Лабораторная работа 1. Определение модели языка. Выбор инструментальной языковой среды.

Необходимо определить подмножество языка программирования (типы констант, переменных, операторов и функций). В подмножество как минимум должны быть включены:

- числовые и текстовые константы;
- 3-4 типа переменных;
- операторы цикла (**do**...**while**, **for**) ;
- условные операторы (**if**...**else, case**).

Определение инструментальной языковой среды, т.е. языка программирования и операционной системы для разработки включает:

- язык программирования с указанием версии, на котором ведётся разработка (напр. Python 3.7);
- операционная система (Windows, Linux и т.д.), в которой выполняется разработка;
- компьютер (PC / Macintosh).

В отчете по лабораторной работе дается полное определение подмножества языка программирования, тексты 2-3-х программ, включающих все элементы этого подмножества. Приводится подробное описание инструментальной языковой среды.

Лабораторная работа 2. Лексический анализ.

Освоение работы с существующими лексическими анализаторами (по желанию). Разработка лексического анализатора подмножества языка программирования, определенного в лабораторной работе 1. Определяются лексические правила. Выполняется перевод потока символов в поток лексем (токенов).

Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называемые лексемами.

На вход программы подается текстовый файл (напр. с именем INPUT.TXT), содержащий строки символов анализируемой программы. Например, строка может задавать переменной значения арифметического выражения в виде

ПЕРЕМЕННАЯ = ВЫРАЖЕНИЕ.

Выражение может включать:

Знаки сложения и умножения («+» и «*»);

Круглые скобки («(» и «)»);

Константы (например, 5; 3.8; 1e+18, 8.41E–10);

Имена переменных.

Имя переменной – это последовательность букв и цифр, начинающаяся с буквы.

Разбор выражения *COST* = (*PRICE*+*TAX*)*0.98.

Проанализируем выражение:

COST, *PRICE* и *TAX* – лексемы-идентификаторы;

0.98 – лексема-константа; =, +, * – просто лексемы.

Пусть все константы и идентификаторы можно отображать в лексемы типа <идентификатор> (<ИД>). Тогда выходом лексического анализатора будет последовательность лексем $<ИД_1>=(<ИД_2>+<ИД_3>)*<ИД_4>$.

Вторая часть компоненты лексемы (указатель, т.е. номер лексемы в таблице имен)

– показана в виде индексов. Символы «=», «+» и «*» трактуются как лексемы, тип которых представляется ими самими. Они не имеют связанных с ними данных и, следовательно, не имеют указателей.

РАБОТА С ТАБЛИЦЕЙ ИМЕН

После того, как в результате лексического анализа лексемы распознаны, информация о некоторых из них собирается и записывается в таблицу имен.

Для нашего примера *COST*, *PRICE* и *TAX* – переменные с плавающей точкой. Рассмотрим вариант такой таблицы. В ней перечислены все идентификаторы вместе с относящейся к ним информацией (табл. 1).

Табл. 1 – Таблица имен

Номер элемента	Идентификатор	Информация
1	*COST*	Переменная с плавающей точкой
2	*PRICE*	Переменная с плавающей точкой
3	*TAX*	Переменная с плавающей точкой
4	0.98	Константа с плавающей точкой

Если позднее во входной цепочке попадается идентификатор, надо справиться в этой таблице, не появлялся ли он ранее. Если да, то лексема, соответствующая новому вхождению этого идентификатора, будет той же, что и у предыдущего вхождения.

Возможно создание таблиц констант, ключевых слов, разделителей, логических/математических операторов

При определении неверной последовательности символов, необходимо обнаружить эту ошибку и выдать сообщение о ней. Всего необходимо показать скриншоты нахождения 4-х **лексических** ошибок.

Лабораторная работа 3. Синтаксический анализатор.

В ходе синтаксического анализа исходный текст программы проверяется на на соответствие синтаксическим нормам языка с построением дерево разбора (синтаксическое дерево), которое отражает синтаксическую структуру входной последовательности и удобно для дальнейшего использования, а также в случае несоответствия – позволяет вывести сообщения об ошибках.

Как правило, результатом синтаксического анализа является синтаксическое строение предложения, представленное либо в виде дерева зависимостей, либо в виде дерева составляющих, либо в виде некоторого сочетания первого и второго способов представления.

Таким образом на основе анализа выражений, состоящих из литералов, операторов и круглых скобок выполняется группирование токенов исходной программы в грамматические фразы, используемые для синтеза вывода.

Представление грамматических фраз исходной программы выполнить в виде дерева. Реализовать синтаксический анализатор с использованием одного из табличных методов (LL-, LR-метод, метод предшествования и пр.).

Разбор выражения *COST* = (*PRICE*+*TAX*)*0.98.

Выходом анализатора служит дерево, которое представляет синтаксическую структуру, присущую исходной программе.

$$<ИД_1>=(<ИД_2>+<ИД_3>)*<ИД_4>.$$

По этой цепочке необходимо выполнить следующие действия:

1) <ИД$_3$> прибавить к <ИД$_2$>;
2) результат (1) умножить на <ИД$_4$>;
3) результат (2) поместить в ячейку, резервированную для <ИД$_1$>. Этой последовательности соответствует дерево, изображенное на рис. 1.

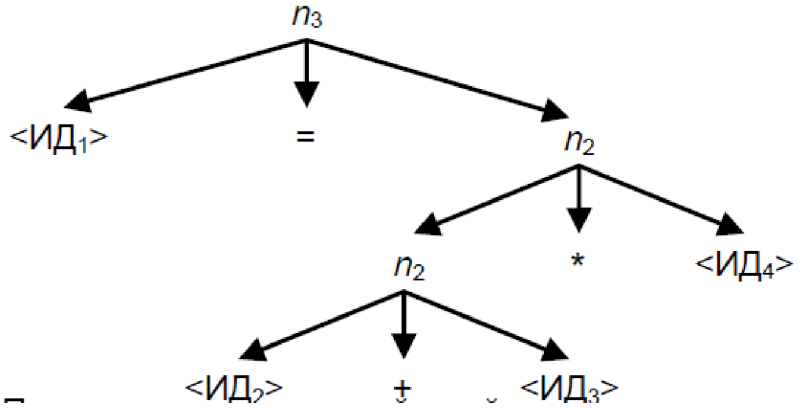

Рис. 1. – Последовательность действий при вычислении выражения

Т.е. мы имеем последовательность шагов в виде помеченного дерева.

Внутренние вершины представляют те действия, которые можно выполнять. Прямые потомки каждой вершины либо представляют аргументы, к которым нужно применять действие (если соответствующая вершина помечена идентификатором или является внутренней), либо помогают определить, каким должно быть это действие, в частности, знаки «+», «*» и «=». Скобки отсутствуют, т.к. они только определяют порядок действий.

LL- и LR- методы позволят обнаружить ошибки на самых ранних стадиях, т.е. когда разбор потока токенов от лексического анализатора в соответствии с грамматикой языка становится невозможен.

Можно использовать нисходящий (англ. top-down parser) со стартового символа, до получения требуемой последовательности токенов. Для этих целей применим метод рекурсивного спуска либо LL-анализатор. Или использовать восходящий (англ. bottom-up parser) - продукции восстанавливаются из правых частей, начиная с токенов и кончая стартовым символом - LR-анализатор и проч.

Лабораторная работа 4. Семантический анализатор.

В процессе семантического анализа проверяется наличие семантических ошибок в исходной программе и накапливается информация о типах для следующей стадии – генерации кода. При семантическом анализе используются иерархические структуры, полученные во время синтаксического анализа для идентификации операторов и

операндов выражений и инструкций.

Важным аспектом семантического анализа является проверка типов, когда компилятор проверяет, что каждый оператор имеет операнды допустимого спецификациями языка типа. Например, определение многих языков программирования требует, чтобы при использовании действительного числа в качестве индекса массива генерировалось сообщение об ошибке. В то же время спецификация языка может позволить определенное насильственное преобразование типов, например, когда бинарный арифметический оператор применяется к операндам целого и действительного типов. В этом случае компилятору может потребоваться преобразование целого числа в действительное.

В большинстве языков программирования имеет место неявное изменение типов (иногда называемое приведением типов(coercion)). Реже встречаются языки, подобные Ada, в которых большинство изменений типов должно быть явным.

В языках со статическими типами, например C, все типы известны во время компиляции, и это относится к типам выражений, идентификаторам и литералам. При этом неважно, насколько сложным является выражение: его тип может определяться во время компиляции за определенное количество шагов, исходя из типов его составляющих. Фактически, это позволяет производить контроль типов во время компиляции и находить заранее (в процессе компиляции, а не во время выполнения программы!) многие программные ошибки.

Лабораторная работа 5. Интерпретация исходного кода.

5.1. На основе результатов анализа лабораторных работ 1-4 выполнить интерпретацию программы.

5.2. Реализация интерпретатора (исходный язык для интерпретации и система получения байт-кода согласовываются).

Примерный перечень языков программирования для лабораторных работ:

C (C#/C++/Ruby C/JRuby/ANSI C).
Java/JavaScript
Python/Jython
Oracle SQL/ PL
Lisp
Perl
Smalltalk
Prolog (диалекты: Edinburgh Prolog, ISO Prolog, Strawberry Prolog)
Kotlin
F#