Никлаус Вирт

От Modula к Oberon

Niklaus Wirth (1990) From Modula to Oberon // Institute for Computer Systems, ETH, Zurich.
Р. Богатырев, перевод с англ.

Язык программирования Oberon является результатом концентрированных усилий по увеличению мощности языка Modula-2 и одновременно по уменьшению его сложности. Несколько средств было удалено, другие же были добавлены, в результате чего возросла выразительная мощность языка и повысилась его гибкость. В данной статье описаны эти изменения и приведены их аргументация. Сам же язык описывается в отдельной работе.

Введение

Язык программирования Oberon родился в результате проекта, целью которого было проектирование современной, гибкой и эффективной операционной системы для однопользовательской рабочей станции. Магистральное направление концентрировалось на тех свойствах, которые наиболее существенны, и, как следствие, мы избегали эфемерных вопросов. Это лучший способ удержать систему в руках и сделать ее понятной, надежной и эффективно реализуемой.

Первоначально планировалось реализовать систему на языке Modula-2 [1], впоследствии названном просто Modula, так как этот язык достаточно эффективно поддерживает концепцию модульного проектирования и поскольку операционная система должна была проектироваться в рамках раздельно компилируемых частей с грамотно подобранными интерфейсами. И в самом деле, операционная система должна быть не более, чем набор базовых модулей, тогда как проектирование приложения должно рассматриваться как проблемно-ориентированное расширение базового набора: программирование — это всегда расширение данной системы.

В то время как современные языки, такие как Modula, поддерживают понятие расширяемости в процедурной области, в области типов данных это понятие еще не прижилось. В частности, Modula не допускает адекватное определение новых типов данных как расширений уже существующих и определенных программистом типов. Потребовался дополнительный механизм, который привел к расширению языка Modula.

Концепция задуманной операционной системы потребовала высокодинамичного централизованного распределения памяти, операющегося на технологию сборки мусора. И хотя Modula в принципе никак не препятствует встраиванию соответствующего сборщика мусора, наличие в языке вариантных записей создает серьезные препятствия. Поскольку новый механизм расширения типов сделал вариантные записи излишней возможностью, то логичным решением было попросту изъять этот ненужный элемент. Такой шаг привел к ограничению (подмножеству) языка Modula.

Кажется весьма очевидным, что правило концентрироваться на существенном и не придавать значения второстепенному должно применяться не только по отношению к проектированию новой операционной системы, но и в равной степени к тому языку, на котором эта система формулируется. Следование этому принципу приводит от языка Modula к новому языку. Тем не менее, прилагательное «новый» надо понимать правильно: Oberon появился из Modula за счет того, что что-то было добавлено, а что-то было изъято. Идя по эволюционному пути развития, а не по революционному, мы сохраняли традиции длительной разработки, которые идут от Algol к Паскалю, затем к Modula-2 и, наконец, к Oberon. Общая характерная черта всех этих языков — это их процедурная, а не функциональная модель, а также жесткая типизация данных. Еще более фундаментальной является идея абстракции: язык должен определяться в терминах математических, абстрактных концепций без ссылки на какой бы то ни было вычислительный механизм. И только если язык удовлетворяет этому критерию, он может называться

«высокоуровневым». Решительно никакая синтаксическая глазурь не в силах заставить язык обойтись без этого атрибута.

Описание языка должно быть стройным и лаконичным. Этого можно достичь лишь путем аккуратного выбора соответствующих абстракций и подходящей структуры, которая позволяет их комбинировать. Руководство по языку должно быть весьма коротким и должно избегать пояснения частных случаев, проистекающих из общих правил. Мощь формализма не должна измеряться длиной описания. Более того, чересчур пространное описание — это характерный симптом двояких толкований. В этом плане целью должна быть не сложность, а простота.

Несмотря на свою краткость описание должно быть законченным. Полнота должна достигаться в рамках выбранных абстракций. Ограничения, накладываемые конкретными реализациями, не относятся собственно к описанию языка. Примерами таких ограничений являются максимальные значения чисел, арифметические ошибки округления и усечения, действия, предпринимаемые в тех случаях, когда программа нарушает установленные правила. Не должна проявляться потребность в таком дополнении к описанию языка, когда толстенные описания стандартов покрывают «непредвиденные» ситуации.

В то же время язык программирования не должен быть одной лишь математической теорией. Он должен быть практическим инструментом. Это подразумевет определенные ограничения, накладываемые на краткость формализма. Несколько языковых средств Oberon с чисто теоретической точки зрения являются излишними. И, тем не менее, они присутствуют в языке из сугубо практических соображений, то ли для удобства самого программиста, то ли для достижения эффективной кодогенерации без использования в компиляторах сложных «оптимизирующих» алгоритмов сопоставления шаблонов. Примерами таких языковых средств являются наличие нескольких форм оператора цикла, а также существование стандартных процедур, таких как INC, DEC и ODD. Они не усложняют ни язык, ни компилятор.

Все эти аргументы нужно иметь в виду, когда производится сравнение Oberon с иными языками. Ни язык, ни описывающий его документ не достигают идеала; но Oberon приближается к этой цели гораздо лучше своих предшественников.

Компилятор для Oberon был реализован для процессоров семейства NS32000 и был встроен в операционную среду Oberon [2]. Этот компилятор требует менее 50 Кбайт памяти, состоит из 6 модулей, общим размером около 4000 строк исходного текста и сам себя компилирует примерно за 15 секунд на рабочей станции с 25 МГц процессором типа NS32532.

После многочисленных экспериментов в программировании на языке Oberon был определен и реализован его новый вариант. Различия между этими двумя версиями собраны в конце данной работы. Далее мы приводим краткое введение в (пересмотренный) Oberon для тех, кто знаком с Modula (или Паскалем), концентрируясь при этом на добавленных средствах и перечисляя те, которые были из Modula удалены. А чтобы было более понятно, начнем с изъятых средств.

Языковые средства, удаленные из Modula

Типы данных

Удалены вариантные записи, поскольку они создают серьезные трудности для реализации надежной системы распределения памяти, основанной на автоматической сборке мусора. Функциональность вариантных записей сохраняется за счет введения расширяемых типов данных.

Скрытые типы служат концепции абстрактных типов данных и инкапсуляции информации. Они также удалены, поскольку их заменяет новое средство расширенных типов записи.

Тип перечисление — слишком простое средство, чтобы оно могло выйти из-под контроля. Однако, оно не позволяет распространять расширяемость за пределы модуля. И либо нужно ввести средство для расширения типа перечисление, либо же от типа перечисление надобно отказаться. Причина, по которой мы выбрали второй путь — путь радикального решения — кроется в том, что во все возрастающем числе программ непродуманное использование

перечислений (и диапазонов) ведет к демографическому взрыву среди типов, что, в свою очередь, ведет не к ясности программ, а к их многословию. В связи с использованием экспорта и импорта перечисления приводят к исключению из правил, и согласно ему импорт идентификатора типа также приводит к автоматическому импорту всех связанных с типом идентификаторов констант. Это исключение нарушает концептуальную простоту и создает для реализаторов языка неприятные проблемы.

Типы диапазонов были введены в Паскале (и сохранены в Modula) по двум причинам: (1) чтобы подчеркнуть тот факт, что переменная принимает значения из ограниченного диапазона базового типа, и чтобы дать возможность компилятору генерировать соответствующий проверочный код для присваиваний; (2) чтобы позволить компилятору выделять минимально необходимое пространство памяти для хранения значений из указанного диапазона. Это желательное свойство в плане использования упакованных записей. Лишь в небольшом числе реализаций используется данное преимущество экономии памяти, поскольку при этом компилятор довольно значительно усложняется. Причины под номером 1 явно недостаточно, чтобы оставить в Oberon работу с диапазонами.

Ввиду отсутствия перечислений и диапазонов избыточным выглядит возможность определять множества на основе данного типа элементов. Вместо этого в язык был введен лишь атомарный тип SET, значениями которого выступают множества целых чисел в диапазоне от 0 до некоторого максимального значения, определяемого реализацией.

Атомарный тип CARDINAL был введен в Модулу, чтобы позволить на 16-разрядных компьютерах работать с адресной арифметикой в диапазоне от 0 до 2^{16}. В связи с преобладанием в современных процессорах 32-разрядных адресов необходимость в беззнаковой арифметике практически отпала, а потому тип CARDINAL в язык Oberon не вошел. Одновременно с этим исчезли и проблемы несовместимости операндов типов CARDINAL и INTEGER.

Ссылочные типы теперь могут строиться как указатели лишь на записи и массивы.

Понятие определяемого программистом типа индексов для массивов также не вошло в Oberon: теперь все индексы могут быть только целыми числами. Более того, нижняя граница индексов теперь строго равна 0 и в описании массива фигурирует только количество элементов, а не пара значений, определяющих границы индексов. Этот отход от уже устоявшейся традиции, восходящей своими корнями еще к языку Algol-60, наглядно демонстрирует принцип удаления из языка всего несущественного. Спецификация произвольной нижней границы вряд ли придает языку дополнительную выразительную силу. В то же время это является скорее ограниченным видом отображения индексов, которое ведет к скрытым вычислениям, несравнимым с удобством использования. Скрытые накладные расходы проступают особенно сильно при проверке границ и при работе с динамическими массивами.

Модули и правила экспорта-импорта

Эксперименты с Modula, проводимые на протяжении последних восьми лет, показали, что локальные модули используются крайне редко. С учетом дополнительной сложности, вносимой в компилятор для их поддержки, и принимая во внимание дополнительные усложнения правил видимости, которые должны быть внесены в описание языка, отказ от локальных модулей выглядит вполне логичным решением.

Квалификацию «M.x» импортируемого объекта «x» именем экспортируемого его модуля «M» в Modula можно обойти за счет использования оператора FROM M IMPORT x. От такой возможности в Oberon мы также отказались. Опыт работы с системами программирования, оперирующими большим количеством модулей, показал, что явная квалификация каждого использования объекта «x» — это наиболее предпочтительное решение. Связанное с этим упрощение компилятора носит в данном случае второстепенное значение.

Двойственная роль головного (main) модуля в Modula приводит к концептуальной неясности. С одной стороны, головной модуль задает модуль в смысле пакета данных и процедур с замкнутой областью видимости, а, с другой, он определяет единственную процедуру, называемую основной программой. Модуль состоит из двух блоков, которые называются описательной частью (definition

part) и исполнительной частью (implementation part). В случае головного модуля описательная часть отсутствует.

В отличие от такого подхода в языке Oberon модуль является самостоятельным и законченным строительным блоком и определяет единицу компиляции. Описательная и исполнительная части слиты воедино: имена, которые должны быть видны в клиентских модулях, т.е. экспортируемые идентификаторы, помечаются специальными знаками: они обычно предшествуют описаниям тех объектов, которые не экспортируются. При компиляции генерируется измененный объектный файл (object file) и новый символьный файл (symbol file). В последнем содержится информация об экспортируемых объектах, которая впоследствии используется при компиляции модулей-клиентов данного модуля. Генерирование нового символьного файла должно управляться специальной опцией компилятора, поскольку новая версия такого файла сделает недействительной компиляцию всех модулей-клиентов.

Понятие основной программы не вошло в язык Oberon. Ее место занял набор модулей, связанных друг с другом через механизм экспорта-импорта и содержащих процедуры без параметров. Эти процедуры рассматриваются как независимо активируемые и они называются «командами». Активация имеет вид «M.P», где «P» обозначает команду, а «M» — содержащий ее модуль. Результат работы команды рассматривается не просто как обычный прием входных данных для основной программы и трансформирование их в выходные, а как изменение состояния, выражаемого глобальными данными.

Операторы

Оператор WITH языка Modula-2 также не вошел в Oberon. Также как и в случае с импортируемыми идентификаторами, здесь предпочтительно явное указание поля идентификатора. В Oberon была введена другая форма оператора WITH; она носит совершенно иную функцию, называется локальным охранником (regional guard) и будет затронута позднее.

Отказ от оператора FOR — другой пример разрыва древней традиции. Замысловатый механизм оператора FOR, принятый в языке Algol-60, был значительно упрощен в языке Паскаль (и в Modula). Незначительная практическая ценность этого оператора привела к тому, что в Oberon он отсутствует.

Низкоуровневые средства

Modula обеспечивает доступ к машиннозависимым средствам через низкоуровневые конструкции, такие, как типы данных ADDRESS и WORD, абсолютные адреса переменных и функции приведения типов. Большинство из них содержится в модуле, который называется SYSTEM. Предполагалось, что эти средства будут использоваться крайне редко, и заметить использование их в программе можно будет легко по наличию идентификатора SYSTEM в списке импорта. Однако практика показала, что значительное число программистов импортирует этот модуль весьма неразборчиво. Особенную опасность здесь составляют функции приведения типов (type transfer functions).

Предпочтительно отказывать в претензии на переносимость тем программам, которые импортируют «стандартный», но системнозависимый модуль. По этим причинам функции приведения типа, которые обозначались идентификатором соответствующего типа, в Oberon не вошли, а модуль SYSTEM был ограничен предоставлением нескольких машиннозависимых функций, которые обычно компилируются в прямые вставки (inline code). Типы ADDRESS и WORD были заменены типом BYTE, для которого правила совместимости несколько ослаблены. Каждая реализация языка может предоставлять в модуле SYSTEM и свои дополнительные средства. Использование модуля SYSTEM делает программу явно зависящей от реализации и следовательно непереносимой.

Параллельность

Система Oberon не требует никаких языковых средств для описания параллельных процессов. Следовательно соответствующим зачаточным средствам языка Modula, в частности, сопрограммам (coroutine), в Oberon места не нашлось. Это является всего лишь отражением наших реальных потребностей в данном конкретном проекте, а отнюдь не отношением к общей актуальности использования параллельности в программировании.

Средства, введенные в Oberon

На фоне большого числа не вошедших в язык средств нововведения в количественном выражении выглядят крайне скромно. Наиболее важными концепциями явились расширение типов (type extension) и включение типов (type inclusion). Кроме того, открытые массивы могут иметь несколько измерений (индексов), тогда как в Modula могло быть только одно.

Расширение типов

Наиболее важным добавлением является механизм расширенных типов записи. Он позволяет конструировать новые типы на основе уже существующих и задает определенную степень совместимости между новыми и старыми типами. Предположим, что у нас имеется тип

```
T = RECORD x,y: INTEGER END
```

В дополнение к существующим полям могут быть заданы новые:

```
T0 = RECORD (T) z: REAL END

T1 = RECORD (T) w: LONGREAL END
```

Здесь определяются типы с полями x,y,z и x,y,w соответственно. Мы говорим, что тип T'

```
T' = RECORD (T) <описание_полей> END
```

является (прямым) расширением типа T, и наоборот, T называется (прямым) базовым (base) типом для T'. Расширенные типы, в свою очередь, могут быть сами расширены. Из этого вытекает следующее определение.

Тип T' является расширением типа T, если T' = T или если T является прямым расширением расширения типа T. И наоборот, T является базовым типом для T', если T = T' или если T является прямым базовым типом базового типа для типа T'. Это отношение мы обозначаем T'—> T.

Правило совместимости по присваиванию устанавливает, что значения расширенного типа могут присваиваться переменным его базового типа. Например, запись типа T0 может быть присвоена переменной типа T. Это присваивание затрагивает только поля «x» и «y» и по сути осуществляет проекцию значения на пространство, определяемое базовым типом.

Важно разрешить модулям, которые импортируют базовый тип, иметь возможность самим определять расширенные типы. На практике это вполне нормальная ситуация.

Концепция расширенного типа данных с особой силой проявляется в случае работы с указателями. Соответственно мы говорим, что ссылочный тип P', указывающий на T', расширяет ссылочный тип P, если P указывает на тип T, который является базовым по отношению к T'. Правило присваивания справедливо и для этой ситуации. Теперь стало возможным формировать такие структуры данных, узлы которых имеют различающиеся типы, иными словами,

неоднородные структуры данных. Неоднородность автоматически появляется за счет того, что узлы соединены через указатели с общим базовым типом.

Как правило, ссылочные поля размещаются в структуре элемента базового типа T, а процедуры, обеспечивающие работу с данной структурой, определяются в том же самом (базовом) модуле, где описан тип T. Отдельные расширения (варианты) определяются в клиентных модулях вместе с процедурами, оперирующими с узлами расширенного типа. Такая схема находится в полном соответствии с понятием расширяемости системы (system extensibility): новые модули, определяющие новые расширения, могут быть добавлены в систему, не требуя ни изменения базовых модулей, ни даже их перекомпиляции.

Доступ к отдельному узлу через указатель, связанный с базовым типом, обеспечивает только проецируемый взгляд на данные узла, а потому необходимо средство для расширения этого взгляда. Это зависит от способности определять фактический тип данного узла, что достигается пробой типа (type test), булевским выражением вида

```
t IS T'
p IS P'
```

Если при взятии пробы получается утвердительный результат, то должно допускаться присваивание t' := t и p' := p, где t' типа T', а p' типа P'. Однако, статический взгляд типов запрещает это. Заметим, что оба присваивания нарушают правило совместимости по присваиванию. Требуемое присваивание становится возможным за счет использования охранников типов (type guard) в виде:

```
t' := t(T')
p' := p(P')
```

а доступ к полю «z» типа T0 (см. предыдущие примеры) достигается за счет использования охранника типа в адресате (designator) «t(T0).z». Здесь охранник гарантирует, что «t» содержит (в настоящий момент) значение типа T0. По аналогии с проверками границ массивов и селекторами оператора CASE, ошибка в охраннике приводит к аварийному завершению программы.

Если охранник вида t(T) гарантирует, что «t» имеет тип T, только в случае адресата, который начинается с «t», то локальный охранник (regional type guard) обеспечивают такую гарантию в рамках целой последовательности операторов. Он имеет вид:

```
WITH t: T DO <последовательность_операторов> END
```

и специфицирует, что «t» должен рассматриваться как имеющий тип T на всей последовательности операторов. Как правило, T является расширением описанного типа объекта «t». Заметим, что присваивание объекту «t» внутри данной области требует, чтобы присваиваемое значение имело тип T (или его расширение). Локальный охранник служит для уменьшения числа вычислений для обычных охранников.

В качестве примера использования пробы типа и охранников рассмотрим следующие типы Node и Object, заданные в модуле «M»:

```
TYPE Node = POINTER TO Object;
     Object = RECORD key, x, y: INTEGER;
       left, right: Node
     END;
```

Элементы в структуре дерева опираются на переменную, называемую «root» (типа Node), а поиск их осуществляется через процедуру «element», определенную в модуле «M».

```
PROCEDURE element (k: INTEGER): Node;
  VAR p: Node;
BEGIN p := root;
  WHILE (p # NIL) & (p.key # k) DO
    IF p.key < k THEN p := p.left ELSE p := p.right END
```

```
    END;
    RETURN p
END element;
```

И пусть расширения типа Object будут определены (вместе с соответствующими ссылочными типами) в модуле M1, который является клиентом модуля M:

```
TYPE
  Rectangle = POINTER TO RectObject;
  RectObject = RECORD (Object) w, h: REAL END;
  Circle = POINTER TO CircleObject;
  CircleObject = RECORD (Object) rad: REAL; shaded: BOOLEAN END;
```

После проведения поиска элемента, для выбора между различными расширениями используется проба типа, а для доступа к полям расширения — охранник типа. Пример:

```
p := M.element(k);
IF p # NIL THEN
  IF p IS Rectangle THEN ... p(Rectangle).w ...
  ELSIF (p IS Circle) & ~p(Circle).shaded THEN ... p(Circle).rad ...
  ELSIF ...
```

Расширение системы базируется на той предпосылке, что новые модули, определяющие новые расширения, могут быть добавлены без какой бы то ни было адаптации или даже перекомпиляции уже существующих частей, хотя компоненты новых типов включаются в уже существующие структуры данных.

Механизм расширения типа не только заменяет вариантные записи языка Modula-2, но и представляет альтернативный путь для обеспечения надежности типов. Столь же важна и та роль, которую играют связанные типы в общей иерархии типов. Сравним, к примеру, типы Modula

```
T0' = RECORD t: T; z: REAL END;
T1' = RECORD t: T; w: LONGREAL END;
```

которые связаны с описанием упомянутого ранее типа T, с приведенными выше расширенными типами Oberon — T0 и T1. Во-первых, типы языка Oberon воздерживаются от введения новой области именования. Считая, что переменная «r0» имеет тип T0, мы пишем «r0.x», а не «r0.t.x» как в случае Modula. Во-вторых, типы T, T0' и T1' различны и друг с другом не связаны. Тогда как типы T0 и T1 связаны с типом T на правах расширения. Это проявляется при взятии пробы типа, которая гарантирует, что переменная «r0» принадлежит не только типу T0, но и базовому типу T.

Описание расширенных записей, проба типа и охранник типа — это единственные дополнительные средства, введенные в данном контексте. Более всестороннее обсуждение данного вопроса можно найти в работе [3]. Концепция эта весьма схожа с понятием класса, используемым в языках Simula-67 [4], Smalltalk [5], Object Pascal [6], C++ [7] и других, где говорят, что свойства базового класса наследуются производными классами. Механизм классов предусматривает, что все процедуры, применимые к объектам класса, будут определены вместе с описанием данных. Эта догма проистекает из понятия абстрактного типа данных, но она служит серьезным препятствием при разработке больших систем, где крайне желательна возможность добавить новые процедуры, определенные в дополнительных модулях. Очень неудобно обязательно переопределять класс только лишь потому, что какой-то метод (процедура) должен быть добавлен или изменен, в особенности когда это изменение требует перекомпиляции описания класса и всех его клиентных модулей.

Мы особо подчеркиваем, что механизм расширения типа также применим и к статически описанным объектам, используемым в качестве параметров. (Хотя, как показано в приведенном выше примере, его ведущая роль возрастает при построении разнородных динамических структур данных.) Под такие объекты отводится память в рабочем пространстве, организованном в виде стека записей об активации процедур, и потому здесь используется преимущество крайне эффективной схемы выделения и утилизации памяти.

В Oberon с объектами в тексте программы связаны скорее процедурные типы, нежели процедуры (методы). Подключение (binding) фактических методов (специфических процедур) к объектам (экземплярам) откладывается до того, как программа начнет исполняться. Ассоциация процедурного типа с типом данных возникает через описание поля записи. Этому полю назначается процедурный тип. Ассоциация метода (если пользоваться терминологией языка Smalltalk) возникает через присваивание специфической процедуры как некоего значения конкретному полю, а не через статическое объявление в описании расширенного типа, который затем «переопределяет» объявления, фигурирующие в базовом типе. Такая процедура называется «обработчиком» (handler). С использованием пробы типа обработчик способен разбираться в различных расширениях базового типа для записи (объекта). В Smalltalk правила совместимости между классом и его подклассами ограничиваются указателями и, тем самым, нежелательным образом смешивают концепции метода доступа и типа данных. В Oberon отношение между типом и его расширениями опирается на уже устоявшееся математическое понятие проекции.

В Modula имеется возможность определять ссылочный тип внутри исполнительного модуля и проэкспортировать его в качестве скрытого типа путем использования того же самого идентификатора в соответствующем описательном модуле. Конечный эффект выражается в том, что тип экспортируется, тогда как все его свойства остаются скрытыми (не видимыми клиентам). В Oberon этот механизм обобщается в том смысле, что выделение экспортируемых полей записи совершенно произвольно и включает те случаи, когда экспортируются сразу все поля и когда не экспортируется ни одно. Набор экспортируемых полей определяет частичный взгляд, иными словами, открытую проекцию (public projection) на своих клиентов.

В клиентских модулях, так же как и в самом модуле, можно определять расширения базового типа (например, TextViewers и GraphViewers). Важен также тот факт, что неэкспортируемые компоненты (поля) могут иметь типы, которые не экспортируются. Следовательно, можно весьма эффективно скрывать определенные типы данных, сохраняя при этом возможность обращения к ним со стороны компонентов (скрытых) экспортируемых типов.

Включение типов

Современные процессоры отличает поддержка арифметических операций сразу в нескольких числовых форматах. Желательно, чтобы все эти форматы нашли свое отражение в языке в качестве атомарных (basic) типов. Oberon поддерживает 5 из таких форматов:

LONGINT, **INTEGER**, **SHORTINT** (целочисленные типы)
LONGREAL, **REAL** (вещественные типы)

С распространением атомарных типов ослабление между ними правил совместимости становится почти обязательным. (Заметим, что в Modula числовые типы INTEGER, CARDINAL и REAL являются несовместимыми.) В связи с этим в язык вводится понятие включения типов (type inclusion). Тип T включает в себя тип T', если значения типа T' являются также и значениями типа T. В Oberon постулируется следующая иерархия типов:

LONGREAL) **REAL**) **LONGINT**) **INTEGER**) **SHORTINT**

Соответственно ослабляется и правило присваивания. Значение типа T' может быть присвоено переменной типа T, если T' включен в T (или если T' расширяет тип T), т.е. если T) T' или T' —> T. В этом плане мы возвращаемся (и расширяем) к гибкости языка Algol-60. Пусть, например, даны переменные

```
i: INTEGER; k: LONGINT; x: REAL
```

тогда присваивания

```
k := i;  x := k;  x := 1;  k := k + 1;  x := x * 10 + i
```

не противоречат принятым правилам, в то время как операторы

```
i := k; k := x
```

недопустимы. Присваивание x := k может использовать усечение.

Наличие нескольких числовых типов — это очевидная уступка реализациям языка, которые могут выделять разные объемы памяти для переменных различных типов и которые, тем самым, предоставляют возможность для экономного использования памяти. Этот практический аспект в отношении математической абстракции нельзя игнорировать. Понятие включения типов минимизирует последствия для программиста и требует лишь нескольких неявных инструкций для изменения представления данных, такие как знаковые расширения и преобразования целых в числа с плавающей точкой.

Различия между Oberon и пересмотренным Oberon (Revised Oberon)

Пересмотр языка Oberon был произведен после большого числа экспериментов в использовании и реализации данного языка. И вновь этот процесс был вызван стремлением к упрощению и обобщению. Различия между исходной версией [8] и пересмотренной версией [9] следующие.

1. Описательная и исполнительная части модуля слиты воедино. Это вызвано желанием иметь спецификацию модуля с точки зрения программиста и с точки зрения компилятора в виде одного документа. Спецификация интерфейса для клиентов модуля (описательная часть) может быть получена автоматически. Объекты, ранее описываемые в описательной части (и еще раз встречающиеся в исполнительной части), помечаются специальным значком экспорта. Таким образом, отпадает необходимость компилятору проводить структурное сопоставление двух текстов.
2. Синтаксис списка типов параметров в описании процедурного типа выглядит точно также, как и для заголовков обычных процедур. Из этого следует, что вводятся фиктивные идентификаторы; они могут быть полезны в качестве комментариев.
3. Ослаблены правила обязательного следования описания типов только после описания констант и описания переменных только после описания типов.
4. Апостроф более не является разграничителем строк.
5. Ослабленное правило совместимости параметров для формального типа ARRAY OF BYTE применимо только к параметрам, передаваемым по ссылке.

Итоги

Язык Oberon возник из языка Modula-2 и вобрал в себя опыт многих лет программирования на Modula. Было удалено значительное количество языковых средств. Они в большей степени усложняли язык и компилятор, нежели представляли истинную силу и гибкость выразительных возможностей. Было добавлено небольшое число новых механизмов, наиболее важным из которых является концепция расширения типа.

Эволюция нового языка, который стал гораздо меньше и при этом еще мощнее, чем его предшественник, резко контрастирует с общепринятой практикой и в то же время имеет неоценимые преимущества. Помимо более простых компиляторов, результатом этого стало лаконичное описание [9] — необходимое условие для любого инструмента, который должен служить для построения изощренных и надежных систем.

Благодарности

Просто невозможно поблагодарить всех тех, кто так или иначе подпитывал своими идеями то, что теперь называется Oberon. Большинство идей пришло от использования и изучения существующих языков, таких как Modula-2, Ada, Smalltalk и Cedar, которые часто предостерегали нас от того, как не надо делать. Особых слов благодарности достоин первый пользователь Oberon — Юрг Гуткнехт (J. Gutknecht). Автор признателен за его настойчивость в деле изъятия отмерших средств и подведения основательного математического фундамента под оставшиеся механизмы. И, наконец, хочется выразить свою признательность моим анонимным рецензентам, кто очень внимательно прочитал рукопись и внес весьма ценные замечания для улучшения этой статьи.

ПРИЛОЖЕНИЕ

Синтаксис языка Oberon

ident = letter {letter | digit}.

number = integer | real.

integer = digit {digit} | digit {hexDigit} «H».

real = digit {digit} «**.**» {digit} [ScaleFactor].

ScaleFactor = («**E**» | «**D**») [«**+**» | «**-**»] digit {digit}.

hexDigit = digit | «**A**» | «**B**» | «**C**» | «**D**» | «**E**» | «**F**».

digit = «**0**» | «**1**» | «**2**» | «**3**» | «**4**» | «**5**» | «**6**» | «**7**» | «**8**» | «**9**».

CharConstant = "«" character "«" | digit {hexDigit} «**X**».

string = "«" {character} "«" .

identdef = ident [«*****»].

qualident = [ident «**.**»] ident.

ConstantDeclaration = identdef «=» ConstExpression.

ConstExpression = expression.

TypeDeclaration = identdef «**=**» type.

type = qualident | ArrayType | RecordType | PointerType | ProcedureType.

ArrayType = **ARRAY** length {«**,**» length} **OF** type.

length = ConstExpression.

RecordType = **RECORD** [«(« BaseType «)»] FieldListSequence **END**.

BaseType = qualident.

FieldListSequence = FieldList {«**;**» FieldList}.

FieldList = [IdentList «**:**» type].

IdentList = identdef {«**,**» identdef}.

PointerType = **POINTER TO** type.

ProcedureType = **PROCEDURE** [FormalParameters].

VariableDeclaration = IdentList «**:**» type.

designator = qualident {«**.**» ident | «**[**« ExpList «**]**» | «**(**« qualident «**)**» | «**^**» }.

```
ExpList = expression {«,» expression}.
expression = SimpleExpression [relation SimpleExpression].
relation = «=» | «#» | «<« | «<=» | «>» | «>=» | IN | IS.
SimpleExpression = [«+»|«-»] term {AddOperator term}.
AddOperator = «+» | «-» | OR .
term = factor {MulOperator factor}.
MulOperator = «*» | «/» | DIV | MOD | «&» .
factor = number | CharConstant | string | NIL | set |
          designator [ActualParameters] | «(« expression «)» | «~» factor.
set = «{« [element {«,» element}] «}».
element = expression [«..» expression].
ActualParameters = «(« [ExpList] «)».
statement = [assignment | ProcedureCall |
        IfStatement | CaseStatement | WhileStatement | RepeatStatement |
        LoopStatement | WithStatement | EXIT | RETURN [expression] ].
assignment = designator «:=» expression.
ProcedureCall = designator [ActualParameters].
StatementSequence = statement {«;» statement}.
IfStatement = IF expression THEN StatementSequence
        {ELSIF expression THEN StatementSequence}
        [ELSE StatementSequence] END.
        CaseStatement = CASE expression OF case {«|» case}
        [ELSE StatementSequence] END.
        case = [CaseLabelList «:» StatementSequence].
        CaseLabelList = CaseLabels {«,» CaseLabels}.
        CaseLabels = ConstExpression [«..» ConstExpression].
        WhileStatement = WHILE expression DO StatementSequence END.
        RepeatStatement = REPEAT StatementSequence UNTIL expression.
        LoopStatement = LOOP StatementSequence END.
        WithStatement = WITH qualident «:» qualident
                        DO StatementSequence END.

ProcedureDeclaration = ProcedureHeading «;» ProcedureBody ident.
ProcedureHeading = PROCEDURE [«*»] identdef [FormalParameters].
ProcedureBody = DeclarationSequence [BEGIN StatementSequence] END.
ForwardDeclaration = PROCEDURE «^» ident [«*»] [FormalParameters].
DeclarationSequence = {CONST {ConstantDeclaration «;»} |
        TYPE {TypeDeclaration «;»} | VAR {VariableDeclaration «;»}}
        {ProcedureDeclaration «;» | ForwardDeclaration «;»}.
FormalParameters = «(« [FPSection {«;» FPSection}] «)» [«:» qualident].
FPSection = [VAR] ident {«,» ident} «:» FormalType.
FormalType = {ARRAY OF} (qualident | ProcedureType).
```

ImportList = **IMPORT** import {«**,**» import} «**;**» .

import = ident [«**:=**» ident].

module = **MODULE** ident «**;**» [ImportList] DeclarationSequence

[**BEGIN** StatementSequence] **END** ident «**.**» .

Литература

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

[2] Gutknecht J., Wirth N. (1989) The Oberon System // Software — Practice and Experience, Vol.19, No.9, p.857-893.

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

[4] Birtwistle G. et al. (1973) Simula Begin // Auerbach.

[5] Goldberg A., Robson D. (1983) Smalltalk-80: The Language and its Implementation // Addison-Wesley.

[6] Tesler L. (1985) Object Pascal Report // Structured Language World, Vol.9, No.3, p.10-14.

[7] Stroustrup B. (1986) The Programming Language C++ // Addison-Wesley.

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

[9] Wirth N. (1990) The programming language Oberon (Revised Report) // ETH.