**Реферат на тему:**

**Языки программирования, их классификация и развитие**

**СОДЕРЖАНИЕ.**

1.Введение

 1.1. Интерпретаторы

 1.2. Компиляторы

2. Классификация языков программирования

 2.1. Машинно – ориентированные языки

 2.1.1. Машинные языки

 2.1.2. Языки символического кодирования

 2.1.3. Автокоды

 2.1.4. Макрос

 2.2. Машинно – независимые языки

 2.2.1. Машинно – независимые языки

 2.2.2. Универсальные языки

 2.2.3. Диалоговые языки

 2.2.4. Непроцедурные языки

3. Развитие языков программирования

 3.1. Ассемблер

 3.2. Лисп

3.4. Бейсик

3.5. Рефал

3.6. Пролог и Пролог++

3.7. Лекс

3.8. Си

3.8.1. Особенности языка Си

3.8.2. Недостатки языка Си

 3.9. Си++

3.9.1. Замечание по проекту языка Си++

4. Заключение

5. Библиография

#### 1. ВВЕДЕНИЕ

*Язык формирует наш способ*

 *мышления и определяет то,*

 *о чем мы можем мыслить.*

Б.Л Ворф

Прогресс компьютерных технологий определил процесс появления новых разнообразных знаковых систем для записи алгоритмов – **языков программирования**. Смысл появления такого языка – оснащенный набор вычислительных формул дополнительной информации, превращает данный набор в алгоритм.

Язык программирования служит двум связанным между собой целям: он дает программисту аппарат для задания действий, которые должны быть выполнены, и формирует концепции, которыми пользуется программист, размышляя о том, что делать. Первой цели идеально отвечает язык, который настолько "близок к машине", что всеми основными машинными аспектами можно легко и просто оперировать достаточно очевидным для программиста образом. Второй цели идеально отвечает язык, который настолько "близок к решаемой задаче", чтобы концепции ее решения можно было выражать прямо и коротко.

Связь между языком, на котором мы думаем/программируем, и задачами и решениями, которые мы можем представлять в своем воображении, очень близка. По этой причине ограничивать свойства языка только целями исключения ошибок программиста в лучшем случае опасно. Как и в случае с естественными языками, есть огромная польза быть, по крайней мере, двуязычным. Язык предоставляет программисту набор концептуальных инструментов, если они не отвечают задаче, то их просто игнорируют. Например, серьезные ограничения концепции указателя заставляют программиста применять вектора и целую арифметику, чтобы реализовать структуры, указатели и т.п. Хорошее проектирование и отсутствие ошибок не может гарантироваться чисто за счет языковых средств.

Может показаться удивительным, но конкретный компьютер способен работать с программами, написанными на его родном машинном языке. Существует почти столько же разных машинных языков, сколько и компьютеров, но все они суть разновидности одной идей простые операции производятся со скоростью молнии на двоичных числах.

Персональные компьютеры **IBM** используют машинный язык микропроцессоров семейства 8086, т.к. их аппаратная часть основывается именно на данных микропроцессорах.

Можно писать программы непосредственно на машинном языке, хотя это и сложно. На заре компьютеризации(в начале 1950-х г.г.), машинный язык был единственным языком, большего человек к тому времени не придумал. Для спасения программистов от сурового машинного языка программирования, были созданы **языки высокого уровня** (т.е. немашинные языки), которые стали своеобразным связующим мостом между человеком и машинным языком компьютера. Языки высокого уровня работают через трансляционные программы, которые вводят "исходный код" (гибрид английских слов и математических выражений, который считывает машина), и в конечном итоге заставляет компьютер выполнять соответствующие команды, которые даются на машинном языке. Существует два основных вида трансляторов: интерпретаторы, которые сканируют и проверяют исходный код в один шаг, и компиляторы, которые сканируют исходный код для производства текста программы на машинном языке, которая затем выполняется отдельно.

**1.1. Интерпретаторы**

Одно, часто упоминаемое преимущество интерпретаторной реализации состоит в том, что она допускает "непосредственный режим". Непосредственный режим позволяет вам задавать компьютеру задачу вроде PRINT 3.14159\*3/2.1 и возвращает вам ответ, как только вы нажмете клавишу ENTER (это позволяет использовать компьютер стоимостью 3000 долларов в качестве калькулятора стоимостью 10 долларов). Кроме того, интерпретаторы имеют специальные атрибуты, которые упрощают отладку. Можно, например, прервать обработку интерпретаторной программы, отобразить содержимое определенных переменных, бегло просмотреть программу, а затем продолжить исполнение.

Больше всего программистам нравится в интерпретаторах возможность получения быстрого ответа. Здесь нет необходимости в компилировании, так как интерпретатор всегда готов для вмешательства в вашу программу. Введите RUN и результат вашего самого последнего изменения оказывается на экране.

Однако интерпретаторные языки имеют недостатки. Необходимо, например, иметь копию интерпретатора в памяти все время, тогда как многие возможности интерпретатора, а следовательно и его возможности могут не быть необходимыми для исполнения конкретной программы.

Слабо различимым недостатком интерпретаторов является то, что они имеют тенденцию отбивать охоту к хорошему стилю программирования. Поскольку комментарии и другие формализуемые детали занимают значительное место программной памяти, люди стремятся ими не пользоваться. Дьявол менее яростен, чем программист, работающий на интерпретаторном Бейсике, пытающийся получить программу в 120К в памяти емкостью 60К. но хуже всего то, что интерпретаторы тихоходны. Ими затрачивается слишком много времени на разгадывание того, что делать, вместо того чтобы заниматься действительно делом.

При исполнении программных операторов, интерпретатор должен сначала сканировать каждый оператор с целью прочтения его содержимого (что этот человек просит меня сделать?), а затем выполнить запрошенную операцию. Операторы в циклах сканируются излишне много.

Рассмотрим программу: на интерпретаторном Бэйсике 10 FOR N=1 TO 1000 20 PRINT N,SQR(N) 30

NEXT N при первом переходе по этой программе Бейсик-Интерпретатор должен разгадать что означает строка 20:

 1.преобразовать числовую переменную N в строку

 2.послать строку на экран

 3.переместить в следующую зону печати

 4.вычислить квадратный корень из N

 5.преобразовать результат в строку

 6.послать строку на экран

При втором проходе цикла все это разгадывание повторяется снова, так как абсолютно забыты все результаты изучения этой строки какую-то миллисекунду тому назад. И так во всех следующих 998 проходах. Совершенно очевидно, что если вам удалось каким-то образом отделить фазу сканирования/понимания от фазы исполнения вы имели бы более быструю программу. И это как раз то, для чего существуют компиляторы.

**1.2. Компиляторы**

Компилятор-это транслятор текста на машинный язык, который считывает исходный текст. Он оценивает его в соответствии с синтаксической конструкцией языка и переводит на машинный язык. Другими словами, компилятор не исполняет программы, он их строит. Интерпретаторы невозможно отделить от программ, которые ими прогоняются, компиляторы делают свое дело и уходят со сцены. При работе с компилирующим языком, таким как Турбо-Бейсик, вы придете к необходимости мыслить о ваших программах в признаках двух главных фаз их жизни: периода компилирования и периода прогона. Большинство программ будут прогоняться в четыре - десять раз быстрее их интерпретаторных эквивалентов. Если вы поработаете над улучшением, то сможете достичь 100-кратного повышения быстродействия. Оборотная сторона монеты состоит в том, что программы, расходующие большую часть времени на возню с файлами на дисках или ожидание ввода, не смогут продемонстрировать какое-то впечатляющее увеличение скорости.

**2. КЛАССИФИКАЦИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ**

**2.1. Машинно – ориентированные языки**

 **Машинно – ориентированные языки** – это языки, наборы операторов и изобразительные средства которых существенно зависят от особенностей ЭВМ (внутреннего языка, структуры памяти и т.д.). Машинно –ориентированные языки позволяют использовать все возможности и особенности Машинно – зависимых языков:

* высокое качество создаваемых программ (компактность и скорость выполнения);
* возможность использования конкретных аппаратных ресурсов;
* предсказуемость объектного кода и заказов памяти;
* для составления эффективных программ необходимо знать систему команд и особенности функционирования данной ЭВМ;
* трудоемкость процесса составления программ ( особенно на машинных языках и ЯСК), плохо защищенного от появления ошибок;
* низкая скорость программирования;
* невозможность непосредственного использования программ, составленных на этих языках, на ЭВМ других типов.

Машинно-ориентированные языки по степени автоматического программирования подразделяются на классы.

**2.1.1. Машинный язык**

Как я уже упоминал, в введении, отдельный компьютер имеет свой определенный **Машинный язык** (далее **МЯ**), ему предписывают выполнение указываемых операций над определяемыми ими операндами, поэтому **МЯ** является командным. Однако, некоторые семейства ЭВМ (например, ЕС ЭВМ, IBM/370/ и др.) имеют единый **МЯ** для ЭВМ разной мощности. В команде любого из них сообщается информация о местонахождении операндов и типе выполняемой операции.

В новых моднлях ЭВМ намечается тенденция к повышению внутренних языков машинно – аппаратным путем реализовывать более сложные команды, приближающиеся по своим функциональным действиям к операторам алгоритмических языков программирования.

* + 1. **Языки Символического Кодирования**

Продолжим рассказ о командных языках, **Языки Символического Кодирования** (далее **ЯСК**), так же, как и **МЯ**, являются командными. Однако коды операций и адреса в машинных командах, представляющие собой последовательность двоичных (во внутреннем коде) или восьмеричных (часто используемых при написании программ) цифр, в **ЯСК** заменены на символы (идентификаторы), форма написания которых помогает программисту легче запоминать смысловое содержание операции. Это обеспечивает существенное уменьшение числа ошибок при составлении программ.

Использование символических адресов – первый шаг к созданию **ЯСК**. Команды ЭВМ вместо истинных (физических) адресов содержат символические адреса. По результатам составленной программы определяется требуемое количество ячеек для хранения исходных промежуточных и результирующих значений. Назначение адресов, выполняемое отдельно от составления программы в символических адресах, может проводиться менее квалифицированным программистом или специальной программой, что в значительной степени облегчает труд программиста.

* + 1. **Автокоды**

Есть также языки, включающие в себя все возможности **ЯСК**, посредством расширенного введения ***макрокоманд* -** они называются **Автокоды**.

В различных программах встречаются некоторые достаточно часто использующиеся командные последовательности, которые соответствуют определенным процедурам преобразования информации. Эффективная реализация таких процедур обеспечивается оформлением их в виде специальных макрокоманд и включением последних в язык программирования , доступный программисту. Макрокоманды переводятся в машинные команды двумя путями – *расстановкой* и *генерированием*. В постановочной системе содержатся «остовы» - серии команд, реализующих требуемую функцию, обозначенную макрокомандой. Макрокоманды обеспечивают передачу фактических параметров, которые в процессе трансляции вставляются в «остов» программы, превращая её в реальную машинную программу.

В системе с генерацией имеются специальные программы, анализирующие макрокоманду, которые определяют, какую функцию необходимо выполнить и формируют необходимую последовательность команд, реализующих данную функцию.

Обе указанных системы используют трансляторы с **ЯСК** и набор макрокоманд, которые также являются операторами автокода.

Развитые автокоды получили название **Ассемблер**ы. Сервисные программы и пр., как правило, составлены на языках типа **Ассемблер**. Более полная информация об языке **Ассемблер**а см. ниже.

* + 1. **Макрос**

Язык, являющийся средством для замены последовательности символов описывающих выполнение требуемых действий ЭВМ на более сжатую форму - называется **Макрос** (средство замены).

В основном, **Макрос** предназначен для того, чтобы сократить запись исходной программы. Компонент программного обеспечения, обеспечивающий функционирование макросов, называется **макропроцессором**. На макропроцессор поступает макроопределяющий и исходный текст. Реакция макропроцессора на вызов-выдача выходного текста.

**Макрос** одинаково может работать, как с программами, так и с данными.

* 1. **Машинно – независимые языки**

**Машинно – независимые языки** – это средство описания алгоритмов решения задач и информации, подлежащей обработке. Они удобны в использовании для широкого круга пользователей и не требуют от них знания особенностей организации функционирования ЭВМ и ВС.

Подобные языки получили название высокоуровневых языков программирования. Программы, составляемые на таких языках, представляют собой последовательности операторов, структурированные согласно правилам рассматривания языка(задачи, сегменты, блоки и т.д.). Операторы языка описывают действия, которые должна выполнять система после трансляции программы на **МЯ**.

Т.о., командные последовательности (процедуры, подпрограммы), часто используемые в машинных программах, представлены в высокоуровневых языках отдельными операторами. Программист получил возможность не расписывать в деталях вычислительный процесс на уровне машинных команд, а сосредоточиться на основных особенностях алгоритма.

**2.2.1. Проблемно – ориентированные языки**

С расширением областей применения вычислительной техники возникла необходимость формализовать представление постановки и решение новых классов задач. Необходимо было создать такие языки программирования, которые, используя в данной области обозначения и терминологию, позволили бы описывать требуемые алгоритмы решения для поставленных задач, ими стали **проблемно – ориентированные языки**. Эти языки, языки ориентированные на решение определенных проблем, должны обеспечить программиста средствами, позволяющими коротко и четко формулировать задачу и получать результаты в требуемой форме.

Проблемных языков очень много, например:

**Фортран, Алгол** – языки, созданные для решения математических задач;

**Simula, Слэнг** - для моделирования;

**Лисп, Снобол** – для работы со списочными структурами.

Об этих языках я расскажу дальше.

* + 1. **Универсальные языки**

**Универсальные языки** были созданы для широкого круга задач: коммерческих, научных, моделирования и т.д. Первый универсальный язык был разработан фирмой IBM, ставший в последовательности языков **Пл/1**. Второй по мощности универсальный язык называется **Алгол-68**. Он позволяет работать с символами, разрядами, числами с фиксированной и плавающей запятой. **Пл/1** имеет развитую систему операторов для управления форматами, для работы с полями переменной длины, с данными организованными в сложные структуры, и для эффективного использования каналов связи. Язык учитывает включенные во многие машины возможности прерывания и имеет соответствующие операторы. Предусмотрена возможность параллельного выполнение участков программ.

Программы в **Пл/1** компилируются с помощью автоматических процедур. Язык использует многие свойства **Фортрана, Алгола, Кобола**. Однако он допускает не только динамическое, но и управляемое и статистическое распределения памяти.

**2.2.3. Диалоговые языки**

Появление новых технических возможностей поставило задачу перед системными программистами – создать программные средства, обеспечивающие оперативное взаимодействие человека с ЭВМ их назвали **диалоговыми языками**.

Эти работы велись в двух направлениях. Создавались специальные управляющие языки для обеспечения оперативного воздействия на прохождение задач, которые составлялись на любых раннее неразработанных (не диалоговых) языках. Разрабатывались также языки, которые кроме целей управления обеспечивали бы описание алгоритмов решения задач.

Необходимость обеспечения оперативного взаимодействия с пользователем потребовала сохранения в памяти ЭВМ копии исходной программы даже после получения объектной программы в машинных кодах. При внесении изменений в программу с использованием диалогового языка система программирования с помощью специальных таблиц устанавливает взаимосвязь структур исходной и объектной программ. Это позволяет осуществить требуемые редакционные изменения в объектной программе.

 Одним из примеров диалоговых языков является **Бэйсик**.

**Бэйсик** использует обозначения подобные обычным математическим выражениям. Многие операторы являются упрощенными вариантами операторов языка **Фортран**. Поэтому этот язык позволяет решать достаточно широкий круг задач.

**2.2.4. Непроцедурные языки**

**Непроцедурные языки** составляют группу языков, описывающих организацию данных, обрабатываемых по фиксированным алгоритмам (табличные языки и генераторы отчетов), и языков связи с операционными системами.

Позволяя четко описывать как задачу, так и необходимые для её решения действия, таблицы решений дают возможность в наглядной форме определить, какие условия должны быть выполнены прежде чем переходить к какому-либо действию. Одна таблица решений, описывающая некоторую ситуацию, содержит все возможные блок-схемы реализаций алгоритмов решения.

Табличные методы легко осваиваются специалистами любых профессий.

Программы, составленные на табличном языке, удобно описывают сложные ситуации, возникающие при системном анализе.

1. **РАЗВИТИЕ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ**
	1. **Ассемблер**

Язык **Ассемблер**а – это символическое представление машинного языка. Он облегчает процесс программирования по сравнению с программированием в машинных кодах.

Программисту не обязательно употреблять настоящие адреса ячеек памяти с размещенными в них данными, участвующими в операции, и вычисляемые результаты, а также адреса тех команд, к которым программа не обращается.

Некоторые задачи, например, обмен с нестандартными устройствами обработки данных сложных структур невозможно решить с помощью языков программирования высокого уровня. Это под силу ассемблеру.

В принципе, язык **Ассемблер** является машинным языком. И программист реализующий какую-либо задачу на языках высокого уровня, с помощью **Ассемблер**а может определить осмыслено ли решение данной задачи, с точки зрения использования ЭВМ.

Умея разобраться в распечатке языка ассемблера, дает возможность облегчить поиск ошибок в программах, т.к. некоторые языки являются компиляторами (см. п. 1.2.).

###  Лисп

## Один из самых старых языков программирования Фортран был создан в 50-х гг. нашего века. **Фортран** и подобные ему языки программирования **(Алгол, ПЛ/1)** предназначались для решения вычислительных задач, возникающих в математике, физике, инженерных расчетах, экономике и т.п. Эти языки в основном работают с числами.

Второй старейший язык программирования **Лисп** (**L**ist **I**nformation **S**ymbol **P**rocessing), Дж. Маккарти в 1962 г. скорее для работы со строками символов, нежели для работы с числами. Это особое предназначение **Лисп**а открыло для программистов новую область деятельности, известную ныне, как «искусственный интеллект». В настоящее время **Лисп** успешно применяется в экспертных системах, системах аналитических вычислений и т.п.

Обширность области возможных приложений **Лисп**а вызвала появление множества различных диалектов **Лисп**а. Это легко объяснимо: применение **Лисп**а для понимания естественного языка требует определенного набора базисных функций, отличных, например, от используемого в задачах медицинской диагностики.

Существование множества различных диалектов **Лисп**а привело к созданию в начале 80-х гг. Common **LISP** Комитета, который должен был выбрать наиболее подходящий диалект **Лисп**а и предложить его в качестве основного. Этот диалект, выбранный Комитетом в 1985г., получил название Common **LISP** . В дальнейшем он был принят в университетах США, а также многими разработчиками систем искусственного интеллекта, в качестве основного диалекта языка **Лисп**.

Язык программирования **Лисп** существенно отличается от других языков программирования, таких, как **Паскаль**, **Си** и т.п. Работа с символами и работа с числами как с основными элементами требует разных способов мышления.

Первоначально **Лисп** был задуман как теоретическое средство для рекурсивных построений, а сегодня он превратился в мощное средство, обеспечивающее программиста разнообразной поддержкой, позволяющей ему быстро строить прототипы весьма и весьма серьезных систем.

Профессор Массачусетского технологического института Дж. Самман заметил, что математическая ясность и предельная четкость **Лисп**а – это еще не все. Главное – **Лисп** позволяет сформулировать и запомнить «идиомы», столь характерные для проектов по искусственному интеллекту.

**3.3. Фортран**

Одним из первых и наиболее удачных компиляторов стал язык **Фортран**, разработанный фирмой IBM. Профессор Дж. Букс и группа американских специалистов в области программирования в 1954 году опубликовало первое сообщение о языке. Дословно, название языка **FOR***mulae* **TRA*N****slation* –преобразование формул.

Среди причин долголетия **Фортрана** (а он один из самых распространенных языков в мире), можно отметить простую структуру, как самого Фортрана, так и предназначенных для него трансляторов. Программа на Фортране записывается в последовательности предложений или операторов (описание некоего преобразования информации), и оформляется по определенным стандартам. Эти стандарты накладывают ограничения, в частности, на форму записи и расположения частей оператора в строке бланка для записи операторов. Программа, записанная на Фортране, представляет собой один или несколько сегментов (подпрограмм) из операторов. Сегмент, управляющий работой всей программы в целом, называется основной программой.

**Фортран** был задуман для использования в сфере научных и инженерно-технических вычислений. Однако на этом языке легко описываются задачи с разветвленной логикой (моделирование производственных процессов, решение игровых ситуаций и т.д.), некоторые экономические задачи и особенно задачи редактирования (составление таблиц, сводок, ведомостей и т.д.).

Модификация языка **Фортран**, появившиеся в 1958 году, получила название **Фортран II** и содержала понятие подпрограммы и общих переменных для обеспечения связи между сегментами.

К 1962 году относится появление языка, известного под именем **Фортран IV** и ставшего наиболее употребительным в настоящее время. К этому же времени относится и начало деятельности комиссии при Американской Ассоциации Стандартов **(ASA)**, которая выработала к 1966 году два стандарта – языки **Фортран** и базисный (основной) **Фортран** **(Basic FORTRAN).** Эти языки приблизительно соответствуют модификациям **IV** и **II**, однако базисный Фортран является подмножеством Фортрана, в то время, как **Фортран II** таковым для **Фортрана IV** не является. Язык **Фортран** до сих пор продолжает развиваться и совершенствоваться, оказывая влияние на создание и развитие других языков. Например, **Фортран** заложен в основу Basic – диалогового языка, очень популярного для решения небольших задач, превосходного языка для обучения навыкам использования алгоритмических языков в практике программирования. Разработан этот язык – Beginner’s All –purpose Symbolic Instruction Code – группой сотрудников Вычислительного центра Дармутского колледжа, штат Нью-Хемпшир созданный в 19…. . Но это уже следующий язык.

**3.4. Бейсик**

Как знаменитые гамбургеры, бейсбол и баскетбол, **Бейсик** - это продукт Новой Англии. Как я говорил, созданный в 1964г., как язык обучения программированию. **Бейсик** является общепринятым акронимом от"**B**eginner's **A**ll-purpose **S**ymbolic **I**nsruction **C**ode" **(BASIC)** - **М**ногоцелевой **С**имволический **О**бучающий **К**од для **Н**ачинающих".

Вскоре как обучаемые, так и авторы программ обнаружили, что **Бейсик** может делать практически все то, что делает скучный неуклюжий **Фортран**. А так как **Бейсик**у было легко обучиться и легко с ним работать, программы на нем писались обычно быстрее, чем на **Фортран**е. **Бейсик** был также доступен на персональных компьютерах, обычно он встроен в ПЗУ. Так **Бейсик** завоевал популярность. Интересно, что спустя 20 лет после изобретения **Бейсик**а, он и сегодня самый простой для освоения из десятков языков общецелевого программирования, имеющихся в распоряжении любителей программирования. Более того, он прекрасно справляется с работой.

Несмотря на высказывания снобов - сторонников языков **Си** и **Паскаля**, **Бейсик** считается деловым языком, снабженным мощными средствами решения специфических задач, которые обычно большинство пользователей решают при помощи небольших компьютеров, а именно: работая с файлами и выводя текстовое и графическое изображение на экране дисплея.

Несмотря на отдельные недостатки **Бейсик**а, никто не будет отрицать, что Кемени и Куртс достигли основной цели: сделать программирование доступнее для большего числа людей.

Исторически **Бейсик** обычно реализовался как интерпретатор (знакомым изомером является сам интерпретаторный **Бейсик**). Причинами перехода от любительского уровня к профессиональному являются многочисленные расширения классической версии языка: возможность отключения нумерации строк, многостроковые структурированные программные конструкции, структуры типа "запись", поименованные подпрограммы с параметрами и локальные переменные.

Более того, с появлением транслятора *QuickBasic* фирмы ***Microsoft*** разработчики получили возможность строить на **Бейсике** приложения из раздельно откомпилированных модулей, некоторые из которых могут быть написаны на других языках. Теперь, как и в случае других ведущих языков программирования, разработчик имеет выбор из нескольких промышленных библиотек подпрограмм, которые содержат готовые решения для распространенных задач программирования.

* 1. **Рефал**

Несомненно надо рассказать и о некоторых языках программирования созданных у нас на родине. Один из таких языков является **Рефал**, разработанный у нас в России (СССР), в 1966г. ИПМ АН СССР. Этот язык прост и удобен для описания манипуляций над произвольными текстовыми объектами.

**Рефал** широко применяется при разработке трансляторов с алгоритмических языков как универсальных и проблемно – ориентированных, так и автокодов. Кроме использования в задачах трансляции, **Рефал** имеет такие важные сферы применения, как машинное выполнение громоздких аналитических выкладок в теоретической физике и прикладной математике; проектирование «умных» информационных систем, осуществляющих нетривиальную логическую обработку информации; машинное доказательство теорем; моделирование целенаправленного поведения; разработка диалоговых обучающих систем; исследования в области искусственного интеллекта и т.п.

Программирование на **Рефал**е имеет специфику, связанную, прежде всего, с тем, что **Рефал** является языком функционального типа в отличие от обычных операторных языков типа **Алгол**, **Фортран** и т.д.. Если программа на операторных языках – ни что иное, как совокупность приказов-операторов, то программа на **Рефал**е представляет собой по существу описание связей и отношений между определенными понятиями.

Вследствие того, что в **Рефале** программист сам определяет структуру обрабатываемой информации, эффективность программы существенно зависит от удачного или неудачного выбора этой структуры. Для задания структур в **Рефал**е используются скобки, а специфика всех реализаций языка такова, что использование скобок резко повышает эффективность выполнения программы. Это достигается с помощью адресного соединения скобок.

Определенной спецификой обладают и переменные типа «выражения» – имеется в виду их способность удлиняться при отождествлении. Правильное использование переменных этого типа также позволяет значительно повысить эффективность **Рефал** – программы.

**3.6. Пролог и Пролог ++**

**Пролог** - это язык, предназначенный для поиска решений. Это декларативный язык, то есть формальная постановка задачи может быть использована для ее решения. **Пролог** определяет логические отношения в задаче, как отличные от пошагового решения этой задачи.

Центральной частью **Пролога** являются средства логического вывода, которые решают запросы, используя заданное множество фактов и правил, к которым обращаются как к утверждениям. **Пролог** также не имеет деления переменных на типы и может динамически добавлять правила и факты к средствам вывода. Таким образом, это гибкий язык, и он более пригоден для объектно-ориентированного расширения, чем язык со строго заданными типами, например, **Паскаль**. **Пролог** **++** представляет собой дополнение к стандартному **Пролог**у.

 Все свойства языка по-прежнему доступны программистам. Следовательно, **Пролог** **++** можно отнести к группе гибридных языков, представителями которой считаются **Object Pascal** и **C++**. Расширение **Пролог** ++ поддерживает все свойства, присущие обычно объектно-ориентированным языкам: концепции объектов и классов, единичное и многократное наследование, разбиение на подклассы и передачу сообщений. Поддерживаются также некоторые усовершенствованные свойства, существующие в таких языках, как **C++** и **Smalltalk**, включая общие и частные методы.

Интересным свойством является поддержка в языке программирования с управлением данными. Эта техника, которая может быть еще названа программированием, "управляемым событиями", используется в большинстве языков объектно-ориентированного программирования, особенно в тех, которые разработаны для машин с интерфейсом, управляемым "мышью".

 Объектно-ориентированная программа реагирует на события, которые определяют поток управления. В **Пролог**е **++** программирование с управлением данными достигается при помощи концепции **демонов**. Демон представляет собой объект, методы которого вызываются в случае определенных событий и могут быть таким образом использованы для поддержки программирования с управлением данными.

 Сам язык основан на концепции передачи сообщений. Программа на **Пролог**е **++** строится вокруг множества объектов **Пролог**а **++**, которые обмениваются сообщениями. В этом смысле **Пролог** **++** ближе к чистому объектно-ориентированному языку, такому, как **Smalltalk**, чем **C++** или **Object Pascal**. Определения объектов строятся исходя из вызовов

 *Open\_Object [имя\_объекта]* и *Close\_Object [имя\_объекта]*, а методы определяются практически так же, как в других объектно-ориентированных языках. Для задания наследования можно явным образом указать, какой метод какого объекта должен наследоваться, что является необходимым для многократного наследования.

* 1. **Лекс**

**Лекс** – генератор программ лексического анализа. Лексический анализ – это распознавание лексем во входном потоке символов. Предположим, что задано некоторое конечное множество слов (лексем) в некотором языке и некоторое входное слово. Необходимо установить, какой элемент множества (если он существует) совпадает с данным входным словом. Обычно лексический анализ выполняется так называемым лексическим анализатором. Лексический анализатор – это программа. Лексический анализ применяется во многих случаях, например, для построения пакетного редактора или в качестве распознавателя директив в диалоговой программе и т.д. Однако, наиболее важное применение лексического анализатора – это использование его в компиляторе. Здесь лексический анализатор выполняет функцию программы ввода данных.

Лексический анализатор выполняет первую стадию компиляции – читает строки компилируемой программы, выделяет лексемы и передает их на дальнейшие стадии компиляции (грамматический разбор, кодогенерацию и т.д.).

Лексический анализатор распознает тип каждой лексемы и соответствующим образом помечает ее. Например, при компиляции Си-программы могут быть выделены следующие типы лексем: число, идентификатор, оператор, ограничитель и т.д.

Лексический анализатор должен не только выделить лексему, но и выполнить некоторые преобразования. Например, если лексема – число, то его необходимо перевести во внутреннюю (двоичную) форму записи как число с плавающей или фиксированной запятой. А если лексема – идентификатор, то его необходимо разместить в таблице, чтобы в дальнейшем обращаться к нему не по имени, а по адресу в таблице.

Хотя лексический анализ по своей идее прост, тем не менее, эта фаза работы компилятора часто занимает больше времени, чем любая другая. Частично это происходит из-за необходимости просматривать и анализировать исходный текст символ за символом. Иногда даже бывает необходимо вернуть прочитанный символ во входной поток с тем, чтобы повторить просмотр и анализ.

**3.8. Cи**

**Си** – это язык программирования общего назначения, хорошо известный своей эффективностью, экономичностью, и переносимостью. Указанные преимущества **Си** обеспечивают хорошее качество разработки почти любого вида программного продукта. Использование **Си** в качестве инструментального языка позволяет получать быстрые и компактные программы. Во многих случаях программы, написанные на **Си**, сравнимы по скорости с программами, написанными на языке ассемблера. При этом они имеют лучшую наглядность и их более просто сопровождать. **Си** сочетает эффективность и мощность в относительно малом по размеру языке.

**Си –** это замечательный язык, и хотя некоторым он не нравится, но все же большинство программистов его любят. На **Си** вы можете создавать программы, которые делают все, что вы пожелаете. Нет другого такого языка, который бы так же стимулировал к программированию. Создается впечатление, что остальные языки программирования воздвигают искусственные препятствия для творчества, а **Си** – нет. Использование этого языка позволяет сократить затраты времени на создание работающих программ. **Си** позволяет программировать быстро, эффективно и предсказуемо. Еще одно преимущество **Си** заключается в том, что он позволяет использовать все возможности вашей ЭВМ. Этот язык создан программистом для использования другими программистами, чего о других языках программирования сказать нельзя.

* + 1. **Особенности языка Си**

Язык **Си** имеет свои существенные особенности, давайте перечислим некоторые из них:

**Си** обеспечивает полный набор операторов структурного программирования. **Си** предлагает необычно большой набор операций. Многие операции **Си** соответствуют машинным командам, и поэтому допускают прямую трансляцию в машинный код. Разнообразие операций позволяет выбирать их различные наборы для минимизации результирующего кода.

**Си** поддерживает указатели на переменные и функции. Указатель на объект программы соответствует машинному адресу этого объекта. Посредством разумного использования указателей можно создавать эффективно-выполняемые программы, так как указатели позволяют ссылаться на объекты тем же самым путем, как это делает машина. **Си** поддерживает арифметику указателей, и тем самым позволяет осуществлять непосредственный доступ и манипуляции с адресами памяти.

В своем составе **Си** содержит препроцессор, который обрабатывает текстовые файлы перед компиляцией. Среди его наиболее полезных приложений при написании программ на **Си** являются: определение программных констант, замена вызовов функций аналогичными, но более быстрыми макросами, условная компиляция. Препроцессор не ограничен процессированием только исходных текстовых файлов **Си**, он может быть использован для любого текстового файла.

**Си**-гибкий язык, позволяющий принимать в конкретных ситуациях самые разные решения. Тем не менее, **Си** налагает незначительные ограничения в таких, например, действиях, как преобразование типов. Во многих случаях это является достоинством, однако программисты должны хорошо знать язык, чтобы понимать, как будут выполняться их программы.

* + 1. **Недостатки языка Си**:

Также, как и особенностей, в языке **Си** присутствует куча недостатков. Ведь от них не защищен не один проект, в том числе проект разработки и выполнения программ, на языке **Си:**

Язык Си предъявляет достаточно высокие требования к квалификации использующего его программиста. При изучении **Си** желательно иметь представление о структуре и работе компьютера. Большую помощь и более глубокое понимание идей **Си**, как языка системного программирования, обеспечат хотя бы минимальное знание языка ассемблер. Уровень старшинства некоторых операторов не является общепринятым, некоторые синтаксические конструкции могли бы быть лучше. Тем не менее, как оказалось **Си** – чрезвычайно эффективный и выразительный язык, пригодный для широкого класса задач.

* 1. **Си++**

Безусловно, **Си++** восходит, главным образом, к **Cи**. **Cи** сохранен как подмножество, поэтому сделанного в **Cи** акцента на средствах низкого уровня достаточно, чтобы справляться с самыми насущными задачами системного программирования. **Cи**, в свою очередь, многим обязан своему предшественнику BCPL.

Название **Си++** - изобретение лета 1983-его. Более ранние версии языка использовались начиная с 1980-ого и были известны как "*Cи с Классами*". Первоначально язык был придуман потому, что автор хотел написать событийно управляемые модели для чего был бы идеален **Simula67**, если не принимать во внимание эффективность. "*Cи с Классами*" использовался для крупных проектов моделирования, в которых строго тестировались возможности написания программ, требующих (только) минимального пространства памяти и времени на выполнение. В "*Cи с Классами*" не хватало перегрузки операций, ссылок, виртуальных функций и многих деталей. **Си++** был впервые введен за пределами исследовательской группы автора в июле 1983-го. Однако тогда многие особенности **Си++** были еще не придуманы.

Название **Си++** выдумал Рик Масситти. Название указывает на эволюционную природу перехода к нему от **Cи**. "**++**" - это операция приращения в **Cи**. Чуть более короткое имя **Cи**+ является синтаксической ошибкой, кроме того, оно уже было использовано как имя совсем другого языка. Знатоки семантики **Cи** находят, что **Си++** хуже, чем **Cи** ++. Названия **D** язык не получил, поскольку он является расширением **Cи** и в нем не делается попыток исцелиться от проблем путем выбрасывания различных особенностей.

**Си++** - это универсальный язык программирования, задуманный так, чтобы сделать программирование более приятным для серьезного программиста. За исключением второстепенных деталей **Си++** является надмножеством языка программирования **Cи**. Помимо возможностей, которые дает **Cи**, **Си++** предоставляет гибкие и эффективные средства определения новых типов. Используя определения новых типов, точно отвечающих концепциям приложения, программист может разделять разрабатываемую программу на легко поддающиеся контролю части. Такой метод построения программ часто называют абстракцией данных. Информация о типах содержится в некоторых объектах типов, определенных пользователем. Такие объекты просты и надежны в использовании в тех ситуациях, когда их тип нельзя установить на стадии компиляции. Программирование с применением таких объектов часто называют объектно-ориентированным. При правильном использовании этот метод дает более короткие, проще понимаемые и легче контролируемые программы.

Изначально **Си++** был разработан, чтобы автору и его друзьям не приходилось программировать на ассемблере, **Cи** или других современных языках высокого уровня. Основным его предназначением было сделать написание хороших программ более простым и приятным для отдельного программиста. Плана разработки **Си++** на бумаге никогда не было. Проект, документация и реализация двигались одновременно. Разумеется, внешний интерфейс **Си++** был написан на **Си**++. Никогда не существовало "Проекта **Си**++" и "Комитета по разработке **Си**++". Поэтому **Си++** развивался и продолжает развиваться во всех направлениях, чтобы справляться со сложностями, с которыми сталкиваются пользователи, а также в процессе дискуссий автора с его друзьями и коллегами.

В качестве базового языка для **Си++** был выбран **Cи**, потому что он:

* многоцелевой, лаконичный и относительно низкого уровня:
* отвечает большинству задач системного программирования:
* идет везде и на всем:
* пригоден в среде программирования **UNIX.**

В **Cи** есть свои сложности, но в наспех спроектированном языке тоже были бы свои, а сложности **Cи** нам известны. Самое главное, работа с **Cи** позволила "*Cи с Классами*" быть полезным (правда, неудобным) инструментом в ходе первых месяцев раздумий о добавлении к **Cи** ***Simula***-подобных классов.

 **Си++** стал использоваться шире, и по мере того, как возможности, предоставляемые им помимо возможностей **Cи**, становились все более существенными, вновь и вновь поднимался вопрос о том, сохранять ли совместимость с **Cи**. Ясно, что отказавшись от определенной части наследия **Cи** можно было бы избежать ряда проблем. Это не было сделано, потому что:

* есть миллионы строк на **Cи**, которые могли бы принести пользу в **Си++** при условии, что их не нужно было бы полностью переписывать с **Cи** на **Си**++;
* есть сотни тысяч строк библиотечных функций и сервисных программ, написанных на **Cи** которые можно было бы использовать из или на **Си++** при условии, что **Си++** полностью совместим с **Cи** по загрузке и синтаксически очень похож на **Cи**;
* есть десятки тысяч программистов, которые знают **Cи**, и которым, поэтому, нужно только научиться использовать новые особенности **Си**++, а не заново изучать его основы;
* поскольку **Си++** и **Cи** будут использоваться на одних и тех же системах одними и теми же людьми, отличия должны быть либо очень большими, либо очень маленькими, чтобы свести к минимуму ошибки и недоразумения.

Позднее была проведена проверка определения **Си**++, чтобы удостовериться в том, что любая конструкция, допустимая и в **Cи,** и в **Си++**, действительно означает в обоих языках одно и то же.

 **Си++** был развит из языка программирования **Cи** и за очень немногими исключениями сохраняет **Cи** как подмножество. Базовый язык, **Cи** подмножество **Си++,** спроектирован так, что имеется очень близкое соответствие между его типами, операциями и операторами и компьютерными объектами, с которыми непосредственно приходится иметь дело: числами, символами и адресами. За исключением операций свободной памяти new и delete, отдельные выражения и операторы **Си++** обычно не нуждаются в скрытой поддержке во время выполнения или подпрограммах.

Одним из первоначальных предназначений **Cи** было применение его вместо программирования на ассемблере в самых насущных задачах системного программирования. Когда проектировался **Си++,** были приняты меры, чтобы не ставить под угрозу успехи в этой области. Различие между **Cи** и **Си++** состоит в первую очередь в степени внимания, уделяемого типам и структурам. **Cи** выразителен и снисходителен. **Си++** еще более выразителен, но чтобы достичь этой выразительности, программист должен уделить больше внимания типам объектов. Когда известны типы объектов, компилятор может правильно обрабатывать выражения, тогда как в противном случае программисту пришлось бы задавать действия с мучительными подробностями. Знание типов объектов также позволяет компилятору обнаруживать ошибки, которые в противном случае остались бы до тестирования. Заметьте, что использование системы типов для того, чтобы получить проверку параметров функций, защитить данные от случайного искажения, задать новые операции и т.д., само по себе не увеличивает расходов по времени выполнения и памяти.

Особое внимание, уделенное при разработке **Си++** структуре, отразилось на возрастании масштаба программ, написанных со времени разработки **Cи**. Маленькую программу (меньше 1000 строк) вы можете заставить работать с помощью грубой силы, даже нарушая все правила хорошего стиля. Для программ больших размеров это не совсем так. Если программа в 10 000 строк имеет плохую структуру, то вы обнаружите, что новые ошибки появляются так же быстро, как удаляются старые. **Си++** был разработан так, чтобы дать возможность разумным образом структурировать большие программы таким образом, чтобы для одного человека не было непомерным справляться с программами в 25 000 строк. Существуют программы гораздо больших размеров, однако те, которые работают, в целом, как оказывается, состоят из большого числа почти независимых частей, размер каждой из которых намного ниже указанных пределов.

Естественно, сложность написания и поддержки программы зависит от сложности разработки, а не просто от числа строк текста программы, так что точные цифры, с помощью которых были выражены предыдущие соображения, не следует воспринимать слишком серьезно.

**3.9.1. Замечание по проекту языка Си++**

Существенным критерием при разработке языка была простота. Там, где возникал выбор между упрощением руководства по языку и другой документации и упрощением компилятора, выбиралось первое. Огромное значение также предавалось совместимости с **Cи**, это помешало удалить синтаксис **Cи**.

В **Си++** нет типов данных высокого уровня и нет первичных операций высокого уровня. В нем нет, например, матричного типа с операцией обращения или типа строка с операцией конкатенации. Если пользователю понадобятся подобные типы, их можно определить в самом языке. По сути дела, основное, чем занимается программирование на **Си++** - это определение универсальных и специально-прикладных типов. Хорошо разработанный тип, определяемый пользователем, отличается от встроенного типа только способом определения, но не способом использования.

Исключались те черты, которые могли бы повлечь дополнительные расходы памяти или времени выполнения. Например, мысли о том, чтобы сделать необходимым хранение в каждом объекте ”хозяйственной” информации, были отвергнуты. Если пользователь описывает структуру, состоящую из двух 16-битовых величин, то структура поместится в 32-битовый регистр.

 **Си++** проектировался для использования в довольно традиционной среде компиляции и выполнения, среде программирования на **Cи** в системе **UNIX**. Средства обработки особых ситуаций и параллельного программирования, требующие нетривиальной загрузки и поддержки в процессе выполнения, не были включены в **Си**++. Вследствие этого реализация **Си++** очень легко переносима. Однако есть полные основания использовать **Си++** в среде, где имеется гораздо более существенная поддержка. Такие средства, как динамическая загрузка, пошаговая трансляция и база данных определений типов могут с пользой применяться без воздействия на язык.

**4. ЗАКЛЮЧЕНИЕ.**

Изобретение языка программирования высшего уровня позволило нам общаться с машиной, понимать её (если конечно Вам знаком используемый язык), как понимает американец немного знакомый с русским языком древнюю азбуку Кириллицы. Проще говоря, мы в нашем развитии науки программирования пока что с ЭВМ на ВЫ. Поверьте мне это не сарказм вы только посмотрите как развилась наука программирования с того времени, как появились языки программирования, а ведь язык программирования высшего уровня, судя по всему ещё младенец. Но если мы обратим внимание на темпы роста и развития новейших технологий в области программирования, то можно предположить, что в ближайшем будущем, человеческие познания в этой сфере, помогут произвести на свет языки, умеющие принимать, обрабатывать и передавать информации в виде мысли, слова, звука или жеста. Так и хочется назвать это детище компьютеризированного будущего: «языки программирования "высочайшего" уровня». Возможно, концепция решения этого вопроса проста, а ближайшее будущее этого проекта уже не за горами, и в этот момент, где нибудь в Запорожье, Амстердаме, Токио или Иерусалиме, перед стареньким 133MHz горбится молодой, никем не признанный специалист и разрабатывает новейшую систему искусственного интеллекта, которая наконец-то позволит человеку, с помощью своих машинных языков, вести диалог с машиной на ТЫ.

Размышляя над этим, хочется верить в прогресс науки и техники, в высоко - компьютеризированное будущее человечества, как единственного существа на планете, пусть и не использующего один, определенный разговорный язык, но способного так быстро прогрессировать и развивать свой интеллект, что и перехода от многоязыковой системы к всеобщему пониманию долго ждать не придется.

Завершить свой труд хорошо бы на такой оптимистичной ноте, но нет, напоследок хочется процитировать человека, фрагменты работы которого, в виде информации о языке Си, вам уже попадались на страницах этого текста:

*Единственный способ изучать новый язык программирования – писать на нём программы.*

 Брайэн Керниган

# **5. БИБЛИОГРАФИЯ**

1) “LEX - генератор программ лексического анализа”

 Давидов Михаил Изгияевич; Антонов Вадим Геннадьевич

 МОСКВА – 1985;

2) "BASIC Face-off", Justin J.Crom,

 PC Tech Journal, September 1987, p.136

 Перевод: Лопухов В.Н. (Интегратор Promt98);

3) “Язык программирования Си.” Б.В. Керниган, Д. Ритчи, А. Фьюэр.

 Русский перевод: Москва: Финансы и Статистика. 1985 г.;

4) “Основы автоматизации” ч.1, Золотарев В.В., 1978 г.;

5) “Языки программирования” кн.5, Ваулин А.С., 1993 г.;

6) “Языки программирования: разработка и реализация”,

 П. Терренс, 1979 г.;

7) “Введение в программирование на языке Ассемблер”

 ч.1, Касвандс Э.Г.;

8) “Языки программирования высокого уровня”,

 Хротко Г., 1982 г.;

9) “Языки программирования”,

 Малютин Э.А., Малютина Л.В., 1982 г.;

10) “Новые языки программирования и тенденции их развития”, Ушкова В., 1982 г.;

11) “Мир Лиспа” т.1, Хьювенен Э., Сеппенен Й., 1990 г.;

12) “Алгоритмические языки реального времени”, Янг С., 1985 г..