

Глава 1

Простой проблемно-ориентированный язык

Проблемно-ориентированные языки программирования (для краткости — DSL, от английского *Domain Specific Language*) являются языками программирования, спроектированными специально для решения определённых задач или классов задач. Они проектируются и реализуются так, чтобы в своих проблемных областях быть проще и эффективнее языков общего назначения, которые могут использоваться для решения любых потенциально разрешимых задач. Достигнуть этого можно несколькими путями:

- 1) Понятия и методы предметной области могут быть представлены в DSL в виде примитивных типов данных и базовых конструкций языка. Пользователь языка (программист) получает возможность записывать сложные алгоритмы, с лёгкостью оперируя понятиями предметной области. Код при этом получается простым и компактным, легким для понимания;
- 2) Авторы языка могут использовать глубокие познания в предметной области для того, чтобы реализовать в компиляторе или интерпретаторе DSL какие-то специальные методы расчёта или оптимизации программного кода. Программист получает возможность писать про-

граммный код на более высоком уровне абстракции, не заботясь о низкоуровневой оптимизации;

- 3) Язык может быть максимально приближён к естественному языку или к какой-то нотации, используемой в предметной области. Это облегчает написание программ пользователям, не обладающим специальной подготовкой.

В качестве разносторонних примеров DSL можно привести:

- 1) язык формул и макросов в электронных таблицах;
- 2) язык написания запросов к реляционным базам данных SQL;
- 3) язык описания грамматик для автоматической генерации синтаксических анализаторов (и эта тема будет отдельно рассмотрена в настоящей книге);
- 4) язык для создания аудио-записей `Csound`, равно как и язык для генерации изображений на основе направленных графов `GraphViz`.

При разработке DSL предпочтение отдаётся высокой эффективности в какой-то определённой проблемной области взамен универсальности, когда язык становится ориентированным исключительно на решение одной задачи (или одного класса задач). Уже упомянутый `GraphViz` эффективно рисует диаграммы и схожие изображения, но не сможет решить ни одной задачи из области межпрограммного взаимодействия, передачи данных и т. д. Другими словами, в противоположность языкам программирования общего назначения, которые могут использоваться для решения произвольных задач, проблемно-ориентированные языки «заточены» под определённые проблемные области.

1.1 Кратко о DSL и их свойствах

По способам использования DSL можно разделить на несколько зачастую пересекающихся классов:

- 1) DSL, используемые независимо от других программ и программных систем. Для таких языков реализуются самодостаточные компиляторы или интерпретаторы. Программы на таких языках записываются в файлах, исполняемых этими интерпретаторами или компиляторами (например, `Makefile`, содержащий программу для утилиты `make`).

- 2) DSL, которые реализованы при помощи системы макроопределений какого-либо базового языка универсального назначения, и записи на которых «разворачиваются» в конструкции на базовом языке специальным препроцессором (классический пример — работа препроцессора языков C и C++).
- 3) DSL, интерпретаторы которых получают вызовы из базовых программ для выполнения действий, связанных с расчётами в заданных проблемных областях. После выполнения расчётов их результат возвращается в базовое приложение. Эта схема реализуется при помощи встраивания в базовое приложение виртуальной машины DSL. Пример: интерпретатор языка Lua, встроенный в клиентскую часть многопользовательской игры World Of Warcraft
- 4) DSL, которые встроены в качестве скриптовых языков в приложения. Пользователи этих приложений могут создавать программы на таких языках и исполнять их в рамках приложений. Например, все приложения офисного пакета Microsoft Office имеют в своём составе встроенный DSL в виде языка программирования VBA.

Многие DSL могут использоваться несколькими способами, а потому могут быть отнесены к нескольким из перечисленных классов. Например, всё тот же `GraphViz` может быть обработан при помощи интерпретатора командной строки, вызван из модулей на языке Perl, а также может быть встроен в приложения для визуализации данных в графическом виде.

Эта многогранность может привести к тому, что с течением времени — по мере привнесения в язык новых возможностей — DSL может вырасти в полноценный язык общего назначения, обладающий рядом узкоспециальных особенностей. Практически все широко известные универсальные языки программирования изначально разрабатывались как специализированные, проблемно-ориентированные. Например, язык FORTRAN разрабатывался как язык для научных расчётов, язык Pascal — для обучения, язык C — для разработки системных программ, язык COBOL — для решения экономических задач, язык LISP — для решения задач искусственного интеллекта и т. д. На каждом из этих языков определённые задачи решаются лучше, проще, понятнее. Здесь ситуация очень похожа на ситуацию с естественными языками: человек использует тот язык, который знает, и чем большим количеством языков он владеет, тем шире его область общения.

В последнее время крайне широко распространены проблемно-ориентированные языки, встроенные в качестве языков исполнения сцена-

риев в различные прикладные программы и автоматизированные системы. Уже упоминалось, что все приложения офисного пакета Microsoft Office поддерживают сценарии и макросы на языке VBA. Другие пакеты офисных программ не отстают от лидера рынка и также поддерживают различные языки. Многие браузеры поддерживают несколько языков исполнения сценариев (тот же Internet Explorer поддерживает несколько диалектов языка JavaScript, VBA и ещё некоторые). Специализированные системы проектирования, разработки и т. д. сегодня уже не могут не поддерживать языки исполнения сценариев, которые позволяют продвинутому пользователю изменять поведение и функциональность своих приложений (в качестве примеров можно привести приснопамятный функциональный язык LISP в автоматизированной системе проектирования AutoCAD, объектно-ориентированный язык PML в системе трёхмерного проектирования PDMS и т. д.). Все эти языки предоставляют прикладному программисту ограниченный или даже полный доступ к модели данных и функциональности своих приложений, что позволяет оперативно производить настройку и расширение функциональности программы.

Как уже говорилось, DSL являются языками программирования, у которых само построение языка или структура данных отражают особенности решаемых задач или соглашения, принятые в предметной области. Как правило, именно сложные структуры данных дают толчок для развития DSL. Нотация для записи сложных структур данных начинает дополняться механизмами условного включения данных или условного вызова той или иной функциональности. Отсюда уже рукой подать до классических управляющих конструкций: циклов, функций, макросов, модулей и т. д. Очень важно вовремя увидеть, что описание сложного формата входных данных начинает превращаться в DSL и направить его эволюцию в правильное русло.

Приведённые примеры существующих DSL во встроеном и внешнем видах показывают, что сама по себе технология создания специализированного языка программирования достаточно популярна у программистов. Действительно, если рассмотреть внутренние DSL, то их реализация позволяет программисту предоставить конечным пользователям программ инструмент для неограниченного наращивания дополнительной функциональности, что, в свою очередь, также позволяет неограниченно расширять возможности программ совершенно сторонними к первоначальному разработчику коллективами и просто энтузиастами-одиночками.

В то же время внешние проблемно-ориентированные языки программирования позволяют решать задачу в терминах той проблемной области,

в которой она формулируется, а это крайне интересно для учёных и исследователей. Поэтому внешние DSL так популярны в академической среде, когда всевозможные научные лаборатории создают специальные языки программирования для своих нужд, а впоследствии такие языки программирования вполне могут стать языками общего назначения.

Тем не менее, у всех типов DSL имеются свои достоинства и недостатки. Резонно будет перечислить некоторые характеристики, а также плюсы и минусы двух наиболее типичных типов DSL: внешних и встроенных. Эта информация представлена в таблице ??.

Таблица 1.1. Характеристики, плюсы и минусы внешних и встроенных DSL

Характеристика	Внешний DSL	Встроенный DSL
Отношение к родительскому языку	Отличается	Совпадает на уровне синтаксиса
Синтаксис	Произвольный	Такой же, как у родительского языка
Необходимость транслятора	Транслятор необходим	Используется имеющийся транслятор
Поддержка разработки	Нет	Среда разработки родительского языка
Сложность создания	Высокая	Средняя

Другая тема, которая часто поднимается при обсуждении достоинств и недостатков DSL касается возможности их использования непрограммистами. Всевозможные эксперты могли бы программировать на «своих» языках (DSL своих проблемных областей) тогда, когда им это необходимо для решения задач. В своё время некоторые языки программирования (FORTRAN, COBOL и др.) как раз считались такими языками и предназначались именно для экспертов — инженеров технических специальностей, экономистов и т. д.

Некоторые специалисты в области построения DSL отмечают, что имеется тенденция к тому, что в недалёком будущем вся проблемно-ориентированная логика автоматизированных и информационных систем будет создаваться их пользователями, а программисты будут лишь создавать для них необходимые инструменты поддержки, позволяющие создавать и использовать такие программы. Приобщение к программированию непрофессионалов — весьма интересный подход, но нет сомнений в том,

что это — не самый веский аргумент в пользу DSL. Хороший проблемно-ориентированный язык позволяет работать более продуктивно и профессиональному программисту, даже если этим DSL не смогут пользоваться непрофессионалы. Вполне возможно, что хороший DSL должен изначально создаваться профессиональными программистами, а уже потом уточняться и пересматриваться экспертами в проблемной области.

Как отмечает М. Фаулер в своей статье [?], выгода от внедрения технологии построения и повсеместного использования DSL трудно было бы переоценить. И дело вовсе не в том, что профессиональных программистов станет меньше (или не будет вовсе), а в том, что эта технология помогла бы исправить то «ужасное состояние дел, которое сегодня наблюдается в области взаимодействия программистов и экспертов. А это является одним из камней преткновения для успешной работы над проектами».

Из всего вышеприведённого видно, что изучение принципов построения DSL и методов синтаксического анализа для разбора исходных кодов на них является достаточно интересной и перспективной задачей. Поэтому далее в настоящей книге будет изучено несколько примеров DSL для разных областей применения и разной степени сложности. В качестве первого примера предлагается рассмотреть язык для записи протоколов шахматных партий. Для этой цели здесь не будет изобретаться что-то своё, а будет использоваться стандартный формат для записи шахматных партий — PGN, описание которого кратко даётся в следующем разделе.

1.2 Нотация для записи шахматных партий

В 1994 году некто С. Дж. Эдвардс озаботился созданием доступной нотации для записи шахматных партий с возможностью обмена этой информацией между приложениями посредством механизмов межпрограммного взаимодействия. Своим создателем нотация PGN задумывалась как простой формат, который может быть прочитан и написан людьми, а также легко может быть проанализирован и сгенерирован компьютерными программами. И эта цель вполне удалась — формат получился текстовым в кодировке ASCII, поэтому может быть прочитан (или записан) человеком без специального программного обеспечения. Эти свойства позволили формату стать стандартом де-факто для указанной цели, и сегодня подавляющее большинство шахматных программ поддерживает этот формат.

Файл в формате PGN представляет собой запись одной или нескольких партий, а каждая запись о партии представляет собой набор метаданных

о партии и запись ходов, составляющих партию, в стандартной шахматной нотации (алгебраической). Кроме этого в файле могут содержаться произвольные комментарии, которые заключаются в фигурные скобки «{}».

Метаданные содержат информацию о партии: наименование турнира, номер тур, имена игроков, результат и т. д. Эта информация записывается при помощи семи стандартных меток (тегов), которые заключаются в квадратные скобки «[]». Вот эти семь меток, которые должны быть обязательно в описании каждой партии:

- 1) Event — название турнира, матча в произвольной строковой форме.
- 2) Site — место проведения в структуре «Город, Регион, Страна», при этом для обозначения страны используется кодировка по стандарту ИОС — трёхбуквенное обозначение стран Международного Олимпийского Комитета.
- 3) Date — дата проведения партии в формате YYYY.MM.DD, причём на месте любого компонента даты может стоять соответствующее количество знаков вопроса (?) — это значит, что соответствующий компонент даты неизвестен.
- 4) Round — номер тура в рамках проводимого турнира, целочисленное значение.
- 5) White — имя игрока, играющего белыми, в произвольной форме.
- 6) Black — имя игрока, играющего чёрными, в произвольной форме.
- 7) Result — результат игры, может принимать только четыре значения: «1-0» (белые выиграли), «0-1» (чёрные выиграли), «1/2-1/2» (ничья) и «*» (нечто иное, например, партия продолжается).

Все эти метки должны располагаться точно в описанном порядке и идти первыми в списке метаданных до алгебраической нотации партии. Кроме этих меток в метадаанные можно добавлять произвольную информацию, вводя для этого специальные метки. Соответственно, все они должны располагаться после перечисленных семи меток и могут содержать произвольные строковые данные. Обычно в качестве дополнительной информации записывают время начала партии, причину прекращения игры, начальную позицию и т. д.

После метаданных записывается последовательность ходов в партии. Для этого используется алгебраическая нотация, которая может быть полной или краткой. Само собой, что используется международная форма именования шахматных понятий и терминов (в России также принята локализованная версия шахматной нотации с некоторыми незначительными отличиями от международного формата). Кратко описание алгебраической нотации выглядит следующим образом:

- 1) Для обозначения фигур используются следующие коды: **K** (King) — король, **Q** (Queen) — ферзь, **R** (Rook) — ладья, **B** (Bishop) — слон, **N** (kNight) — конь, **p** (pawn) или ничего — пешка.
- 2) Клетки на шахматной доске обозначаются указанием соответствующей горизонтали и вертикали, на пересечении которых находится клетка. Горизонтالي нумеруются цифрами от 1 до 8 от белых к чёрным, вертикали — латинскими буквами от a до h слева направо (если смотреть со стороны белых). Например, в начальной позиции чёрный король стоит на поле e8, а белый ферзь — на поле d1.
- 3) Запись хода состоит из номера хода, отделяемого от описания самого хода точкой, а также описаний перемещений фигур белой и чёрной сторон, которые состоят из следующих компонентов (друг от друга записи ходов сторон отделяются пробелом):
 - (a) Тип фигуры, которая ходит.
 - (b) Поле, с которого сделан ход.
 - (c) Для простого хода тире (—), для хода со взятием вражеской фигуры буква «x».
 - (d) Поле, на которое сделан ход.
 - (e) Если пешка дошла до последней горизонтали и совершила превращение, то после целевого поля записывается знак (=) и фигура, в которую превратилась пешка.
 - (f) Для шаха знак (+), для двойного шаха — (++) , для мата — (#).
 - (g) Комментарии к ходу: «!» или «!!» для сильного хода, «?» или «??» для слабого, «?!» для рискованного, «!?» для интересного хода.
- 4) Рокировка в длинную сторону записывается как «O—O—O», в короткую: «O—O». Чтобы уменьшить неоднозначности при автоматическим

разборе файлов PGN, тут используется заглавная буква «O», а не цифра ноль «0».

При помощи этой нотации запись партии, заканчивающейся «детским» матом, выглядит следующим образом:

1. e2–e4 e7–e5
2. Bf1–c4 Nb8–c6
3. Qd1–h5?! Ng8–f6??
4. Qh5xf7#

Краткая нотация позволяет опускать начальное поле хода и символ (–) для обозначения простого хода. В этом варианте та же партия записывается следующим образом:

1. e4 e5
2. Bc4 Nc6
3. Qh5?! Nf6??
4. Qxf7#

В случае, если на указанное поле могут сделать ход несколько фигур одного типа, в краткой нотации делается уточняющее указание на то, какая именно из фигур имеется в виду. Для этого указывается либо вертикаль, на которой находится нужная фигура, либо горизонталь, либо её полные координаты. При этом ход «Ng8–f6» может быть записан как «Ngf6» или «N8f6».

Наконец, можно привести пример, который показывает типовую запись обычной партии в формате PGN:

```
[Event "Moscow Ch "]\n[Site "Moscow RUS "]\n[Date "1984.?.?.??"]\n[Round "120"]\n[White "Antoshin, Vladimir"]\n[Black "Belov, Lev"]\n[Result "1-0"]
```

```
1.f4 e5 2.fxe5 d6 3.exd6 Bxd6 4.Nf3 Nf6\n5.e3 Nc6 6.Bb5 0–0 7.Bxc6 bxc6 8. 0–0 Re8\n9.Nc3 Bg4 10.Qe1 Rb8 11.d3 Qe7 12.e4 Bxf3\n13.gxf3 Nh5 14.f4 f5 15. e5 Bc5+ 16.Kh1 Qf7\n17.Qe2 Bd4 18.Qf3 Bxc3 19.bxc3 Qd5 20.c4 Qxf3+
```

21.Rxf3 g6 22.Ba3 Kf7 23.d4 Red8 24.Rd1 Ke6
25.Bc1 Rb1 26.Rfd3 Ra1 27.d5+ Kf7 28. Be3 Rxa2
29.dxc6 Rxd3 30.cxd3 Re2 31.Bc1 Ng7 32.d4 Rc2
33.d5 Rxc4 34.e6+ Kg8 35.Be3 Ne8 36.Bxa7 Kf8
37.Bd4 Ke7 38.Be5 Nd6 39.Re1 Ra4 40.Bxd6+ cxd6
41.Rb1 1–0

Как видно, нотация PGN представляет собой достаточно мощный дескриптивный язык, который может использоваться для описания множества шахматных партий в целях последующей передачи этого описания между разными программами, архивного хранения, передачи другим пользователям и т. п. Другими словами, саму нотацию PGN можно понимать в качестве DSL для описания шахматных партий, при этом сам язык не имеет в себе управляющих команд, а использует только описательные конструкции, потому он является именно дескриптивным (описательным).

Авторы предлагают в начале изучения методов синтаксического анализа и способов проектирования и построения DSL детально изучить рассмотренную нотацию PGN и взять её в качестве методического экспериментального материала для вникания в суть современных методик синтаксического анализа. До рассмотрения методов проектирования DSL в последующих главах, нотация PGN будет использоваться в качестве достаточно примитивного DSL, на котором будут рассмотрены самые базовые основы математической лингвистики в части лексического и синтаксического разбора.

Далее в этой главе будет построен простейший синтаксический анализатор (парсер) нотации PGN, который позволит написать программу, отображающую на экране шахматную доску и ходы, сделанные в партии. В следующих главах будут рассмотрены более продвинутые методы — использование расширенной нотации Бэкуса-Наура (глава ??), автоматическая генерация на её основе парсера при помощи существующих утилит (глава ??), а также использование библиотек комбинаторов синтаксического анализа для построения парсеров вручную (глава ??).

1.3 Простой синтаксический анализатор для формата PGN

Для того чтобы реализовать простейший синтаксический анализатор для предлагаемого к рассмотрению DSL, необходимо определиться с вопросом внутреннего представления анализируемой информации. Дело в том,

что любой DSL можно понимать как язык, имеющий некоторый внешний синтаксис, который проецируется на внутреннюю структуру данных. Более того, типов внешнего синтаксиса может быть несколько. К примеру, для рассматриваемого PGN можно придумать структуру на универсальном языке XML, которая будет также описывать те же самые данные о шахматных партиях. Проекция этой информации в некоторую структуру данных для последующей обработки — одна из задач синтаксического анализатора.

После проектирования внутреннего формата представления информации необходимо создать собственно набор функций, которые будут анализировать входной поток символов, «вынимать» из него значащую информацию и помещать её в создаваемую в памяти структуру. Дальнейшие подразделы описывают два этих процесса — проектирование внутреннего представления и реализацию функций для анализа входной информации и заполнения внутренней структуры.

1.3.1 Типы данных, необходимые для воспроизведения шахматной партии

Результатом работы синтаксического анализатора является определённое внутреннее представление входной информации. Если парсер задуман как универсальный инструмент (например, как часть некоторой программной библиотеки), то внутреннее представление является полной моделью входных данных. Если же парсер создаётся для решения конкретной проблемы, то представление может быть сознательно упрощено и подстроено под решаемую задачу.

В рассматриваемом примере DSL решение вопроса о том, какое конкретное представление необходимо для хранения информации, находящейся в файле формата PGN, напрямую проистекает из описания формата. Ниже приводится предлагаемая для внутреннего представления информации структура данных, описываемая в терминах алгебраических типов данных языка Haskell.

Как уже было показано, в одном файле может находиться описание нескольких партий, каждое из которых состоит из набора метаданных (меток) и собственно ходов, составляющих партию. Тип данных языка Haskell, позволяющий хранить эту информацию, может быть описан так:

```
data Game = Game [Tag] [Exchange]
```

В свою очередь, метка `Tag` — это просто именованное строковое значение, поэтому представить его в памяти можно при помощи пары строк (введя для удобства специальные синонимы типов):

```
type TagName = String
type TagValue = String
data Tag      = Tag TagName TagValue
```

Представление одного хода в партии должно содержать порядковый номер хода и детальное описание действий обоих противников. Помимо стандартных полных ходов, когда фигуры перемещаются обоими игроками, в записи партии могут быть частичные ходы — например, если после хода белых был поставлен мат, партия была отложена и т. п. Для простоты представления финальный результат партии будет считаться разновидностью хода. Тип данных `Exchange` представляет собой объединение всех этих вариантов:

```
data Exchange
  = Exchange Int Move Move
  | PartialExchange Int Side Move
  | GameOver Result
```

Шахматная партия может закончиться победой одной из сторон, ничьей, или завершиться без определённого результата (если партия была прервана или отложена). Эти варианты дословно кодируются типом данных `Result`:

```
data Result
  = Win Side
  | Draw
  | Unknown
```

Запись о частичном ходе или победе одной из сторон содержит указание на цвет фигур, которыми играет соответствующих игрок. Для представления этого факта подойдёт простое перечисление:

```
data Side
  = White
  | Black
```

Поскольку запись о ходе может сопровождаться разнообразными необязательными пометками, разумно будет отделить обязательную часть записи от необязательной и представить их разными типами данных. В простейшей реализации парсера нотации PGN тип `BasicMove` будет содержать информацию о передвижении фигур, взятиях и рокировках, а тип `Extras` будет

содержать отметки о шахе, двойном шахе или мате. Все прочие возможные необязательные элементы будут добавлены позднее при возникновении такой необходимости.

```
data Move = Move BasicMove [Extras]
```

```
data BasicMove
  = PawnJump Coords (Maybe Piece)
  | PawnAttack Column Coords (Maybe Piece)
  | Jump Piece (Maybe Disambiguation) Coords
  | Attack Piece (Maybe Disambiguation) Coords
  | ShortCastling
  | LongCastling
```

```
data Extras = Mark String
```

Конструкторы в типе `BasicMove` говорят сами за себя. Так конструкторы `PawnJump` и `Jump` описывают обыкновенный ход пешки и более серьёзной фигуры. Необходимость применения двух конструкторов связана с тем, что пешка, в отличие от фигуры, может превратиться в фигуру при достижении последней горизонтали на своём пути. Именно по этой же причине разделены конструкторы `PawnAttack` и `Attack` — пешка может превратиться в полноценную фигуру при взятии фигуры противника, если фигура противника стояла на последней горизонтали пешки. Пустые конструкторы `ShortCastling` и `LongCastling` обозначают короткую и длинную рокировку соответственно.

Для хранения уточняющей информации о том, какая именно фигура производит ход или взятие (в случае, если несколько фигур одного типа могут совершить ход на указанное поле) используется тип `Disambiguation`. Как уже говорилось, такое уточнение может быть сделано указанием вертикали, горизонтали или полных координат фигуры:

```
data Disambiguation = Disambiguation (Maybe Column)
                      (Maybe Row)
```

Неопределённость в ходе пешки со взятием фигуры противника устраняется не при помощи значения типа `Disambiguation`, но всего лишь при помощи указания вертикали, с которой бьёт пешка — этой информации вполне достаточно для того, чтобы сделать верный выбор.

Осталось определить типы для хранения информации о координатах и видах фигур:

```

type Row = Char
type Column = Char
type Coords = (Column, Row)

```

```

data Piece
  = Pawn
  | Knight
  | Bishop
  | Rook
  | Queen
  | King

```

В заключение можно привести пример внутреннего представления для следующей последовательности ходов:

```

1.f4 e5
2.fxe5 d6
3.exd6 Bxd6
4.Nf3 g5

```

Внутреннее представление этих ходов в типах данных, определённых в текущем разделе, выглядит так:

```

Exchange 1 (Move (PawnJump ('f', '4') Nothing) [])
           (Move (PawnJump ('e', '5') Nothing) [])
Exchange 2 (Move (PawnAttack 'f' ('e', '5') Nothing) [])
           (Move (PawnJump ('d', '6') Nothing) [])
Exchange 3 (Move (PawnAttack 'e' ('d', '6') Nothing) [])
           (Move (Attack Bishop Nothing ('d', '6')) [])
Exchange 4 (Move (Jump Knight Nothing ('f', '3')) [])
           (Move (PawnJump ('g', '5') Nothing) [])

```

1.3.2 Синтаксический анализ формата PGN при помощи разбора строк и сопоставления с образцом

Задача синтаксического анализа файла формата PGN в терминах языка Haskell определяется следующим образом: необходимо создать функцию `parsePGN`, которая принимает в качестве аргумента содержимое PGN-файла, анализирует его, преобразует во внутреннее представление и возвращает список партий, содержащихся в этом файле:

```
parsePGN :: String -> [Game]
```

Очевидно, что необходимо каким-то образом разделить входную информацию на части, описывающие отдельные компоненты каждой партии, разобраться, как представить каждый из компонентов в виде определённых в предыдущем разделе типов данных, а также собрать из этих составных частей записи типа `Game`. Один из возможных способов решения этой задачи заключается в анализе префикса входных данных и принятия решения о том, какую информацию он содержит и как её перевести во внутреннее представление.

Этот подход можно проиллюстрировать таким примером. Как уже говорилось, описание партии начинается с перечисления меток, содержащих метаданные о партии. Известно также, что метка начинается с символа «[» и завершается символом «]». Если входная строка, подлежащая анализу, начинается символом «[», то можно выделить из неё префикс минимальной длины, завершающийся символом «]». Далее префикс можно преобразовать в элемент типа `Tag`, выделив из него имя метки и её значение путём дополнительного анализа. Если оставшаяся часть входной строки снова начинается символом «[», то можно повторить вышеописанные операции и выделить из входных данных следующую метку, и так далее, пока префикс входной строки не перестанет представлять собой запись о метке.

Все эти действия можно оформить в виде следующей функции:

```
parseTags :: String -> ([Tag], String)
```

Она возвращает список распознанных меток и нераспознанный остаток входной строки. Можно предположить, что подобный подход подойдёт и для анализа записей о шахматных ходах, составляющих партию, и необходимо создание такой функции:

```
parseExchanges :: String -> ([Exchange], String)
```

В этом случае стратегия анализа файла в формате PGN может быть такой: из входной строки распознаются метки при помощи функции `parseTags`, из остатка входной строки распознаётся последовательность ходов при помощи функции `parseExchanges`. Полученные списки меток и ходов используются для создания записи о партии. Если остаток входной строки пуст, то анализ файла успешно завершён. В противном случае процесс извлечения меток и ходов повторяется.

Запись этого алгоритма на языке Haskell практически дословно повторяет его изложение на русском языке, приведённое в предыдущем абзаце:

```

parseGames :: String -> ([Game], String)
parseGames input = parseGames' input []
  where
    parseGames' "" acc = (reverse acc, "")
    parseGames' input acc
      | isSpace (head input) = parseGames' (tail input) acc
      | head input == '[' = parseGames' rest' ((Game tags exchanges):acc)
      | otherwise = (reverse acc, input)
    where
      (tags, rest) = parseTags input
      (exchanges, rest') = parseExchanges rest

```

Функция `parseGames'` использует функции `parseTags` и `parseExchanges` для создания записей типа `Game`, которые накапливаются в списке-аккумуляторе `acc`. Все незначащие (пробельные) символы, встречающиеся в промежутках между партиями, пропускаются. Все накопленные в аккумуляторе записи типа `Game` возвращаются по достижению конца входной строки, либо в том случае, если первый значащий символ остатка входной строки отличается от «`]`». Поскольку новые записи добавляются в начало аккумулятора, порядок записей в нём будет обратным по отношению к тому, как партии следовали во входном файле, поэтому после окончания анализа список-аккумулятор обращается при помощи стандартной функции `reverse`.

Функцию `parseTags` можно реализовать по этому же шаблону, с использованием списка-аккумулятора и вспомогательной функции `parseTags'`:

```

parseTags :: String -> ([Tag], String)
parseTags input = parseTags' input []
  where
    parseTags' "" acc = (reverse acc, "")
    parseTags' input acc
      | isSpace (head input) = parseTags' (tail input) acc
      | head input == '[' = parseTags' rest ((Tag n v):acc)
      | otherwise = (reverse acc, input)
    where
      (tag, ']:rest) = break (==' ')] (tail input)
      n = head (words tag)
      v = read (concat $ tail $ words tag)

```

Разбор тела метки происходит по простому правилу: первое слово (набор символов до первого незначащего символа) считается именем метки, остаток текста, заключённый в кавычки — значением метки.

Код функций `parseGames` и `parseTags` практически полностью совпадает, поскольку реализованные в них правила разбора похожи и просты. В обеих функциях достаточно различать всего два случая: незначащие символы (которые необходимо пропустить) и значащие символы (с которых начинается разбор). Функция `parseExchanges` построена по тому же общему принципу, но реализует гораздо более сложные правила разбора:

```

parseExchanges :: String -> ([Exchange], String)
parseExchanges input = parseExchanges' input []
  where
    parseExchanges' "" acc = (reverse acc, "")
    parseExchanges' input acc
      | isSpace (head input) = parseExchanges' (tail input) acc
      | isGameOver input     = parseExchanges' rest'''' ((game_over game_over_text2)
      | isFullExchange       = parseExchanges' rest''''
      (exchange:acc)
      | isWhiteWins          = parseExchanges' rest''''
      ((game_over move_text2):part_exchange:acc)
      | otherwise            = (reverse acc, input)
    where
      isGameOver t = any ('isPrefixOf' t) ["1-0","0-1","1/2-1/2","*"]
      isFullExchange = isDigit (head input) && not (isDigit (head move_text2))
      isWhiteWins    = isDigit (head input) && isGameOver move_text2

      (move_no, _:rest')
= break (=='.') input
      (move_text1, _:rest'')
= break isSpace $ dropWhile isSpace rest'
      (move_text2, _:rest''')
= break isSpace $ dropWhile isSpace rest''
      (game_over_text, _:rest''''') = break isSpace input

      exchange      = Exchange (read move_no) (parseMove move_text1) (parseMove
      part_exchange = PartialExchange (read move_no) White (parseMove move_text1)
      game_over txt  = GameOver (parseResult txt)

```

Чтобы осуществить разбор очередного хода партии, необходимо различать и по-разному обрабатывать три варианта: полный ход (ходили оба игрока); ход, приведший к победе белых (чёрные не ходили); ход, приведший к победе чёрных (ходили оба игрока, следом указан результат партии). Это невозможно сделать на основании анализа первого символа остатка входной строки. Более того — это невозможно сделать на основании анализа префикса входной строки. Необходимо определить, где заканчивается текст, описывающий текущий ход, разобрать его на компоненты (описывающие ход каждой стороны) и уже на основании анализа этих компонентов принимать решение о том, какой вариант распознавания следует применить. Впрочем, после разделения текста хода на компоненты, правила анализа довольно просты: так как запись о передвижении фигуры не может начинаться с цифры, достаточно проверить первый символ текста, которых должен описывать ход чёрных. Но чтобы наверняка не ошибиться в выборе, в функции `parseExchanges` реализована проверка того, что текст, начинающийся с цифры, является правильной записью о результате окончания партии.

Далее, полученные компоненты хода должны быть преобразованы во внутреннее представление. Каждый компонент содержит всю информацию о ходе одной из сторон и ничего более, и соответствующие функции должны использовать в ходе распознавания всю входную информацию, без остатка:

```
parseMove    :: String -> Move
parseResult  :: String -> Result
```

Анализ результата партии сводится к прямому сопоставлению строк с конструкторами типа данных `Result`:

```
parseResult "1-0"      = Win White
parseResult "0-1"      = Win Black
parseResult "1/2-1/2" = Draw
parseResult "*"        = Unknown
parseResult x          = error $ "Can't parse Result from " ++ x
```

Таким же образом могут быть распознаны простейшие ходы — рокировки в длинную и короткую стороны:

```
parseMove "0-0"      = Move ShortCastling []
parseMove "0-0-0"    = Move LongCastling  []
```

Все прочие формы записи хода требуют посимвольного анализа входной строки. Например, для разбора хода со взятием, содержащего полные координаты ходившей фигуры, необходимо убедиться, что первый символ действительно описывает одну из шахматных фигур, следующие два — являются корректной записью координат одной из клеток шахматной доски, следом идёт символ «x», за которым — ещё одна пара символов, описывающих координаты на шахматной доске. Ход может завершаться пометкой о шахе или мате, которую тоже надо распознать. Механизм сопоставления с образцом языка Haskell идеально подходит для реализации этого алгоритма:

```

parseMove (p:x:y:'x':c:r:rest)
  | isPiece p &&
    isCoords x y &&
      isCoords c r = Move (Attack (decodePiece p)
                            (Just $ Disambiguation (Just x) (Just y))
                            (c, r))
                        (parseExtras rest)

decodePiece :: Char -> Piece
decodePiece 'p' = Pawn
decodePiece 'N' = Knight
decodePiece 'B' = Bishop
decodePiece 'R' = Rook
decodePiece 'Q' = Queen
decodePiece 'K' = King

isPiece, isRow, isCol :: Char -> Bool
isPiece p = p `elem` "pNBRQK"
isRow    r = r `elem` "12345678"
isCol    c = c `elem` "abcdefgh"

isCoords :: Char -> Char -> Bool
isCoords c r = isCol c && isRow r

```

Разбор всех прочих вариантов записи хода и реализация функции `parseExtras :: String -> [Extras]` предлагается читателю в качестве упражнения для самостоятельного выполнения. Авторский вариант программы, производящей синтаксический анализ файла формата PGN и распечатывающей полученное внутреннее представление можно найти на диске, прилагаемом к данной книге.

На первый взгляд кажется, что выбранный подход достаточно хорош для применения в любых практических задачах. Некая громоздкость кода функций `parseMove` и `parseExchanges` представляется разумной платой за сложность входного формата, но в целом читаемость или эффективность кода выглядит близкой к оптимальной. Значит ли это, что самый простой и очевидный подход к задаче синтаксического анализа является одновременно и самым правильным?

В следующей главе будет сделана попытка расширить полученный парсер для обработки более сложных конструкций формата PGN, и на этом примере будет показано, что сложность реализации примитивного подхода к синтаксическому анализу растёт непропорционально усложнению анализируемого формата. Будет рассмотрен ряд механизмов и методов, которые позволяют создавать серьёзные синтаксические анализаторы для непростых форматов выходных данных.