**2. НАПИСАНИЕ ПРОГРАММ НА ЯЗЫКЕ CF-PASCAL**

В этом разделе мы узнаем о том, как писать простые программы на языке CF-Pascal, проверять корректность синтаксиса и использовать принципы проектирования и разработки программ в процессе пошагового совершенствования написания, исполнения и тестирования простейших программ.

**2.1. Введение в синтаксис языка CF-Pascal**

Программы CFPascal составляются в соответствии с четко заданным синтаксисом. Точный набор правил определяет синтаксис программ и процедуры его проверки.

Новые идеи: Синтаксис программы, нотация BNF, синтаксическое правило, альтернативное синтаксическое правило, рекурсивное синтаксическое правило, доказательство синтаксической корректности.

Для того, чтобы научиться писать на английском, в первую очередь необходимо освоить синтаксис и грамматику языка, чтобы быть способным писать сложные предложения, объединять их в параграфы и т.д. Синтаксис английского языка, требует многих лет изучения, потому что он является естественным языком и постоянно обновляется и изменяется.

Аналогично, для того, чтобы писать на CF-Pascal, тоже необходимо овладеть синтаксисом, чтобы быть способным писать операторы и составлять с их использованием программы. Однако CF-Pascal является формальным языком с четко заданным набором синтаксических правил. Эти правила позволяют изучить синтаксис языка гораздо быстрее и полнее, чем это возможно в случае естественных языков. Изучив синтаксис языка, можно смело переходить к вопросу о том, как писать программы для решения предстоящих задач.

Паскаль-машина рассматривает Паскаль-программу как последовательность символов. Люди, в свою очередь, узнают знакомые слова и могут догадаться, что имеется в виду в следующей последовательности символов, не являющейся Паскаль-программой:

PROGRAM CopyChar(INPUT, OUTPUT);

VAR

Ch; CHAR;

BEGIN

READ(Ch);

WRITELN(Ch)

END.

Паскаль-программа, с таким же именем, которая была рассмотрена на предыдущей лекции, выглядит так:

PROGRAM CopyChar(INPUT, OUTPUT);

VAR

Ch: CHAR;

BEGIN

READ(Ch);

WRITELN(Ch)

END.

Разница между этими строками символов в один-единственный символ не позволит компилятору языка Паскаль откомпилировать первый пример.

Из-за необходимости составления Паскаль-программ с точностью до символа, эта глава ознакомит вас со строгими правилами синтаксиса языка CF-Pascal. Конечно, эксперименты и примеры полезны во время изучения, но точные и лаконичные правила позволят вам уверенно проверять корректность программ. Изучения языков программирования только по примерам и использования компилятора для проверки синтаксиса приведет не только к ограниченному использованию ЯП, но также создаст неверное понимание о написания программ вообще.

**2.1.1. Синтаксис Pascal-программы**

Программа CopyChar – всего лишь пример последовательности символов, удовлетворяющей правилам синтаксиса языка Pascal. Правила определяют элементы программ (это что-то вроде частей речи в русском языке) при помощи синтаксических категорий, а также показывают, как элементы программ попадают в эти категории.

Синтаксические элементы могут быть описаны по правилам грамматики в нотации BNF (Backus-Naur Form) названной по имени компьютерных ученых, которые вместе с другими учеными участвовали в разработке языка ALGOL-60 – концептуальным предшественником языка Паскаль. Синтаксические элементы записываются в виде слова или фразы внутри угловых скобок (<…>) и определяются при помощи правила, которое может относиться к другим синтаксическим элементам.

**Правило 1** определяется часть синтаксического элемента «программа».

**SR1. <программа> ::= <заголовок> ; < блок > .**

Синтаксическое правило 1 утверждает, что синтаксический элемент <**программа**> может быть составлен из синтаксического элемента <**заголовок**>, следующей за ней **точки с запятой**, синтаксического элемента <**блок**> и последующей **точки**. Угловые скобки, заключающие в себе названия других синтаксических элементов, означают, что каждый из этих элементов также определяются синтаксическим правилом. Правило 1 определяет элемент <**программа**>, тогда как <**заголовок**> и <**блок**> будут определены при помощи других правил. Точка с запятой и точка – это синтаксические литералы или строки символов, которые должны присутствовать именно в том виде, в каком они показаны.

Символ **::=** означает «**является любым примером**». Синтаксическое правило 1 читается так:

<программа> является любым примером <**заголовка**>, последующей **точки с запятой**, последующего <**блока>,** последующей **точки.**

Символы **::=** и угловые скобки < > являются **метасимволами,** поскольку они не являются частью определяемого языка. Ни ::=, ни <программа> не могут присутствовать в Пасколь программы, за исключением, возможно, комментариев. В случае программы **CopyChar** последовательность символов

PROGRAM CopyChar(INPUT,OUTPUT)

соответствует синтаксической части <заголовок>, а последовательность:

VAR

Ch: CHAR;

BEGIN

READ(Ch);

WRITELN(Ch);

END

соответствует синтаксической части <блок>.

Для того, чтобы вся программа соответствовала синтаксической части <**программа**>, надо вставить точку с запятой после заголовка и точку после части <блок>, как это указано в синтаксическом правиле 1. Мы будем говорить об элементах программ, говоря о том какая последовательность символов какому синтаксическому элементу соответствует, и затем будет ссылаться к данной последовательности символов по имени элемента.

Синтаксическое правило 2 определяет синтаксический элемент <заголовок>.

**SR2. <заголовок> ::= PROGRAM <идентификатор> (INPUT, OUTPUT)**

Заголовок начинается с символьной последовательности **PROGRAM**, последующего <**идентификатора**> и символьной строки **(INPUT, OUTPUT)**. В программе CopyChar, <идентификатором> является последовательность сама символов **CopyChar**.

Стандартные слова вроде **PROGRAM** и <**идентификатор>** должны бать разделены друг от друга как минимум одним пробелом, в том случае, когда между ними не находится знак препинания вроде **точки с запятой**. В синтаксических правилах пробелы никогда явно не показываются. Пробелы используются в них только для того, чтобы повысить их читаемость. В тех случаях, когда может быть вставлен один пробел, может быть использовано любое их количество пробелов. Символ конца строки может заменить собой любой пробел.

**<Идентификатором>** является любая последовательность символов, которая начинается с буквы, за которой может следовать любое (0 или больше) количество букв и/или цифр, без пробелов и других символов.

Синтаксическое правило 3 задает определение идентификатора, используя два новых понятия. Во первых – это вертикальная черта **|** которая означает «**или**». Во вторых, данное правило использует в определении синтаксического элемента тот же самый синтаксический элемент. Вертикальная черта – это другой метасимвол, в данных синтаксических правилах.

**SR3. <идентификатор> ::= <буква>**

**| <идентификатор> <буква>**

**| <идентификатор> <цифра>**

Данное правило читается так:

<Идентификатором> является <буква>, либо (другой) <идентификатор>, за которым следует <буква>, либо (другой) <идентификатор>, за которым следует <цифра>.

Каждый возможный вариант в таком правиле называется **альтернативой** правила. В том случае, когда нам будет необходимо сослаться на одну из альтернатив, мы будем пользоваться нотацией Dewey. Например, правило SR3 имеет первую альтернативу 3.1 (<буква>), и две другие альтернативы 3.2 и 3.3. Короче говоря, синтаксическое правило SR3 гласит, что любая буква или цифра, добавленная в конец <идентификатора> создает новый <идентификатор>. Правило 3.1 определяет <идентификатор> как единственную букву, а правила 3.2 и 3.3 определяют <идентификаторы> длиной более одного символа в терминах <идентификаторов> меньшей длины.

**SR4. <буква> := A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z**

**SR5. <цифра > := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

Альтернатива 4.7, например, позволяет <букве> быть равной (например) равной **G.**

Теперь, используя эти определения мы можем построить <идентификатор> CopyChar по одному символу за раз:

|  |  |
| --- | --- |
| <идентификатор> | По синтаксическому правилу |
| C | С является <буквой> по правилу 4.3, и, следовательно, <идентификатором> по правилу 3.1 |
| Co | **С** является <буквой> как описано выше, **o** является <буквой> по правилу 4.41, поэтому **Co** является идентификатором по правилу 3.2 |
| Cop | Подобным образом, как в случае с **Co, Cop** является идентификатором по правилам 4.42 и 3.2 |
| … |  |
| CopyChar | подобно CopyCha и правилам 1.44 и 3.2 |

Подобным образом последовательности символов Copy1, XxXxX, One4Two являются <идентификаторами>.

Поскольку пробелы встречаются в синтаксических правилах только для улучшения читаемости, они не могут встречаться внутри <идентификаторов>

Почему ниже перечисленные символьные строки не являются идентификаторами?

|  |  |
| --- | --- |
| Последовательность символов | Не является <идентификатором> потому что |
| 2nd | не начинается с буквы |
| C 75 | содержит пробел |
| Ch1+Ch2 | символ + не является ни буквой, ни цифрой |

**2.1.2. Синтаксические деревья**

Правила синтаксиса языка CF-Pascal определяют пошаговое построение любой программы в виде последовательности символов. В случае, когда такое построение программы невозможно, символьная последовательность не может быть текстом Паскаль-программы. Построение программы может быть изображено в виде **синтаксического дерева**. На **рисунке 1** частично построена <программа> CopyChar.

fig1

**Рис1. Частичное синтаксическое дерево для программы CopyChar**

В отличие от обычных деревьев, синтаксические деревья растут сверху вниз. В данном случае, корнем дерева является элемент <**программа>,** конечные узлы дерева называются **листьями**.

Данное дерево не показывает нам подробности строения <идентификатора>, но результат его построения показывается присоединенным пунктирной линией, сигнализируя о том, что результаты были сокращены.

Проходя по листьям дерева слева направо, мы получаем следующее:

PROGRAM CopyChar(INPUT, OUTPUT); <блок> .

Это законченная программа, за исключением элемента <блок>, синтаксическое правило которого еще не было дано и поэтому не может быть заполнено.

На рисунке 2 показано полное построение <идентификатора> **Copy1**

**fig2**

**Рис. 2. Синтаксическое дерево для <идентификатора> Copy1**

Когда синтаксическое дерево построено с использование всех необходимых правил синтаксиса, листья этого дерева должны явно присутствовать в программе. Когда мы проходим слева направо по листьям дерева, вставляя где положено пробелы, они как бы проговаривают нам элемент программы, название которого находится в корне дерева. Например, часть рисунка 1, изображенная на рисунке 3, показывает, что дерево с корнем <заголовок>, может построить следующее:

PROGRAM CopyChar(INPUT, OUTPUT)

fig3

**Рис 3. Синтаксическое дерево <заголовок>**

Иными словами

PROGRAM CopyChar (INPUT, OUTPUT)

является <заголовком>. Когда мы говорим о <заголовке>, мы имеем в виду эту строку. Но обычно значение идентификатора не имеет большого значения, и ссылка на <заголовок> обозначает любую строку, которая может быть таким образом построена. В данном случае единственная часть дерева, которая может меняться – это <идентификатор>.

Синтаксические деревья могут быть использованы для доказательства корректности любой последовательности символов являющейся элементом программы. Начиная с синтаксического правила, имеющего этот элемент слева от знака ::=, дерево строится по этому и другим синтаксическим правилам. Синтаксис последовательности символов считается корректным, если он формируется из листьев дерева путем вставки соответствующих пробелов. Основное правило вставки идентификаторов гласит:

1. Пробелы не могут быть помещены внутрь <идентификатора>
2. Пробел должен быть помещен между стандартными словами и идентификаторами, за исключением случаев, когда вставлен символ пунктуации. Пробел может быть вставлен перед или после стандартных слов, идентификаторов и символов пунктуации
3. Там, где может быть помещен один пробел, там может быть помещено любое их количество.
4. Разрыв строки может быть помещен в любом месте, где может быть помещен пробел. Однако разрыв строки не заменяет пробел в пункте 2.

Например, дерево <заголовок> рассмотренное выше, показывает, что

PROGRAM CopyChar (INPUT, OUTPUT)

является <заголовком>, как и

PROGRAM

CopyChar

(INPUT, OUTPUT)

показывая различные случаи расстановки пробелов и разрывов строк. В некоторых Паскаль-машинах разрыв строки игнорируется. В других – он заменяет пробел. Правило 4 является консервативным и безопасным при любых обстоятельствах.

В качестве проверки правильного построения синтаксического дера\ева, номера синтаксических правил, используемые при построении, могут быть приписаны к каждому узлу, как показано на **рисунке 4** для <идентификатора> B4U

fig4

**Рис 4. Синтаксическое дерево для <идентификатора> B4U**

По мере накопления опыта доказательство корректности синтаксиса станет для вас простым. Вы станете делать это на полном автоматизме. Как только вы прочувствуете идею, очень редко вам придется рисовать синтаксические деревья.

**2.1.3. Синтаксис блока.**

Синтаксис элемента <блок> описывается синтаксическим правилом 6.

**SR6. <блок> :== <раздел объявлений> ; <оператор BEGIN>**

**| <оператор BEGIN>**

**SR7. <раздел объявлений> :== VAR <список идентификаторов> : CHAR**

**SR8. <список идентификаторов> :== <идентификатор>**

**| <список идентификаторов> , <идентификатор>**

Помимо синтаксических правил, программы CF-Pascal должны удовлетворять следующим контекстным правилам:

**CR1. Все идентификаторы в разделе объявлений должны быть уникальными.**

**CR2. Ни одно из зарезервированных слов, например IF, WHILE, и т.д. не могут быть использованы в качестве идентификаторов. (Список зарезервированных слов будет дан позднее)**

Рисунок 5 показывает, что VAR Ch : CHAR является <разделом объявлений>

**fig5**

**Рис. 5. Синтаксическое дерево для <раздела объявлений>**

Синтаксис оператора BEGIN задается синтаксическими правилами 9, 10 и контекстным правилом 3

**SR9. <оператор BEGIN> ::= BEGIN <список операторов> END**

**SR10. <список операторов> ::= <оператор>**

**| <список операторов> ; <оператор>**

**CR3. Любой идентификатор в операторе, отличный от стандартных идентификаторов вроде READ, WRITE и т.д. должен присутствовать в <списке идентификаторов> <раздела объявлений>.**

**2.2. Синтаксис операторов языка CF-Pascal**

В данном разделе дается описание синтаксиса операторов языка Pascal.

**SR11. <оператор> ::= <оператор READ>**

**| <оператор WRITE>**

**| <оператор присвоения>**

**| <пустой оператор>**

**| <оператор BEGIN>**

**| <оператор IF>**

**| <оператор WHILE>**

**SR12. <оператор READ> ::= READ (<список идентификаторов>)**

**SR13. <оператор WRITE> ::= WRITE(<список write>)**

**| WRITELN (<список write>)**

**| WRITELN**

**SR14. <список write> ::= <элемент write>**

**| <список write> , <элемент write>**

**SR15. <элемент write> ::= <идентификатор>**

**| '<строка символов>'**

**SR16. <строка символов> ::= <символ>  
 | <строка символов> <символ>**

**SR17. <символ> ::= <буква> | <цифра> | <особый символ>**

**SR18 <особый символ> ::= ( | ) | = | { | } |: |; | ’’ | < | > | , | . | #**

Некоторые Паскаль-машины имеют больший набор специальных символов, но вышеупомянутых символов нам будет достаточно для CF-Pascal.

**SR19. <оператор присвоения> ::= <идентификатор> := <выражение>**

**SR20. <выражение> ::= <идентификатор>**

**| '<символ>'**

**SR21. <пустой оператор> ::=**

**SR22. <оператор IF> ::= IF <условие> THEN <оператор>**

**| IF <условие> THEN <оператор> ELSE <оператор>**

**SR23. <условие> ::= <выражение> <сравнение > <выражение>**

**SR24. <сравнение> ::= < | = | > | <= | <> | >=**

**SR25. <оператор WHILE> ::= WHILE <условие> DO <оператор>**

**Комментарии** в Паскаль-программах не упоминаются в синтаксических правилах BNF, поскольку они имеют особое правило:

Комментарий начинается со специального символа { и заканчивается }; между этими маркерами может встречаться любой символ. Комментарий может располагаться в любом месте синтаксически корректной программы, где может располагаться пробел, за исключением <строки символов> <элемента write>.

Паскаль-машина будет пытаться выполнить любую последовательность символов, соответствующую синтаксическим и контекстным правилам, считая ее Паскаль-программой. Однако синтаксически корректная программа может попасть в бесконечный цикл. Она может прерваться из-за того, что оператор READ требует данных больше, чем имеется, она может решить задачу ошибочно. Эти проблемы лежат вне **синтаксиса**, в области, называемой **семантикой.**

**2.3. Проектирование и разработка программ.**

Знание правильного синтаксиса Паскаль-операторов – это основы. Однако написание правильно работающей программы требует тщательного планирование того, что будет делать программа во время свого выполнения. План программы может быть написан в виде проекта, а его подробности могут быть добавлены путем дальнейшей доработки. В хорошем процессе разработки части проекта собираются в исполняемую программу все большего и большего размера. Проект может быть проверен во время исполнения с помощью хорошо продуманный тестов.

Новые понятия: набросок программы, пошаговое совершенствование, раздел проекта, разработка программ, тестирование программ.

Написанная программа считывается и исполняется Паскаль-машиной, поэтому при написании программы необходимо точно и внятно указывать машине, что вы имеете в виду. Требование точности не ослабляется и в случае хорошего правописания, но оно усиливает значимость следующего правила – правила написания с проектированием и слипания субъекта проектирования в руке.

Проектирование имеет иерархическую структуру. Она позволяет программисту как бы снижаться над одной частью проекта, не учитывая в данный момент тонкости других участков проекта, а затем взмывать вверх. Это позволяет программисту постоянно изучать и подвергать сомнению общую композицию, в то время как происходит написание программы. Если процесс написания программы идет плохо, то возможно проект следует изменить, что бы он соответствовал неожиданным сложностям.

При написании на естественном языке процесс проектирования или создания наброска происходит лишь до начала композиции параграфов и предложений. Программирования – это поле точная штука, поэтому проектирования должно продолжаться до окончательного текста программы. Сам процесс программирования может относиться к процессу проектирования.

Термин «Пошаговое совершенствование» был предложен Никлаусом Виртом, разработчиком языка Паскаль. Каждый шаг в процессе композиции описывает то, что делает программа на новом уровне детализации. Финальные шаги описывают текст программы с мельчайшими подробностями.

При хорошем проектировании интеллектуальный контроль над задачей должен быть как можно проще. При пошаговой разработке программы вероятность упустить что-либо уменьшается, и помнить множество подробностей нет необходимости.

Проект при пошаговом совершенствовании программы записывается особым и точным способом. Текст программы с корректным синтаксисом всегда является основой проекта. Там, где подробности пока не известны, комментарий описывает задачу, которая должна быть выполнена. Затем, на следующем уровне этот комментарий заменяется текстом программы, возможно с другими комментариями, скрывающими следующие детали. И так далее. Окончательная программа будет состоять из текста на CF-Pascal, и изначально написанных комментариев, которые описывают, что делает последующий текст. В самом конце, каждый комментарий будет иметь свою реализацию в виде текста программы.

**2.3.1. Скачка Paul Rever**

**Шаг 1.** Составляем набросок нашей программы:

PROGRAM PaulRevere(INPUT,OUTPUT);

*{Выводит соответствующие сообщения, в зависимости от значения INPUT: ‘By land’ в случае 1; 'By sea’ в случае 2.}*

VAR

Lanterns: CHAR;

BEGIN *{PaulRevere}*

*{cчитываем Lanterns}*

*{выводим сообщение Paul Rever'а}*

END. {PaulRevere}

**Шаг 2.** Дополним нашу программу вводом и эхом ввода:

PROGRAM PaulRevere(INPUT,OUTPUT);

*{Выводит соответствующие сообщения, в зависимости от значения INPUT: ‘By land’ в случае 1; 'By sea’ в случае 2.}*

VAR

Lanterns: CHAR;

BEGIN *{PaulRevere}*

*{считываем Lanterns}*

READ(Lanterns);

WRITELN('Lanterns is ', Lanterns); *{тестовый оператор}*

*{выводим сообщение Paul Rever'а}*

END. {PaulRevere}

Запустим ее на выполнение:

INPUT :1

OUTPUT:Lanterns is 1

INPUT :X

OUTPUT:Lanterns is X

Теперь усовершенствуем программу так, чтобы она выполняла действие в комментарии после WRITELN. После того, как значение Lanterns было задано оператором READ, его значение может быть проверено в операторе IF:

*{выводим сообщение Paul Rever'а}*

IF Lanterns = ‘1’

THEN

WRITELN(‘The British are coming by land.’)

ELSE

WRITELN(‘The British are coming by sea.’)

В данном случае в операторе IF подразумевается, что если Lanterns не равен 1, то он равен 2. Будет лучше, если мы будем более внимательны в данном случае:

{выводим сообщение Paul Rever'а}

IF Lanterns = ‘1’

THEN

WRITELN(‘The British are coming by land.’)

ELSE

IF Lanterns = ‘2’

THEN

WRITELN(‘The British are coming by sea.’)

ELSE

WRITELN(‘The North Church shows only ’’’, Lanterns, ’’’.’)

Теперь окончательный вариант нашей программы принял вид:

PROGRAM PaulRevere(INPUT,OUTPUT);

{Выводит соответствующие сообщения, в зависимости от значения INPUT: ‘By land’ в случае 1; 'By sea’ в случае 2.}

VAR

Lanterns: CHAR;

BEGIN {PaulRevere}

{считываем Lanterns}

READ(Lanterns);

{выводим сообщение Paul Rever'а}

IF Lanterns = ‘1’

THEN

WRITELN(‘The British are coming by land.’)

ELSE

IF Lanterns = ‘2’

THEN

WRITELN(‘The British are coming by sea.’)

ELSE

WRITELN(‘The North Church shows only ’’’, Lanterns, ’’’.’)

END. {PaulRevere}

INPUT :1

OUTPUT:The British are coming by land.

INPUT :A

OUTPUT:The North Church shows only ‘A’.

INPUT :2

OUTPUT:The British are coming by sea.

**2.3.2. Разделы проекта и разработочные программы**

В маленьких программах типа PaulRevere пошаговое совершенствование может быть выполнено естественно и непринужденно. Однако по мере того, как задачи и программы становятся более сложными, тщательное и систематический подход сохранит ваше время в конечном итоге.

В программировании существуют 2 различные фазы:

1. Проектирование – продумывание того, какой должна быть программа для надлежащего решения проблемы.

2. Разработка – ввод текста программы для использования Паскаль-машиной

При пошаговом совершенствовании текст программы, спроектированный, с целью решения задачи с большими деталями – называется разделом проекта. Каждый раздел проекта, в свою очередь может содержать более подробное описание задачи (в виде комментариев), что повлечет за собой написание дополнительный разделов. Результатом фазы проектирования будет иерархия разделов проекта, которые совместно решают проблему. Каждый раздел проекта при этом будет оператором Паскаля – обычно, это оператор BEGIN, IF или WHILE. Это операторы 5-15 строк длиной + 2-4 задачи, оставшиеся на следующем уровне.

После того, как программа целиком будет доведена до иерархии разделов проекта, можно приступать к трансляции в форму, понятную ЭВМ. На первый взгляд, это может покаазться простым шагом. Однако, если большая программа будет переведена в Паскаль машину даже с одной ошибкой, последствия могут быть катастрофическими. В самом лучшем случае эта ошибка будет обнаружена компилятором Паскаля. Это даст программисту второй шанс на трансляцию программы. Если программа будет запущена на исполнение, то она может выдавать неправильные результаты, однако программист может не подозревать об этом и потратитm время изучая проект на наличие ошибок. Результаты могут быть настолько причудливыми и странными, что будет трудно связать их с исходным текстом. Например, неверное условие, например, < вместо >, может завести во время выполнения в неожиданное место программы, в общем, ошибки будут расти как снежный ком. В самом худшем случае будет казаться, что программа работает корректно, поскольку тестирование не выявило ошибку во время трансляции.

Последовательность исполняемых программ, каждая и которых отражает бОльший раздел проекта, может содействовать правильной и систематической трансляции в код для Паскаль-машины. Каждая программа из такой последовательности будет называться разработочной программой. Разработочные программы – являются накоплением разделов проекта, которая растет в размере до тех пор, пока весь проект не будет преобразован в Паскаль-программу. Каждая разработочная программа определяется таким образом, чтоб она может быть выполнена и оттестирована для поверки корректной трансляции на каждом этапе разработки.

На практике, трансляция проекта в Паскаль-программу может быть выполнена при помощи блоков больших, чем один шаг проекта, обычно 15-50 строк за один раз. Таким образом успешная разработочная программа создается при помощи комбинации нескольких частей проекта в окончательно оттестированную разработочную программу. Для проверки корректного исполнения может быть необходимо добавить временные операторы WRITE для создания видимого вывода.

**2.3.3. Письмо Sarah Rever**

“Dear Paul: I have been to the Chelsea headland…”

BEGIN {сдвигаем окно}

W1 := W2;

W2 := W3;

W3 := W4;

READ(W4)

END

{Проверка на ‘land’}

IF W1 = ‘l’

THEN

IF W2 = ‘a’

THEN

IF W3 = ‘n’

THEN

IF W4 = ‘d’

THEN {‘land’ было найдено}

WHILE {‘land’ или ‘sea’ не найдены}

DO

BEGIN

{сдвигаем окно}

{проверка на ‘land’}

{проверка на 'sea'}

END

|  |  |
| --- | --- |
| Looking | Смысл |
| Y | Продолжаем искать |
| L | Британцы наступают с земли |
| S | Британцы наступают с моря |
| N | В конце ввода, ни **land**, ни **sea** не были найдены |

**Раздел проекта 1**

PROGRAM SarahRevere(INPUT,OUPUT)

{выводит сообщение о том, как наступают британцы, в зависимости от того, найдено ‘land’ или ‘sea’}

VAR

W1, W2, W3, W4, Looking : CHAR;

BEGIN {SarahRevere}

{инициализируем переменные W1, W2, W3, W4, Looking}

WHILE {‘land’ или ‘sea’ не найдены}

DO

BEGIN

{сдвигаем окно, проверяем на конец данных}

{проверка на ‘land’}

{проверка на 'sea'}

END

{Создаем письмо Сары}

END. {SarahRevere}

Раздел проекта 1.1

BEGIN {инициализируем переменные W1, W2, W3, W4, Looking}

W1 := ‘ ‘;

W2 := ‘ ‘;

W3 := ‘ ‘;

W4 := ‘ ‘;

Looking := ‘Y‘;

END

Раздел проекта 1.2

BEGIN {сдвигаем окно, проверяем на конец данных}

W1 := W2;

W2 := W3;

W3 := W4;

READ(W4);

IF W4 = ‘#’

THEN {конец данных}

Looking := ‘N’

END

Раздел проекта 1.3

BEGIN {Проверка на ‘land’}

IF W1 = ‘l’

THEN

IF W2 = ‘a’

THEN

IF W3 = ‘n’

THEN

IF W4 = ‘d’

THEN {‘land’ было найдено}

Looking := ‘L’

END

Раздел проекта 1.4

BEGIN {Проверка на ‘sea'}

IF W2 = ‘s’

THEN

IF W3 = ‘e’

THEN

IF W4 = ‘a’

THEN {‘sea’ было найдено}

Looking := ‘S’

END

Раздел проекта 1.4

BEGIN {Создаем письмо Сары}

IF Looking = ‘L’

THEN

WRITELN(‘The British are coming by land.’)

ELSE

IF Looking = ‘S’

THEN

WRITELN(‘The British are coming by sea.’)

ELSE

WRITELN(‘Sarah didn’’t say’)

END

# Стадия разработки программы «Письмо Sarah Revere»

Разработочная программа 1А

PROGRAM SarahRevere(INPUT,OUPUT)

{выводит сообщение о том, как наступают британцы, в зависимости от того, найдено ‘land’ или ‘sea’}

VAR

W1, W2, W3, W4, Looking : CHAR;

BEGIN {SarahRevere}

BEGIN {инициализируем переменные W1, W2, W3, W4, Looking}

W1 := ‘ ‘;

W2 := ‘ ‘;

W3 := ‘ ‘;

W4 := ‘ ‘;

Looking := ‘Y‘;

END;

WHILE Looking = ‘Y’

DO

BEGIN

BEGIN {сдвигаем окно, проверяем на конец данных}

W1 := W2;

W2 := W3;

W3 := W4;

READ(W4);

IF W4 = ‘#’

THEN {конец данных}

Looking := ‘N’

END;

WRITE(W1,W2,W3,W4,‘,’); {тестовый оператор}

{проверка на ‘land’}

{проверка на 'sea'}

END

{Создаем письмо Сары}

END. {SarahRevere}

Результаты выполнения программы:

INPUT :#

OUTPUT: #,

INPUT :D#

OUTPUT: D, D#,

INPUT :De#

OUTPUT: D, De, De#,

INPUT :Dea#

OUTPUT: D, De, Dea,Dea#

INPUT :Dear Pau#

OUTPUT: D, De, Dea,Dear, ear ,ar P,r Pa, Pau,Pau#,

Разработочная программа 1B

PROGRAM SarahRevere(INPUT,OUPUT)

{выводит сообщение о том, как наступают британцы, в зависимости от того, найдено ‘land’ или ‘sea’}

VAR

W1, W2, W3, W4, Looking : CHAR;

BEGIN {SarahRevere}

BEGIN {инициализируем переменные W1, W2, W3, W4, Looking}

W1 := ‘ ‘;

W2 := ‘ ‘;

W3 := ‘ ‘;

W4 := ‘ ‘;

Looking := ‘Y‘;

END;

WHILE Looking = ‘Y’

DO

BEGIN

BEGIN {сдвигаем окно, проверяем на конец данных}

W1 := W2;

W2 := W3;

W3 := W4;

READ(W4);

IF W4 = ‘#’

THEN {конец данных}

Looking := ‘N’

END;

BEGIN {Проверка на ‘land’}

IF W1 = ‘l’

THEN

IF W2 = ‘a’

THEN

IF W3 = ‘n’

THEN

IF W4 = ‘d’

THEN {‘land’ было найдено}

Looking := ‘L’

END;

BEGIN {Проверка на ‘sea}

IF W2 = ‘s’

THEN

IF W3 = ‘e’

THEN

IF W4 = ‘a’

THEN {‘sea’ было найдено}

Looking := ‘S’

END

END;

WRITELN(‘Looking is ’, Looking); {тестовый оператор}

{Создаем письмо Сары}

END. {SarahRevere}

Результаты:

INPUT :Dear Paul: I have been #

OUTPUT:Looking is N

INPUT :Dear Paul: I have been to Chelsea #

OUTPUT:Looking is S

INPUT :Dear Paul: My land lies along the sea #

OUTPUT:Looking is L

Разработочная программа 1C

PROGRAM SarahRevere(INPUT,OUPUT)

{выводит сообщение о том, как наступают британцы, в зависимости от того, найдено ‘land’ или ‘sea’}

VAR

W1, W2, W3, W4, Looking : CHAR;

BEGIN {SarahRevere}

BEGIN {инициализируем переменные W1, W2, W3, W4, Looking}

W1 := ‘ ‘;

W2 := ‘ ‘;

W3 := ‘ ‘;

W4 := ‘ ‘;

Looking := ‘Y‘;

END;

WHILE Looking = ‘Y’

DO

BEGIN

BEGIN {сдвигаем окно, проверяем на конец данных}

W1 := W2;

W2 := W3;

W3 := W4;

READ(W4);

IF W4 = ‘#’

THEN {конец данных}

Looking := ‘N’

END;

BEGIN {Проверка на ‘land’}

IF W1 = ‘l’

THEN

IF W2 = ‘a’

THEN

IF W3 = ‘n’

THEN

IF W4 = ‘d’

THEN {‘land’ было найдено}

Looking := ‘L’

END;

BEGIN {Проверка на ‘sea}

IF W2 = ‘s’

THEN

IF W3 = ‘e’

THEN

IF W4 = ‘a’

THEN {‘sea’ было найдено}

Looking := ‘S’

END

END;

BEGIN {Создаем письмо Сары}

IF Looking = ‘L’

THEN

WRITELN(‘The British are coming by land.’)

ELSE

IF Looking = ‘S’

THEN

WRITELN(‘The British are coming by sea.’)

ELSE

WRITELN(‘Sarah didn’’t say’)

END

END. {SarahRevere}

**Принципы проектирования и разработки программ**

**Фаза проектирования.**

**Решение задачи** (выполняется совместно с проектированием программы)

1. **Поймите дачу.** В случае необходимости, напишите краткое изложение задачи и корректируйте его по мере обнаруживать путей решения проблемы.
2. **Разработайте стратегию решения.** Наметьте в общих чертах решение на русском языке с одним или несколькими уровнями детализации. Корректируйте стратегию решения по мере создания окончательного проекта программы.
3. **Анализируйте и осуждайте свою стратегию решения.** На каждом уровне детализации ищите простейшие пути решения задачи. Не является ли оно более сложным, нежели требуется?
4. **Организуйте свое решение путем написания улучшенного проекта.** Не требуется ли реорганизация отдельных разделов проекта для лучшей понятности? Могут ли быть обнаружены подобные операции и организованы в общих разделах проекта?

**Проектирование программы** (выполняется совместно с решением задачи)

* 1. **Разработайте план написания** путем преобразования вашей стратегии решения в иерархию разделов придуманного текста программы и комментариев. Эти описания задачи должны, в свою очередь, служить в качестве имен разделов проекта на следующем уровне детализации.
  2. Как правило, **разделы проекта не должны быть не более 15 строк**, и должны иметь вразумительные и легко понимаемые комментарии. Удержание разделов проекта в таких маленьких размерах и тщательное комментирование требует дисциплинированности, но это впоследствии будет вознаграждено легкостью чтения исходных текстов.
  3. **Храните разделы вашего проекта физически отдельно** для того, чтобы обеспечить гибкость их написания и переписывания, однако думайте об их иерархической структуре во время анализа и упрощения своей программы.
  4. **Пишите и переписывайте разделы вашего проекта и вашу стратегию решения.** Решите задачу, отложите ее решение, и попытайтесь его улучшить.

**Организация программ с процедурами (подпрограммами).**

По мере роста программ, организация разделов проекта становится более важной при создании понятных программ.

**Простые процедуры**

PROCEDURE Copy;

BEGIN

READ(Ch);

WRITE(Ch);

END

PROGRAM UsingCopy(INPUT, OUTPUT);

{Копирует 2 символа из INPUT в OUTPUT. Выполнение программы прекращается, если INPUT пуст}

VAR

Ch: CHAR;

PROCEDURE Copy;

BEGIN

READ(Ch);

WRITE(Ch);

END;

BEGIN

Copy;

Copy;

WRITELN

END.

INPUT :ABCDE

OUTPUT:AB

**SR6. <блок> ::= <оператор BEGIN>**

**| <раздел объявлений> ; <оператор BEGIN >**

**| <список процедур> ; <оператор BEGIN >**

**|<раздел объявлений> ; < список процедур> ; <оператор BEGIN >**

**SR11. <оператор> ::= <оператор READ>**

**| <оператор WRITE>**

**| <оператор присвоения>**

**| <пустой оператор>**

**| <оператор BEGIN>**

**| <оператор IF>**

**| <оператор WHILE>**

**| <оператор вызова процедуры>**

**SR26. <список процедур> ::= <описание процедуры>**

**| <список процедур> ; <описание процедуры>**

**SR27. <описание процедуры> ::= PROCEDURE <идентификатор> ; <оператор BEGIN>**

**SR28. <оператор вызова процедуры> ::= <идентификатор>**

**CR3. Любой идентификатор в операторе, не являющийся <оператором вызова процедуры>, отличный от стандартных идентификаторов вроде READ, WRITE и т.д. должен присутствовать в <списке идентификаторов> <раздела объявлений>.**

**CR4. *Правило появления. <*Идентификатор> <оператора вызова процедуры> должен появиться в роли идентификатора, следующего за словом PROCEDURE в <описании процедуры>, до своего появления в <операторе вызова процедуры>. Иными словами, процедура должна быть описана в тексте программы до своего первого использования.**

**Фаза разработки.**

**Разработка программ** (совместно с тестированием программы)

1. **Разработайте план сборки.** Составьте части проекта (начав с простейшей программы) в последовательность все более сложных программ. План должен позволять выполнение программы на каждом шаге.
2. Как правило, **на каждом шаге добавляйте не более 50 строк из проекта в разработку.**
3. **Не начинайте с первых 50 строк до тех пор, пока ваш проект не будет завершен.** Записан должен быть ваш последний проект, а не первый.
4. **Каждые 50 строк программы должны работать без ошибок с первого раза.** Сосредоточенность в настоящем сохранит время и нервы в будущем.

**Тестирование программы** (совместно с разработкой программы)

1. **Думайте серьезно о вашем плане сборки и о вашей конечной программе.** Составьте правильную последовательность программ для получения положительной обратной связи на каждом этапе сборки программы.
2. **Разработайте план тестирования.** Создайте программы с временными операторами WRITE для получения выразительных результатов выполнения, несмотря на то, что это не окончательные результаты, необходимые вам.
3. **Для каждого шага вашего плана сборки проведите тестовые испытания,** и не забывайте об исключительных ситуациях, даже несмотря на их малую вероятность.