Никлаус Вирт

Краткая история Modula и Lilith

Niklaus Wirth (1994) A Brief History of Modula and Lilith // JMLC'94 Conference, Ulm.
Р. Богатырев, перевод с англ.

Введение

Для того чтобы можно было взяться за серьезный технический проект, необходимо иметь две предпосылки. Должна быть очевидная потребность в решении конкретной проблемы и должна существовать идея, которая ведет к решению этой проблемы. Такие условия выполнились во время профессорского отпуска, который автор имел счастье провести в Исследовательском центре фирмы Xerox в Пало-Альто (Xerox Palo Alto Research Center — PARC) в 1977 году. Проблемой, о которой я говорил, была потребность в языке структурного программирования, предназначенного для построения больших и сложных систем, а решением, или даже точнее — необходимой особенностью языка, была конструкция для выражения единиц отдельно откомпилированных системных компонент, известных как модули.

Мне довелось вплотную познакомиться с разработанным в PARC языком Mesa [1], который имел такую конструкцию. Я чувствовал, что изящный потомок Паскаля мог бы при новой попытке следовать духу своего предка, если его создавать как крайне экономный язык [2, 3]. Его имя Modula (здесь и далее Вирт использует имена Modula и Modula-2 для обозначения одного и того же языка, а именно, Modula-2; это может ввести в заблуждение, поскольку в 1975 году Вирт разработал другой, экспериментальный язык с именем Modula; не называя его имени, о нем он упоминает лишь вскользь, в разделе о параллельности) свидетельствует о доминировании в наших взглядах концепции модуля, и имя это было выбрано — возможно, к сожалению — не Pascal++. (Здесь Вирт улыбнулся, и аудитория разразилась дружным смехом. — Прим. переводчика.)

Потребность в структурном языке с механизмом модулей в программистском сообществе признавалась куда слабее, чем среди нас. Объяснение этого требует некоторого отступления. Вычислительные средства, доступные в 1977 году, состояли преимущественно из огромных мэйнфреймов с изощренными системами разделения времени (time-sharing), доступ к которым осуществлялся через терминалы и удаленные спутники. Революционная идея мощной персональной рабочей станции — компьютер Alto, разработанный в PARC — явилась для меня настоящим откровением [4]. Я тут же проникся мыслью, что нет смысла продолжать разработку программного обеспечения, если оно не опирается и не ориентировано на эту новую вычислительную среду. Однако такие устройства не были доступны на рынке и оставался только один путь — а именно, проектирование и создание их собственными силами. Итак, еще раз, была осознана потребность и найдена идея ее решения. В результате проекта была построена рабочая станция Lilith [5, 6].

Нет смысла в создании новой аппаратуры без нового программного обеспечения. Базовая операционная система, утилиты и первые приложения нужно было создавать одновременно, а следовательно, нам требовался язык программирования и его компилятор. И в самом деле, определяющим стимулом проектирования Modula-2 была потребность в простом универсальном языке, способном выражать весь диапазон программ, необходимых для превращения Lilith в мощное инструментальное средство. Четкой целью было использовать один и тот же язык для всего программного обеспечения Lilith. Очевидно, что Modula и Lilith создавались в связке и было бы неразумно отделять историю одного от истории другого.

Хотя предварительный документ, отражающий определенные цели и концепции языка, был написан в 1977 году, само эффективное проектирование языка началось лишь в период 1978-79 годов. Одновременно был запущен проект реализации компилятора. Из доступного оборудования был один компьютер DEC PDP-11 с 64K памяти. Мы не могли применить однопроходную стратегию наших Паскаль-компиляторов; многопроходная схема была неизбежна в силу

ограничений памяти. Именно PARC-реализация языка Mesa и убедила меня в том, что ранее казалось мне совершенно неосуществимым, а именно, что можно построить законченный компилятор, работающей на небольшом компьютере. Первый Modula-компилятор, написанный ван Ли (K. van Le) в 1977 году, состоял из 7 проходов, каждый из которых генерировал последовательный вывод (промежуточный код), который записывался на диск (2Мбайт). Их количество было сокращено до 5 проходов Эмманом (U.Ammann) при втором проектировании в 1979 году. На первом проходе лексический анализатор (scanner) порождал строку лексем и хеш-таблицу идентификаторов. Второй проход (парсер) осуществлял синтаксический анализ, а третий решал задачу проверки типов. Четвертый и пятый проходы занимались кодогенерацией. Компилятор заработал в начале 1979 года.

Lilith

К этому моменту только-только стал подавать признаки жизни первый макет Lilith, который разрабатывался Ричардом Ораном (R.Ohran) совместно с автором этих строк. Первичной целью проектирования была архитектура, которая наиболее оптимальным образом подходила бы для интерпретации M-кода, Modula-эквивалента соответствующего P-кода Паскаля. Нужно помнить, что эра безграничной памяти наступила лишь 10 лет спустя. А потому высокая плотность кода рассматривалась как фактор первостепенной важности для реализации сложных систем на маленьких рабочих станциях.

Lilith

1978-80	16-разрядная рабочая станция 64 Кслов АЛУ: 4 Am2901 bit-slices, 7 МГц микропрограммируемый интерпретатор M-код высокая компактность кода стек выражений, поля под короткие адреса barrel shifter экран 768*592, затем 704*928 пикселов мышь 10 Мб картридж-диск
1980 декабрь	20 компьютеров Lilith в ETH
1982	Ethernet (3 Мбита/сек) лазерный принтер (Canon LBP-10) и сервер
1983	файловый сервер (Fujitsu Eagle), 500 Мбайт

Lilith была организована как компьютер, ориентированный на работу с 16-разрядными словами, M-код был ориентирован на поток байтов. Размер памяти был 2^{16} слов (128 Кб). Макет был построен из микросхем 4К*1 бит.

Лучшим способом получения высокой компактности кода было определить довольно широкое разнообразие инструкций, среди которых были и довольно сложные, и построить аппаратуру в виде интерпретатора микрокода. Каждая инструкция M-кода состояла из одного или нескольких байтов. Первым байтом был код операции, в зависимости от которого выбирались для интерпретации последующие инструкции микрокода. Преимущества этой схемы были самые разнообразные.

1. Реальное оборудование могло значительно упроститься и следовательно работать быстрее. Частота системных часов была 7 МГц, что приводило к циклу в 140 наносекунд. Функциями арифметико-логического устройства были сложение и логические операции.
2. Простые инструкции M-кода можно было реализовать в виде очень коротких последовательностей, состоящих всего лишь из 2-3 микроинструкций. На их выполнение требовалось 280-420 наносекунд.

3. Более сложные инструкции, такие как умножение и деление, могли быть реализованы более длинными последовательностями микроинструкций без использования вспомогательного оборудования (19 инструкций для умножения и 36 — для деления).
4. Одна и та же операция могла быть представлена с различными параметрами (адресами или значениями) длиной 1, 2 или 4 байта, опять-таки без усложнения аппаратного обеспечения.

Достижению высокой компактности кода, который, как оказалось, превзошел коммерческие процессоры в несколько раз (M68000 — 2,5 раза; i8086 — в 3,5 раза), способствовали следующие два фактора.

1. Использование разных длин адресов и операндов (4, 8, 16 и 32 бита). Оказалось, что 70% всех параметров инструкций имело значения в диапазоне от 0 до 15. В этом случае операнд упаковывался в одном байте с кодом операции.
2. В Lilith был организован стек для промежуточных результатов, возникающих в ходе вычисления выражений. Этот стек был реализован как 16-словная быстрая SRAM-память. M-код, таким образом, не содержал номеров регистров, так как они покрывались схемой стека.

Наиболее заметной новацией, которую принесла с собой рабочая станция, был растровый дисплей с высоким разрешением. Он очень сильно контрастировал с обычными дисплеями размером 24 строки по 80 символов. Вместе с мышью он открыл новую эру интерактивного человеко-машинного интерфейса, основанного на окнах, курсорах, планках скроллинга и заголовков, пиктограмм и меню. Растровый дисплей со своими данными, хранящимися в основной памяти, требовал специальных средств для генерации видеосигнала с достаточно высокой частотой. Lilith была оборудована дисплеем 768*592 пиксела, который в более поздних версиях приобрел вертикальную форму с размером 704*928 пикселов; 50 перекрывающихся (interlaced) полукадров требовали частоты около 30 МГц.

Схема инструкций микрокода доказала, что она наиболее удобна для реализации небольшого набора растровых операций рисования линий, изображения символов, пересылки и дублирования блоков пикселов. Эти операции были в явном виде представлены в отдельных операциях M-кода. Соответствующие микропрограммы выполняли растровые операции с потрясающей эффективностью. Что касается оборудования, то одна только специфическая поддержка растровых операций явилась заметным достижением. Она позволяла осуществлять произвольный сдвиг за один микроцикл. Ни один коммерческий микропроцессор не предоставлял тогда такого механизма.

В то же самое время Лео Гайсманом (L.Geissmann) и Кристианом Якоби (Ch.Jacobi) на основе компилятора для PDP-11 был разработан новый Modula-компилятор, который учел все достоинства архитектуры Lilith. Он состоял из четырех проходов, причем архитектура Lilith резко упростила кодогенерацию. Разработка велась на PDP-11 с последующим переносом на Lilith. Аналогичным образом реализовывались все части операционной системы Medos. Этим занимался Свен Эрик Кнудсен (S.Knudsen). Medos в сущности шла по стопам систем пакетной обработки с загрузкой и выполнение команд, вводимых с клавиатуры. В параллель с этими работами Кристиан Якоби реализовал программную поддержку дисплея и окон. Она служила основой первых прикладных программ, таких как текстовый редактор, который интенсивно использовался для написания наших программ. Этот редактор опирался на ныне хорошо известные методы интерфейса "курсор/мышь" и раскрывающиеся меню.

Modula-2 вошла в повседневное использование и быстро доказала свою ценность как мощный эффективно реализованный язык. В декабре 1980 года первая пилотная серия из 20 Lilith была выпущена в штате Юта под непосредственным контролем Ричарда Орана. Затем эти машины прибыли в Швейцарский федеральный технологический институт в Цюрихе (ETH). Дальнейшая разработка программного обеспечения уже велась с более чем 20-кратным увеличением аппаратной мощи, появившейся в нашем распоряжении. Итак, была успешно воплощена в жизнь среда истинно персональной рабочей станции.

Modula

Из предшествующих слов должно быть ясно, при каких условиях могла появиться Modula-2. Очевидно, что нашим первичным делом был не язык, удовлетворяющий всем желаниям всех программистов, а создание достаточно мощного инструмента для реализации всей программной системы в кратчайшие сроки с ограниченными людскими ресурсами. Modula-2 была, таким образом, подцелью, средством для завершения проекта.

Modula-2	
1977-79	проектирование языка
1977	первый компилятор, 7-проходный (!) для PDP-11,
1979	второй компилятор, 5-проходный
1980	перенос компилятора на Lilith 4-проходный компилятор
1984	1-проходный компилятор

Модули и раздельная компиляция

Решение перейти от Паскаля к его потомку было в основном связано с высокой важностью, которую имела ясная формулировка понятия модуля и реализации его как средства для раздельной компиляции. Поэтому решение, принятое в некоторых коммерческих расширениях Паскаля, а именно, слияние программных единиц (units) на уровне исходного текста, было нами отвергнуто. Понятие модуля с явными списками импортируемых и экспортируемых идентификаторов было взято из языка Mesa. Из его реализации также была заимствована идея символьного файла (symbol file) как специальной информации для компилятора об объектах, экспортируемых соответствующим модулем. Концепция модуля породила новый взгляд на операционную систему для Lilith: традиционное разбиение на систему и прикладные программы было заменено единственной иерахией модулей, которая могла расширяться или сокращаться по мере того, как это требовалось разработчику.

Идея модуля в Modula-2 состояла в том, что синтаксическая единица отделялась стеной, через которую нельзя было увидеть ее идентификаторы, кроме тех, что присутствовали в списках экспорта и импорта.

Поскольку синтаксические конструкции обычно определяются рекурсивно, то модули поддерживают вложенность с очевидным сохранением правил видимости идентификаторов. Внутренний, или локальный, модуль показал на практике свою весьма ограниченную пользу, и потому применялся редко. Если учитывать, что реализации должны обрабатывать глобальные и локальные модули по-разному (глобальные — как отдельно скомпилированные), то локальные модули возможно и не надо было тогда вводить.

Списки импорта могли принимать одну из двух форм. Простейшая форма состояла из списка идентификаторов модулей:

```
IMPORT M0,M1;
```

Объект x, экспортируемый, к примеру, модулем M1, обозначался в клиенте модуля M1 как квалифицированный идентификатор M1.x. Вторая форма списка импорта приводила к сокращению текста программ. Она позволяла опускать имя модуля за счет явного указания источника этого идентификатора:

```
FROM M1 IMPORT x,y,z;
```

Как и большинство сокращений, это было весьма спорное решение. При этом не только усложнялся компилятор, но и возникал негативный и неизбежный побочный эффект: конфликт имен. Если два модуля экспортируют объекты с одинаковым именем, то при импорте по второй форме возникает конфликт имен, даже если спорный идентификатор вообще не используется.

С другой стороны, критики языка Modula-2 высказывали мнение, что наша концепция была чересчур уж проста и ограничена. Их взгляд на модульный интерфейс, т.е. на DEFINITION-часть модуля, касался выборки видимого снаружи набора объектов данного модуля. Modula позволяет определять только одну такую выборку, тогда как в некоторых случаях оказываются полезными различные выборки для различных клиентов. Этот механизм был реализован в языке Cedar [7]. Он привел к дальнейшим расширениям и усложнениям. Очевидно, что тут же появилось желание получать пересечения и объединения таких выборок.

Другим моментом, который возможно имеет смысл упомянуть в этой связи, является обработка экспортируемых типов перечисления. Желание избежать длинного списка экспорта приводит к тому (исключительному) правилу, что экспорт идентификатора перечислимого типа подразумевает экспорт всех идентификаторов констант этого типа. Сколь хорошо это может звучать для приверженца сокращений, к столь же негативным последствиям это может и привести, которые выражаются все в той же форме конфликта имен. Конфликт наступает, когда два импортируемых типа перечисления имеют по крайней мере по одной константе, носящих совпадающие имена. Более того, идентификаторы теперь могут появляться не будучи локально объявленными, не будучи квалифицированы именем модуля и не появляясь в списке импорта — ситуация крайне неожиданная для структурного языка.

Статическая проверка типов

Проектирование Modula-2 следовало бескомпромисному правилу — скорее даже догме — строгой статической типизации. Тем самым, компилятор вменялось в обязанности осуществлять всю проверку целостности типов и избегать всех возможных задержек на этапе выполнения программы. В этом отношении Паскаль содержит несколько ошибок, о которых здесь следует упомянуть. Одна из них состоит в незавершенности спецификации параметров, унаследованной от Algol-60. Это можно видеть на следующем примере:

```
PROCEDURE  P (x,y: INTEGER);
BEGIN  ...  END;

PROCEDURE  Q (p: PROCEDURE);
  VAR a, b: REAL;
BEGIN  ...  p(a+b)  ...  END;

...  Q(p)  ...
```

Здесь вызов p(a+b) не подчиняется спецификациям параметров процедуры P, но этот факт не может быть обнаружен при статическом анализе. Modula исправила этот недостаток за счет требования полной спецификации параметров:

```
PROCEDURE Q(p: PROCEDURE (REAL,REAL));
```

Затем Паскаль разрешает использовать в одном выражении как целые, так и вещественные операнды (смешанные выражения), постулируя автоматическое преобразование числового представления так, как это необходимо.

Наличие большого количества базовых типов значительно усложняет генерацию кода, поэтому было решено не осуществлять автоматическое преобразование. Это, в свою очередь, потребовало, чтобы правила языка запрещали использование смешанных выражений. Это решение действительно спорно, но его можно понять в плане наших непосредственных потребностей, которые в отношении арифметики были весьма скромны.

Тип CARDINAL

Как можно объяснить введение типа CARDINAL — типа натуральных чисел — в виде еще одного базового типа? Потребность эта возникла из желания представлять адреса как числовые типы. Lilith была 16-разрядной машиной с памятью 2^{16} слов, поэтому нельзя было позволить себе такую роскошь, как наличие бесполезного бита знака. На первых порах после ввода этого типа все выглядело благополучно, но затем возникло несколько проблем. Как, например, можно эффективно реализовать сравнение i < c? Вопрос это легко снять, если смешанные выражения запрещены.

Для целочисленной (знаковой) и натуральной (беззнаковой) арифметики требуются два разных набора инструкций. Даже если одна и та же инструкция может использоваться для сложения, все равно существует различие в определении переполнения. Для деления ситуация еще более проблематична. В математике частное q = x DIV y и остаток r = x MOD y определяются отношениями

```
x = q * y + r,  0 <= r < y
```

Целочисленное деление в противоположность вещественному асимметрично в отношении 0. Например,

```
 10 DIV 3 = 3          10 MOD 3 = 1
-10 DIV 3 = 4         -10 MOD 3 = 2
```

Большинство компьютеров, однако, предоставляет инструкции, которые неверны в отношении данного определения. Они трактуют целочисленное деление наподобие вещественного деления, симметричного относительно 0:

```
(-x) DIV y = -(x DIV y)
(-x) MOD y = -(x MOD y)
```

Если целочисленное деление реализуется корректно, то деление на число в степени двойки может быть реализовано более эффективно путем простых сдвигов. В случае неверных инструкций деления это невозможно:

```
x*2^n = left(x,n)
x DIV 2^n = right(x,n)
```

Следующие процессоры помимо прочего осуществляют целочисленное деление вопреки математическому определению: Motorola 680x0, Intel 80x86, Intel 80960, DEC VAX, IBM Power. Хуже того, язык Ada возвел эту ошибку в ранг стандарта.

Дабы не вступать в противоречие с другими, мы решили принять неверную арифметику. Оглядываясь назад, можно с уверенностью сказать, что это было, конечно, серьезной ошибкой. Компилятор использовал инструкции сдвига для умножения и деления чисел в степенях двойки только в случае выражений типа CARDINAL.

В общем и целом введение типа CARDINAL объясняется необходимостью обработки 16-разрядных адресов. С точки зрения математиков (которые изобрели отрицательные числа ради алгебраической полноты) его следовало бы назвать шагом назад, так как он создавал новые сложности для программиста, которых до этого не было. Рассмотрим, к примеру, следующие операторы, выполняющие действие Q для x = N-1, ..., 1, 0:

```
x := N-1;
WHILE x >= 0 DO Q; x := x-1 END
```

Если "x" типа INTEGER, то программа корректна, но если "x" типа CARDINAL, то такая запись приводит к переполнению, которого можно избежать путем корректной альтернативной формулировки

```
x := N;
WHILE x > 0 DO x := x-1; Q END
```

Приход 6 лет спустя 32-разрядных компьютеров дал возможность забыть тип CARDINAL. Этот эпизод показал, насколько неадекватность аппаратуры может заставить разработчика языка идти по пути усложнений и компромиссов, если тот хочет в полной мере использовать доступные аппаратные возможности. Урок состоит еще и в том, что память размером 2^n требует целочисленной арифметики как минимум с (n+1) разрядом.

Процедурные типы

Бесспорной, непосредственной и наиболее существенной новацией явился процедурный тип, также взятый из языка Mesa. В ограниченном виде он был представлен и в Паскале, а именно, в форме параметрических процедур. Следовательно, идею нужно было только обобщить, т.е. сделать применимой к параметрам и переменным. Хотя это средство и редко использовалось в программном обеспечении Lilith — за исключением появившихся впоследствии редакторов и некоторых компонент в операционной системе — оно составляет никак не меньше, чем базис для объектно-ориентированного программирования, с полями записи (объекта), которые хранят процедуры как свои "значения".

И все же Modula-2 никогда не претендовала на поддержку объектно-ориентированного программирования. Единственным упущенным средством был механизм поддержки новых типов данных, опирающихся на другие, т.е. типов, которые снизу вверх совместимы с существующими типами и которые — в объектно-ориентированной терминологии — наследуют свойства существующих типов. Это важное средство было введено в потомке языка Modula-2 — в языке Oberon [8, 9], где оно было соответственно названо расширением типа (type extension) [10].

Хотя Modula-2 не поддерживает объектно-ориентированное программирование, она, по крайней мере, делает его возможным за счет использования типа ADDRESS и того правила, что значение адреса может быть присвоено любой ссылочной переменной. Из этого следует, конечно, что отвергается проверка типов и приносится в жертву надежность, которая стоит очень дорого. Трюк состоит в том, что каждому типу записи, который должен быть расширяемым, отводится дополнительное поле типа ADDRESS. Расширения записи далее требуют выделяемого отдельно второго блока памяти, указатель на который и присваивается этому полю. Нечего и говорить, что дополнительное разыменование при доступе к полям расширения также, как и громоздкая формулировка нарушения системы типов, являются неудовлетворительным решением. Аккуратно встроенная в Oberon концепция расширения типов решила эти проблемы просто великолепно; ценой за такой подход являются проверки целостности типов на этапе выполнения в тех случаях, где этого никак не избежать.

Низкоуровневые средства

Средства, что дают возможность выражать те ситуации, которые не могут уложиться в набор абстракций, предоставляемый языком, называются низкоуровневыми средствами. Их наличие — это симптом неполноты набора предоставляемых абстракций. В то время, когда мы поставили перед собой задачу построить законченную операционную систему для Lilith, они были неизбежны. Тем не менее, я полагаю, что они были введены с наивной верой в то, что программисты будут использовать их только как последний ресурс. В частности, концепция функции преобразования типа (type transfer) была серьезной ошибкой. Она позволяет идентификатор типа использовать в выражении как идентификатор функции: значение T(x) равно "x", причем переменная "x" интерпретируется как принадлежащая типу T. Такая интерпретация существенно зависит от соответствующего бинарного представления типа переменной "x" и типа T. Следовательно, каждая программа, использующая это средство, неизбежно зависит от реализации, что находится в явном противоречии с фундаментальной целью высокоуровневых языков — с понятием абстракции. Ситуация особенно удручающая, поскольку такую зависимость нельзя распознать по одному лишь заголовку модуля.

К той же самой категории неудачных средств относится и вариантная запись. Настоящий камень преткновения — это вариантная запись без поля признака (тэга). Значение поля признака предназначено для индикации той структуры, которая ныне присвоена записи. Если тэг опущен, то нет возможности проверить (или обнаружить) текущий вариант. Это именно тот недостаток, который может проявиться при доступе к полям записи с "неверными" типами:

```
VAR  x: RECORD
          CASE : INTEGER OF
            0: a: INTEGER
          | 1: b: CARDINAL
          | 2: c: BITSET
          END
        END
```

Из присваивания x.a := -16 следуют равенства x.b = 65520 и x.c = 4..15. Или может потом будет x.c = 0..11 ? Безусловно, подобные программы плохо разработаны, трудны в понимании и они пренебрегают концепциями абстракции и переносимости.

После 1982 года к Modula-2 возник интерес сразу в нескольких промышленных организациях. Первый коммерческий компилятор был построен в Logitech S.A., затем последовали и другие [11]. IBM создала компилятор и запрограммировала свою операционную систему для AS/400 на Modula, а Центр системных исследований фирмы Digital (DEC Systems Research Center) принял Modula-2 для своих внутренних проектов, расширив язык до Modula-2+ [12]. Его создатели обнаружили недостатки Modula-2 в трех областях: параллельности, распределении памяти и обработке исключений.

Параллельность

Проект по исследованиям методов и проблем мультипрограммирования велся в нашем институте в 1974-1976 годах. Он привел к появлению небольшого экспериментального языка, реализованного на компьютере PDP-11 [13]. В качестве основы для мультипрограммирования в нем были выбраны мониторы и условия (которые здесь назывались сигналами), причем в том виде, как они были предложены Тони Хоаром (C.A.R.Hoare) [14]. Мониторы обобщались до модулей. Тем самым концепция инкапсуляции (encapsulation) была отделена от концепции взаимного исключения (mutual exclusion). Обработка прерываний была встроена в диспетчеризацию процессов в том смысле, что переключение процессов могло производиться либо с помощью запрограммированных операций работы с сигналами (send, wait), либо с помощью прерываний. Последние рассматривались как внешние сигналы, тем самым, в концептуальном смысле проводилась унификация сигналов и прерываний. Этот проект привел к выводу о том, что не существовало явно предпочтительного набора базовых конструкций для выражения параллельных процессов и их взаимодействий, и что обычные и облегченные процессы (threads), сигналы и семафоры, мониторы и критические интервалы — все они обладали как достоинствами, так и недостатками, в зависимости от конкретных приложений. Следовательно, было решено, что в Modula-2 нужно включить только одно базовое понятие сопрограмм (coroutine), и что верхние уровни абстракции нужно программировать как модули, опирающиеся на сопрограммы. Это решение было даже особенно убедительным, поскольку требования к системному обеспечению Lilith как к однопользовательской операционной системе легко удовлетворялись без применения изощренных средств для мультипрограммирования, что не требовало уделять повышенного внимания всем этим вопросам.

Но вера в то, что сопрограммы смогут также служить подходящей основой для истинных мультипроцессорных реализаций, была ошибочной, что и было показано авторами языка Modula-2+. Мы также оставили мысль о том, что обработка прерываний должна поддерживаться тем же механизмом, что и запрограммированное переключение процессов. Прерывания — суть предмет конкретных условий реального времени. Время отклика в реальном времени выйдет за допустимые пределы, если прерывания будут обрабатываться универсальными и усложненными процедурами диспетчеризации и переключения процессов.

Распределение памяти и сборка мусора

Modula-2, как и Паскаль, имеет указатели и, тем самым, подразумевает динамическое распределение памяти. Выделение памяти переменной x^ выражается внутренней процедурой NEW(x), обычно реализованной через системный вызов. Предпосылка соотоветствующей реализации в Modula-2 заключалась в том, что программы будут обрабатывать пулы динамически порожденных переменных, по одному для каждого типа (записи). Такие индивидуально запрограммированные обработчики могут быть оптимально настроены на специфические условия приложения, тем самым гарантируя наиболее эффективное использование памяти. В свое время это было очень важно из-за небольшого количества доступной памяти компьютера и отсутствия

ее централизованного распределения. Сегодня гигантские объемы памяти делают эту стратегию устаревшей, а гибкое централизованное распределение памяти — незаменимым.

Даже не имея неудачных и мощных средств для нарушения системы типов, Modula-2 по-прежнему удовлетворяет потребностям централизованного распределения памяти, т.е. позволяет выполнять работу на основе глобальной сборки мусора. Однако, сборщик мусора должен опираться на неразрушаемую во время выполнения информацию о типах. Нарушение типизации должно быть невозможным. В действительности Modula-2 — чересчур "мощный" язык. Это напоминает мне старую мудрость, что мощность языка определяется не только тем, что он позволяет выражать, но в одинаковой мере и тем, что он выражать запрещает.

Обработка исключений

Под обработкой исключений (exception handling) мы обычно понимаем передачу управления неявно назначенной последовательности операторов при редком возникновении определенного условия. Что же отличает "обработку исключений" от традиционных операторов if помимо редкого возникновения условия? Здесь нужно иметь ввиду три момента.

1. Последовательность операторов (обработчик исключений) размещается в модуле, отличном от того, где обнаружено (возбуждено) исключение.
2. Несколько работающих процедур должно быть завершено аварийно (прервано).
3. Предполагается, что отделение программного текста, отвечающего за обработку редкого условия, от программы, задающей "нормальное" поведение, способствует улучшению понятности программы.

Только второй момент обращается к механизму, который не может быть выражен традиционными конструкциями. Вопрос в следующем: соответствует ли такой новый механизм важной абстракции, которая действительно упрощает программы, или же это только удобное сокращение для часто возникающих ситуаций? И еще: возможна ли такая реализация, которая не идет во вред эффективности?

На первый вопрос никогда нельзя найти однозначный ответ. Все зависит не только от взглядов того или иного человека и стиля программирования, но также и от соответствующих приложений. Предложение по включению конструкции обработки исключений присутствовало уже в первом проекте (draft) Modula-2, но впоследствии было из него удалено по причине неубедительности. Разработка законченных операционных систем [15] по крайней мере показала, что отказ от обработки исключений не был уж таким ошибочным. Появление систем с языками, где стала доступной обработка исключений, которая, как следствие, использовалась с трудом, а нередко и ошибочно, наводит на мысль о том, что решение отказаться было скорее правильным. Тем не менее, я прекрасно понимаю, что другие люди — не менее опытные в области конструирования систем — не разделяют этого взгляда. Одно, безусловно, остается бесспорным: механизм всегда можно добавить в язык, а вот удалить — никогда. Поэтому рекомендуется начинать с тех средств, без которых обойтись нельзя.

Последующие разработки

В течение 1984 года автор этих строк разработал новый компилятор для Modula-2. Я чувствовал, что многие части компиляции могли бы выполняться проще и более эффективно, если бы тогда можно было использовать доступные ныне объемы памяти (те, в которых мы работали, по сегодняшним меркам крайне скромны). Память Lilith составляла 64 Кслов и ее высокая компактность кода позволяла реализовать однопроходный компилятор. Это подразумевает существенное сокращение операций с диском, которые поглощали наибольшую часть времени компиляции. Действительно, время компиляции самого компилятора сократилось с 4 минут до всего лишь 45 секунд.

Новый компилятор сохранял, однако, разбиение на задачи. Но вместо того, чтобы каждая задача задавала свой проход — с последовательным вводом и выводом на диск — в нем был выделен специальный модуль с наиболее общим процедурным интерфейсом. Общие структуры данных, такие как таблица символов, были определены в отдельном модуле описания данных, который

импортировался (почти) всеми другими модулями. Те модули представляли собой сканер, парсер, кодогенератор и обработчик символьных файлов.

Когда завершается разработка нового компилятора, редко удается удержаться от искушения внести в язык некоторые изменения. Вот и в этом случае были внесены некоторые уточнения и изменения, которые нашли свое отражение в пересмотренном описании языка. Единственное важное изменение было связано с интерфейсными модулями (или даже точнее, с интерфейсной частью модуля). Это изменение постулировало, что все идентификаторы, описанные в интерфейсной части, экспортируются, что делает список экспорта уже ненужным.

Со стороны операционной системы значительным шагом вперед явилась реализация сети. Мы остановились на экспериментальной Ethernet-технологии из PARC, шинной топологии с 3 МГц полосой пропускания и манчестерском кодировании, обеспечивающим скорость передачи на уровне 3 Мбит/сек. Аппаратные интерфейсы и программное обеспечение были разработаны Хоппе (J.Hoppe). Среди сетевого сервиса доминировала передача файлов, поэтому сеть была соединена с файловой системой на нижнем уровне так, что доступ к локальным и удаленным файлам различался лишь по префиксу имени файла.

Наличие сети, соединяющей все рабочие станции, привело к потребности в серверах, т.е. в "неперсональных рабочих станциях". Первым сервером, который заработал в 1982 году, был принтерный сервер, подключенный к лазерному принтеру (Canon LBP-10). Благодаря гибкости и мощности Lilith обычная рабочая станция могла использоваться и для генерации растровых изображений и для управления принтером. Аппаратный интерфейс состоял только из канала прямого доступа к памяти (DMA — direct memory access). В нем было много общего с драйвером дисплея, но он был соответствующим образом адаптирован для приема синхронизирующих импульсов от принтера. Так как память была слишком мала для хранения целой страницы (около 1 Мбита), генерация растра и передача данных на принтер и видео велись параллельно. К счастью выяснилось, что Lilith была достаточно мощной для генерации растровых изображений на лету. Тем самым, Lilith стала первым компьютером в Европе, который полностью использовал возможности лазерной печати, позволяя получать тексты с различными стилями и фонтами и графику, начиная от отсканированных изображений и кончая электронными схемами.

Второй сервер, задействованный в 1983 году, был файловым сервером. В отличие от обычных рабочих станций он использовал не только 10 Мбайт картридж-диски, но и объемные (500 Мбайт) винчестеры (Fujitsu Eagle), которые в то время задавали передовой уровень технологии. Большие объемы требовали иной организации файловой системы. Кроме того, была реализована концепция устойчивой памяти. Эти исследования велись Остлером (F.Ostler).

В период с 1981 по 1985 годы были предприняты значительные усилия в области разработки приложений. Первым в этом ряду стоял редактор документов, позволяющий в полной мере использовать все новации Lilith — от растрового дисплея и мыши до сети и лазерного принтера. Первая попытка Ю. Гуткнехта (J.Gutknecht) и Винигера (W.Winiger) привела к появлению редактора Andra [16]. В нем для представления редактируемого текста применялся метод списка фрагментов (piece list technique), автором которого был Лэмпсон (B.Lampson) из PARC. Также в нем для пользовательского интерфейса использовались окна и раскрывающиеся меню. Редактор позволял использовать множество фонтов и различных режимов форматирования и рассматривал документ как иерархически структурированную единицу. Следом за ним начался проект Гуткнехта и Шера (H.Shaer), в результате которого был создан редактор Lara [17]. Он стал основной прикладной программой для пользователей Lilith на многие, многие годы.

Среди других важных приложений был набор инструментальных средств для интерактивного проектирования фонтов, реализованный Коэном (E.Kohen), а также графический редактор для рисования линий, разработанный автором этих строк. Общей характерной чертой всех этих средств было то, что они вошли в повседневный быт всей нашей команды, обеспечивая обратную связь не только в отношении самих средств, но и в отношении Modula и всей системы в целом.

Lilith и Modula-2 были основными средствами в обучении программированию в Департаменте компьютерных наук ETH (Computer Science Department of ETH). Около 60 компьютеров Lilith использовались с 1982 года, предоставляя современные средства для программирования и для подготовки документов. И это было за пять лет до того, как аналогичные средства стали доступны коммерчески. Компьютеры Lilith были сняты с эксплуатации в 1990 году; к тому моменту они использовались ежедневно на протяжении 10 лет.

Литература

[1] (1976) Mesa Language Reference // Xerox PARC.

[2] Wirth N. (1982) Programming in Modula-2 // Springer-Verlag.

[3] Gutknecht J. (1984) Tutorial on Modula-2 // BYTE, No.8, p.157-176.

[4] Thacker C.P. et al. (1979) Alto: A Personal Computer // Xerox PARC, CSL-79-11.

[5] Wirth N. (1981) Lilith: A Personal Computer for the Software Engineer // Proc. 5th Int'l Conf. on Software Engineering, San Diego // IEEE 81CH1627-9.

[6] Ohran R. (1984) Lilith and Modula-2 // BYTE, No.8, p.181-192.

[7] Teitelman W. (1984) A Tour through Cedar // IEEE Software, Vol.1, No.2, p.44-73.

[8] Wirth N. (1988) The Programming Language Oberon // Software — Practice and Experience, Vol.18, No.7, p.671-690.

[9] Reiser M., Wirth N. (1992) Programming in Oberon // Addison-Wesley.

[10] Wirth N. (1988) Type Extension // ACM Transactions on Programming Languages and Systems, Vol.10, No.2, p.204-214.

[11] Hartel P.H., Starreveld D. (1985) Modula-2 Implementation Overview // Journal of Pascal, Ada, and Modula-2, No.4, p.9-23.

[12] Rovner P. (1986) Extending Modula-2 to Build Large, Integrated Systems // IEEE Software, November, p.46-57.

[13] Wirth N. (1977) Modula: A Language for Modular Multiprogramming // Software — Practice and Experience, Vol.7, p.3-35.

[14] Hoare C.A.R. (1974) Monitors: An Operating Systems Structuring Concept // Communications of the ACM, Vol.17, No.10, p.549-557.

[15] Wirth N., Gutknecht J. (1992) Project Oberon // Addison-Wesley.

[16] Gutknecht J., Winiger W. (1984) Andra: The Document Preparation System of the Personal Workstation Lilith // Software — Practice and Experience, Vol.14, No.1, p.73-100.

[17] Gutknecht J. (1985) Concepts of the Text Editor Lara // Communications of the ACM, Vol.28, No.9.