span>
)
;
<BR>
<span class="keyword">return</span>
<span class="const">0</span>
;
<BR>
}
</CODE>
</BODY>
</HTML>
```

Лабораторная работа №7

Объектно-ориентированная модель синтаксического дерева программы

Главной задачей работы синтаксического анализатора, основанного на контекстно-свободной грамматике языка, является определение последовательность правил грамматики, примененных для построения анализируемой цепочки. Этой последовательности вполне достаточно для получения полного представления о типе и структуре разобранной синтаксической конструкции входного языка. В частности, по данной последовательности может быть построено синтаксическое дерево программы. Обычно для представления синтаксических деревьев используются связанные списочные структуры.

При представлении списочных структур удобно использовать объектно-ориентированный подход, при котором каждый узел дерева представляет собой объект.

При таком подходе для описания классов за основу можно брать порождающие правила грамматики языка:

– каждому нетерминальному символу обычно соответствует отдельный *класс* (иногда абстрактный);

– нескольких альтернативных правых частей правил (или несколько правил для одного и того же нетерминального символа) обычно реализуется путём *наследования*, т.е. создания нескольких потомков одного и того же базового (абстрактного) класса;

– наличие нетерминальных символов в правой части правила обычно выражается в *использовании* одного класса другим, т.е. в описываемом классе должны присутствовать ссылки на экземпляры других классов;

– необязательные нетерминальные символы в правой части правила означает возможность нулевой ссылки на экземпляры соответствующих классов, для чего требуется соответствующая проверка при последующей обработке дерева;

– наличие терминальных символов в правой части порождающего правила никак не отражается в описании соответствующего класса (они используются только синтаксическим анализатором для распознавания правила, или, например, при форматированном выводе).

Задание. Используя фрагмент грамматики языка С, разработайте комплект классов для представления синтаксического дерева. Предусмотрите наличие методов для обращения к полям классов (set* и get*); текстового вывода на экран (в файл).

Разработайте программу, демонстрирующую создание синтаксического дерева для алгоритма Евклида и вывод его на печать.

Указание. Для правил, помеченных цветом, реализовывать соответствующие классы не обязательно.

Грамматика языка С (фрагмент)

statement:
- *labeled-statement*
- *compound-statement*
- *expression-statement*
- *selection-statement*
- *iteration-statement*
- *jump-statement*

labeled-statement:
- *identifier* **:** *statement*
- **case** *constant-expression* **:** *statement*
- **default :** *statement*

compound-statement:
- **{** *block-item-list*$_{opt}$ **}**

block-item-list:
- *block-item*
- *block-item-list block-item*

block-item:
- *declaration*
- *statement*

expression-statement:
- *expression*$_{opt}$ *;*

selection-statement:

if (*expression* **)** *statement*

if (*expression* **)** *statement* **else** *statement*

switch (*expression* **)** *statement*

iteration-statement:

while (*expression* **)** *statement*

do *statement* **while (** *expression* **) ;**

for ($expression_{opt}$ **;** $expression_{opt}$ **;** $expression_{opt}$ **)** *statement*

jump-statement:

goto *identifier* **;**

continue ;

break ;

return $expression_{opt}$ **;**

Лабораторная работа №8

Программирование синтаксического анализатора

Задание. С использованием одного из алгоритмов нисходящего синтаксического анализа, разработайте синтаксический анализатор, который:

а) проверяет, является ли некоторый текст синтаксически правильной конструкцией (statement) на языке программирования C;

б) строит синтаксическое дерево данной конструкции;

в) выводит построенное синтаксическое дерево на экран (в файл).

Указания.

1. При программировании синтаксического анализатора достаточно использовать ограниченный синтаксис языка программирования C (см. лаб. раб. №7).

2. Выполнять полный синтаксический анализ арифметических выражений нет необходимости. Достаточно корректно определять их границы (по знаку «)» или «;»).

Лабораторная работа № 9

Синтаксический анализ арифметических выражений

Задание 1. С использованием одного из алгоритмов восходящего синтаксического анализа, разработайте синтаксический анализатор, который:

а) проверяет, является ли некоторый текст синтаксически правильным арифметическим выражением на языке программирования C;

б) строит синтаксическое дерево данной конструкции.

Задание 2. Интегрируйте разработанный синтаксический анализатор и синтаксический анализатор из предыдущей лабораторной работы для их совместного использования. Убедитесь в работоспособности полученной системы.

Указание. Минимальный набор арифметических операторов для синтаксического анализатора:

= + – * / %

Лабораторная работа №10

Семантический анализ

Задание. Дополните ранее разработанный синтаксический анализатор возможностями элементарного семантического анализа:

– все используемые в арифметических выражениях переменные должны быть заранее объявлены и находиться в области видимости;

– запрещается повторное объявление переменной внутри одного и того же блока (*compound-statement*);

– повторное объявление переменной внутри вложенного блока (*compound-statement*) является допустимым.

Указание. Для объявления переменных достаточно реализовать анализ простейшей синтаксической конструкции (*declaration*) вида:

```
ИмяБазовогоТипа ИмяПеременной;
```

например:

```
int N;
```

Задание*. Попробуйте реализовать:

– объявление не отдельной, а набора переменных:

```
int a,b,c,d;
```

– инициализацию переменных при объявлении:

```
int N=10;
```

– контроль использования переменных (т.е. выявите те переменные, которые были объявлены в блоке, но не использовались);

– и т.п.

Лабораторная работа №11

Виртуальные машины

Часть 1. Проектирование стековой виртуальной машины

Задание 1. Разработайте набор классов для представления команд виртуальной стековой машины. Система команд должна позволять:

– заносить в стек виртуальной машины заданную числовую константу;

– заносить в стек виртуальной машины значение заданной переменной (из общего хранилища переменных);

– выполнять арифметические действия (+, –, *, /, %);

– выполнять присваивание (=) и т.п.

Задание 2. Дополните синтаксический анализатор арифметических выражений (см. лаб. раб. 9) возможностью построения на основе синтаксического дерева арифметического выражения соответствующего списка (массива) команд для стековой виртуальной машины (реализуйте алгоритм полного обхода дерева).

Задание 3. Разработайте стековую виртуальную машину, которая на основе списка (массива) команд и общего хранилища переменных вычисляет значение арифметического выражения.

Лабораторная работа №11

Виртуальные машины

Часть 2. Проектирование виртуальной машины-интерпретатора

Задание. Разработайте виртуальную машину-интерпретатор синтаксического дерева программы. Предусмотрите три режима работы интерпретатора:

– «тихий»: отсутствует какой-либо вывод, за исключением сообщений об ошибках времени исполнения, например, «Деление на 0» или «Переменной не присвоено значение»;

– «просмотр»: при каждой модификации хранилища переменных (добавление, удаление, изменение значения) выводится его содержимое;

– «отладка»: дополнительно к предыдущей выводится информация о каждой интерпретируемой инструкции, например:

```
интерпретируется оператор if
выполняется проверка условия
вычисляется арифметическое выражение
условие истинно
интерпретируется оператор ...
```

Указания.

1. В качестве параметра для виртуальной машины-интерпретатора должно выступать ранее построенное синтаксическое дерево программы.

2. Хранилище переменных должно динамически строиться во время работы виртуальной машины-интерпретатора (с учётом области видимости переменных).

3. Вычисление арифметических выражений должно осуществляться с помощью ранее разработанной стековой виртуальной машины.

primary-expression:
    identifier
    constant
    (*expression* **)**

unary-expression:
    primary-expression
    unary-operator unary-expression

unary-operator: one of
    + – !

multiplicative-expression:
    unary-expression
    multiplicative-expression ***** *unary-expression*
    multiplicative-expression **/** *unary-expression*
    multiplicative-expression **%** *unary-expression*

additive-expression:
    multiplicative-expression
    additive-expression **+** *multiplicative-expression*
    additive-expression **–** *multiplicative-expression*

relational-expression:
    additive-expression
    relational-expression < *additive-expression*
    relational-expression > *additive-expression*
    relational-expression <= *additive-expression*
    relational-expression >= *additive-expression*

equality-expression:
    relational-expression
    equality-expression = = *relational-expression*

equality-expression **!=** *relational-expression*

assignment-expression:
conditional-expression
unary-expression assignment-operator assignment-expression

assignment-operator: one of
= *= /= %= += – =

expression:
assignment-expression