Oldies... But Goldies!

Да здравствует ПЛ/1!

или разъяснение молодым, каким должен быть
настоящий язык программирования и как он сделан

Памяти Гарри Килдэлла (Gary Kildall) посвящается

Содержание

1. Введение

Написать эту книгу меня подтолкнуло много причин. Одна из них – результаты поисков в Интернете по запросу «ПЛ/1» или «PL/1». Это не лучший из образцов для запросов. Поисковики часто игнорируют косую черту, и приходится читать о подводных лодках или рекламу с обратным адресом площадь такая-то, дом № 1 или вообще листать заголовки на польском потому, что там домен PL. К вопросу, почему язык имеет такое странное название, мы еще вернемся. Однако среди найденного и имеющего отношение к программированию складывается совершенно безрадостная картина пренебрежения к языку, которым я имею честь пользоваться (в том числе поддерживаю транслятор). Кратко это предвзятое отношение можно выразить так: ПЛ/1 давно мертвый ужасный монстр вроде динозавра. Он собрал все ошибки развития программирования, им нельзя пользоваться, у него было 2000 страниц документации, и т.д. и т.п.

Обидно это слышать, причем большей частью от людей, которые только что-то слышали об этом языке или, в лучшем случае, когда-то «проходили» его в институтах. А критерием правильности и необходимости того или иного свойства чаще всего объявляется наличие или отсутствие этого свойства в своем, любимом языке. Захотелось заступиться за язык и рассказать, как оно есть «на самом деле». Уж если и сравнивать PL/1 с животными времени динозавров, то это, скорее, крокодил, который и сейчас, миллионы лет спустя, не бедствует.

Кроме этого, полистав учебник дочери по информатике, я огорчился плохой, на мой взгляд, подаче материала. Спору нет, сложно написать хороший учебник, а легко только отбить охоту у учеников к любому предмету нудным изложением. Но неужели нельзя кратко и понятно объяснить основные моменты по теме программирование? Хотя «научно-популярный» стиль изложения обычно не одобряется, хотелось найти такие формулировки, как будто пишешь учебник для новичков. Чтобы было и просто и понятно и не нудно.

И я решил совершить свою попытку. С одной стороны, внести очередной вклад в разжигание «религиозной войны», т.е. объяснить, в том числе и на примерах, свою точку зрения на то, каким должен быть язык программирования. С другой стороны, рассказать о незаслуженно обиженном языке под необычным углом – с точки зрения работы транслятора.

Есть такая шутка: хочешь понять язык программирования – напиши для него транслятор. Но ведь и в самом деле: любое свойства языка должно реализовываться транслятором. А если транслятор ничего в данной части не делает, то и такого свойства у языка нет. Я один из программистов, имеющих доступ к исходному тексту используемого

транслятора. Правда, этот текст не совсем исходный, я сам его сделал из имевшегося транслятора и затем развивал. Такой доступ действительно помогает глубже понять замысел и возможности языка. Становится понятной и «стоимость» той или иной возможности с точки зрения реализации в трансляторе. Думаю, что если бы подобный исходный текст был перед глазами у авторов новых языков, многое было бы сделано лучше и проще.

Конечно, некоторые (например, специалисты из фирмы IBM) сразу отметят, что у меня не «настоящий» язык, а его урезанный вариант и поэтому не «настоящий» транслятор. Не буду спорить, хотя не всегда «урезанное» обязательно хуже «полного».

2. История

Конечно, не будем начинать с момента «когда Земля была еще теплая и по ней бродили мамонты, а потом слоны». Тем не менее, истории языков программирования уже больше полувека. На мой взгляд, настоящие языки программирования начались с совещания математиков из разных стран, на котором были зафиксированы принципы языка Алгол. Да, к этому моменту уже был Фортран. Но еще не было теории и солидной математической базы.

То, что у истоков стояли математики, было благом. Они привыкли к точной и лаконичной записи. Для них рекурсивные определения были также привычным делом. С тех пор придумано много языков, однако все они вышли из этого «Пересмотренного сообщения об Алголе» как русская проза из гоголевской «Шинели». Отцы-основатели объявили Алгол не концом истории, а лишь стартовой площадкой, с которой и должно начаться развитие языков программирования.

В отношении PL/1 так и вышло. Фирма IBM решила выпустить новое поколение ЭВМ и озаботилась переделкой Фортрана, чтобы он соответствовал новым требованиям. Была создана комиссия по переделке, которая как тот цыган решила, что «проще новых детей народить, чем этих отмывать» и стала разрабатывать новый язык, помня заветы Алгола. Как всегда был устроен конкурс, на котором выбрали один проект и передали его на реализацию в лондонское отделение IBM. А там поступили по-чапаевски: «все, что вы здесь напридумывали – наплевать и забыть» и сделали почти все по-своему.

Вот так появился очень интересный язык, вобравший весь имевшийся тогда опыт программирования. А название ему так и не придумали, как, например, не придумали название Международной космической станции – есть только аббревиатура. У этого языка тоже оказалась только аббревиатура с прибавлением глупой косой черты и единицы (что и затрудняет поиск в Интернете). Единица (иногда римская), между прочим, означает вовсе не порядковый номер (ни два, ни полтора потом так и не появилось), а «степень крутизны» - т.е. номер первый во всем.

Фирма IBM мечтала вообще его сделать единственным хотя бы в области инженерных и экономических задач. Сначала язык распространялся очень широко, в том числе и в нашей стране. Однако затем судьба распорядилась по-иному.

Особенно удивительно, что этот язык не стал стандартом де-факто для персональных компьютеров, хотя к 1981 году для этого были все предпосылки. Был такой Гарри Килдэлл, который создал прекрасный ассемблер PL/M и прекрасную операционную систему CP/M. Вот этому-то Гарри фирма IBM и предложила создать операционную систему для нового компьютера IBM PC. Но не срослось, а затем подвернулся недоучившийся студент Билл Гейтс и пошло-поехало. По легенде Гарри отказался от контракта с IBM, потому что хотел в этот день полетать на своем самолете. Однако некоторые утверждают, что ничего подобного: Килдэлл со своей женой-юристом явился на переговоры, но не удовлетворился предложенной суммой контракта. И мы получили недоразвитую MS DOS вместо CP/M. А вот если бы Килдэлл стал главным программистом у Гейтса (или, если хотите, Гейтс стал бы начальником отдела продаж у Килдэлла) мы бы жили сейчас в совсем другом компьютерном мире. Правда, в последние годы ходят слухи, что не все так просто и у Килдэлла тогда не было ни малейшего шанса.

А причем здесь язык PL/1? А при том, что уже с 1980 года команда Килдэлла работала над транслятором с PL/1 для нового тогда процессора 8086. Угадайте, какой язык стал бы основным на IBM PC, если бы Гарри со своей командой и этим тогда единственным и почти готовым транслятором стал бы там главным? В действительности же с августа 1981 года MS DOS затопила все остальное, в том числе и CP/M. Общедоступные и очень простые трансляторы (Бейсик, Паскаль, Си) опередили PL/1 для MS DOS, который начал продаваться не раньше 1984 года. Но поезд уже ушел. Для большинства программистов первым практическим языком стал язык, доступный им на их персональном компьютере.

Простота языков, граничащая с примитивностью, давала ощущение полного понимания всего и уверенность, что языкам уже больше ничего не нужно и вообще никаких других языков не нужно. Быстро образовались фан-клубы языков Си и Паскаля, до сих пор обливающие друг друга грязью в непрекращающихся перепалках, метко названных «религиозными войнами». Наверное, это свойство человеческой натуры: то, что ты создаешь, хочется считать самым совершенным, так же как и средства, которыми ты это создаешь.

Каким же образом автор ухитрился связаться с PL/1, а не с Паскалем или Си? Самым что ни на есть естественным и случайным образом. С персональным компьютером я познакомился довольно рано для нашей страны – в самом начале 1987 года. А компьютер был куплен еще раньше, в 1985 году и имел тогда только две прикладные программы: текстовый редактор «SideKick» фирмы «Borland» (чуть ли не первый их

продукт) и транслятор с PL/1 подмножества «G» от фирмы «Digital Research» (фирмы Гарри Килдэлла).

Я и не собирался заниматься языками и трансляторами. Задача стояла быстрее освоить имеющийся язык и вести инженерные расчеты с помощью персонального компьютера. Однако каждый шаг давался с трудом: непонимание свойств языка (до этого я работал с БЭСМ-Алгол) накладывалось на ошибки транслятора и на особенности продуктов для американских пользователей, например, кириллицу нельзя было писать даже в комментариях.

Постепенно пришло понимание, что надо разобраться в трансляторе без помощи Гарри (который, кстати, погиб в 1994 году при неясных обстоятельствах), а затем исправить и дополнить его так, как нужно. А как насчет авторских прав? Знаю, знаю. Сам читал на коробках странную надпись «дизассемблирование запрещено». Угу, запрещено, значит, смотреть, как оно устроено? Но к счастью, по нашим законам разрешено вносить в купленные программы исправления для устранения ошибок и повышения функциональности. Ошибок в трансляторе было предостаточно. Кстати, на меня в свое время произвела сильное впечатление фраза из инструкции к транслятору: «если ошибка в нашем компиляторе приведет к порче Ваших данных, фирма обязуется восстановить эти данные, а если это невозможно, заплатит Вам 6 тысяч долларов». Сравните с условиями в продуктах Microsoft. Так что не думаю, что Килдэлл и его команда имели бы ко мне претензии.

С тех пор прошло много лет. Пришлось последовательно переходить с MS-DOS на Windows-95, затем на Windows-98, Windows-2000 и далее со всеми остановками. Изучение и доработка транслятора превратилось в хобби с элементами навязчивой идеи. Я уже привык к тому, что при очередной возникшей проблеме рассматривается и вопрос о доработке транслятора (начиная от дополнительных диагностических сообщений и кончая изменением вида операторов). Распечатка транслятора и сейчас лежит передо мной. Если разобраться, не такая это уж и сложная программа. Например, объем всех ее команд составляет лишь 89 442 байта. Используя свои же комментарии с этой распечатки как тезисы, приступим собственно к обсуждению языка.

3. Общая структура языка и программы

В некоторых учебниках описание языка начинается с набора символов. По-моему набор символов это мелочь, которую надо упоминать только в приложениях. Начинать надо с главного - с общей структуры.

Мне очень нравится деление общей структуры программы на три части: во-первых, собственно последовательность действий, ради которых пишется программа, во-вторых, всякие описания, информация для транслятора. А третья сторона – иерархия программы, т.е. главные и

подчиненные части. Программа как книга должна иметь начало, середину, конец. В ней есть главы, которые могут состоять из своих глав и т.д.

Как все это обрабатывает транслятор? Действия транслятор переводит в команды ЭВМ. Описания транслятор принимает к сведению и составляет из них таблицу всех объектов программы, помогающую ему правильно сформировать команды. А вот что он делает с иерархией? Да собственно ничего. Иерархия определяет путь, порядок, по которому транслятор разбирает программу. Вот и все.

Значит, в языке должны быть слова, обозначающие начало и конец составных частей программы, расставляя их определенным образом, мы заставляем транслятор разбирать программу определенным путем.

В языке должны быть слова, описывающие объекты программы. Сами по себе они не приводят к созданию каких-либо команд, но всю эту информацию транслятор запоминает в своей таблице и затем использует такую таблицу, когда нужно сформировать уже конкретные команды.

Наконец, в языке должны быть слова, описывающие собственно действия, алгоритм решения какой-либо задачи. Неудобно сразу из этих действий генерировать готовые команды, поэтому транслятор сперва переводит их в некоторое «внутреннее представление». Программу в этом внутреннем представлении уже легко разбирать. Если бы программу в таком представлении было бы легко и писать, то, пожалуй, и языков программирования не требовалось бы, так как программы в таком виде все очень похожи друг на друга даже для разных языков. Как раз одна из основных задач языков программирования – это позволить писать программы не в таком ужасном, а в более «человеческом» виде. А соответственно одна из основных задач транслятора – перевод алгоритма из «человеческого» вида в «нечеловеческий», но зато с простой и регулярной структурой.

Первое из отличий языка PL/1 от многих других языков, которое мы рассмотрим, касается описаний. Описания в PL/1 могут быть в любом месте программы, т.е. и до использования описанного в программе и после. На самом деле это вовсе не свойство языка, а свойство транслятора. Транслятор сначала ищет в программе только одни описания и составляет из них полную таблицу. Затем заново начинает разбирать программу, но уже наоборот, пропуская ранее разобранные описания и анализируя все остальное.

Вот и первый повод участия в «религиозной войне». Возможность помещать описания где угодно, это, конечно, не бог весть что. Но в некоторых случаях (отладка, временные исправления) это очень удобно. Для обеспечения этого удобства нужно сначала (на первом «проходе») искать и обрабатывать только описания. Причем, даже если бы не было такого требования располагать описания где угодно, все равно, очень удобно выделить разбор всех описаний как отдельный этап трансляции. Тогда ко второму «проходу», т.е. к собственно разбору программы мы

приступаем уже с полной таблицей всего, что описано в программе, а значит, точно знаем, что имел в виду программист в каждом месте программы. Такой отдельный разбор описаний быстро работал у Гарри Килдэлла даже в его первом трансляторе для процессора 8080 с жалкими ресурсами. Для современных процессоров время на такой предварительный просмотр программы вообще будет неощутимо и про правило многих языков «сначала опиши, только потом используй» можно было бы забыть. Тем более что все равно нашлись логические ловушки, не позволяющие описывать все до использования, например, процедуры, вызывающие друг друга. Вывод: если описания в языке нельзя писать в произвольном месте, это значит, что транслятор сделан не лучшим образом.

А транслятор, сделанный лучшим образом, распадается на три почти независимые части. Первая часть разбирает только описания и передает таблицу, построенную из них, второй части. Вторая часть – это анализ программы и составление внутреннего представления, сама трансляция. Наконец третья часть из внутреннего представления и данных таблицы генерирует команды для исполнения на заданной ЭВМ.

Каждую часть транслятора мог бы писать свой программист, каждая часть могла бы при работе затирать предыдущую, уже ненужную (так раньше и делали для экономии памяти). И хотя исходный текст приходится просматривать целых два раза, первый раз это делается быстро, поскольку большая часть текста при этом просто пропускается.

Теперь вернемся к главному вопросу о структуре языка и программы. Отцы-основатели еще в Алголе предложили до смешного мало.

Во-первых, программа может состоять из кусков, каждый из которых сам может состоять из кусков.

Во-вторых, запись алгоритма разбивается на ряд действий, которые так и выполняются «справа налево, сверху вниз», если только очередной элемент явно не указывает своего преемника.

В-третьих, имеется возможность не выполнять некоторые действия в зависимости от некоторых условий.

В-четвертых, имеется возможность повторять ряд одних и тех же действий заданное число раз.

Сами же действия в основном сводятся к записям, похожим на запись математических формул. Результату такой формулы назначается собственное имя и тогда такое действие называется присваиванием.

Конечно, в Алголе все это относилось к математическим, даже скорее к алгебраическим вычислениям. Но, тем не менее, эти предложения составляют суть и большинства других языков программирования, включая PL/1. Разумеется, есть множество других возможностей и особенностей, но, на мой взгляд, они уже вторичны, вытекают из этих.

И если наложить эти предложения, сформулированные еще в 1958 году, на три формы представления программы (действия, описания, иерархия) мы и получим основную структуру языка и программы.

После этого краткого рассмотрения всего «леса», начнем рассматривать отдельные «деревья».

4. Структура языка и программы на низком уровне

Язык должен быть организован так, чтобы в нем легко можно было бы выделить отдельные элементы. А для этого должны быть ясные границы начала и конца каждого элемента.

Символ конца каждого элемента был предложен еще в Алголе – это точка с запятой. Этот символ был на клавиатуре телетайпов и пишущих машинок. Он не очень часто используется в настоящих языках (в английском и еще реже в русском). Наконец, просто точка и просто запятая были уже заняты.

Можно было бы найти и какой-нибудь символ для начала элемента, но тогда транслятору все равно пришлось бы затем анализировать, что это за элемент начался. Целесообразней совместить границу начала элемента с указанием, что это за элемент. Так и организована структура всех элементов языка PL/1. Каждый элемент (кроме двух исключений, о них дальше) начинается с ключевого слова. Затем следует тело элемента, оканчивающееся точкой с запятой. Если тела нет – сразу идет точка с запятой.

Кстати, отцов-основателей Алгола такие конструкции без тела, как BEGIN; или END; сильно коробили (в первом случае точка с запятой стоит «слишком в начале», а во втором она вроде вообще ни к чему). Но практика показала, что такая организация легко воспринимается, гораздо легче попыток «не писать лишние точки с запятой». Транслятору также очень легко разбирать такую простую и всегда одинаковую структуру: ключевое слово → тело→ точка с запятой.

Мало того, элементы самого тела после ключевого слова часто могут следовать в любом порядке, что облегчает запись исходного текста.

И мало того, в ряде случаев можно разделить несколько тел запятыми и тем самым совместить в одном действии несколько однотипных.
Например, вместо:

```
CLOSE FILE(F1); CLOSE FILE(F2);
```

можно объединить действия и написать:

```
CLOSE FILE(F1),FILE(F2);
```

Транслятору легко разобрать такую конструкцию. Встретив запятую вместо конечной точки с запятой, он просто начинает разбор очередного тела сначала. И для программиста такая запись приятней: она и короче, и явно подчеркивает однотипность и неделимость этих действий. А в ряде случаев даже повышается надежность. Сравните

```
IF N>1000 THEN DO; CLOSE FILE(F1); CLOSE FILE(F2); END;
IF N>1000 THEN CLOSE FILE(F1),FILE(F2);
```

Во втором случае «скобки» из DO-END становятся ненужными, и исчезает потенциальная опасность их забыть.

Простое правило языка: «каждая конструкция начинается с ключевого слова» приводит к тому, что транслятор только один раз в начале конструкции ищет ключевое слово. Затем все слова для него – это обычные имена. Иногда это приводит к странным ситуациям. Я встречал в Интернете обсуждение возможности написать в PL/1 конструкции типа

```
IF THEN^=ELSE THEN ELSE=THEN;
```

Комментарии к такому примеру, как правило, сводились к тому, какие же идиоты авторы языка, что такое придумали и как же, наверное, трудно было в трансляторе такое реализовать. Но, как видите, на самом деле, никто таких конструкций в языке не придумывал и в трансляторе не реализовывал. Это всего лишь следствие из принятого правила, использованное каким-то остряком. Лично мне эта особенность языка не мешает. Если я забыл, что есть такое ключевое слово, то, скорее всего, я и не использую его в своей программе, а значит, вполне могу назвать этим именем свой объект.

5. Русский язык в исходных текстах программ

Национальность человека определяется тем языком, на котором он думает. Эта расхожая мудрость, по-моему, слабо учитывается в современном программировании.

Например, поиск в Интернете по запросу «кириллица+транслятор» дает очень скромные результаты. Т.е. даже возможность называть в программах переменные по-русски часто недоступна, что уж говорить о переводе на русский самих языков программирования. В советские времена, кстати, это было вполне естественно: например, служебные слова языка Эль-76 были русскими. Однако, с исчезновением БЭСМ и «Эльбрусов» исчезли и отечественные системы программирования.

На несчастных студентов и школьников сейчас выливается поток малопонятных английских слов и аббревиатур, превращая программирование в составление своего рода заклинаний. Конечно, через некоторое время для программиста набор этих слов становится привычным и об их смысле многие не задумываются. Один раз показали человеку, что здесь надо писать какой-нибудь «enum», он и пишет его как некий иероглиф, не привязывая к смыслу.

Мне возразят, что английский язык международный, что текст должен понять программист любой страны и т.п. А я считаю, что главное назначение языка программирования - помочь изложить свои мысли транслятору наиболее естественным для себя образом. Для русскоговорящего естественно излагать свои мысли и записывать алгоритм по-русски.

Конечно, перевод языка программирования на русский не должен приводить к результатам, которые получаются у украинских властей, заменяющих, к примеру, слово «проктолог» на «прямокишкознатец» (я не шучу). Скорее иллюстрацией к такому переводу должна служить сцена из фильма «Джентльмены удачи», когда герой Крамарова с удивлением читает собственную фразу, переведенную на нормальный язык: «этот нехороший человек предаст нас при первой опасности».

Через много лет работы на языке PL/1, я с удивлением понял, что заменить все ключевые слова на русские очень легко. И даже не заменить, а лишь дополнить русскими эквивалентами, чтобы не менять старых программ. Для этого можно использовать встроенный препроцессор, лишь слегка доработав его так, чтобы список слов был доступен еще до начала разбора программы.

Транслятор так же поступает и со стандартными встроенными в язык функциями. Он просто до начала разбора создает самый-самый внешний блок и переписывает туда таблицу описаний, как будто эти описания написал в начале программы программист.

Сложнее было придумать принципы перевода.

Для каждого английского ключевого слова необходимо было найти наиболее естественное отражение (по крайней мере, для автора) в русском, не считаясь с дословным переводом. Поясню примером. В языке есть отдельные операторы обмена для произвольных данных и отдельные – только для данных в виде текста. Одни называются READ/WRITE другие GET/PUT. Я специально спрашивал у коллег, не знакомых с языком, что у них ассоциируется с текстовым обменом, а что с произвольным. Конечно же, читать и писать у всех подразумевает текстовый обмен, но в языке как раз все наоборот. Второй пример. Смысл атрибутов файла STREAM/RECORD к моему удивлению никто из моих сотрудников внятно объяснить не смог. Только когда я предложил вариант перевода ТЕКСТОВЫЙ/НЕ_ТЕКСТОВЫЙ, до некоторых дошло, что же за файлы они открывают в своих программах.

Ключевые слова, которые обозначают действия и стоят в начале операторов, должны быть по возможности действиями - глаголами повелительного наклонения: ЧИТАТЬ, ПЕРЕВЕСТИ, ОТМЕНИТЬ. Конечно, есть исключения, например, ЕСЛИ или ЦИКЛ (а не ЦИКЛИРОВАТЬ).

Атрибуты при описаниях обозначаются прилагательными: ДВОИЧНОЕ, ДЕСЯТИЧНОЕ, КОСВЕННОЕ. Если не очень хорошо выглядят как прилагательные – можно писать термин вместе со словом «для»: ДЛЯ_ВВОДА, ДЛЯ_ВЫВОДА, ДЛЯ_ПЕЧАТИ.

Нужно стараться сделать слова короче, но не очень зажиматься. Язык придуман 45 лет назад, когда каждый символ набивали на перфокартах. Сейчас любой текст легко набирается и редактируется. Я отношусь к тем программистам, кто приветствует многословный текст, считая, что программа пишется, в общем-то, один раз, а читается многократно.

Нельзя сливать слова вместе, надо писать их через подчеркивание: на мой взгляд, исходное слово типа UNDEFINEDFILE это не ключевое слово, а просто безобразие.

Чтобы исходный текст был ближе к русскому, чем к ломанному русскому, надо позволять ключевое слово в разных формах: например, ФАЙЛ, ИЗ_ФАЙЛА и В_ФАЙЛ эквивалентны. Т.е. ЧИТАТЬ ИЗ_ФАЙЛА, но ПИСАТЬ В_ФАЙЛ. Вообще там, где для одних естественней одна форма, а для других - другая, разрешать и так и так.

Обрусевшие термины надо и переводить дословно: ФАЙЛ, БИТ, СТОП, СИГНАЛ.

Конечно, не надо отказываться от привычной математической записи и не переводить на русский SIN, COS и т.д. хотя здесь есть и разночтения, например, в математике нам все-таки привычнее ARCTG чем ATAN.

Постараться избегать аббревиатур, сделав несколько исключений для часто употребляемых слов. Сокращения должны выглядеть понятно и естественно (например, УКАЗ это УКАЗАТЕЛЬ).

Таким образом, вместо какого-нибудь

```
ON ZERODIVIDE PUT SKIP LIST('ОШИБКА В ДАННЫХ',STOP);
```

теперь пишем

```
КОГДА ДЕЛЕНИЕ_НА_0 ПЕЧАТАТЬ С_НОВОЙ В_ВИДЕ ('ОШИБКА
В ДАННЫХ',СТОП);
```

вместо

```
OPEN FILE(Ф1) PRINT TITLE('ПРОТОКОЛ.TXT') LINESIZE(250)
                                ENV(B(10000));
```

пишем

```
ОТКРЫТЬ ФАЙЛ(Ф1) ДЛЯ_ПЕЧАТИ ПО_ИМЕНИ
('ПРОТОКОЛ.TXT') С_ДЛИНОЙ_СТРОКИ(250) ДЛЯ_ОС(Б(10000));
```

а вместо

```
DO I=1 TO 100;
...
IF Z(I)<0 THEN LEAVE;
...
```

пишем

```
ЦИКЛ I=1 ДО 100;
...
ЕСЛИ Z(I)<0 ТОГДА ХВАТИТ;
...
```

Подчеркну, что речь идет только об исходных текстах, а собственно результат трансляции от этого никак не меняется. Текст программы удлинился, зато он стал естественней потому, что стало использоваться преимущество родного языка. Даже стало немного похоже на текст

инструкции на русском. На мой взгляд, это серьезное преимущество, которое особенно важно на этапе освоения языка. Весь словарь русских эквивалентов ключевых слов приведен в приложении. В дальнейшем в примерах я буду использовать русские ключевые слова. Язык и транслятор от этого ничуть не меняются.

Хотя нет. Отставить! Транслятор-то не меняется, но вот программисту бывает трудно изменить привычное восприятие. Из уважения к тем, кто уже привык, что во всех языках ключевые слова могут быть только английскими, я буду писать сразу две формы и даже попробую подкрасить их разным цветом, чтобы легче воспринимать при чтении. Тогда пусть каждый читает форму того цвета, которая ему понятнее.

6. Разбор меток и присваивания

Напомню, что пока мы касаемся только формы представления в языке, а не сути представления. Поэтому и к меткам и к присваиванию как к важным элементам мы еще вернемся. Пока же разбор меток и присваивания выделен отдельно потому, что они являются теми самыми двумя исключениями из общего правила построения элементов языка PL/1. Они начинаются не с ключевого слова, а с произвольного имени.

Поэтому в трансляторе приходится создавать довольно хитрый механизм «подсматривания вперед» только для того, чтобы разобрать эти конструкции. Цель такого «подсмотра» добраться до ближайшего символа равенства (тогда это присваивание) или до двоеточия (тогда это метка). При этом обычный разбор «не знает», что исходный текст уже прочитан. Предварительный просмотр «забежав вперед», затем направляет транслятор или на разбор метки или на разбор левой части присваивания.

Следует остановиться и на самом символе присваивания. Еще в Алголе озаботились тем, что должно быть отдельное действие «присвоить» в смысле записать куда-либо результат. К сожалению, уже не нашлось удобного значка для этого.

Лучше всего, наверное, подошла бы какая-нибудь стрелочка. Например, X←Y+1, т.е. сосчитать выражение Y+1 и результат записать под именем X. Увы, такого значка у телетайпов не было. Ближайшим подходящим значком оказалось равенство. Действительно X=Y+1 вполне можно интерпретировать, как вычислить и присвоить («приравнять») переменной X, хотя в математике так выглядит обычное уравнение, а не действие.

В ряде языков для присваивания все-таки ввели специальное обозначение «:=», чтобы подчеркнуть это особое действия, а заодно не путать с действием «сравнение». В одной умной книжке такое действие назвали «направленным равенством». Но практика, в том числе языков PL/1 и Си, показала, что опасения были преувеличены, и обычный знак

равенства вполне прижился как знак присваивания.

В PL/1 путаницы присваивания и сравнения не происходит потому, что сравнение не может быть на месте присваивания и наоборот. А вот в языке Си может, поэтому там введено отдельное обозначение сравнения как «==». Мне не нравится такое решение: и выглядит неестественно, и легко ошибиться (пропустить одно равенство), получить не то действие и не получить сообщения об ошибке при трансляции.

7. Разбор вызовов подпрограмм (процедур)

В завершении обзора форм представления в PL/1 следует обратить внимание еще и на вызов подпрограмм или процедур. Вообще-то это самый обычный элемент языка, составленный по общим правилам: сначала следует ключевое слово CALL или ВЫЗОВ, затем тело (в данном случае имя подпрограммы и, возможно, список параметров) и точка с запятой. Но здесь я сам рискнул отступить от стандарта языка и сделал в трансляторе это ключевое слово необязательным.

Выяснилось, что «запас прочности» представления настолько велик, т.е. структура элементов настолько однозначна, что транслятору не представляет труда разобрать еще один особый случай отдельного имени и точки с запятой. Транслятор проверяет, что это имя подпрограммы (процедуры) и дальше работает так, как если бы встретил это имя после ключевого слова CALL или ВЫЗОВ.

Конечно, некоторые отличия все же появились. Во-первых, имена процедур уже не могут совпасть с ключевыми словами (если это случилось, надо писать по-старому со словом CALL или ВЫЗОВ). Во-вторых, при ошибке в имени вместо диагностического сообщения «не описанное имя» выдается сообщение «это не процедура». Но, на мой взгляд, удобства от более краткой записи перевешивают эти мелкие неудобства.

Вероятно, если бы я еще пытался где-либо нарушить систему ключевых слов, то, скорее всего, получил бы логические ошибки и ловушки. А так, еще одно исключение (третье помимо метки и присваивания) не привело ни к каким неприятностям, и многолетний опыт подтвердил это.

8. Блочная структура

Теперь от формы представления перейдем к сути представляемых объектов. И начнем с очень важного свойства языка – его блочной структуры. Это свойство имеет два проявления.

Во-первых, выполняется один из основных заветов отцов-основателей: программа состоит из блоков, каждый блок может включать в себя другие блоки, однако их границы не должны пересекаться.

Во-вторых, это та самая иерархия программы, ее главные и подчиненные части. Вся программа это один блок в виде процедуры. Разбирая этот главный блок, транслятор может встретить другие блоки. В этом случае он запоминает место и состояние текущего разбора и начинает разбор как бы заново, но уже не всей программы, а только ее составной части. Затем, вернувшись из внутреннего разбора, продолжает с запомненного места и состояния. В результате сам транслятор становится очень простой и компактной программой. Он умеет разбирать только небольшое число элементов, практически только то, что предлагали еще в Алголе (присваивание, условие, цикл) и блоки.

Для описания блоков и иерархии программы в языке PL/1 применяются лишь две пары конструкций-границ: BEGIN; … END; или БЛОК; … КОНЕЦ; для обычного блока и PROCEDURE; … END; или ПРОЦЕДУРА; … КОНЕЦ; для блока-процедуры. Зачем нужны два типа блоков? Для гибкости.

Рассмотрим, что такое блоки вообще и зачем они нужны. Блок – это обособленная часть программы (в PL/1 с границами в виде указанных ключевых слов). А зачем нужно обособлять части программ? Чтобы из более простых частей можно было составлять более сложные. Обособленность позволяет этим частям не мешать друг другу при склейке в единое целое, т.е. обособленность здесь благо. Части могут создаваться даже разными людьми и в разное время.

Обычный блок в PL/1 – это просто кусок программы, вставленный в нужное место. Именно в этом месте он и будет выполняться столько раз, сколько раз само это место выполняется в программе.

И блок-процедура это тоже кусок программы. Но выполняется он не там, где вставлен, а там где его вызовут с помощью специального действия, уже упомянутого выше. И вызывать его можно сколько угодно раз в разных местах. И даже менять данные при каждом вызове можно с помощью специальных действий – передачи параметров, о которых мы еще поговорим.

Но раз процедура мощнее обычных блоков, почему не заменить их все процедурами? Потому, что иногда обычный блок удобнее. Например, я пишу отдельные процедуры для общего проекта. Я оформляю исходный текст в виде обычного блока, где записаны эти процедуры. Сам этот блок как кусок программы никогда исполняться не будет, вызываться будут процедуры, а блок в данном случае служит лишь «складом» для них. Кроме этого, блок может содержать общие для процедур части, например, описания, которые другим программистам не нужны и недоступны. Такой блок с набором процедур и данных для использования другими и, возможно, с процедурами и данными, которые используются только в нем, называется модулем. Есть даже целый язык «Модула-2», где такие модули – одна из главных целей.

Как видите, в PL/1 есть и модульность и блочная структура.

9. Блочная структура и «видимость» имен

Та самая обособленность, ради которой и была предложена блочная структура, проявляется через «видимость» или область действия имен.

В самом деле, если я назвал в своем блоке какой-то объект именем X, а другой программист в своем блоке назвал свой объект тем же именем, то, вероятно, мы с ним имели ввиду разные вещи.

С другой стороны, нам нужно с ним взаимодействовать, а для этого нужны какие-то общие объекты, именно с одними и теми же именами.

Как эти вопросы решает блочная структура? Отцы-основатели предложили простой подход. Если объект описан в каком-либо блоке, его именем можно пользоваться внутри этого блока и во всех блоках, входящих в данный блок. Но при условии, что в этих вложенных блоках нет своего, внутреннего описания с точно таким же именем. А если такое описание есть во внутреннем блоке, то оно отменяет или, как говорят, «затеняет» (тоже мне козырек!) имя из объемлющего блока.

Звучит заковыристо, но для транслятора это довольно простое правило. Встретив новое имя, он ищет его среди описаний текущего блока. Не нашел – идет искать в объемлющий блок. Не нашел и там – ищет в более старшем блоке и так пока не дойдет до самого верхнего блока-процедуры, т.е. всей программы. Если и там не нашел – ну все, значит, забыли вообще описать.

Этот принцип вполне естественен, но в языке PL/1 его дополнили возможностью прямо указать, что заданные имена общие для нескольких блоков, независимо от вложенности этих блоков. Для этого нужно у имени указать атрибут EXTERNAL или ОБЩЕЕ. Тогда, если в любом другом блоке также указать то же имя и тоже с атрибутом EXTERNAL или ОБЩЕЕ, это и будет одно и то же общее имя. Такой простой и одновременно мощный механизм «видимости» позволяет в PL/1 легко манипулировать областями действия имен и создавать очень большие программы.

Уже после PL/1 появились языки программирования, где правила видимости были значительно усложнены и расширены. На мой взгляд, это не всегда оправданно.

Приведу пример из собственной практики. Много лет назад я создавал транслятор для персонального компьютера с очень специфического языка. В этом языке правила видимости были очень жесткие: для каждой переменной требовалось перечислить имена всех процедур, которым разрешалось этой переменной пользоваться. Когда я начал отлаживать транслятор на уже готовых программах, то получил много сообщений, что эти правила не выполняются. Я решил, что в моем трансляторе ошибка. Но оказалось, что ошибка в «большом» трансляторе и он пропустил часть программ без указания видимости, а программисты были довольны тем, что не получили сообщений об ошибках. К чему это я? Ведь если бы в том трансляторе не было ошибки,

он бы заставил программистов указать нужные имена. Конечно, боюсь только, что они бы также бездумно их ставили, пока транслятор не перестанет «ругаться», как раньше бездумно их вообще не ставили. И сложные правила видимости не заставили программистов думать и не улучшили качество разработки, хотя для этого и были введены.

Теперь давайте посмотрим, а что делает транслятор, чтобы обеспечить такую видимость. Кстати, у языка Си правила видимости, на мой взгляд, похуже. Может быть, действительно у авторов просто не было перед глазами распечатки какого-либо транслятора?

Ничего такого особо сложного транслятор не делает. Границы блоков не пересекаются, значит, если заносить имена в таблицу по мере нахождения описаний в программе, они так и лягут в таблицу кучками поблочно.

Для PL/1 как раз не совсем так. Ведь здесь описания могут идти в любом месте блока. Например, в программе могут идти описания, затем внутренний блок со своими описаниями, а затем продолжатся описания текущего блока. Поэтому транслятору приходится описания внутренних блоков сдвигать в конец таблицы, а когда текущий блок закончится, обратно передвигать их, но уже за описаниями текущего блока. В конце концов, в таблице все устанавливается в строгом порядке: сначала описания самого внешнего блока, затем описания вложенных блоков и т.д.

Таким образом, если имя встречается в нескольких блоках, оно и в таблице транслятора запишется в нескольких местах с указанием, к какому блоку оно относится. А сам блок в таблице характеризуется одним числом – «вложенностью». Самый внешний блок имеет вложенность ноль, а самый «вложенный» - самое большое значение. Обеспечение видимости означает для транслятора поиск имени в кучках с вложенностью блоков меньше текущей. И все.

Механизм же общих имен (с атрибутом EXTERNAL или ОБЩЕЕ) от транслятора вообще никаких действий не требует, для него это обычное блочное имя, а механизм видимости здесь включится гораздо позднее – на этапе генерации объектного модуля с его общими сегментами.

10. Пример не очень полезной программы

Чтобы перевести дух и уложить в голове изложенное, давайте разберем пример программы. Я не одобряю этот глупый «Здравствуй мир!», но традиция есть традиция. Итак, текст программы:

```
hello:procedure main;
put('Hello, world!');
end hello;
привет: процедура главная;
печатать('Здравствуй, мир!');
конец привет;
```

Давайте разберем структуру этого шедевра. Вся программа это блок, начальная граница которого начинается ключевым словом procedure или процедура и заканчивается (граница!) точкой с запятой. У границы есть даже тело в виде атрибута main или главная.

Заканчивается вся программа-блок «задней» границей: ключевое слово end или конец, тело в виде имени и точка с запятой.

Сама программа состоит из одного элемента-«оператора», который, как и положено, начинается с ключевого слова «put» или «печатать» и продолжается до точки с запятой. Что именно может встретиться между ключевым словом и точкой с запятой транслятор догадывается по конкретному ключевому слову put или печатать. Кстати, самую-самую последнюю точку с запятой можно и не ставить. Но, по-моему, это исключение даже не стоит того, чтобы о нем помнить.

Ловко используется место «тела» после слова end или конец. Там можно написать (а можно и не писать) имя процедуры, в данном случае всей программы. Если там имя стоит, транслятор проверит его на соответствие имени в начале блока. Это удобно, если где-то обсчитались в границах блоков. Тогда имя станет не соответствовать, транслятор сообщит об этом и нужно искать ошибку выше в тексте.

Даже в таком микроскопическом примере у меня есть отступление от стандарта языка. В стандарте атрибут main или главная должен быть закопан внутрь еще одного атрибута. Вероятно, идея здесь была в том, чтобы все параметры, относящиеся не к языку, а к окружающей среде были выделены в одном месте в программе (в атрибуте OPTIONS). Но здесь авторы явно перестарались. Атрибут, указывающей, что эта процедура самая главная и именно с нее надо начинать выполнение всей программы – это, конечно, понятие самого языка, а не операционной системы. Я позволил себе смелость и здесь поправить транслятор, чтобы главную процедуру можно было указывать безо всяких выкрутасов.

Может показаться, что вообще указание о «главности» лишнее: здесь и так все ясно. Не скажите. Без атрибута main или главная ниоткуда не следует, что мы хотим запускать эту процедуру отдельно, а не вызывать из другой процедуры.

11. Типы данных

Ох уж эти типы! В последние годы в программировании им уделяется исключительное внимание. Почему-то многие уверены, что чем более развита система типов данных в языке, тем легче программисту будет решать задачи. Проблема в том, что задачи-то совершенно разные и подходы к ним разные. Волшебной палочкой или, если хотите, «серебряной пулей» типы не стали. При слове «тип» мне почему-то обязательно вспоминается старая шутка: «вчера была зарегистрирована партия нового типа, личность типа выясняется».

Если говорить серьезно, язык PL/1 создавался до эпохи «типизации», тогда главными задачами считались вычислительные, математические. То, что в описаниях языка гордо именовалось типами, на самом деле является более узким понятием - числовыми типами, числами.

Сложности возникали не с самими числовыми типами, а в случаях встречи разных представлений чисел в одном выражении. Для вычисления результата нужно было их привести к одинаковому виду.

При поиске в Интернете материалов о PL/1 я во многих местах натолкнулся на книгу Кернигана о языке Си, так как там упоминался и PL/1. Найденный контекст прямо преследовал меня одной и той же фразой: «он (язык) не будет преобразовывать данные с буйной непринужденностью PL/1». Но PL/1 создавали вовсе не буйно-помешанные. Просто преобразования разных представлений чисел транслятор пытается провести так, чтобы максимально сохранить точность.

Непонимание этого процесса программистами и приводит часто к неожиданным результатам, например, к переполнению при записи совсем вроде простого выражения 25.0+1.0/3.

Что касается «настоящих» типов, то в языке PL/1 их, на мой взгляд, вполне достаточно. Точнее, их столько, сколько пришло в голову создателям языка. (Честно говоря, я тоже больше ничего не могу придумать). Кроме чисел (разной степени точности), это файлы, файлы-переменные, метки, метки-переменные, процедуры-переменные, указатели, текстовые строки и даже «сырые» биты. Для большинства этих типов преобразования не нужны, да и невозможны. Поэтому и проблем с использованием таких типов нет.

Давайте теперь рассмотрим типы подробнее.

12. Числовые типы

Числа, это, наверное, величайшая абстракция, придуманная человечеством. Каких высот достигла математика! Да вот беда, у них, у математиков числа хорошие: и непрерывные, и какой угодно точности.

А у нас, у программистов какие-то убогие: дискретные, да еще и приближенные (а все потому, что ограниченная длина). Вот и пришлось изворачиваться, вводить в языках программирования целые и вещественные числа вместо просто чисел вообще.

Создатели языка PL/1 пошли еще дальше, ведь язык должен был использоваться в экономических расчетах. У меня впечатление, что авторы остальных языков (кроме, разумеется, Кобола) об экономических расчетах имели весьма смутное представление. На уровне «опись-пропись, отпечатки пальцев» и странной фамилии «итого».

В экономических расчетах важна точность, там не должно получаться 0.9999 копеек. Я сам верил легендам о программистах, которые разницу между точным и округленным значением ухитрялись перечислять на

свой банковский счет. (Пока не прочитал в книге Джадда «Обработка файлов», изданной еще в 1962 году, что все это сказки, придуманные испуганными комиссиями по проверке банковских программ.)

Так вот, поэтому в PL/1 числа, т.е. числовые типы, делятся не на целые и вещественные, а на точные и приближенные. А подмножество точных чисел – это целые числа. Что такое точные, но не целые числа? Это такие числа, которые складываются, умножаются и т.п. как будто на бумаге «столбиком». Это, конечно, медленнее, зато совершенно точно. Кстати, в принципе такие числа можно сделать очень большой длины и, значит, очень большой точности. На практике, обычно длина таких десятичных чисел с дробной частью не превышает 15 десятичных знаков (и еще знак плюс-минус) и занимает 8 байт.

Наличие дробной части, это очень большая забота для транслятора. Приходится все время следить за ее длиной при арифметических операциях, точно так же, как приходится в уме следить за положением десятичной точки при умножении «столбиком».

Следствием возможности точных вычислений является случай, когда Вы пишете числа-константы и не указываете, что они могут быть приближенными (для PL/1 это показатель степени «E»). Транслятор с испугу считает, что теперь он должен обязательно обеспечить Вам точный результат (ну а Вы бы как поступили?). Именно поэтому невинное выражение 25.0+1.0/3 вызывает непонятное переполнение. Сначала получается три в периоде, которые занимают все 15 разрядов, затем к ним до кучи требуется еще два разряда перед точкой и происходит крах. Стоит всего лишь записать выражение как 25.0+1.0E0/3 и ничего страшного уже не происходит.

Эти случаи разжеваны в учебниках. Но есть программисты, которые, даже не подозревая об этих ненужных им точных расчетах, ругают PL/1. ставя в пример другие языке, где таких проблем нет. Но там и не ставилась задача точных вычислений.

Остальные множества целых и вещественных числовых типов в PL/1 не отличаются от подобных типов в других языках. Ну, разве что в данном трансляторе нет беззнаковых чисел. У таких чисел нет знакового разряда (т.е. они только положительны), зато их максимальное значение в два раза больше, чем у обычных за счет дополнительного разряда вместо знака.

13. Комплексные числа

Я горжусь тем, что PL/1 один из немногих языков, где комплексные числа это «законные» объекты самого языка. В исходном трансляторе, правда, их не было, но ввести то, что уже давно было разработано и введено в язык гораздо легче, чем придумывать самому. Это я вот к чему: попалась мне как-то книжка «Язык C# для начинающих», где автор в качестве примера вводил класс комплексных чисел, используя

возможности языка. И все бы ничего, да вот после того как были созданы операции сложения, вычитания и умножения, удивила фраза «а операция деления для комплексных чисел не предусмотрена». Как же так? Да возьмите любой справочник, например М. Я. Выгодского по элементарной математике. На странице 194 имеется целая глава 39 «Деление комплексных чисел», а для сомневающихся еще и на странице 205 имеется глава 45 «Геометрический смысл деления комплексных чисел». Правда, в новом учебнике это могла быть всего лишь техническая ошибка. После исчезновения в издательствах нормальной корректуры, в научной и технической литературе приходится читать еще и не такие ужасы. А ведь кроме деления для работы нужно много еще чего: извлечение корня, экспонента, синусы-косинусы и т.п. Я все это клоню к тому, что не надо самому придумывать то, что давно уже придумано и используется людьми не нашего уровня ума. Конечно, невозможно втиснуть в язык и транслятор все богатство математических понятий и методов. Но минимум необходим, а для научно-технических расчетов в него явно входит и понятие комплексных чисел. Иначе пользователи будут городить все по своему разумению и обязательно наломают дров.

Комплексные числа в «полном» PL/1 приводят к очень громоздким преобразованиям. Например, в операторе X=1+2i; сначала действительная 1 дополняется нулевой мнимой частью, затем «чисто мнимая» 2i дополняется нулевой действительной частью и полученные комплексные числа складываются. Чтобы сильно не менять транслятор, я придумал писать константы как X='1+2i'; т.е. в кавычках. Тогда и разбор упростился и «лишние» преобразования исчезли.

14. Строковые типы

В первых языках вообще не предусматривалась работа с текстами. В каком-то старом учебнике я читал, что «печать текста (в Алголе) нужна лишь для того, чтобы выводить комментарии к численным результатам». Сейчас текстовая обработка в программировании чуть ли не превосходит числовую.

Язык PL/1 был одним из первых, позволяющих свободно оперировать с текстовыми строками. Их можно склеивать и резать на кусочки, выковыривать из них отдельные символы и т.д. Можно сравнивать на «больше-меньше» и сортировать. Основным типом таких данных является строка переменной длины. Т.е. в результате присваиваний такая строка может стать или совсем пустой или раздуться до максимальной для нее длины.

Текущую длину строки всегда можно определить с помощью функции LENGTH или ДЛИНА. В описываемом трансляторе вообще максимальная возможная длина одной строки всего 254 символа. С одной стороны, вроде немного. Но практика показала, что в принципе

хватает. Во-первых, строку больше 100-150 символов вряд ли кто станет печатать на принтере. Во-вторых, в файлы можно выводить и из файлов читать строку любой длины, создавая ее из нескольких коротких строк.

Есть и еще один тип данных – строка всегда постоянной длины. На мой взгляд, в язык его ввели зря, и практика показала, что без этого типа вполне можно было обойтись.

Но в целом, работа с текстами удобна, а набор функций работы с текстами достаточно полный. (Я опять ничего сверх этого придумать не могу.) Удивительно, но в некоторых более поздних языках (на мой взгляд, и в Паскале и в Си) работа с текстами менее удобная и менее полная.

Создатели PL/1 предложили программистам и возможности манипулирования просто «сырыми» битами. По-моему, была удачно подмечена общность такой работы и работы с текстами. Т.е. цепочки бит – это такие же символьные строки, только символы могут быть или нулями или единицами. Их также можно резать, склеивать. И поэтому в языке нет «лишнего» набора функций – он почти един и для бит и для символов.

К битам можно еще применять функции Булевой алгебры, которые встроены в язык. Конечно же, для простоты битовые строки короткие (сейчас только до 32 бит). На практике мне пригодились почти все возможности языка по работе с битами, а их короткая длина не приводила к особым трудностям.

Очень приятным следствием наличия данных типа «битовая строка», является отсутствие необходимости иметь отдельно логические типы (которые считались обязательно нужными еще с Алгола). Логический тип, т.е. объект, который может быть только «истиной» или «ложью», это просто битовая строка длиной в один бит.

Как и в случае целых чисел (подмножество точных), логические типы являются подмножеством битовых. Лично мне очень нравится, что авторы PL/1 оперировали более общими понятиями, частные случаи которых сами представляют практическую пользу. В этом чувствуется продуманность, ясность.

Есть сложности с переводом этих типов в числа и обратно. Но эти сложности часто возникают от непонимания программистом, что он хочет получить. Например, я знаю случаи, когда пытались переводить битовые строки в числа и получали странные, на их взгляд, результаты. А на самом деле просто нужно было читать одно и то же содержимое, то как число, то как цепочку бит. Перевод там был просто не нужен.

А вот при переводе числового типа в текстовую строку есть тонкость. Сначала число переводится в текстовое представление во внутренней, служебной строке. А уж затем содержимое этой недоступной напрямую строки переносится в указанную программистом строку.

Например, если программист хочет перевести число N в строку S из двух символов (зная, что N не больше 99) и напишет S=CHAR(N,2); или

S=ТЕКСТ(N,2); то получит вместо числа пробелы. Это происходит потому, что в PL/1 символы в строке считаются слева. Из внутренней строки, у которой длина большая, чтобы поместилась любое число, берутся два крайних левых символа (здесь с пробелами). Это еще один редкий случай, когда пришлось чуть подправить транслятор и ввести указание, что строку иногда нужно брать, начиная справа, а не слева. Запись S=CHAR(N,-2); или S=ТЕКСТ(N,-2); возьмет два не левых, а правых символа, как раз со значением числа N. Признаком таких нестандартных действий является знак «минус» в заданной длине.

Проблема в том, что подобные эффекты могут получиться и просто в результате ошибок, например, программист почему-то считал, что S это не строка, а тоже числовой тип. Это действительно недостаток языка, подвергаемый заслуженной критике. Я пытался уменьшить вероятность подобных ошибок, добавляя в транслятор выдачу предупреждений, например, если при преобразованиях транслятор обрезает строку. Очень часто, это признак того, что программист хотел совершенно другое.

15. Тип данных метка

Еще раз возвращаемся к меткам в языке. Во-первых, метки обозначают имена процедур, во-вторых, они отмечают определенные места в программе, чтобы на эти места можно было бы перейти при исполнении программы.

В PL/1 метки не могут стоять в любом месте. Например, нельзя пометить описания, которые ведь на самом деле не имеют никакого места среди действий алгоритма. Несмотря на отрицательное отношение многих теоретиков к операторам перехода, а, значит, и к меткам, в языке PL/1 возможности и переходов и меток очень большие.

Метки, расставленные в программе, могут иметь индексы, как элементы массива, а в операторах перехода можно указывать у метки индекс в виде выражений. Таким образом, можно создавать сколь угодно сложные переходы типа переключателей.

Но и это еще не все. В PL/1 есть еще и тип данных метка, т.е. метки-переменные, которым можно только присваивать значения обычных меток-констант, сравнивать их на равенство и использовать в переходах. Это дает возможность модифицировать саму программу в процессе ее исполнения, формально не меняя текста. Не все признают пользу такой возможности. Даже знаток PL/1 Л.Ф. Штернберг, писал, что возможность присваивания метке-переменной значения метки-константы на практике бесполезна. Не согласен. В моей практике эта возможность пригодилась очень быстро, при программировании таких объектов, как «конечные автоматы». И все получилось быстро и удобно. В других языках обычно таких возможностей нет, разве что в Алголе метки можно было передавать как параметр процедуры.

16. Тип данных процедура

Этот тип данных очень похож на предыдущий. Процедура-переменная, это тип данных, которому можно присваивать имена процедур-констант. Полезность этого типа данных теоретиками не отрицается.

Например, если программируется интегрирование заранее неизвестной функции. В универсальной программе интегрирования ставим вызов процедуры-переменной. Затем присваиваем процедуре-переменной имя процедуры конкретной функции и проводим интегрирование.

Есть и более характерные примеры. Например, методы в классах объектно-ориентированного программирования обычно представляют собой процедуры-переменные, расположенные как отдельные поля в общей структуре класса.

Конечно, PL/1 не объектно-ориентированный язык, но с учетом таких возможностей это не особенно заметно.

17. Тип данных указатель

Тип данных указатель на область памяти появился в языках не сразу. И здесь PL/1 был одним из первых. Потребность в указателях появилась, когда программы стали создавать сложные объекты динамически, так сказать, на ходу.

Понятие указатель, т.е. адрес в памяти, где находятся нужные данные, быстро стало привычным. В PL/1 указатели применяются очень широко. Причем можно один раз описать, что к объекту программы нужно обращаться через указатель, затем присвоить указателю нужный адрес и далее забыть про него. Транслятор сам будет подставлять указатель при каждом обращении.

А можно наоборот, явно писать указатели (причем разные) при каждом обращении к объекту. Т.е. динамически обращаться к разным данным. У любого объекта можно узнать адрес с помощью встроенной функции ADDR или АДРЕС и запомнить его в указателе. В общем, гибкость получается необычайная.

Авторы PL/1 вообще много внимания уделили работе с указателями. Жаль только, что опять не нашлось подходящего значка для обозначения операции «применить указатель». И такой значок пришлось городить из знака «минус» и знака «больше».

Например, P->X означает обратиться к участку памяти с длиной и смыслом как у X по адресу, который записан в переменной P. Это еще называют косвенной адресацией. Можно задать и цепочку длиннее, тогда получится как в сказке по Кощея: игла в яйце, яйцо в утке, утка в зайце, заяц…

Например, P1->P2->P3->X; означает, что обращаются к памяти,

адрес которой указан в P3, а адрес P3 указан в P2, а адрес… На практике редко доходит до подобных случаев, но когда на лету создаются сложные объекты, которые сами содержат ссылки на объекты, бывает и такое.

Мне и здесь потребовалась очень маленькая доработка. В языке есть встроенная переменная NULL или ПУСТО, которая заведомо не является правильным адресом и предназначена для проверки, что в указателе (например, P) тоже нет адреса: IF P=NULL THEN … или ЕСЛИ P=ПУСТО ТОГДА… Так вот, мне потребовалось ввести еще и встроенную переменную NULL_PTR или ПУСТОЙ_УКАЗ, которая является указателем, всегда содержащим значение NULL или ПУСТО. Зачем такие сложности? Очень часто приходится передавать указатели как параметры процедур, а часто и передавать пустое значение. В таких случаях в PL/1 я не могу написать просто NULL или ПУСТО. Ведь это тип-адрес, а мне нужен тип-указатель. Теперь я пишу в этих местах NULL_PTR или ПУСТОЙ_УКАЗ, который не надо описывать. В действительности NULL или ПУСТО это просто ноль, а NULL_PTR или ПУСТОЙ_УКАЗ ссылка на адрес, где тоже ноль.

18. Тип данных файл

Отцы-основатели в Алголе не рискнули определить стандарт на обмен программы с внешним миром, хотя и понимали, что без такого обмена любая программа бесполезна: ее результат так и умрет в ЭВМ. Но уж больно разными, «нестандартными» были в то время сами ЭВМ. И опять практика (на примере Фортрана и Кобола) показала, что определить стандарт обмена не так уж и сложно.

В конце концов, основой обмена в программировании стали файлы. Дать хорошее определение понятию файл непросто. Например, название американского сериала «X-files» у нас перевели как «Секретные материалы». Ну, материалы это как-то расплывчато. Но в общем, да, такие наборы данных, которые находятся все вместе и самое главное, могут существовать и до и после работы программ.

С точки зрения транслятора тип данных «файл» – это просто указатель. Но указывает он всегда на довольно сложную структуру данных, где хранится вся необходимая информация о конкретном файле. Например, сколько уже прочитали из данного файла, какая у него длина, можно ли его менять и т.п.

При каждом обращении к файлу данные в этой структуре могут меняться. При «открытии» файла структура заполняется нужными данными, при «закрытии» - обнуляется. Обычно где-то там же находится и «буфер» файла, своеобразное «окно» связи с внешним миром. Обычно, для быстроты, обмен идет не «поштучно», а целыми такими «окнами».

В PL/1 кроме «обычных» файлов есть также и файлы-переменные. Они могут принимать значения файлов-констант, подобно меткам-

переменным и процедурам-переменным. Тем самым достигается общность типов данных и их логическая законченность. Нужны ли файлы-переменные? Мне это пригодилось, например, при выдаче результатов в файл четных и файл нечетных страниц. Я открыл два файла, затем в программе формирования очередной страницы попеременно присваивал файлу-переменной то одно значение, то другое. Поэтому в программе и вся выдача попеременно шла в разные файлы при одних и тех же операторах вывода.

19. Константы

Константы, которые приходится писать в программах, не вызывают сложности в понимании, но представляют большой кусок работы в трансляторе. Транслятор сначала разбивает весь текст на неделимые «атомы»: числа, имена, знаки препинания, которые по-научному называются лексемами. И константы как лексемы транслятор сначала просто записывает в свою таблицу. Конечно, для них уже никаких правил «видимости» нет, и если константа встретилась в программе несколько раз, например текст «ошибка», транслятор, естественно, хранит в таблице только один ее экземпляр.

В PL/1 некоторую сложность вызывают числа-константы. Было принято, что у них символ «точка» означает наличие дробной части, а символ «E» (показатель степени) означает, что число приближенное. Признак приближенности при написании констант часто забывают, и тогда включается механизм точных расчетов, часто вовсе и ненужный программисту.

В простых случаях, например, X=1.2; это никак не сказывается на программе. Но когда константы встречаются рядом в выражениях, то транслятор начинает переводить их в точное представление, а результат затем переводит в приближенное значение. Например, если X имеет вещественный тип, то оператор X=1.2E0+1.4E0; не приведет к лишним преобразованиям, а оператор X=1.2+1.4; заставит транслятор сначала точно сложить две константы, а затем результат 2.6 перевести в приближенное представление.

Текстовые строки пишутся в кавычках (обе одинаковы) и могут в конце иметь коэффициент повторения в круглых скобках, например строку из 80 звездочек можно записать как '*'(80). В трансляторе я перенес коэффициент повторения в конец текстовой строки, тогда (в отличие от стандарта языка) он перестал путаться с коэффициентом повторения при инициализации начальных значений. Например,

```
DECLARE X(10) CHAR(5) STATIC INITIAL((10) '*'(5));
ОПИСАНИЕ X(10) ТЕКСТ(5) ПОСТОЯННОЕ ЗАДАТЬ((10) '*'(5));
```

в этом примере внутри атрибута INITIAL или ЗАДАТЬ 10 – повторение констант в массиве, а 5 – повторение символа в константе.

Пришлось расширить представление текстовых строк-констант и по

причине свистопляски кодировок в Windows: в текстовом окне осталась кодировка от MS DOS, а в самой Windows кто-то умный решил кириллицу представить по-другому. Тут позавидуешь англоязычным программистам, даже не задумывающихся о подобных проблемах.

Таким образом, появился еще и «квалификатор» W, обозначающий кодировку Windows, например:

```
DECLARE
S1 CHAR(*) VAR STATIC INITIAL ('Привет!');
S2 CHAR(*) VAR STATIC INITIAL ('Привет!'W);
ОПИСАНИЕ
S1 ТЕКСТ(*) РДЛ ПОСТОЯННОЕ ЗАДАТЬ ('Привет!');
S2 ТЕКСТ(*) РДЛ ПОСТОЯННОЕ ЗАДАТЬ ('Привет!'W);
```

Первый символ строки S1 имеет код 8F.

Первый символ строки S2 имеет код CF.

Константы для битовых строк тоже пишутся в кавычках, после которых указывается основание системы счисления: двоичная, четверичная, восьмеричная или шестнадцатеричная, например:

'11111111'B или '3333'B2 или '377'B3 или 'FF'B4

'11111111'Б или '3333'Б2 или '377'Б3 или 'FF'Б4

Вряд ли кто сейчас пользуется архаичными системами Б2 и Б3, но они до сих пор остаются в трансляторе, символизируя общность подхода к представлению величин. Транслятор разбирает константы для всех представлений единым способом, подтверждая эту общность.

20. Комментарии

Сначала комментарии считались в языках второстепенным элементом потому, что единственное, что должен делать транслятор с комментариями – это пропускать их. Например, создатели Алгола предлагали писать что угодно между словом «END» и точкой с запятой и это казалось им удачным решением. Постепенно пришло понимание, что комментарий – это такой же равноправный элемент языка, как и другие. Во-первых, комментарий выделяет информацию в тексте программы, которую должен читать человек, а не транслятор. Во-вторых, комментарий позволяет убирать фрагменты программы, не выбрасывая их из текста, а оставляя, так сказать, «под рукой».

Получилось так, что авторы языка Си взяли вид границ комментариев из PL/1 и теперь в большом числе языков, включая и PL/1, комментарии выглядят как /*…*/. Кстати, в языке Си это было непродуманное решение и при делении с использованием указателей могло случайно получиться начало комментария. Конечно, один пробел все это уже исправляет, но, как говорится, «грабли» для несчастливых программистов были тем самым положены.

Поскольку комментарии – это то, с чем постоянно имеет дело

программист, я, конечно, не мог пройти мимо этого элемента и принялся совершенствовать транслятор в этой части.

Во-первых, я столкнулся с тем, что транслятор не разбирает структуру внутри комментария. Поэтому нельзя было закомментировать кусок программы, имеющий другие комментарии. Первый же конец вложенного комментария сразу воспринимался как конец объемлющего комментария, и получалась чушь. Поскольку в PL/1 вообще не используются квадратные скобки, я применил их для пропуска вложенных комментариев:

```
/*[
/*…*/
]*/
```

Теперь, встретив открывающую квадратную скобку, транслятор ищет закрывающую, пропуская и все вложенные комментарии.

Во-вторых, я ввел в транслятор «однострочные» комментарии в стиле языка Си, т.е. комментарии, идущие до конца текущей строки и начинающиеся с «//». Произошел, так сказать «взаимовыгодный обмен» между PL/1 и Си в части комментариев. Правда, в результате я и себе положил «грабли», например старый комментарий можно было вставить даже сразу после знака деления: `X=Y//*комментарий*/Z;` теперь же между косыми нужен пробел, а то получится однострочный комментарий. Но если серьезно, практика показала, что в языке совершенно необходимы и однострочные и многострочные комментарии, причем однострочные должны писаться проще многострочных.

В третьих, мне пришло в голову, что поскольку в «кодировке MS DOS» полно всяких псевдографических значков («уголков» и «полочек» для рисования таблиц на дисплее и распечатке), их тоже можно использовать для украшения текста программы и своеобразных комментариев. При чтении программы транслятор использует единую таблицу «перекодировки». Если в этой таблице на месте псевдографических значков поместить пробелы – сами значки можно будет писать в программе где угодно, среди имен и выражений. Разумеется, внутри текстовых констант в пробелы ничего не переводится. После этого, тексты наших программ стали выглядеть совершенно по-другому. Можно, например, длинной вертикальной линией с двумя уголками выделить начало и конец цикла, тянущегося на много строк. Можно нарисовать описание, как таблицу с колонками: колонка имен, колонка типов, колонка начальных значений. И эти колонки разделены одинарными или двойными линиями. Получается и красиво и, самое главное, очень наглядно. Причем, практически без каких-либо затрат в трансляторе.

Наконец, в-четвертых, я придумал нечто вроде условной трансляции для отладочной печати. Если в начале строки поставить символ «%» и за ним одну цифру от 0 до 9, то транслятор переводит эти два значка или в

два пробела или в «//», в зависимости от числа, заданного при запуске самого транслятора. Например, в тексте программе написано:

```
%2 PUT SKIP LIST(X);
%2 ПЕЧАТАТЬ С_НОВОЙ В_ВИДЕ(X);
```

Если теперь я запускаю транслятор, не указывая числа (по умолчанию ноль) или указав единицу, данная строка превращается в однострочный комментарий. Если же я запускаю транслятор и указываю число от 2 до 9, данная строка транслируется со слова PUT или ПЕЧАТАТЬ. Таким образом, не меняя текст программы можно менять подробность отладочной печати, а при окончательной трансляции убрать отладочную печать совсем. Разумеется, можно применять это не только для печати, а и для любых изменений текста программы.

21. Чтение лексем

Доставанием из исходного текста уже неделимых далее кусочков («атомов»-лексем), которые являются элементарными понятиями языка, занимается отдельная процедура транслятора с гордым названием лексический анализатор. Казалось бы, эта подпрограмма должна быть очень маленькой и простой. В действительности это довольно громоздкая часть транслятора, которая реализует некоторые свойства языка, например, свободный формат. В данном трансляторе есть аж целых семь уровней чтения исходного текста. Давайте потратим немного времени и рассмотрим их, потому что это позволяет лучше понять правила записи в языке.

Самый нижний, первый уровень собственно и читает символы (байты) или из файла с программой или из файлов, указанных в директиве %INCLUDE. Никаких тонкостей здесь нет, именно этот уровень обнаруживает конец текста и выдает в этом случае специальный код 1A.

Следующий второй уровень, используя результат предыдущего, ищет в тексте строчные константы в кавычках, комментарии и, самое главное, директивы транслятору, начинающиеся с символа «%». На этом уровне просто выбрасываются однострочные комментарии, начинающиеся с «//». Однако ничего с обычными комментариями и строками этот уровень не делает. Его задача определить, что %INCLUDE и другие директивы транслятору стоят не внутри комментария и не внутри строки. Тогда этот уровень обрабатывает их.

Следующий третий уровень просто запоминает символы, если их приходится «подсматривать» для разбора присваивания и метки. Когда в обыкновенном разборе дойдет дело до уже «подсмотренных» символов, они берутся из «буфера» этого уровня, а не из файла. В других языках такого уровня может и не быть.

Следующий четвертый уровень обеспечивает разбивку исходного текста на строки. Если бы не требовалось выводить распечатку

исходного текста или строку программы при обнаружении ошибки, то и этого уровня не потребовалось бы. Здесь подсчитывается номер строки, табуляция переводится в пробелы и происходит печать строки, если это нужно. По сути это уровень реализации свободного формата: исходные строки сохраняются и печатаются, но далее в самом языке они уже никак не используются.

Следующий пятый уровень тесно связан с предыдущим. Он заставляет сразу прочитать всю строку в свой «буфер», хотя вышестоящим уровням передает символы по одной штуке. Это как раз то место, где исходные строки исчезают, и остается только одна длинная цепочка байт.

Следующий шестой уровень и надо было бы назвать настоящим лексическим анализатором. Получая отдельные байты от пятого уровня, он составляет из них лексемы: числа, имена, строки, скобки и т.п. Именно этот уровень сообщает о неожиданном конце программы, если получает символ 1A посреди разбора лексемы. Здесь выполняются всякие перекодировки и коэффициенты повторения, а также выясняется, входит ли точка в состав числа или это отдельный символ-разделитель. Заодно все лексемы-имена проверяются в таблицах оператора %REPLACE.

Полный лексический анализатор реализован на последнем, седьмом уровне. Здесь определяются ключевые слова языка и составные действия типа «>=». Этот уровень интересен тем, что он разный для первой и второй частей транслятора (поиск описаний и собственно трансляция), потому что на этих этапах ищутся разные конструкции языка. Например, указанный оператор «>=» определяется как раз только на втором «проходе» транслятора, а при разборе описаний эти две отдельные лексемы просто пропускаются. Кстати, комментарии /* */ пропускаются только на этом последнем этапе.

Сложная семиуровневая структура позволяет производить на каждом уровне довольно простые и понятные действия. Обращаясь к лексическому анализатору транслятор каждый раз получает удобную готовую лексему, а если это имя – заодно и информацию не ключевое ли это слово языка. Как уже было рассказано выше, транслятор обращает внимание на эту дополнительную информацию при разборе начала каждого оператора языка и в нескольких других случаях.

После рассказа о всех типах данных и их представлении в PL/1 самое время поговорить о распределении памяти для них.

22. Распределение памяти

Память является одной из главных сущностей в программировании. Если хотите, это наша среда обитания. Все программы исполняются где-то в памяти. Все придуманные нами объекты тоже живут, пока для них

есть память. В старом учебнике я прочитал утверждение (типа мрачного пророчества), что «памяти, вероятно, всегда будет не хватать».

Поэтому вопросам распределения памяти в PL/1 уделено большое внимание. В язык даже ввели понятие класса памяти и этот класс транслятор записывает в свою таблицу для каждого объекта программы. Таких классов набежало целых семь. Некоторые может явно задать программист, некоторые назначаются автоматически. Перечислим эти классы.

Постоянная память или память для констант. Как нетрудно догадаться, изменить что-либо в такой памяти нельзя. В PL/1 этот класс памяти назначается только транслятором. Например, метка в программе относится к этому классу памяти.

Статическая память. Объекты с этим классом памяти как раз вполне можно менять. Этот класс памяти может назначать сам программист. Основное свойство такой памяти – она выделяется объектам на все время жизни программы. Объектам с такой памятью можно назначить начальные значения даже еще до начала работы программы.

Динамическая память. Объекты с таким классом памяти живут лишь пока активен блок или процедура, в которой они описаны. Начальные значения задать таким объектам нельзя. Признаюсь, что в данном трансляторе для простоты динамическая память почти не отличается от статической. Только для рекурсивных процедур транслятор начинает действительно относиться к динамической памяти как задумано.

Базированная память. Эта та память, которую программисту разрешено явно отщипывать по кусочку от имеющегося свободного места при выполнении программы. Обычно именно для этой памяти используются указатели и именно эту память часто теряют и портят.

Наложенная память. Объекты с этим классом памяти находятся там же где объекты, на которые они «наложены». Конечно, те объекты должны уже иметь «настоящую» память, статическую или динамическую. Этот класс позволяет обращаться к одному и тому же участку памяти как к разным объектам с разными типами.

Память для параметров. Этот класс памяти назначается транслятором всем параметрам процедур. Для программиста может быть этот класс и не важен, но транслятору от этого не легче. Формальные и фактические параметры реализованы механизмом, похожим на базированную память.

Память для возвращаемых значений у процедур-функций. Этот класс памяти тоже волнует только транслятор, которому приходится выделять память и для таких объектов.

Реализация распределения памяти составляет солидную часть всего транслятора. Это главная цель на этапе разбора описаний. Сам процесс распределения довольно прост. Транслятор определяет размер каждого объекта программы и выделяет в качестве начального адреса этого объекта адрес текущего «свободного места». А затем прибавляет размер объекта к начальному адресу и получает адрес следующего «свободного

места» для следующих объектов.

Для разных классов памяти и «свободные места» разные. Базированную память транслятор вообще не обрабатывает – она будет распределяться на этапе выполнения.

Зато динамическую память приходится распределять в два этапа: сначала отсчитывая «свободное место» от начала того блока, где встретились описания, а уже после разбора всей программы определяется абсолютное расположение в программе всех блоков. Затем адреса всех объектов с динамическим классом памяти пересчитываются в абсолютные (от начала всей программы), ко всем прибавляется начало соответствующего блока.

И в заключение стоит упомянуть о выравнивании памяти. Выравнивание не имеет отношения к языкам, но также является заботой транслятора. Физическая память ЭВМ устроена так, что даже если нужно достать только один байт, она для быстроты достает сразу кусочек побольше, из которого уже и достается нужный байт. Таким образом, если объект в программе имеет размер больше байта и попадает на границу двух порций, которыми всегда читается память, он (как тот слон в лифте) будет доставлен за два приема.

Чтобы уменьшить число обращений, транслятор проводит выравнивание по следующему принципу: объект в два байта и выравнивается на границу в два байта, объект в четыре байта - на границу четыре байта, объект в восемь байт - на границу в восемь байт. Обычно более длинные объекты выравнивать нет смысла.

При определении очередного начала объекта транслятор просто добавляет от 1 до 7 байт к текущему «свободному месту», чтобы оно стало кратно длине объекта. Такой прием дает ощутимое ускорение программы, но в памяти появляется россыпь неиспользуемых кусочков.

23. Агрегаты данных

Больше всего памяти требуют не одиночные объекты, а их объединения или, по-научному, агрегаты. Еще создатели первых языков поняли, что в программах наверняка потребуется одинаковая обработка одинаковых элементов и ввели понятия массивов.

Массив это набор одинаковых элементов, которые и в памяти расположены друг за другом. Место элемента в массиве определяется, конечно, только его порядковым номером, который называется индексом.

Элемент массива сам может быть массивом, и тогда получаются «многомерные» массивы, каждый элемент которых имеет несколько индексов.

При описании массивов указывается, в каких границах могут меняться его индексы. Причем нижняя граница не обязательно должна начинаться с нуля или единицы, например, у массива

УРОЖАЙ_ТЫКВ(1997:2009) минимально возможный индекс 1997 и поэтому в памяти резервируется место для урожаев не от новой эры, а всего лишь на 13 лет.

В PL/1 разрешено манипулировать не только отдельными элементами массивов, но и их группами, при условии, что в памяти они занимают непрерывную область. Например, если есть массив X(1:10,1:20), то X(10,10) – это одиночный элемент, а X(10) – это уже группа из 20 элементов. Такую группу можно переносить или обнулять одним действием, сравнивать с другой такого же размера и т.п.

Для транслятора обработка массивов не очень сложное дело. Если в программе индекс элемента где-то указан прямо константой, транслятор сразу рассчитывает место элемента в массиве и работает с таким элементом как с обычным «одиночным» объектом (не массивом).

Но, конечно, чаще индекс задается переменным и транслятору приходится генерировать команды умножения размера элемента на его индекс для получения адреса от начала массива. Если массив многомерный, то и таких команд получается много.

Главная неприятность при этом в том, что индекс в конкретном месте программы может из-за ошибки программиста выйти за пределы указанных при описании границ. Тогда обращение произойдет не к заданному элементу массива, а к другому элементу или вообще неизвестно к чему. Выход индекса за границы массива одна из наиболее тяжелых ошибок и одна из немногих возможностей у программиста испортить собственную, а иногда и чужую программу, да еще и с катастрофическими последствиями.

Конечно, транслятор может генерировать и команды проверки выхода индекса за границы. Беда в том, что таких команд будет, вероятно, много и они заметно замедлят работу. Поэтому есть возможность отключить проверки. Опытные программисты помнят, что выход индекса за границы – самый частый случай поломки программ и обычно начинают поиск ошибок с включения таких проверок.

По мере развития программирования появилась потребность не только в одинаковой обработке множества одинаковых элементов, но и в общей обработке разных элементов. Так появились агрегаты данных – структуры.

Типичный пример структуры во всех учебниках – учетная карточка сотрудника. Там есть фамилия, возраст, пол и т.д. все эти данные записаны разными типами: текстами, числами, битами. Какая же общая обработка может быть у этих данных? В основном запись в файл всех сразу и чтение из файла всех сразу.

Как элемент массива сам может быть массивом, так и элемент структуры сам может быть структурой. Вот только обратиться к элементу структуры по индексу нельзя, нужно указывать имя. Зато можно объединять массивы и структуры как угодно и получать массивы структур и структуры массивов сложного вида.

Рассмотрим пример структуры-массива.

```
1 Студент (1:100),
 2 Фамилия текст(254) разной_длины,
 2 Успеваемость,
  3 Курс_1,
   4 Математика точное(7),
   4 Физика     точное(7),
  3 Курс_2,
   4 Математика точное(7),
   4 Физика     точное(7);
```

Цифрами в PL/1 вполне естественно обозначаются «уровни» структуры при ее описании. Самый верхний уровень (всегда единица) является заодно и признаком, что это вообще структура. Возможности обращений при использовании структур становятся очень велики.

Можно, например, записать сразу все данные всех студентов или только для одного, или отдельно данные по курсам. Обращение к отдельному полю структуры, правда, усложнилось. Приходится для однозначности перечислять и вышестоящие имена через точку.

Например, конечно, нельзя указать просто Математика(10) для десятого студента (так как непонятно для какого курса). Нужно указать Курс_1.Математика(10) или Курс_1(10).Математика. В PL/1 это одно и то же.

Уровни Успеваемость, Курс_1 и Курс_2 представляют «пустые» уровни или подструктуры, основное назначение которых, позволить оперировать входящими в них данными как единым целым. Как и в случае обычных массивов можно оперировать и группами данных, если они в памяти занимают непрерывную область. Поэтому, например Курс_1(10) – законный объект программы, а вот обратиться сразу ко всем полям Математика не получится, они в памяти идут с промежутками.

Для транслятора разбор таких составных имен структур довольно хлопотное дело. К тому же, обычно еще добавляются и указатели, так как часто структуры создаются в памяти (базированного класса) во время работы программы.

Главная сложность для транслятора – это убедиться, что указанная цепочка имен однозначно приводит к единственному требуемому имени. Разбор имени в таком общем виде (с учетом возможных структур) это еще один солидный кусок во всем объеме транслятора. По пути разбора цепочки имен транслятору могут попасться уровни-массивы со своими индексами, потом «пустые» уровни и т.д.

Причем, пока транслятор не доберется до последнего заданного имени, он даже не может узнать, какого типа у него получится этот объект. Кстати о типах. Для программиста структуры и массивы не добавляют новых типов, поскольку они являются объединениями уже известных типов. Но для транслятора структуры и массивы по существу

образуют новый тип – составной. Транслятор учитывает, что такие типы можно лишь переслать в точно такие же или сравнить с точно такими же. В данном трансляторе объекты составного типа можно еще обнулять (присваивать нулю) или сравнивать с нулем одним махом.

24. Управляющие структуры

Мне уже надоело все время говорить о данных. Давайте рассмотрим, наконец, действия в программе.

Именно действия придумывает программист, создавая программу. Все остальное (описания, иерархия) так, антураж. Еще до появления ЭВМ было доказано, что сколь угодно сложную задачу можно решить, выполняя лишь несколько очень простых действий. К счастью для нас, программистов, машины, которые выполняют совсем простые действия, вроде машины Тьюринга, остались на бумаге. Иначе мы бы все застрелились, пытаясь на таких машинах хоть что-нибудь решить.

Но и на современных ЭВМ существует довольно ограниченный набор простых команд. И, по сути, только две команды: проверка условия и переход на какую-либо другую команду или по условию или безо всякого условия и позволяют организовать сколь угодно сложную программу. Все остальные действия в программе напоминают больше математические расчеты по формулам: что-то сосчитали, запомнили результат, потом этот результат где-нибудь используем.

Поскольку у истоков языков программирования стояли математики, они понимали, что переходы на другие действия не украсят запись алгоритма, а лишь запутают его. Было решено оставить транслятору удовольствие генерировать переходы, а для программистов придумать что-нибудь более человеческое.

Таких действий, скрывающих переходы, предложили два: условный оператор и цикл. Вместе с вызовом процедур они составляют трех китов, на которых и держится большая часть программирования. Эти три главные управляющие структуры составляют как бы каркас программы, а все, что между каркасом заполняется операторами присваивания. (Ну, пошутил, пошутил.) Конечно, есть и еще элементы, управляющие работой программы.

Но действительно, в «Пересмотренном сообщение об Алголе» почти ничего другого и не было. И транслятор с PL/1 отдельно разбирает в программе вызовы процедур, условный оператор, цикл и оператор присваивания. И это чуть не три четверти от всего разбора программы.

Давайте поподробнее остановимся на каждом из этих четырех элементов.

25. Оператор присваивания

Присваивание это действительно «тело» алгоритма. Его можно определить как назначение имени результату. По традиции это имя пишется слева от знака равенства, а действия, которыми мы получаем результат, справа. Все это очень близко к математической записи формул, поэтому запись типа X=Y+1; вполне понятна и людям, не имеющим отношения к программированию.

Для транслятора разбор присваивания представляет известную трудность потому, что (помните?) он начинается не с ключевого слова. К тому же имя объекта слева, обычно называемого переменной может иметь другой тип, чем типы объектов, написанных справа. Поэтому транслятор часто генерирует не только команды записи результата в нужный участок памяти, но и команды преобразования результата к требуемому виду.

В PL/1 можно объединить несколько одинаковых присваиваний в одно, например, вместо X=1; Y=1; Z=1; можно записать X,Y,Z=1; Преимущества такой записи не только в краткости, но и в явном указании, что действия выполняются одинаково и вместе. Например, при переносе такого присваивания в другое место нельзя забыть один оператор из трех.

Для транслятора такая форма записи, довольно трудная вещь. Ведь каждая из переменных, перечисленных через запятую, может иметь свой тип и, следовательно, свои команды присваивания. Данный транслятор, встретив запятую, начинает обманывать сам себя. Он сразу читает все до точки с запятой, а затем выдает вместо каждой запятой кусок прочитанного текста от равенства до точки с запятой. Таким образом, разбор опять идет как обычно по отдельности для каждого присваивания.

Есть и еще одна форма записи присваивания. С легкой руки языка Си, присваивание типа X=X+1; стали писать как X+=1; Я долго не хотел менять транслятор в этой части, но когда увидел, что IBM в своем трансляторе сделала так же, сдался. В данном трансляторе можно указать арифметические действия (плюс, минус, разделить, умножить), а также И, ИЛИ или ‖ перед знаком равенства и это означает, что выражение справа начинается с такого же имени, как и слева и сразу после имени идет этот знак. Например, X=X/2; можно написать как X/=2;

Практика показала, что для простых действий это и удобно и надежно и понятно. А вот для более сложных выражений я такой формой не пользуюсь. Например, даже запись типа X/=8+Y+Z; вместо X=X/8+Y+Z; лично у меня уже вызывает сложность понимания при чтении программы.

И еще, чтобы закрыть эту тему. Главная цель такой формы записи в языке Си была не краткость, а возможность указания транслятору, что

можно применить более эффективные команды.

Мое мнение: задача эффективных команд – это задача исключительно транслятора, а не программиста. Например, данный транслятор с PL/1 анализирует возможность использования таких команд независимо от формы записи и тратит на это не так уж много сил.

26.Оператор обмена

Оператор обмена как два одновременных взаимных присваивания во многих языках отсутствует или реализован как сложный оператор присваивания типа X,Y=Y,X;

Но поскольку сейчас в процессорах имеются явные команды типа XCHG, позволяющие обменивать значения объектов без создания промежуточного объекта такого же размера, было бы жалко не иметь возможности прямо указывать такое же действие и в программе. Точнее, жалко не иметь в языке то, что все равно умеет делать процессор. Правда, тогда необходим и новый знак операции, мне, например, больше всего понравился оператор вида X<=>Y; хотя знак операции здесь и состоит из трех символов.

Как бы то ни было, в данном трансляторе имеется разновидность оператора присваивания – оператор обмена, который часто используется при сортировках или, например, при работе с «семафорами» в параллельных процессах. Понятно, что «обмениваемые» объекты должны иметь одинаковый тип. Если их размер не превышает 4 байт, транслятор прямо генерирует команду XCHG, иначе используется цикл из этих команд. Таким образом, можно «обменять» и две переменные размером в байт и две структуры размером в 500 Мбайт.

27. Условный оператор

Я, кажется, не объяснил, что такое оператор? Ну, это то самое, что оканчивается точкой с запятой. Нечто вроде логически завершенного действия. Так вот, условный оператор – это витрина, визитная карточка программирования. Любая нормальная программа содержит условия. Если хотите, условные операторы – это признак наличия разумной жизни на Земле, потому что разум это и есть способность задавать и проверять условия.

Устроен условный оператор в PL/1 очень просто: обычный оператор (например, присваивание) предваряется записью условия. Передняя граница условия – это ключевое слово IF или ЕСЛИ, а задняя граница условия – это ключевое слово THEN или ТОГДА. Если условие выполняется, выполняется и оператор сразу после THEN или ТОГДА.

Разумеется, транслятор генерирует команду перехода, которая срабатывает или не срабатывает. Главное, в самой записи никаких явных переходов не просматривается.

Математики сообразили и то, что часто требуется выполнять или один оператор или другой. Поэтому условный оператор предложили иногда писать длиннее, дополнить его еще одним, альтернативным действием, а начинать такое действие ключевым словом ELSE или ИНАЧЕ. Иначе (извините за каламбур) потребовалось бы писать два условных оператора друг за другом и во втором проверять условие наоборот.

Конечно, оператор внутри условного, сам может быть условным оператором. Например:

```
IF X < 100 THEN Y=1; ELSE
IF X < 200 THEN Y=2; ELSE
IF X < 300 THEN Y=3; ELSE
                 Y=4;
ЕСЛИ X < 100 ТОГДА Y=1; ИНАЧЕ
ЕСЛИ X < 200 ТОГДА Y=2; ИНАЧЕ
ЕСЛИ X < 300 ТОГДА Y=3; ИНАЧЕ
                   Y=4;
```

Здесь несколько условных операторов, вложенных друг в друга как матрешки.

В Интернете мне попадался очень старый тест-шутка: «напишите программу: если переменная равна 1, сделать ее 2, а если равна 2, сделать ее 1». Потом разбирались разные ответы, вроде подхода математика: X=3-X; (Ага, а если, не один и не два? Ха-ха-ха.) Так вот, именно условными операторами можно не задумываясь написать и именно так, как задача поставлена:

```
IF X=1 THEN X=2; ELSE IF X=2 THEN X=1;
ЕСЛИ X=1 ТОГДА X=2; ИНАЧЕ ЕСЛИ X=2 ТОГДА X=1;
```

А, на мой взгляд, это еще и самая естественная форма записи.

В PL/1 условие в условном операторе должно строго отвечать или «да» или «нет». По типу результат должен быть строкой в один бит.

Правило: «если получился не ноль, то условие выполнено», которое применяется в некоторых языках, я не одобряю. По-моему такой подход подталкивает программиста к небрежному и нечеткому составлению условий. Правда в стандарте языка PL/1 такие фокусы тоже разрешены, но данный транслятор обязательно предупредит, что так делать не стоит.

28. Составной оператор

О составных операторах выше я вообще не говорил. Он появился лишь как следствие условного оператора. В самом деле, если нужно по условию выполнить не одно действие-оператор, а сразу несколько? Как быть? Ведь после слова THEN или ТОГДА можно ставить лишь один оператор. Правда этот оператор может быть даже целым блоком, например:

```
IF X=0 THEN BEGIN; Y=1; Z=2; END;
ЕСЛИ X=0 ТОГДА БЛОК; Y=1; Z=2; КОНЕЦ;
```

Создатели PL/1 решили в таких случаях предоставить дополнительные возможности, помимо блоков. Может быть, они были недовольны тем, что из-за такой ерунды нужно писать целый блок. А блок может иметь свои описания и будет казаться, что эти описания действуют только по условию. А может быть, просто хотели запись короче. И они придумали более простые «скобки», для указания, что несколько действий зависят от одного условия.

Такие «скобки» создают один составной оператор из нескольких операторов. А сами они сделаны из «вырожденного» цикла. Новых ключевых слов при этом не потребовалось. В условном операторе можно поставить оператор цикла, а если заголовок у такого цикла пустой – получается составной оператор. И все довольны. Все, кроме меня. Конечно, запись типа:

```
IF X=0 THEN DO; Y=1; Z=2; END;
ЕСЛИ X=0 ТОГДА ЦИКЛ; Y=1; Z=2; КОНЕЦ;
```

уже не сбивает с толку лишним блоком. И в английском оригинале ключевое слово называется «DO», не подразумевая именно цикл. Однако я здесь нашел еще более короткое решение. Среди значков теперь широко доступны и фигурные и квадратные скобки. Например, язык Си активно использует фигурные скобки и выглядит это вполне понятно. И я ввел фигурные скобки как синонимы DO;…END; или ЦИКЛ; … КОНЕЦ; Запись стала и короче и яснее, например:

```
IF X=0 THEN {; Y=1; Z=2; }; ELSE {; Y=3; Z=4; };
ЕСЛИ X=0 ТОГДА {; Y=1; Z=2; }; ИНАЧЕ {; Y=3; Z=4; };
```

Первые точки с запятой здесь выглядят чуть странно, но это уж особенность регулярной структуры PL/1.

Я даже пошел еще дальше и нашел применение составному оператору не только внутри условного оператора, но и так сказать, отдельно стоящему.

Часто в программе требуется выполнить некоторые действия только один раз, причем само это место в программе выполняется много раз. Т.е. нужно выполнить сложную инициализацию. Конечно, можно поставить условный оператор и внутри него так «испортить» само условие, чтобы он больше ни разу и не выполнился. Но я для этой цели использую отдельный составной оператор, перед которым стоит цифра «1», показывающая, что данный оператор должен исполниться только один раз, например:

```
1 {; НАЧАЛЬНОЕ_ЗНАЧЕНИЕ=X+Y+Z; };
```

К счастью для меня, использование единицы и фигурных скобок для составного оператора не нарушает систему ключевых слов PL/1.

Кстати, если транслятор встречает отдельный составной оператор (не внутри условного и не с цифрой 1), он выдает предупреждение. Ведь такой составной оператор просто не нужен, он ничего не выделяет. И без

него все вложенные в него действия точно так же будут выполнены.

29. Оператор цикла

Оператор цикла это, пожалуй, тоже вид составного оператора, но уже не «вырожденного» как в условном операторе. Между ключевым словом DO или ЦИКЛ и точкой с запятой пишется так называемый «заголовок» цикла, показывающий сколько раз и как будут выполняться все действия, написанные от заголовка до границы цикла END; или КОНЕЦ;

Помните, между словом END или КОНЕЦ и точкой с запятой можно вставлять имя для проверки? Так вот, я чуть подправил транслятор, чтобы можно было туда вставлять и имя, которое написано в заголовке цикла. Это приятный пустяк позволяет красиво оформить текст цикла. И, конечно же, телом оператора цикла может быть другой цикл.

Давайте посмотрим это на простом примере для циклов. Вот задача: какова вероятность, что Вам в автобусе попадется счастливый билет (номер из шести цифр) ? А вот решение в виде вложенных циклов:

```
ЦИКЛ X1=0 ДО 9;
  ЦИКЛ X2=0 ДО 9;
    ЦИКЛ X3=0 ДО 9;
      ЦИКЛ X4=0 ДО 9;
        ЦИКЛ X5=0 ДО 9;
          ЦИКЛ X6=0 ДО 9;
            ЕСЛИ X1+X2+X3=X4+X5+X6 ТОГДА СЧАСТЬЕ+=1;
          КОНЕЦ X6;
        КОНЕЦ X5;
      КОНЕЦ X4;
    КОНЕЦ X3;
  КОНЕЦ X2;
КОНЕЦ X1;
```

Чуть замысловато, но, по-моему, довольно ясно, что тут делается. Самый внешний цикл выполняется десять раз, а самый внутренний – миллион раз. Кстати, получается на миллион 55252 счастливых билета (т.е. примерно каждый двадцатый, вероятность один к двадцати).

Все особенности циклов определяется их заголовками, например:

```
DO I=0 TO 10;
ЦИКЛ I=0 ДО 10; или
DO I=10 TO 0 BY -1;
ЦИКЛ I=10 ДО 0 С_ШАГОМ -1;
```

Заголовки могут быть и довольно сложными, например:

```
DO I=1000e0 REPEAT(I/2e0) WHILE(Z(I)<=0);
ЦИКЛ I=1000e0 ПОВТОРЯЯ(I/2e0) ПОКА(Z(I)<=0);
```

В заголовке обычно указывается начальное и конечное значения той переменной, которая должна как-то меняться при повторных действиях цикла.

Начальное значение на радость транслятору задается обычным оператором присваивания и идет в заголовке первым. Затем можно в любом порядке указать конечное значение (после ключевого слова TO или ДО) и шаг изменения (после ключевого слова BY или С_ШАГОМ). Если шаг не задан, он будет считаться единицей, а если конечное значение не задано, цикл будет идти «до посинения», например, до переполнения указанной переменной.

Вместо изменения на заданный постоянный «шаг» можно указать закон, по которому будет меняться переменная, указав его в виде выражения в скобках после ключевого слова REPEAT или ПОВТОРЯЯ. И еще прямо в заголовке можно указать условие, по которому цикл надо закончить. Такое условие пишется также в круглых скобках после ключевого слова WHILE или ПОКА. Получается, что в заголовке описывается большая часть встречаемых на практике случаев организации цикла.

Удивительно, но программистам все-таки потребовался и еще один вид цикла – «бесконечный». Конечно, ничего бесконечного в мире нет, и даже из такого цикла программа как-нибудь да выйдет.

В PL/1 бесконечный цикл можно сделать, например, написав в заголовке только одно всегда выполняемое условие после слова WHILE или ПОКА. Но мне так не нравится, выглядит в тексте глупо, вроде условия «ПОКА(рак на горе не свистнет)». Я чуть поправил транслятор и сделал бесконечным цикл с заголовком из одного слова REPEAT или ПОВТОРЯЯ. Тоже не идеальное решение, но все-таки такой бесконечный цикл выглядит естественней, например:

```
DO REPEAT;
   GET FILE(Ф) LIST(X);
   …
END REPEAT;
ЦИКЛ ПОВТОРЯЯ;
   ЧИТАТЬ ИЗ_ФАЙЛА(Ф) В_ВИДЕ(X);
   …
КОНЕЦ ПОВТОРЯЯ;
```

Для транслятора разбор цикла сводится, главным образом, к разбору заголовка цикла. Из заголовка транслятор генерирует команды изменения указанной переменной и всякие проверки. Далее транслятор просто разбирает сам цикл как отдельный блок (или, точнее, как

составной оператор) до слова END или КОНЕЦ. В конце цикла транслятор обычно просто генерирует переход в начало.

С циклами связано немало тонких моментов. Например, в некоторых языках переменная цикла может быть только целой. В PL/1 переменная цикла может быть любого числового типа, но уж программист сам должен следить за тем, чтобы в результате округлений приближенных значений не получилось недостачи в числе повторений цикла.

Большинство трансляторов так генерирует команды, что при выходе из цикла в переменной цикла остается значение, при котором проверки из заголовка не выполнились, например, в приведенном выше примере с билетами по окончании циклов в переменных X1-X6 будет значение 10.

Несмотря на богатые возможности заголовков циклов, программистам потребовались еще средства управления циклами. Очень часто «внутри» цикла становится ясно, что этот цикл нужно вообще прекратить или бросить выполнять «этот раз» и перейти к следующему. Чтобы не ставить переходы, в языках есть специальные операторы для такого управления циклом «изнутри».

Транслятор вместо них генерирует переход на метку, которую сам же и ставит или перед словом END или КОНЕЦ; (цикл продолжается, текущее исполнение прекращается) или после слова END или КОНЕЦ; (цикл вообще прекращается).

Я придумал русские ключевые слова для таких операторов ОПЯТЬ и ХВАТИТ. Но это дело вкуса. В языке Си, например, оператор выхода называется «break» (очень похоже на бокс), а в английском оригинале PL/1 такой оператор был назван «leave» (может в честь старой песни «She's Leaving Home»?)

Как бы то ни было, такие операторы вполне обычное дело, например:

```
DO REPEAT;
   GET FILE(Ф) LIST(X);
   IF LENGTH(X)=0 THEN CONTINUE;
   IF X='КОНЕЦ'     THEN LEAVE;
…
END REPEAT;
ЦИКЛ ПОВТОРЯЯ;
   ЧИТАТЬ ИЗ_ФАЙЛА(Ф) В_ВИДЕ(X);
   ЕСЛИ ДЛИНА(X)=0 ТОГДА ОПЯТЬ;
   ЕСЛИ X='КОНЕЦ'   ТОГДА ХВАТИТ;
…
КОНЕЦ ПОВТОРЯЯ;
```

30. Сложные циклы

Думаете, автор совсем озверел, раз придумал еще какие-то «сложные» циклы? Ведь оператор цикла и так сложный и громоздкий элемент языка, куда еще сложнее-то? И тут ноги растут еще из Алгола.

Было предложено соединять несколько последовательных циклов в один, отделяя каждый заголовок от другого запятыми. Может быть, авторы-математики вспомнили о «кусочно-непрерывных» функциях, задаваемых разными законами на разных отрезках, а может быть просто хотели записать циклическую конструкцию в более общем виде. Разумеется, я и здесь не удержался и еще «усложнил» циклы, введя в свой транслятор правило, что каждый новый заголовок после запятой должен опять начаться с первого присваивания.

Это позволило сделать несколько заголовков в одном цикле полностью независимыми и менять в цикле не только закон изменения параметра, но и само имя параметра!

Например, вот задание прохождения фрезы станка с программным управлением по «квадратной» траектории:

```
Y=0;
DO X=0 TO 99, Y=0 TO 99, X=100 TO 1 BY -1, Y=100 TO 0 BY -1;
     ПЕРЕМЕСТИТЬ_ФРЕЗУ(X,Y);
END;


Y=0;
ЦИКЛ X=0 ДО 99, Y=0 ДО 99, X=100 ДО 1 С_ШАГОМ -1,
        Y=100 ДО 0 С_ШАГОМ -1;
         ПЕРЕМЕСТИТЬ_ФРЕЗУ(X,Y);
КОНЕЦ;
```

здесь на самом деле 4 цикла с двумя параметрами и когда один параметр меняется, второй – «замораживается» и остается таким, каким был при выходе из предыдущего цикла.

А вот другой пример: при работе с частными производными требуется определять «полное приращение» функции, скажем, от шести переменных. Конечно, можно написать длинное выражение с шестью вызовами этой функции, а можно использовать циклическую природу таких вычислений:

```
X=X0; Y=Y0; Z=Z0; U=U0; V=V0; W=W0; F0=F(X,Y,Z,U,V,W); DF=0;

DO X=X0+DX,Y=Y0+DY,Z=Z0+DZ,U=U0+DU,V=V0+DV,W=W0+DW;
     DF=DF+F(X,Y,Z,U,V,W)-F0;
     X=X0; Y=Y0; Z=Z0; U=U0; V=V0; W=W0;
END;
```

Здесь, по сути, и цикла-то в обычном понятии нет, однако есть повторяющаяся часть алгоритма с разными параметрами. Не очевидно, что такая запись будет работать быстрее или занимать меньше памяти, однако она элегантно и явно подчеркивает замысел программиста повторять одни и те же действия.

В языке Си тоже можно создать «сложные» циклы, поскольку внутри

каждого из трех выражений заголовка можно написать несколько конструкций через запятую. Однако, это, скорее, «параллельная», а не «последовательная» как здесь обработка.

31. Оператор вызова процедур

Последний из базовых «китов» программирования - это вызов процедуры или подпрограммы. Именно эта возможность позволяет строить алгоритмы из более простых частей, накапливать программистское «богатство» в виде библиотек готовых программ и вообще существовать в современном компьютерном мире.

Попробуйте, например, общаться с Windows, не обращаясь к ее 1500 основным процедурам! (Это к вопросу о 2000 страниц описания PL/1. А здесь сколько страниц ? И кто все это читал?)

Мощь и гибкость такого механизма подтверждается всей историей математики с ее обращениями к функциям. Есть даже целый язык Лисп, построенный на вызовах функций. Транслятору с Лиспа хорошо живется: ему надо разбирать лишь одну базовую конструкцию. Только вот читать человеку такие программы сложновато.

К счастью для всех трансляторов, процессоры современных ЭВМ имеют вполне адекватные команды вызова процедур, и генерация таких команд из операторов вызова не представляет никакой сложности.

Было бы еще проще, если бы надо было только «запускать» готовые кусочки программ и не передавать им ничего «внутрь». Но как раз соль и состоит в том, что можно передать в некую универсальную программу конкретные данные и выполнить с ее помощью конкретный расчет. Такие изменяемые данные называются параметрами процедуры. Собственно описание параметров и манипулирование ими составляет основную часть работы с процедурами в языках программирования.

Напомню, что процедура или подпрограмма в блочно-структурированном PL/1 занимает отдельный блок, а значит, имеет в программе свой отдельный мир со своими описаниями и областями «видимости».

Процедура начинается с имени-метки, за которым следует ключевое слово PROCEDURE или ПРОЦЕДУРА и «заголовок» процедуры до точки с запятой, например:

```
ИЗВЛЕЧЬ_КОРЕНЬ:PROCEDURE(X);
…
END ИЗВЛЕЧЬ_КОРЕНЬ;
ИЗВЛЕЧЬ_КОРЕНЬ:ПРОЦЕДУРА(X);
…
КОНЕЦ ИЗВЛЕЧЬ_КОРЕНЬ;
```

Хороший стиль программирования требует писать формально необязательное имя процедуры после слово END или КОНЕЦ. Во-первых, получается дополнительная проверка правильности всей

структуры программы. Во-вторых, расшифровывается назначение каждого «конца» в тексте программы, потому что, к сожалению, в PL/1 все «концы» одинаковые.

В заголовок процедуры кроме перечисления параметров в круглых скобках входят еще и характеристики самой этой процедуры. Это самое удобное место для них, поскольку их здесь можно перечислять в любом порядке и транслятор будет их глотать, пока не встретит точку с запятой.

Какие же характеристики процедуры могут быть в заголовке?

Прежде всего, процедура может быть MAIN или ГЛАВНАЯ. Вся программа - это как раз одна такая процедура, когда она кончается – кончается и вся программа. Именно с главной процедуры начинается работа программы и уже она может вызывать другие процедуры.

Главная процедура даже может иметь свой параметр – командную строку. Обычно это тот текст, который набирает на клавиатуре тот, кто вызывает данную программу.

Транслятор оформляет главную процедуру иначе, чем все остальные потому, что в начале главной происходит одноразовая подготовка к работе всей программы. Кстати, встретив параметр MAIN или ГЛАВНАЯ, транслятор не просто транслирует ее, но и пытается сразу создать полную программу. Когда вся программа состоит не только из этой процедуры, но и еще и из других файлов с процедурами, ничего, конечно, не получается, и транслятор помалкивает, делая вид, что он ничего и не пытался. Зато, когда вся программа состоит только из одного файла с главной процедурой, транслятору удается сразу получить полную программу без дополнительных указаний от программиста.

Процедура может быть RECURSIVE или РЕКУРСИВНАЯ. Это одно из фундаментальных понятий математики, органично перешедшее в программирование. Рекурсия (от слова «возвращение») - это определение чего-либо через «само себя».

Как Вам такое определение: факториал целого положительного числа равен самому этому числу, умноженному на факториал от этого же числа минус единица. Да так мы никогда и не узнаем, что такое факториал! Ничего, математики разорвали замкнутый круг, определив, что факториал нуля – это один.

Таким образом, на каждом рекурсивном шаге мы немного упрощаем общую картину, пока не дойдем до конечного, совсем простого шага, где рекурсия, наконец, исчезает.

Сам язык PL/1 насквозь рекурсивный: блок может быть последовательностью блоков, а условный оператор сам может состоять из условных операторов. Транслятор тоже весь рекурсивный и поэтому такой маленький и простой. Ну, так сделали бы все процедуры в языке рекурсивными (т.е. такими, которые могут сами себя внутри вызывать) и дело с концом!

Нет, это не лучшее решение. Дело в том, что при каждом вызове рекурсивной процедуры (вроде факториала) нужно запомнить все ее предыдущее состояние, например, все ее «местные» переменные, а это и время и память. А на практике рекурсивные алгоритмы не так уж и часто встречаются. Правда, если все переменные хранить в памяти-«стеке», то и лишних запоминаний не будет. Зато все обращения к таким переменным будут «двухступенчатыми»: адрес начала текущей копии данных плюс смещение для конкретной переменной.

Тогда пусть сам транслятор определит, рекурсивная процедура или нет, и потом сам же с ней разбирается! Так тоже не получается. Процедура может вызвать не просто саму себя, а другую процедуру из другого файла, а та возьмет и опять вызовет исходную. Как транслятор об этом может догадаться?

В PL/1 нашли очень простой выход: свалили все на программиста. Раз он пишет алгоритм, то сам должен понимать, рекурсивен алгоритм или нет. Укажет явно рекурсию – транслятор будет генерировать запоминание состояния процедуры, не укажет – транслятор ничего такого делать и не будет. Дешево и сердито.

Процедура может быть EXTERNAL или ОБЩАЯ. Это прямое управление правилами видимости. Процедурой можно пользоваться в том блоке, где описан ее текст, но с помощью этого атрибута можно разрешить пользоваться этой процедурой («увидеть» это имя) и в других блоках. Например, если я составляю библиотеку готовых процедур в отдельном файле, я, конечно, должен все их сделать общими.

Еще одна характеристика процедуры начинается с ключевого слова RETURNS или ВОЗВРАЩАЕТ, после которого в круглых скобках идет описание того, что же собственно она возвращает.

Такое возвращаемое значение подсмотрено у математиков, которые всю жизнь имеют дело с функциями, возвращающими некий результат. Наличие или отсутствие этого ключевого слова в заголовке поделило все процедуры в PL/1 на простые процедуры и процедуры-функции.

Наличие значения позволяет использовать процедуры-функции, так же как обычные переменные, например запись: X=SQRT(Y); обращается к встроенной процедуре-функции извлечения квадратного корня и очень похожа на те записи, которые делают математики.

Процедуры-функции породили также теоретические споры о том, должна ли и может ли процедура-функция что-либо менять в программе, помимо того, что она вычисляет свой результат-значение. Некоторые теоретики доказывали, что в «правильной» программе процедура-функция имеет право только считать свое значение и не имеет право ничего менять снаружи себя, т.е. не создавать «побочных эффектов» (будто она опасное лекарство). Пока теоретики спорили, программисты все перевернули с ног на голову. Например, в Windows все процедуры

теоретически являются процедурами-функциями. Но большинство используется как раз ради «побочного эффекта», а возвращаемое значение обычно лишь сообщает, удался или не удался такой эффект. Иногда вообще на такой ответ не обращают внимания.

Я тоже поддался теоретически «неправильной» моде. В данном трансляторе сделано так, что если процедура-функция возвращает «короткий» ответ (типа рапорта о выполнении, как у большинства процедур Windows), то можно нагло обращаться к таким процедурам-функциям как к простым процедурам. Конечно, ответ и в таком случае процедура-функция вернет, но он просто так и пропадет.

Наконец, процедура может быть IMPORT или ЧУЖАЯ. Не бойтесь, это не вражеская процедура, подброшенная нам злобными инопланетянами. Чужой здесь обычно выступает Windows. Иногда требуется, чтобы сама Windows вызывала написанные Вами, например, «оконные» процедуры. Windows написана на языке Си, а не PL/1 (это, конечно, ее серьезный недостаток!), поэтому правила передачи параметров не совпадают. Увидев такой атрибут, транслятор генерирует команды приема параметров по правилам Windows.

До сих пор мы говорили о заголовке, так сказать о «входе» в процедуру. Теперь поговорим о выходе из процедуры. Понятно, что если операторы внутри процедуры выполнятся до конца, процедура завершится, и дальше будет выполняться оператор, стоящий после вызова данной процедуры. Но для удобства можно выскочить из процедуры и когда угодно, используя оператор RETURN или ВОЗВРАТ.

Для процедур-функций RETURN или ВОЗВРАТ должен быть не простой, а со значением, которое записывается в круглых скобках после этого ключевого слова. В процедуре-функции не может быть простых возвратов и не может быть простого окончания процедуры без расчета возвращаемого значения. Транслятор обязательно предупредит программиста о таких ошибках.

Сам оператор возврата со значением к удовольствию транслятора представляет собой просто оператор присваивания без левой части, зато начинающийся с ключевого слова RETURN или ВОЗВРАТ. Кстати, если главную процедуру сделать процедурой-функцией, то ее возвращаемое значение в конце работы всей программы достанется Windows.

Честно говоря, что-то мы сильно ушли в сторону от самого оператора вызова. Вызов процедуры начинается с ключевого слова CALL или ВЫЗОВ (необязательного после моих переделок), затем идет имя процедуры и список передаваемых параметров в круглых скобках. Для процедуры-функции слово CALL или ВЫЗОВ тем более не нужно потому, что процедура-функция встречается уже только где-то в

середине операторов, например в выражениях.

Есть еще один нюанс. В некоторых случаях в операторах вызова перед списком параметров в круглых скобках может стоять еще один список и тоже в круглых скобках. Помните, есть такой тип данных процедура-переменная? Ведь этот тип может быть сам по себе массивом значений процедур-констант.

Например: X=ИЗВЛЕЧЕНИЕ_КОРНЯ(3)(Y); означает, что Вы извлекаете корень по рецепту №3. Т.е. Вы завели массив из процедур-функций и каждому элементу такого массива присвоили значение конкретной процедуры-функции. Эти конкретные процедуры-функции могут иметь разные имена, но все они принимают один параметр и возвращают значение. В программе можно сложным образом вычислить нужный номер и вызывать любую процедуру единственным общим оператором. Ну, а транслятору приходится проверять не процедура-переменная ли это и не имеет ли она размерности. Если это так, что сначала должны идти индексы, а уж затем параметры.

О передаче параметров мы еще поговорим, а сейчас давайте вернемся от действий в программе назад к сопроводительной информации, к описаниям.

32.Описания в программе

Описания в PL/1 делают программы довольно многословными. Транслятору желательно как можно больше узнать об объектах, с которыми нужно будет выполнять указанные действия. Чем больше такой информации ему будет доступно, тем теоретически лучшую программу он сможет создать.

Когда-то длинные тексты описаний вызывали неудовольствие. Ведь каждый символ нужно было превратить в дырки на перфокарте или перфоленте. Старались возможно короче записать исходный текст. Для этого из Фортрана в PL/1 притащили правило, что если в программе есть неописанное имя, значит это обычная переменная, тип которой зависит от первой буквы имени. Но практика показала, что это плохое правило. Например, я работаю в программе с двумя переменными X1 и X2. Набирая текст, промахнулся пальцем и вместо X2 написал X3. Представляете, что будет? Транслятор радостно решит, что я завел новую переменную X3 и запишет в нее результат, не говоря мне об этом, а я потом буду удивляться, почему в X2 не то.

В новом стандарте языка PL/1 (он называется «подмножество общего назначения G» или стандарт X3.74) вернулись к более естественному правилу: почти все, что программист использует, он должен явно описать транслятору.

Доля описаний в исходных текстах на PL/1 увеличилась, но я приветствую это увеличение. Я вообще люблю много всего писать в исходных текстах, например, ссылки на книги, откуда взял алгоритмы (а

то все забывается). Пусть и полный список объектов, что я использую в программе, будет в ее исходном тексте, а не где-то на клочке бумаги.

Оператор описания естественно начинается с ключевого слова DECLARE или ОПИСАНИЕ и продолжается до точки с запятой. Каждый элемент в описании отделяется друг от друга запятой.

Описания могут быть довольно сложными, но можно наоборот, разбить их на множество простых и размещать в программе почти где угодно (конечно, с учетом блочной структуры и «видимости»). Обычно описания собирают в общую кучку в начале каждой процедуры или даже записывают их в отдельных файлах.

Создатели PL/1 озаботились тем, чтобы писать описания все же покомпактнее. Им пришло в голову разрешить выносить «общие члены» за круглые скобки, так, как это делают математики, упрощая выражения. Например, вместо описаний

```
DECLARE
X1 (1:100) FIXED(31),
X2 (1:100) FIXED(31),
X3         FIXED(31);
ОПИСАНИЕ
X1 (1:100) ТОЧНОЕ(31),
X2 (1:100) ТОЧНОЕ(31),
X3         ТОЧНОЕ(31);
```

можно написать

```
DECLARE
(X1,X2) (1:100) FIXED(31), X3 FIXED(31);
ОПИСАНИЕ
(X1,X2) (1:100) ТОЧНОЕ(31), X3 ТОЧНОЕ(31);
```

или даже так

```
DECLARE ((X1,X2) (1:100),X3) FIXED(31);
ОПИСАНИЕ ((X1,X2) (1:100),X3) ТОЧНОЕ(31);
```

Из-за рекурсивного разбора транслятор легко ориентируется в скоплениях круглых скобок (так же, как, например, транслятор Лиспа), главное, чтобы сам программист не запутался.

Собственно описание заключается в перечислении всех характеристик очередного объекта после указания его имени. Все эти характеристики транслятор в виде нулей и единиц заносит в таблицу вместе с именем.

Характеристики объекта программы обычно существуют парами, и одна исключает другую. Например,

BINARY/DECIMAL ДВОИЧНОЕ/ДЕСЯТИЧНОЕ,
FIXED/FLOAT ТОЧНОЕ/ВЕЩЕСТВЕННОЕ,
STATIC/AUTOMATIC ПОСТОЯННОЕ/ВРЕМЕННОЕ и т.п.

Некоторым исключением в описании является указание пар границ массивов. Если они указаны в круглых скобках, то обязательно должны идти сразу после имени, например:

```
DECLARE X(1:100,1:100) CHAR(254) VARYING;
ОПИСАНИЕ X(1:100,1:100) ТЕКСТ(254) РАЗНОЙ_ДЛИНЫ;
```

Остальные характеристики идут в любом порядке. Чтобы в языке было поменьше всяких исключений из правил, разрешили писать границы не просто в скобках, а после ключевого слова DIMENSION или РАЗМЕРНОСТЬ, но тогда уже и не сразу после имени, например:

```
DECLARE
X CHAR(254) VARYING DIMENSION (1:100,1:100);
ОПИСАНИЕ
X ТЕКСТ(254) РАЗНОЙ_ДЛИНЫ РАЗМЕРНОСТЬ (1:100,1:100);
```

К счастью, перечислять все до единой характеристики для каждого объекта обычно не требуется. Часть характеристик присваивается по умолчанию и о них нужно вспоминать лишь в некоторых, нестандартных случаях.

Транслятору можно заказать создание файла, куда он запишет полную таблицу всех описаний. Вот там и будут указаны все характеристики каждого объекта полностью, с учетом умолчаний. Иногда программисту бывает очень полезно критически посмотреть на такую таблицу своей программы. Транслятор также может отметить все объекты, которые описаны в программе, но к которым так никто и не обратился.

В описаниях числовых типов PL/1 указывается прямо число разрядов мантиссы числа. Это кажется анахронизмом, да и ЭВМ с «нестандартным» числом разрядов уже не осталось. Но, на мой взгляд, описания типа «float» и «double» в языке Си выглядят гораздо туманнее, чем, например, описания в PL/1 FLOAT(24) ВЕЩЕСТВЕННОЕ(24) и FLOAT(53) ВЕЩЕСТВЕННОЕ(53), явно показывающие, на мантиссу какой длины может рассчитывать программист.

Разумеется, транслятор не создает объектов, занимающих нецелое число байт. Поэтому и FIXED(16) ТОЧНОЕ(16) и FIXED(31) ТОЧНОЕ(31) в памяти будут занимать по четыре байта. Однако, иногда все же есть смысл в описаниях с «кривым» числом байт. Например, для переменной, описанной как FIXED(16) ТОЧНОЕ(16), транслятор проверит, что хоть константы, большие, чем 65536, в нее не пытаются

засунуть.

Сейчас начинается массовое производство процессоров уже с 64-битной архитектурой. Если я когда-нибудь соберусь доработать транслятор для этой архитектуры, я просто расширю ТОЧНОЕ(31) до ТОЧНОЕ(63). А вот в языке Си будет «double long int» что ли?

33. Выражения

Если операторы присваивания составляют «тело» алгоритма, то выражение – это «тело» самого присваивания. Выражения тоже попали в программирование из математики и кажутся вполне естественными для записи алгоритмов.

В те времена, когда я понятия не имел, как устроены трансляторы, они казались мне фантастически сложной штукой именно из-за выражений. Шутка ли, пишешь любое сложное выражение, и транслятор его понимает!

Оказалось, что разбор выражений – это совсем небольшая часть транслятора, хотя в некотором смысле, самая главная. Это и есть перевод (трансляция).

Небольшим разбор выражений делают, во-первых, рекурсия, в во-вторых, тот факт, что транслятор ничего и не «понимает», а лишь тупо переводит выражение из того вида, в котором оно записано в программе, в другой вид, тоже придуманный математиками. Первым такой вид употребил польский математик Лукашевич, поэтому он стал называться «польской записью». Потом, правда, писать стали зеркально тому, что писал Лукашевич, и все это стало называться «обратной польской записью».

Что это за зверь? Оказывается, обычную, привычную нам, нам запись, например, A+B-C+D, можно записать как A B C D + - +. При этом однозначность останется, если мы будем действовать по правилу: когда встречается операнд (например, число), мы его просто запоминаем; когда встречается операция (например, сложение), мы ее выполняем, используя операнды, запомненные последними, при этом ответ запоминаем так же, как и просто встреченный операнд. К чему такие сложности и почему нельзя применить подобные правила к обычной записи? Все дело в том, что в обратной польской записи границы каждого действия в выражении получаются сами собой и выражение разбивается на ряд совершенно одинаковых действий с точки зрения порядка работы.

Такую запись алгоритма очень неудобно читать человеку. Но она легко читается третьей, отдельной частью транслятора (которая генерирует команды ЭВМ). Эта третья часть тоже «тупо» переводит каждую операцию из обратной польской записи в одну или несколько команд ЭВМ, совершенно не заботясь, как же из этих команд получатся требуемые исходные выражения. Получатся, уверяю Вас! При этом

выражения могут быть настолько сложными, насколько у транслятора хватит памяти и терпения их разбирать.

В одной книге перевод в обратную польскую запись сравнивался с процессом формирования железнодорожного состава. Исходный состав состоит из вагонов (операндов) и платформ (операций). Если диспетчеру встречается вагон, он его стразу пропускает на формирование. А когда встречается платформа, диспетчер сначала отгоняет ее в тупик, а в нужный момент выпускает из тупика тоже на формирование. В результате в новом составе платформы займут другие места. Не знаю, стало ли от этого понятнее. Лучше рассмотрим, как данный транслятор разбирает выражения PL/1.

Итак, любое выражение в PL/1 - это последовательность идущих друг за другом операций «ИЛИ».

Каждая операция «ИЛИ» - это последовательность из операций «И», между которыми стоит знак «!».

Каждая операция «И» - это последовательность из операций отношения, между которыми стоит знак «&».

Каждая операция отношения - это последовательность операции склейки, между которыми стоит знак отношений (типа больше-меньше).

Каждая операция склейки - это последовательность аддитивных операций, между которыми стоит знак склейки «!!».

Каждая аддитивная операция - это последовательность мультипликативных операций, между которыми стоит знак «плюс» или «минус».

Каждая мультипликативная операция - это последовательность унарных операций, между которыми стоит знак «умножить» или «разделить».

Каждая унарная операция - это знак «плюс» или «минус» или «отрицание», за которым идет или имя объекта, или опять выражение (которое начинается с круглой скобки).

После имени объекта или выражения в круглых скобках может идти знак возведения в степень. Если такой знак степени стоит, за ним опять идет выражение (это показатель степени).

Наконец, имя объекта, это или обычное имя или константа.

Уффффф! Все!

На самом деле, эта сказка про белого бычка или попа с собакой никак не кончится потому, что обычное имя само может иметь список индексов или список параметров в круглых скобках, а каждый индекс или параметр опять может быть выражением и см. в начало.

Тем не менее, это именно разбор произвольного выражения в PL/1, который именно так и записан в трансляторе и именно так транслятором и выполняется.

Могут вызвать недоумения все эти операции «И» и операции склейки. А если в выражении нет никаких операций «И», как их разбирать? В данном случае операцию «И» можно было назвать как угодно, хоть

операцией «Ы». Это всего лишь ступень разбора, на которой транслятор проверяет наличие конкретных значков. Если он этот значок нашел, он запоминает сей факт и затем, вовсе не в этом месте, а в конце всей ступени пишет соответствующую операцию. Если хотите, выпускает соответствующую платформу из тупика. Имя или константа просто записываются в тот момент, когда встретились (т.е. вагон прямо идет на формирование).

Приведенная формула выражения еще не самая сложная из всех языков программирования. Даже в Алголе были добавлены операции «исключающего ИЛИ» и таинственная операция «импликация», правда, не было операции «склейки» (как и в выражениях языка Си). Для транслятора не вызовет никакой сложности разбор и даже вдесятеро более сложных выражений, т.е. можно было бы придумать и еще какие-нибудь типы операций. Но практика показывает, что, например, даже «исключающее ИЛИ» используется редко, и уж совсем редко в сложных выражениях. Для таких редких операций можно использовать отдельные специальные функции, не захламляя обычный разбор выражений ступенями, которые почти никогда и не попадаются.

Выражения могут встретиться как условие внутри условного оператора или в заголовке оператора цикла или еще где-нибудь. Транслятор это совершенно не заботит, во всех таких местах он просто обращается к одной и той же рекурсивной процедуре разбора выражения.

Таким образом, ничего волшебного в трансляторе не оказалось. Это примитивная программа, выдающая длинную цепочку операций и операндов для обработки их в своей третьей части. А третья часть почти не имеет дела с целой программой. Она отдельные операции переводит в отдельные команды. И если нужно сделать транслятор для новой ЭВМ, нужно заменить только эту часть транслятора. Причем, скорее всего, даже не заменить, а лишь изменить конкретный вид генерируемых команд.

Данный транслятор может даже выдать обратную польскую запись в файл. Хотя она обычно никому не интересна, кроме разработчиков трансляторов, я один раз сумел ее использовать. Требовалось ускорить один специфический числовой расчет насколько это возможно. Я написал на языке PL/1 свою «третью часть», гораздо более простую, чем у самого транслятора и годящуюся только для оптимизации этого расчета. Моя программа всего лишь разбирала уже готовую обратную польскую запись и генерировала несколько простых команд. Именно тогда я убедился, что ничего сложного в процессе трансляции нет.

34. Передача параметров

Поговорив об описаниях и выражениях, вернемся к передаче параметров в процедурах. Потому, что параметры надо описывать и

потому, что параметры могут быть выражениями. А почему параметрам уделяется столько внимания? Дело в том, это как раз то место, где программист взаимодействует с другими. И часто это взаимодействие происходит как раз через параметры.

Если вы хотите использовать в своей программе чужую процедуру, прежде всего Вы должны ее описать в своей программе. В некотором роде это сильно отличается от Ваших, так сказать, «местных» описаний: здесь Вы не можете сами назначить имя и типы параметров процедуры. Их нужно взять готовыми из какого-нибудь документа или другой программы. Причем, даже если это Ваши собственные процедуры, просто расположенные в других файлах, все равно, для транслятора это чужая, неизвестная процедура. Ниоткуда не следует, что она расположена в соседнем файле и что транслятор ее вообще когда-либо будет транслировать.

Это порождает серьезную проблему. Если перепутать число, порядок или типы параметров таких чужих, внешних процедур, то транслятор не заметит ошибки в описании: он тоже не знает, как должно быть правильно. Поэтому у программиста появляется вторая прекрасная возможность нарушить работу программы с непредсказуемыми последствиями.

Именно поэтому описания процедур часто составляют сами разработчики (например, Windows) и предоставляют остальным программистам уже в виде готовых отдельных файлов. К сожалению, обычно такие стандартные описания составляются не на PL/1, и мне приходится самому переводить их в описания PL/1 (например, для Direct X), правда, это приходится делать не часто.

В данном трансляторе имеется возможность проверить согласованность описаний всех процедур, если их исходные тексты на PL/1 доступны. Помните, транслятору можно заказать вывод в отдельный файл полную таблицу объектов? Так вот, если сначала оттранслировать все процедуры с выводом таких таблиц, то затем можно запустить специальную программу (в составе транслятора), которая просто проверит одинаковость описаний процедур во всех найденных таблицах и покажет отличия, если таковые найдутся. К несчастью, в реальных проектах не все исходные тексты процедур доступны, да и написаны они не все на PL/1.

Следует отметить и еще одну особенность PL/1. В языке Си, например, имеется только четыре типа параметров: указатель, короткая строка бит, целое число и вещественное число. В языке PL/1 параметры могут быть любого, даже составного типа.

Внутри описания внешней процедуры имена параметров не ставятся, а ставятся только их типы. Поэтому описания внешних процедур в PL/1 могут выглядеть, например, так:

```
DECLARE
P1 ENTRY((1:100) FIXED(31),
         (1:100) CHAR(80) VARYING)
         RETURNS(FIXED(31));
ОПИСАНИЕ
P1 ДЛЯ_ВЫЗОВА((1:100) ТОЧНОЕ(31),
              (1:100) ТЕКСТ(80) РАЗНОЙ_ДЛИНЫ)
             ВОЗВРАЩАЕТ(ТОЧНОЕ(31));
```

В этом примере у процедуры P1 два параметра: массив из 100 чисел и массив из 100 текстов. Транслятору приходится обрабатывать в программе описания двух видов: обычные описания и описания списков параметров процедур. Во втором случае в описаниях нет имен и в них нельзя указывать никакого класса памяти.

Если Вы не используете внешние процедуры, а наоборот, пишете процедуру, которой будут пользоваться другие, никаких особенностей не возникает. Параметры перечисляются в заголовке процедуры, а затем в теле процедуры описываются как обычные объекты.

Для транслятора, правда, параметры внутри процедуры это не обычные переменные, а такие объекты, которые всегда используются с неявными указателями. Сами указатели берутся из списка, который передается процедуре в момент ее вызова.

Организовать взаимодействие с другими процедурами можно, разумеется, и без всяких параметров. Если описать общие переменные с одними и теми же именами во всех процедурах, это будет тоже способ «передачи параметров», причем самый быстрый. И тогда для транслятора это действительно будут самые обычные переменные.

Но все-таки часто удобнее организовать взаимодействие именно через параметры. Но тогда возникает вопрос, как именно передавать значения? Рассмотрим такую ситуацию. Имеется процедура с параметром-числом. При вызове процедуры на месте этого числа стоит выражение 3+2. В заголовке процедуры этот параметр назван X, а теле этой процедуры имеется присваивание X=0; Ну, и куда теперь присвоится этот ноль? Между константами два и три?

Нет, ничего страшного, конечно, не произойдет. Транслятор выделит память для результата выражения, и ноль будет записан туда, где раньше стояло пять. Программа, где стоял вызов процедуры, это не почувствует и не сломается. Почему это происходит? Потому, что параметром передался «безопасный» адрес памяти, специально выделенный для параметра.

Некоторые теоретики утверждают, что только так и надо передавать параметры. Т.е. даже если бы передавалось не выражение 3+2, а некая переменная Y, надо специально сделать ее копию и передать адрес этой копии. Такая передача называется передачей параметров по значению.

Но во всех случаях использовать такую передачу невозможно. Ведь

параметры могут быть не только «входные», но и «выходные», т.е. процедура должна заполнить их ответами. Часто одного возвращаемого значения процедуры-функции для этого не хватает. Можно, конечно, «выходные» параметры оформлять как общие переменные или городить новые механизмы (как, например, в некоторых языках объекты-«делегаты»). Но, на мой взгляд, это не всегда естественно.

Посмотрим, как это реализовано в PL/1. Здесь «лишние» копии не создаются. Т.е. если при вызове параметр представляет выражение – все происходит так описано выше. Но если передается не выражение и тип параметра в операторе вызова и в описании процедуры совпадают, то безо всякой копии передается адрес «настоящей» переменной. Такая передача параметров называется передачей по имени. Правда в случае, если вызываемая процедура изменит этот параметр, а вызывающая программа этого не ожидала, в программе может произойти ошибка.

В PL/1 программист из любой переменной может легко сделать выражение, написав саму переменную в круглых скобках, и, таким образом, явно указав, что нужны копии и нельзя портить параметры. Интересно, что в PL/1 можно защититься копиями не только в операторах вызова, но и так сказать «изнутри» процедуры, тоже записав ее параметры в заголовке в круглых скобках, например:

```
P1:PROCEDURE((X)) RETURNS(FIXED(31));
DECLARE X FIXED(31);
…
P1:ПРОЦЕДУРА((X)) ВОЗВРАЩАЕТ(ТОЧНОЕ(31));
ОПИСАНИЕ X ТОЧНОЕ(31);
```

Теперь как бы ни вызвали процедуру, и `P1(Y);` и `P1((Y));` все равно, уже в самой процедуре будет сделана копия параметра и сам параметр после вызова никогда не изменится. Во втором вызове будет сделано даже две копии переменной `Y`: одна «снаружи» (в операторе вызова), другая «внутри» (в самой процедуре).

Приходится бороться и с обратной ситуацией, когда параметр должен бы быть выходным, но из-за создания копии он к удивлению программиста после оператора вызова не меняется. Это бывает, когда программист в описании процедуры и в операторе вызова использовал объекты разного типа. Тогда транслятор преобразует параметр к тому типу, который указан в описании процедуры и из-за этого опять получается копия, как и для выражения.

В данном трансляторе в операторах вызова перед такими параметрами можно ставить знак «*», явно указывая, что этот параметр должен быть «выходной». Если транслятору придется делать копию параметра со звездочкой (т.е. тот самый случай несовпадения типов), он обязательно сообщит об этом. Понятно, что бессмысленно одновременно ставить звездочку и заключать параметр в круглые скобки. Первое – это указание, что не должно быть копии, а второе – как раз указание сделать копию.

35. Параллельная работа

Пока компьютеры, с которыми я работал, имели только один «центральный» процессор, я не очень интересовался параллельной работой, поскольку все равно реального выигрыша во времени она не давала. Да и в исходном трансляторе параллельности не было. Все изменилось после начала массового выпуска многоядерных процессоров. Стало просто глупо не использовать их возможности, тем более что операционная система Windows позволяет организовать «потоки», не вдаваясь в аппаратные тонкости. Пришлось вернуться к стандарту языка, который предусматривает параллельный запуск процедур с помощью оператора:

```
CALL P1(X,Y) TASK;
ВЫЗОВ P1(X,Y) ПРОЦЕСС;
```

После такого вызова процедура P1 начинает работать одновременно с остальной программой (которая ее вызвала) как отдельный «поток» Windows. И здесь пришлось чуть «доработать» стандарт, добавив возможность указывать в операторе кроме TASK ПРОЦЕСС еще и слово STOP СТОП, что позволяет запустившей процедуру программе принудительно завершить ее. Кроме этого, в языке был предусмотрен еще и оператор ожидания WAIT ЖДАТЬ, при достижении которого программа «засыпает», пока не произойдет указанное в данном операторе событие. Я чуть упростил этот оператор, сделав событие в операторе ожидания обычной переменной-строкой в один бит. С данными операторами и с учетом использования операторов обмена для установки/снятия «семафоров» параллельная организация работы становится возможной.

36.Исключительные ситуации

Я каждый раз отмечаю, что та или иная штука в PL/1 реализована как предложение еще создателей Алгола. Но исключительные ситуации, это тот случай, когда все «с нуля» появилось, начиная именно с PL/1. Сначала целью нового понятия была борьба с ошибками при выполнении программы. Но оказалось, что механизм ситуаций (событий) и реакций на них универсален и очень удобен для большого класса задач.

Можно отнести его к еще одному «киту» (ну хотя бы китенку) базовых понятий программирования. Обработка «сообщений» как основа реализации Windows и полная замена переходов исключениями в уважаемом мною языке Эль-76 являются прекрасными примерами универсальности данного механизма.

А сначала все было просто. Например, в Вашей программе на Алголе или Паскале где-то что-то разделилось на ноль. Что будет с программой? Она прекратится и программисту придется искать это место по уже

напечатанной информации или по какой-нибудь «посмертной» выдаче. Как этого избежать? Ну, например, поставить проверки на ноль перед каждым делением. Вряд ли получится хорошо.

Во-первых, программа раздуется, а алгоритм может затеряться среди множества проверок.

Во-вторых, на ноль может разделиться не в Вашей программе, а в какой-нибудь внешней процедуре из-за неверных параметров при ее вызове. Проверить правильность параметров - это уже более сложная задача, особенно, если Вы не представляете, как работает внешняя процедура.

А можно ли в общем случае, несмотря на случившееся, продолжить работу? Конечно можно, надо только описать в программе, что делать в таких случаях.

В PL/1 действия, которые надо делать, если случится ситуация вроде деления на ноль, описываются в отдельном блоке. Перед блоком ставится ключевое слово ON или КОГДА и название предвиденной ситуации (может быть даже один блок действий для нескольких разных ситуаций). Например:

```
ON ZERODIVIDE, FIXEDOVERFLOW
      BEGIN; X=0; GOTO ПОВТОРНЫЙ_РАСЧЕТ; END;
КОГДА ДЕЛЕНИЕ_НА_0, ЧИСЛО_НЕПРЕДСТАВИМО
      БЛОК; X=0; ИДТИ ПОВТОРНЫЙ_РАСЧЕТ; КОНЕЦ;
```

Здесь в случае деления на ноль или получения результата, который не помещается в место, отведенного для точного числа, будет присвоен ноль переменной X, после чего управление передастся на метку ПОВТОРНЫЙ_РАСЧЕТ.

Да, исключительная ситуация – это именно тот случай, где естественно применение оператора перехода GOTO или ИДТИ, так не любимого теоретиками. Ведь блок, описывающий реакцию на ситуацию, очень похож на процедуру, которую вызовет неизвестно кто и неизвестно когда. Возврат из такой процедуры непонятен. Куда возвращаться-то? Опять в то место, где случилась неприятность? Скорее, нужно чуть «отступить назад», поправить данные и повторить по новой. Транслятор сам такое место найти не сможет. Логично явно указать, куда надо бежать в случае ошибки.

Кстати, вот и прекрасное место, где можно поговорить о «проблеме GOTO». Что же это за «религиозная война» без такого обсуждения? Сама история «полемики о GOTO» удивительна. Все начиналось в весьма разумного заявления известного теоретика Дейкстры, что множество переходов ясности алгоритму не добавляют. Правда, так и осталось неизвестным, программы каких «умельцев» он пытался разобрать. Но все схватились за очередную «волшебную палочку» - исключение переходов, что якобы автоматически сделает программы и понятнее и надежнее. Вместо того чтобы определить множество случаев желательного использования переходов (вроде выхода из исключений),

любое использование GOTO стало считаться дурным тоном и именно такое отношение отложилось в памяти у многих программистов. Я сам с интересом наблюдал, как одна моя коллега (с интеллектом явно не как у «блондинки») в течение часа безуспешно пыталась переделать кусочек программы «чтобы не было GOTO». Мои убеждения, что понятный фрагмент она тем самым превратила в непонятный и вдобавок внесла логическую ошибку, совершенно не подействовали: «нас учили, что GOTO – это плохо». Я сравниваю оператор перехода в языке с остро заточенным ножом: конечно, резать нужно осторожно и аккуратно, тогда и не порежешься. Но призывы вообще убрать все «колюще-режущие предметы» смешны: неумеха умудрится порезаться даже ниткой. В PL/1 операторы перехода очень «острые», например, можно уйти командой перехода из процедуры, минуя RETURN ВОЗВРАТ. Для этой цели транслятор определяет все метки в блоке, на которые можно попасть из других блоков и автоматически ставит после них служебную процедуру восстановления памяти-«стека».

В общем случае, конечно, могут быть события, которые ничего в ходе программы не ломают. В таких случаях, в конце блока, описывающего соответствующую реакцию, можно никуда и не переходить. Тогда, выполнив все действия, описанные в блоке, программа продолжит выполнение с места, где случилось это событие.

Такое поведение программы быстро нашло отражение и в аппаратуре ЭВМ, где появился механизм «прерываний», полностью совпадающий с описанным выше общим случаем. (Возможно, «аппаратчики» обидятся и скажут, что прерывания первыми появились как раз в их процессорах, а создатели языков просто скопировали их.)

Ну, некоторые создатели языков (например, Н. Вирт) сначала воротили нос от исключений. Но постепенно необходимость заставила включать их в языки. Помню, в начале 90-х на семинаре по программированию докладчик сообщил о введении исключений в язык Си, пробурчав, что «еще немного и Си станет совсем похож на PL/1».

Возможно, что если бы Windows была действительно написана на PL/1, обработка «сообщений», которая является основой взаимодействия с Windows, была бы реализована проще и естественнее через встроенный механизм этого языка.

Использование исключительных ситуаций сопряжено и с некоторыми сложностями. Например, блок реакции на событие может быть описан в таком месте программы, что не «увидит» переменные, из-за которых ситуация и произошла. Реакция на одну и ту же ситуацию может быть описана в нескольких процедурах, вызывающих друг друга.

В PL/1 обработка идет самым простым образом: запоминаются все реакции (операторы ON или КОГДА), через которые прошло выполнение программы. Если произошла событие, реакция не него ищется среди всех запомненных реакций, начиная с последней

запомненной. При выходе из процедуры, все реакции, описанные в ней, теряются.

В PL/1 были также придуманы оператор REVERT или ОТМЕНИТЬ для явной отмены заданной и ранее описанной реакции и оператор SIGNAL или СИГНАЛ для явной генерации заданного события. Можно считать, что при выходе из процедуры ко всем описанным реакциям на ситуации неявно применяется соответствующий оператор REVERT или ОТМЕНИТЬ.

Еще до эпохи Windows мы использовали механизм ситуаций PL/1 для создания интерактивного интерфейса с пользователем. У нас тоже были «меню», «сообщения» и даже «заказы» (меню, где выбран не один, а сразу несколько пунктов). События возникали, когда нажимались клавиши клавиатуры и кнопки мыши. С имеющимся в PL/1 механизмом получалось удобно. Мы использовали тот факт, что для наиболее общей причины исключительных ситуаций (вида ERROR или ОШИБКА) транслятор использовал не все возможные 255 типов уточнений и сами генерировали и обрабатывали такие фиктивные «ошибки». Я подправил транслятор, чтобы ключевое слово ERROR или ОШИБКА в описании вида причины стало необязательным. Например, вместо оператора

```
ON ERROR(125) BEGIN;…
КОГДА ОШИБКА(125) БЛОК;…
```

стало можно писать

```
ON (125) BEGIN;…
КОГДА (125) БЛОК;…
```

подчеркивая, тем самым, что это не какая-то ошибка в программе, а вполне законное событие, которое мы же сами где-то и генерируем оператором SIGNAL(125) или СИГНАЛ(125);. Через несколько лет мы обнаружили в описании Windows похожий механизм из событий и их обработчиков.

37. Операторы обмена

Как вижу из распечатки транслятора, операторы обмена, это последние операторы, которые разбирает транслятор с PL/1. Больше уже ничего нет! Если, конечно, не считать всякой мелочи вроде оператора STOP; СТОП; или анекдотичного «пустого» оператора. Зато разбор операторов обмена это приличный кусок транслятора с PL/1, да еще я его раздул дописыванием всяких отладочных форм обмена (почему-то сразу не сделанных командой Гарри Килдэлла).

Кажется, и эта глава получится больше остальных. Ну, ничего, ведь на самом деле операторы обмена это, пожалуй, все-таки еще один «кит» программирования. Без других «китов» программа невозможна, а без этого еще и бесполезна.

С философской точки зрения, операторы обмена – это те же операторы присваивания, но взаимодействующие уже с внешним миром, а не внутри программы.

Операторы обмена достались PL/1 по наследству от Кобола. Этот язык прижился только в Америке и сейчас на нем не столько пишут, сколько поддерживают уже написанное. А вот идеи обмена этого языка как взаимодействие программы с файлами оказались плодотворными и получили дальнейшее развитие, даже выйдя за пределы программирования. Например, просто работая на персональном компьютере, мы сами тоже взаимодействуем с файлами.

Файл в программе имеет немного двойственный характер. С одной стороны это объект программы (с типом «файл»), которому мы можем дать любое удобное нам имя. С другой стороны, файл – это внешний объект, который может существовать и до, и после программы и может иметь, например, собственное имя, не зависящее от программы.

Эта двойственность в PL/1 проявилась в том, что, по сути, у файла в программе не одно, а целых два описания. Одно «обычное» описание, где для переменной просто указывается «абстрактный тип» FILE или ФАЙЛ. Другое «настоящее» описание перечисляет все действительные характеристики файла, в том числе и его «настоящее» имя. Следствием разделения описаний является возможность использовать одну и ту же переменную типа «файл» для работы с разными файлами на протяжении разных этапов выполнения программы.

Второе описание перенесено внутрь оператора «открытия» файла. После ключевого слова OPEN или ОТКРЫТЬ может следовать множество характеристик (конечно, как и все в PL/1 в любом порядке). Если очередная характеристика имеет еще и параметры, они ставятся за ключевым словом этой характеристики в круглых скобках.

Создатели PL/1 разделили все характеристики на две кучки и одну кучку засунули как параметры внутрь одной из характеристик. Идея здесь была вот в чем: есть характеристики, которые являются вполне себе понятиями языка (например, файл для ввода и файл для вывода), а есть характеристики, которые не входят в понятия языка (например, размер «буфера» файла), а являются заботой какой-нибудь Windows. Вот такие характеристики и нужно записать внутри одной характеристики ДЛЯ_ОС (английский оригинал называется ENVIRONMENT). Тогда, если нужно перенести программу в другую операционную систему, то надо найти в тексте программы все ENVIRONMENT ДЛЯ_ОС и при необходимости изменить их содержимое. Весь остальной текст программы не зависит от ЭВМ и операционной системы (на самом деле, есть еще отличия, мы о них потом поговорим). Идея правильная, однако, с этой точки зрения внешнее имя файла (характеристика TITLE или ПО_ИМЕНИ) тоже надо было бы писать внутри, а не отдельно, ну да ладно.

Как бы то ни было, обычно многословный оператор «открытия» файла в PL/1 присваивает объекту обмена в программе разные атрибуты и параметры внешнего объекта.

При этом еще одна характеристика выделяется среди остальных. Она

начинается с ключевого слова FILE или ФАЙЛ и именно она связывает в программе объект обмена с переменной «абстрактного типа» FILE или ФАЙЛ, которую мы описываем отдельно. Такая связь сохраняется до оператора «закрытия» файла, после чего, переменная освобождается для новых файлов.

Естественно, «открывать» и «закрывать» файлы в программе можно неограниченное число раз, а вот число одновременно «открытых» файлов в принципе ограничено, ведь для каждого из них нужна память, место в таблицах Windows и т.п.

В самих пяти операторах обмена READ, WRITE, GET, PUT и REWRITE ВВОД, ВЫВОД, ЧИТАТЬ, ПИСАТЬ и ПЕРЕЗАПИСАТЬ указывается, конечно, уже не внешнее имя файла, а имя переменной типа FILE или ФАЙЛ, которое связанно с нужным объектом обмена после выполнения оператора «открытия».

Я и здесь не удержался и чуть подправил транслятор, сделав ключевое слово FILE или ФАЙЛ в операторах обмена необязательным. По-моему, в данном случае и одних круглых скобок вполне достаточно для ясности текста. Сравните две формы:

```
OPEN FILE(ИСХОДНЫЕ_ДАННЫЕ) INPUT…
READ FILE(ИСХОДНЫЕ_ДАННЫЕ)…
ОТКРЫТЬ ФАЙЛ(ИСХОДНЫЕ_ДАННЫЕ) ДЛЯ_ВВОДА…
ЧИТАТЬ ИЗ_ФАЙЛА(ИСХОДНЫЕ_ДАННЫЕ)…
```

и

```
OPEN (ИСХОДНЫЕ_ДАННЫЕ) INPUT…
READ (ИСХОДНЫЕ_ДАННЫЕ)…
ОТКРЫТЬ (ИСХОДНЫЕ_ДАННЫЕ) ДЛЯ_ВВОДА…
ЧИТАТЬ (ИСХОДНЫЕ_ДАННЫЕ)…
```

Поскольку все равно открывать и читать в PL/1 можно только файлы, стало быть, можно и описывать все это чуть короче.

Вообще операторы обмена и оператор «открытия» – это как раз то место, где транслятору мешается разрешение PL/1 перечислять параметры в любом порядке. Ведь параметры типа имени файла или размера строки могут включать в себя сложные выражения. Вот и приходится транслятору затем сортировать разобранные выражения («пузырьковой» сортировкой), чтобы параметры поступали в служебные процедуры языка всегда в одинаковом порядке.

Рассмотрим характеристики файлов в PL/1.

Файл может быть INPUT ДЛЯ_ВВОДА или OUTPUT ДЛЯ_ВЫВОДА или UPDATE ДЛЯ_ИЗМЕНЕНИЙ.

Файл может быть STREAM ТЕКСТОВЫЙ или RECORD НЕ_ТЕКСТОВЫЙ.

Файл может быть SEQUENTIAL ПООЧЕРЕДНЫЙ или DIRECT ПРЯМОЙ.

Наконец, файл может быть KEYED ИНДЕКСНЫЙ.

Придумали еще файл PRINT ПЕЧАТНЫЙ или ДЛЯ_ПЕЧАТИ, что просто заменяет объединение параметров OUTPUT SEQUENTIAL STREAM ДЛЯ_ВЫВОДА ПООЧЕРЕДНЫЙ ТЕКСТОВЫЙ.

Как видите, характеристик много и все они описывают разные свойства файлов. Некоторые не указанные явно характеристики транслятор может «вывести» из имеющихся, но все равно часто в операторе открытия приходится писать много всего.

Наверное, первые три характеристики и так понятны. Можно только отметить, что если файла для вывода еще не было, то оператор его открытия превращается в оператор его создания.

Остальные характеристики немного обсудим.

STREAM или ТЕКСТОВЫЙ файл предназначен для чтения и программой и человеком, поэтому в нем значения должны быть не в виде нулей и единиц, а в виде вполне читаемых текстовых строк. Для удобства там могут быть еще кавычки, основание системы счисления и т.п. Например, HTML-формат, это типично текстовый файл.

RECORD или НЕ_ТЕКСТОВЫЙ файл предназначен для чтения-записи только программой. Здесь, например, числа можно писать именно так, как они представлены в памяти. Размер такого файла может быть и короче предыдущего, а самое главное - при обмене с таким файлом не нужно преобразовывать все данные в текст и наоборот. Сейчас модно называть такие файлы бинарными.

SEQUENTIAL или ПООЧЕРЕДНЫЙ файл предполагает чтение или запись только в одном направлении: от начала к концу. Вернуться назад в таком файле нельзя. Каждый следующий обмен начинается с того места, где кончился предыдущий. Текстовые файлы могут быть только такими.

DIRECT или ПРЯМОЙ файл допускает возможность вернуться к любому, в том числе уже прочитанному или записанному месту.

KEYED или ИНЕКСНЫЙ файл очень похож на массив. Предполагается, что он состоит из объектов одинаковой длины и к ним можно обращаться по номеру индекса и в любом порядке. Разумеется, такой файл может быть только DIRECT ПРЯМОЙ.

Следует подчеркнуть, что здесь ТЕКСТОВЫЙ и НЕ_ТЕКСТОВЫЙ это всего лишь характеристики объекта, а не смысловое содержание самого файла. Ничего не мешает, например, открыть HTML-файл как ПРЯМОЙ ИНДЕКСНЫЙ с размером элемента в байт и затем менять в нем отдельные буквы операторами прямого ввода-вывода.

В завершении темы открытия и закрытия файлов упомяну о логике их работы. Это уже не имеет отношения к работе транслятора. В PL/1 принято, что если пытаться закрыть уже «закрытый» файл, то ничего не происходит. Это разумно. Но авторы языка предложили также ничего не делать и при попытке открыть уже «открытый» файл. А вот это хуже, так как если Вы забыли закрыть файл и открываете новый в ту же переменную, то работа продолжится со старым файлом и это может быть

даже сразу не замечено. Я подправил теперь уже административную часть исполняемой программы (системную библиотеку) так, чтобы в подобных случаях выдавался сигнал ошибки. Жить стало намного легче.

Отмечу также, что в программе файл можно вообще не открывать, тогда системная библиотека откроет его по своему разумению при первом обращении к файлу. Есть также два стандартных файла SYSIN СТД_ВВОД и SYSOUT СТД_ВЫВОД, которые открываются автоматически в начале каждой программы и по умолчанию являются выводом на дисплей и вводом с клавиатуры. При необходимости их можно открыть и с другими параметрами. Эти файлы хороши тем, что именно их (и только их) можно вообще не указывать в операторах обмена, тогда обмен считается «стандартным». В смехотворной программе «Здравствуй мир!» никакого файла не указано, поэтому фраза выдается в стандартную «текстовую консоль» (стандартное окно) Windows, а с точки зрения программы - в файл SYSOUT СТД_ВЫВОД.

Я как-то отвлекся от самих операторов обмена (READ, WRITE, GET, PUT и REWRITE ВВОД, ВЫВОД, ЧИТАТЬ, ПИСАТЬ и ПЕРЕЗАПИСАТЬ). Во-первых, почему их целых пять? По идее должно быть только два: ввод и вывод. Ну, операторов получается две пары потому, что текстовый обмен и обмен произвольными данными идет отдельными операторами, и они имеют попарно одинаковый, но между парами очень разный вид.

А «пятое колесо» в виде оператора REWRITE ПЕРЕЗАПИСАТЬ появилось вот откуда: когда в файле нужно только что-то изменить, то получается многовато действий. Сначала старый кусок файла переписывается в «буфер» файла в памяти. Затем из «буфера» данные переносятся в объекты программы. Затем эти объекты программа меняет и начинается обратный процесс переноса в «буфер», а затем наружу.

В PL/1 предложили подобные случаи ускорить: программист может получить не данные в переменных, а прямо указатель на считанный «буфер». Теперь, используя этот указатель, он может прямо в буфере же что-то заменить и дать команду REWRITE ПЕРЕЗАПИСАТЬ в смысле «положи, где взял». В самом этом операторе указывается только файл и все.

Теперь Windows с ее файлами «отображаемыми на память» затмевает скромные возможности оператора REWRITE ПЕРЕЗАПИСАТЬ, я им и не пользуюсь. К тому же, честно говоря, лень было вводить в оператор OPEN ОТКРЫТЬ новое ключевое слово для признака таких файлов (какое-нибудь «MAPPING») и я отмечаю «отображаемые на память» файлы просто указанием размера «буфера», равным -1. Повторяю, для транслятора это все равно, он ничего не проверяет, все делает системная библиотека.

Кстати о проверках: во время исполнения программы перед каждым оператором обмена вызывается служебная процедура. Она проверяет, а

может ли быть применен такой оператор для имеющихся атрибутов «открытого» файла? Если нет – выдается сигнал ошибки. Получается очень мощная защита правильности всего обмена. Мне пришлось лишь чуть-чуть дополнить ее проверкой на рекурсивный текстовый вывод. Бывали случаи, когда в операторах печати указывали не только переменные, но и прямо вызов процедур-функций. Если внутри такой процедуры-функции забывали отладочную печать, то получался вывод текста внутри другого вывода текста, и не рекурсивные служебные процедуры вывода ломались.

Первая пара операторов READ ВВОД и WRITE ВЫВОД, которую мы рассмотрим, проводит обмен любыми данными. Вообще пара операторов обмена в принципе зеркальна. То, что записал один, может прочитать другой, если он записан в том же виде. Даже «пятый» оператор REWRITE ПЕРЕЗАПИСАТЬ в этом смысле зеркален сам себе. Так вот, эти операторы короткие, в них нужно указать лишь файл и переменную. Переменная может быть и сложной структурой и массивом, т.е. некоторым непрерывным участком памяти. Например,

```
WRITE FILE(Ф1) FROM(X); или WRITE FROM(X) FILE(Ф1);
ВЫВОД В_ФАЙЛ(Ф1) ИЗ(X); или ВЫВОД ИЗ(X) В_ФАЙЛ(Ф1);
```

Данные операторы и пишутся и понимаются просто. Но есть одна сложность. Длину информации для обмена транслятор определяет по длине переменной. А иногда длина обмена, например, читаемых данных, неизвестна и зависит от самих этих данных. Приходится в программе организовывать цикл и читать, например, по байту в массив элементов размером в байт. Это и громоздко и лишние действия. Здесь я подправил транслятор, чтобы можно стало иногда явно задавать длину обмена после имени переменной, например:

```
READ FILE(Ф1) INTO(X,100);
ВВОД ИЗ_ФАЙЛА(Ф1) В_ПЕРЕМ(X,100);
```

В данном примере прочитается 100 байт и запишется в памяти по адресу переменной X, независимо от длины самой этой переменной X. Конечно, тогда нужно самому следить за тем, чтобы X была достаточно «длинная» и обмен не испортил память, выйдя за ее пределы. Транслятор, найдя явную длину, игнорирует длину самой переменной, и размер очередной порции обмена можно задавать прямо по ходу работы.

Теперь перейдем к гораздо более обширной части PL/1: к «текстовому» вводу/выводу, т.е. к виду, предназначенному в основном для чтения человеком, а не программой. Разбор операторов такого вывода занимает весьма приличный кусок транслятора.

Часто считают, что этот вывод стал почти не нужен: информация часто выдается в графическом виде или виде сигналов, а текстовый вывод – это, мол, эхо тех времен, когда все задачи были чисто вычислительными и выдавали ответ на бумагу. Замечу, что

вычислительных задач не стало меньше, просто их доля в общем числе программ уже не 100%. Но даже для «графических» программ текстовый вывод может быть очень даже нужен. Например, отладочный вывод самого программиста.

Я сам работаю с довольно сложной программой, цель которой – выдать картинку. Но я обязательно веду и текстовые файлы-протоколы, да и просто выдаю печать в отдельное текстовое окно Windows. Это здорово помогает. Поэтому я считаю любое «лишнее» удобство вывода в языке очень важным.

Помню, на каком-то компьютерном форуме программисты перебрасывались пожеланиями тех или иных возможностей в языке. И вдруг один заявил: «а я хочу PUT DATA». Ничего такого особенного этот программист не хотел. Он хотел мелочь, позволяющую быстрее и удобнее выдавать отладочную печать. Оператор PUT DATA или в моем переводе ПЕЧАТАТЬ С_ИМЕНАМИ – это оператор вывода, который кроме печати самих значений еще и автоматически печатает имена переменных, имеющих эти значения. Например, если оператор

```
PUT LIST(X,Y);
ПЕЧАТАТЬ В_ВИДЕ(X,Y); печатает 1280 1024
```

то оператор

```
PUT DATA(X,Y);
ПЕЧАТАТЬ С_ИМЕНАМИ(X,Y); напечатает X= 1280 Y= 1024
```

Почему в языках Ява и Си нет таких возможностей? Мне непонятно. Большую часть времени программист тратит на отладку. Здесь любая помощь со стороны транслятора дает эффект. Транслятору такую форму несложно сделать, он дописывает строки-константы, беря имена из своей таблицы. Правда, в языке Си вывод - это просто процедура, а не встроенная конструкция. Но таблицы имен все равно доступны, уж можно было как-нибудь изловчиться.

Кстати, один мой коллега использует оператор PUT DATA ПЕЧАТАТЬ С_ИМЕНАМИ вовсе не для отладки, а для запоминания настроек и текущих результатов и зеркальный ему оператор GET DATA ЧИТАТЬ С_ИМЕНАМИ - для возобновления расчетов. Конечно, обычно он и не читает получившиеся файлы. Но в случае ошибок начинает читать, и имена помогают ему легко разобраться в информации.

Операторы PUT ПЕЧАТАТЬ (он же ПИСАТЬ) и GET ЧИТАТЬ имеют два типа LIST и EDIT, я их назвал В_ВИДЕ и В_ФОРМЕ. После каждого из этих ключевых слов идет список обмена в круглых скобках.

Разница между ними в том, что типу LIST или В_ВИДЕ кроме списка объектов ввода-вывода уже больше ничего не нужно, внешний вид каждого значения он определяет сам, исходя из типа значения.

Для типа EDIT или В_ФОРМЕ дополнительно требуется список «форматов», явно указывающих, как будет выглядеть то или иное

значение. Например,

```
PUT LIST(X);
ПЕЧАТАТЬ В_ВИДЕ(X);
PUT EDIT(X)(Ч(8,2));
ПЕЧАТАТЬ В_ФОРМЕ(X)(Ч(8,2));
```

Если в первом случае напечатается что-нибудь вроде 1.0013456E+03, то во втором 1001.35

Обычное дело, что первый тип используется для «черновой», отладочной печати и писать его быстрее. Зато второй, более громоздкий тип, используется для «парадной», окончательной выдачи.

Рассмотрим немного сами списки обмена. Понятно, что в «текстовом» вводе-выводе не может быть объектов типа указателей, не имеющих «текстового» формата. Зато могут быть агрегаты данных или процедуры-функции, возвращающие значение подходящего типа. Я еще разрешил прямо в списке выводимого указывать оператор STOP СТОП. Зачем это надо? Видите ли, мне кажется что, например, оператор

```
IF X<0 THEN PUT LIST('ОШИБКА',STOP);
ЕСЛИ X<0 ТОГДА ПЕЧАТАТЬ В_ВИДЕ('ОШИБКА',СТОП);
```

лучше читается в тексте, чем

```
IF X<0 THEN {; PUT LIST('ОШИБКА'); STOP; };
ЕСЛИ X<0 ТОГДА {; ПЕЧАТАТЬ В_ВИДЕ('ОШИБКА'); СТОП; };
```

потому, что лишние скобки не мешают.

Создатели PL/1 разрешили в списке объектов обмена даже оператор цикла! Правда, здесь он может быть в более простом, урезанном виде, а полный оператор цикла проще организовать снаружи операторов ввода-вывода. Тем не менее, разрешены, например, такие операторы:

```
PUT LIST((X(I),Y(I) DO I=1 TO 20));
ПЕЧАТАТЬ В_ВИДЕ((X(I),Y(I) ЦИКЛ I=1 ДО 20));
```

В отличие от обычного оператора цикла, здесь границами цикла являются круглые скобки, а заголовок цикла переехал в конец. Вдобавок, тело цикла превратилось в список ввода-вывода. Как всегда, из-за рекурсивности транслятору несложно разобрать и более вычурные вложенные циклы, беда только, что программисту тяжеловато разбираться в большом количестве скобок.

В основном, я не пользуюсь такими циклами, но зато я использовал их прямо в трансляторе для того, чтобы разрешить вывод целых массивов и структур по одному имени. В исходном варианте транслятора этого не было сделано. Оказалось, достаточно небольшой переделки: встретив объект обмена целиком, а не «поштучно», транслятор на лету начинает создавать текст цикла вывода, подобный приведенному примеру. Дальше этот текст разбирается обычным способом и получается ввод-вывод для всех элементов массива или структуры.

В целом, в операторах текстового ввода-вывода PL/1 больше внимания уделено удобству отладочного вывода, в расчете на то, что

именно такой вывод часто и использует программист.

Еще одной особенностью текстового ввода-вывода является возможность обмена не с файлами, а просто с переменной типа «строка». Например, я хочу вывести рапорт о ходе работы в строку «заголовка» окна Windows. Ну не буду же я сначала выводить в файл, а затем опять читать из него. Это можно легко сделать через текстовую переменную:

```
PUT STRING(C) EDIT('СДЕЛАНО', X,'%')(A,F(6,2));
ПЕЧАТАТЬ В_СТРОКУ(С) В_ФОРМЕ('СДЕЛАНО', X,'%')(Т,Ч(6,2));
SETCONSOLETITLEA(C);
```

Здесь вместо файла указывается ключевое слово STRING В_СТРОКУ (для оператора ЧИТАТЬ соответственно ИЗ_СТРОКИ). Вся сформированная печать попадает просто в переменную C, которую как угодно можно использовать далее. Это очень удобный способ обработки строк.

Ну, и напоследок о самих форматах, т.е. явно указанных правилах представления данных. Здесь возможности очень большие, придуманные еще для составления таблиц в Коболе. Например, можно выравнивать столбцы таблицы знаками табуляции или просто задавать конкретную позицию столбца. Можно, используя так называемые «шаблоны», вывести числа с дробной частью, отделенной запятой, а не точкой и т.п. Для экономических задач можно даже автоматически выводить значки дебета-крЕдита (или все-таки кредИта?).

Список форматов записывается в отдельных круглых скобках и при обмене читается до исчерпания, после чего начинает читаться опять с начала. Такое правило позволяет иногда не писать отдельный формат для каждого значения, например, вывод всех чисел по одинаковому формату:

```
PUT EDIT(A,B,C,D,E,F)(F(8,2));
ПЕЧАТАТЬ В_ФОРМЕ(A,B,C,D,E,F)(Ч(8,2));
```

А вот пример печати трех чисел по одному формату, а трех других – по другому:

```
PUT EDIT(A,B,C,D,E,F)(3 F(8,2), 3 F(10,5));
ПЕЧАТАТЬ В_ФОРМЕ(A,B,C,D,E,F)(3 Ч(8,2), 3 Ч(10,5));
```

Используя также вложенные круглые скобки и коэффициенты повторения, можно создать очень сложные виды форматов, что иногда требуется при печати сложных массивов-структур.

38. Встроенные функции

Я же утверждал, что в трансляторе больше ничего не осталось? Ну что Вы! PL/1 так же неисчерпаем, как и атом. Действительно, практически все, что разбирает транслятор в части описаний, операторов и структуры, мы уже рассмотрели. Но осталось еще много такого, что также полезно рассмотреть. Например, встроенные в язык стандартные функции. Иногда их даже называют элементарными функциями.

С точки зрения языка встроенная процедура, это процедура (обычно процедура-функция) с уже известным стандартным алгоритмом и которую в программе не надо описывать.

С точки зрения блочной структуры все получается довольно ловко. Транслятор просто переписывает готовую таблицу стандартных функций в самое начало своей полной таблицы описаний. И все стандартные функции оказываются как бы в самом внешнем блоке, в котором затем умещается вся разбираемая программа.

Таким образом, внутри программы все стандартные имена «видны» и доступны. А если кто-нибудь написал свою процедуру с тем же именем (и, значит, этот двоечник не знает, что такая уже есть), то новая процедура «затеняет» стандартную и действует вместо нее. Может ли это привести к ошибкам? Теоретики нашли такой случай. Если внутри блока с «затененной» процедурой есть вложенный блок, автор которого хотел как раз обратиться к стандартной (это, естественно, отличник), то ему вместо стандартной вызовется какая-то другая. В этих случаях нужно явно описывать встроенные функции с единственным атрибутом BUILTIN или РОДНАЯ, например:

```
DECLARE SIN BUILTIN;
ОПИСАНИЕ SIN РОДНАЯ;
```

Встретив подобное описание, транслятор проверяет, есть ли такая среди его встроенных и если есть, то через голову всех старших блоков, достает нужную ссылку прямо из самого внешнего блока.

Но без таких редких случаев транслятору абсолютно все равно, что у него в таблице больше функций, чем встретилось в программе. Для него все они одинаковы и без затрат можно ввести в язык большое число встроенных функций!

Признаюсь, тут я загнул. Не все встроенные функции так уж неотличимы от не встроенных. Например, у функции SUBSTR или ПОДСТРОКА бывает и два и три параметра и транслятору приходится все же разбирать ее иначе, чем остальные. А, например, встроенные функции преобразования явно меняют тип результата. Наконец, встроенная функция LENGTH или ДЛИНА иногда заменяется константой или обращением к переменной, а не вызовом процедуры. Разбор таких особенных встроенных функций занимает в трансляторе столько же места, сколько разбор выражений. И все-таки затраты на ввод в язык встроенной функции действительно невелики. Осталось лишь найти золотую середину по части их количества и качества.

Вопрос о встроенных в язык функциях – это вопрос очень важный. Не менее важный, чем структура языка и программы. Структура языка плюс набор его встроенных функций, по сути, определяют стиль и подход программиста к решению задач. К тому же, это набор рабочих повседневных инструментов программиста, который должен быть удобным и быстро применяемым.

Если набор встроенных функций очень большой – язык будет труден в понимании и осваивании. А если очень маленький или вообще отсутствует – придется по любому пустяковому поводу изобретать свой велосипед или обращаться к описаниям библиотек, размер которых может быть большим (больше набора встроенных), а структура неочевидной.

Я не понимаю гордости заявления, что, например, операторы ввода-вывода не входят в язык Си, а являются обычными процедурами. Для программиста именно входят потому, что изучение языка он начинает с них, а не пишет сразу свои процедуры обмена. То, что эти процедуры находятся в отдельной библиотеке и не входят в транслятор, по-моему, должен помнить только транслятор, а не программист.

Так же и со встроенными функциями. Считаю, что в языке обязательно должен быть минимальный набор общепринятых, привычных действий. Такой набор еще и дает начинающему программисту подсказку, намек на то, как начать составлять алгоритмы, не задаваясь при этом вопросом, откуда транслятор все это возьмет. Без такого набора работа начинающего может действительно стать похожей на работу машины Тьюринга, где самая простая задача превращается в астрономическое число действий.

Например, я видел программу новичка, который не дочитал описание встроенных функций и пытался чуть ли не в 20 строках своей программы выяснить, задан ли ключ при ее вызове, вместо того, чтобы написать одной единственной строкой что-нибудь вроде:

```
ЕСТЬ_КЛЮЧ_М=INDEX(КОМАНДНАЯ_СТРОКА,'/М')>0;
ЕСТЬ_КЛЮЧ_М=ИСКАТЬ(КОМАНДНАЯ_СТРОКА,'/М')>0;
```

Также помню, один мой коллега показал мне тоненькую синюю книжечку «Язык программирования Си (автор Болски)» и сказал, что это все, что нужно ему для работы. Но через пару дней я попросил его немного изменить обработку текста в его программе. «Да ну его! Это надо искать в библиотеке подходящую функцию» был ответ. Я удивился, обработка была настолько простой, мне даже в голову не пришло, что в языке ничего нет для поддержки решения. Это я возвращаюсь к вопросу о 2000 страницах документации PL/1. Две тысячи это, конечно, так, для красного словца. Никто же их не суммировал. А сколько страниц, помимо описания языка в тоненькой книжечке, нужно было бы прочитать этому моему коллеге, чтобы выполнить несложную задачу? И как их читать, эти библиотеки? Как быстро узнать, какая из имеющихся процедур подходит? Я, например, мучился с описанием библиотек Windows, представляющим прямо запутанный клубок со многими концами ниток.

Набор встроенных функций PL/1 сначала казался и мне великоватым, а ведь напомню, это всего лишь подмножество языка. Но потом я понял, что такой набор невозможно придумать компактным умозрительно, так

сказать, «в тиши кабинета». Он получается на основе многолетней практики. За много лет работы мне действительно потребовались многие встроенные функции, о которых я читал, изучая язык. Вы будете смеяться, но когда было нужно сосчитать объем емкости, состоящей и кусочка сферы и кусочка тора, потребовался даже гиперболический синус, тоже являющийся встроенной функцией. А уж он-то казался мне совсем экзотикой, явно недостойной встраивания в язык.

Теперь я считаю, что в PL/1 встроенных функций достаточно мало для того, чтобы их можно было целостно воспринять при изучении, но достаточно много для решения очень и очень разных задач даже без привлечения специальных библиотек.

Я слежу за «бюллетенями» по PL/1 в Интернете и вижу, что IBM постоянно достраивает язык новыми встроенными функциями. Это общая беда всех давно используемых языков: они постоянно «набирают сложность».

Для опытных программистов этот процесс почти незаметен, они просто получают очередной инструмент, сам по себе вполне обозримый для понимания. Но для начинающих, это очень сложно: процесс эволюции языка они не застали, а в реальных проектах сразу сталкиваются с текущим многообразием и теряются. Я читал в Интернете черный юмор как раз на эту тему: менеджер жалуется, что PL/1 труден в изучении, а программисты отвечают ему: «Ну, а нам-то, что. Мы-то его уже знаем». В этом смысле иногда урезанное подмножество может оказаться даже лучше «целого» языка.

39. Описание встроенных функций

Нет-нет, встроенные функции описывать в программе не нужно! Это здесь их опишем в алфавитном порядке. Будем рассматривать только те функции, которые транслятор разбирает отдельно, а не как обычные процедуры, описанные в программе.

В значительной мере именно эти функции формируют «внутренний мир» программиста. Почти подсознательно он будет пытаться сводить к ним решение любой задачи. Кроме этого, формирование из встроенных функций команд для ЭВМ составляет более половины «третьей части» транслятора. Эта «третья часть» вообще значительно больше двух других частей транслятора как раз из-за большого набора операций и функций, которые надо преобразовать в команды, хотя каждый отдельный разбор и очень прост.

Многие функции имеют пару, совсем как в детской карточной игре в «Акулину» (синус - арксинус, логарифм - экспонента, искать совпадение – искать несовпадение). Дополняя друг друга в паре, функции делают данную возможность языка более полной, а сам набор функций завершенным.

Итак, приступим к описанию понятий, которые авторы посчитали

неотъемлемой частью PL/1.

Функция **ABS** убирает знак у числа, например X=ABS(X); делает из X строго положительное число или ноль. Зачем транслятору возиться с разбором такой мелочи? Дело в том, что для разных числовых типов нужно генерировать разные команды. У транслятора не одна функция ABS, а целых пять, а именно: для целых чисел размером в байт, в два байта, в четыре байта, для вещественных чисел и для точных чисел с дробной частью. А для комплексных чисел эта функция работает вообще по-другому. Нужную функцию транслятор сам выбирает из типа операнда и для программиста по-прежнему остается только одна функция.

Кстати, в языке Си торчат уши от решения «не встраивать» функции. Там есть, например, ABS и LABS и FABS. По-моему, несколько десятков дополнительных команд в трансляторе стоят того, чтобы не морочить программистам головы выбором функций по операндам. Далее я уже не буду упоминать о разных типах чисел, понятно, что транслятор так делает и во многих других функциях. Поэтому в системной библиотеке число функций гораздо больше, чем будет здесь описано.

Функция **ADDR** или **АДРЕС** возвращает адрес своего аргумента. Обычно результат этой функции запоминается в переменной типа указатель, но можно использовать ADDR АДРЕС и как параметр-выражение при вызове процедур. Жадный транслятор не отдает программисту АДРЕС от константы или выражения, хотя сам и имеет их при разборе. Данная функция в оттранслированной программе просто растворяется, преобразуясь в константы, операнды команд ЭВМ и т.п.

Функция **ASCII** возвращает строку в один символ, взятый из стандартной таблицы по указанному номеру. Например, ASCII(88) вернет латинское 'X'. Это можно применять для автоматического формирования имен файлов или чего-нибудь подобного. Да и просто выдать символ, который Вы не найдете на клавиатуре. Я не стал переводить аббревиатуру, означающую американский комитет по стандартизации ввода-вывода, на русский.

Функция **BIN** или **BINARY** или **ДВОИЧНОЕ** это одна из функций преобразования числовых типов. Главное ее назначение убрать дробную часть как класс. Причем, если операнд был точным, ответ точным и останется (превратится в целый), а если ничего не сказано, то превратится в приближенный. Странное название BINARY ДВОИЧНОЕ идет из «глубины веков» и здесь означает лишь отсутствие точной дробной части. Функция может иметь еще один параметр, показывающий, какой длины нужна отрезаемая целая часть.

Например, если X это FIXED DECIMAL(6,3) ТОЧНОЕ

ДЕСЯТИЧНОЕ(6,3) со значением 12.675, то BINARY(X,15) ДВОИЧНОЕ(X,15) вернет целое 12, хотя BINARY(12.675,15) ДВОИЧНОЕ(12.675,15) вернет приближенное 1.2000000E+01.

Вообще, по-моему, ДВОИЧНОЕ и идущее в описании ниже ДЕСЯТИЧНОЕ не самые удачные названия. Начинающих иногда они сбивают с толку. Названия идут от формы представления чисел в ЭВМ. Форма может быть или «обычная» двоичная или специальная для точных с дробной частью, которые представляются в «двоично-десятичном» коде, по четыре бита на десятичную цифру. Путаница получается потому, что точные числа могут быть как двоичными (это целые), так и двоично-десятичными (с дробной частью). В общем, функция ДВОИЧНОЕ на самом деле просто убирает двоично-десятичное представление, оставляя одно двоичное, а значит и точная дробная часть тоже исчезает.

Функция **BIT** или **БИТ** переводит свой аргумент в битовую строку, т.е. в последовательность нулей и единиц. Как и предыдущая функция, она может иметь еще один параметр-число, показывающий какой длины должна получиться строка. Иногда это функция вызывает непонимание у программистов. Например, если целое число перевести с помощью этой функции, то получится вовсе не та комбинация бит «как число в памяти и лежало», а положительная мантисса, прижатая влево. Таким образом, например, BIT(-4,4) БИТ(-4,4) вернет строку ’1000’, что к удивлению программиста совсем не соответствует отрицательному числу. А вот если, например, нужно вклеивать значение мантиссы числа как кучку бит вместе с информацией другого типа, то с помощью этой функции результат будет вполне разумен.

Функция **BOOL** или **БУЛЕВА_АЛГЕБРА** реализует все возможные типы операций логики, основанной математиком Джоном Булем. (Может мне надо было перевести и фамилию? А что? АЛГЕБРА БЫКОВА! Звучит!). В языке уже есть отдельно операции «И», «ИЛИ», «НЕ». Но создатели языка захотели окончательно решить вопросы логики и разрешили все 16 возможных комбинаций ДА-НЕТ, включая бессмысленные «всегда ДА» и «всегда НЕТ». Транслятор честно генерирует любую из 16 заданных логических операций. Например, логическую операцию «исключающее ИЛИ», она же «сравнение на тождество» можно записать как

```
X=BOOL(Y,Z,’0110’Б);
X=БУЛЕВА_АЛГЕБРА(Y,Z,’0110’Б);
```

Здесь Y и Z сравниваемые строки бит, а третий операнд – описание «логики»: что ставить в ответе, если оба разряда нули, если первый ноль, второй один, если первый один, второй ноль, наконец, если оба единицы. Кстати, эту конкретную операцию можно применять, например, для шифрования данных. Большинство других операций мне пока не

пригодились. А зачем создатели Алгола вводили в язык операцию «импликация» для меня вообще осталось загадкой. Я так и не смог придумать ни одного практического примера для нее.

Функция **CHAR** или **CHARACTER** или **ТЕКСТ** предназначена для перевода своего аргумента в текстовую строку. Можно указать необязательным параметром длину такой строки. Уже рассматривался пример ее использования. Как и функцию БИТ, функцию ТЕКСТ можно явно не указывать, особенно при работе с числами, транслятор при переводах подставит ее неявно. Например, при текстовом вводе-выводе все время неявно используется эта функция.

Функция **CEIL** или **ЦЕЛОЕ_НЕ_МЕНЬШЕ** ищет наименьшее целое, большее или равное своему аргументу. По сути, это округление в большую сторону. Например, ЦЕЛОЕ_НЕ_МЕНЬШЕ(7.9) вернет 8. Сами понимаете, что дальше обязательно будет описано и округление в меньшую сторону.

Функция **DEC** или **DECIMAL** или **ДЕСЯТИЧНОЕ** в противоположность функции ДВОИЧНОЕ «приклеивает» точную дробную часть. Поэтому, у нее может быть даже три параметра: аргумент, длина всего числа и длина точной дробной части. Например, DECIMAL(125,6,2) ДЕСЯТИЧНОЕ(125,6,2) вернет 125.00. Хотя у 125.00 дробной части и нет, но зато уже появилось место для нее в последующих точных расчетах.

Функция **DIM** или **DIMENSION** или **РАЗМЕРНОСТЬ** возвращает размерность указанного массива и по заданному индексу как целое число. Это очень удобно при задании границ циклов: можно писать размерность только в описании массивов. На самом деле эта функция превращается просто в константы в программе.

Я много экспериментировал с доделками этой функции. Например, часто массивы одномерные и раздражает необходимость писать номер размерности всегда единицу. Сделал, что единица подставляется по умолчанию. Потом сделал, что если задать размерность ноль – выдается максимально возможная длина строки-аргумента. А если задать размерность вообще минус единица – можно получить размер заданной структуры в байтах. Таким образом, я использовал эту функцию для доставания размеров разных объектов в программе. В IBM предпочли ввести встроенные функции типа SIZEOF

Функция **DIVIDE** или **ДЕЛИТЬ** как и следует из названия, делит числа. Но ведь деление и так есть в выражениях, зачем же нужна еще и функция? Она нужна для явного управления точностью деления, когда получается дробный результат. Никаких дополнительных команд

деления эта функция не вызывает, но принимает от программиста указание, как записывать результат. Если оба аргумента функции (делимое и делитель) целые типа FIXED(7) ТОЧНОЕ(7), вызов функции часто просто подтверждает транслятору, что программист не забыл, что он делит целые числа с отбрасыванием остатка от деления. У этой функции может быть даже четыре аргумента: делимое, делитель, общая длина ответа и длина дробной части.

Функция **FIXED** или **ТОЧНОЕ** преобразует свой аргумент в число с точным представлением. Английское название идет от формы с «фиксированной» десятичной точкой, которая и обеспечивает постоянное место для целой и дробной частей, делая возможным точное представление. Как и у функции ДЕСЯТИЧНОЕ, у данной функции может быть до трех параметров, задающих аргумент-число, общую длину результата и размер дробной части.

Функция **FLOAT** или **ВЕЩ** или **ВЕЩЕСТВЕННОЕ** по сути обратна предыдущей функции, она преобразует аргумент в приближенное представление. Английское название идет от формы с «плавающей» десятичной точкой, которая обеспечивает постоянную общую длину числа при изменяемых длинах целой и дробной частей. Форматы таких чисел давно «забронзовели» в стандарте IEEE-754 и поддерживаются всеми современными ЭВМ. В функции может быть второй, необязательный параметр, задающий размер мантиссы. Реальной границей является мантисса в 24 бита, все числа с большей мантиссой занимают 8 байт, а меньше или равные – 4 байта. Т.е. так их переводит транслятор: только в два типа, чтобы процессор не мучился.

В данном описании это была последняя функция преобразования данных. Поскольку типов много, много и встроенных функций преобразования. Вообще, это общее правило языков: если есть типы, которые можно переводить друг в друга, то обязательно в языке есть и готовые средства для этого.

Функция **FLOOR** или **ЦЕЛОЕ_НЕ_БОЛЬШЕ** округляет в меньшую сторону, естественно дополняя уже рассмотренную функцию ЦЕЛОЕ_НЕ_МЕНЬШЕ. Например, ЦЕЛОЕ_НЕ_БОЛЬШЕ(7.9) вернет 7. Замечу, что часто под округлением понимают не строго эту функцию, а FLOOR(X+0.5E0) ЦЕЛОЕ_НЕ_БОЛЬШЕ(X+0.5E0).

Функция **HBOUND** или **ВЕРХ_ГРАНИЦА** выдает верхнюю границу указанного массива по указанной размерности. Если бы описания массивов разрешали границу только от одного или от нуля, хватило бы и одной функции РАЗМЕРНОСТЬ. Если массив имеет только одну размерность, в функции можно указывать только имя. Никакой реальной процедуры не вызывается, она превращается в константы.

Функция **INDEX** или **ИСКАТЬ** является одной из основных функций работы со строками. Она ищет в указанной строке указанный образец и возвращает номер позиции первого найденного образца в строке или ноль (если не нашла). Например, ИСКАТЬ('01234567897','7') вернет 8. Мне потребовалось доработать функцию и ввести необязательный третий параметр – позицию в строке, с которой искать (по умолчанию с 1). Например, ИСКАТЬ('01234567897','7',9) вернет 11. Таким образом, если требуется найти все образцы в строке, от нее не нужно отрезать куски, а нужно лишь менять этот третий параметр в функции поиска.

Функция **LBOUND** или **НИЖ_ГРАНИЦА** выдает нижнюю границу указанного массива и приходится сестрой-двойняшкой функции ВЕРХ_ГРАНИЦА.

Функция **LENGTH** или **ДЛИНА** предназначена для определения длины текстовых строк. Отличие этой функции от функции РАЗМЕРНОСТЬ в том, что здесь иногда возвращается переменное значение, а не константа, т.е. генерируется команда доставания текущего размера строки. Я немного расширил эту функцию. Очень полезно знать длину «открытого» файла в программе, особенно файла «отображенного на память». Поэтому в данном трансляторе аргументом функции LENGTH ДЛИНА может быть и переменная типа «файл».

Функция **MAX** возвращает наибольший из двух аргументов. Если нужно найти наибольшее из нескольких аргументов, приходится вкладывать функции друг в друга.

Например, X=MAX(Y1,MAX(Y2,MAX(Y3,Y4)));

Функция **MIN** возвращает наименьший из двух аргументов и приходится сестрой-близняшкой предыдущей функции.

Функция **MOD** является дополнением к функции DIVIDE ДЕЛИТЬ. Та определяла, что и как делать с частным, а эта определяет остаток от деления. Например, конструкция типа

ЕСЛИ MOD(СЕКУНДЫ,60)=0 ТОГДА БИБИКНУТЬ;

Заставляет программу «бибикать» раз в минуту.

Функция **NULL** или **ПУСТО** на самом деле является встроенной переменной и случайно «затесалась» в ряды функций. При работе с указателями и памятью часто нужно убедиться, что какой-то адрес уже есть. Тогда он точно не будет равен этой переменной. Иногда ПУСТО специально помещают в указатель, отмечая конец «связанного» списка, т.е. списка, где каждый элемент через указатель показывает на следующий. Как нетрудно догадаться, ПУСТО это просто ноль так

красиво названный для транслятора.

Функция **RANK** является обратной по отношению к функции ASCII, т.е. возвращает номер заданного символа-аргумента в стандартной таблице. Например, RANK('Y') вернет 89. Я использовал эту функцию, например, для организации переключателя при разборе текста. В зависимости от прочитанной буквы управление передавалось на метку с соответствующим индексом:

```
ИДТИ СИМ(RANK(СИМВОЛ));
…
СИМ(32): …./* БЫЛ ПРОБЕЛ*/
```

Функция **REPLACE** или **ЗАМЕНИТЬ** ищет в заданной строке все вхождения указанного образца и заменяет его на другой образец. Эта функция очень похожа на функцию ПЕРЕВЕСТИ, но та работает с одиночными символами, а эта со строками. Например,

ЗАМЕНИТЬ(СТР, 'палки', 'елки'); заменит в строке СТР все найденные слова «елки» на слова «палки».

Функция **ROUND** или **ОКРУГЛИТЬ** округляет заданное число на указанное число разрядов вправо или влево от точки. Например,

ОКРУГЛИТЬ(34567.12345,-3) возвращает 35000.00000

ОКРУГЛИТЬ(34567.12345, 3) возвращает 34567.12300

Мне казалось, что округлений в языке и так навалом, поэтому это уже лишнее. Но однажды потребовалось запомнить очень большие расчеты. Нужно было иметь файлы поменьше. Поэтому, выделять 8 байт на каждое число не хотелось. С другой стороны, писать по 4 байта на число - это значит потерять точность. Оператором X=ОКРУГЛИТЬ(X,32); я по сути перевел мантиссы чисел в «пятибайтовые» значения. Т.е. стало возможно писать в файлы только часть их мантиссы с правильным округлением младшего разряда.

Я использовал функцию ОКРУГЛИТЬ и для других целей. Если аргумент этой функции строка бит, то она не округляется, а сдвигается «по кругу» влево или вправо на указанное число раз. В конце концов, исходное ключевое слово ROUND имеет и значение «кругом». Например, X=ОКРУГЛИТЬ(X,4); где X – строка из восьми бит, поменяет «тетрады» байта местами. Изначально в PL/1 не было функции сдвига битовых строк, а использовать для этого функцию ПОДСТРОКА иногда слишком громоздко.

Функция **SIGN** является дополнением функции ABS. Та достает значение, выбрасывая знак, а эта наоборот, выбрасывает значение, оставляя знак. Поскольку сам по себе знак числа взять нельзя, он переводится в удобные +1, 0,-1.

Функция **SUBSTR** или **ПОДСТРОКА** является, наверное, самой часто используемой функцией для строк. Она выкусывает из указанной строки кусочек заданной длины, начиная с указанного места. Заданное место обычно получается с помощью функции ИСКАТЬ. Если длина куска явно не указано, берется весь кусок до конца строки.

Например, отрезать кусок строки до первой точки можно так:

```
I=INDEX(CTP,'.'); CTP=SUBSTR(CTP,I+1);
I=ИСКАТЬ(CTP,'.'); CTP=ПОДСТРОКА(CTP,I+1);
```

Функция ПОДСТРОКА как и еще одна такая же «ненормальная» функция МАШ_КОД может стоять и в левой части оператора присваивания, например, заменить вторую букву в строке CTP на букву «А» можно и так: ПОДСТРОКА(CTP,2,1)='A';

Функция **TRANSLATE** или **ПЕРЕВЕСТИ** заменяет символы в указанной строке, используя две строки-таблицы: нужных символов и имеющихся в строке символов. Например, перевод некоторых букв из малых в большие в строке CTP можно выполнить оператором:

```
CTP=ПЕРЕВЕСТИ(CTP,'ABCDEF','abcdef');
```

Функция **TRIM** или **ОЧИСТИТЬ** выбрасывает из начала и из конца строки все подряд идущие пробелы, например:

```
CTP=ОЧИСТИТЬ(CTP);
```

Я чуть подправил эту функцию, приравняв к пробелам еще и нулевые коды.

Функция **TRUNC** или **ЦЕЛАЯ_ЧАСТЬ** это общий вид округления чисел, включая отрицательные. Для положительных чисел она совпадает с ЦЕЛОЕ_НЕ_БОЛЬШЕ, а для отрицательных - с ЦЕЛОЕ_НЕ_МЕНЬШЕ. Например,

ЦЕЛАЯ_ЧАСТЬ(52.146) возвращает 52
ЦЕЛАЯ_ЧАСТЬ(-52.146) возвращает -52

Функция **VERIFY** или **ИСКАТЬ_НЕСОВПАДЕНИЕ** является по сути обратной к функции ИСКАТЬ. Ее задача найти первое несовпадение строки и образца. Например,

ИСКАТЬ_НЕСОВПАДЕНИЕ('ABCDE','ABDE') вернет 3.

Функция **UNSPEC** или **МАШ_КОД** является совершенно особой функцией. Я даже поставил ее не совсем по алфавиту. Она позволяет брать из памяти и класть в память прямо «сырые» биты. В большинстве языков ничего подобного нет, а создатели этих языков утверждают, что и ни в коем случае нельзя вводить ничего подобного.

Я с этим не согласен. Иногда суровая необходимость заставляет обращаться к подобным вещам, плюя на пресловутую «переносимость» программ. Я пошел еще дальше и решил, как тульский Левша, превзойти

этих «аглицких мастеров». В данном трансляторе можно даже написать одиночную функцию МАШ_КОД (не в операторе присваивания). Тогда строка бит, указанная в аргументе пишется прямо среди команд программы! Например, контрольная точка для отладчика задается командой INT 3, имеющий код ’СС’Б4. В программе можно записать, что-нибудь вроде: ЕСЛИ X<0 ТОГДА МАШ_КОД(’СС’Б4); и программа вылетит в отладчик, если X станет меньше нуля.

Напомню, что остальные встроенные функции (тригонометрия, логарифмы и т.п.) транслятор обрабатывает как самые обычные процедуры. Пожалуй, стоит отдельно отметить еще встроенные функции **TIME ВРЕМЯ** и **DATE ДАТА**, которые выдают текущий момент жизни ЭВМ и тоже влияют на «внутренний мир» программиста на PL/1.

Кажется невероятным, что такой странный и, в общем, небольшой набор функций с упором на математику – это все, что было вставлено в язык, претендовавший на применение во всех классах задач. За много лет работы я сталкивался, конечно, не со всеми возможными, но с большим числом совершенно разных задач и все эти задачи были реальными, не надуманными. Подтверждаю, что предложенный набор функций обладает большой универсальностью и целостностью. Полагаю, что это получилось из-за того, что функции появлялись по мере попыток использования PL/1 в разных областях применения.

40. Работа с памятью

Мы уже говорили о важности памяти и о том, как ее распределяет транслятор. Теперь поговорим о том, как может распределять память программист. Основателям Алгола даже не приходило в голову, что создание объектов в программах прямо в процессе исполнения, а не в описаниях станет совершенно обычным делом.

PL/1 предоставляет для этой цели два оператора: ALLOCATE ДАТЬ_ПАМЯТЬ и FREE ВЕРНУТЬ_ПАМЯТЬ, выполняющих зеркальные по отношению к друг другу действия.

Разумеется, ALLOCATE ДАТЬ_ПАМЯТЬ и FREE ВЕРНУТЬ_ПАМЯТЬ это ключевые слова, после которых следует список имен через запятую и обязательная точка с запятой. В первом случае список показывает «куда насыпать» выделенную память, а во втором – откуда «ссыпать ее обратно». Имена в списке могут быть только «базированного» класса памяти, т.е. такие, которые могут использоваться только с указателями.

Обычно указатель явно присутствует в описании переменной, тогда в этих операторах его указывать не надо, нужны только имена объектов, а указатель транслятор подставит автоматически. Именно в этот указатель

и запишется адрес начала выделенной памяти после ALLOCATE ДАТЬ_ПАМЯТЬ. Из этого же указателя берется адрес для оператора FREE ВЕРНУТЬ_ПАМЯТЬ.

Однако, указатель можно написать и явно, например:

```
ALLOCATE X SET(P);
ДАТЬ_ПАМЯТЬ X В_УКАЗ(P);
```

тогда выделенная память размером с объект X запишется в указатель P, независимо как был описан X: с указателем P, с другим указателем или вообще без указателя.

Со временем оказалось, что при выделении памяти даже и имя не всегда нужно. Например, адрес выделенной памяти нужно передать как параметр, а для этого достаточно одного указателя. В таких случаях вместо имени можно написать выражение в круглых скобках – размер требуемой памяти в байтах, например:

```
ALLOCATE (10000) SET(P);
ДАТЬ_ПАМЯТЬ (10000) В_УКАЗ(P);
```

Интересно, что и кому-то в IBM и мне эта мысль (писать выражение) пришла независимо, как наиболее очевидная. Но я оставил оператор в его прежнем виде, а они сделали из него еще и оператор присваивания (типа P=ДАТЬ_ПАМЯТЬ(10000);).

Как бы то ни было, программист может отрезать себе памяти, даже не сообщая, для каких объектов он это делает. Это породило маленькую смешную проблему: при возврате памяти нужно тоже указать имя, а никакого имени-то и нет, есть только указатель. Приходится вводить любую «базированную» переменную только для этого случая:

```
DECLARE P POINTER, X FIXED BASED;
FREE P->X;
ОПИСАНИЕ P УКАЗАТЕЛЬ, X ТОЧНОЕ ОСНОВА;
ВЕРНУТЬ_ПАМЯТЬ P->X;
```

Здесь переменная X нужна только для соблюдения правил записи.

Возможность манипулировать памятью и по ходу работы создавать сколь угодно сложные объекты сделала современное программирование необычайно мощным. Но и ошибки посыпались такие, какие невозможно было себе представить, например, в Алголе.

Я уже говорил, что у программиста есть две прекрасные возможности все сломать: указать индекс массива вне границ и ошибиться в передаче параметров. Теперь появляется третья: испортить память при распределении. Обычно считается, что контролировать такие ошибки даже на этапе исполнения (не говоря уж о трансляции) невозможно – адреса памяти ведь могут быть любыми! Хорошо, если неправильный адрес хоть вылетает за пределы программы, тогда этот случай может поймать операционная система (например, Windows). А если не вылетает?

Давайте разберемся, можно ли хоть что-то проверять?

Одна из типичных ошибок – это попытка использовать указатель после оператора FREE ВЕРНУТЬ_ПАМЯТЬ. В указателе остается больше не действующий адрес, который после очередного выделения памяти может показывать, например, куда-то в середину нового объекта. Получилась, как говорят, «висячая ссылка». Я доработал транслятор так, чтобы он при генерации процедуры освобождения памяти еще и пытался записать ноль (или, если хотите, NULL ПУСТО) в переменную, откуда берется указатель. Разумеется, указатель с этим адресом, к несчастью, уже может быть запомнен в другом указателе или вообще является возвращаемым значением процедуры-функции. В таких случаях обнулить уже ненужный указатель не получится. Однако в остальных случаях (а их большинство) «висячие ссылки» после такого приема исчезают. Жить мне стало заметно легче.

Сложнее оказалась вторая проблема: как поймать момент, когда программа как слон в посудной лавке начнет давить собственные объекты в памяти? Обычно, это происходит тогда, когда в выделенную память пытаемся записать объект большего размера, чем выделенный кусок.

Я использовал следующие обстоятельства: вся память, доступная для распределения (иногда ее называют «куча») представляет собой список свободных и занятых кусков. Каждый кусок имеет ссылку на предыдущий и следующий (последний на первый). Если при освобождении два свободных куска оказываются подряд, они автоматически сливаются в один, чтобы свободная память не разбивалась на россыпь маленьких и поэтому бесполезных кусочков.

Я обратил внимание на то, что когда случается несчастье с разрушением памяти, то всегда нарушается и эта стройная система. Т.е. после затирания одним объектом другого совершенно невероятно, чтобы остальные условия сохранились: чтобы каждый кусок имел одну ссылку строго вперед, а другую строго назад, чтобы не было двух формально свободных подряд, чтобы общий объем всех свободных и занятых кусков остался неизменным. Какие бы случайные данные не попали внутрь области памяти, они обязательно нарушат одно из этих правил (конечно, если служебная информация вообще испортилась).

Я написал программу почти такую же, как служебная процедура выделения памяти, но она ничего не выделяет, а просто пытается «добежать до конца» списка всех свободных и занятых кусков, считая по пути их общий размер. Если все правила соблюдены, а общий размер не изменился (а с чего ему меняться?), значит, память еще цела, иначе нужно выдавать сигнал ошибки с указанием последнего места, где все еще было нормально. Понятно, что ошибка будет замечена не в момент затирания, а лишь при запуске такой программы, но и это уже кое-что.

Потом я сделал разновидность этой программы, которая не сразу «добегает до конца», а проверяет за раз по 100 очередных кусочков.

Такая программа работает всегда предсказуемое время. Я вставляю ее в «главный цикл» своей программы и при небольших накладных расходах постоянно контролирую целостность памяти. И еще не было случая, чтобы при разрушении памяти такая сигнализация не сработала бы.

41. Работа на «низком» уровне

Ох уж эти вредные программисты! Языки программирования специально были сделаны для того, чтобы они не смели лазить на этот самый «низкий» уровень и якшаться там с регистрами, портами ввода-вывода и прочими странными вещами, никак не отраженными в большинстве языков. А программисты все лезут и лезут.

К каким неприятностям это может привести?

Первый довод, который приводят теоретики: программа становится непереносимой, не «мобильной». Мне кажется, что мобильность становится каким-то жупелом. Я как-то не наблюдаю особого разнообразия процессорных архитектур. Нет, я, конечно, знаю, что есть «мэйнфреймы» со своими процессорами, а в сотовых телефонах и плейерах вряд ли стоят «пентиумы». Но, знаете ли, и я свои расчеты на плейер переносить не собираюсь. Уж если программа полезла обращаться к аппаратуре, вопрос о ее переносимости обычно вообще не стоит.

Второй довод: трюки с регистрами и прямым залезанием в память чреваты ошибками, которые транслятор уже не может найти. Но «законный» способ через написание процедур на ассемблере тоже транслятором не контролируется. Уж лучше следовать принципу: не мешайте программисту сделать то, что он хочет (потому, что он это делает по необходимости).

Посмотрим, что может предложить PL/1. У него нет «регистрового» типа памяти, как, например, у языка Си. Но данный тип, это не низкий уровень, а, скорее, забота об эффективности. Транслятор с Си не сообщает программисту, в какой регистр он засунул переменную и вообще сделал ли так, несмотря на указание. С этой точки зрения в рассматриваемом трансляторе все переменные «регистровые». Третья часть транслятора постоянно отслеживает даже целую «модель процессора архитектуры IA-32», чтобы использовать регистры максимально возможно. Такая модель, например, помогает транслятору понять, что если поместить результат в регистр AL, то изменятся регистры AX, EAX и еще, возможно испортится результат умножения, оставшийся в EAX+EDX.

Я уже упоминал об удивительной функции низкого уровня МАШ_КОД. Удивительность ее состоит в том, что, во-первых, авторы языка вообще разрешили такую функцию, а во-вторых, эта функция может стоять и в левой части оператора присваивания.

Например, МАШ_КОД(X)=’11001100’Б; запишет в память, где была

переменная X прямо строку с указанными нулями и единицами. А что они означают? А черт его знает! Кто пишет подобные вещи, уж, наверное, понимает, что он пишет и как это все выглядит внутри ЭВМ.

И все-таки на практике, несмотря на такие возможности, а также на возможности той же функцией МАШ_КОД писать среди команд программы любые коды, потребовался еще и доступ к регистрам. Причем, не столько доступ, сколько связь между переменными языка и регистрами.

Рассмотрим такой пример: в программе требуется читать состояние «параллельного интерфейса» из порта 378. Разумеется, можно написать отдельную процедуру на ассемблере, возвращающую нужное значение, и объединить ее с модулями программы. Но чего не хватает, чтобы читать прямо из программы на PL/1? Команда загрузки регистра DX значением 378 это просто постоянный код, который записывается как МАШ_КОД('66BA7803'Б4); Сама команда чтения IN AL,DX представляет еще одну константу и ее можно записать как МАШ_КОД('EC'Б4); А вот как теперь достать нужное значение из AL? И я ввел в транслятор правило: если переменная типа «целое число» или «строка бит» или указатель имеет имя, начинающееся с символа «?» и совпадающее с именем регистра IA-32. значит, это не переменная, а сам этот регистр. Ведь транслятор все равно пытается распределить переменные по регистрам, а тут может прямо использовать указанный регистр. Таким образом, вводим переменные ОПИСАНИЕ (X, ?AL) БИТ(8); и поставленная задача решается заклинанием:

```
МАШ_КОД('66BA7803'Б4);
МАШ_КОД('EC'Б4);
X=?AL;
```

а требуемое значение попадает в переменную X. Кстати, имея доступ к регистрам, теперь можно и команду МАШ_КОД('66BA7803'Б4); заменить простым оператором присваивания ?DX='0378'Б4; тогда в программе останется только один код EC, непредставимый в языке.

Знающие Windows XP сразу скажут, что так все равно не получится: обращение к портам из обычной программы запрещено операционной системой в принципе. Интересующихся как это можно сделать отсылаю к последней, отдельной главе. «Отдельной» я ее назвал потому, что по содержанию она не связана с остальными. Но, в конце концов, и Гоголь вставлял в свою поэму отдельную главу «Повесть о капитане Копейкине». Правда, у Николая Васильевича и поэма вышла получше моей, но хотя бы в этой части (одна глава вроде не по теме) я с ним сравнялся.

Конечно, функцией МАШ_КОД большие программы написать трудно, да и не нужно, но в подобных простых случаях все получается легко.

Что еще может требоваться? Самому рассчитать адрес в памяти и достать что-нибудь оттуда? Ну, в PL/1 это несложно. Правда, к

указателю ничего прибавлять нельзя, но можно «наложить» на указатель другую переменную типа целое число и менять ее как нужно. При этом будет меняться и указатель. Например:

```
DECLARE P POINTER,
        X FIXED(31) DEFINED(P),
        Y FIXED(31) BASED(P);
ОПИСАНИЕ Р УКАЗАТЕЛЬ,
          X ТОЧНОЕ(31) НА_МЕСТЕ(Р),
          Y ТОЧНОЕ(31) ОСНОВА(Р);
…
Y=1;       // переменная в памяти по адресу в Р
X=X+10; // фактически прибавляем 10 к указателю Р
Y=1;       // переменная «сместилась» в памяти на 10 байт
```

Возвращаясь к вопросу о мобильности программ: не так страшно, если программу просто не удается запустить на другой ЭВМ, сколько то, что иногда совершенно непонятно, из-за чего мобильность пропала. В PL/1 авторы постарались, чтобы такие места можно было легко найти.

Во-первых, это опции ENVIROMENT ДЛЯ_ОС в операторе открытия файла и заголовке главной процедуры.

Во-вторых, это все места использования функции UNSPEC МАШ_КОД.

В-третьих, это все «наложения» переменных атрибутом DEFINED НА_МЕСТЕ.

Для данного транслятора еще и в-четвертых: все переменные в программе, начинающиеся с вопросительного знака. Кстати, с такого знака начинаются и все служебные процедуры и переменные, поэтому программисту нежелательно начинать свои имена с этого знака.

Добавлю, что иногда мобильность касается не разных типов ЭВМ, а разных операционных систем. В этом случае мобильной должна быть, похоже, только системная библиотека. Это ведь через нее программа общается с операционной системой. Например, оператор «открытия» файла транслятор переводит в вызов служебной процедуры открытия, где-то в недрах которой и стоит вызов процедуры открытия файла уже для Windows. Если механизм динамических библиотек, например, в Lunix действует так же, то достаточно заменить полтора десятка вызовов Windows на вызовы Lunix в системной библиотеке и этот PL/1 «переедет» на Lunix. Пока, правда, не пробовал.

42. Эффективность

Раз уж заходила речь о регистрах и эффективности, поговорим о ней подробнее. Всегда хочется иметь эффективность побольше, непонятно только откуда ее взять. Например, я не удовлетворен эффективностью всем известных программ. В 1987 году мой IBM PC загружался за 30

секунд, а сейчас… А Вы знаете, насколько с тех пор ЭВМ стали быстрее? А я случайно знаю точно. Сейчас поделюсь.

Как-то попробовали мы запустить древнюю программу на «турбо»-Паскале, но ничего не вышло - деление на ноль. Стал я разбираться, на какой ноль там делится и выяснил любопытную вещь: оказывается, в начале работы старых программ всегда выясняется скорость их работы относительно «исходного» IBM PC. Т.е. в течение доли секунды много раз выполняется эталонный кусок программы, и затем это число раз делится на число раз, которое показал первый IBM PC. Годы шли, и каждый год скорость ЭВМ почти удваивалась. Наконец, частное перестало помещаться в 16 разрядов и получилось не деление на ноль, а другой случай: «непредставимое частное». Я разделил эти числа на калькуляторе и получил, что скорость ЭВМ, стоящей у меня сейчас на столе, возросла относительно первой IBM PC в 118351 раз.

Но это я опять отвлекся. Эффективность программы во многом зависит и от эффективности самого процессора.

В описании процессора фирмы «Intel» есть большая глава, посвященная эффективности, набитая противоречащими друг другу рекомендациями. Типа: не пользуйтесь сложными командами, они работают медленнее простых, но не пишите много простых команд, они раздувают код и его медленнее доставать из памяти. Для разгрузки конвейеров процессора пишите подряд команды, не зависящие друг от друга. А откуда такие команды возьмутся в большинстве задач, где результат как раз и зависит от предыдущих вычислений? Кстати о конвейере, например, в «Pentium 4» конвейер имеет 20 ступеней и почему-то называется гипер-конвейером (предыдущий назывался супер-конвейером). А если введут еще ступень, какую приставку к названию придумают? Могу только предложить приставку «супер-пупер».

Изначально в PL/1 много внимания уделялось эффективности, предполагалось, что язык будет обязательно компилируемым, т.е. исполняемые команды готовятся транслятором заранее, а не выполняются во время каждого разбора программы. Выполнение команд прямо при разборе программы называется интерпретацией (так, например, был реализован язык Ява) и работает гораздо медленнее, как бы не старались убедить в обратном. Особенно, если программа-интерпретатор, которая все это и делает, сама написана на языке, который интерпретируется. Справедливости ради, следует заметить, что интерпретация и компиляция не имеют отношения к языкам программирования и программу на любом языке можно обрабатывать и так и так.

Теперь рассмотрим, как описываемый транслятор генерирует команды передачи параметров, поскольку он делает это иначе, чем многие другие трансляторы.

Пусть, например, в программе стоит вызов процедуры ТОЧКА(X,Y);

Обычно трансляторы генерируют команду засылки в память-«стек»

значение переменной Y, затем засылки в стек X, а затем генерируется вызов процедуры ТОЧКА. Эти три команды зависят друг от друга, поскольку все используют стек.

Данный транслятор один раз готовит список из двух адресов-констант X и Y где-то в памяти и генерирует две команды: передачу всегда в один и тот же регистр адреса списка, а затем команду вызова процедуры. Эти две команды уже не зависят друг от друга и могут в процессоре выполняться параллельно.

Конечно, под списки адресов расходуется память, но зато нет команд засылки параметров в стек, которые тоже расходуют память. Независимо от числа параметров, подготовка к вызову занимает всегда одну команду, кроме самого вызова. Эффективно ли это? Пожалуй, да. Хотя, рекурсивные процедуры как раз эффективнее реализовывать через стек. Кстати, в свои служебные процедуры транслятор генерирует передачу параметров еще быстрее - просто через регистры. Но служебные процедуры – особые, да и обычно имеют мало параметров, поэтому так легко сделать.

Другой момент. При работе с вещественными числами данный транслятор не генерирует прямо команды арифметики таких чисел. А манипулирует семейством маленьких служебных процедур: сложить, вычесть, умножить, разделить, сравнить. Каждая из процедур имеет два параметра и выдает значение-результат. С одной стороны, эффективность снижается, так как требуется выполнять вызов и возврат. С другой стороны, объем программы уменьшается, так как нет длинных команд, а служебные действия (вроде проверок на ноль) выполняются только внутри служебных процедур, не повторяясь во многих местах программы.

С расчетами вещественных значений у меня связана одна поучительная история, на которую я отвлекусь, тем более что она тоже касается эффективности.

Когда-то я помогал перевести довольно сложные программы с ЕС ЭВМ на персональный компьютер. Чтобы не дай бог не ошибиться, было сделано много промежуточных печатей, и я пошагово сравнивал результаты работы своего транслятора с результатами ЕС ЭВМ. Тут же нашлись отличия, хотя и не в старших знаках. Оказывается, использовались и данные длиной в 4 байта (ВЕЩЕСТВЕННОЕ(24)) и длиной в 8 байт (ВЕЩЕСТВЕННОЕ(53)). Когда число из 8 байт преобразовывалось в 4 байта, расхождений не было. Но когда число из 4 байт преобразовывалось в 8, начинались отличия! Я был удивлен: ведь я просто записывал 4 байта в процессор и тут же доставал это же значение, но уже как 8 байт. Где же ошибка? Например, если я так переводил значение 1.2512E+00, то почему-то получал в результате 1.25119996070860E+000, а на ЕС ЭВМ получалось издевательски точно 1.25120000000000E+000!

Наконец, я догадался, что процессор просто дополняет более

длинную мантиссу нулями. Такое «круглое» в двоичном виде число выглядит совсем некруглым в обычном виде. Наоборот, ЕС ЭВМ находило такое «некруглое» двоичное значение, что в обычном виде оно представлялось расширенное нулями, которые в данном случае интуитивно и ждет программист от преобразования.

Формально процессор был прав: погрешность расширенного значения не превышала половины величины младшего разряда. Но убедить в этом заказчика не получилось. Он тыкал мне в нос распечаткой и говорил «так это ж просто другой результат получается».

Тогда я вставил в транслятор преобразование через текстовую строку: число 4 байта преобразуется в текст, текст дописывается нулями и опять преобразуется, но уже в 8 байт. Стало медленнее, зато строго совпадать с ЕС ЭВМ.

Через несколько лет один мой коллега пожаловался, что у него расчет на PL/1 идет раза в три медленнее, чем аналогичный расчет, но реализованный на языке Си. Я был уязвлен. Неужели вызов служебных процедур в моем трансляторе вместо прямых команд так сильно снижает эффективность? Но действительность была проще. Оказалось, что из-за описки в программе несколько переменных вместо атрибута FLOAT DECIMAL(16) т.е. того, что здесь я называю ВЕЩЕСТВЕННОЕ(53), получили атрибут FLOAT DECIMAL(6), а это как раз 4 байта или ВЕЩЕСТВЕННОЕ(24). В результате постоянно включался механизм того самого преобразования с нулями через строку. Стоило исправить описку, как скорость расчета стала, мягко говоря, не хуже, чем в других системах программирования.

После этого, я даже ввел параметр трансляции для предупреждений, что генерируются «медленные» преобразования, которые, кстати, вообще можно исключить, если не пользоваться одновременно двумя форматами представлений вещественных чисел.

Объектно-ориентированный стиль программирования, увы, не способствует повышению эффективности. Например, для каждого вызова процедуры (извините, метода класса) нужно сначала достать из памяти адрес этого метода и только потом вызывать. Это хорошо видно на примере кода ядра Windows. Там часто используемые вызовы оптимизирующий транслятор пытается удерживать в регистрах, чтобы поменьше обращаться к памяти, но получается такое удерживание неважно.

Самым большим резервом эффективности, как ни странно, является сам программист. Но для этого, на мой взгляд, ему нужно изучать команды ЭВМ, хотя языки программирования и должны были бы от этого избавить. Программист, совершенно не представляющий, что собственно делает ЭВМ, когда выполняется его программа, будет бороться за эффективность, исходя из своих (часто неверных) представлений или во всем полагаться на транслятор. А транслятор не

понимает конечной цели задачи, реализованной в программе, и может лишь локально улучшить то, что написал программист.

Мое мнение: для возможного повышения эффективности транслятор, по крайней мере, ничего не должен скрывать от программиста. В данном трансляторе есть параметр, который заказывает выдачу получающейся программы вместе с командами ЭВМ. Всегда можно разобраться, что и как будет делать программа, хотя бы на уровне манипулирования служебными процедурами. Конечно, не требуется всегда анализировать, какие команды получаются, но при поиске путей повышения эффективности программы, лучше ясно представлять, на что уходит время работы программы.

Именно по распечатке созданных транслятором команд программы, мне удалось понять причину трехкратного замедления, о которой я уже говорил. Конечно, я бы не смог быстро разобраться в чужом алгоритме, да еще в виде команд. Я просто обратил внимание на то, что в программе полным-полно функций преобразований, что называется «туда-сюда», бессмысленных, не вытекающих из самой логики задачи.

Все, что я здесь говорил об эффективности, можно было бы отнести к любому языку. А как насчет эффективности именно в PL/1? Здесь я хочу обратить внимание вот на какую вещь: процессор Intel 8086, архитектура которого во многом сохранилась и в современных процессорах, разрабатывался в середине 70-х (выпущен в 1978 году). Похоже, что когда его разработчики решали задачу поддержки языков высокого уровня, они держали в уме именно PL/1, который тогда широко использовался. Доказательства? Пожалуйста.

Работа с текстовыми строками в PL/1? Да это же «цепочечные» команды процессора. Например, единственной командой CMPSB можно сравнить две текстовые строки (не на равенство, а сразу на больше-меньше!) при сортировке.

Использование меток-переменных? Да это же команды «косвенных» переходов в процессоре.

Точные экономические расчеты? Это все команды двоично-десятичной арифметики.

Работа с битовыми строками? Все логические команды процессора работают с последовательностями бит.

Да что там говорить! В каком-то описании процессора я прочитал, что команда «перекодировки» XLAT оказывается, прямо предназначена «для реализации встроенной функции TRANSLATE»!

Поэтому транслятор с PL/1 вовсе не мучается, отображая понятия языка PL/1 в понятия процессора архитектуры IA-32. Конечно, и, например, транслятор с Си ничуть не затрудняется при таком отображении. Но, как бы это сказать, в PL/1 это происходит на ступень выше, что ли. Т.е. даже для объектов языка сложнее, чем указатель или целое число продолжает сохраняться отображение в команды

процессора. Это вовсе не означает, что PL/1 на ступень эффективнее языка Си. Это означает, что в PL/1 на ступень выше, чем в Си, эффективность продолжает не теряться.

43. Заключительная часть

Пожалуй, пора закругляться. Можно наговорить и еще на несколько таких книжек, но боюсь, любопытно это будет лишь читателям, которые заходят на форумы, посвященные трансляторам. На протяжении всех глав я пытался доказать, почему PL/1 должен здравствовать и почему он не «монстр». Не очень получилось? Попробую усилить.

Скажите, можете ли Вы, взглянув на это число-«монстр», повторить его уже не глядя: 169144121100816449362516941. Говорите, что Вам делать больше нечего? А я могу. Потому что это шутка, это всего лишь квадраты чисел от 13 до 1. Когда понятно правило, сразу же исчезает «монстровость», необходимость держать все в голове. Так же и с языком. Простое и регулярное правило построения элементов языка позволяет не задавать вопросы, типа: а как надо писать в операторе открытия файла? Сначала ФАЙЛ, потом ИНДЕКСНЫЙ или наоборот? Да как угодно, транслятору важно лишь слово ОТКРЫТЬ в начале и точка с запятой в конце!

Почему еще PL/1 называют «монстром» или, как говорят молодые, «жутким извратом»? Ах, да! Потому, что в него «понапихали всего, что можно». Ну, насчет «понапихали всего», это не по адресу. Это, скорее, к Windows. Но если серьезней, то представьте, что Вы строитель, которого учили строить из кирпичей. Вам показали разные типы кирпичей и разные виды кладки. Но задание Вы получили: построить пешеходный мост через ущелье. Если Вы будете строить этот мост из кирпичей, у Вас получится что-то вроде римского виадука. А блестящее решение в виде подвесного моста из четырех тросов и дощечек, оказывается, видите ли, теоретически «не чисто». Надо ведь все строить из кирпичей! Да кто сказал, что все надо обязательно строить из кирпичей? (Например, обязательно всегда создавать объекты.) Потому, что так легче учить строителей? У программиста должны быть и «кирпичи» и «тросы».

Было бы замечательно, если бы все реальные задачи можно было бы решать одной простой возможностью языка. Увы, так получается только на машине Тьюринга и то умозрительно. Не приступаете же Вы к любому ремонту, имея лишь молоток и гвозди. Для разных работ требуется и дрель, и рубанок, и Вы же не ругаетесь, глядя на набор инструментов: «понапихали тут». Да, можно в язык не вводить понятие строки, а считать их массивами символов. Только в этом случае простое присвоение строки строке становится похожим на строительство того самого моста из кирпичей.

Наличие в языке совершенно разных, непохожих друг на друга средств, есть отражение объективной реальности существования

совершенно разных задач. В реальности программист может никогда и не сталкиваться с каким-то классом задач и никогда не использовать (и поэтому даже не знать) какое-то средство в языке. Например, человеку, занимающемуся экономическими расчетами с пресловутым дебетом-кредитом, совершенно ни холодно, ни жарко оттого, что в языке есть и комплексные числа и гиперболические функции. Скорее всего, он и не представляет, что это такое. А когда читал описание языка, пожал плечами и пропустил. И наоборот, программист, реализующий баллистические расчеты, обычно и не подозревает, что печать в языке оказывается можно организовать с автоматической расстановкой значков DB и CR. Поскольку ему это не нужно.

Вообще-то на свете есть защитники PL/1 и покруче меня. В нашей стране (скорее, в СССР) это Н.Н. Безруков, который одно время вел любопытный журнал «Софтпанорама». А на Западе еще в 1994 году некий Эберхард Штурм выступил на конференции в Вене (так сказать, в «осином гнезде» сторонников PL/1, так как там есть отделение IBM) с небольшим докладом, озаглавленным «Power vs. Adventure – PL/I and C». Интересный доклад (он доступен в Интернете). Его можно было бы считать заказной статьей IBM, чтобы облить грязью конкурирующие языки, если бы в качестве примеров не приводились реальные «грабли», на которые, несомненно, не раз наступали неудачливые программисты. «Adventure» в заголовке – это имелись в виду приключения на свою … хм, голову, которые могут получить программисты при использовании языка Си.

Но ведь с тех пор прошло столько лет! В современных языках уже должно быть все по-другому! Да как Вам сказать. Вот, например, «усовершенствованный Си» или C++. В чем-то, типа правил «видимости», это шаг от Си к PL/1 или даже к заветам Алгола. У языка C# массивы больше чем у Си похожи на массивы PL/1. А встроенный текстовый ввод-вывод в Яве сразу мне напомнил возможности ввода-вывода в PL/1, которым уже 45 лет. PL/1 устарел? Да не больше, чем логарифмы или полиномы Чебышева. У него нет классов и методов? У него есть модули, структуры и переменные типа «процедура». Нельзя «наследовать» классы? А почему именно «наследование» всегда лучший способ создания объектов? Например, если в программе я рассчитываю орбиты сразу 24 спутников, почему я должен размножать структуру одного спутника как кроликов? Почему сразу не завести массив структур? И для понимания, и для эффективности реализации это проще.

Когда Пентагон решил создать язык «на все времена» (язык Ада), в качестве базы для разработки было предложено три языка: Паскаль, Алгол-68 (вот это действительно «монстр»!) и, конечно же, язык PL/1. В конце концов, делали на основе Паскаля. Но поклонникам этого языка нечего особо радоваться: Ада не смогла потеснить другие языки. Кроме этого, критики проекта Ады прямо заявляли, что если бы в задании на

язык цели формулировались не в терминах Паскаля, то и язык был бы лучше и основой его был бы не Паскаль. Наверное, они намекали на то, что заказчики-генералы ничего, кроме первых 10 страниц описания Паскаля, осилить не смогли.

Чувствую, что моя полемика, как и во многих других книгах на такие темы, начала сводится к формуле: вы все козлы, а я Д'Артаньян. Мне бы очень этого не хотелось. Хочется не уязвить читателя, а поделиться с ним своим пониманием вещей и опытом. В конце концов, объективным критерием «нормальности» программных средств, в том числе и языков программирования, является, прежде всего, наличие успешно реализованных больших проектов. У Вас есть успешные проекты? Ну так и Вы Д'Артаньян.

Я отношусь к, увы, уже старшему поколению программистов. Наверное, и Вы это заметили по датам и упорному желанию автора компьютеры называть ЭВМ. Мне нравится это слово. Кстати, компьютерами когда-то в Америке называли «девочек из бухгалтерии», а ЭВМ'ами в нашей стране всегда называли только ЭВМ.

История языков программирования, особенно бурная после появления «персоналок», прошла и на моих глазах. А в далеком 1979 году я и понятия не имел, что Американский национальный институт стандартов вдруг озаботился новым стандартом языка PL/1. Я думаю, не случайно это почти совпало с началом эры «персоналок». Одной из задач создания нового стандарта (он получил индекс X3.74 и дополнялся в 1981, 1987, 1993, 1998 гг.) являлось упрощение транслятора и переноса его на маленькие ЭВМ. При этом постарались как можно меньше потерять в PL/1. Конечно, я не знаю, но догадываюсь по каким критериям группа стандартизации «подрезала» так называемый «полный» PL/1 (его стандарт имеет индекс X3.53). Например, члены группы могли по очереди вспоминать, какие возможности языка им так и не пригодились в работе. (Шутка.)

Но это и неважно. Важно, что вовремя появился этот новый стандарт, что очень сильный программист Килдэлл этот стандарт успел воплотить в маленький транслятор, а затем мне повезло, и этот транслятор оказался моим собственным инструментом в том смысле, что я могу и изменять его по своему желанию. В общем, «как пожелаем, так и сделаем» любил говорить один персонаж книги «Золотой теленок».

Я искренне считаю, что мне повезло с этим языком. Выбрал я его случайно, даже и выбора-то в тот момент не было. Но язык оказался «человечным», вполне осваиваемым, заботящимся о программисте. И уж если пытаться заложить в школе основы «компьютерной грамотности», то, по-моему, в старших классах надо изучать язык, похожий на этот, т.е. с русскими словами и ясным синтаксисом.

Правда, может показаться, что я так исковеркал транслятор, что это уже и не совсем PL/1. Не согласен. Исправления большей частью касаются или удобных мелочей (типа фигурных скобок и параметра цикла, указанного как комментарий к концу цикла) или являются логичным развитием правил языка (функция ДЛИНА для переменных типа «файл»).

Запаса логичности заложенных в язык идей мне вполне хватило для совершенствования языка без радикальных его изменений. И такое совершенствование вполне естественный процесс. Если бы Вы сопровождали свою программу несколько лет, уж, наверное, Вы бы ее улучшали. Транслятор ничем не отличается от других программ. Считаю, что и язык не должен раз и навсегда застывать как памятник, а должен развиваться вместе с людьми, его использующими, несмотря на печальное «набирание сложности». Например, фирма IBM и сейчас продолжает совершенствовать свои трансляторы. Мои возможности, конечно, скромнее, но и задачи гораздо проще (да и о сбыте транслятора думать не нужно).

Кроме этого, транслятор действительно очень маленький. Вместе с другими утилитами: редактором связи, библиотекарем, транслятором с ассемблера и системной библиотекой, он до сих пор помещается на одной дискете (хотя дискетами уже давно и не пользуются).

Может показаться также, что я всю жизнь только и делал, что исправлял транслятор. Конечно, нет. Если разделить исправления на годы, получится совсем немного в год. И исправления шли не от желания выпендриться, а для исправления ошибок или даже по предложению коллег: «ну ты, это, транслятор курочишь, сделай так, чтобы…». Приходилось задумываться, нужно ли вообще это «чтобы» делать, почему это «чтобы» не было сделано с самого начала и как его сделать. В свою очередь, это заставляло задумываться о принципах языка, о его структуре, о логичности формы и т.п.

Хотя нет, соврал. Одно из первых исправлений было как раз ради выпендрежа. Коллега-программист, работавший на Паскале, глянул в мою распечатку и протянул: «Да ну его, твой ПЛ. Вот у нас в Паскале никаких дурацких CALL перед процедурами писать не надо». На следующий день я принес новую распечатку и гордо заявил: «теперь и в PL/1 писать CALL необязательно!» Коллега почесал затылок и сказал: «а я, к сожалению не могу вот так же заменить все := на =». Кстати, а можно ли действительно в Паскале заменить все «:=» на просто «=»?

Надеюсь, что эта книга помогла Вам что-то узнать, а может быть, лучше понять. Даже, если Вы не согласны с тем, что я утверждал, это все равно помогает лучше понять суть вещей потому, что для опровержения Вы же ищете свои доводы, правда?

На прощание, хотелось бы пожелать Вам успехов и побед в нелегком деле программирования. Как любит говорить один спортивный комментатор: до будущих побед, друзья!

44. Отдельная глава

Эта глава не связана с остальным сюжетом и лишь демонстрирует один из возможных способов разрешения обращения к портам ввода-вывода в Windows (о чем шла речь в главе, посвященной работе на «низком» уровне).

Доработка Windows дедуктивным методом

Элементарно, Ватсон
А.Конан-Дойль «Приключения Шерлока Холмса»

Здесь автор ни в коей мере не пытается пародировать сэра Артура Конан-Дойля, которого чтит. Просто прием подачи любого материала (от анекдотов до литературоведения) в виде диалога Холмса и Ватсона прочно вошел в нашу жизнь и иногда повышает занимательность изложения темы. Итак …

Ненастным вечером Холмс и Ватсон сидели у камина, глядя на огонь.

- Послушайте, Ватсон, мне кажется, что после того как Вы окончили курсы Микрософт с углубленным изучением Windows, Вы стали слишком мрачно смотреть на вещи. Например, сегодня Вы даже не притронулись к своему ноутбуку, который стоит на каминной полке. Вам необходимо отвлечься от грустных мыслей. Мой брат Майкрофт предложил мне решить небольшую задачу, связанную с операционной системой. Он позвонит около десяти, так что у нас есть целых полчаса.

- Что за задача, Холмс?

- Они (Холмс показал глазами куда-то в потолок) хотят соединить обычный ноутбук с неким прибором через параллельный интерфейс. Проблема в том, что когда-то написанная для этого прибора программа теперь в среде Windows-XP работает неверно: она пропускает часть информации.

- Я думаю, Холмс, это происходит из-за того, что в Windows программа не может напрямую обращаться к аппаратуре, а стандартные средства не успевают за этим прибором. Нужно разработать драйвер режима ядра.

- Да, Майкрофт тоже говорил мне что-то о режиме ядра. Но дело в том, доктор, что информация идет так быстро, что единственная возможность не пропустить ее – это полностью остановить работу всех программ и опросом читать параллельный интерфейс. К счастью, это требуется всего лишь на несколько минут, затем обычную работу можно возобновить. Брат предлагает рассмотреть возможность исправления Windows для этой цели.

- Что за странные фантазии Холмс? Исправлять операционную систему ради одной программы?

- Во-первых, Ватсон, результат работы очень важен. Во-вторых, не требуется тиражировать программу на тысячах компьютерах, да и работает она недолго. Наконец сроки. Разработка и отладка драйвера может занять большее время по сравнению с минимальными доработками имеющейся программы. Кроме этого, реализация в виде драйвера, когда все нужно остановить на несколько минут, бессмысленна, ведь никакие интерфейсы и службы Windows в этот момент просто не работают.

- Но исправление Windows - это очень трудная задача, Холмс!

- Я рассчитываю на Ваши знания, доктор. Вы же с удовольствием изучали архитектуру процессора Intel. Давайте применим наш дедуктивный метод. Начнем с главного. Нам требуется, чтобы программа прямо обращалась к портам аппаратуры и умела останавливать и вновь пускать все остальные задачи и процессы. Пока отвлечемся от операционной системы. Подумайте, Ватсон, какими командами процессора это можно сделать?

- Ну, Холмс! Э-э-э… Хотя вот, пожалуйста. В регистре флагов процессора есть флаги привилегий ввода-вывода. Если задать максимальное значение три, то можно обращаться к портам, и… Смешно, как я раньше не подумал. Ведь в этом случае становятся допустимы также и команды CLI/STI, которыми как раз и можно остановить любую другую работу, а потом опять восстановить нормальную работу. Это то, что нам и нужно!

- Вот видите, Ватсон, для решения нам нужно всего лишь установить две единички в регистре.

- Но, Холмс, даже если мы и установим их, Windows сама сломается. Ведь ее обработчики рассчитаны на то, что обращение к портам пользовательским задачам запрещено.

- Вряд ли работа Windows от этого изменится. Все обращения к аппаратуре и так идут в режиме ядра безо всяких запретов. Лучше поразмыслите, Ватсон, каким образом вообще мы могли бы устанавливать их. Прямые команды PUSHFD/POPFD, я думаю, запрещены?

- Запрещены архитектурой процессора, Холмс!

- А средствами Windows?

- Ну, тем более все запрещено. Хотя постойте-ка! В Windows пользователь может установить собственный обработчик исключения через функцию SetUnhandledExceptionFilter. Тогда появляется возможность возобновлять работу задачи с заданными значениями регистров, в том числе и с регистром флагов! Я как раз разрабатываю свой отладчик и это легко попробовать. При выходе из обработчика в задачу, я установлю командой OR FLAGS,3000h максимальное значение привилегий ввода-вывода. Сейчас… Нет, не получается, Холмс. При следующем входе в обработчик значение в регистре флагов опять минимальное.

- Это легко было предсказать, Ватсон. Если бы таким несложным способом можно было бы менять служебные флаги, любой воришка с лондонского рынка мог бы с помощью примитивной программы нарушить работу любого компьютера.

- Что же делать, Холмс?

- Надо найти место в Windows, где гасятся эти флаги, и исправить его.

- Но, Холмс! Как же мы найдем это место за несколько минут? А если и найдем, как же мы сможем вставить туда новые команды. Ведь нельзя же «раздвинуть» имеющийся код.

- А может быть, и не надо раздвигать код. Поставьте себя на место Windows. Какими бы командами Вы бы гасили и устанавливали служебные флаги?

- Ну, я бы погасил ненужные флаги командой AND, а затем установил бы нужные командой OR.

- Правильно, Ватсон, а так как флаги для всех задач одинаковы, вероятно, команды AND и OR использовали бы операнды-константы. Нам просто нужно найти эти константы и исправить одну из них.

- Но где мы будем искать Холмс? Windows состоит из тысячи файлов!

- Ну а курсы Микрософт, доктор! Какой файл отвечает за работу ядра?

- Вы снова правы, Холмс! Единственный файл, где могут быть такие команды, - это файл NTOSKRNL.EXE. Но и этот файл слишком велик для прямого просмотра. Мы будем очень долго искать нужные операнды.

- Давайте рассмотрим регистр флагов, Ватсон. Точнее, его младшую часть. Там находятся все обычные флаги, которые может менять пользователь. Причем флаг IF Windows обязательно должна установить, даже если пользователь сбросит его. Обратите внимание, что флаги не всегда идут подряд, там есть и константы, которые не меняются. Выпишем воображаемую маску разрешенных разрядов для пользователя. Флаг NT и наши искомые флаги IOPL конечно, запрещены. Что получается, Ватсон?

- У меня получился шифр, вроде пляшущих человечков, Холмс

0000110111010111

- Или 0DD7h, доктор! Это вполне редкое сочетание, друг мой. Попробуйте поискать его Вашим любимым отладчиком в файле NTOSKRNL.EXE.

- Есть, Холмс! По адресу F2C5. Сейчас я посмотрю моим отладчиком то же место в виде команд, а не кодов… Это поразительно, Холмс! Все, как Вы и предсказали!

```
F2C4            25D70D3E00    AND  EAX,003E0DD7
F2C9            0D00020000     OR  EAX,00000200
```

Несомненно, это именно то место, где гасятся и устанавливаются флаги. Устанавливается, кстати, действительно один флаг IF.

- Ну вот, Ватсон, задача и близка к разрешению. Вместо кода 02h по адресу F2CB надо поместить код 32h, тогда установится максимальное значение IOPL.

- Но кроме исправления этого кода, Холмс, надо исправить контрольный код самого файла. Мой отладчик умеет считать его процедурой CheckSumMappedFile. Вот, пожалуйста. Вместо кода 2160C9h по адресу 140h нужно будет записать код 2190C9h.

- Не знал этого, Ватсон. Я думаю, для различных версий Windows адреса и контрольные коды будут разными. Но сам принцип остается одинаковым.

- Но ведь это ужасно, Холмс! Надежность всей операционной системы так сильно зависит всего от двух битов… А что, если главари преступного мира Лондона…

- К сожалению, дорогой доктор, главари преступного мира используют более ужасные вещи: человеческую глупость, жадность и доверчивость. Зачем им менять биты, если можно рассылать по электронной почте письма с уверениями, что вы выиграли миллион, надо только отослать недостающие десять тысяч по указанному адресу… Но вот и звонок! Да, Майк! Нам с Ватсоном потребовалось для решения этой задачи чуть более двадцати минут. Ага, ты тоже знаешь, что нужно установить флаги IOPL. А как ты об этом догадался? А, тебе по телефону объяснил Чарльз Симони, понятно. Но у нас-то с Ватсоном не было возможности выслушать его объяснения. Тебе потребуется изменить в файле NTOSKRNL.EXE байт 02 на байт 32 по адресу F2CB и исправить код по адресу 140 с 2160C9 на 2190C9. Потребуется также искусственно создать исключение в программе, и после возврата из него, доступ к портам, я уверен, будет разрешен. Желаю успеха, Майк!

- Но как же он исправит файл, Холмс? При загрузке компьютера этот файл недоступен даже для администратора!

- Элементарно, Ватсон. Вы же знаете предусмотрительность моего брата. На каждом компьютере у него два одинаковых диска с Windows. Если запустить ее с одного диска, служебные файлы на втором становятся доступными и наоборот.

Вот видите, друг мой, дедуктивный метод опять дал положительный результат, а компьютерный мир при внимательном рассмотрении не кажется таким уж сложным и запутанным.

Список русских эквивалентов ключевых слов

```
НЕТ                    = '0'B1
ДА                     = '1'B1

АДРЕС                  = ADDR
ДАТЬ_ПАМЯТЬ            = ALLOCATE
ВРЕМЕННОЕ              = AUTOMATIC
ОСНОВА                 = BASED
БЛОК                   = BEGIN
ДВОИЧНОЕ               = BINARY
БИТ                    = BIT
БУЛЕВА_АЛГЕБРА         = BOOL
РОДНАЯ                 = BUILTIN
ЭТО                    = BY
С_ШАГОМ                = BY
ВЫЗВАТЬ                = CALL
ЦЕЛОЕ_НЕ_МЕНЬШЕ        = CEIL
ТЕКСТ                  = CHARACTER
ЗАКРЫТЬ                = CLOSE
СТОЛБЕЦ                = COLUMN
КОМПЛЕКСНОЕ            = COMPLEX
ОПЯТЬ                  = CONTINUE
С_ИМЕНАМИ              = DATA
ДАТА                   = DATE
ДАТА_4Г                = DATE4Y
ОПС                    = DCL
ДЕСЯТИЧНОЕ             = DECIMAL
ОПИСАНИЕ               = DECLARE
НА_МЕСТЕ               = DEFINED
ЗАСНУТЬ                = DELAY
РАЗМЕРНОСТЬ            = DIMENSION
ПРЯМОЙ                 = DIRECT
ЦИКЛ                   = DO
В_ФОРМЕ                = EDIT
ИНАЧЕ                  = ELSE
КОНЕЦ                  = END
КОНЕЦ_ФАЙЛА            = ENDFILE
КОНЕЦ_СТРАНИЦЫ         = ENDPAGE
ДЛЯ_ВЫЗОВА             = ENTRY
ДЛЯ_ОС                 = ENVIRONMENT
ОШИБКА                 = ERROR
ОБЩЕЕ                  = EXTERNAL
ФАЙЛ,ИЗ_ФАЙЛА,В_ФАЙЛ   = FILE
ТОЧНОЕ                 = FIXED
ЧИСЛО_НЕ_ПРЕДСТАВИМО   = FIXEDOVERFLOW
ВЕЩ, ВЕЩЕСТВЕННОЕ      = FLOAT
ЦЕЛОЕ_НЕ_БОЛЬШЕ        = FLOOR
ВВЕСТИ_ФОРМАТ          = FORMAT
ВЕРНУТЬ_ПАМЯТЬ         = FREE
ИЗ                     = FROM
ЧИТАТЬ                 = GET
ИДТИ                   = GOTO
ВЕРХ_ГРАНИЦА           = HBOUND
ЕСЛИ                   = IF
ЧУЖАЯ                  = IMPORT
ИСКАТЬ                 = INDEX
ЗАДАТЬ                 = INITIAL
ДЛЯ_ВВОДА              = INPUT
МЕСТНОЕ                = INTERNAL
В_ПЕРЕМ                = INTO
ИНДЕКС                 = KEY
ИНДЕКСНЫЙ              = KEYED
ИЗ_ИНДЕКСА             = KEYFROM
ДЛЯ_ИНДЕКСА            = KEYTO
МЕТКА                  = LABEL
НИЖ_ГРАНИЦА            = LBOUND
ХВАТИТ                 = LEAVE

ДЛИНА                  = LENGTH
КАК                    = LIKE
ПЕРЕВОД_СТРОКИ         = LINE
ПЕЧАТАЕТСЯ_СТРОКА      = LINENO
С_ДЛИНОЙ_СТРОКИ        = LINESIZE
В_ВИДЕ                 = LIST
ГЛАВНАЯ                = MAIN
ПУСТО                  = NULL
ПУСТОЙ_УКАЗ            = NULL_PTR
КОГДА                  = ON
КОГДА_КОД              = ONCODE
КОГДА_ФАЙЛ             = ONFILE
КОГДА_ИНДЕКС           = ONKEY
ОТКРЫТЬ                = OPEN
ДЛЯ_ВЫВОДА             = OUTPUT
ЧИСЛО_ВЕЛИКО           = OVERFLOW
ПЕРЕВОД_СТРАНИЦЫ       = PAGE
ПЕЧАТАЕТСЯ_СТРАНИЦА     = PAGENO
С_ДЛИНОЙ_СТРАНИЦЫ      = PAGESIZE
УКАЗАТЕЛЬ              = POINTER
ПЕЧАТНЫЙ, ДЛЯ_ПЕЧАТИ   = PRINT
ПРОЦ                   = PROC
ПРОЦЕДУРА              = PROCEDURE
УКАЗ                   = PTR
ПИСАТЬ, ПЕЧАТАТЬ       = PUT
ВВОД                   = READ
НЕ_ТЕКСТОВЫЙ           = RECORD
РЕКУРСИВНАЯ            = RECURSIVE
ПОВТОРЯЯ               = REPEAT
ЗАМЕНИТЬ               = REPLACE
ВОЗВРАТ                = RETURN
ВОЗВРАЩАЕТ             = RETURNS
ОТМЕНИТЬ               = REVERT
ПЕРЕЗАПИСАТЬ           = REWRITE
ОКРУГЛИТЬ              = ROUND
ПООЧЕРЕДНЫЙ            = SEQUENTIAL
В_УКАЗ                 = SET
СИГНАЛ                 = SIGNAL
С_НОВОЙ                = SKIP
СО_СТЕКОМ              = STACK
ПОСТОЯННОЕ             = STATIC
СТОП                   = STOP
ТЕКСТОВЫЙ              = STREAM
В_СТРОКУ, ИЗ_СТРОКИ    = STRING
ПОДСТРОКА              = SUBSTR
СТД_ВВОД               = SYSIN
СТД_ВЫВОД              = SYSPRINT
ТАБУЛЯЦИЯ              = TAB
ТОГДА                  = THEN
ВРЕМЯ                  = TIME
ПО_ИМЕНИ               = TITLE
ДО                     = TO
ПЕРЕВЕСТИ              = TRANSLATE
ОЧИСТИТЬ               = TRIM
ЦЕЛАЯ_ЧАСТЬ            = TRUNC
НЕТ_ФАЙЛА              = UNDEFINEDFILE
ЧИСЛО_МАЛО             = UNDERFLOW
МАШ_КОД                = UNSPEC
ДЛЯ_ИЗМЕНЕНИЙ          = UPDATE
РД                     = VAR
КОСВЕННОЕ              = VARYABLE
РАЗНОЙ_ДЛИНЫ           = VARYING
ИСКАТЬ_НЕСОВПАДЕНИЕ    = VERIFY
ПОКА                   = WHILE
ВЫВОД                  = WRITE
ДЕЛЕНИЕ_НА_0           = ZERODIVIDE
```

Литература

За много лет у меня накопилось такое количество книг по PL/1 (правда, давно изданных), что из них можно было бы создать музей языка. Я не вижу смысла приводить их полный список, тем более что многие дублируют друг друга.

Вот одна из первых:

«ПЛ/1 для программистов» Р.Скотт, Н.Сондак, Москва «Статистика», 1977.

Вполне удачный учебник, когда-то это был курс лекций. Непонятно только название «для программистов». А для кого же еще?

В нашей стране наиболее полным справочником, наверное, следует считать:

«Практический курс программирования на языке PL/1» Г.Д. Фролов, В.Ю. Олюнин, Москва «Наука», 1987.

Как и положено книге такого солидного издательства здесь все полно и последовательно, хотя многие вещи ушли вместе с ЕС ЭВМ.

Очень полезная книга:

«Ошибки программирования и приемы работы на языке ПЛ/1» Л.Ф. Штернберг, Москва «Машиностроение», 1993.

На мой взгляд, это лучшая книга по использованию PL/1, и вообще по разбору ошибок программирования. Прекрасная и, главное, конструктивная критика PL/1. Многие исправления в трансляторе я сделал под ее влиянием и не раз вступал в заочную полемику с автором. Жаль только, что книга опирается на транслятор для ЕС ЭВМ.

Остальная литература очень специфична, но вполне доступна в Интернете:

G.U.I.D.E. & SHARE Europe Joint Conference (10-13 October 1994, Vienna, Austria) «Power vs. Adventure – PL/I and C». Eberchard Sturm. Műnster, Germany.

American National Standard: programming language PL/I. New York, NY, American National Standards Institute, 1979 (Rev 1998); 403p. ANSI Standard X3.53-1976

American National Standard: information systems - programming language - PL/I general-purpose subset. New York, NY, American National Standards Institute, 1987; 449p. ANSI Standard X3.74-1987