**Абстрактный и конкретный синтаксис.**

При рассмотрении приемов программирования и примеров на разных языках необходимо как можно больше отвлекаться от частных особенностей и учиться видеть за ними общее с тем, чтобы оставшиеся различия были бы уже принципиальными. Отделить существенное от несущественного помогает, в частности, соотношение между синтаксисом и семантикой. Например, оказывается, что некоторые синтаксические особенности КС-грамматики языка не нужны для описания его семантики.

Обычное синтаксическое определение языка задает конкретные синтаксические правила построения программы как строки символов. При этом определяется, какие структурные элементы могут быть выделены в тексте программы (конкретное представление программы).

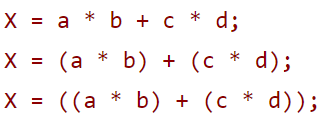
Действия абстрактного вычислителя определяются на структурном представлении программы и не зависят от многих особенностей конкретного синтаксиса. Например, для присваивания важно лишь то, что в нем есть получатель и источник, сам по себе знак присваивания (=,:= или, скажем, LET ) совершенно не важен. Более того, неважно, в каком порядке расположены составные части присваивания в тексте программы. Например, конструкция языка COBOL



выражающая присваивание получателю Z значения выражения X+Y, совершенно аналогична обычному оператору присваивания.

Для того, чтобы четко отличать конкретное представление от существенной структуры, стоит рассматривать конкретный и абстрактный синтаксис. Скажем, абстрактный синтаксис всех перечисленных форм операции присваивания один и тот же.

Аналогично, три оператора



и подобные им полностью эквивалентны с точки зрения абстрактного синтаксиса, тогда как с точки зрения текстового представления — различны.

Таким образом, нужна структура синтаксических понятий, которая соответствует некоторому алгоритмически разрешимому8 понятию эквивалентности программ. Но это понятие эквивалентности должно быть исключительно простым, поскольку теоретические результаты показывают, что нетривиальные понятия эквивалентности программ неразрешимы. Выбранное понятие эквивалентности определяет структурное представление синтаксиса, используемого для задания абстрактного вычислителя (абстрактно-синтаксическое представление).

Фрагментом абстрактно-синтаксического представления является чаще всего применяемый на практике ход. Задают понятие синтаксической эквивалентности, которое очевидным образом согласуется с функциональной эквивалентностью. Так, например, предложения, перечисленные в примере 4.1, могут описываться следующим понятием синтаксической эквивалентности: скобки вокруг подвыражений, связанных операцией более высокого приоритета, чем операция, примененная к их результату, могут опускаться. В данном смысле присваивание рассматривается как операция, имеющая более низкий приоритет, чем любая из арифметических операций. Таким образом, например, определяется эквивалентность выражений в языке Prolog. Подвыражение X + 3, скажем, является в нем всего лишь другим вариантом записи для + (X, 3), и при вычислении характеристик выражения оно прежде всего преобразуется в форму без знаков операций.

Еще одну возможность, открываемую переходом к абстрактно-синтаксическим определениям, можно увидеть, если определить эквивалентность подвыражений для сложения и умножения

A + B \leftrightarrow  B + A.

Здесь абстрактная эквивалентность выражает свойство самой операции. Опыт показывает, что дальше учета ассоциативности и коммутативности в абстрактном синтаксисе двигаться весьма опасно.

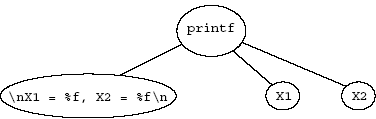
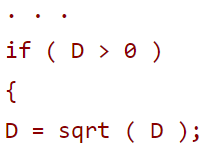


Рис. 1. Оператор печати

Пример на рис.1 иллюстрирует вызов функции:

Показанная на рисунке структура абстрактного синтаксиса данного оператора показывает, как можно ликвидировать привязку конструкции языка к конкретному синтаксису: остались только имя функции, строка и два параметра.

Рассмотрим более сложный пример, показывающий родство абстрактного синтаксиса традиционных языков. На рис. 2 показано абстрактное представление фрагмента текста следующей программы:



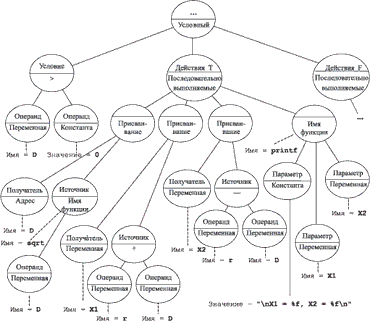


Рис. 2. Представление фрагмента текста программы

**Статическая и динамическая семантика**

Динамическая семантика - это структура в логике и семантике естественного языка, которая трактует значение предложения, как возможность обновления контекста.

В статической семантике знание значения предложения равносильно знанию того, когда оно истинно; в динамической семантике знание значения предложения означает знание «изменения, которое оно вызывает в информационном состоянии любого, кто принимает передаваемые им новости». В динамических системах предложения отображаются в функции, называемые потенциалами изменения контекста, которые принимают входной контекст и возвращают выходной контекст. Первоначально динамическая семантика была разработана Ирен Хейм и Хансом Камп в 1981 году для моделирования анафоры, но с тех пор широко применяется к явлениям, включая предпосылку, множественное число, вопросы, дискурсивные отношения и модальность.

**Компиляция и интерпретация**

Компиляция - преобразование объектов (данных и операций над ними) с входного языка в объекты на другом языке для всей программы в целом с последующим выполнением полученной программы в виде отдельного шага.

Интерпретация - анализ отдельного объекта на входном языке с одновременным выполнением (интерпретацией).

Следовательно, компиляция и интерпретация отличаются не характером и методами анализа и преобразования объектов программы, а совмещением фаз обработки этих объектов во времени. То есть при компиляции фазы преобразования и выполнения действий разнесены во времени, но зато каждая из них выполняется над всеми объектами программы одновременно. При интерпретации, наоборот, преобразование и выполнение действий объединены во времени, но для каждого объекта программы.

Если посмотреть на эти различия несколько с другой стороны, то можно заметить, что интерпретатор непосредственно выполняет действия, связанные с определением или преобразованием объектов программы, а компилятор - переводит их на другой (не обязательно машинный язык). Отсюда можно сделать несколько выводов:

* для выполнения программы, написанной на определенном формальном языке после ее компиляции необходим интерпретатор, выполняющий эту программу, но уже записанную на выходном языке компилятора;
* процессор и память любого компьютера (а в широком смысле и вся программная среда, создаваемая операционной системой, является интерпретатором машинного кода);
* в практике построения трансляторов часто встречается случай, когда программа компилируется с входного языка на некоторый промежуточный уровень (внутренний язык), для которого имеется программный интерпретатор. Многие языковые системы программирования, называемые интерпретаторами, на самом деле имеют фазу компиляции во внутренне представление, на котором производится интерпретация.

Выходной язык компилятора может быть машинным языком для компьютера с другой архитектурой, нежели тот, в котором работает компилятор. Такой компилятор называется кросс-компилятором, а сама система программирования кросс-системой программирования. Такие системы используются для разработки программ для архитектур, не имеющих собственных операционных систем или систем программирования (контроллеры, управляющие микропроцессоры).

Таким образом, граница между компиляцией и интерпретацией в трансляторе может перемещаться от входного языка (тогда мы имеем чистый интерпретатор) до машинного кода (тогда речь идет о чистом компиляторе).

Создание слоя программной интерпретации для некоторого промежуточного языка в практике построения трансляторов обычно встречается при попытке обеспечить совместимость для имеющегося многообразия языков программирования, операционных систем, архитектур и т.д. То есть определяется некоторый внутренний промежуточный язык, достаточно простой, чтобы для него можно было написать интерпретатор для всего имеющегося многообразия операционных систем или архитектур. Затем пишется одни (или несколько) компиляторов для одного (или нескольких) входных языков на этот промежуточный уровень. Приведем примеры такой стандартизации:

* для обеспечения совместимости и переносимости трансляторов на компьютеры с различной архитектурой или с различными операционными системами был разработан универсальный внутренний язык (P-код). Для каждой такой архитектуры необходимо реализовать свой интерпретатор P-кода. При этом все разнообразие имеющихся компиляторов с языков высокого уровня на P-код может быть использовано без каких-либо изменений.
* язык программирования Java аналогично был разработан для обеспечения переносимости различных приложений в среде Internet.

**Проекции Футамуры-Ершова.**

В операционных семантиках обычные и смешанные вычисления существенно отличаются (хотя и обладают сходной функцией). Это противоречие устраняет трансформационный подход.

Идея трансформационного подхода:

Рассматривается программа, и ищутся команды, которые могут быть выполнены в данный момент. Выполненные команды редуцируются, т.е. убираются из программы.

Из-за итераций не все команды могут быть удалены после первого выполнения. Поэтому в правила преобразования программного текста включаются также “раскрытия”, которые отщепляют от итерации один экземпляр (для редукции).

Команды итерации хранятся в виде особого объекта-замыкания, где они либо представлены явно, либо в виде ссылки на соответствующую подпрограмму - шаг итерации (соответственно рекурсия без памяти - среда хранится явно и - с памятью).

Трансформационная семантика не показывает, какая команда должна выполняться, лишь показывает список команд, которые могут быть выполнены.

Определение:

Неподвижная точка - это состояние программы, при котором не может быть выполнена ни одна команда.

Если заданы все аргументы правильно составленной программы и выполнены все команды, то неподвижная точка есть результат (его выдача).

Если же заданы не все аргументы, то неподвижная точка - остаточная программа.

Определение:

Система преобразований любой текстовой информации, которая приводит к одной и той же неподвижной точке независимо от порядка применения допустимых преобразований называется системой Черча - Россера.

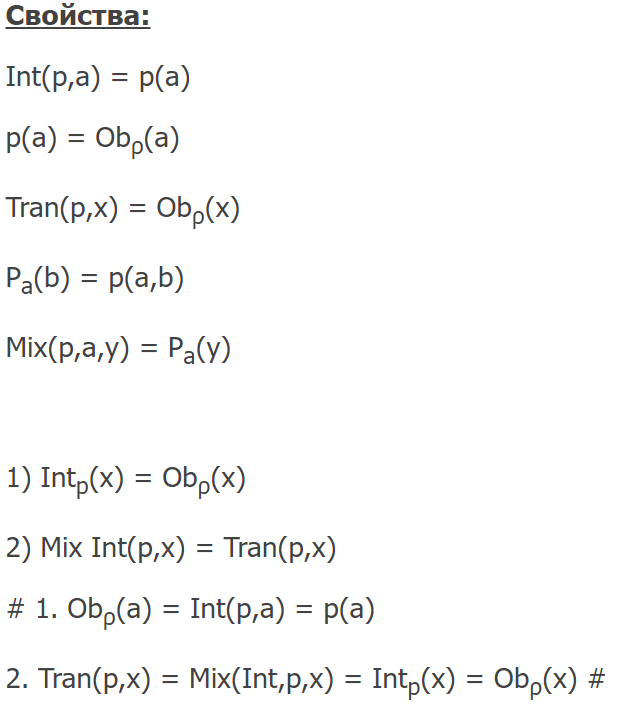
Трансформационные семантики опираются на ряд правил преобразования текста программ, которые могут быть легко расширены.

При построении трансляторов:

Проекция смешанного вычислителя на интерпретатор входного языка – это программа транслятора с этого языка на машинный. Таким образом, по теореме Успенского (следствие из проекций Футамуры), для того, чтобы на данный машинный язык можно было перевести любой другой язык программирования, необходимо и достаточно, чтобы в машинном языке были определены смешанные вычисления.

Tran(ρ,a) - транслятор с входного языка.

Obρ(x) - объектная программа, перевод ρ в машинный язык.



Смешанные вычисления хорошо соответствуют λ-исчислению, каррируется каждый аргумент, его подстановка ведет к преобразованию тела λ-абстракции, т.е. получению остаточной программы.