

하스켈로 배우는 프로그래밍

Programming in Haskell

Graham Hutton 지음
안기영, 박정훈 옮김
우균 감수

March 14, 2015

For Annette, Callum and Tom

목차

머리말	i
옮긴이 머리말	iii
추천사	xii
1 소개	28
1.1 함수	28
1.2 함수형 프로그래밍 functional programming	30
1.3 하스켈의 특징	33
1.4 역사적 배경	35
1.5 하스켈 맛보기	36
1.6 살펴보기	40
1.7 연습문제	41
2 첫걸음 빼기	42
2.1 Hugs 시스템	42
2.1.1 GHC 설치방법	43
2.1.2 Hugs 설치방법	47
2.2 표준 서막 standard prelude	49
2.3 함수 적용	54
2.4 하스켈 스크립트	54
2.5 살펴보기	63
2.6 연습문제	64
3 타입과 클래스	65
3.1 기본 개념	65
3.2 기본 타입	68
3.3 리스트 타입	72
3.4 순서쌍 타입	74
3.5 함수 타입	75
3.6 커리된 curried 함수	77
3.7 여러모양 polymorphic 타입	80
3.8 여러의미 overloaded 타입	82
3.9 기본 클래스	83
3.10 살펴보기	96
3.11 연습문제	97
4 함수 정의	98
4.1 새것을 옛것으로부터	98
4.2 조건식 conditional expression	100
4.3 보초선 등식 guarded equation	101
4.4 패턴 매칭 pattern matching	102

4.5 람다식 lambda expression	108
4.6 잘린식 section	111
4.7 살펴보기	112
4.8 연습문제	113
5 리스트 조건제시식 comprehension	114
5.1 생성원 generator	114
5.2 보초 guard	117
5.3 zip 함수	120
5.4 글줄 string 조건제시식 comprehension	122
5.5 카이사르 암호 Caesar cipher	123
5.6 살펴보기	132
5.7 연습문제	133
6 되도는 함수 recursive function	135
6.1 기본 개념	136
6.2 리스트로 되돌기	138
6.3 인자가 여럿일 때	142
6.4 여러 갈래로 되돌기	144
6.5 서로 부르며 되돌기 mutual recursion	145
6.6 되도는 함수 정의를 위한 도움말	147
6.7 살펴보기	154
6.8 연습문제	155
7 함수를 주고받는 higher-order 함수	157
7.1 기본 개념	157
7.2 리스트 다루기	159
7.3 foldr 함수	163
7.4 foldl 함수	167
7.5 함수 합성 연산자	169
7.6 글줄 string 전송기	171
7.7 살펴보기	178
7.8 연습문제	179
8 함수형 문법 분석기 functional parser	181
8.1 문법 분석기 parser	181
8.2 문법 분석기 타입	182
8.3 기본 문법 분석기	184
8.4 순서대로 엮기 sequencing	186
8.5 선택 choice	188
8.6 간단한 문법 분석기 이끌어내기	189
8.7 빙칸 처리	194
8.8 산술식 문법 분석하기	200
8.9 살펴보기	207
8.10 연습문제	208
9 대화식 interactive 프로그램	210
9.1 대화 interaction	210
9.2 입출력 타입	211
9.3 기본 동작 basic action	213
9.4 순서대로 엮기 sequencing	214
9.5 간단한 동작 이끌어내기	216
9.6 계산기	219
9.7 생명 게임 game of life	230
9.8 살펴보기	240

9.9 연습문제	241
10 타입과 클래스 선언	242
10.1 타입 선언	242
10.2 데이터 선언	244
10.3 되도는 타입	247
10.4 늘 침tautology 검사기	251
10.5 추상 기계abstract machine	260
10.6 클래스와 인스턴스 선언	264
10.7 살펴보기	269
10.8 연습문제	270
11 카운트다운 문제 countdown problem	272
11.1 소개	272
11.2 문제를 수식으로 정리하기	274
11.3 짐승같이 무식한brute-force 풀이	278
11.4 생성generation 하면서 계산evaluation 하기	280
11.5 대수적 성질을 이용하기	282
11.6 살펴보기	291
11.7 연습문제	292
12 느긋한 계산법lazy evaluation	293
12.1 소개	293
12.2 계산 방식evaluation strategy	295
12.3 끝남termination	299
12.4 줄이기reduction 횟수	300
12.5 무한 구조infinite structure	302
12.6 모듈 방식modular 프로그래밍	304
12.7 깔깐한 적용strict application	308
12.8 살펴보기	311
12.9 연습문제	312
13 프로그램에 대한 논리적 증명	314
13.1 등식 바탕의 논증equational reasoning	314
13.2 하스켈 프로그램에 대한 논증	316
13.3 간단한 보기들	317
13.4 자연수에 대한 귀납법	319
13.5 리스트에 대한 귀납법	323
13.6 이어붙이기 연산자 없애기	326
13.7 번역기 정확성compiler correctness	331
13.8 살펴보기	338
13.9 연습문제	339
A 표준 서마standard prelude	341
A.1 클래스class	341
A.2 논리값logical value	343
A.3 글자character 와 글줄string	344
A.4 수number	346
A.5 순서쌍tuples	347
A.6 아마도Maybe	348
A.7 리스트list	348
A.8 함수function	353
A.9 입출력input/output	354
B 기호표 및 Hugs 명령	356

C 번역 용례	358
참고문헌	360

머리말

... 소프트웨어를 설계하는 두 가지 방법이 있다. 하나는 매우 간단하게 만들어 흄이 없음을 대번에 알 수 있도록 하는 것이고, 다른 하나는 매우 복잡하게 만들어 대번에 알아볼 수 있는 흄이 없도록 하는 것이다. 첫 번째 방법이 훨씬 더 어렵다.

Tony Hoare, 1990,
컴퓨터학회연합 튜링 상(ACM Turing Award) 강연

이 책은 프로그램을 간결하고 명확하며 우아하게 짜는 방법을 알아보는 책이다. 더 구체적으로는 하스켈(Haskell)을 통해 함수형 프로그래밍(functional programming) 방식을 소개한다.

함수형 프로그래밍은 Java, C, C++, Visual Basic 등 요즘 쓰는 대부분의 언어에서 추구하는 것과는 상당히 다른 방식이다. 요즘 쓰는 언어들은 대개 그 밑에 있는 하드웨어와 밀접한 관계를 갖는데, 이는 저장된 값을 바꿔 나간다는 생각을 바탕으로 프로그램을 짠다는 점에서 그러하다. 반면, 하스켈은 더 추상적인 방식으로 프로그래밍하는 것을 추구하는데, 이는 함수를 인자에 적용한다는 생각에 바탕을 두고 있다. 앞으로 살펴보겠지만, 더 높은 차원으로 옮겨가면 프로그램이 훨씬 간결해지며 여러 가지 강력하고 새로운 방법으로 프로그램을 구성하고 그에 대해 추론할 수 있다.

이 책은 컴퓨터 과학을 공부하는 대학교 학부생을 염두에 두고 쓴 것이지만 하스켈로 프로그래밍을 공부하고자 하는 다른 독자들이 읽기에도 적당하다. 이전에 프로그램을 전혀 해 보지 않았더라도 이 책을 읽어나가는 데 무리가 없도록 모든 개념을 처음부터 차근차근 엄선된 보기를 곁들여 설명해 나간다.

책에서 다루는 하스켈 언어는 최근¹ 발표된 언어 표준인 하스켈 98로서, 지난 15년간 하스켈을 설계한 사람들의 노력의 결정체다. 이 책은 입문서이기에, 하스켈 및 관련 라이브러리의 모든 면면을 다루지는 않는다. 책의 절반 정도는 이 언어의 주요 특징을, 나머지 절반은 하스켈로 프로그래밍하는 보기와 사례 연구를 다룬다. 매장마다 연습문제가 있으며, 더 높은 수준의 전문적 주제에 대한 읽을거리도 추천한다.

이 책은 여러 해 동안 노팅엄 대학(University of Nottingham)에서 강의하는 동안 다듬은 교재의 내용을 바탕으로 하고 있다. 20시간 분량의 강의와 약 40시간 분량의 개인 공부, 실습 및 숙제로 책의 대부분을 소화할 수 있다. 하지만 책의 뒷부분에 있는 몇몇 보기 프로그램들까지 다루려면 시간이 더 많이 들 수도 있다.

책의 웹사이트²에 각 장에 대한 파워포인트 슬라이드와 각각의 큰 보기에 대한 하스켈 코드들 및 여러 종류의 참고자료가 올라와 있다. 강사들은 solutions@cambridge.org로 이메일을 보내 각 장의 끝에 있는 연습문제 해답과 시험문제 모음 및 모범답안을 얻을 수 있다.

고맙습니다

노팅엄 대학 프로그래밍 기초론(Foundations of Programming) 모임은 함수형 프로그래밍에 대해 연구하고 가르치기에 아주 좋은 환경을 제공해 주었다. 이 책을 쓸 수 있도록 안식년을 준 대학과, 내 하스켈 과목에 대한 의견을 제시해 준 모든 학생들과 조교들, 함수형 프로그래밍에 관한 아이디어 및 그것을 어떻게 설명할지에 대한 많은 토론을 나눈 프로그래밍 기초론 그룹의 Thorsten Altenkirch, Neil Ghani, Mark Jones(현재 포틀랜드 주립대학Portland State University 소속), Conor McBride, Henrik Nilsson에게 고마움을 표하고 싶다.

또한 캠브리지 대학 출판사(Cambridge University Press)에서 편집을 맡은 David Tranhah과 Dawn Preston, 하스켈 인터프리터인 Hugs를 만든 Mark Jones, 하스켈 조판을 위한 lhs2TeX 시스템을 만든 Ralf Hinze와 Anders Löh, 책의 초고를 강의에 써 본 후 의견을 들려준 Rik van Geldrop와 Japp van der Woude, 오류를 지적해 준 Kees van den Broek, Frank Heitmann, Bill Tonkin, Ian Bayley, 그리고 책을 검토하며 도움을 준 익명의 여러분들과, 또 카운트다운 프로그램과 관련하여 Jel Wright에게 감사한다.

Graham Hutton
2006년 노팅엄에서

¹옮긴이 주: 하스켈 98 표준은 2003년 발표되었다. 독자들 중 젊은 학생들이나 매년 급변하는 IT 업계에 종사하는 전문가들은 벌써 5년 전을 사건을 최근이라고 하는 것이 어색하겠지만 오랫동안 하스켈 관련 연구를 한 저자 입장에서는 이를 최근 사건으로 여길 수 있다. 하스켈 프로그래머들은 이 때 발표된 하스켈 98 표준을 안정된 표준으로 생각한다.

²옮긴이 주: 우리말 판 웹사이트는 <http://pl.pusan.ac.kr/~haskell/> 영문 원서 홈페이지는 <http://www.cs.nott.ac.uk/~gmh/book.html>

옮긴이 머리말(을 가장한 장광설)

이미 하스켈에 관심을 갖고 이 책을 선택한 분들에게 이 옮긴이 머리말은 그저 지은이 머리말과 책 1장 내용의 장황한 사족에 지나지 않을 것입니다. 특히 프로그래밍을 처음 배울 목적으로 이 책을 선택하신 분들은 이 장황한 옮긴이 머리말을 건너뛰셔도 좋습니다. 그럼에도 굳이 옮긴이로서 머리말을 쓰는 까닭은, 현업 IT 전문가의 입장에서 하스켈이라는 프로그래밍 언어가 어떤 의미로 다가올 수 있을지 좀더 구구절절한 이야기를 풀어놓고 싶기 때문입니다.

하스켈은 원래 더 나은 프로그래밍 언어를 고안하려는 프로그래밍 언어 연구자들이 느긋한 계산법을 따르는 순수 함수형 언어를 표준화함으로써 연구자들끼리 아이디어를 더욱 효과적으로 교류하려는 연구 목적으로 만들어지기 시작한 언어입니다.³ 그래서 하스켈은 자연히 프로그래밍 언어 분야의 연구 성과를 충실히 반영하도록 설계되었고, Hugs나 GHC와 같은 하스켈 구현에 꾸준히 연구자들이 참여하여 하스켈 98 표준 이외에도 최근의 프로그래밍 언어 연구 결과를 적용한 여러가지 확장 기능을 추가로 제공하고 있습니다. 이러한 설계와 구현은 언어 자체의 학술적 심미성으로만 그치지 않았으며, 안목 있는 IT 전문가들의 눈에 하스켈을 빼놓고 정확하게 소프트웨어를 개발할 가능성을 열어주는 프로그래밍 언어로 돋보이게 했습니다. 이런 IT 전문가들이 업무에 하스켈을 도입했고 그 과정에서 축적된 경험의 일부를 오픈 소스로 공개하여 활발한 하스켈 개발자 공동체를 이루기에 이르렀습니다. 그리하여 하스켈은 오늘날 연구자들에게 각광받는 학술적 프로그래밍 언어인 동시에 범용적인 프로그램을 빼놓고 정확하게 작성할 수 있는 실무적 프로그래밍 언어로 발전하였습니다.

실무적인 범용 언어로서의 하스켈

하스켈은 군용 보안 소프트웨어 작성 (예: Galois), 금융상품 분석 (예: Credit Suisse), 하드웨어 설계 (예: Bluespec), DNA 및 고분자 화합물을 이용한 신약 개발 (예: Amgen) 등 다양한 업계에서 이미 많은 이윤을 창출하는 상용 소프트웨어의 핵심 기술 구현에 요긴하게 쓰이고 있습니다. 예로 든 대표적인 기업 외에도 매년 함수형 프로그래밍 국제 학회(The ACM SIGPLAN International Conference on Functional Programming)와 함께 열리는 함수형 프로그래밍 사용 기업(Comercial Users of Functional Programming) 모임이 최근 몇 년간 성황을 이루며 하스켈을 비롯한 함수형 언어를 업무에 성공적으로 도입한 기업들의 사례 보고가 잇따르고 있는데, 그 중 하스켈을 업무에 쓰는 기업들의 목록⁴이 하스켈 홈페이지에 정리되어 있습니다. 그리고 하스켈을 업무에 도입하려는 기업에 자문을 제공하는 컨설턴트들⁵도 생겨나고 있습니다.

하스켈 오픈 소스 소프트웨어 공동체의 활동 또한 요즘 들어 더욱 활기를 띠고 있습니다. 널리 쓰이는 스크립트 언어인 펄의 차세대 판 Perl 6 구현 Pugs⁶, 패치 이론 바탕의 분산 버전 관리 시스템 Darcs⁷, 그리고 하스켈 개발에 가장 많이 쓰이는 글래스고우 하스켈 컴파일러⁸ (Glasgow Haskell Compiler, GHC) 또한 하스켈로 개발이 진행되는 오픈 소스 프로젝트입니다. 이런 굽직한 프로젝트 외에도 각종 하스켈 라이브러리 및 응용프로그램, 그리고 C 등 다른 언어로 작성된 기존 라이브러리와의 연동 등 요긴한 중소 규모 프로젝트가 활발히 진행되고 있습니다. 아침 리눅스는 현재 500개 이상의 하스켈 관련 배포판 패키지를 제공하고 있으며⁹, 데비안이나 우분투 리눅스도 약 250개 가량의 배포판 패키지를 제공하고 있습니다. 이렇게 하스켈 오픈 소스 개발 활동이 활기를 띠는 것은 최근 몇 년간에 걸쳐 BSD의 port 패키지 관리 시스템과 유사한 하스켈 패키지 관리 시스템인 Cabal¹⁰이 하스켈 라이브러리와 프로그램을 배포하는 사실상의 표준으로 자리잡도록 했으며, 펄의 CPAN과 같은 Hackage¹¹라는 하스켈 패키지 저장소를 중심으로 하스켈 개발자들이 만든 오픈 소스 패키지들을 한데 모아 배포하는 등 하스켈 개발자 공동체의 노력이 모인 결과입니다.

참고로 이 책은 처음 프로그래밍을 배우는 분들도 읽을 수 있도록 구성된 프로그래밍 입문서이므로 하스켈을 실무에 응용하는 보기로 다루지는 않습니다. 이 책을 읽고 하스켈을 실무에 응용하는 데 관심을 갖게 될 IT 전문가들께는 최근 출간된 “Real World Haskell”¹²을 이 책 다음으로 살펴볼 것을 추천합니다. 그 책은 실제로 하스켈을 업무에 쓰는 개발자들이 자신들의 경험을 정리해 놓은 책으로 데이터베이스, 웹, 시스템, GUI, 네트워크, 병렬 프로그래밍 등을 아우르는 실무적인 예제를 포함합니다.

³ <http://research.microsoft.com/~simonpj/papers/history-of-haskell/>

⁴ http://www.haskell.org/haskellwiki/Haskell_in_industry

⁵ <http://www.haskell.org/haskellwiki/Consultants>

⁶ <http://www.pugscode.org/>

⁷ <http://darcs.net/>

⁸ <http://haskell.org/ghc/>

⁹ <http://kldp.org/node/97330>

¹⁰ <http://www.haskell.org/cabal/>

¹¹ <http://hackage.haskell.org/>

¹² <http://www.realworldhaskell.org/>

프로그래밍 언어 연구 성과를 충실히 반영한 하스켈의 우수성

그렇다면 여러 분야에서 업계를 선도하는 기업의 IT 전문가들이 더 잘 알려진 다른 프로그래밍 언어들을 두고 왜 굳이 하스켈을 업무에 도입했을까요? 업계마다 기업마다 업무가 판이하게 다르기 때문에 정확히 어떤 이유라고 억측할 수는 없습니다. 하지만 우리가 하스켈을 써 보면서 발견한 것과 마찬가지로 그들도 하스켈을 쓰면서 다른 프로그래밍 언어와 대조되는 장점을 발견하여 하스켈에 매료되었음에 틀림없습니다. 옮긴이들의 제한된 경험으로 하스켈의 면면을 모두 비출 수는 없겠지만, 우리가 하스켈을 직접 쓰면서 그리고 하스켈을 도입하여 성공한 프로젝트들을 접하면서 알게 된 바를 간단히 정리해 보았습니다.

효율성 높은 컴파일 언어이면서도 스크립트 언어처럼 캡싼 하스켈

하스켈은 스크립트 언어의 장점과 컴파일 언어의 장점을 동시에 갖는 언어입니다. 스크립트 언어는 개발을 잽싸게 진행할 수 있다는 장점을 내세우고 있습니다. 이는 스크립트 언어가 대개 고급언어를 직접 실행하는 해석기(interpreter)를 바탕으로 하기 때문인데, 프로그램을 수정한 후 바로 해석기에 불러들여 그 수정한 부분만 따로 검사하기에 매우 편리합니다. 하지만 해석기를 바탕으로 한 언어는 번역기(compiler)로 미리 저급 언어로 번역한 다음 실행하는 컴파일 언어보다 효율이 확연히 떨어지기 때문에 반복적인 수치연산 등 계산량이 많은 코드를 직접 작성하기에 알맞지 않다는 단점이 있습니다. 반면 컴파일 언어는 스크립트 해석기로 돌리는 것보다 훨씬 효율이 뛰어난 실행파일을 생성할 수 있다는 것이 장점입니다. 그러나 C와 같은 컴파일 언어는 작은 부분을 고치더라도 전체 프로그램을 실행 파일로 다시 연결한 다음 돌려보아야 하며 프로그램 시작 지점부터 항상 실행을 시작하기 때문에 수정한 부분만 따로 떼어 검사하기 번거롭다는 단점이 있습니다. 하스켈과 같은 함수형 언어의 경우 프로그래밍 언어 연구자들이 이미 수십 년 전부터 확립한 해석기와 번역기의 경계를 넘나드는 기술을 이전부터 구현하고 있었습니다. 그렇기 때문에 마치 스크립트 언어처럼 캡싼 개발도 가능하고 저급 언어로 번역된 효율성 높은 실행 파일도 얻을 수 있습니다. 최근에는 고급 언어와 그를 번역한 저급 언어를 연동하는 기술이 함수형 언어 구현 뿐 아니라 가상머신 기반의 언어나 스크립트 언어가 가지는 효율상의 한계를 극복하기 위한 JIT 번역(just-in-time compile) 등에도 응용되고 있습니다.

많은 설명보다 간단한 보기로 대표적인 스크립트 언어인 파이썬과 하스켈을 비교해 보면 하스켈 역시 캡싼 개발을 지원하는 언어라는 사실이 와닿을 것입니다. 예컨대, 파이썬 스크립트 `test.py`에 제곱을 구하는 함수를 아래와 같이 잘못 정의한 다음

```
def square(x): return x + x
```

해석기에서 불러들여 함수를 시험해 보면 잘못된 결과를 얻습니다.

```
>>> import test  
>>> test.square(3)  
6
```

이제 `test.py`의 `square` 함수 정의를 다음과 같이 바르게 고친 다음

```
def square(x): return x * x
```

해석기에서 다시 불러들이면 바뀐 함수 정의대로 올바른 결과를 얻습니다.

```
>>> reload(test)  
<module 'test' from 'test.py'>  
>>> test.square(3)  
9
```

이렇듯 캡싸게 오류를 수정하고 스크립트를 다시 불러들여 바로 시험해 볼 수 있다는 장점이야말로 스크립트 언어가 각광받게 된 가장 큰 이유라고 생각합니다. 하스켈과 같은 함수형 언어의 대화식 환경¹³에서도 이와 마찬가지로 개발을 캡싸게 진행할 수 있습니다. 하스켈 스크립트 `Main.hs`에 제곱을 구하는 함수를 다음과 같이 잘못 정의한 다음

```
square x = x + x
```

GHC의 대화식 환경인 `ghci`에서 불러들여 함수를 시험해 보면 잘못된 결과를 얻습니다.

¹³ 대화식 환경을 해석기라고 부르지 않는 이유는 대화식 환경도 엄밀히 말하자면 번역기이기 때문입니다. 고급 언어를 직접 실행하는 해석기와는 달리 대화식 환경을 통해 불러들인 스크립트는 하스켈과 기계어의 중간 단계인 바이트코드로 번역됩니다.

```
Prelude> :load Main
Main> square 3
6
```

이제 *Main.hs*의 *square* 함수 정의를 다음과 같이 바르게 고친 다음

```
square x = x * x
```

다시 불러들이면 바뀐 함수 정의대로 올바른 결과를 얻습니다.

```
Main> :reload
Main> square 3
9
```

하지만 GHC에는 대화식 환경 뿐만 아니라 ghc 번역기가 있으므로 완전히 기계어로 번역된 실행 파일을 얻을 수도 있다는 것이 일반적인 스크립트 언어와 차별화되는 점입니다. 다음과 같이 *Main.hs*에 프로그램 시작을 나타내는 간단한 *main* 함수를 추가한 후

```
square x = x * x
main = print (square 3)
```

GHC 번역기로 실행 파일을 만들어 돌려볼 수 있습니다.

```
$ ghc Main.hs
$ ./a.out
9
```

안전한 정적 타입 언어이면서도 동적 타입 언어처럼 유연한 하스켈

하스켈은 안전한 정적 타입 시스템을 바탕으로 하고 있으면서도 동적 타입 언어에 가까운 유연성을 갖는 언어입니다. 그 이유를 전문 용어를 동원해 설명하자면 하스켈은 Hindley-Milner 타입 시스템을 바탕으로 할 뿐 아니라, 타입 클래스라는 독특한 기능을 갖고 있기 때문이라 할 수 있습니다. Hindley-Milner 타입 시스템을 바탕으로 하는 언어는 타입 유추(type inference)를 바탕으로 인자 여러모양새(parametric polymorphism)를 자연스럽게 구사함으로써 동적 타입 언어처럼 유연하게 포괄적인 코드 작성이 가능합니다. 타입 클래스는 여러 의미(overloaded) 함수를 Hindley-Milner 타입 시스템과 잘 어우러지면서도 깔끔하게 정의하기 위해 하스켈에서도 입한 독특한 기능으로, 물건 중심 언어의 인터페이스(interface)와 비슷하지만 그보다 더 유연하고 확장성이 있습니다.

여러모양새 (polymorphism)란 정적 타입 언어에서 타입에 따른 코드 중복을 줄이고자 함수를 한 번만 정의하여 여러모양으로 쓸 수 있도록 하는 기능을 일컫는데, 그 방법에 따라 여러 종류가 있습니다. 함수형 언어에서는 대개 인자 여러모양새를 구사하며 물건 중심(object oriented) 언어에서는 대개 하위타입 여러모양새(subtype polymorphism)를 구사합니다. 대표적인 물건 중심 언어인 자바(Java)에서 아래와 같이 입력 스트림으로부터 내용을 읽어들이는 *getContents* 함수를 한 번만 정의하면

```
String readContents(InputStream is) { /* ... */ }
```

인자 타입으로 선언한 *InputStream* 타입의 물건(object)인 *System.in*은 물론 그 하위 타입인 *FileInputStream*이나 *DataInputStream* 타입의 물건인 *fis*나 *dis*에도 적용할 수 있는데

```
FileInputStream fis;
DataInputStream dis;
// ...
String s1 = readContents(System.in); // InputStream
String s2 = readContents(fis); // FileInputStream
String s3 = readContents(dis); // DataInputStream
```

이러한 기능을 바로 하위타입 여러모양새라고 합니다. 한편, 현대적인 타입 시스템을 갖춘 함수형 언어인 하스켈에서는 아래와 같이 *length* 함수를 한 번만 정의하면

$$\begin{aligned} \text{length} [] &= 0 \\ \text{length} (x : xs) &= 1 + \text{length} xs \end{aligned}$$

정수, 글자 등 아무 타입 원소를 갖는 리스트에 모두 적용할 수 있는데

```
Main> length [2,3,5,7] -- [Int] (정수 리스트)
4
Main> length ['a','b','c'] -- [Char] (글자 리스트)
3
Main> length [[],[]] -- [[Int]] (정수 리스트의 리스트)
2
```

이러한 기능을 바로 인자 여러모양새라고 합니다.

과거에 인자 여러모양새를 직접 구사할 수 없었던 정적 타입 물건 중심 언어들이 다른 방법으로 예둘러 인자 여러모양새를 시늉내느라 큰 불편을 겪다 결국 포괄적 프로그래밍(generic programming)을 지원하는 언어 기능을 덧붙일 수밖에 없었습니다. 초기 C++의 경우, 인자 여러모양새를 흉내내기 위해 C에서처럼 조악한 전처리 매크로나 타입 안정성이 전혀 보장되지 않는 *void**를 거치는 임의적 형변환 방식에 의존해야 했습니다. 이후 C++ 표준 라이브러리를 제대로 설계하면서 템플릿을 언어에 추가하여 템플릿 메타프로그래밍으로 인자 여러모양새를 직접 표현할 수 있게 됩니다. Java의 경우 모든 물건들의 최상위 타입인 *Object*에 대한 하위타입 여러모양새를 이용하되 필요에 따라 동적으로 *Object* 타입으로부터 원래의 상위 타입으로 강제 변환을 하는 방식을 쓸 수밖에 없었습니다. C의 *void**보다야 낫지만 동적인 하위 타입으로의 강제 변환 역시 타입 안전성을 보장하지 못하므로 정적 타입 언어의 장점을 거스릅니다. 결국 Java도 뒤늦게 제너릭을 추가면서 타입을 검사하고 나중에 지우는 방식으로 인자 여러모양새를 직접 표현할 수 있게 됩니다. 이러한 역사적 사실을 종합해 보면 C++ 템플릿이나 Java 제너릭과 같은 포괄적 프로그래밍의 유래가 바로 탄탄한 이론을 바탕으로 설계된 정적 타입 함수형 언어에서 일상적으로 사용하던 인자 여러모양새라는 것을 알 수 있습니다.

함수형 언어의 인자 여러모양새와 물건 중심 언어의 포괄적 프로그래밍이 다른 점은 함수형 언어에서는 인자 여러모양새가 걸음마 프로그래머도 구사하는 기본 초식인 반면 물건 중심 언어에서는 포괄적 프로그래밍을 상대적으로 많은 노력을 요하는 고급 기술로 여긴다는 것입니다.

물건 중심 언어에서 포괄적 함수나 클래스를 작성하려면 보통 함수나 클래스를 작성할 때와 달리 포괄적 인자를 별도로 표시하는 등의 노력이 추가로 필요합니다. 그렇기에 C++나 Java와 같은 언어로는 라이브러리 설계자라면 여유를 두고 포괄적 프로그래밍을 잘 지원하는 라이브러리를 만들 수 있을 것입니다. 하지만 상대적으로 신속한 개발을 요하는 응용프로그램 개발자라면 이미 만들어진 포괄적 라이브러리를 즐겨 쓸지는 몰라도 포괄적 프로그래밍의 장점을 살리는 코드 작성을 선뜻 내켜하지 않을 가능성이 높습니다. C++나 Java를 다루는 책들도 대개 포괄적 프로그래밍을 고급 기법으로 분류하곤 합니다. 정적 타입 물건 중심 언어에만 익숙했던 개발자들이 파이썬과 같은 동적 타입 스크립트 언어에 열광한 또 하나의 이유가 바로 이것이라 생각합니다. 동적 타입 스크립트 언어는 타입을 무시하고 상대적으로 적은 노력으로 포괄적 프로그래밍을 할 수 있습니다. 그래서 이런 스크립트 언어가 유행한 후 동적 타입 언어만이 유연하며 정적 타입 언어는 유연하지 않다는 생각이 퍼지게 되었습니다.

그러나 현대적 타입 시스템을 갖춘 함수형 언어를 접해 보았다면 정적 타입 언어가 유연할 수 없다는 것은 설부른 편견이 아닌가 의문을 제기할 수밖에 없을 것입니다. 왜냐하면 앞서 살펴본 *length* 함수처럼 하스켈에서는 타입 정보를 전혀 표시하지 않고도 여러모양 함수를 작성할 수 있으며 타입 시스템이 자동으로 그 타입을 유추해 주기 때문입니다. 함수형 언어를 전혀 접해 보지 않은 개발자가 *length*의 정의를 보면 하스켈을 동적 타입 스크립트 언어로 오해할 정도입니다. 이렇듯 하스켈로 프로그래밍을 배우기 시작하면 인자 여러모양새를 자신도 모르게 구사하며 유연한 포괄적 프로그래밍을 하게 된다는 것을 이 책을 통해 발견하게 될 것입니다. 그리고 비록 당장의 업무에 하스켈을 도입하지 않더라도 하스켈을 익히면서 물건 중심 언어에서 고급 기법으로 여기는 포괄적 프로그래밍을 자신있게 구사할 내공을 쌓는 재미 또한 쏠쏠할 것입니다.

타입 클래스 (type class)란 말 그대로 타입 (type)의 분류 (class)로 덧셈이나 곱셈과 같이 모든 타입에 다 적용 가능한 여러모양 함수는 아니지만 공통점이 있는 많은 타입에 적용 가능한 함수의 타입을 딱 떨어지게 정의하기 위해 하스켈에서 도입한 독특한 기능입니다. 물건 중심 언어의 인터페이스와 유사한 개념이지만 그보다 더 확장성이 있다는 점에서 뛰어나다는 것을 하스켈의 타입 클래스와 자바의 인터페이스를 비교함으로써 알아보기로 합시다. 하스켈의 타입 클래스와 Java의 인터페이스를 비교하기 전에 한 가지 짚고 넘어갈 것은, 물건 중심

언어에서 클래스라 불리는 기능은 같은 이름이지만 완전히 다른 개념이라는 것입니다. Java와 같은 물건 중심 언어에서는 클래스 정의가 곧 하나의 타입에 대응되지만 하스켈에서 타입 클래스는 타입을 모아 놓은 집합을 대표한다고 생각하면 이해하기 좋습니다.

하스켈의 타입 클래스는 물건 중심 언어에서 주로 쓰는 인터페이스라는 기능과 비슷합니다. 예를 들면 Java에는 `java.lang.Comparable<T>`라는 인터페이스가 있는데 이것이 바로 하스켈의 `Ord` 클래스에 해당한다고 볼 수 있습니다. 예컨대, Java에서는 `MyType`이라는 새로운 타입을 정의하면서 다음과 같이 인터페이스를 상속받아 `compare` 메서드를 구현합니다.

```
// Java 인터페이스 정의
interface Comparable<T> {
    int compareTo(T x);
    // ...
}

// 타입을 정의하며 인터페이스 상속
public class MyType implements Comparable<T> {
    public int compareTo(MyType x) {
        // ... 메서드 구현 ...
    }
    // ...
}
```

하스켈도 이와 비슷하게 새로 정의한 `MyType` 타입을 `Ord` 클래스의 인스턴스로 선언하고 메서드를 구현합니다.

```
-- 하스켈 타입 클래스 정의
class Ord a where
    compare :: a → a → Ordering
    (<), (≤), (≥), (>) :: a → a → Bool
    max, min :: a → a → a

-- 타입 정의
data MyType = -- ...

-- 타입 정의와는 별도로 인스턴스 선언
instance Ord MyType where
    compare x y = -- ... 메서드 구현 ...
```

여기서 눈여겨 볼 점은 Java 인터페이스와 하스켈 타입 클래스의 차이점입니다. Java에서는 인터페이스 상속이 타입 정의의 일부분이지만 하스켈에서는 타입을 정의한 다음 별도로 인스턴스 선언을 합니다. 작은 차이 같지만 타입 정의에 종속된 Java 인터페이스에 비해 타입 정의로부터 자유로운 하스켈 타입 클래스는 월등한 확장성을 갖습니다. 하스켈 타입 클래스는 타입을 정의한 후 언제라도 인스턴스 선언을 추가할 수 있습니다. 다른 모듈에 타입이 정의되어 있고 심지어 그 소스를 수정할 수 없는 라이브러리에 정의된 타입이라도 클래스 인스턴스로 선언하는 데는 아무 문제가 없습니다. 아직 하스켈에서 타입 정의와 타입 클래스가 느슨한 결합을 함으로써 갖는 장점이 실무에서 어떤 의미가 있을까 겨우뚱하는 분들을 위해 좀더 괴부에 와닿는 구체적인 상황을 상상해보겠습니다.

OO 기업 홍보부 지원 담당 데이터베이스 프로그래머 김대리는 Java에 능숙합니다. 홍보부에서 데 이터베이스 검색은 웹 인터페이스로 자동화되어 있지만 검색한 고객 정보로 통계를 내기 위해 검색 결과를 매번 스프레드시트로 일일이 옮겨적어야 하는 애로사항이 있었습니다. 김대리는 이를 자동화하는 프로젝트를 Java로 시작하여 `Xmlizable`이라는 인터페이스를 정의하고 데이터베이스 엔트리 등 각종 데이터 구조를 `Xmlizable` 인터페이스를 상속받아 엑셀에서 읽어들일 수 있는 XML로 변환하도록 깔끔하게 처리했습니다. 이후 추가되는 테이블 엔트리도 `Xmlizable` 인터페이스를 상속해 XML 변환 메서드만 정의하면 되는 확장성 있는 시스템을 구축했다는 생각에 김대리는 행복합니다. 이 일로 김대리는 홍보부에서 신임을 받게 되었고 부장님은 더 정확한 통계 분석을 위해 이전 텍스트 기반 시스템에 남아있는 데이터도 스프레드시트로 불러올 수 있게 통합하는 프로젝트를 김대리에게 맡깁니다. 이전 시스템을 관리하던 개발자는 한참 전에 그만뒀지만 이전 시스템도 Java로 작성되어 있고 소스도 남아있다니 다행이었습니다. 업무가 갑자기 늘어나긴 했지만 김대리에게 큰 문제는 아닙니다. 기존 시스템에서 사용하던 소스코드에서 필요한 타입들이 `Xmlizable` 인터페이스를 상속하도록 고쳐 XML 변환 메소드만 구현해 주면 되므로 늘상 하던 작업과 별다를 것이 없었습니다.

문제는 OO 기업이 XX 기업을 인수합병하면서부터 시작됩니다. XX 기업 홍보부에서 쓴 시스템을 통합하는 일이 김대리에게 떨어졌고 부장님은 당연히 김대리가 이번에도 금방 처리해 주겠거니 기대 합니다. 김대리도 XX 기업 역시 Java를 썼다는 보고에 따라 예전처럼 일정을 잡습니다. 그런데 XX 기업의 홍보부는 효율을 매우 중시해서 비공개 상용 라이브러리를 구입해 쓰고 있었으며 데이터도 일반 DBMS가 아닌 Berkeley DB에 들어있었던 것입니다. 이전처럼 인터페이스를 상속해 XML 변환 메소드만 추가하면 될 줄 알았는데, 소스도 없는 라이브러리라 인터페이스 추가는 언감생심입니다. 김대리는 임베디드 파일 DB 란 게 있다는 말만 들었지 한번도 다뤄 본 일이 없어 DBMS 수준의 데이터 통합을 하려 해도 일정을 맞출 자신이 없습니다. 인수합병 과정의 칼바람같은 구조조정으로 XX 기업 홍보부 전산팀은 온데간데 없고 회사 분위기도 위축 어수선해 어디 하소연할 데도 없습니다. 김대리는 분명 물건 중심 디자인 원칙을 충실히 따랐는데 뭐가 잘못됐는지 영문을 알 수 없습니다. 내키지는 않지만 밤샘을 해서라도 그 상용 라이브러리에서 필요한 모든 타입을 한 겁씩 감싸안은 데이터 타입을 한 번 새로 정의하는 방법밖에 없는 듯 합니다. 이런 걸 디자인 패턴에서 데코레이터(decorator) 패턴이라고도 한다니 그래도 아주 잘못된 방법은 아닐 것이라 애써 자위해 봅니다만 뭔가 찜찜하고 억울하다는 느낌은 지울 수 없습니다.

만일 OO 기업과 XX 기업이 하스켈을 쓰고 있었고 김대리가 *Xmlizable*이라는 타입 클래스를 쓰고 있었다면 어땠을까요? 김대리는 고민할 필요도 없이 예전에 하던 대로 XX 기업의 데이터 타입들을 *Xmlizable* 클래스의 인스턴스로 선언하여 XML 변환 메서드만 추가로 구현하기만 하면 될 것입니다. 왜냐하면 하스켈 타입 클래스는 타입 정의에 묶여 있지 않기 때문에 이전의 소스코드를 건드릴 필요가 없기 때문입니다.

마지막으로 하스켈의 타입 클래스와 Java의 인터페이스를 활용하는 보기자를 통해 하스켈 타입 클래스가 물건 중심 언어의 진화에 미친 영향을 짚어 보겠습니다. 하스켈에서 특정 클래스와 관련된 타입에만 제한적으로 적용 가능한 함수를 여러 의미 함수(overloaded function)라 합니다. 크기 비교가 가능한 타입으로 이루어진 리스트를 정렬해 주는 라이브러리 함수 *sort* 가 바로 대표적인 여러 의미 함수의 예입니다. ghci 대화식 환경에서 *sort*의 타입을 알아보면 다음과 같습니다.

```
List> :t sort
sort :: Ord a => [a] -> [a]
```

Ord a => 라는 클래스 제약이 있기 때문에 아무 리스트 *[a]* 에 다 적용 가능한 것이 아니라 타입 인자 *a* 가 *Ord* 클래스의 인스턴스일 때만 함수 적용이 가능합니다. Java에서도 하스켈의 *sort*에 상응하는 *sort* 함수를 다음과 같이 자바 제너릭 인자 *T*가 만족해야 할 인터페이스 제약 *extends Comparable<T>*를 표시하여 정의할 수 있습니다.

```
public static <T extends Comparable<T>>
List<T> sort(List<T> list) {
    // ... 아무 정렬 알고리듬으로 list를 정렬 ...
    return list;
}
```

이와 같이 제너릭과 인터페이스 제약을 함께 사용해 하스켈의 여러 의미 함수에 대응하는 자바 함수를 정의할 수 있는 것은 결코 우연이 아닙니다. 하스켈 타입 클래스 디자인에 핵심적 역할을 한 필립 와들러(Philip Wadler)라는 유명한 프로그래밍 언어 연구자가 이후 자바 제너릭을 디자인하는 데도 핵심적인 역할을 했기 때문입니다. 또한 참고로 현재 C++ 표준안(C++ 98)에는 Java의 인터페이스 제약 표시처럼 하스켈의 클래스 제약에 해당하는 개념을 직접 표현할 방법이 없었는데, 최근에 디자인된 컨셉(concept)이라는 기능이 새 C++ 표준의 초안(C++ 0x draft)에 들어갔다고 합니다. 따라서 다음 C++ 표준부터는 어떤 의미에서는 Java보다 하스켈 타입 클래스와 같은 개념을 더 잘 표현할 수 있는 강력한 기능이 추가될 것으로 기대하고 있습니다. 이와 관련해 더 자세히 알아보려면 필립 와들러 본인이 하스켈 타입 클래스와 자바 제너릭에 대해 구글에서 강연한 동영상¹⁴을 추천합니다.

감사합니다

우리말로 읊기는 동안 조언을 주신 Graham Hutton 교수님 (지은이), 감수해 주신 부산대학교 우균 교수님, 추천사를 써 주신 마이크로소프트 김재우 플랫폼 전략 부장(Platform Strategy Advisor) 님, 그리고 출판 기획을

¹⁴ 동영상 <http://video.google.com/videoplay?docid=-4167170843018186532>
슬라이드 <http://homepages.inf.ed.ac.uk/wadler/papers/oops1a/oops1a.pdf>

담당하신 대림출판사 박기덕 기획실장님과 수고하신 모든 대림출판사 직원분들, 그리고 초고를 다듬는 데 크나큰 도움을 주신 강성훈(하제: KAIST 게임 프로그래밍 동아리), 고우종(서울대학교 컴퓨터 공학 전공 대학생), 김보훈(충남대학교 컴퓨터 공학 전공 대학생), 김승범(Stanford University 컴퓨터 과학 전공 대학원생), 김정준(LIG 넥스원), 류기환(연세대학교 공대극회), 리종우((주)한글과 컴퓨터), 서상원(SNAGs: 서울대학교 컴퓨터 동아리, 졸업생), 심지웅(University of Pennsylvania 컴퓨터 과학 전공 대학원생) 정지용(NHN, 검색엔진 개발), 이대엽((주)넷스루, 개발자), 이상기(Mobile IT, 삼비안 스마트폰 S/W 개발), 전언섭(University of Waterloo 컴퓨터 과학 전공 대학생)님께 감사드립니다.

인사말

좋은 인연이 있었기에 이 책을 출간할 수 있었습니다. 뒤돌아 보면 십 년 전 한글 유즈넷에서 당시 대학원생이었던 김재우님으로부터 C++ 표준라이브러리의 포괄적 프로그래밍 패러다임이 하스켈과 같은 함수형 언어에서 비롯되었다는 이야기를 주워들은 것이 아마도 프로그래밍 언어에 관심있는 분들과 인연을 맺게 된 계기였습니다. 재작년 포틀랜드에서 ICFP라는 학회와 함께 열리는 하스켈 관련 행사 등을 문의하는 이메일을 통해 박정훈 형을 만나게 된 것이 이렇게 책을 읽기는 계기가 되리라고는 생각지도 못했습니다. 또한 책을 읽기며 Graham Hutton 교수님, 우균 교수님, 그리고 초고를 검토해 주신 분들과 새로운 인연도 생겼습니다.

미국으로 일터를 옮기느라 바쁜 중에도 출판사와 연락 등 한국에서 저 대신 궂은 일을 도맡아 해 주신 정훈이 형, 지금 공부할 수 있는 기초를 가르쳐 주신 이광근, 최광무, 한태숙, 박종철, 박우석 교수님을 비롯한 대학교 스승님들, 그리고 이곳에서 프로그래밍 언어를 같이 연구하는 Tim Sheard 교수님 및 여러 교수님들과 동료 대학원생들, 또한 멀리서도 항상 사랑으로 기도해 주시는 부모님과 가족들에게 고마움을 표하고 싶습니다. 이 책이 누군가에게 도움이 되고 새로운 인연을 열어 간다면 모든 사람의 마음과 마음 사이에 섭리하시는 하나님께 영광을 돌립니다.

2008년 11월 30일, 비오는 겨울의 오레곤 포틀랜드에서. 안기영

국내에 처음으로 하스켈 입문서를 번역 할 수 있어서 너무도 기쁩니다. 너무나도 부족한 저에게 함께 번역을 같이 하자고 제의를 해주신 ‘안기영’님께 정말 송구스럽고 고맙다는 말씀부터 전합니다. 안기영님이 다 차려주신 밥상에 그저 수저 하나만 더 올려 놓은 것밖에 없음에도 끝까지 같이 해주셔서 감사합니다. 번역이 쉽지 않은 일이었지만 안기영님과 스승님의 도움으로 잘 끝낼 수 있었습니다. 이번 번역을 계기로 무엇이 부족한지 어떤 면으로 더 공부해야 할지 깨닫는 좋은 시간이었습니다.

2008년은 저에게 많은 고난과 시련 속에서 스스로에게 길을 묻는 한해였습니다. 하지만 제 주위에는 저를 너무나도 아껴준 가족, 우리 지존나갈 친구들(대원, 상현, 남규, 민영, 영욱, 승하, 경민, 원준), 그리고 Irvine 친구들(용재, 한성, 문순, 창하, Nathan), Daizo's family(Noriko, Aki), 분당에서 고군분투중인 김재윤님, 그리고 Arizona의 든든한 형 Blaine에게 감사드립니다. 항상 격려와 질책으로 저를 잡아 준 MVP 정홍주 형, 짧은 시간이었지만 저에게 늘 따뜻한 마음을 보여주신 ‘강기하’님, 그리고 항상 저의 룰 모델이 되어 주신 스승님께도 진심으로 감사드립니다. 이 지면을 통해서 다 인사 드리지 못한 많은 분들께는 제가 직접 인사 드리겠습니다.

2008년 11월 30일, 비 내리는 뉴욕 밤하늘을 바라보며. 박정훈

“Haskell로 프로그램 짜보니까 어떤 느낌 들어요 ?” (추천사)

안기영씨에게 짤막한 서평 써달란 부탁을 몇 달 전에 받았지만, 이런 프로그래밍을 해온 지도 벌써 스무 해가 다 되어가는지라 왜 Haskell 같은 언어로 프로그램 짜는 법을 다시 배워야 하는지를 가슴 벅차게 전할만한 신선한 감정이 좀처럼 일어나질 않았다. 고민 끝에 같은 부서에서 일하는 김부장에게 물었다. 내 편에 넘어가서 처음으로 F#나 M 같은 언어를 쓰고 있고, 그 가운데 자연스럽게 Scheme이나 Haskell 프로그래밍도 맛보게 된 사람인데다, 지금까지 C#, Java, C/C++ 같은 언어로 십 수년간 프로그램 잘만 짜던 사람이라 그 견해나 평가가 궁금하던 차이기도 했다. 전화기 너머로 한참 뜬들이는 기척이 느껴지더니, 아주 간단한 답으로 이야기를 시작했다.

“아주 또렷하죠.” 맞다. 그랬다.

“그러니까, 풀려는 문제와 답을 쓰는 일이 아주 가깝다는 느낌이 들어요. 그런데도 자연스럽게 어떤 틀이 잡혀간다는 점이 달라요.”

“여태 쓰시던 프로그래밍도 표현력이나 … 특히 그저 어떻게 돌아가는 코드를 짜기보다 좋은 설계를 강조한다고 볼 수 있지 않을까요 ?”

“네, 뭐 그렇기는 하죠. 그런데 다른 점이 뭐냐 하면, 제가 보통 전체 프로그램의 틀을 한 눈에 파악하면서 일하는 것을 좋아하는데, 이 경우는 그런 목표를 이루는 게 훨씬 자연스럽다고 할까 … 보통 튼튼한 구조를 갖추려면 문제의 해답을 드러내는 코드보다 구조를 짜맞추는 코드가 늘어나는 경우가 다반사인데 … 이 경우는 그렇지 않아서, 문제 푸는 방법을 표현하는 데 애쓰다 보면, 그 과정에서 프로그램이 좋은 구조를 갖출 수 있게 된다는 것이 참 즐거운 경험이랄까 … 그러니까 예전에는 잘 설계된 프로그램을 짜려면, 설계 패턴(Design Pattern)도 정확히 이해하고, 문제도 정확히 이해해야 하는데, 그 둘이 완전히 나누어 져서 적지 않은 부담이 되거든요. 실제 시간이 촉박한 경우에는 뭐가 옳은지 알고는 있지만, 실천하기도 쉽지 않고 한데, 이번에는 문제 푸는 방식 그대로를 충실히 표현하려는 데 신경을 쓰고, 작은 크든 나눌 수 있는 것은 나누고 불필요한 제약을 차근차근 없애기만 하면, 틀이 잘 잡힌 프로그램을 이끌어낼 수가 있다는 거죠.”

이 대답에 마치 모르던 것을 알게 된 것처럼 연거푸 ‘그렇구나’하는 추임새를 넣어가며 얘기를 듣다가, 맨 처음 이런 언어로 프로그램을 배우게 되었을 때 느꼈을 법한 옛 느낌이 가물가물해서 뻔하다 싶은 질문을 이어 던졌다.

“처음 만져 봤을 때는 어땠어요? 받아들이기가 쉬웠어요?”

“에이, 아니죠. 처음에는 엄청 충격이었죠, 이건 뭐, 줄곧 상태를 바꾸어가며 프로그램 짜던 게 몸에 배인 사람인데, 일부 가능하다고는 하지만, 변수 값 그러니까 상태를 아예 바꾸지 않는 것이 이런 방식의 기본기라고 하니까 … 그런 이상한 (웃음) 생각으로 프로그램을 짜려니까 처음에는 참 …”

“이런 프로그래밍 방식을 처음 익히는 사람에게 어떤 얘기를 해주고 싶어요 ?”

왜 난데 없이 밤에 전화를 걸어, 이런 질문을 던지는지 앞뒤 연유를 찬찬히 설명한 다음 마지막으로 던진 질문이다.

“간단하죠. 알던 것은 잊어라. 그 편이 배우기 쉽다. 뭐 그런 말? (웃음)”

그럴 법도 하다. 나도 처음에는 그랬다. 하지만, 모두는 곧 알게 될 것이다. 이런 낯선 과정이 지나고 나면, 끝내 프로그램이란 것은 문제 푸는 방법을 옮겨 담는 일이고, 언어의 표현력이란 그 방법 그 자체를 있는 그대로 표현하는 데 얼마나 큰 공을 들이느냐로 판단할 수 있다는 사실을. 그에 이르면 문제를 보고 언어를 고르는 법이지, 문제를 언어에 맞추지 말아야 한다는 데 동의할 수 밖에 없다. 그리고 어는 순간 모든 게 같아진다. 알던 것이나 알게 된 것이나 모두 같게 보이는 때가 온다. 공부를 멈추지 않는다면 반드시.

이 책은 Haskell 이런 언어로 프로그램 짜는 법을 일러주는 책이다. 하지만, Haskell 이런 언어에만 집중하지 않았으면 한다. 프로그램 짜는 법을 공부할 때, 생각의 틀을 잡는 방법보다 언어 배우기에 골몰하는 것은 언제나 바람직하지 않다. 물론, Haskell 이런 언어가 돋보이기는 한다. 비할 데 드문 수준 높은 표현력을 갖추고 있으면서도, 언어의 알맹이는 지극히 작고 그 의미가 명료하여 애매하게 짠 프로그램에서 답이 나오는 경우가 드물고, 언어의 허점을 교활하게 활용하여 잔재주 부리는 코드를 쓰기가 쉽지 않다. 하지만, 이런 장점을 갖추고 이런 식으로 프로그램을 짜는 언어는 Haskell 하나가 아니다. 이와 견줄만한 언어도 얼마든지 많다. 다만, Haskell은 연구 공동체에 의하여 설계되고 실현되었으며, 다른 언어에 비해 좀더 많이 관심을 받고 널리 알려지는 운을 타고 났을 뿐이다. 따라서 이미 이런 프로그래밍에 익숙해진 사람이라면 굳이 이 책을 읽으라 강요하기는 어렵다. 물론, Haskell 이런 언어를 빨리 익히려 한다면 괜찮다.

하지만, 그와 달리, 지금 까지 이런 식으로 프로그램을 짜본 경험이 전혀 없었다면, 이 책에서 쓰고 있는 언어를 돋보이게 하지 않고서는 굳이 이런 책으로 공부할 까닭을 스스로 일깨우기가 쉽지는 않다. 왜냐면, 말은 생각을 드러내는 수단이기도 하지만, 동시에 생각을 가능하게 하는 수단이기도 하기에, 생각은 말에 같하고, 말은 다시 생각에 같히기 마련이기 때문이다. 언어에는 문화와 경험과 이론과 사상이 담겨있기에, 아무리 자유로운 생각도 그 표현 수단인 언어의 영향권을 벗어나기 어렵다. 다시 말해, 재료의 성질과 품질이 음식의 맛과 품질을 가늠하듯이, 생각의 색깔과 품질은 그 재료가 되는 언어의 장단점을 벗어나서 논하기 어렵다. 알맞은 재료, 싱싱한 재료는 좋은 음식을 만드는 데 기본이다. 프로그램도 마찬가지다. 물론 알던 것을 깨부수고, 새로운 것을 배우는 일은 어렵다.

선의 격언 가운데 이런 말이 있다. “처음 차를 마실 적엔 차는 차요, 찻잔은 찻잔이다. 차를 배우기 동안에는 더 이상 차는 차가 아니고, 찻잔도 찻잔이 아니다. 끝내, 차를 알게 된 다음엔 차는 다시 차요, 찻잔은 다시 찻잔이다.” 여태 이보다 새 배움의 길을 또렷이 나타내는 글귀를 본 적이 없다. 새로운 것을 배우는 것은 언제나 이러하다. 그리고 이런 배움에는 용기가 아닌, 재미가 필요하다. 재미없는 것은 노력으로 익숙해 질 수는 있을지 언정, 진짜로 배울 수가 없다.

이 책을 옮겨 쓴 사람도 그러했을 것이다. 이런 책을 옮겨 쓰기로 마음 먹는 데는 용기만 가지고는 어렵다. 두 사람 모두 나와 특별한 인연을 맺고 있는 것은 바로 이런 프로그래밍 방식에 대한 즐거움을 같이하는 데서 비롯되었기 때문이다. 내가 몇 해 앞서, “프로그램의 구조와 해석” 이런 책을 옮겨 쓴 데서 용기를 얻어, 이런 일을 벌였다고 말하기는 하였지만, 나는 안다. 이런 일은 용기로 마무리하기에는 너무 번거롭다. 용감한 자도 부지런한 자도 재미있어서 일하는 자를 넘어서지 못하는 법이다.

무슨 말을 더 보태야 할까? 여기까지 내가 쓴 글을 읽었는지 아닌지 알 길은 없지만, 이제 이런 책으로 공부하기를 여전히 며뭇거리다, 갑자기 우리 나라 소프트웨어 기술자의 현실을 생각하느라 쓴웃음을 머금고 책을 덮으려다가, 뭔가 아쉬움이 남아 다시 이 책을 열었다 덮었다 하고 있을 것만 같은 분들에게 솔직한 내 마음을 전한다.

본디 처음 이 글을 풀어 낼 적에는, 현재 산업에서 즐겨 쓰고 있는 기술 속에 벌써부터 이 책에서 집중하는 기술과 경험들이 오래 전부터 녹아 들어가 있었고, 앞으로 프로세서의 병행 / 병렬 아키텍처가 더욱 일반화되고 소프트웨어 품질이 산업의 성패를 가늠할 만치 중요해 지는 마당에서는, 이렇듯 작고 견고한 언어로 튼튼할 뿐더러, 되 쓰임새(재사용성) 뛰어난 소프트웨어 개발 기술이 중요하다는 말을 꺼내며, 그 증거를 요목 조목 들이대려고도 하였다. 한데, 그런 식의 논리는 솔직히 나조차 지겹다. 새 기술이 나올 때마다 안 배우면 뭔가 큰일 나는 듯 호통치며 억지로 산업에 밀려나온 엉성한 기술과 제품들이 어디 한 둘이던가. 이 척박한 소프트웨어 산업의 불모지에서는 더욱 그러하다.

더욱 속내를 털어 놓자면, 앞으로 이 나라 이 땅의 소프트웨어 기술자들이, 스스로 애써 갖춘 실력과 공부에 투자한 만큼 대접을 받고, 오랫동안 기술자로 살아남아서 그 경험의 깊이와 안목 때문에 ‘대목’으로 우러를 받는 세상은 쉽사리 오기 어려울 것 같다. 이게 한참 딴 세상을 기웃거리다 다시 한 해 남짓 산업을 둘러본 뒤 느낀 감정의 요약이다. 따라서, 이 책으로 뭔가를 배우겠다고 마음 먹었다면 그것은 오로지 재미 때문이어야 한다. 그 말고, 감히 다른 어떤 까닭으로 이런 공부를 해야 할 이유를 댈 수 있겠는가? 재미 말고 그 어떤 다른 이유로 젊은 피에게 뜻을 같이 하자 소리칠 수 있겠는가? 다행히, Haskell로 프로그램을 짜면 재미있다. 처음 얼마쯤 낯설어서 헤매는 시간을 넘기고 나면 갈수록 깊은 맛이 우러나오는 재미가 있다. 내가 아직도 틈나는 대로 프로그램을 짜고, 기술을 공부하고, 생각을 멈추지 않으며 소프트웨어 기술을 떠들고 다니는 이유가 그 때문이다. 오로지 그 뿐이다.

서울 살이 시작한지 딱 1년 째 되는 날
한국 마이크로소프트 개발자 플랫폼 총괄 부서 내,
양로원 식구들을 대표하여

김재우

a

a

제 1 장

소개

이 장에서 우리는 앞으로 살펴볼 책의 나머지 부분을 위한 무대를 마련할 것이다. 우선 함수의 개념을 돌아보는 것으로 시작해, 함수형 프로그래밍이라는 개념을 소개하고, 하스켈의 주요한 특징과 역사에 대해 간단히 살펴본 후, 하스켈을 맛볼 수 있는 몇 가지 작은 보기로 이 장을 맺는다.

1.1 함수

하스켈에서 함수란 하나 혹은 그 이상의 인자를 받아 어떤 하나의 결과를 내는 대응을 말하며, 함수 이름과 각각의 인자에 이름을 붙여 그 인자를 가지고 결과를 어떻게 계산할지 낱낱이 적어 놓은 몸체로 이루어진 등식으로 정의한다.

예를 들어, *double* 함수는 인자 x 라는 수를 인자로 받아 그 결과로 $x + x$ 를 내며, 다음 등식으로 정의할 수 있다.

double $x = x + x$

함수를 실제 인자들에 적용하면, 함수 몸체에 나타나는 인자 이름들을 실제 인자로 바꿔 넣은 결과를 얻는다. 이렇게 해서 얻은 결과는 더 이상 줄일 수 없는 숫자 같은 것이 될 수도 있다. 하지만 일반적으로 그 결과가 다른 함수 적용식들을 포함하는 식인 경우가 더 많고, 그 안에 있는 함수 적용식들을 마찬가지 방법으로 계속 바꿔 나감으로써 최종적인 결과를 얻는다.

옮긴이 주: 이 보기는 실제로 컴퓨터로 돌려 볼 수 있는 하스켈 스크립트이며, 2 장에서 컴퓨터로 실습하는 방법을 소개한다.

예컨대, 함수 *double* 을 3에 적용하는 *double 3*이라는 적용식의 결과는 다음과 같은 계산을 통해 얻을 수 있다. 계산의 각 단계마다 중괄호 안에 짧은 설명을 달아 놓았다.

$$\begin{aligned} & \text{double } 3 \\ = & \quad \{ \text{ double 을 적용 } \} \\ = & \quad \begin{aligned} & 3 + 3 \\ = & \quad \{ \text{ + 를 적용 } \} \\ = & \quad \begin{aligned} & 6 \end{aligned} \end{aligned}$$

마찬가지로, *double* 이 두 겹 적용된 식 *double (double 2)*는 다음과 같이 계산할 수 있다.

$$\begin{aligned} & \text{double (double 2)} \\ = & \quad \{ \text{ 안쪽 double 을 적용 } \} \\ = & \quad \begin{aligned} & \text{double (2 + 2)} \\ = & \quad \{ \text{ + 를 적용 } \} \\ = & \quad \begin{aligned} & \text{double 4} \\ = & \quad \{ \text{ double 을 적용 } \} \\ = & \quad \begin{aligned} & 4 + 4 \\ = & \quad \{ \text{ + 를 적용 } \} \\ = & \quad \begin{aligned} & 8 \end{aligned} \end{aligned} \end{aligned}$$

안쪽부터 적용한 위와는 달리 바깥의 *double* 함수부터 적용해도 같은 결과를 얻는다.

$$\begin{aligned}
 & \text{double (double 2)} \\
 = & \quad \{ \text{바깥 double 을 적용 } \} \\
 & \text{double 2 + double 2} \\
 = & \quad \{ \text{첫째 double 을 적용 } \} \\
 & (2 + 2) + \text{double 2} \\
 = & \quad \{ \text{첫째 + 를 적용 } \} \\
 & 4 + \text{double 2} \\
 = & \quad \{ \text{double 을 적용 } \} \\
 & 4 + (2 + 2) \\
 = & \quad \{ \text{둘째 + 를 적용 } \} \\
 & 4 + 4 \\
 = & \quad \{ \text{+ 를 적용 } \} \\
 & 8
 \end{aligned}$$

하지만 이 계산은 원래 계산보다 두 걸음이 더 듈다. 이는 *double 2*라는 식이 첫 걸음에서 두 개로 복제되는 바람에 같은 식을 두 번 줄였기 때문이다. 일반적으로 어느 함수를 먼저 적용하는지가 최종 결과에 영향을 미치지는 않으나, 계산을 하는 데 몇 걸음이나 들지 그 걸음 수에는 영향을 미칠 수 있으며, 계산이 끝날지 끝나지 않을지에도 영향이 있을 수 있다. 이러한 주제를 12장에서 더 자세히 다룰 것이다.

1.2 함수형 프로그래밍 functional programming

함수형 프로그래밍 functional programming 이란 무엇인가? 여러가지 의견이 있어 정확한 정의를 내리기는 어렵지만, 일반적으로 함수형 프로그래밍이라 함은 함수를 그 인자에 적용하는 것을 계산의 기본으로 삼는 방식을 말한다. 함수형 프로그래밍 언어란 함수형으로 프로그래밍하는 것을 지원하고 장려하는 언어를 말한다.

컴퓨터로 1부터 n 까지 정수의 합을 구하는 작업을 보기로 들어 이 개념에 대해 설명해 보겠다. 대부분의 프로그래밍 언어에서는 값을 저장하며 시간이 지남에 따라 그 저장된 값이 바뀔 수 있는 두 개의 변수, 곧 n 까지 헤아려 올라가는 변수와 총합을 구하기 위해 쌓아 나가는 변수를 써서 이 작업을 수행할 것이다.

예컨대, 변수의 값을 바꾸는 덮어쓰기 기호 `:=` 및 어떤 조건이 만족할 때까지 일련의 연산을 되풀이하여 실행하는 `repeat` 와 `until` 키워드를 써서 다음과 같이 그 합을 구할 수 있다.

```
count := 0
total := 0
repeat
    count := count + 1
    total := total + count
until
    count = n
```

위 프로그램이 뜻하는 바는, 먼저 `count` 와 `total` 의 값을 처음에는 0 으로 놓고 시작해서, `count` 를 증가시키며 그 값을 `total` 에 더하는 명령을 `count` 가 n 이 되어 프로그램이 끝날 때까지 되풀이해서 실행하는 것이다.

위 프로그램에서 계산의 기본 방법은 저장된 값을 바꾸는 것이며, 이는 프로그램이 돌면서 덮어쓰기를 여러 번 실행하게 됨을 뜻한다. 예컨대, $n = 5$ 인 경우 다음과 같이 실행되는데, 마지막에 총합 변수 `total` 에 덮어쓴 값이 바로 구하고자 하는 합이다.

```
count := 0
total := 0
count := 1
total := 1
count := 2
total := 3
count := 3
total := 6
count := 4
total := 10
count := 5
total := 15
```

일반적으로 계산의 기본 방법을 저장된 값을 바꿔 나가는 것으로 하는 프로그래밍 언어를 명령형 언어^{imperative language}라 하며, 그러한 언어로 짠 프로그램은 계산이 어떻게 진행되어야 하는지 하나하나 기술하는 명령 연산들로 구성된다.

그렇다면 이제 하스켈로 1에서 n 까지 합을 구하는 방법을 알아보자. 이러한 계산은 보통 두 개의 라이브러리 함수를 써서 할 수 있는데, 1부터 n 까지의 수로 이루어진 리스트를 만드는 [...] 함수와 리스트의 합을 구하는 *sum* 함수를 써서 다음과 같이 할 수 있다.

sum [1 .. n]

이 프로그램에서 계산의 기본이 되는 방법은 함수를 인자에 적용하는 것이며, 함수를 인자에 적용해 나가는 것을 계속하면 프로그램의 결과를 얻을 수 있다. 예컨대, $n = 5$ 인 경우 다음과 같은 순서로 구하고자 하는 합을 최종 결과로 얻는다.

$$\begin{aligned} & \text{sum } [1..5] \\ = & \quad \{ \dots \text{를 적용 } \} \\ & \text{sum } [1, 2, 3, 4, 5] \\ = & \quad \{ \text{ sum 을 적용 } \} \\ & 1 + 2 + 3 + 4 + 5 \\ = & \quad \{ + \text{를 적용 } \} \\ & 15 \end{aligned}$$

대부분의 명령형 언어에서도 함수를 써서 프로그램을 할 수 있도록 지원하므로, *sum* [1.. n] 이라는 하스켈 프로그램을 그러한 언어로 옮겨 적을 수 있다. 하지만 명령형 언어는 대개 함수형으로 프로그래밍하는 것을 장려하지 않는다. 예컨대, 많은 언어에서는 함수를 리스트와 같은 데이터 구조에 저장한다든지, 함수가 함수를 인자로 받거나 함수를 결과로 낸다든지, 혹은 함수 자신을 이용해 함수를 정의한다든지 하는 것이 제한되어 있거나 가능하더라도 삼가도록 권한다. 반면 하스켈에서는 함수를 쓰는 데 있어 그러한 제한이 전혀 없으며, 함수를 써서 간단하면서도 강력한 프로그래밍을 할 수 있도록 여러가지 기능을 지원한다.

1.3 하스켈의 특징

참고로, 하스켈의 주요 특징과 함께 그 특징에 대해 좀더 자세히 내용을 다루고 있는 책의 관련 장은 다음과 같다.

- 간결한 프로그램 (2, 4 장)

함수형으로 프로그램을 짜면 고차원적 high-level 성질을 자연스럽게 다룰 수 있어, 앞절의 보기에서 나타난 바와 같이 하스켈로 쓴 프로그램은 다른 언어로 쓴 프로그램보다 더 간결한 경우가 많다. 또한, 하스켈 문법은 프로그램을 간결하게 짤 수 있도록 하는 것을 염두에 두고 설계했기 때문에 예약어가 적고 들여쓰기로 프로그램 구조를 나타낼 수 있는 것이 특징이다. 객관적으로 비교하기는 어렵겠지만, 하스켈로 프로그램을 짜면 다른 요즘 언어들로 짤 때보다 절반에서 십분의 일 정도로 짧아지곤 한다.

- 강력한 타입 시스템 (3, 10 장)

대부분의 현대적인 프로그래밍 언어는 어떤 모양으로든 타입 시스템을 갖추고 있어 이를테면 수와 글자를 더하려 하는 등의 아귀가 맞지 않는 잘못을 잡아낸다. 하스켈의 타입 시스템은 타입 유추 type inference라는 정교한 방법으로 프로그래머가 제공해야 하는 적은 양의 타입 정보만을 가지고 자동으로 범주의 아귀가 맞지 않는 많은 잘못을 프로그램 실행 전에 잡아낸다. 또한, 하스켈 타입 시스템은 요즘 다른 언어들과 견주어 볼 때 함수의 타입이 “여러모양” polymorphic, 또 “여러의미” overloaded 일 수 있다는 점에서도 요즘 다른 언어들보다 강력하다.

- 리스트 조건제시식 list comprehension (5 장)

컴퓨터로 자료를 다룰 때 매우 자주 쓰이곤 하는 것이 바로 리스트다. 이 때문에 하스켈에서는 리스트가 언어에서 쓰이는 기본 개념이며, 간단하면서도 강력한 조건제시식을 써서 하나 또는 여러 개의 리스트로부터 고르고 걸러 새로운 리스트를 만들어 낼 수 있다. 조건제시식을 쓰면 리스트를 다루는 많은 함수들을 곁으로 드러나 보이는 되돌기 recursion 없이 깔끔하고 간결하게 정의할 수 있다.

- 되도는 함수 recursive function (6 장)

어느 정도 이상 복잡한 프로그램은 대개 되풀이되거나 제자리로 도는 부분이 어떤 식으로든

필요하다. 하스켈에서는 이렇게 돌기 위한 기본적인 방법으로써 자기 자신을 써서 정의된 되도는 함수를 이용한다. 되도는 함수로 많은 계산들을 간단하고 자연스럽게 정의할 수 있으며, 이 때에 “패턴 매칭”pattern matching과 “보초”guards를 쓰면 조건을 여러 개의 등식으로 나누기가 특히 수월하다.

- 함수를 주고받는 함수higher-order function (7 장)

하스켈은 차수 높은higher-order 함수형 언어다. 이는 함수가 자유로이 함수를 인자로 받거나 결과로 낼 수 있음을 뜻한다. 함수를 주고받는 함수higher-order function를 이용해서 두 함수의 합성함수와 같은 보편적인 프로그래밍 양식을 얻어 자체 내에서 함수로 정의할 수 있다. 더 일반적으로는, 함수를 주고받는 함수로 하스켈 안에서 “전문 영역 언어”domain specific language를 정의하여, 리스트 처리, 문법 분석, 대화식 프로그램 작성 등을 하는 데 쓸 수 있다.

- 모나드에 담긴 효과monadic effect (8, 9 장)

하스켈 함수는 입력을 모두 인자로 받고 출력을 모두 결과로 내는 순수 함수pure function이다. 하지만 많은 경우에, 프로그램이 도는 동안 글쇠판 입력을 읽는다든가, 화면에 출력한다던가 하는 순수함과는 거리가 먼 듯한 결따르는 반응side effect이 있어야 하다. 하스켈에서는 함수의 순수함을 잊지 않으면서 모나드monad라는 수학적 기법에 바탕을 둔 일관된 틀에 따라 이러한 반응을 처리한다.

- 느긋한 계산법lazy evaluation (12 장)

하스켈 프로그램은 실제로 결과가 꼭 필요한 때가 이르기까지는 계산을 미룬다는 생각에 바탕을 둔 기술인 느긋한 계산법에 따라 실행된다. 느긋한 계산법은 불필요한 계산을 피함은 물론, 가능한 모든 경우에 프로그램이 끝남을 보장한다. 따라서 느긋한 계산법에서는 중간 전달용 데이터 구조를 이용해 모듈 병식으로 프로그램을 짜도록 장려한다. 심지어는 무한수열과 같이 무한히 많은 원소들로 이루어진 데이터 구조를 써서 프로그램을 짜는 것도 장려한다.

- 프로그램에 대한 논리적 증명 (13 장)

하스켈 프로그램은 순수 함수이므로 간단한 등식 바탕의 논증equational reasoning에 따라 프로그램을 실행하고, 프로그램을 변환하거나, 프로그램의 성질을 증명할 수 있으며, 심지어는

프로그램이 어떻게 동작해야 하는지에 대한 조건 명세로부터 프로그램을 직접 이끌어낼 수도 있다. 등식 바탕의 논증과 함께 “귀납법”^{induction}을 쓰면 자기 자신을 이용해 정의된 되도는 함수 성질에 대한 논리적 증명을 할 때 특히 유용하다.

1.4 역사적 배경

하스켈의 특징들 중에는 새로운 것이 아니라 다른 언어에서 먼저 도입한 것들도 많다. 하스켈이 어떠한 위치에 있는지 이해를 돋기 위해, 언어와 관련된 중요한 역사적 발달 과정을 아래에 요약해 놓았다.

- 1930년대, 알론조 쳐치^{Alonzo Church}, 함수에 대한 간단하면서도 강력한 수학적 이론인 람다 셈법^{lambda calculus}을 개발하다.
- 1950년대, 존 매카시^{John McCarthy}, 일반적으로 최초의 함수형 프로그래밍 언어라 불리는 Lisp (“LISt Processor”)을 개발하다. Lisp은 람다 셈법의 영향을 받았으나, 여전히 변수 덮어쓰기가 언어의 중요한 기능이다.
- 1960년대, 피터 란딘^{Peter Landin}, 람다 셈법에 굳건한 기반을 두고 변수 덮어쓰기가 없는 최초의 순수 함수형 언어 ISWIM (“If you See What I Mean”)을 개발하다.
- 1970년대, 존 백커스^{John Backus}, 함수를 주고받는 함수와 프로그램에 대한 논리적 증명이라는 개념에 특히 중점을 둔 함수형 프로그래밍 언어 FP (“Functional Programming”)을 개발하다.
- 또한 1970년대, 로빈 밀너^{Robin Milner} 외 여러 사람들, 여러모양 타입과 타입 유추 개념을 처음으로 도입한 최초의 현대적인 함수형 언어인 ML (“Meta-Language”)을 개발하다.
- 1970년대와 1980년대, 데이빗 터너^{David Turner}, 여러 종류의 느긋하게 계산하는 함수형 프로그래밍 언어를 개발하고, 그 경험을 상용으로 만들어진 언어 미란다^{Miranda} (훌륭하다, 감탄할 만하다^{admirable} 는 뜻)에 집대성하다.

- 1987년, 국제적인 위원회를 조직하여 표준적인 느긋하게 계산하는 함수형 프로그래밍 언어인 (논리학자 하스켈 Curry의 이름을 딴) 하스켈 Haskell 개발을 시작하다.
- 2003년, 위원회는 하스켈 Haskell의 안정판을 정의하는 오래도록 기다려 온 하스켈 보고서 Haskell Report 를 발표하다. 이는 15년간 하스켈을 설계한 사람들의 노력의 결정체다.

위의 연구자들 중 매카시 McCarthy, 백커스 Backus, 밀너 Milner 이렇게 세 사람이 흔히 컴퓨터 분야의 노벨상으로 일컫는 컴퓨터학회연합 튜링 상(ACM Turing Award) 수상자라는 것은 주목할 만한 사실이다.

1.5 하스켈 맛보기

하스켈로 프로그래밍하는 것이 어떤 것인지 맛볼 수 있는 작은 보기들을 들면서 이 장을 마무리하고자 한다. 일단, 앞서 이 장에서 살펴본 *sum* 함수에 대해 좀더 알아보자. 하스켈에서는, 수로 이루어진 리스트의 합을 구하는 이 함수를 다음 두 개의 등식으로 정의할 수 있다.

$$\begin{aligned} \text{sum} [] &= 0 \\ \text{sum} (x : xs) &= x + \text{sum} xs \end{aligned}$$

첫째 등식은 빈 리스트의 합이 0임을 나타낸다. 둘째 등식은 비어 있지 않은 리스트, 즉 첫 수 x 와 그 나머지 리스트 xs 로 이루어진 리스트의 합은 xs 의 합에 x 를 더한 것과 같음을 나타낸다. 예컨대, 다음과 같은 계산으로 $\text{sum} [1, 2, 3]$ 의 결과를 얻을 수 있다.

$$\begin{aligned} &\text{sum} [1, 2, 3] \\ &= \{ \text{sum} 을 적용 } \\ &= 1 + \text{sum} [2, 3] \\ &= \{ \text{sum} 을 적용 } \\ &= 1 + (2 + \text{sum} [3]) \\ &= \{ \text{sum} 을 적용 } \\ &= 1 + (2 + (3 + \text{sum} [])) \\ &= \{ \text{sum} 을 적용 } \\ &= 1 + (2 + (3 + 0)) \\ &= \{ + 를 적용 } \\ &= 6 \end{aligned}$$

sum 함수를 자기 자신을 써서 정의했기 때문에 되돌고 있지만, 끝없이 돌지는 않는다는 점을 주목하라. 더 자세히는, *sum*을 적용할 때마다 그 인자의 길이가 하나씩 줄어들어 빈 리스트가 되면 되돌기가 끝나게 된다. 0이 덧셈의 항등원이므로 빈 리스트의 합으로 0을 돌려주는 것은 적절하다. 0이 덧셈의 항등원이라는 말은 모든 x 에 대해 $0 + x = x$ 이고 $x + 0 = x$ 라는 뜻이다.

하스켈에서는 모든 함수에 타입이 있다. 함수 정의로부터 자동으로 유추되는 이 타입은 함수의 인자와 결과가 어떠한지 그 특징을 나타낸다. 예컨대, *sum* 함수는 다음과 같은 타입이다.

$$\text{Num } a \Rightarrow [a] \rightarrow a$$

이 타입은 어떤 종류의 수든지, *sum* 함수가 그 수의 리스트를 같은 종류의 어떤 한 수로 대응시킴을 나타낸다. 여기서 말하는 수의 종류에는 123 같은 정수나 3.141592 같은 떠돌이 소수점 수가 들어간다. 예를 들면, *sum*은 앞서 살펴 본 것처럼 정수 리스트에 적용할 수도 있고, 떠돌이 소수점 수로 이루어진 리스트에도 적용할 수 있다.

함수에 어떤 성질이 있는지 알려준다는 점에서도 타입이 유용하지만, 프로그램 자체를 돌려 보기도 전에 많은 잘못을 자동으로 잡아낸다는 점이야말로 타입이 있음으로써 얻는 더 큰 이익이다. 더 구체적으로 말하자면, 프로그램에 나타나는 각각의 함수 적용식마다 실제 인자의 타입과 함수 자체의 타입이 아귀가 맞아떨어지는지를 하스켈 타입 시스템이 검사해 준다. 예컨대, *sum* 함수를 글자 리스트에 적용하려 하면, 글자는 수의 한 종류가 아니기 때문에 잘못되었다고 나온다.

이제 리스트를 다루는 더 흥미로운 함수를 살펴보겠다. 지금 살펴볼 함수를 통해 하스켈의 또 다른 여러 가지 특징을 알 수 있을 것이다. 다음과 같이 두 개의 등식으로 *qsort*라는 함수를 정의하자.

```


$$\begin{aligned} qsort [] &= [] \\ qsort (x : xs) &= qsort smaller ++ [x] ++ qsort larger \\ \text{where} \\ smaller &= [a \mid a \leftarrow xs, a \leq x] \\ larger &= [b \mid b \leftarrow xs, b > x] \end{aligned}$$


```

위 정의에서, ++ 는 두 리스트를 연결하는 연산자다. 예컨대, $[1, 2, 3] \text{++} [4, 5] = [1, 2, 3, 4, 5]$. 뒤이어 나오는 **where** 는 지역 정의 local definition 를 위한 예약어로, 여기서는 xs 라는 리스트 중에서 x 보다 작거나 같은 원소를 모두 모아 만든 $smaller$ 라는 리스트와, xs 라는 리스트 중에서 x 보다 큰 원소를 모두 모아 만든 $larger$ 라는 리스트를 정의하는 데 쓰인다. 예컨대, $x = 3$ 이고 $xs = [5, 1, 4, 2]$ 면, $smaller = [1, 2]$ 고 $larger = [5, 4]$ 다.

옮긴이 주: ‘지역 정의’란 어떤 정의 안에서만 쓰며 바깥에서는 보이지 않는 정의를 말한다. 예컨대, $smaller$ 와 $larger$ 는 $qsort$ 를 정의하는 둘째 등식의 오른편에서만 쓸 수 있다. 지역 정의를 다른 말로는 ‘우물안 정의’라고도 한다.

그렇다면 $qsort$ 는 대체 무엇을 하는 함수인가? 일단, 한원소 리스트에 대해서는 아무런 효과가 없다는 것, 즉 모든 x 에 대해 $qsort [x] = [x]$ 라는 것을 보이자.

```


$$\begin{aligned} qsort [x] &= \{ qsort \text{를 적용 } \} \\ &= qsort [] \text{++} [x] \text{++} qsort [] \\ &= \{ qsort \text{를 적용 } \} \\ &= [] \text{++} [x] \text{++} [] \\ &= \{ \text{++를 적용 } \} \\ &= [x] \end{aligned}$$


```

이제 위에서 보인 성질을 이용하면 $qsort$ 를 다음의 보기와 같이 리스트에 적용할 때의 계산 과정을 좀더 간단히 보일 수 있게 된다.

```


$$\begin{aligned} qsort [3, 5, 1, 4, 2] &= \{ qsort \text{를 적용 } \} \\ &= qsort [1, 2] \text{++} [3] \text{++} qsort [5, 4] \\ &= \{ qsort \text{를 적용 } \} \\ &= (qsort [] \text{++} [1] \text{++} qsort [2]) \text{++} [3] \text{++} (qsort [4] \text{++} [5] \text{++} qsort []) \\ &= \{ qsort \text{와 위의 성질을 적용 } \} \\ &= [1, 2] \text{++} [3] \text{++} [4, 5] \\ &= \{ \text{++를 적용 } \} \\ &= [1, 2, 3, 4, 5] \end{aligned}$$


```

요컨대, *qsort* 보기에 나타난 리스트를 수의 크기 순으로 정렬했다. 더 일반적으로 말해, 이 함수는 수로 이루어진 리스트를 받아 정렬된 리스트를 결과로 낸다. *qsort*의 첫째 등식은 빈 리스트는 이미 정렬되어 있다는 것을 나타내며, 둘째 등식은 비어 있지 않은 리스트는 첫째 원소를 제외한 나머지 원소를 첫째 원소보다 작거나 같은 수들끼리 정렬하고 큰 수들끼리 정렬한 다음 그 사이에 첫째 원소를 넣음으로써 정렬할 수 있다는 것을 나타낸다. 이렇게 정렬하는 방법을 쿠크정렬(quicksort)이라 하며, 정렬로서는 아주 좋은 방법에 든다.

위에서 살펴본 쿠크정렬 구현은 깔끔하고 간결한 하스켈의 표현력을 보여주는 아주 좋은 보기다. 또한, *qsort*는 생각했던 것보다 아마 더 일반적인 함수로, 단지 수에만 적용할 수 있는 것이 아니라 순서를 매길 수 있는 값이라면 어떤 타입에든 적용 가능하다.

$qsort :: Ord a \Rightarrow [a] \rightarrow [a]$

더 정확히 말하자면 *qsort*의 타입은 위와 같으며, 이는 리스트의 원소가 순서를 매길 수 있는 타입이기만 하면 *qsort*는 그러한 원소로 이루어진 리스트끼리 대응시키는 함수임을 말한다. 하스켈에는 수, 'a' 와 같은 글자 하나하나, "abcd" 와 같은 글줄 등 순서있는 타입이 여럿 있다. 즉, *qsort* 함수로 글자^{character} 리스트나 글줄^{string} 리스트도 정렬할 수 있다는 말이다.

1.6 살펴보기

하스켈 보고서 Haskell Report 는 하스켈 홈페이지 (<http://www.haskell.org/>)에서 얻을 수 있으며 책(25)으로도 출판되었다. 함수형 프로그래밍 언어의 발전에 관한 역사에 대해 더 자세히 알고 싶다면 후닥 Hudak 의 모두풀이 논문(11)을 참조하라.

옮긴이 주: 후닥이 공동저자로 참여한 “하스켈의 역사”라는 모두풀이 논문이 이 책과 비슷한 시기에 발표되었는데, 하스켈의 역사에 대해 더 살펴보고자 한다면 추천한다. 원제는 “A History of Haskell: begin lazy with class”. Paul Hudak (Yale University), John Hughes (Chalmers University), Simon Peyton Jones (Microsoft Research), Philip Wadler (Edinburgh University). The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III). San Diego, California. June 9-10, 2007.

1.7 연습문제

1. 책에서 제시한 방법과 다른 방법으로 *double (double 2)*를 계산해 보라. 제대로 계산한다면 당연히 그 계산 결과가 같아야 한다.
2. 모든 x 에 대해 $\text{sum } [x] = x$ 임을 보이라.
3. 수로 이루어진 리스트의 곱을 구하는 함수 *product*를 정의하고, 정의한 함수를 써서 $\text{product } [2, 3, 4] = 24$ 임을 확인해 보라.
4. *qsort* 정의를 어떻게 바꾸면 반대로 정렬된 리스트를 내겠는가?
5. *qsort* 정의에서 \leq 를 $<$ 로 바꾸면 어떻게 되겠는가?

귀띔: *qsort* $[2, 2, 3, 1, 1]$ 의 경우를 생각해 보라

제 2 장

첫걸음 떼기

이 장에서는 하스켈을 실제로 사용하기 위한 첫걸음을 내딛는다. Hugs 시스템과 표준 서막^{standard prelude}에 대한 소개로 시작해, 함수 적용을 어떻게 표시하는지 설명하고, 첫 번째 하스켈 스크립트를 만들어 본 후, 스크립트와 관련한 몇 가지 문법적 관례에 대해 알아보는 것으로 마무리한다.

2.1 Hugs 시스템

이전 장에서와 같이 작은 하스켈 프로그램은 손으로 돌려볼 수 있다. 하지만 현실적으로는 자동으로 프로그램을 돌릴 수 있는 시스템이 필요하다. 이 책에서는 가장 널리 쓰이는 하스켈 구현인 Hugs라는 대화식 시스템을 쓰겠다.

Hugs는 대화식이라 학습과 시제품 제작에 적당하며 대개의 응용프로그램을 돌리는 데 충분한 성능을 보인다. 하지만, 더 나은 성능을 요하거나 자체로 실행되는 프로그램의 판이 필요한 경우에 쓸 수 있는 하스켈 컴파일러가 몇 가지 있는데, 그 중 글래스고우 하스켈 컴파일러 (Glasgow Haskell Compiler, 줄여서 GHC) 가 가장 널리 쓰인다.

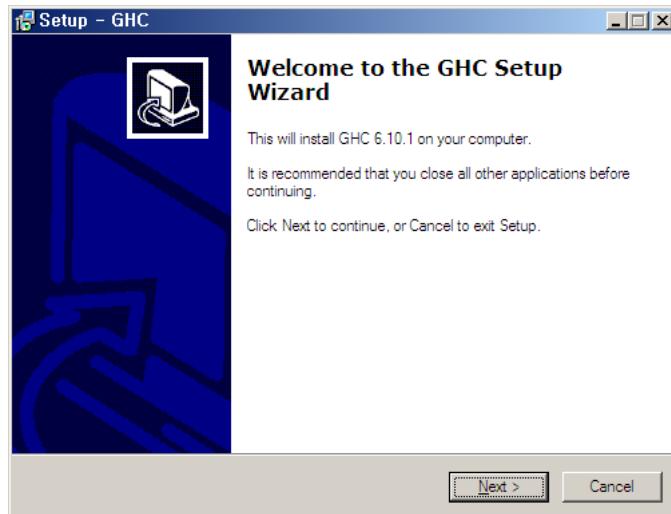
이 컴파일러에는 Hugs와 비슷하게 동작하는 대화식 프로그램이 있어 이 책으로 공부할 때 Hugs 대신 사용할 수도 있다.

옮긴이 주: 2.1.1 절 GHC 설치방법, 2.1.2 절 Hugs 설치방법 및 책 중간중간에 이미지와 함께 나오는 ‘따라하기’는 영문판 원서에는 없으나 대림출판사의 요청으로 옮긴이가 독자의 편의를 위해 추가로 작성한 도움글이다. 이 책에 있는 대부분의 프로그램은 Hugs로 실행할 수 있지만 9장과 11장에 나오는 프로그램은 GHC로만 돌릴 수 있다. 유닉스나 리눅스 사용자들은 배포판 패키지를 이용하면 설치가 간편하므로, 상대적으로 설치가 복잡한 윈도우즈 환경을 기준으로 프로그램 설치를 안내하도록 하겠다. 윈도우즈에서 GHC를 먼저 설치하고 Hugs를 설치해야 Hugs가 하스켈 스크립트 파일인 .hs 및 .lhs 파일에 대한 기본 연결 프로그램이 된다. 따라서 윈도우즈에서는 GHC를 먼저 설치하고 나서 Hugs를 설치할 것을 권한다.

2.1.1 GHC 설치방법

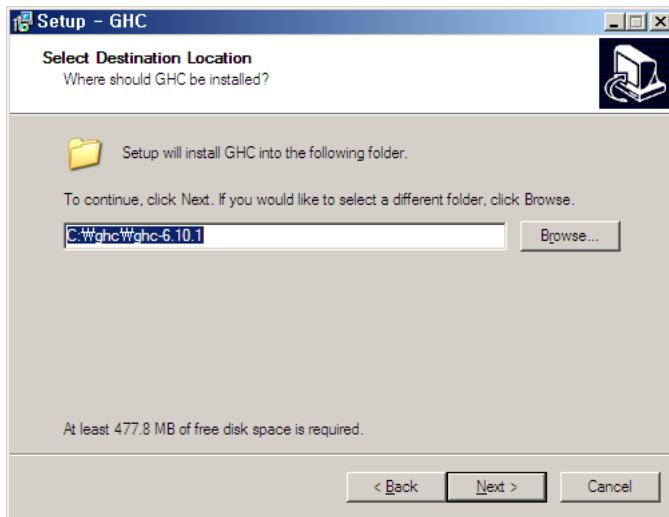
GHC 홈페이지 (<http://haskell.org/ghc/>)의 내려받기 download 페이지에서 윈도우즈 인스톨러 파일 최신판 GHC 파일 (현재는 ghc-6.10.1-i386-windows.exe)을 내려받는다.

- 설치 프로그램 실행하면 다음과 같은 설치 시작 화면이 나온다.



Next 단추를 눌러 다음으로 넘어가라.

2. 다음은 설치 디렉토리 설정 화면이다.



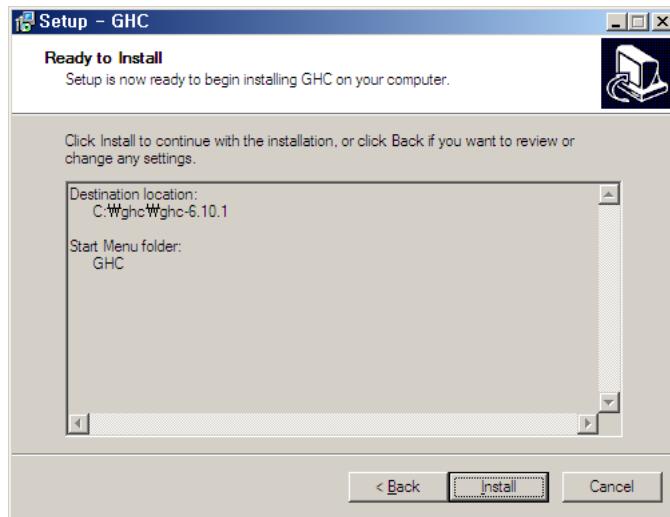
(필요하다면 설치 디렉토리를 바꿀 수 있다.)
Next 단추를 눌러 다음으로 넘어가라.

3. 다음은 시작 메뉴 폴더 이름 설정 화면이다.



(필요하다면 시작 메뉴에 나타날 폴더 이름을 바꿀 수 있다.)
Next 단추를 눌러 다음으로 넘어가라.

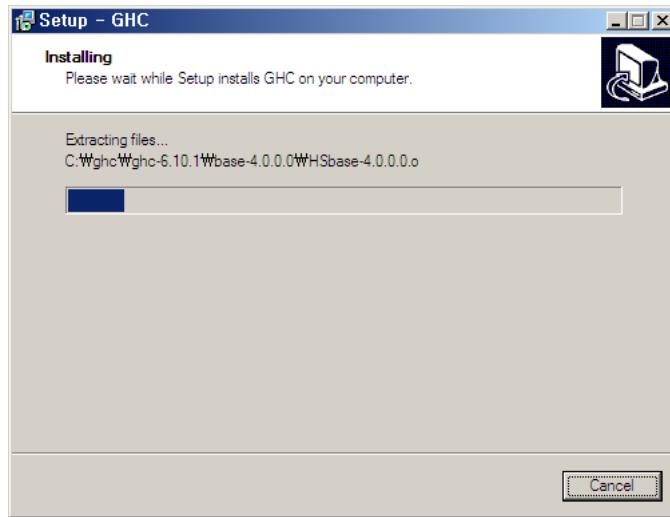
4. 다음은 설정 확인 화면이다.



(설치 디렉토리와 시작 메뉴 폴더 이름 설정을 고치고 싶다면 Back 단추를 눌러 이전 단계로 돌아갈 수 있다.)

Next 단추를 눌러 다음으로 넘어가라.

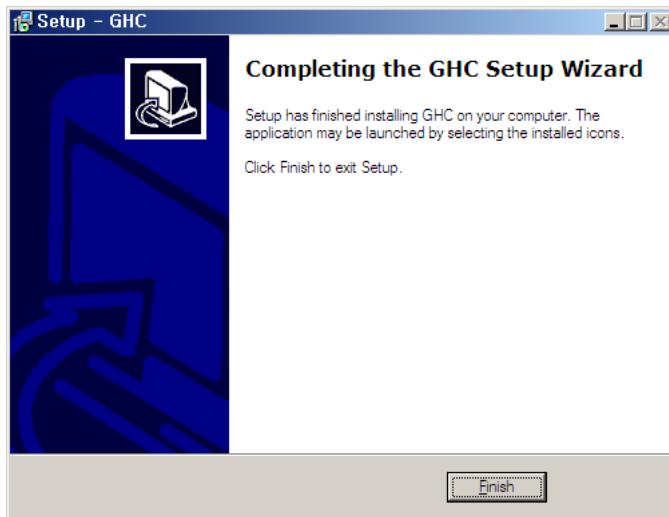
5. 다음은 설치 진행 화면이다.



설치가 끝날 때까지 기다린다.

(설치를 취소하려면 Cancel 단추를 눌러 그만둘 수 있다.)

6. 다음은 설치 종료 화면이다.



Finish 단추를 눌러 설치 프로그램을 종료한다.

7. 다 설치한 다음에는 윈도우즈 명령줄 (cmd.exe)에서 GHC의 대화식 환경인 ghci를 실행할 수 있다.

A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe - ghci". The window shows the following text:

```
C:\> Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\>Documents and Settings\kya>ghci
GHCi, version 6.10.1: http://www.haskell.org/ghc/  ?: for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
Prelude> 1+2+3+4+5+6+7+8+9+10
55
Prelude> sum [1..10]
55
Prelude> -
```

(위 화면은 ghci에서 1부터 10 까지의 합을 두 가지 방법으로 구해 본 것이다.)

더 자세한 GHC 사용방법은 GHC로 실습하는 9장에서 소개한다.

2.1.2 Hugs 설치방법

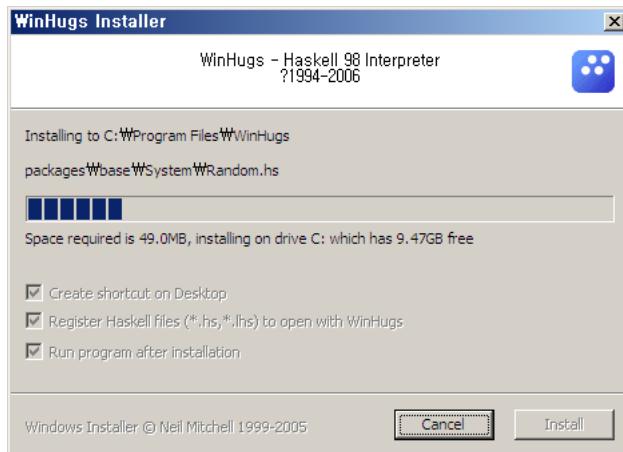
Hugs 홈페이지 (<http://haskell.org/hugs/>)의 내려받기 download 페이지에서 윈도우즈 인스톨러 파일 최신판 파일 (현재는 WinHugs-Sep2006.exe)을 내려받는다.

- 설치 프로그램을 실행하면 다음 설정 화면이 나타난다.



(필요하다면 설치 디렉토리 등의 설정을 바꿀 수 있다.)
Install 단추를 눌러 설치를 시작한다.

- 다음은 설치 진행 화면이다.



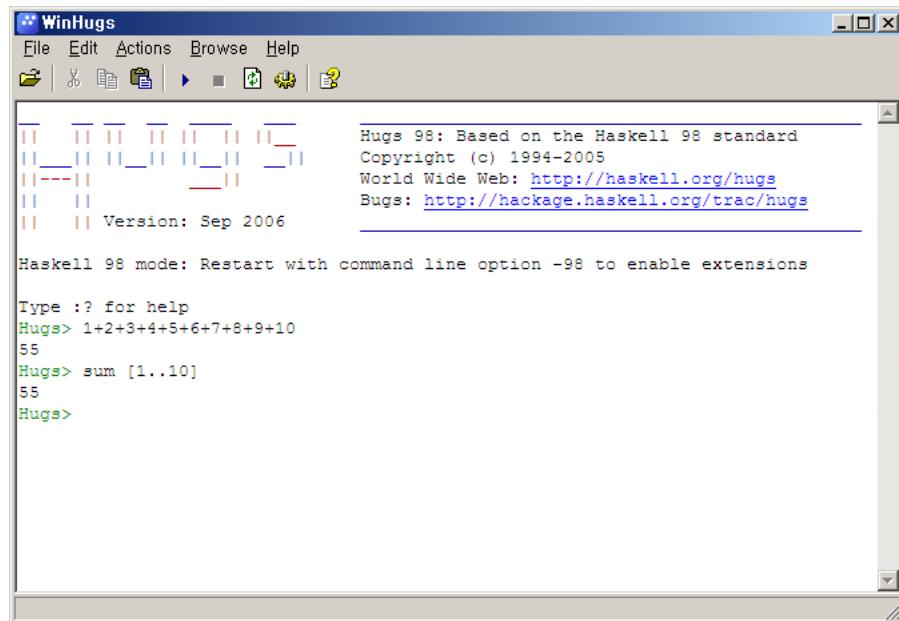
설치가 끝날 때까지 기다린다.
(설치를 취소하려면 Cancel 단추를 눌러 그만둘 수 있다.)

3. 설치를 성공하면 다 설치되었음을 알리는 대화상자가 나타난다.



확인 단추를 눌러 설치를 끝낸다.

4. 설치가 끝나면 WinHugs가 자동으로 실행된다.



(위 화면은 Hugs에서 1부터 10까지의 합을 두 가지 방법으로 구해 본 것이다.)

더 자세한 Hugs 사용방법은 2장에서 소개한다.

2.2 표준 서막standard prelude

Hugs 시스템을 시작하면 *Prelude.hs*라는 라이브러리 파일을 먼저 불러들이고, 길잡이 >를 표시하여 시스템이 사용자가 계산하고자 하는 식을 쳐 넣기를 기다리고 있다는 것을 알린다.

옮긴이 주: Prelude란 원래 연극이나 뮤지컬 교향곡 등의 공연 예술에서 제일 처음 시작할 나오는 도입부나 서곡을 뜻하는 낱말이다. 마치 공연이 본격적으로 시작하기 전에 서막의 도입부를 보여주거나 서곡 연주가 깔리듯이, 표준 서막이란 하스켈로 다른 무엇인가를 하기 전에 먼저 불러들이는 일련의 정의들을 가리키는 용어다. 하스켈 프로그램을 적어 놓은 파일을 일컬을 때 원래 대본을 뜻하는 낱말인 script를 쓰는 것을 생각해 보면 잘 어울리게 붙인 이름이라고 생각한다.

예컨대, 이 라이브러리 파일에는 우리에게 익숙한 덧셈, 뺄셈, 곱셈, 나눗셈, 거듭제곱 등의 정수 연산 함수가 많이 정의되어 있다.

```
Hugs> 2 + 3
5
```

```
Hugs> 2 - 3
-1
```

```
Hugs> 2 * 3
6
```

```
Hugs> 7 `div` 2
3
```

```
Hugs> 2 ↑ 3
8
```

정수의 나눗셈 연산자를 ‘*div*’라고 쓰는 것에 주목하라. 정수 나눗셈 연산이 나누어 떨어지지 않을 때는 소수점 이하를 버린다.

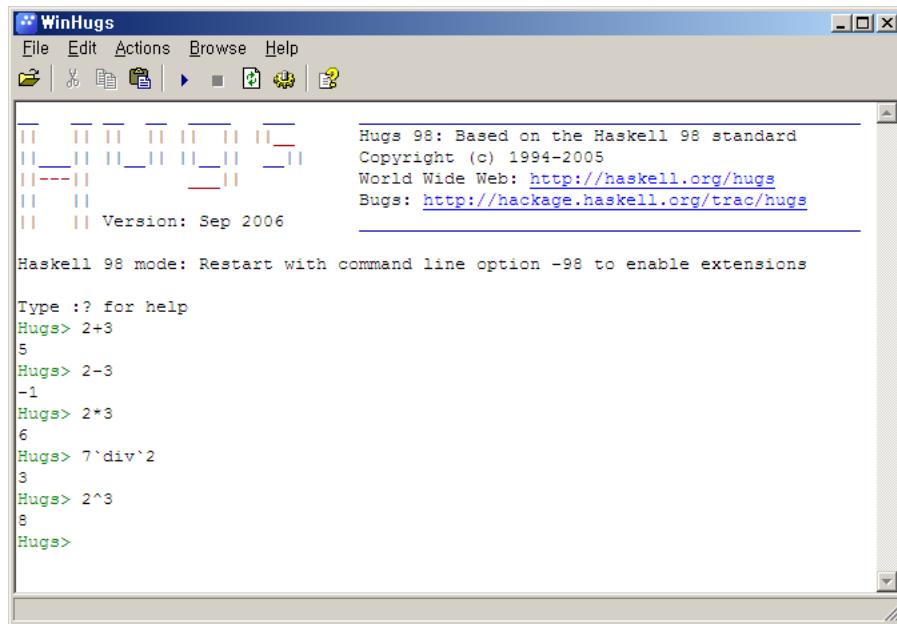
옮긴이 주: ‘div’를 쓸 때는 따옴표(‘)가 아닌 역따옴표(‘backquote’)를 써야 한다. 앞으로 나오겠지만, 따옴표(‘)는 ‘a’와 같은 글자를 나타낼 때나 x' 와 같은 변수 이름에 쓴다. 참고로 ‘div’는 원래 보통 함수 *div*를 연산자로 쓰기 위해 역따옴표로 감싼 것일 뿐이며, 이에 대해서는 3장에서 정수같은 타입을 나타내는 *Num* 클래스를 설명할 때 다시 다룰 것이다. 역따옴표 글쇠를 써 본 적이 없다면 부록 B에서 역따옴표 글쇠가 어떻게 생겼는지 찾아보라.

따라하기

비탕화면의 다음 WinHugs 아이콘을 두번딸깍(double click)하여 WinHugs를 실행할 수 있다.



다음은 WinHugs에서 앞의 사칙연산 보기를 실습한 화면이다.



책에서는 거듭제곱 연산자를 ↑로 조판하고 있지만 일반 텍스트 편집기에서 스크립트를 편집할 때나 WinHugs 실행 화면에서는 ^로 나타난다. 이와 같이 책에 나타난 특수 기호를 보통의 글쇠판으로 입력하는 방법을 부록 B에서 찾아볼 수 있다.

따라하기 끝 일반적인 수학 표기의 관례를 따라, 거듭제곱이 곱셈이나 나눗셈에 우선하며, 곱셈이나 나눗셈이 덧셈이나 뺄셈에 우선한다. 예컨대, $2 * 3 \uparrow 4$ 는 $2 * (3 \uparrow 4)$ 를, $2 + 3 * 4$ 는 $2 + (3 * 4)$ 를 뜻한다. 또한, 왼쪽으로부터 (괄호로) 묶이는 다른 산술 연산과 달리 거듭제곱은 오른쪽으로부터 (괄호로) 묶인다. 예컨대, $2 \uparrow 3 \uparrow 4$ 는 $2 \uparrow (3 \uparrow 4)$ 를, $2 - 3 + 4$ 는 $(2 - 3) + 4$ 를 뜻한다. 하지만, 실제로 스크립트를 쓸 때, 그러한 연산자 우선순위 관례를 이용하기보다는 산술식에 일일이 괄호를 쳐 주는 것이 더 알아보기 분명한 경우가 많다.

옮긴이 주: -1 처럼 음수를 나타내는 연산자와 2 – 3 처럼 뺄셈을 나타내는 두 종류의 (-) 연산자가 있다. 또한 ‘div’ 연산자는 음수를 나타내는 (-) 보다 우선순위가 높다. -3 ‘div’ 2 는 -(3 ‘div’ 2) 와 같은 뜻으로 그 값이 -1 이지만, (-3) ‘div’ 2 의 값은 -2 이다. 저자의 말대로 헛갈리기 쉬운 식에는 괄호를 쳐서 뜻을 분명히 해 주는 것이 좋다.

정수 연산을 하는 함수 외에, 라이브러리 파일에는 리스트 연산을 하는 쓸모있는 함수들이 많이 있다. 하스켈에서 리스트를 나타낼 때는 대괄호 안에 원소를 쓰고 쉼표로 구분한다. 리스트를 다룰 때 가장 자주 쓰이는 라이브러리 함수들을 아래에 간단한 보기와 함께 설명해 놓았다.

- 비어 있지 않은 리스트의 첫 원소를 고른다.

```
Hugs> head [1, 2, 3, 4, 5]
1
```

- 비어 있지 않은 리스트의 첫 원소를 버린다.

```
Hugs> tail [1, 2, 3, 4, 5]
[2, 3, 4, 5]
```

- 리스트의 (0부터 세기 시작해서) n 번째 원소를 고른다.

```
Hugs> [1, 2, 3, 4, 5] !! 2
3
```

- 리스트에서 첫 n 개의 원소를 고른다.

```
Hugs> take 3 [1, 2, 3, 4, 5]
[1, 2, 3]
```

- 리스트에서 첫 n 개의 원소를 버린다.

```
Hugs> drop 3 [1, 2, 3, 4, 5]
[4, 5]
```

- 리스트의 길이를 계산한다.

```
Hugs> length [1,2,3,4,5]
5
```

- 수로 이루어진 리스트의 합을 구한다.

```
Hugs> sum [1,2,3,4,5]
15
```

- 두 리스트를 이어붙인다.

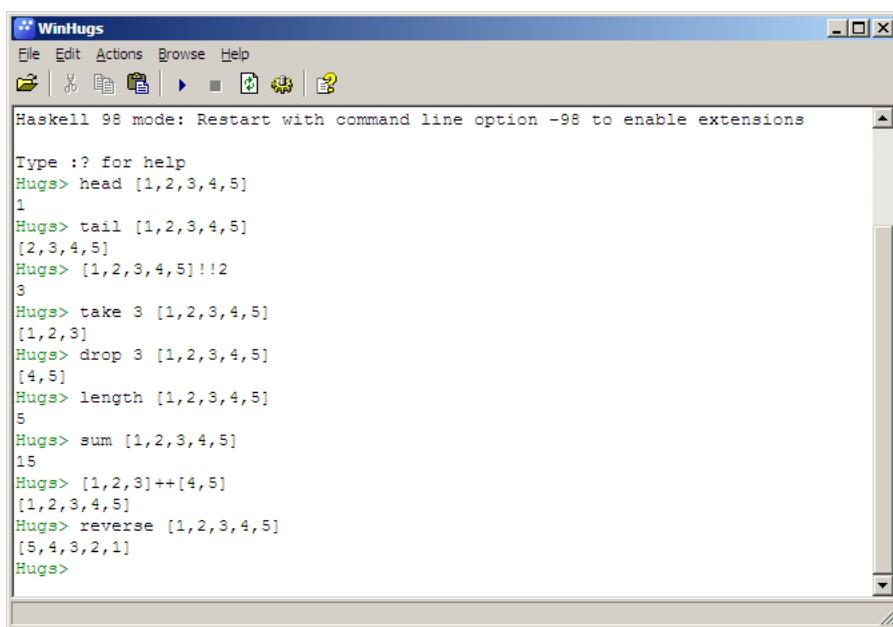
```
Hugs> [1,2,3] ++ [4,5]
[1,2,3,4,5]
```

- 리스트를 뒤집는다.

```
Hugs> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

따라하기

다음은 WinHugs에서 리스트를 다루는 표준 라이브러리 함수들을 실습해 본 화면이다.



```
WinHugs
File Edit Actions Browse Help
Haskell 98 mode: Restart with command line option -98 to enable extensions
Type :? for help
Hugs> head [1,2,3,4,5]
1
Hugs> tail [1,2,3,4,5]
[2,3,4,5]
Hugs> [1,2,3,4,5]!!2
3
Hugs> take 3 [1,2,3,4,5]
[1,2,3]
Hugs> drop 3 [1,2,3,4,5]
[4,5]
Hugs> length [1,2,3,4,5]
5
Hugs> sum [1,2,3,4,5]
15
Hugs> [1,2,3]++[4,5]
[1,2,3,4,5]
Hugs> reverse [1,2,3,4,5]
[5,4,3,2,1]
Hugs>
```

책에서는 리스트를 이어붙이는 연산자를 `++`로 조판하고 있지만 일반 텍스트 편집기에서 스크립트를 편집할 때나 WinHugs 실행 화면에서는 `++`로 나타난다. 이와 같이 책에 나타난 특수 기호를 보통의 글쇠판으로 입력하는 방법을 부록 B에서 찾아볼 수 있다.

따라하기 끝

표준 서막의 일부 함수들은 그 인자의 값에 따라 잘못^{error} 을 내기도 한다. 예컨대, 0으로 나누려 하거나 빈 리스트의 첫째 원소를 얻고자 하면 잘못이 난다.

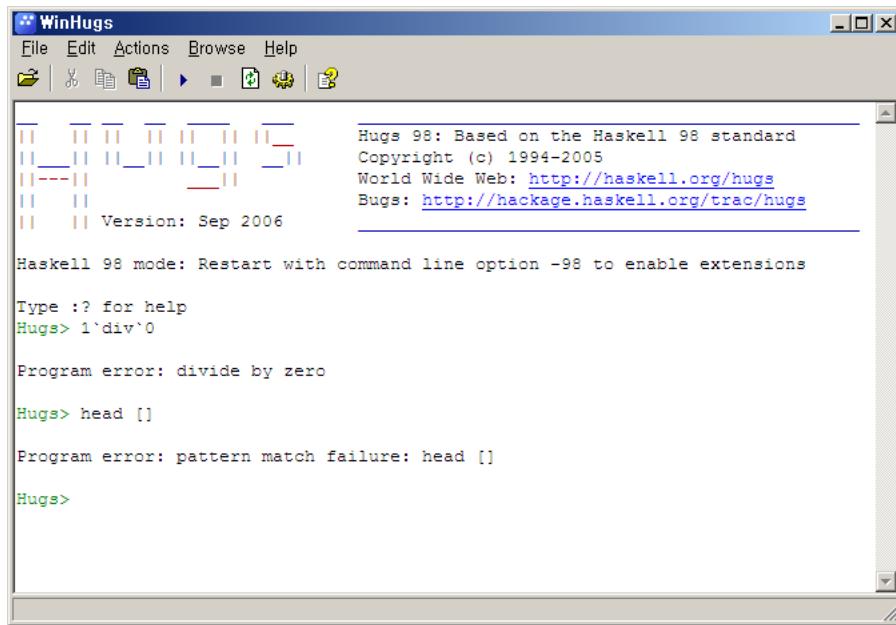
```
Hugs> 1 `div` 0
Error
```

```
Hugs> head []
Error
```

실제로 Hugs 시스템은 잘못^{error} 과 함께 무엇 때문에 잘못된 것 같은지 알려주는 글귀를 같이 낸다.
참고로, 부록 A에서는 표준 서막에서 가장 자주 사용하는 정의들을 소개하고, 부록 B에서는 ↑나 ++ 같은 특별한 하스켈 기호를 보통의 글쇠판으로는 어떻게 쳐야 하는지 알려준다.

따라하기

WinHugs에서 0으로 나누거나 빈 리스트의 첫 원소를 얻으려 했을 때 잘못^{error} 이 나는 화면이다.



따라하기 끝

2.3 함수 적용

수학에서는 함수를 인자에 적용하는 것을 나타낼 때 주로 괄호 안에다 인자를 쓰며, 두 수의 곱셈은 두 수를 바로 이어 쓰는 식으로 있는 듯 없는 듯 나타내곤 한다. 예컨대, 수학에서 다음과 같은 식은

$$f(a, b) + c \cdot d$$

함수 f 를 두 개의 인자 a 와 b 에 적용하고 그 결과를 c 와 d 의 곱과 더하는 것을 뜻한다. 이와 달리 하스켈에서는 함수가 차지하는 중요성을 반영하여 하스켈에서는 함수 적용을 있는 듯 없는 듯 나타내며, 두 수의 곱셈은 * 연산자를 써서 드러나 보이게 나타낸다. 예컨대, 위의 식을 하스켈로는 다음과 같이 쓸 수 있다.

$$f\ a\ b + c * d$$

그리고 함수 적용이 다른 어떤 연산자보다 우선순위가 높다. 예컨대, $f\ a + b$ 는 $(f\ a) + b$ 를 뜻한다. 수학과 하스켈에서 함수 적용을 나타내는 표기법이 어떻게 서로 다른지 보여주는 몇 개의 보기가 더 다음 표에 나타나 있다.

수학	하스켈
$f(x)$	$f\ x$
$f(x, y)$	$f\ x\ y$
$f(g(x))$	$f\ (g\ x)$
$f(x, g(y))$	$f\ x\ (g\ y)$
$f(x)g(x)$	$f\ x * g\ x$

2.4 하스켈 스크립트

표준 서막에 있는 함수 말고도 새로이 함수를 더 정의할 수도 있다. 새로운 함수를 Hugs의 길잡이 > 일에서는 정의할 수 없으며, 반드시 스크립트 안에 정의해야 한다. 스크립트란 일련의 정의를 포함하는 텍스트 파일이다. 관례상 하스켈 스크립트 파일에 대개 .hs 확장자를 붙임으로써 다른 종류의 파일과 구분한다.

옮긴이 주: GHC의 대화식 환경인 ghci에서는 함수를 정의할 수도 있다. 하지만 이렇게 정의하고 나면 어떻게 정의했는지 다시 찾아보기 힘들기 때문에 ghci로 실습할 때도 정의는 스크립트에 하는 것이 좋다.

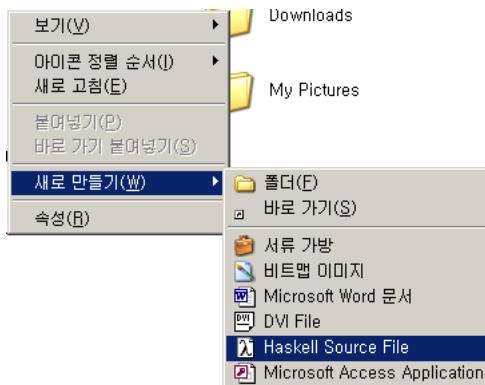
첫 번째 스크립트

하스켈 스크립트를 편집할 때는, 스크립트 편집기 창 하나와 Hugs를 돌리는 창 하나, 이렇게 창을 두 개 열어 놓는 것이 좋다. 예를 들어, 텍스트 편집기를 열어 다음 두 함수 정의를 쳐 넣은 후 스크립트를 *test.hs*라는 이름의 파일로 저장한다:

```
double x      = x + x
quadruple x = double (double x)
```

따라하기

파일 탐색기에서 오른 땜까^{right click}으로 팝업 메뉴를 열어 다음과 같이 하스켈 스크립트를 새로 만든다. Hugs를 설치했으면 팝업 메뉴의 새로 만들기에 ‘Haskell Source File’이라는 항목이 나타날 것이다.



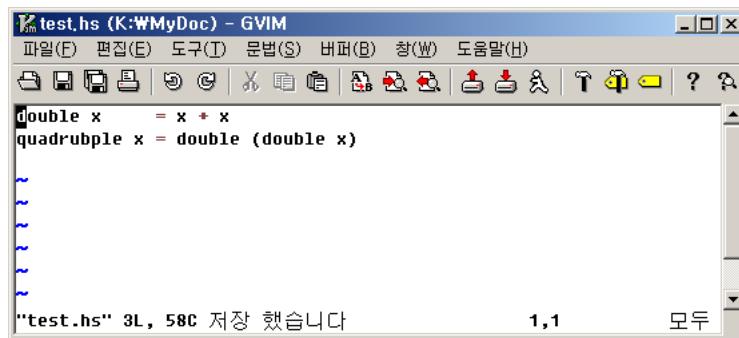
다음은 새로 파일을 하스켈 스크립트의 이름을 수정하는 화면이다.



새로 만든 하스켈 스크립트의 이름을 다음과 같이 *test.hs*로 고친다.



이제 자신이 애용하는 텍스트 편집기로 *test.hs*를 열어 하스켈 함수 *double* 과 *quadruple* 을 정의하고 스크립트 파일을 저장하면 된다.



위 그림은 옮긴이가 애용하는 Vim에서 *double* 과 *quadruple* 을 정의하고 *test.hs*에 저장한 화면이다. 잘 살펴보면 =나 + 같은 기호가 다른 색으로 나타나는 것을 알 수 있다. 이는 Vim 편집기가 하스켈 문법 강조를

지원하기 때문이다. 새로 편집기를 설치하고 싶지 않다면 윈도우즈에서 기본적으로 제공하는 메모장(notepad)를 써도 되지만, 아무래도 하스켈 문법 강조를 기본적으로 지원하는 편집기(Vim, Emacs, Notepad++, Yi 등)를 쓰는 것이 나중에 더 큰 스크립트를 작성할 때 알아보기 편하므로 이런 편집기를 설치할 것을 추천한다. 그 외에도 ConTEXT 편집기와 같이 기본 프로그램과 같이 배포되지는 않지만 하스켈 문법 강조 기능을 추가로 설치할 수 있는 편집기도 많이 있으므로 본인이 가장 애용하는 에디터가 설명 하스켈 문법 강조를 기본적으로 제공하지 않더라도 추가로 설치할 수 있는지 확인해 보라. 또한 널리 쓰이는 UltraEdit 나 Eclipse 에도 하스켈 관련 플러그인이 있다. UltraEdit 나 Eclipse 에서 하스켈 관련 설정 및 사용법은 “함수형 프로그래밍의 모든 것” 사이트의 하스켈 정보 페이지 (<http://functional.or.kr/haskell>)를 참고하라.

따라하기 끝 그 다음에는, 편집기를 그대로 열어 둔 채, 다른 창에서 Hugs와 대화를 시작하고 다음과 같이 새로 만든 스크립트를 불러오라고 명령한다.

```
Hugs> :load test.hs
```

옮긴이 주: 윈도우즈에서 Hugs를 설치하여 .hs 파일의 기본 실행 프로그램으로 등록되어 있다면 스크립트 파일 아이콘을 두번딸까double click 하여 WinHugs를 시작함으로써 스크립트를 불러들일 수도 있으며, 앞으로 살펴볼 따라하기에서와 같이 팝업 메뉴에서 연결 프로그램으로 WinHugs를 선택해 불러들일 수도 있다.

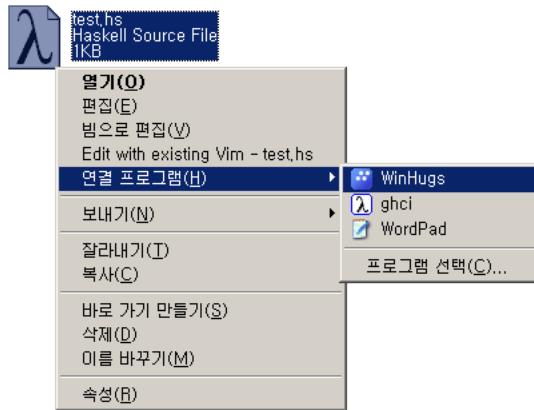
이제 *Prelude.hs*와 *test.hs*를 둘 다 불러들였으므로, 두 스크립트의 함수를 모두 자유로이 쓸 수 있다.
 옮긴이 주: 파일을 제대로 불러들였으면 Hugs> 대신 Main>로 바뀌게 된다. 길잡이 > 오른쪽에는 모듈 이름이 나타나는데, 모듈 이름을 지정하지 않은 파일을 불러들이면 기본적으로 Main 모듈이라고 간주한다. 이 책에서는 모듈에 대해 자세히 다루지는 않으며, 5장, 7장, 8장에서 표준 라이브러리 *Char* 모듈을 불러 쓸 뿐이다. 다음 장부터는 모듈 이름을 쓰지 않고 간단히 > 로만 Hugs 화면을 나타내겠다.

```
Main> quadruple 10  
40
```

```
Main> take (double 2) [1,2,3,4,5,6]  
[1,2,3,4]
```

따라하기

하스켈 스크립트 파일 아이콘을 오른쪽 **right click** 하여 팝업 메뉴에서 연결 프로그램 중 WinHugs를 선택하여 WinHugs를 시작함으로써 *test.hs*를 불러들일 수도 있다. 이 방법은 *ghci*를 기본 연결 프로그램으로 놓고 더 자주 사용하는 독자들이 WinHugs로 실습해 보려고 할 때 유용하다.



다음은 WinHugs에서 *test.hs*를 불러들여 실습한 화면이다.

```

Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-2005
World Wide Web: http://haskell.org/hugs
Bugs: http://hackage.haskell.org/trac/hugs

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Main> :load test.hs
Main> quadruple 10
40
Main> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]
Main> factorial
ERROR - Undefined variable "factorial"
Main> average
ERROR - Undefined variable "average"
Main>

```

사실 WinHugs를 시작하면서 *test.hs*를 이미 불러들였으므로 *:load test.hs* 명령을 굳이 실행하지 않아도 된다. *quadruple*과 *double*를 *test.hs*에 정의하였으며 *take*도 기본적으로 불러들이는 표준 서막 스크립트인 *Prelude.hs*에 정의되어 있으므로 위와 같이 실습하면 원하는 결과를 얻을 수 있다. 하지만 *factorial*과 *average*는 정의되지 않은 함수이므로 그런 변수 이름을 모른다는 잘못 글귀가 나타나는 것을 볼 수 있다.

따라하기 끝

이제 Hugs 를 그대로 열어 놓고 편집기로 돌아가서 다음 두 개의 정의를 더 쳐 넣고서 파일을 다시 저장했다 치자.

```
factorial n = product [1.. n]
average ns = sum ns `div` length ns
```

위에서 $\text{average } ns = \text{div}(\text{sum } ns)(\text{length } ns)$ 라고 정의해도 같은 뜻이고 맞는 정의지만, div 를 두 인자의 사이에 쓰는 것이 더 자연스럽다. 일반적으로, 인자가 두 개인 함수를 역따옴표로 감싸면 두 인자의 사이에 쓸 수 있다.

Hugs는 고친 스크립트를 자동으로 다시 불러들이지 않으므로, 다음과 같이 `:reload` 명령을 써야 한다.

```
Main> :reload
```

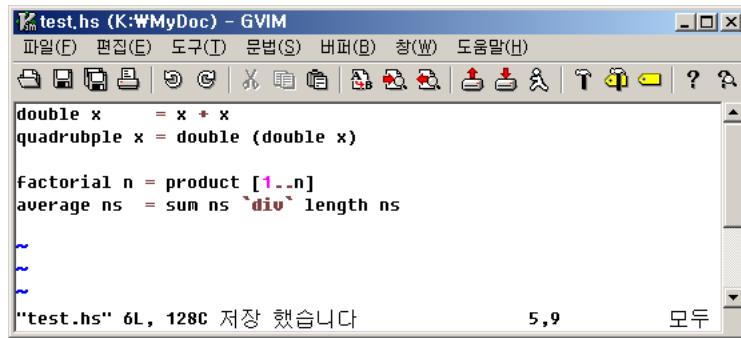
```
Main> factorial 10
362880
```

```
Main> average [1, 2, 3, 4, 5]
3
```

옮긴이 주: WinHugs는 파일 수정을 감지하여 고친 스크립트를 자동으로 다시 불러들이므로 `:relaod` 명령을 일일이 쳐넣지 않아도 된다.

따라하기

다음은 Vim 편집기에서 *factorial*과 *average*를 새로 정의하고 *test.hs*를 다시 저장한 화면이다.



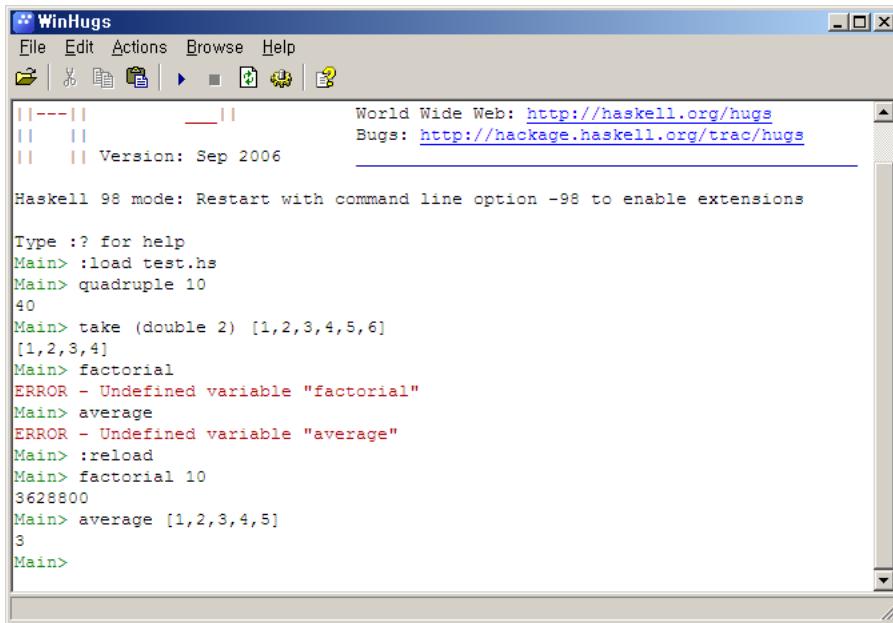
```
test.hs (K:\MyDoc) - GVIM
파일(F) 편집(E) 도구(I) 문법(S) 버퍼(B) 창(W) 도움말(H)
double x = x * x
quadruple x = double (double x)

factorial n = product [1..n]
average ns = sum ns `div` length ns

"
"
"
"test.hs" 6L, 128C 저장 했습니다      5,9      모두
```

1과 같은 숫자가 다른 색으로 문법 강조됨을 알 수 있다.

다음은 WinHugs에서 위와 같이 고친 스크립트를 다시 불러들여 실행해 본 화면이다. 이제 새로이 정의된 *factorial*과 *average* 함수를 불러 쓸 수 있음을 알 수 있다.



```
WinHugs
File Edit Actions Browse Help
World Wide Web: http://haskell.org/hugs
Bugs: http://hackage.haskell.org/trac/hugs
Version: Sep 2006
Haskell 98 mode: Restart with command line option -98 to enable extensions
Type :? for help
Main> :load test.hs
Main> quadruple 10
40
Main> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]
Main> factorial
ERROR - Undefined variable "factorial"
Main> average
ERROR - Undefined variable "average"
Main> :reload
Main> factorial 10
3628800
Main> average [1,2,3,4,5]
3
Main>
```

WinHugs는 불러온 스크립트 파일의 변화를 감지하여 수정된 *test.hs* 내용을 알아서 다시 불러들이므로 *:reload* 명령을 일일이 쳐넣지 않아도 된다. 위 그림에서는 다만 파일을 다시 불러들였다는 것을 드러나 보이게 하기 위한 교육적인 목적으로 일부러 실행해 본 것이다.

따라하기 끝

참고로, 다음 표에 Hugs에서 자주 쓰이는 몇 가지 명령의 쓰임새를 요약해 놓았다. 모든 명령은 그 첫 글자만으로 줄여 써도 된다는 점에 주목하라. 예컨대, `:load` 를 `:l` 로 줄여 쓸 수 있다. 명령어 `:type` 은 다음 장에서 더 자세히 설명한다.

명령	뜻
<code>:load name</code>	<code>name</code> 이라는 스크립트를 불러들인다
<code>:reload</code>	현재 스크립트를 다시 불러들인다
<code>:edit name</code>	<code>name</code> 이라는 스크립트를 편집한다
<code>:edit</code>	현재 스크립트를 편집한다
<code>:type expr</code>	식 <code>expr</code> 의 타입을 보여준다
<code>:?</code>	모든 명령어를 다 보여준다
<code>:quit</code>	Hugs를 끝낸다

더 자세한 Hugs 명령어 목록은 부록 B에 나와 있다.

이름짓기

새로운 함수를 정의할 때 함수와 그 인자의 이름은 모두 소문자로 시작하며, 뒤이어 0개 또는 그 이상의 숫자, 글자 (대문자와 소문자 모두), 밑줄, 작은따옴표가 올 수 있다. 예컨대, 다음과 같은 이름을 쓸 수 있다.

`myFun` `fun1` `arg_2` `x'`

다음은 언어에서 특별한 뜻을 가지는 예약어 keyword 목록으로, 함수나 인자 이름으로 쓸 수 없는 것들이다.

```
case class data default deriving do else
if import in infix infixl infixr instance
let module newtype of then type where
```

관례상, 하스켈에서 리스트를 나타내는 인자 이름은 `s`로 끝나곤 하는데, 이는 리스트 안에 여러 개의 값이 들어있을 수 있다는 것을 나타내기 위함이다. 예컨대, 수로 이루어진 리스트는 `ns`, 아무 종류의 리스트는 `xs`, 글자 리스트의 리스트는 `css`로 이름을 붙일 수 있다.

들여쓰기 규칙^{layout rule}

스크립트에서 각각의 정의는 반드시 정확하게 같은 열에서 시작해야 한다. 들여쓰기 규칙으로 정의들을 그 들여 쓴 정도에 따라 서로 다른 묶음으로 구분할 수 있다. 예컨대, 다음 스크립트에서

```
a = b + c
  where
    b = 1
    c = 2
d = a * 2
```

*b*와 *c*가 *a*의 정의 안에서만 쓰이는 지역 정의라는 것은 들여쓰기를 보면 대번에 알 수 있다. 원한다면, 정의들을 중괄호로 묶고 세미콜론으로 구분하여 그러한 구분을 더 뚜렷이 나타낼 수도 있다. 예컨대, 위 스크립트를 아래와 같이 쓸 수도 있다.

```
a = b + c
  where
    { b = 1;
      c = 2}
d = a * 2
```

주석^{comment}

스크립트에는 새로운 정의 외에도 주석을 달 수 있는데, 이 주석을 Hugs는 무시한다. 하스켈에는 한줄 주석과 포개지는 주석, 이렇게 두 종류의 주석이 있다. 한줄 주석은 다음 보기에서 볼 수 있듯이, -- 기호로 시작해 그 줄의 끝까지 간다.

-- 주어진 양의 정수의 사다리곱을 구한다.

factorial n = product [1..n]

-- 정수 리스트 원소들의 평균값을 구한다.

average ns = sum ns `div` length ns

포개지는 주석은 {- 기호로 시작해서 -} 기호로 끝맺으며 여러 줄에 걸쳐 있을 수 있고, 주석이 다른 주석을 포함하며 포개져 있을 수도 있다. 포개지는 주석은 다음 보기와 같이 스크립트의 정의를 뭉텅이로 얼마 동안 없는 것처럼 막아놓고자 할 때 특히 쓸모가 있다.

```
{-
double x      = x + x
quadruple x = double (double x)
-}
```

따라하기

다음은 Vim 편집기에서 주석을 실습하고 *test.hs*를 다시 저장한 화면이다.

```
{-
double x      = x + x
quadruple x = double (double x)
-}

-- 주어진 양의 정수의 사다리곱을 구한다
factorial n = product [1..n]
-- 정수 리스트 원소들의 평균값을 구한다
average ns = sum ns `div` length ns
"test.hs" 9L, 214C 저작 했습니다      5, 0-1      모두
```

주석이 다른 색으로 문법 강조되고 있음을 알 수 있다.

따라하기 끝

2.5 살펴보기

Hugs 시스템은 하스켈 홈페이지 (<http://www.haskell.org/>)에서 자유롭게 구해 쓸 수 있으며, 홈페이지에는 그 외의 다른 많은 유용한 자료들도 있다.

2.6 연습문제

1. 다음 산술식을 연산 순서에 따라 괄호로 묶으라:

$2 \uparrow 3 * 4$
 $2 * 3 + 4 * 5$
 $2 + 3 * 4 \uparrow 5$

2. 이 장에 있는 보기들을 Hugs로 돌려 가면서 확인해 보라.
3. 다음 스크립트에 문법 실수가 세 군데 있다. 실수를 바로잡고 스크립트가 Hugs에서 잘 돌아가는지 확인해 보라.

```
N = a `div` length xs
where
  a = 10
  xs = [1, 2, 3, 4, 5]
```

4. 비어 있지 않은 리스트에서 마지막 원소를 고르는 라이브러리 함수 *last*를 이 장에서 소개한 다른 라이브러리 함수들로 정의할 수 있음을 보이라. 또, 다른 방법으로 정의할 수 있는지 생각해 보라.
5. 비어 있지 않은 리스트에서 마지막 원소를 버리는 라이브러리 함수 *init*을 위와 마찬가지로 두 가지 서로 다른 방법으로 정의해 보라.

제 3 장

타입과 클래스

이 장에서는 하스켈에서 매우 중요한 두 가지 개념인, 타입과 클래스를 소개한다. 우선 타입이란 무엇이며 하스켈에서 어떻게 쓰이는지 알아보는 것부터 시작해서, 몇 가지 종류의 기본 타입 및 작은 타입을 한데 묶어 더 큰 타입을 만드는 방법에 대해 소개하고, 함수의 타입에 대해 좀더 자세히 알아본 후, 여러모양 타입^{polymorphic type}과 타입 클래스^{type class}에 대한 개념을 설명하는 것을 끝으로 이 장을 맺는다.

3.1 기본 개념

타입은 서로 관계 있는 값들의 모임이다. 예컨대, *Bool* 타입은 두 논리값^{logical value} *False* 와 *True* 를 포함하며, $Bool \rightarrow Bool$ 타입은 논리 부정 함수 \neg 과 같이 *Bool* 타입의 인자를 *Bool* 타입의 결과로 대응시키는 모든 함수를 포함한다. $v :: T$ 라고 쓰면 v 라는 값이 T 라는 타입에 속한다는 뜻이며, “ v 의 타입은 T 이다” 라고 읽는다. 예컨대,

$\text{False} :: \text{Bool}$
 $\text{True} :: \text{Bool}$
 $\neg :: \text{Bool} \rightarrow \text{Bool}$

일반적으로, 아직 계산이 다 끝나지 않은 식과 함께 :: 기호를 쓸 수 있으며, 이 때 $e :: T$ 는 식 e 를 계산하면 T 타입의 값이 나올 것이라는 뜻이다. 예컨대,

$\neg \text{False} :: \text{Bool}$
 $\neg \text{True} :: \text{Bool}$
 $\neg (\neg \text{False}) :: \text{Bool}$

하스켈에서는 모든 식에 반드시 타입을 붙일 수 있어야 하며, 식을 계산하기 전에 타입 유추라는 과정을 통해 그 타입이 무엇인지 계산한다. 이 타입 유추 과정의 핵심은 함수 적용에 관한 다음 규칙으로서, f 가 A 타입의 인자를 B 타입의 결과로 대응시키는 함수이고 e 가 A 타입의 식이면, 적용식 $f e$ 의 타입은 B 임을 말한다.

$$\frac{f :: A \rightarrow B \quad e :: A}{f e :: B}$$

예컨대, $\neg \text{False} :: \text{Bool}$ 라고 타입을 알아내는 것은 위 규칙으로부터 $\neg :: \text{Bool} \rightarrow \text{Bool}$ 이고 $\text{False} :: \text{Bool}$ 라는 사실을 이용하면 가능하다. 한편, $\neg 3$ 이라는 식은 위 함수 적용에 관한 규칙에 의해 타입을 찾을 수 없는데, 이는 위 규칙에 따르면 $3 :: \text{Bool}$ 이어야 하지만 3은 논리값이 아니기 때문이다. $\neg 3$ 과 같이 그 타입을 찾을 수 없는 식을 타입 실수가 있다고 하며, 그러한 식은 그릇된 것으로 간주한다.

타입 유추가 계산^{evaluation} 전에 (프로그램이 돌기 전에) 이루어지므로, 하스켈 프로그램은 타입 안전^{type safe}하다고 말하는데, 이는 타입 실수로 인한 잘못이 프로그램이 도는 동안 절대로 발생하지 않는다는 뜻이다. 실제로, 타입 유추로 아주 많은 종류의 프로그램 실수를 잡아낼 수 있으며, 이는 하스켈의 가장 쓸모 있는 특징 중 하나로 꼽힌다. 하지만, 실행 시간에 발생하는 잘못들 중에 타입 유추로 잡아낼 수 없는 것들도 있다는 점에 유의해야 한다. 예컨대, 1 `div` 0라는 식에 타입 실수는 없지만, 0으로 나누는 것이 정의되지 않으므로 프로그램을 돌렸을 때 잘못이 난다.

타입 안전성의 부정적인 면이 있다고 한다면, 그것은 돌려 보면 잘 돌아갈 식들이 타입을 근거로 받아들여지지 않을 수도 있다는 점이다. 예컨대, 조건식 **if** *True* **then** 1 **else** *False*를 돌리면 1이 나오지만, 타입 실수가 있으므로 그릇된 것으로 간주한다. 더 자세히 설명하자면, 조건식에 대한 타입 규칙에서는 조건식의 가능한 두 가지 결과가 서로 같은 타입이어야 하는데, 첫째인 1은 수이고 둘째인 *False*는 논리값이라는 것이 문제다. 그러나, 프로그래머들은 타입 시스템의 제한 속에서 그러한 문제를 피하는 방법을 쉽사리 터득하므로, 실제로 문제가 되지는 않는다.

Hugs에서 *:type* 명령 뒤에 식을 쓰면 다음과 같이 그 식의 타입을 보여준다.

```
> :type ~  
~ :: Bool → Bool
```

```
> :type ~ False  
~ False :: Bool
```

```
> :type ~ 3  
Error
```

따라하기

다음은 WinHugs에서 `:type` 명령어를 실행해 본 화면이다.

```

*** WinHugs
File Edit Actions Browse Help
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-2005
World Wide Web: http://haskell.org/hugs
Bugs: http://hackage.haskell.org/trac/hugs
Version: Sep 2006

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type ?: for help
Hugs> :type not
not :: Bool -> Bool
Hugs> :type False
False :: Bool
Hugs> :type not False
not False :: Bool
Hugs> :type not 3
ERROR - Cannot infer instance
*** Instance   : Num Bool
*** Expression : not 3

Hugs>

```

위 그림에서 $\neg False$ 는 타입이 `Bool`인 식이지만 $\neg 3$ 는 타입을 찾을 수 없는 잘못된 식이라는 것을 알 수 있다. 특히, $\neg 3$ 의 타입을 알아보려 했을 때 빨강색으로 표시되어 나오는 잘못 글귀의 내용은 이 장의 뒷부분에서 클래스에 대해 살펴보고 나면 더 잘 이해할 수 있을 것이다.

따라하기 끝

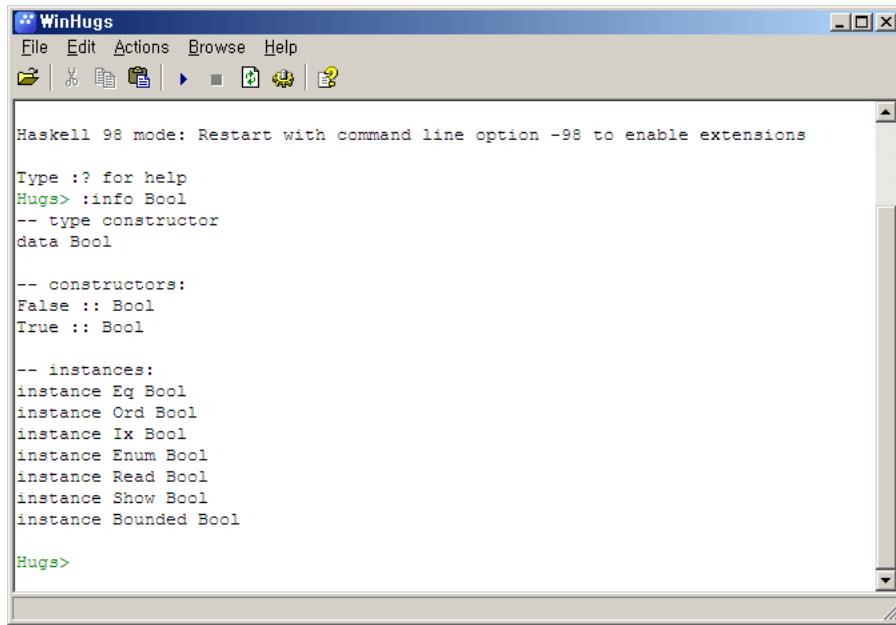
3.2 기본 타입

하스켈 언어에서 제공하는 주어지는 여러 종류의 기본 타입 중 가장 자주 쓰이는 것을 아래에 설명하였다.

Bool (논리값) 이 타입은 두 논리값 `False`와 `True`를 포함한다.

따라하기

다음은 :info 명령어로 Bool 타입의 정보를 알아본 화면이다.



```

** WinHugs
File Edit Actions Browse Help
| | | | | | | | | |
Haskell 98 mode: Restart with command line option -98 to enable extensions
Type :? for help
Hugs> :info Bool
-- type constructor
data Bool

-- constructors:
False :: Bool
True :: Bool

-- instances:
instance Eq Bool
instance Ord Bool
instance Ix Bool
instance Enum Bool
instance Read Bool
instance Show Bool
instance Bounded Bool

Hugs>

```

WinHugs에서 타입에 대한 정보를 :info 명령으로 알아보면 그 타입의 값을 만들어 내는 생성자constructor가 어떤 것들이 있는 알려준다. 또한 그 타입이 어떤 클래스들의 인스턴스인지 알려주는데, 이에 대해서는 이 장의 뒷부분에서 클래스에 대해 살펴보고 나면 더 잘 이해할 수 있을 것이다.

따라하기 끝

Char (글자) 이 타입은 모든 한 개의 글자를 포함한다. 이는 보통의 글쇠판으로 칠 수 있는 'a', 'A', '3', '_' 와 같은 글자는 물론, '\n'(줄바꾸기)이나 '\t'(다음 들여쓰기 위치로 이동)와 같은 특별한 반응을 일으키는 여러 제어 글자control character들을 모두 포함한다. 대부분의 프로그래밍 언어에서와 마찬가지로 한 개의 글자는 (작은) 따옴표 ' ' 안에 써서 나타내야 한다.

String (글줄) 이 타입은 한줄로 된 여러 개의 글자를 모두 포함한다. 이는 "abc", "1+2=3" 와 같은 글줄은 물론 빈 글줄 "" 을 모두 포함한다. 이것 역시 대부분의 프로그래밍 언어와 마찬가지로 글줄은 큰따옴표 " " 로 감싸야 한다.

따라하기

다음은 WinHugs에서 글자와 글줄에 대해 실습한 화면이다.

The screenshot shows the WinHugs Haskell 98 mode interface. The window title is "WinHugs". The menu bar includes File, Edit, Actions, Browse, and Help. The toolbar has icons for file operations like Open, Save, and Print. The main window displays Haskell code. It starts with a message about Haskell 98 mode and command line options. Then it shows several type queries using the ':type' command:

```
Haskell 98 mode: Restart with command line option -98 to enable extensions
Type :? for help
Hugs> ':type 'a'
'a'
Hugs> ':type 'a'
'a' :: Char
Hugs> '\n'
'\n'
Hugs> ':type '\n'
'\n' :: Char
Hugs> "abc"
"abc"
Hugs> ':type "abc"
"abc" :: String
Hugs> ""
""
Hugs> ':type ""
"" :: String
Hugs>
```

따라하기 끝

Int (고정 크기 정수) 이 타입은 $-100, 0, 999$ 등의 정수를 포함하는데, 그 값을 저장하기 위해 고정된 크기의 저장 공간을 사용한다. 예컨대, Hugs 시스템에서 *Int* 타입의 범위는 -2^{31} 부터 $2^{31} - 1$ 까지다. 이 범위를 넘어 가게 되면 그 결과를 예측할 수 없다. 예컨대 Hugs에서 $2 \uparrow 31 :: Int$ 을 계산하면 ($::$ 를 사용해 결과를 다른 수치 타입^{numeric type} 이 아닌 *Int*로 계산한 것을 강제했다) 그 결과는 음수로 잘못된 값이 나온다.

Integer (임의 크기 정수) 이 타입은 모든 정수를 포함한다. 필요한 만큼 저장 공간을 많이 사용하기 때문에 아무리 큰 수나 아무리 작은 수도 표현할 수 있어 그 값의 범위에 제한이 없다. 예컨대, $2 \uparrow 31 :: Integer$ 를 아무 하스켈 시스템에서 계산하더라도 올바른 결과가 나온다.

수의 타입으로 *Int* 와 *Integer* 중 어느 것을 고르냐에 따라 필요한 메모리의 양이나 정확도가 달라진다는 것 외에, 둘 중 어느 한 타입을 선택하는 것은 성능을 좌우하는 요인이 된다. 더 자세히 설명하자면, 대부분의 컴퓨터는 하드웨어로 고정 크기 정수를 처리하는 반면, 임의 크기 정수는 숫자 여러개로 하나의 정수값을 나타내도록 표현하며 이를 소프트웨어로 처리하기 때문에 더 느린다.

Float (단일정밀도 single-precision 떠돌이소수점 floating-point 수) 이 타입은 -12.34 , 1.0 , 3.14159 와 같이 소수점이 있는 수를 포함하며, 그 값을 저장하기 위해 고정된 크기의 저장 공간을 사용한다. 떠돌이소수점 floating-point 이라는 용어는 소수점 다음에 올 수 있는 자릿수가 수의 크기에 따라 달라지는 데서 온 낱말이다. 예컨대, $\text{sqrt } 2 :: \text{Float}$ 를 Hugs로 계산하면 (라이브러리 함수 *sqrt*는 제곱근을 계산한다) 나오는 1.414214 는 소수점 다음에 여섯 자리가 있는 반면, $\text{sqrt } 99999 :: \text{Float}$ 를 계산하면 나오는 316.2262 는 소수점 다음에 네 자리밖에 없다. 떠돌이소수점 수로 프로그래밍에서 마무리 오차 rounding error 를 주의 깊게 처리해야 하는 전문적인 주제이므로, 입문서인 이 책에서는 깊이 다루지 않겠다.

하나의 수가 하나 이상의 수치 타입 numeric type 일 수 있다는 사실을 언급하면서 이 절을 마치고자 한다. 예컨대, $3 :: \text{Int}$, $3 :: \text{Integer}$, $3 :: \text{Float}$ 는 모두 수 3에 대한 올바른 타입 판단이다. 그렇다면 타입 유추를 하는 동안 이런 수들에 어떤 타입이 주어져야 하는지 하는 흥미로운 의문이 들 텐데, 이 장의 뒷부분에서 타입 클래스에 대해 살펴보게 되면 이러한 의문이 해소될 것이다.

따라하기

다음은 WinHugs에서 고정 크기 정수(*Int*), 임의 크기 정수(*Integer*), 단밀도 떠돌이소수점 수(*Float*)에 대해 실습한 화면이다.

```

WinHugs
File Edit Actions Browse Help
Haskell 98 mode: Restart with command line option -98 to enable extensions

Type ?: for help
Hugs> 2^31 :: Int
-2147483648
Hugs> 2^32 :: Int
0
Hugs> 2^31 :: Integer
2147483648
Hugs> 2^32 :: Integer
4294967296
Hugs> sqrt 2 :: Float
1.414214
Hugs> sqrt 99999 :: Float
316.2262
Hugs> 3 :: Int
3
Hugs> 3 :: Integer
3
Hugs> 3 :: Float
3.0
Hugs>

```

따라하기 끝

3.3 리스트 타입

리스트는 서로 같은 타입의 원소를 한줄로 늘어놓은 것이며, 대괄호 안에 그 원소들을 쉼표로 구분하여 나타낸다. 타입이 T 인 원소로 이루어진 리스트의 타입은 $[T]$ 로 표시한다. 예컨대,

```

[False, True, False]      :: [Bool]
['a', 'b', 'c', 'd']     :: [Char]
["One", "Two", "Three"]  :: [String]

```

리스트의 원소 개수를 리스트의 길이라 부른다. 길이 0인 리스트 $[]$ 를 빈 리스트^{empty list}라 부르며, $[False]$ 나 $['a']$ 와 같이 길이 1인 리스트를 한원소 리스트^{singleton list}라 부른다. $[[[]]]$ 와 $[]$ 는 서로 다른 리스트라는 것에 주목하라. 전자는 빈 리스트를 유일한 원소로 갖는 한원소 리스트이고, 후자는 그냥 빈 리스트다.

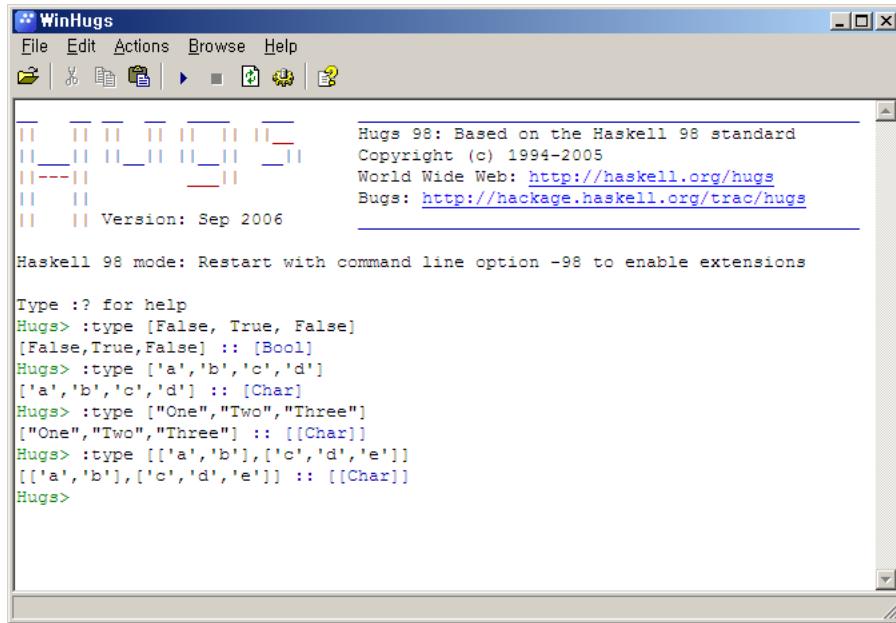
리스트에 대해 더 알아 두어야 할 것이 세 가지 있다. 우선 첫째로, 리스트의 타입에는 리스트의 길이에 대한 정보가 없다. 예컨대, 리스트 $[False, True]$ 와 $[False, True, False]$ 는 그 길이는 서로 달라도 그 타입은 둘 다 $[Bool]$ 로 같다. 둘째로, 리스트의 원소로 가능한 타입에는 아무 제한이 없다. 지금까지 기본 타입이 아닌 타입으로 우리가 다룬 것은 리스트 타입밖에 없기 때문에 보기로 들 수 있는 범위가 현재 한정되어 있기는 하지만, 다음과 같이 리스트의 리스트를 만드는 것도 얼마든지 가능하다.

```
[['a', 'b'], ['c', 'd', 'e']] :: [[Char]]
```

마지막으로, 리스트가 유한한 길이를 가져야 한다는 제한도 없다. 더 구체적으로 설명하자면, 하스켈은 느긋한 계산법을 취하기 때문에 무한한 길이의 리스트를 다루는 것이 자연스럽고 실제로 쓸모가 있다는 것을 12장에서 보게 될 것이다.

따라하기

다음은 WinHugs에서 리스트에 대해 실습한 화면이다.



특히 세 번째 리스트 $["One", "Two", "Three"]$ 의 타입이 책에서처럼 $[String]$ 으로 나타나는 것이 아니라 $[[Char]]$ 로 나타나는 것을 눈여겨 보라. $String$ 이 곧 $[Char]$ 이므로 $[String]$ 은 $[[Char]]$ 과 같다.

따라하기 끝

3.4 순서쌍 타입

순서쌍(혹은 순서있는 n 짹)^{tuple} 이란 유한한 개수의 서로 타입이 달라도 되는 성분^{component} 을 한줄로 나열한 것으로서, 각각의 성분을 쉼표로 구분하여 괄호 안에 써서 나타낸다. 순서쌍 타입은 (T_1, T_2, \dots, T_n) 와 같이 쓰며 이는 1부터 n 까지의 정수 i 에 대해 i 번째 성분의 타입이 T_i 인 모든 가능한 순서쌍을 포함한다. 예컨대,

```
(False, True)      :: (Bool, Bool)
(False, 'a', True) :: (Bool, Char, Bool)
("Yes", True, 'a') :: (String, Bool, Char)
```

순서있는 n 짹에서 성분의 개수 n 을 항수^{arity} 라 부르며, 항수가 0인 순서있는 0 짹 () 를 빈 순서쌍이라 부르고, 항수가 2인 순서있는 2 짹을 순서쌍이라 부르며, 항수가 3인 것은 순서있는 3 짹^{triple} 등으로 부른다. (*False*) 와 같이 항수가 1인 순서있는 1 짹은 허용되지 않는데, 이는 $(1+2)*3$ 와 같이 계산 순서를 분명히 하기 위해 치는 괄호와 구분이 되지 않기 때문이다.

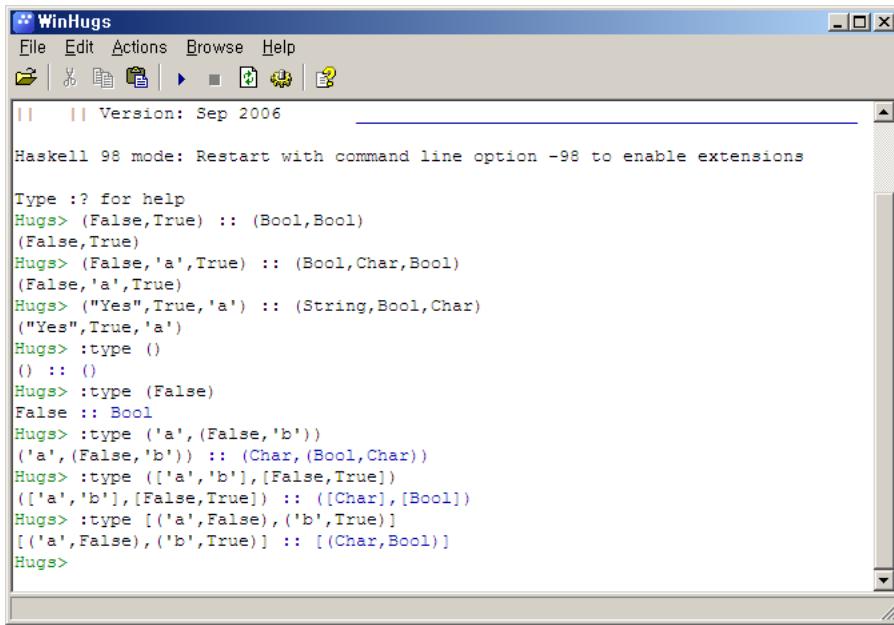
리스트 타입과 마찬가지로, 순서쌍 타입에 대해 더 알아야 할 것이 세 가지 있다. 우선 첫째로, 순서쌍 타입을 보고 그 항수^{arity} 를 알 수 있다. 예컨대, (*Bool*, *Char*) 는 첫째 성분이 *Bool* 타입이고 둘째 성분이 *Char* 타입인 모든 순서쌍을 포함한다. 둘째로, 각 성분이 가질 수 있는 타입에는 아무 제한이 없다. 예컨대, 다음과 같이 순서쌍을 성분으로 하는 순서쌍이나 리스트를 성분으로 하는 순서쌍도 만들 수 있다.

```
('a', (False, 'b'))      :: (Char, (Bool, Char))
(['a', 'b'], [False, True]) :: ([Char], [Bool])
[('a', False), ('b', True)] :: [(Char, Bool)]
```

마지막으로, 순서쌍 타입의 항수^{arity} 가 유한해야만 한다. 그렇지 않다면 프로그램을 돌리기 전에 어떤 순서쌍 타입인지 계산할 수 없기 때문이다.

따라하기

다음은 WinHugs에서 순서쌍 타입에 대해 실습해 본 화면이다.



```
|| Version: Sep 2006
Haskell 98 mode: Restart with command line option -98 to enable extensions
Type ?: for help
Hugs> (False,True) :: (Bool,Bool)
(False,True)
Hugs> (False,'a',True) :: (Bool,Char,Bool)
(False,'a',True)
Hugs> ("Yes",True,'a') :: (String,Bool,Char)
("Yes",True,'a')
Hugs> :type ()
() :: ()
Hugs> :type (False)
False :: Bool
Hugs> :type ('a',(False,'b'))
('a',(False,'b')) :: (Char,(Bool,Char))
Hugs> :type ('a','b',[False,True])
(['a','b'],[False,True]) :: ([Char],[Bool])
Hugs> :type [('a',False),('b',True)]
[('a',False),('b',True)] :: [(Char,Bool)]
Hugs>
```

따라하기 끝

3.5 함수 타입

함수는 어떤 타입의 인자를 또 다른 타입의 결과로 대응시킨다. T_1 타입의 인자를 T_2 타입의 결과로 대응시키는 모든 함수는 $T_1 \rightarrow T_2$ 타입이다. 예컨대,

:: $Bool \rightarrow Bool$
 $isDigit :: Char \rightarrow Bool$

(라이브러리 함수 $isDigit$ 는 주어진 글자가 숫자인지 검사하며 표준라이브러리 $Char$ 모듈에 정의되어 있다.) 함수의 인자나 결과로 가능한 타입에는 어떠한 제약도 없기 때문에, 하나의 인자를 받아 하나의 결과를 계산하는 것을 나타내는 이 간단한 함수 개념만 있으면 여러 개의 인자를 받거나 여러 개의 결과를 돌려주는 함수를 표현하기에 이미 충분하다. 순서쌍 타입이나 리스트 타입을 쓰면 여러 개의 값을 하나의 묶음으로 처리할 수 있기 때문이다. 예컨대, 한 쌍의 정수의 합을 계산하는 add 함수와 0부터 주어진 수까지의 모든 정수를 리스트로 돌려주는 $zeroto$ 함수를 다음과 같이 정의할 수 있다.

```

add      :: (Int, Int) → Int
add (x, y) = x + y
zeroTo  :: Int → [Int]
zeroTo n = [0 .. n]

```

위 보기에서는 함수 정의 앞에 함수의 타입을 적는 하스켈의 관례를 따랐다. 사용자가 타입을 직접 적어 넣으면 스크립트 내용을 이해하는데 도움을 줄 뿐 아니라 타입 유추를 통해 자동으로 계산된 타입과 견주어 모순이 없는지 검사할 수 있다.

하스켈 함수가 항상 인자 타입의 모든 값에 대해 그 결과값을 정의하는 ‘모두 정의된 함수’total function 일 필요는 없으며, 일부 인자값에 대해서는 그 결과값을 정의하지 않는 ‘일부만 정의된 함수’partial function 인 경우도 있음을 기억하라. 예를 들면, 리스트의 첫 원소를 고르는 라이브러리 함수 *head*는 빈 리스트를 넘겼을 때 결과값을 정의하지 않는 일부만 정의된 함수다.

따라하기

다음은 *add* 와 *zeroTo* 함수를 *function.hs*에 정의한 화면이다.



The screenshot shows a GVIM window titled "function.hs (K:WMyDoc) - GVIM". The code editor contains the following Haskell code:

```

add      :: (Int, Int) → Int
add (x, y) = x + y

zeroTo  :: Int → [Int]
zeroTo n = [0 .. n]

```

The status bar at the bottom right indicates "function.hs" 6L, 99C 저장 했습니다" and "3, 0-1".

다음은 *function.hs*를 불러와 실행한 화면이다.

```

WinHugs
File Edit Actions Browse Help
|| Version: Sep 2006
Haskell 98 mode: Restart with command line option -98 to enable extensions
Type :? for help
Main> :type add
add :: (Int,Int) -> Int
Main> :type zeroto
zeroto :: Int -> [Int]
Main> add (2,3)
5
Main> zeroto 5
[0,1,2,3,4,5]
Main> zeroto (add (2,3))
[0,1,2,3,4,5]
Main> head [0,1]
0
Main> head []
Program error: pattern match failure: head []
Main>

```

함수 타입을 나타내는 기호 \rightarrow 가 편집기나 WinHugs에서는 $->$ 로 나타난다. 이와 같이 책에 나타난 특수 기호를 보통의 글쇠판으로 입력하는 방법을 부록 B에서 찾아볼 수 있다.

따라하기 끝

3.6 커리된 curried 함수

인자가 여럿인 함수를 다루는 또 다른 방법으로, 함수를 함수의 결과로 얼마든지 돌려줄 수 있음을 이용할 수 있다. 이 방법을 처음 보면 좀 의아할 수도 있으니, 보기로 통해 살펴보기로 하자. 예컨대, 다음 정의를 살펴보자.

$$\begin{aligned} add' &:: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \\ add' x y &= x + y \end{aligned}$$

타입을 살펴보면, add' 는 타입이 Int 인 인자를 받아 타입이 $\text{Int} \rightarrow \text{Int}$ 인 함수를 돌려주는 함수임을 알 수 있다. 함수 add' 의 정의 자체는 정수 x 와 y 를 받아 $x + y$ 를 돌려주게 되어 있다. 더 정확히 말해서, add' 은 정수 x 를 받아 함수를 돌려주는데, 이 함수가 정수 y 를 받으면 $x + y$ 를 돌려준다.

add' 함수의 결과는 바로 앞 절 add 함수와 같지만, add 는 두 개의 인자를 순서쌍으로 묶어 한꺼번에 받는 반면 add' 는 두 개의 인자를 한 번에 하나씩 받는다는 것에 주목하라. 이러한 차이는 다음과 같이 두 함수의 서로 다른 타입에도 나타난다.

$$\begin{aligned} add &:: (Int, Int) \rightarrow Int \\ add' &:: Int \rightarrow (Int \rightarrow Int) \end{aligned}$$

더 많은 인자를 받는 함수도 마찬가지 방법으로, 함수를 돌려주는 함수를 계속해서 돌려줌으로써 다룰 수 있다. 예컨대, 세 개의 정수를 한 번에 하나씩 인자로 받아 결과로 그 곱을 돌려주는 함수를 다음과 같이 정의할 수 있다.

$$\begin{aligned} mult &:: Int \rightarrow (Int \rightarrow (Int \rightarrow Int)) \\ mult\ x\ y\ z &= x * y * z \end{aligned}$$

위 정의는 $mult$ 라는 함수가 정수 x 를 인자로 받아 함수를 돌려주는데, 그 함수는 이어서 정수 y 를 받아 또 다른 함수를 돌려주며, 이 함수는 마지막으로 정수 z 를 받아 그 결과로 $x * y * z$ 를 돌려준다.

add' 이나 $mult$ 와 같이 인자를 한 번에 하나씩 받는 함수를 커리되었다curried고 한다. 커리된 함수는 그 자체로 흥미롭기도 하지만 순서쌍을 받는 함수보다 더 유연하다는 장점이 있는데, 이는 커리된 함수에 인자를 끝까지 다 적용하지 않고 일부만 적용해서 쓸모 있는 함수를 만들어 낼 수 있는 경우가 많다는 점에서 그러하다. 예컨대, 정수를 하나만큼 증가시키는 함수는 커리된 함수인 add' 에 인자 하나만 적용한 부분 적용식 $add'\ 1 :: Int \rightarrow Int$ 와 같이 만들 수 있다.

커리된 함수를 쓸 때 너무 괄호를 많이 치게 되는 것을 피하고자 두 가지 규칙을 도입한다. 첫째, 함수 타입에 쓰는 화살표 \rightarrow 는 오른쪽부터 묶이는 것으로 한다. 예컨대,

$$Int \rightarrow Int \rightarrow Int \rightarrow Int$$

위와 같이 쓴 것은 아래와 같은 뜻이다.

$$Int \rightarrow (Int \rightarrow (Int \rightarrow Int))$$

둘째, 묵묵히 빈칸으로 나타내는 함수 적용은 왼쪽부터 묶이는 것으로 한다. 예컨대,

mult x y z

위와 같이 쓴 것은 아래와 같은 뜻이다.

$((\text{mult } x) y) z$

꼭 순서쌍으로 묶어야 할 필요가 없다면, 하스켈에서 인자가 여럿인 함수는 일반적으로 커리된 함수로 정의하며, 위의 두 규칙에 따라 쳐야 하는 괄호 개수를 줄일 수 있다.

따라하기

다음은 *add'* 와 *mult* 를 추가로 *function.hs* 에 정의한 화면이다.



K function.hs (K:WMyDoc) – GVIM

파일(D) 편집(E) 도구(I) 문법(S) 버퍼(B) 창(W) 도움말(H)

상단 메뉴와 도구바가 보입니다.

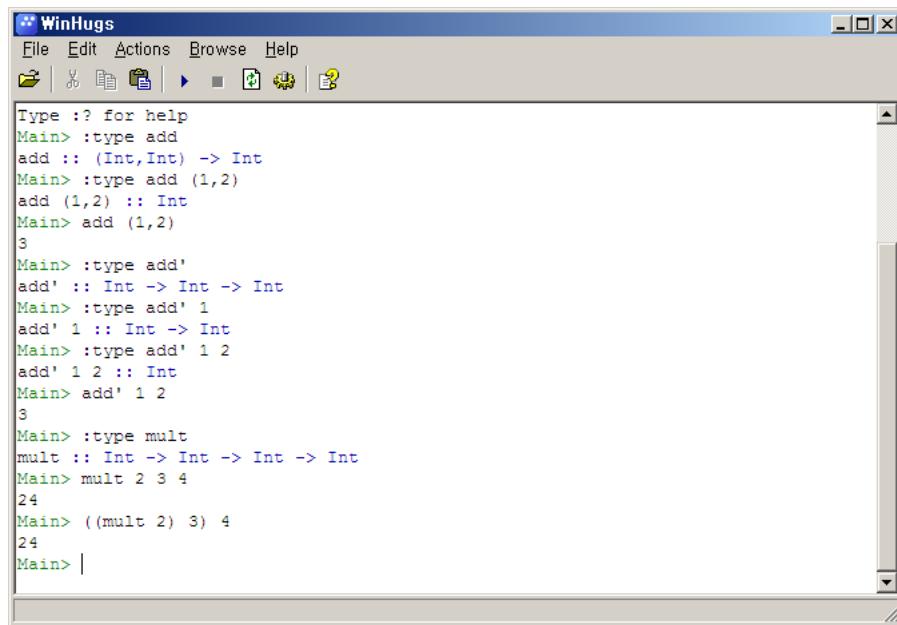
```
add      :: (Int,Int) -> Int
add (x,y) = x * y

zeroTo  :: Int -> [Int]
zeroTo n = [0..n]

add'     :: Int -> (Int -> Int)
add' x y = x + y

mult    :: Int -> (Int -> (Int -> Int))
mult x y z = x * y * z
~
"function.hs" 11L, 221C 저장 했습니다 6,0-1 모두
```

다음은 *add'* 와 *mult* 를 추가로 정의한 *function.hs* 를 불러와 실행한 화면이다.



The screenshot shows the WinHugs Haskell interpreter window. The menu bar includes File, Edit, Actions, Browse, and Help. The toolbar has icons for opening files, saving, and running. The main window displays the following session:

```
Type :? for help
Main> :type add
add :: (Int,Int) -> Int
Main> :type add (1,2)
add (1,2) :: Int
Main> add (1,2)
3
Main> :type add'
add' :: Int -> Int -> Int
Main> :type add' 1
add' 1 :: Int -> Int
Main> :type add' 1 2
add' 1 2 :: Int
Main> add' 1 2
3
Main> :type mult
mult :: Int -> Int -> Int -> Int
Main> mult 2 3 4
24
Main> ((mult 2) 3) 4
24
Main> |
```

add 와 *add'* 의 차이를 눈여겨 보라.

따라하기 끝

3.7 여러모양의 polymorphic 타입

라이브러리 함수 *length* 는 리스트 원소의 타입에 관계없이 아무 리스트의 길이라도 계산한다. 예컨대, 이 함수는 정수 리스트나 글줄 리스트의 길이는 물론 함수 리스트의 길이까지도 계산할 수 있다.

```
> length [1,3,5,7]
4

> length ["Yes", "No"]
2

> :load Char
> length [isDigit, isLower, isUpper]
3
```

(*isDigit*, *isLower*, *isUpper*는 표준라이브러리 *Char* 모듈에 정의된 함수다.) *length*를 아무 타입의 리스트에나 적용할 수 있다는 개념을 타입 변수 type variable를 포함하는 타입으로 딱 떨어지게 나타낼 수 있다. 타입 변수는 소문자로 시작하며, 대개 *a*, *b*, *c* 등으로 이름을 붙인다. 예컨대, *length*의 타입은 다음과 같다.

length :: [*a*] → Int

즉, *a*는 아무 타입이어도 되며, *length* 함수의 타입은 [*a*] → Int이다. 하나 또는 그 이상의 타입 변수가 있는 타입이나 그러한 타입의 식을 여러모양 polymorphic라고 말한다. 따라서, [*a*] → Int는 여러모양 타입이며 *length*는 여러모양 함수다. 일반적으로, 표준 서막에 있는 많은 함수들은 여러모양 함수다. 예컨대,

```
fst    :: (a, b) → a
head   :: [ a ] → a
take   :: Int → [ a ] → [ a ]
zip    :: [ a ] → [ b ] → [ (a, b) ]
id     :: a → a
```

따라하기

다음은 여러모양 함수에 대해 실습한 화면이다.

```
WinHugs
File Edit Actions Browse Help
Type :? for help
Hugs> length [1,3,5,7]
4
Hugs> length ["Yes","No"]
2
Hugs> :load Char
Char> length [isDigit, isLower, isUpper]
3
Char> :type length
length :: [a] -> Int
Char> :type fst
fst :: (a,b) -> a
Char> :type head
head :: [a] -> a
Char> :type take
take :: Int -> [a] -> [a]
Char> :type zip
zip :: [a] -> [b] -> [(a,b)]
Char> :type id
id :: a -> a
Char>
```

isDigit, *isLower*, *isUpper* 함수를 쓰기 위해 *Char* 모듈을 :load 명령으로 불러오는 것에 유의하라.

따라하기 끝

3.8 여러의미 overloaded 타입

산술 연산자 $+$ 는 같은 수치 타입^{numeric type}의 두 수의 합을 계산한다. 예컨대, 두 정수의 합이나 두 떠돌이소수점 수의 합을 다음과 같이 구할 수 있다.

$$\begin{aligned} &> 1 + 2 \\ &\quad 3 \end{aligned}$$

$$\begin{aligned} &> 1.1 + 2.2 \\ &\quad 3.3 \end{aligned}$$

$+$ 를 아무 수치 타입에 적용할 수 있다는 개념을 클래스 제약^{class constraint}이 붙은 타입으로써 딱 떨어지게 나타낼 수 있다. 클래스 제약은 $C\ a$ 와 같은 모양인데, 이 때 C 가 클래스 이름이고 a 는 타입 변수다. 예컨대, $+$ 의 타입은 다음과 같다.

$$(+) :: Num\ a \Rightarrow a \rightarrow a \rightarrow a$$

즉, a 라는 타입이 수치 타입을 나타내는 Num 클래스의 인스턴스이기만 하다면, 함수 $(+)$ 의 타입을 $a \rightarrow a \rightarrow a$ 라 할 수 있다. (연산자를 괄호로 감싸면 커리된 함수로 바꿀 수 있으며, 이에 대해서는 다음 장에서 더 자세히 설명하겠다.) 하나 또는 그 이상의 클래스 제약이 붙은 타입이나 그러한 타입의 식을 여러의미^{overloaded}라고 말한다. 따라서, $Num\ a \Rightarrow a \rightarrow a \rightarrow a$ 은 여러의미 타입이고 $(+)$ 는 여러의미 함수이다. 일반적으로, 표준 서막에 있는 대부분의 수치를 다루는 함수들은 여러의미 함수다. 예컨대,

$$\begin{aligned} (-) &:: Num\ a \Rightarrow a \rightarrow a \rightarrow a \\ (*) &:: Num\ a \Rightarrow a \rightarrow a \rightarrow a \\ \textit{negate} &:: Num\ a \Rightarrow a \rightarrow a \\ \textit{abs} &:: Num\ a \Rightarrow a \rightarrow a \\ \textit{signum} &:: Num\ a \Rightarrow a \rightarrow a \end{aligned}$$

또한, 각각의 수들 그 자체로 여러의미이다. 예컨대, $3 :: Num\ a \Rightarrow a$ 는 a 가 수치 타입이기만 하다면, 3이라는 수의 타입이 a 가 될 수 있다는 뜻이다.

따라하기

다음은 여러의미 함수에 대해 실습한 화면이다.

```

** WinHugs
File Edit Actions Browse Help
Haskell 98 mode: Restart with command line option -98 to enable extensions

Type ?: for help
Hugs> 1+2
3
Hugs> 1.1+2.2
3.3
Hugs> :type (+)
(+) :: Num a => a -> a -> a
Hugs> :type (-)
(-) :: Num a => a -> a -> a
Hugs> :type (*)
(*) :: Num a => a -> a -> a
Hugs> :type negate
negate :: Num a => a -> a
Hugs> :type abs
abs :: Num a => a -> a
Hugs> :type signum
signum :: Num a => a -> a
Hugs> :type 3
3 :: Num a => a
Hugs>

```

이제 곧 다음 절에서 *Num* 클래스를 비롯한 여러 기본 클래스에 대해 알아볼 것이다.

책에서는 클래스 제약을 나타내는 기호를 \Rightarrow 로 조판하고 있지만 일반 텍스트 편집기에서 스크립트를 편집할 때나 WinHugs 실행 화면에서는 $=>$ 로 나타난다. 이와 같이 책에 나타난 특수 기호를 보통의 글쇠판으로 입력하는 방법을 부록 B에서 찾아볼 수 있다.

따라하기 끝

3.9 기본 클래스

타입이란 서로 관련 있는 값들의 모임이라는 것을 기억하라. 이러한 개념의 연장선상에서, 클래스란 메서드 method라 불리는 특정한 여러의미 연산을 지원하는 타입들의 모임이다. 하스켈 언어에서 제공하는 주어지는 여러 종류의 기본 클래스 중 가장 자주 쓰이는 것을 아래에 설명하였다.

Eq 같기 타입 equality type

이 클래스는 다음의 두 메서드로 같거나 다른을 비교할 수 있는 타입을 포함한다.

$(==) :: a \rightarrow a \rightarrow Bool$
 $(\neq) :: a \rightarrow a \rightarrow Bool$

Bool, Char, String, Int, Integer, 및 Float 와 같은 모든 기본 타입은 *Eq* 클래스 인스턴스이며, 그 원소 타입과 각 성분의 타입이 이 클래스의 인스턴스인 리스트나 순서쌍 타입도 역시 이 클래스의 인스턴스이다. 예컨대,

> $False == False$

True

> $'a' == 'b'$

False

> $"abc" == "abc"$

True

> $[1, 2] == [1, 2, 3]$

False

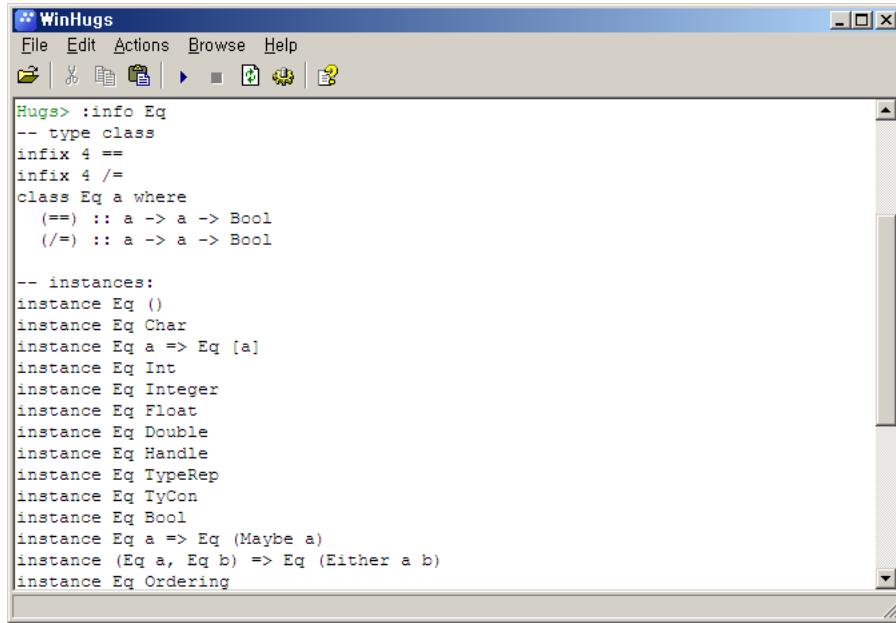
> $('a', False) == ('a', False)$

True

일반적으로 함수 타입은 *Eq* 클래스의 인스턴스가 아니라는 점에 주목하라. 왜냐하면 두 함수가 서로 같은지 비교한다는 것은 그럴싸하지 않은 일이기 때문이다.

따라하기

다음은 WinHugs에서 :info 명령으로 *Eq* 클래스의 정보를 알아본 화면이다.

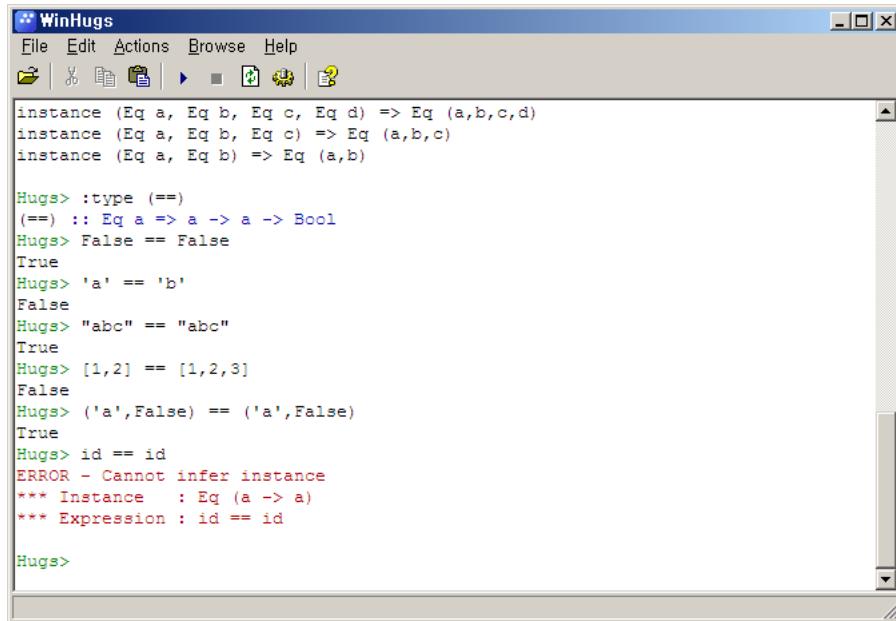


```
Hugs> :info Eq
-- type class
infix 4 ==
infix 4 !=
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

-- instances:
instance Eq ()
instance Eq Char
instance Eq a => Eq [a]
instance Eq Int
instance Eq Integer
instance Eq Float
instance Eq Double
instance Eq Handle
instance Eq TypeRep
instance Eq TyCon
instance Eq Bool
instance Eq a => Eq (Maybe a)
instance (Eq a, Eq b) => Eq (Either a b)
instance Eq Ordering
```

WinHugs에서 :info 명령으로 클래스의 메서드와 인스턴스들을 볼 수 있다.

다음은 *Eq* 클래스에 대해 실습한 화면이다.



```
Hugs> :type (==)
(==) :: Eq a => a -> a -> Bool
Hugs> False == False
True
Hugs> 'a' == 'b'
False
Hugs> "abc" == "abc"
True
Hugs> [1,2] == [1,2,3]
False
Hugs> ('a',False) == ('a',False)
True
Hugs> id == id
ERROR - Cannot infer instance
*** Instance : Eq (a -> a)
*** Expression : id == id

Hugs>
```

함수는 *Eq* 클래스의 인스턴스가 아니므로 잘못이 나는 것을 볼 수 있다.

따라하기 끝

Ord 순서 타입^{ordered type}

이 클래스는 *Eq* 클래스의 인스턴스인 타입들 중에서도 그 값들이 전순서(선형순서)^{totally (linearly) ordered} 를 이루고 있어, 다음의 여섯 메서드로 비교할 수 있는 타입들을 포함한다.

```
(<) :: a → a → Bool
(≤) :: a → a → Bool
(>) :: a → a → Bool
(≥) :: a → a → Bool
min :: a → a → a
max :: a → a → a
```

옮긴이 주: 어떤 두 값이든지 순서를 매길 수 있을 경우를 전순서^{total order} 라고 하며, 이런 경우 순서에 따라 한줄로 나열할 수 있다는 뜻에서 선형순서^{linear order} 라고도 한다.

Bool, *Char*, *String*, *Int*, *Integer*, 및 *Float* 와 같은 모든 기본 타입은 *Ord* 클래스의 인스턴스이며, 그 원소 타입과 각 성분의 타입이 *Ord* 클래스의 인스턴스인 리스트나 순서쌍 타입 역시 *Ord* 클래스의 인스턴스다. 예컨대,

```
> False < True
True

> min 'a' 'b'
'a'

> "elegant" < "elephant"
True

> [1,2,3] < [1,2]
False

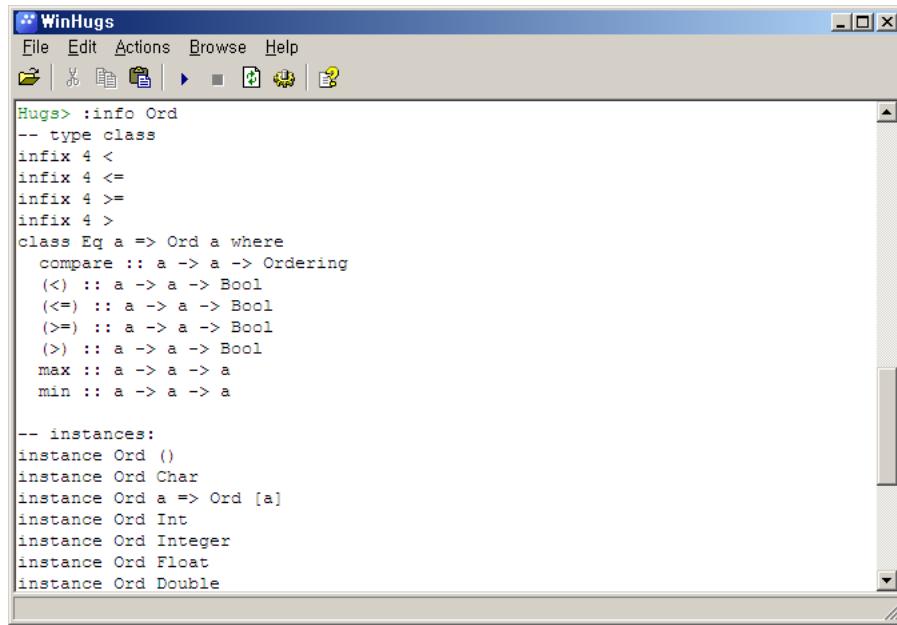
> ('a',2) < ('b',1)
True

> ('a',2) < ('a',1)
False
```

글줄, 리스트, 순서쌍이 사전식 순서^{lexicographic order}를 따름을 알아 두라. 즉, 사전에서 단어가 나타나는 순서와 같은 원리를 따른다. 예컨대, 같은 타입의 두 순서쌍의 경우 첫째 성분에 따라 순서가 결정되며, 첫째 성분이 같은 경우는 둘째 성분에 따라 순서가 결정된다.

따라하기

다음은 WinHugs에서 `:info` 명령으로 `Ord` 클래스의 정보를 알아본 화면이다.



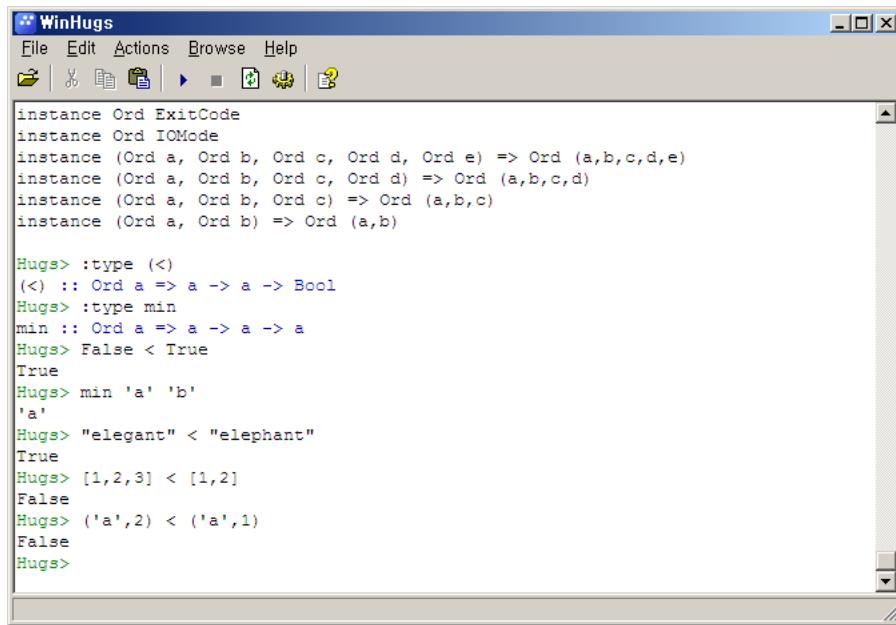
```

WinHugs
File Edit Actions Browse Help
Hugs> :info Ord
-- type class
infix 4 <
infix 4 <=
infix 4 >=
infix 4 >
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a

-- instances:
instance Ord ()
instance Ord Char
instance Ord a => Ord [a]
instance Ord Int
instance Ord Integer
instance Ord Float
instance Ord Double
  
```

WinHugs에서 `:info` 명령으로 클래스의 메서드와 인스턴스들을 볼 수 있다.

다음은 *Ord* 클래스에 대해 실습한 화면이다.



```

WinHugs
File Edit Actions Browse Help
instance Ord ExitCode
instance Ord IOMode
instance (Ord a, Ord b, Ord c, Ord d, Ord e) => Ord (a,b,c,d,e)
instance (Ord a, Ord b, Ord c, Ord d) => Ord (a,b,c,d)
instance (Ord a, Ord b, Ord c) => Ord (a,b,c)
instance (Ord a, Ord b) => Ord (a,b)

Hugs> :type (<)
(<) :: Ord a => a -> a -> Bool
Hugs> :type min
min :: Ord a => a -> a -> a
Hugs> False < True
True
Hugs> min 'a' 'b'
'a'
Hugs> "elegant" < "elephant"
True
Hugs> [1,2,3] < [1,2]
False
Hugs> ('a',2) < ('a',1)
False
Hugs>

```

따라하기 끝

Show 보이는 타입^{showable type} 이 클래스는 다음 메서드를 써서 글줄로 바꿀 수 있는 타입을 포함한다.

show :: *a* → *String*

Bool, *Char*, *String*, *Int*, *Integer*, 및 *Float*와 같은 모든 기본 타입은 *Show* 클래스의 인스턴스이며, 그 원소 타입과 각 성분의 타입이 이 클래스의 인스턴스인 리스트나 순서쌍 타입도 역시 이 클래스의 인스턴스이다. 예컨대,

```

> show False
"False"

> show 'a'
"'a'"

> show 123
"123"

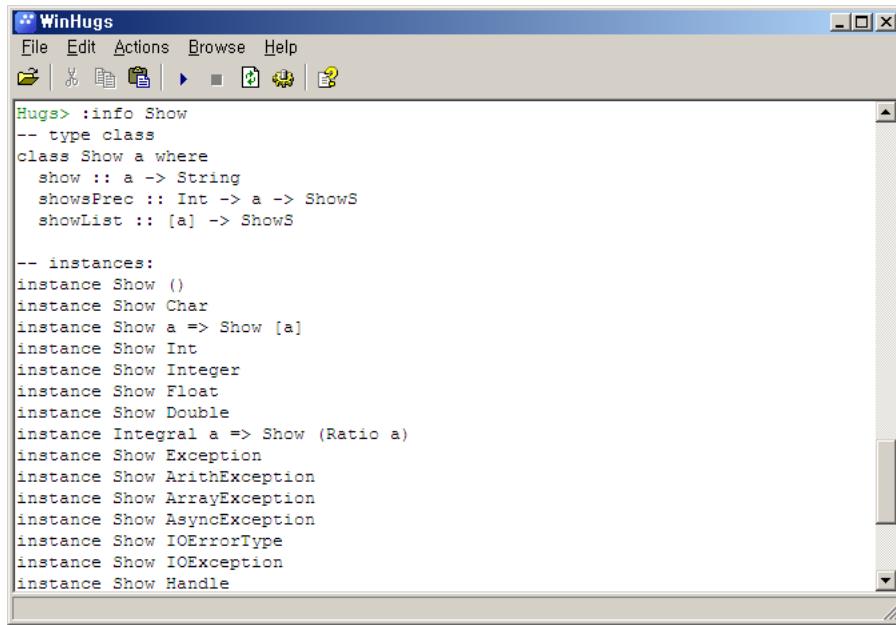
> show [1,2,3]
"[1,2,3]"

> show ('a',False)
"('a',False)"

```

따라하기

다음은 WinHugs에서 :info 명령으로 *Show* 클래스의 정보를 알아본 화면이다.

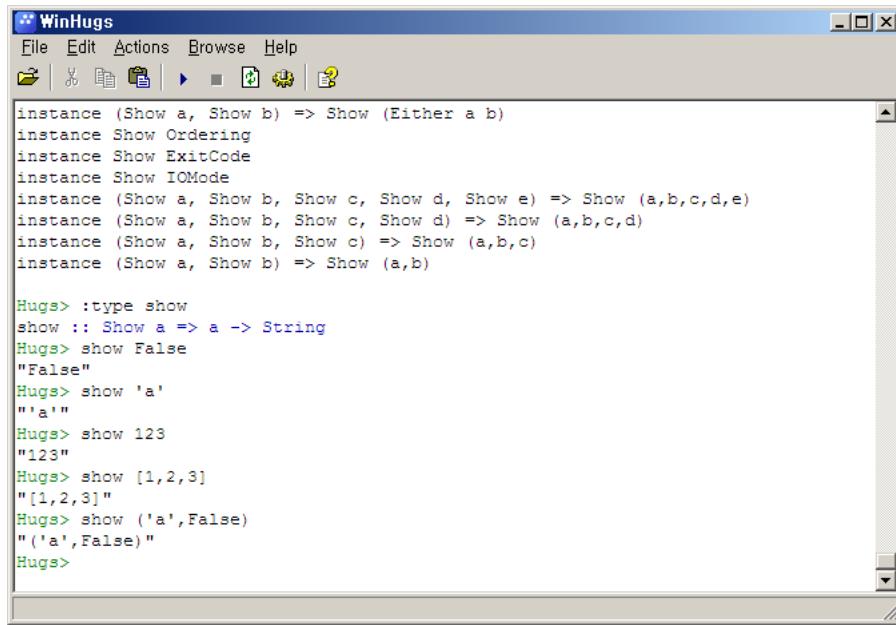


```
*** WinHugs
File Edit Actions Browse Help
| | | | | | | |
Hugs> :info Show
-- type class
class Show a where
  show :: a -> String
  showsPrec :: Int -> a -> ShowS
  showList :: [a] -> ShowS

-- instances:
instance Show ()
instance Show Char
instance Show a => Show [a]
instance Show Int
instance Show Integer
instance Show Float
instance Show Double
instance Integral a => Show (Ratio a)
instance Show Exception
instance Show ArithException
instance Show ArrayException
instance Show AsyncException
instance Show IOErrorType
instance Show IOException
instance Show Handle
```

Show 클래스에는 *show* 외에도 다른 메서드가 둘 더 있다.

다음은 *Show* 클래스에 대해 실습한 화면이다.



```
*** WinHugs
File Edit Actions Browse Help
| | | | | | | |
instance (Show a, Show b) => Show (Either a b)
instance Show Ordering
instance Show ExitCode
instance Show IOMode
instance (Show a, Show b, Show c, Show d, Show e) => Show (a,b,c,d,e)
instance (Show a, Show b, Show c) => Show (a,b,c)
instance (Show a, Show b) => Show (a,b)

Hugs> :type show
show :: Show a => a -> String
Hugs> show False
"False"
Hugs> show 'a'
"'a'"
Hugs> show 123
"123"
Hugs> show [1,2,3]
"[1,2,3]"
Hugs> show ('a',False)
"('a',False)"
Hugs>
```

따라하기 끝

Read 읽히는 타입^{readable type}

이 클래스는 *Show* 와 맞짝을 이루는데, 다음 메서드를 써서 글줄로부터 그 값을 얻어 낼 수 있는 타입을 포함한다.

```
read :: String → a
```

Bool, *Char*, *String*, *Int*, *Integer*, 및 *Float* 와 같은 모든 기본 타입은 *Read* 클래스의 인스턴스이며, 그 원소 타입과 각 성분의 타입이 이 클래스의 인스턴스인 리스트나 순서쌍 타입도 역시 이 클래스의 인스턴스이다. 예컨대,

```
> read "False" :: Bool  
False
```

```
> read "'a'" :: Char  
'a'
```

```
> read "123" :: Int  
123
```

```
> read "[1,2,3]" :: [Int]  
[1,2,3]
```

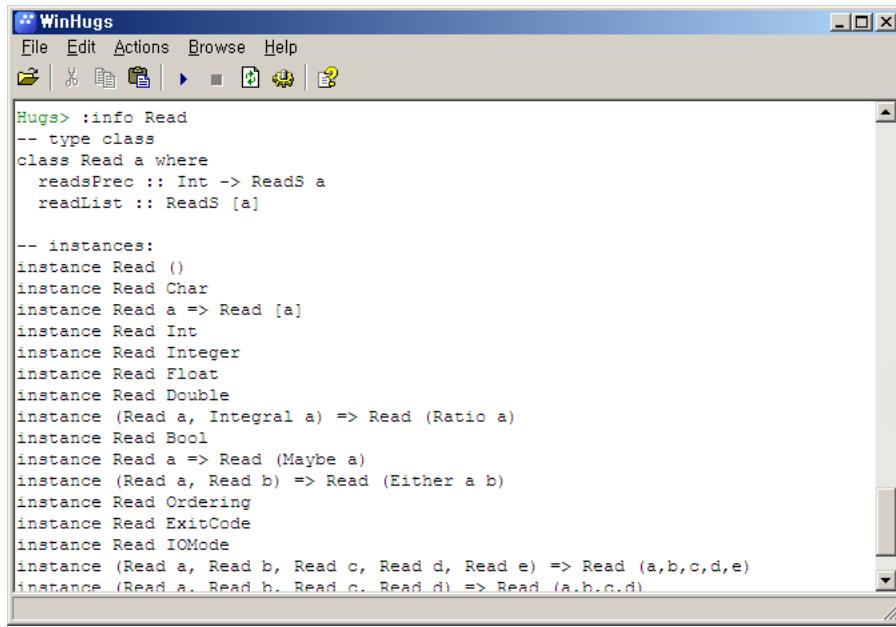
```
> read "('a',False)" :: (Char,Bool)  
('a',False)
```

위 보기에서는 :: 를 써서 결과값의 타입이 무엇인지 지정했다. 하지만 실제로, 그런 타입 정보를 문맥상 자동으로 유추해 낼 수 있는 경우가 많다. 예컨대, $\neg(\text{read } \text{"False"})$ 와 같은 식에서는 타입 정보를 드러나 보이게 적을 필요가 없는데, 이는 논리 부정 함수 \neg 으로 인해 *read "False"* 의 타입이 *Bool* 이어야만 한다는 것을 알 수 있기 때문이다.

함수 *read* 는 문법적으로 올바르지 못한 인자에 대해서는 그 돌려주는 값이 정의되지 않는다는 사실에 주목하라. 예컨대, $\neg(\text{read } \text{"hello"})$ 와 같은 식을 계산하면 잘못이 나게 되며, 이는 "hello" 를 어떤 논리값으로서 읽어들일 수 없기 때문이다.

따라하기

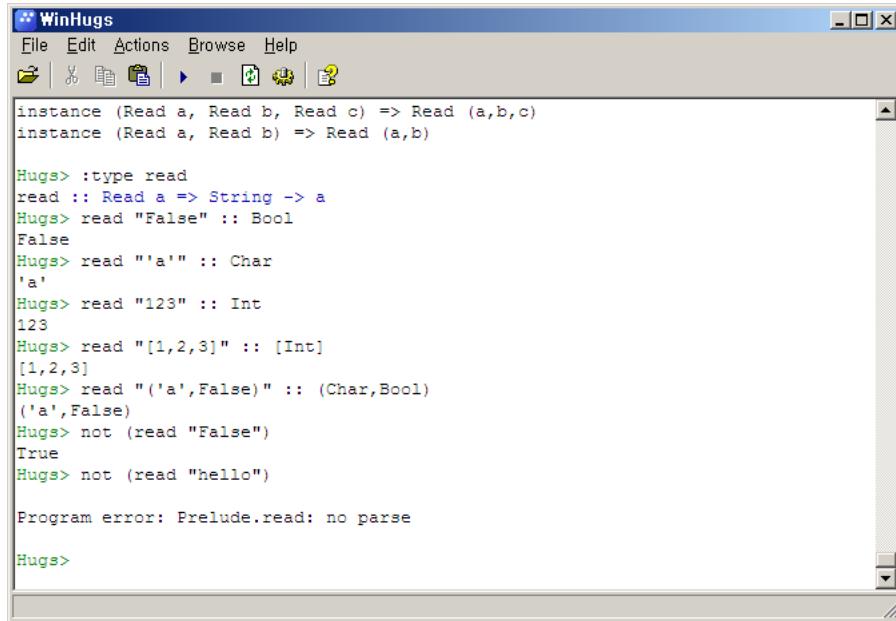
다음은 WinHugs에서 :info 명령으로 *Show* 클래스의 정보를 알아본 화면이다.



```
*** WinHugs
File Edit Actions Browse Help
| | | | | | | | | |
Hugs> :info Read
-- type class
class Read a where
  readsPrec :: Int -> ReadS a
  readList :: ReadS [a]

-- instances:
instance Read ()
instance Read Char
instance Read a => Read [a]
instance Read Int
instance Read Integer
instance Read Float
instance Read Double
instance (Read a, Integral a) => Read (Ratio a)
instance Read Bool
instance Read a => Read (Maybe a)
instance (Read a, Read b) => Read (Either a b)
instance Read Ordering
instance Read ExitCode
instance Read IOMode
instance (Read a, Read b, Read c, Read d, Read e) => Read (a,b,c,d,e)
instance (Read a, Read b, Read c, Read d) => Read (a,b,c,d)
```

*read*는 실제로는 *Read* 클래스의 메서드는 아니며 *Read* 클래스의 메서드를 이용해 정의된 함수이다.
다음은 *Read* 클래스에 대해 실습한 화면이다.



```
*** WinHugs
File Edit Actions Browse Help
| | | | | | | | | |
Hugs> :type read
read :: Read a => String -> a
Hugs> read "False" :: Bool
False
Hugs> read "'a'" :: Char
'a'
Hugs> read "123" :: Int
123
Hugs> read "[1,2,3]" :: [Int]
[1,2,3]
Hugs> read "('a',False)" :: (Char,Bool)
('a',False)
Hugs> not (read "False")
True
Hugs> not (read "hello")

Program error: Prelude.read: no parse

Hugs>
```

따라하기 끝

Num 수치 타입^{numeric type}

이 클래스는 같기 클래스 *Eq*의 인스턴스이면서 보이는 클래스 *Show*의 인스턴스 중에서도, 다음 여섯 메서드로 처리할 수 있는 수치값 타입을 포함한다.

(+)	:: <i>a</i> → <i>a</i> → <i>a</i>
(-)	:: <i>a</i> → <i>a</i> → <i>a</i>
(*)	:: <i>a</i> → <i>a</i> → <i>a</i>
<i>negate</i>	:: <i>a</i> → <i>a</i>
<i>abs</i>	:: <i>a</i> → <i>a</i>
<i>signum</i>	:: <i>a</i> → <i>a</i>

(*negate* 메서드는 수의 부호를 바꾸고, *abs* 메서드는 수의 절대값^{Absolute value}을 돌려주며, *signum* 메서드는 수의 부호^{signature}를 돌려준다.) 기본 타입인 *Int*, *Integer*, 및 *Float*가 *Num* 클래스의 인스턴스이다. 예컨대,

```
> 1 + 2
3
```

```
> 1.1 + 2.2
3.3
```

```
> negate 3.3
-3.3
```

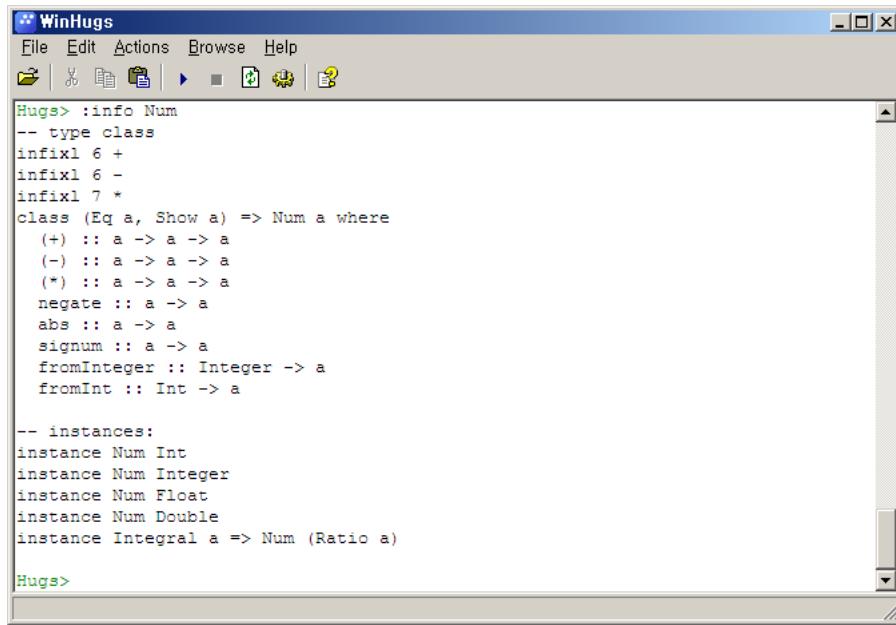
```
> abs (-3)
3
```

```
> signum (-3)
-1
```

Num 클래스에 나눗셈 메서드가 없다는 사실에 주목하라. 앞으로 살펴보겠지만, 나눗셈은 정수같은 수^{integral numbers} 일 때와 분수같은 수^{fractional numbers} 일 때의 두 경우 각각에 대해 별도의 클래스로써 나누어 처리한다.

따라하기

다음은 WinHugs에서 :info 명령으로 Num 클래스의 정보를 알아본 화면이다.



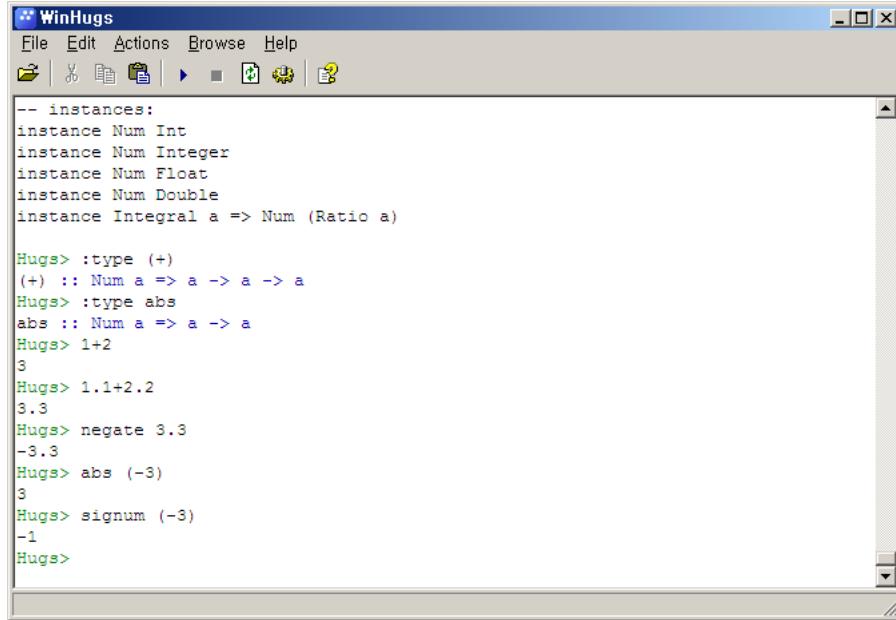
```
Hugs> :info Num
-- type class
infixl 6 +
infixl 6 -
infixl 7 *
class (Eq a, Show a) => Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  fromInt :: Int -> a

-- instances:
instance Num Int
instance Num Integer
instance Num Float
instance Num Double
instance Integral a => Num (Ratio a)

Hugs>
```

Num 클래스에는 Integer 와 Int 를 임의의 수치 타입으로 변환하는 fromInteger 와 fromInt 메서드도 있음을 눈여겨 보라.

다음은 Num 클래스에 대해 실습한 화면이다.



```
-- instances:
instance Num Int
instance Num Integer
instance Num Float
instance Num Double
instance Integral a => Num (Ratio a)

Hugs> :type (+)
(+) :: Num a => a -> a -> a
Hugs> :type abs
abs :: Num a => a -> a
Hugs> 1+2
3
Hugs> 1.1+2.2
3.3
Hugs> negate 3.3
-3.3
Hugs> abs (-3)
3
Hugs> signum (-3)
-1
Hugs>
```

따라하기 끝

Integral 정수같은 타입 integral type

이 클래스는 *Num* 클래스의 인스턴스 중에서도, 그 값이 정수같은 수라서 정수 나눗셈 정수 나머지 메서드로 연산이 가능한 타입을 포함한다.

```
div :: a → a → a
mod :: a → a → a
```

위 두 메서드를 실제로 쓸 때는 메서드 이름을 역따옴표 안에 써서 두 인자의 사이에 쓰는 경우가 많다. 기본 타입인 *Int* 와 *Integer* 가 *Integral* 클래스의 인스턴스이다. 예컨대,

```
> 7 `div` 2
3
```

```
> 7 `mod` 2
1
```

리스트와 정수를 다루는 (*length*, *take*, *drop* 같은) 다수의 표준 서막 함수들은 그 의미상 *Integral* 클래스의 아무 인스턴스 타입을 다루어도 되지만, 효율상의 문제를 고려하여, 고정 크기 정수 타입인 *Int* 를 쓰도록 제한하고 있다. 하지만, 필요에 따라, 표준 라이브러리 *List* 모듈에서 제공하는 그런 일반적인 판의 함수들을 쓸 수 있다.

따라하기

다음은 WinHugs에서 :info 명령으로 *Integral* 클래스의 정보를 알아본 화면이다.

The screenshot shows the WinHugs application window. The menu bar includes File, Edit, Actions, Browse, and Help. Below the menu is a toolbar with icons for file operations. The main window contains a text area with the following content:

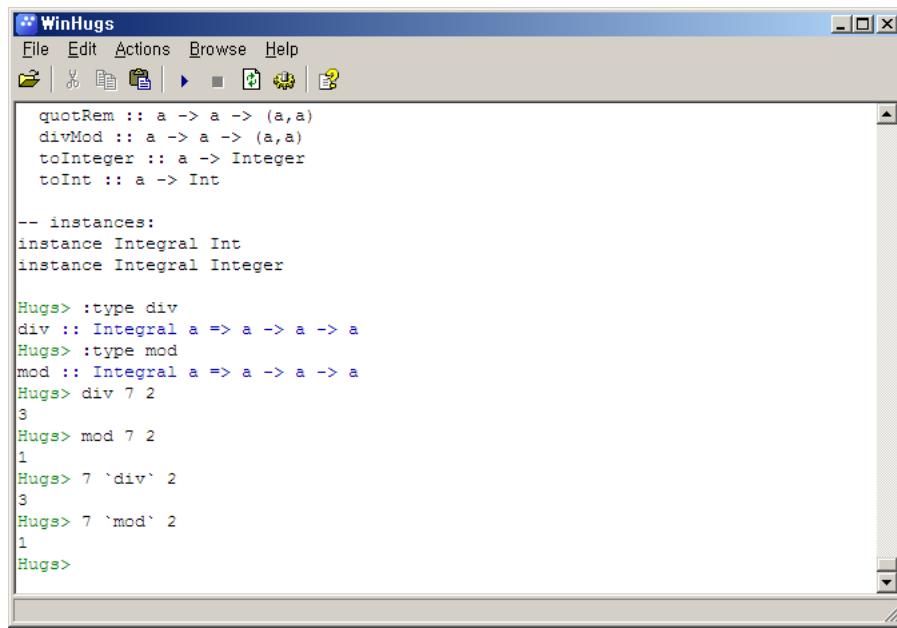
```
Hugs> :info Integral
-- type class
infixl 7 `quot'
infixl 7 `rem'
infixl 7 `div'
infixl 7 `mod'
class (Real a, Enum a) => Integral a where
  quot :: a -> a -> a
  rem :: a -> a -> a
  div :: a -> a -> a
  mod :: a -> a -> a
  quotRem :: a -> a -> (a,a)
  divMod :: a -> a -> (a,a)
  toInteger :: a -> Integer
  toInt :: a -> Int

-- instances:
instance Integral Int
instance Integral Integer

Hugs>
```

기본 타입 중 *Integral* 클래스의 인스턴스는 *Int* 와 *Integer* 뿐이며, *Integral* 클래스의 메서드 중에 임의의 정수 같은 타입을 *Integer* 와 *Int* 로 변환하는 *toInteger* 와 *toInt* 메서드도 있음을 눈여겨 보라.

다음은 *Integral* 클래스에 대해 실습한 화면이다.



```

WinHugs
File Edit Actions Browse Help
quotRem :: a -> a -> (a,a)
divMod :: a -> a -> (a,a)
toInteger :: a -> Integer
toInt :: a -> Int

-- instances:
instance Integral Int
instance Integral Integer

Hugs> :type div
div :: Integral a => a -> a -> a
Hugs> :type mod
mod :: Integral a => a -> a -> a
Hugs> div 7 2
3
Hugs> mod 7 2
1
Hugs> 7 `div` 2
3
Hugs> 7 `mod` 2
1
Hugs>

```

div 와 *mod* 는 원래 보통의 여러의미 함수지만, 다른 함수와 마찬가지로 역따옴표로 감싸면 두 인자 사이에 쓰는 연산자로 쓸 수 있음을 눈여겨 보라.

따라하기 끝

Fractional 분수같은 타입 fractional type

이 클래스는 수치 클래스 *Num* 의 인스턴스 중에서도, 그 값이 정수 같지 않아서 나눗셈 및 역수를 계산하는 다음 메서드를 적용할 수 있는 타입을 포함한다.

```
(/) :: a -> a -> a
recip :: a -> a
```

기본 타입 *Float* 가 *Fractional* 클래스의 인스턴스이다. 예컨대,

```
> 7.0 / 2.0
3.5
```

```
> recip 2.0
0.5
```

따라하기

다음은 WinHugs에서 `:info` 명령으로 *Fractional* 클래스의 정보를 알아본 다음 *Fractional* 클래스에 대해 실습한 화면이다.

The screenshot shows the WinHugs Haskell interpreter interface. The menu bar includes File, Edit, Actions, Browse, Help, and a toolbar with icons for file operations. The main window displays the following text:

```
Hugs> :info Fractional
-- type class
infixl 7 /
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
  fromDouble :: Double -> a

-- instances:
instance Fractional Float
instance Fractional Double
instance Integral a => Fractional (Ratio a)

Hugs> :type (/)
(/) :: Fractional a => a -> a -> a
Hugs> :type recip
recip :: Fractional a => a -> a
Hugs> 7.0/2.0
3.5
Hugs> recip 2.0
0.5
Hugs>
```

Fractional 클래스의 인스턴스로 단일정밀도 부동소수점 수 타입인 *Float* 외에 두배정밀도 부동소수점 수 타입인 *Double*도 있음을 눈여겨 보라.

따라하기 끝

3.10 살펴보기

논리값 타입을 나타내는 *Bool*은 기호 논리학symbolic logic 분야에서 선구적인 업적을 남긴 조지 부울George Boole을 기리는 뜻에서 따온 것이며, 인자를 한 번에 하나씩 받는 함수를 일컬어 커리되었다curried 고 하는 용어는 그런 함수에 대한 하스켈 커리Haskell Curry(하스켈 언어의 이름도 이 사람에게서 따온 것)의 업적을 기리는 뜻에서 따온 것이다. 하스켈의 타입 시스템에 대한 더 자세한 설명은 하스켈 보고서(25)에 나와 있으며, 전문기를 위한 엄밀한 형식적 기술은 (20; 6)에서 찾을 수 있다.

3.11 연습문제

1. 다음 값의 타입은 각각 무엇인가?

['a', 'b', 'c']
 ('a', 'b', 'c')
 [(False, '0'), (True, '1')]
 ([False, True], ['0', '1'])
 [tail, init, reverse]

2. 다음 함수의 타입은 각각 무엇인가?

<i>second xs</i>	= <i>head (tail xs)</i>
<i>swap (x, y)</i>	= <i>(y, x)</i>
<i>pair x y</i>	= <i>(x, y)</i>
<i>double x</i>	= <i>x * 2</i>
<i>palindrome xs</i>	= <i>reverse xs == xs</i>
<i>twice f x</i>	= <i>f (f x)</i>

귀띔: 함수 정의에 여러의미 overloaded 연산자가 쓰였다면, 함수 타입에 그에 알맞은 클래스 제약을 빼뜨리지 말라.

3. 위의 문제들을 제대로 풀었는지 Hugs를 써서 확인해 보라.

4. 어째서 일반적으로 함수 타입을 *Eq* 클래스의 인스턴스 사용할 수 없는가? 그렇다면 어떤 경우에 함수 타입을 *Eq* 클래스의 인스턴스로 사용할 수 있겠는가?

귀띔: 두 함수가 같으려면 타입이 같아야 함은 물론, 같은 인자에 대해 항상 같은 결과를 돌려주어야 한다.

제 4 장

함수 정의

이 장에서는 하스켈에서 함수를 정의하는 여러가지 방법을 알아본다. 먼저 조건식conditional expression과 보초선 등
식guarded equation에 대해 알아보고, 간단하지만 강력한 패턴 매칭이라는 개념을 소개한 후, 람다식lambda expression
의 개념과 잘린식을 살펴보는 것으로 이 장을 맺는다.

4.1 새것을 옛것으로부터

새로운 함수를 정의하는 가장 간단하고 쉬운 방법은 아마도 하나 또는 그 이상의 이미 만들어 놓은 함수를 결합
하는 것이다. 예컨대, 아래와 같은 여러 라이브러리 함수들을 이런 방법으로 정의하였다.

- 글자가 숫자인자 알아본다.

```
isDigit  :: Char → Bool
isDigit c = c ≥ '0' ∧ c ≤ '9'
```

- 정수가 짝수인지 알아본다.

```
even    :: Integral a => a → Bool
even n = n `mod` 2 == 0
```

- 리스트를 n 번째 원소 위치에서 둘로 나눈다.

```
splitAt     :: Int → [a] → ([a], [a])
splitAt n xs = (take n xs, drop n xs)
```

- 역수^{reciprocal}를 구한다.

```
recip    :: Fractional a => a → a
recip n = 1 / n
```

*even*과 *recip*의 타입에 붙은 클래스 제약^{class constraint}에 주의하라. 이 클래스 제약들은 각각의 함수들을 정수 같은 타입과 분수같은 타입에 적용할 수 있음을 명확하게 드러낸다.

따라하기

다음은 라이브러리 함수들을 *fundef.hs*에 정의한 화면이다.

```
fundef.hs (K:WMyDoc) - GVIM
파일(E) 편집(E) 도구(I) 문법(S) 버퍼(B) 창(W) 도움말(H)
isDigit'    :: Char → Bool
isDigit' c = c >= '0' && c <= '9'

even'      :: Integral a => a → Bool
even' n = n `mod` 2 == 0

splitAt'   :: Int → [a] → ([a], [a])
splitAt' n xs = (take n xs, drop n xs)

recip'     :: Fractional a => a → a
recip' n = 1 / n
"fundef.hs" 12L, 269C 저장 했습니다          3, 0-1      빠대기
```

이름 충돌을 피하기 위해 일부러 이름 끝에 따옴표(')를 하나씩 붙였다. 예컨대 *even*을 그냥 정의하면 이미 있는 변수 이름이므로 다시 정의할 수 없지만 *even'*은 정의할 수 있다. 앞으로 이 장에서 살펴볼 다른 대부분의 보기도 표준 라이브러리 함수를 다시 정의해 보는 것이므로 이와 같이 이름을 약간 달리하여 정의해야 문제없이 실습할 수 있을 것이다.

다음은 *fundef.hs* 를 WinHugs에서 불려들여 실습한 화면이다.

```

WinHugs
File Edit Actions Browse Help
Type ?: for help
Main> :type isDigit'
isDigit' :: Char -> Bool
Main> isDigit' '7'
True
Main> isDigit' 'b'
False
Main> :type even'
even' :: Integral a => a -> Bool
Main> even' 2
True
Main> even' 3
False
Main> :type splitAt'
splitAt' :: Int -> [a] -> ([a], [a])
Main> splitAt' 2 [0,1,2,3]
([0,1],[2,3])
Main> :type recip'
recip' :: Fractional a => a -> a
Main> recip' 5.0
0.2
Main>

```

지금까지는 Hugs 사용법을 어려워할지 모르는 독자들을 위해 모든 보기마다 일일이 따라하기로 화면을 보여주며 추가로 설명을 덧붙였다. 하지만 책의 원래 내용상 설명이 충분히 자세하고 독자들도 이미 4장까지 읽어오면서 Hugs 사용법에 어느 정도 익숙해졌을 것이므로 앞으로는 몇몇 보기만 골라 따라하기 설명을 하겠다.

따라하기 끝

4.2 조건식 conditional expression

하스켈에서는 함수가 여러 가능한 값들 중 하나를 결과값으로 고르도록 정의할 때 쓸 수 있는 다양한 방법을 제공한다. 그 중 가장 간단한 것이 조건식으로, 조건 condition 이라 불리는 논리식 logical expression 의 값에 따라 같은 타입의 두 결과값 중 하나를 선택한다. 만약 조건이 참 *True* 이면 첫째 결과값을 취하고, 거짓 *False* 이면 둘째 결과를 취한다. 예컨대, 정수의 절대값을 구하는 라이브러리 함수 *abs* 를 다음과 같이 정의할 수 있다.

```

abs   :: Int → Int
abs n = if n ≥ 0 then n else - n

```

조건식은 포개어질 수 있는데, 이는 조건식이 다른 조건식을 포함할 수 있다는 뜻이다. 예컨대, 정수의 부호를 구하는 라이브러리 함수 *signum*을 다음과 같이 정의 할 수 있다.

```
signum    :: Int → Int
signum n = if n < 0 then -1 else
            if n == 0 then 0 else 1
```

몇몇 프로그래밍 언어들과는 달리, 하스켈에서는 조건식이 언제나 *else* 갈래를 가지므로 조건식이 동강나는^{dangling else} 잘 알려진 문제가 생길 일이 아예 없다. 예컨대, 하스켈에서 *else* 갈래가 없어도 된다면 *if True then if False then 1 else 2* 와 같은 이러한 조건식이 가능한데, 끝에 하나만 쓰인 *else* 갈래가 안쪽 조건식에 붙느냐 바깥쪽 조건식에 붙느냐에 따라 그 값이 2가 되든지 잘못이 나든지 할 것이다.

4.3 보초선 등식 guarded equation

함수를 조건식 대신 보초선 등식으로 정의할 수도 있다. 보초선 등식에는 보초^{guard} 라 불리는 일련의 논리식이 있어 같은 타입의 여러 결과들 중의 하나를 취하도록 할 수 있다. 첫째 보초가 참이면 첫째 결과를 취하고, 그렇지 않고 둘째 보초가 참이면 둘째 결과를 취하고, 이런 식으로 계속한다. 예컨대, *abs* 라이브러리 함수를 다음과 같이 정의할 수도 있다.

```
abs n | n ≥ 0      = n
      | otherwise = -n
```

기호 $|$ 는 “만족하는”이라고 읽고, *otherwise* 보초는 표준서막^{standard prelude}에서 그냥 *otherwise = True*로 정의한다. 마지막 보초로 꼭 *otherwise*를 세울 필요는 없지만, “나머지 모든 경우”를 편하게 처리할 수 있으므로 통과할 수 있는 보초가 하나도 없어서 (즉, 참^{True} 인 보초가 하나도 없어서) 잘못^{error}이 나는 것을 확실히 막을 수 있다.

보초가 여러 개 있어도 읽기 쉽다는 것이 조건식과 견주어 볼 때 보초선 등식의 주된 장점이다. 예컨대, 라이브러리 함수 *signum*을 아래와 같이 정의하면 더 이해하기 쉽다.

$$\begin{array}{l} \text{signum } n \mid n < 0 = -1 \\ \quad \mid n == 0 = 0 \\ \quad \mid \text{otherwise} = 1 \end{array}$$

4.4 패턴 매칭 pattern matching

많은 함수들이 패턴 매칭으로 정의했을 때 특히 간단하고 직관적이다. 패턴 매칭에서는 일련의 패턴이 있으며, 이 패턴이라 불리는 문법 표현에 어떤 값이 들어맞는지에 따라 같은 타입의 결과들 가운데 하나를 취하도록 할 수 있다. 첫째 패턴이 들어맞으면 첫째 결과를 취하고, 그렇지 않고 둘째 패턴에 들어맞으면 둘째 결과를 취하고, 이런 식으로 계속한다. 예컨대, 라이브러리 함수 \neg 은 논리값에서 부정을 다음과 같이 보여준다.

$$\begin{array}{ll} \neg & :: \text{Bool} \rightarrow \text{Bool} \\ \neg \text{False} & = \text{True} \\ \neg \text{True} & = \text{False} \end{array}$$

인자가 여러 개인 함수도 패턴 매칭으로 정의할 수 있다. 같은 등식 안에 있는 패턴은 나타나는 순서대로 왼쪽부터 차례로 맞춰 보게 된다. 예컨대, 두 논리값의 논리합을 구하는 라이브러리 연산자 \wedge 를 다음과 같이 정의할 수 있다.

$$\begin{array}{ll} (\wedge) & :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \\ \text{True} \wedge \text{True} & = \text{True} \\ \text{True} \wedge \text{False} & = \text{False} \\ \text{False} \wedge \text{True} & = \text{False} \\ \text{False} \wedge \text{False} & = \text{False} \end{array}$$

하지만, 위 정의에서 마지막 세 등식은 아무런 값이나 들어맞는 아무거나 패턴 wildcard $_$ 을 써서 다음과 같이 두 인자의 값에 관계없이 $False$ 를 돌려주도록 하나의 등식으로 간단히 할 수 있다.

$$\begin{aligned} True \wedge True &= True \\ - \wedge - &= False \end{aligned}$$

이 정의는 또한 느긋한 계산법에서 첫째 인자가 거짓이면 둘째 인자를 계산할 필요 없이 결과가 거짓임을 계산할 수 있다는 점에서도 더 좋다. 실제 표준 서막(standard prelude)에 있는 \wedge 도 이와 같은 성질을 갖지만, 첫째 인자의 값에 따라 어느 등식을 적용할지 결정할 수 있도록 다음과 같이 정의하고 있다.

$$\begin{aligned} True \wedge b &= b \\ False \wedge - &= False \end{aligned}$$

즉, 첫째 인자가 참이면 결과는 둘째 인자의 값이 된다. 첫번째 인자가 거짓이면 결과는 거짓이다.

기술적인 이유로, 한 등식에서 하나 이상의 인자에 똑같은 이름을 붙이지 못하도록 하는 것에 유의하라. 예컨대, 다음과 같은 \wedge 연산자 정의는 두 인자가 서로 같다면 결과는 같은 값이고 아니면 결과는 거짓이라는 성질을 나타내려 하고 있으나, 바로 앞에서 이야기한 이름 붙이는 규칙을 따르지 않으므로 잘못된 정의다.

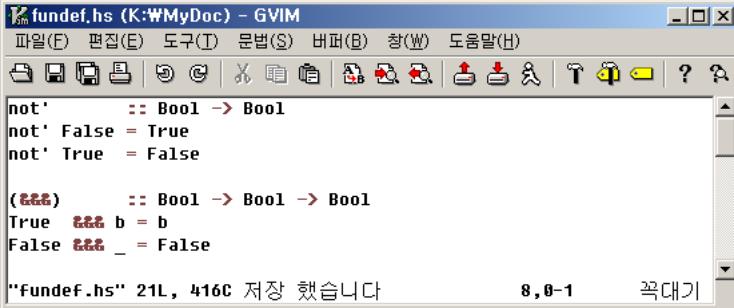
$$\begin{aligned} b \wedge b &= b \\ - \wedge - &= False \end{aligned}$$

필요하다면, 위 정의를 다음과 같이 두 인자가 같은지 검사하는 보초 guard 를 써서 다음과 같이 올바른 정의로 고쳐 쓸 수 있다.

$$\begin{aligned} b \wedge c \mid b == c &= b \\ \mid otherwise &= False \end{aligned}$$

따라하기

다음은 논리 연산과 관련된 라이브러리 함수들을 *fundef.hs*에 정의한 화면이다.



```

fundef.hs (K:WMyDoc) - GVIM
파일(F) 편집(E) 도구(I) 문법(S) 버퍼(B) 창(W) 도움말(H)
문서 새 문서 새 창 파일 관리 편집 키보드 툴바 도구 도움말
not'      :: Bool -> Bool
not' False = True
not' True  = False

(&&)     :: Bool -> Bool -> Bool
True && b = b
False && _ = False

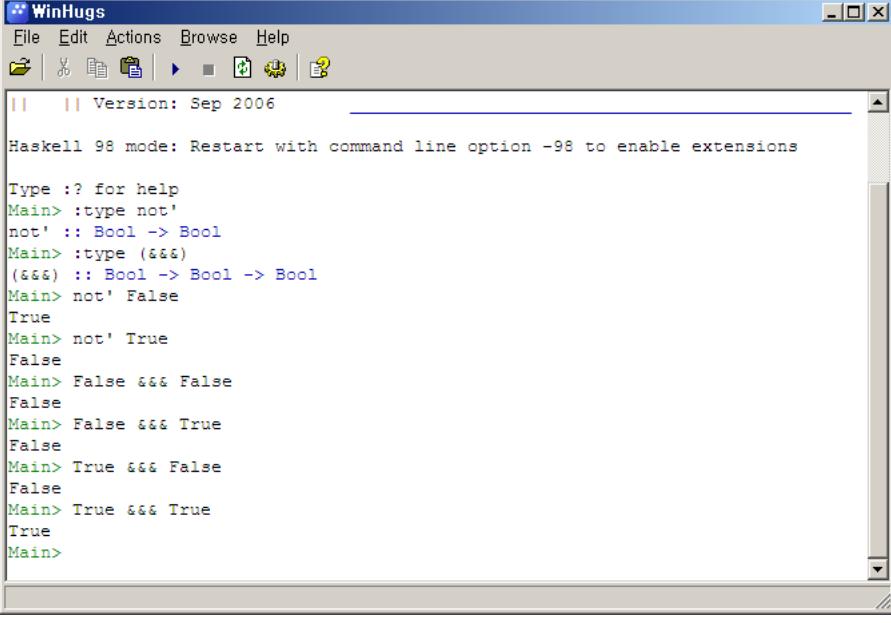
"fundef.hs" 21L, 416C 저장 했습니다      8, 0-1      최근에

```

앞서와 마찬가지로 이름 충돌을 피하기 위해 \neg 대신 *not'*을, \wedge 대신 *&&*를 썼다.

책에서는 논리곱 연산자를 \wedge 로 조판하고 있지만 일반 텍스트 편집기에서 스크립트를 편집할 때나 WinHugs 실행 화면에서는 *&&*로 나타난다. 이와 같이 책에 나타난 특수 기호를 보통의 글쇠판으로 입력하는 방법을 부록 B에서 찾아볼 수 있다.

다음은 *fundef.hs*를 WinHugs에서 불러들여 실습한 화면이다.



```

WinHugs
File Edit Actions Browse Help
File | Open | Save | Print | Run | Exit | Help | About | ? | 
|| || Version: Sep 2006
Haskell 98 mode: Restart with command line option -98 to enable extensions
Type :? for help
Main> :type not'
not' :: Bool -> Bool
Main> :type (&&)
(&&) :: Bool -> Bool -> Bool
Main> not' False
True
Main> not' True
False
Main> False && False
False
Main> False && True
False
Main> True && False
False
Main> True && True
True
Main>

```

따라하기 끝

순서쌍 패턴

패턴의 순서쌍 또한 패턴이다. 순서쌍 패턴은 성분의 개수가 같은 아무 순서쌍과도 들어맞을 수 있는데, 순서쌍의 성분을 순서쌍 패턴에 나타나는 각각의 패턴과 차례대로 맞춰 나간다. 예컨대, 라이브러리 함수 *fst* 와 *snd* 는 아래와 같이 각각 순서쌍의 첫째와 둘째 성분을 취하도록 정의한다.

$$\begin{aligned} fst &:: (a, b) \rightarrow a \\ fst(x, -) &= x \\ snd &:: (a, b) \rightarrow b \\ snd(-, y) &= y \end{aligned}$$

리스트 패턴

순서쌍 패턴과 마찬가지로, 패턴으로 이루어진 리스트 또한 패턴이다. 리스트 패턴은 같은 길이의 리스트와 들어맞는데, 리스트의 모든 원소들을 순서대로 리스트 패턴에 있는 각각의 패턴과 맞춰 나간다. 예컨대, 리스트가 세 개의 글자로 이루어졌으며 'a'로 시작하는지 검사하는 *test* 함수를 다음과 같이 정의할 수 있다.

$$\begin{aligned} test &:: [Char] \rightarrow Bool \\ test['a', -, -] &= True \\ test_ &= False \end{aligned}$$

지금까지는 리스트를 하스켈의 기본 개념으로 간주했다. 사실, 리스트는 그렇게 기본적인 개념이 아니라 빈 리스트 []로부터 시작하여 콘스^{cons} 라 불리는 연산자 :를 써서 한 번에 한 원소씩 기존 리스트의 맨 앞에 붙여나가는 방식으로 만들어 나간다. 예컨대, 리스트 [1, 2, 3]를 다음과 같이 분해할 수 있다.

$$\begin{aligned} [1, 2, 3] &= \{ \text{리스트 풀어쓰기} \} \\ &= 1 : [2, 3] \\ &= \{ \text{리스트 풀어쓰기} \} \\ &= 1 : (2 : [3]) \\ &= \{ \text{리스트 풀어쓰기} \} \\ &= 1 : (2 : (3 : [])) \end{aligned}$$

리스트 $[1, 2, 3]$ 는 $1 : (2 : (3 : []))$ 를 줄인 표현이다. 리스트를 이렇게 표시할 때 괄호가 너무 많아지는 것을 피하고자 콘스 연산자는 오른쪽으로부터 묶이는 것으로 한다. 예컨대, $1 : 2 : 3 : []$ 는 $1 : (2 : (3 : []))$ 를 뜻한다.

리스트를 만들 때 쓰는 콘스 연산자를 비어 있지 않은 리스트와 들어맞는 패턴을 만드는 데에도 쓸 수 있다. 이 패턴은 리스트의 첫째 원소와 첫 원소를 제외한 나머지 리스트에 각각 들어맞는다. 예컨대, 길이에 관계없이 리스트가 'a'로 시작하는지 알아보는 더 일반적인 *test* 를 함수를 다음과 같이 정의할 수 있다.

```
test      :: [Char] → Bool
test ('a' : _) = True
test _       = False
```

마찬가지로, 리스트가 비어 있는지 알아보는 *null*, 비어 있지 않은 리스트의 첫째 원소를 취하는 *head*, 비어 있지 않은 리스트의 첫째 원소를 버리는 *tail* 과 같은 라이브러리 함수들을 다음과 같이 정의한다.

```
null     :: [a] → Bool
null []   = True
null (_ : _) = False
head      :: [a] → a
head (x : _) = x
tail      :: [a] → [a]
tail (_ : xs) = xs
```

함수 적용이 어떤 연산자보다도 높은 우선순위를 가지기 때문에, 콘스 패턴을 쓸 때는 반드시 괄호로 감싸야 한다는 점에 유의하라. 예컨대, $\text{tail } _ : xs = xs$ 정의 할 때 괄호가 없다면 $(\text{tail } _) : xs = xs$ 와 같은 뜻을 나타낸다. 이것은 뜻도 맞지 않고 잘못된 정의다.

정수 패턴

하스켈에서는 n 이 정수를 나타내는 변수이고 $k > 0$ 인 상수일 때 $n + k$ 라는 정수 패턴을 쓸 수 있는데, 이는 특별한 경우에 가끔 쓸모가 있다.

옮긴이 주: 정수 패턴 혹은 $n + k$ 패턴은 예전부터 논란이 많아 다음 하스켈 표준 Haskell'에서는 제외할 가능성이 크다. 이 책에서는 $n + k$ 패턴을 자주 사용하지만, 책의 보기나 연습문제 외에 업무나 취미로 작성하는 프로그램에서는 다음 하스켈 표준과 호환을 위해 정수 패턴을 쓰지 말 것을 권장한다.

예컨대, 함수 0은 그대로 0으로 양수는 그보다 하나 작은 바로 앞 수로 대응시키는 함수 pred 를 다음과 같이 정의할 수 있다.

$$\begin{aligned}\text{pred} &:: \text{Int} \rightarrow \text{Int} \\ \text{pred } 0 &= 0 \\ \text{pred } (n + 1) &= n\end{aligned}$$

옮긴이 주: 여기서 정의한 pred 는 음수가 아닌 경우에만 동작하지만 실제 표준 서마 pred 의 정의는 이와는 조금 다르다. 예컨대, $\text{pred } (-1)$ 과 같이 음수에도 pred 를 적용할 수 있으며 $\text{pred } 0$ 의 값은 -1 이다. 그리고 pred 는 *Enum* 클래스의 메서드인 여러의미 함수로 그 타입은 $\text{pred} :: \text{Enum } a \Rightarrow a \rightarrow a$ 이다.

$n + k$ 패턴을 쓸 때 유의할 점이 두 가지 있다. 첫째, 정수가 k 보다 크거나 같을 경우에만 들어맞는다. 예컨대, 함수 $\text{pred } (-1)$ 를 계산하려면 잘못이 난다. 왜냐하면, 정의에 쓰인 두 패턴이 모두 음의 정수와는 들어맞지 않기 때문이다. 둘째, 콘스 패턴과 마찬가지 이유로 정수 패턴은 반드시 괄호로 감싸야 한다. 예컨대, 함수를 정의할 때 괄호 없이 $\text{pred } n + 1 = n$ 과 같이 써 버리면 $(\text{pred } n) + 1 = n$ 와 같은 뜻이 되며, 이는 잘못된 정의다.

4.5 람다식 lambda expression

함수를 등식을 써서 정의하는 대신, 람다식^{lambda expression}을 써서 함수를 만들 수 있다. 람다식은 각각의 인자를 나타내는 패턴과, 인자로 결과를 어떻게 계산하는지 나타내는 몸체로 이루어지지만, 그 만들어진 함수에는 이름을 붙이지 않는다. 그래서, 람다식을 이름없는 함수라고도 한다. 예컨대, 어떤 한 숫자 x 를 인자로 받아 그 결과로 $x + x$ 를 만들어 내는 이름없는 함수를 다음과 같이 만들 수 있다.

$$\lambda x \rightarrow x + x$$

기호 λ 는 “람다”라고 읽는 그리스 소문자다. 람다식으로 만들어진 함수는 비록 이름이 없지만 다른 함수를 쓰듯 똑같이 쓸 수 있다. 예컨대,

$$> (\lambda x \rightarrow x + x) 2 \\ 4$$

람다식은 그 자체로도 흥미롭지만, 실제 프로그램을 짤 때도 여러 모로 쓸모가 있다. 첫째로, 커리된 함수 정의가 뜻하는 바를 람다식으로써 형식적으로 나타낼 수 있다. 예컨대, 다음 정의는

$$add\ x\ y = x + y$$

아래와 같은 뜻으로 이해할 수 있다.

$$add = \lambda x \rightarrow (\lambda y \rightarrow x + y)$$

이는 함수 add 가 숫자 x 를 받아 함수를 돌려주며, 그 함수가 다음에 오는 숫자 y 를 받아 $x + y$ 를 돌려준다는 것을 분명하게 드러낸다.

둘째로, 람다식은 커리된 함수이다 보니 함수를 돌려주는 것이 아니라, 그 함수 자체의 본질상 함수를 돌려주도록 정의하는 것이 더 자연스러운 경우에 쓸모가 있다.

예컨대, 그 결과값이 항상 어떤 주어진 값으로 일정한 함수를 상수 함수^{constant function}라 하는데, 이런 상수 함수를 만들어 내는 라이브러리 함수 *const*를 다음과 같이 정의할 수도 있다.

```
const    :: a → b → a
const x_ = x
```

하지만, 타입에는 괄호를 쓰고 함수 정의에는 람다식을 써서 아래와 같이 *const* 함수가 그 결과로 함수를 만들어 낸다는 사실을 드러나 보이도록 정의하는 쪽이 더 뜻이 와 닿는다.

```
const    :: a → (b → a)
const x = λ_ → x
```

마지막으로, 람다식을 쓰면 딱 한번만 쓰이는 함수를 이름짓지 않아도 된다. 예컨대, 첫 n 개의 홀수를 만들어 내는 *odds* 함수를 다음과 같이 정의할 수 있다.

```
odds    :: Int → [Int]
odds n = map f [0..n - 1]
  where f x = x * 2 + 1
```

(라이브러리 함수 *map*은 주어진 함수를 리스트의 각 인자에 적용한다.) 하지만, 지역 정의 함수 *f*는 딱 한번만 쓰기 때문에, 람다식으로써 함수 *odds*를 다음과 같이 더 간단히 정의할 수 있다.

```
odds n = map (λx → x * 2 + 1) [0..n - 1]
```

따라하기

다음은 *add* 와 *odds* 함수를 각각 보통의 커리된 함수와 람다식을 이용해 정의한 함수로 각각 두 번씩 *lambda.hs*에 정의한 화면이다.

```

lambda.hs (K:WMyDoc) - GVIM
파일(F) 편집(E) 도구(I) 문법(S) 버퍼(B) 창(W) 도움말(H)
add' x y = x + y
add = \x -> (\y -> x + y)

odds' :: Int -> [Int]
odds' n = map f [0 .. n-1]
  where f x = x*2 + 1

odds :: Int -> [Int]
odds n = map (\x -> x*2 + 1) [0 .. n-1]
"lambda.hs" 10L, 198C 저장했습니다          8, 0-1      모두

```

보통의 커리된 함수와 람다식을 이용해 정의한 함수를 비교하기 위해 보통의 커리된 함수 이름에는 따옴표(')를 하나씩 붙여 구분했다.

다음은 *lambda.hs*를 WinHugs에서 불러들여 실행해 본 화면이다.

```

WinHugs
File Edit Actions Browse Help
Type :? for help
Main> :type \x -> x + x
\x -> x + x :: Num a => a -> a
Main> (\x -> x + x) 2
4
Main> :type add'
add' :: Num a => a -> a -> a
Main> :type add
add :: Integer -> Integer -> Integer
Main> add' 2 3
5
Main> add 2 3
5
Main> :type odds'
odds' :: Int -> [Int]
Main> :type odds
odds :: Int -> [Int]
Main> odds' 5
[1,3,5,7,9]
Main> odds 5
[1,3,5,7,9]
Main>

```

두 가지 서로 다른 방법으로 정의했지만 대응되는 함수들 (*add'* 과 *add*, *odds'* 과 *odds*)의 타입과 실행 결과가 같음을 눈여겨 보라.

따라하기 끝

4.6 잘린식 section

$+$ 와 같이 두 개의 인자 사이에 쓰는 함수를 연산자라 부른다. 앞서 살펴본 바와 같이, 보통 함수도 7 ‘*div*’ 2처럼 작은 따옴표로 감싸면 두 개의 인자 사이에 쓸 수 있는 연산자가 된다. 한편, 그 반대로 가능하다. 더 구체적으로 말하자면, 연산자를 괄호로 감싸 커리된 함수로 만들면 $(+)$ 1 2와 같이 보통 함수처럼 두 개의 인자 앞에 쓸 수 있다. 게다가, 이렇게 괄호로 감싸는 표기법으로 필요하다면 $(1+)$ 2나 $(+2)$ 1처럼 두 인자 중 하나를 연산자와 함께 괄호 안에 묶어 넣는 것까지 할 수 있다.

일반적으로 $+$ 가 연산자이고 x 와 y 가 연산자의 인자일 때, $(+)$, $(x+)$, $(+y)$ 와 같은 형태의 식을 잘린식 section이라 부르며, 이런 잘린식이 뜻하는 함수를 람다식으로써 아래와 같이 형식화할 수 있다.

$$\begin{aligned}(+) &= \lambda x \rightarrow (\lambda y \rightarrow x + y) \\ (x+) &= \lambda y \rightarrow x + y \\ (+y) &= \lambda x \rightarrow x + y\end{aligned}$$

잘린식은 크게 세 가지로 쓰인다. 첫째로, 잘린식으로 그것들은 다음의 예처럼 특별히 간편한 방법으로 여러 가지의 단순하지만 유용한 함수들을 구성하는데 사용 할 수 있다.

- (+) 는 덧셈 함수 $\lambda x \rightarrow (\lambda y \rightarrow x + y)$ 를 나타낸다.
- (1+) 는 다음 수를 구하는 함수 $\lambda y \rightarrow 1 + y$ 를 나타낸다.
- (1/) 는 역수 함수 $\lambda y \rightarrow 1 / y$ 를 나타낸다.
- (*2) 는 두배 함수 $\lambda x \rightarrow x * 2$ 를 나타낸다.
- (/2) 는 절반 함수 $\lambda x \rightarrow x / 2$ 를 나타낸다.

둘째로, 연산자의 타입을 나타낼 때 잘린식이 필요하다. 왜냐하면 하스켈에서 연산자 그 자체로는 식을 이루지 못하기 때문이다. 예컨대, 논리곱 연산자 \wedge 의 타입을 다음과 같이 나타낸다.

$$(\wedge) :: Bool \rightarrow Bool \rightarrow Bool$$

4.7. 살펴보기

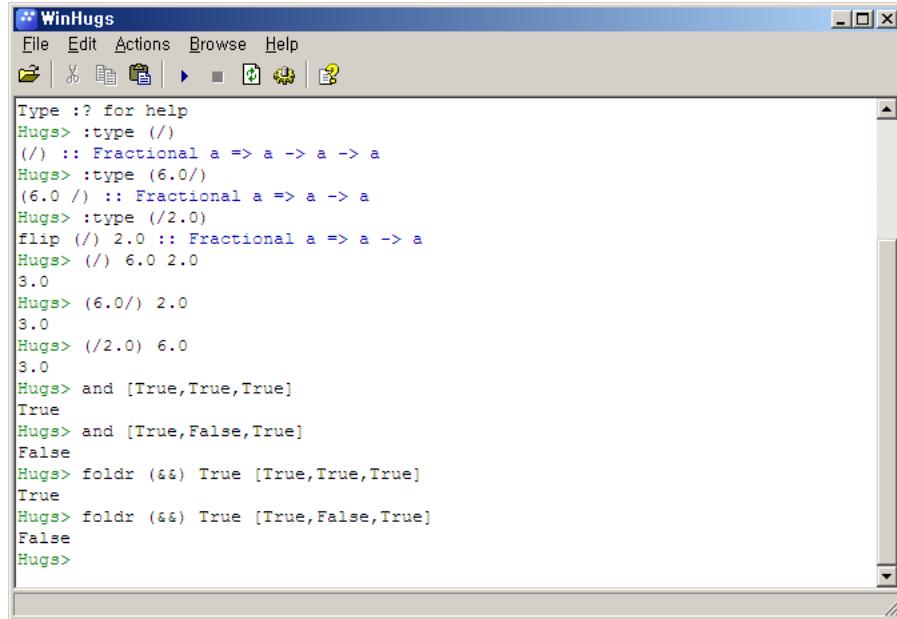
마지막으로, 연산자를 다른 함수의 인자로 넘길 때도 잘린식이 필요하다. 예컨대, 리스트의 모든 원소가 참인지 알아보는 라이브러리 함수 *and* 는, 라이브러리 함수 *foldr* 의 인자로 \wedge 연산자를 써서 다음과 같이 정의한다.

```
and :: [Bool] → Bool
and = foldr (Λ) True
```

foldr 라이브러리 함수에 대해서는 7장에서 다룰 것이다.

따라하기

다음은 잘린식을 실습해 본 화면이다.



The screenshot shows the WinHugs Haskell interpreter interface. The menu bar includes File, Edit, Actions, Browse, and Help. The toolbar has icons for opening files, saving, and running scripts. The main window displays a command-line session:

```
Type :? for help
Hugs> :type (/)
(/) :: Fractional a => a -> a -> a
Hugs> :type (6.0/)
(6.0 /) :: Fractional a => a -> a
Hugs> :type (/2.0)
flip (/) 2.0 :: Fractional a => a -> a
Hugs> (/) 6.0 2.0
3.0
Hugs> (6.0/) 2.0
3.0
Hugs> (/2.0) 6.0
3.0
Hugs> and [True,True,True]
True
Hugs> and [True,False,True]
False
Hugs> foldr (Λ) True [True,True,True]
True
Hugs> foldr (Λ) True [True,False,True]
False
Hugs>
```

and 와 *foldr* 함수와 논리곱 연산자 (\wedge)는 표준 서막에 정의된 함수이므로 따로 다른 스크립트를 불러들이지 않고도 쓸 수 있다.

따라하기 끝

4.7 살펴보기

폐된 매칭을 하스켈 언어의 더 기본적인 기능으로 바꿔서 생각하는, 폐된 매칭의 형식적인 의미를 하스켈 보고서 Haskell Report (25) 에서 찾아볼 수 있다. 그리스 문자 λ 는 이름없는 함수로 수학적인 이론인 람다 션법 lambda calculus 에서 나왔고 이러한 함수들은 하스켈의 근간이 된다.

4.8 연습문제

1. 라이브러리 함수를 써서 원소가 짝수개인 리스트를 똑같은 길이의 두 리스트로 나누는 함수 $halve :: [a] \rightarrow ([a], [a])$ 를 정의하라. 예컨대

```
> halve [1,2,3,4,5,6]
([1,2,3],[4,5,6])
```

2. 라이브러리 함수 $tail$ 와 같이 동작하지만, 빈 리스트에 적용했을 때 잘못이 나는 $tail$ 과는 달리, 빈 리스트에 적용했을 때 그대로 빈 리스트가 나오는 함수 $safetail :: [a] \rightarrow [a]$ 을 다음 세 가지 방법으로 정의해 보라.

- (a) 조건식
- (b) 보초선 등식
- (c) 패턴 매칭

귀띔: 라이브러리 함수 $null$ 을 쓰라.

3. 논리합 연산자 \vee 를 \wedge 처럼 패턴 매칭을 써서 4 가지 서로 다른 방법으로 정의해 보라.

4. 다음과 같은 논리곱 연산자의 정의를 패턴 매칭 대신 조건식을 써서 나타내 보라.

$$\begin{array}{l} True \wedge True = True \\ - \quad \wedge - = False \end{array}$$

5. 다음의 정의에 대해서도 마찬가지로 해 보고, 필요한 조건식의 개수가 어떻게 다른지 알아보라.

$$\begin{array}{l} True \wedge b = b \\ False \wedge - = False \end{array}$$

6. 커리된 함수 $mult\ x\ y\ z = x * y * z$ 를 람다식으로 바꿔 정의해 보고 어떠한 뜻으로 이해할 수 있는지 생각해 보라.

제 5 장

리스트 조건제시식 comprehension

이 장에서는, 리스트 조건제시식 list comprehension 을 소개한다. 조건제시식을 쓰면 리스트를 다루는 많은 함수들을 간단하게 정의할 수 있다. 먼저 생성원 generator 과 보초 guard 를 설명하는 것으로 시작해, zip 함수와 글줄 조건제시식 string comprehension 에 대해 소개하고, 카이사르 암호를 깨는 프로그램을 만드는 것으로 끝맺는다.

5.1 생성원 generator

수학에서는 조건제시법을 써서 기존의 집합으로부터 새로운 집합을 만들 수 있다. 예컨대, 조건제시법 $\{x^2 \mid x \in \{1..5\}\}$ 는 $\{1, 4, 9, 16, 25\}$ 를 만들어 내며, 이는 x 가 $\{1..5\}$ 의 원소일 때 모든 x^2 을 모은 것이다. 하스켈에서도 이와 비슷한 조건제시법을 써서 다음과 같이 기존의 리스트로부터 새로운 리스트를 만들어 낼 수 있다.

```
> [x↑2 | x ← [1..5]]  
[1, 4, 9, 16, 25]
```

기호 $|$ 와 \leftarrow 의 뜻은 각각 “다음 조건을 만족하는”, “다음으로부터 뽑은”이며 $x \leftarrow [1..5]$ 와 같은 표현을 생성원이라고 부른다. 리스트 조건제시식에는 쉼표로 구분된 여러 개의 생성원이 있을 수 있다. 예컨대, 리스트 $[1, 2, 3]$ 의 원소와 $[4, 5]$ 의 원소로 만들어낼 수 있는 모든 순서쌍 리스트는 다음과 같이 구할 수 있다.

$$> [(x, y) \mid x \leftarrow [1, 2, 3], y \leftarrow [4, 5]]$$

$$[(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)]$$

위 보기에서 두 생성원의 순서를 바꿔도 여전히 같은 순서쌍들을 만들어 내긴 하지만, 그 나타나는 순서가 달라진다.

$$> [(x, y) \mid y \leftarrow [4, 5], x \leftarrow [1, 2, 3]]$$

$$[(1, 4), (2, 4), (3, 4), (1, 5), (2, 5), (3, 5)]$$

좀더 구체적으로 말하자면, 이번 경우에는 순서쌍의 x 성분이 y 성분보다 더 자주 바뀌는 반면 $(1, 2, 3, 1, 2, 3)$ 와 $4, 4, 4, 5, 5, 5$, 면접번 경우에는 y 성분이 x 성분보다 더 자주 바뀐다 ($4, 5, 4, 5, 4, 5$ 와 $1, 1, 2, 2, 3, 3$). 이러한 동작은 나중에 오는 생성원일수록 더 깊숙이 포개져 들어가 있어 그 변수에 해당하는 값이 더 자주 바뀌는 것으로 생각함으로써 이해할 수 있다.

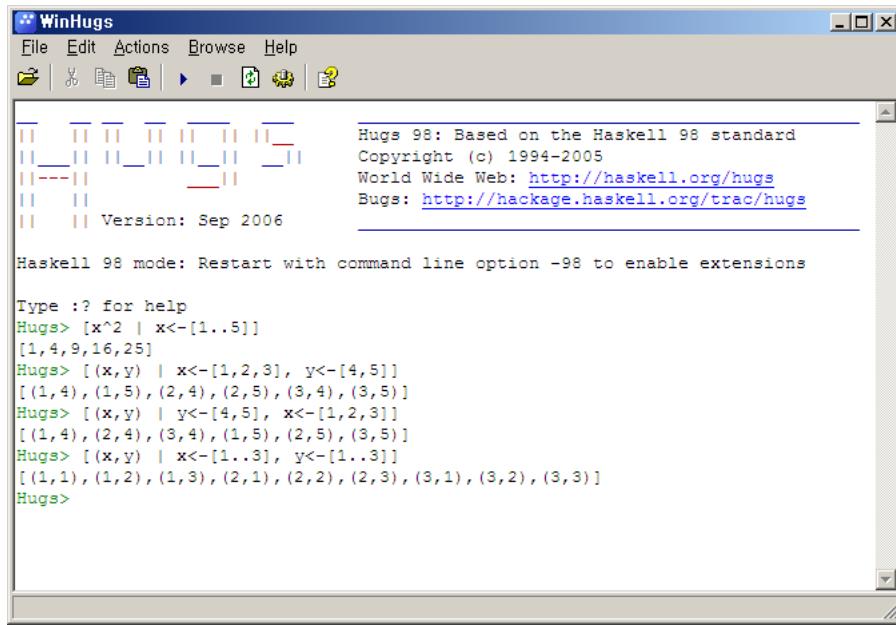
또한, 뒤에 오는 생성원은 그보다 앞선 생성원이 만들어내는 변수의 값에 의존할 수 있다. 예컨대, 순서쌍의 첫째 수가 둘째 수보다 크지 않도록 정렬된 $[1..3]$ 의 원소들로 짹지울 수 있는 가능한 모든 순서쌍의 리스트는 다음과 같이 만들어 낼 수 있다.

$$> [(x, y) \mid x \leftarrow [1..3], y \leftarrow [x..3]]$$

$$[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]$$

따라하기

다음은 리스트 조건제시식을 WinHugs로 실습해 본 화면이다.



The screenshot shows the WinHugs Haskell interpreter window. The title bar says "WinHugs". The menu bar includes "File", "Edit", "Actions", "Browse", and "Help". The toolbar has icons for file operations like Open, Save, and Print. The main window displays Haskell code and its results:

```

Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-2005
World Wide Web: http://haskell.org/hugs
Bugs: http://hackage.haskell.org/trac/hugs

Version: Sep 2006

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs> [x^2 | x<-[1..5]]
[1,4,9,16,25]
Hugs> [(x,y) | x<-[1..3], y<-[4..5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
Hugs> [(x,y) | y<-[4..5], x<-[1..2..3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
Hugs> [(x,y) | x<-[1..3], y<-[1..3]]
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]
Hugs>

```

책에서는 생성원을 쓸 때 특수 기호 \leftarrow 가 나타나지만 텍스트 편집기나 WinHugs에서는 $<-$ 로 나타난다. 다른 특수 기호들은 부록 B를 참조하라.

따라하기 끝 조건제시식의 또 다른 보기로, 리스트의 리스트를 하나의 리스트로 이어붙이는 라이브러리 함수 *concat*을 들 수 있는데, 하나의 생성원은 리스트를 차례로 하나씩 골라내고, 또 다른 생성원은 그 골라 낸 리스트에서 원소를 하나씩 골라내는 방법을 써서 다음과 같이 정의할 수 있다.

```

concat    :: [[a]] → [a]
concat xss = [x | xs ← xss, x ← xs]

```

생성원에 아무거나 다 들어맞는 패턴인 $_$ 을 쓰기도 하는데, 리스트에서 특정 원소를 버리고자 할 때 때때로 쓸모가 있다. 예컨대, 순서쌍의 리스트에서 순서쌍의 첫째 성분만 모두 골라내는 함수를 다음과 같이 정의할 수 있다.

$$\begin{aligned} \text{firsts} &:: [(a, b)] \rightarrow [a] \\ \text{firsts } ps &= [x \mid (x, _) \leftarrow ps] \end{aligned}$$

이와 비슷한 방법으로, 리스트의 길이를 계산하는 라이브러리 함수 *length*도 리스트의 모든 원소를 1로 바꾼 뒤 모든 원소의 합을 구하도록 다음과 같이 정의할 수 있다.

$$\begin{aligned} \text{length} &:: [a] \rightarrow \text{Int} \\ \text{length } xs &= \text{sum} [1 \mid _ \leftarrow xs] \end{aligned}$$

이 경우, 생성원 $_ \leftarrow xs$ 는 단지 리스트의 길이에 맞게 적절한 개수의 1을 생성할 수 있도록 관리하는 역할을 할 뿐이다.

5.2 보초 guard

리스트 조건제시식에 보초라 불리는 논리식을 쓸 수도 있는데, 이를 이용해 앞선 생성원으로부터 만들어진 값들을 걸러낼 수 있다. 보초가 참 *True* 이면 그 값을 그대로 취하고, 거짓 *False* 이면 그 값을 버린다. 예컨대, 조건제시식 $[x \mid x \leftarrow [1..10], \text{even } x]$ 는 리스트 $[1..10]$ 중 짝수만을 고른 리스트 $[2, 4, 6, 8, 10]$ 를 만들어낸다. 비슷한 방법으로, 양의 정수를 그 인수의 리스트에 대응시키는 함수를 다음과 같이 정의할 수 있다.

$$\begin{aligned} \text{factors} &:: \text{Int} \rightarrow [\text{Int}] \\ \text{factors } n &= [x \mid x \leftarrow [1..n], n \text{ `mod' } x == 0] \end{aligned}$$

예컨대, 위 함수를 양의 정수 15와 7에 적용하면 다음과 같은 결과를 얻는다.

```
> factors 15
[1, 3, 5, 15]
```

```
> factors 7
[1, 7]
```

1보다 큰 정수가 그 인수로 1과 자기 자신만을 가질 때 소수임을 기억하라.

따라서, *factors* 함수를 쓰면 주어진 정수가 소수인지 아닌지 검사하는 함수를 다음과 같이 간단히 정의할 수 있다.

```
prime    :: Int → Bool  
prime n = factors n == [1, n]
```

예컨대, 위 함수를 양의 정수 15와 7에 적용하면 다음과 같은 결과를 얻는다.

```
> prime 15  
False
```

```
> prime 7  
True
```

15가 소수가 아니라는 것을 검사하기 위해 모든 인수를 다 만들어낼 필요가 없다는 사실에 주목하라. 느긋한 계산법을 따르기 때문에 1이나 자기 자신이 아닌 인수가 나타나자마자 *False*를 그 결과로 돌려주게 된다. 위 보기에서는 3이 나타나자마자 15가 소수가 아니라는 결과를 돌려준다.

다시 리스트 조건제시식으로 돌아와서, *prime* 함수를 써서 주어진 수 이하의 모든 소수 리스트를 만들어 내는 다음과 같은 함수를 정의할 수 있다.

```
primes    :: Int → [Int]  
primes n = [x | x ← [2..n], prime x]
```

예컨대,

```
> primes 40  
[2, 3, 5, 7, 11, 17, 19, 23, 29, 31, 37]
```

12장에서 이보다 더 효율적으로 소수를 만들어 내는 프로그램을 선보일 것이다. 그 유명한 “에라토스테네스의 체” sieve of Eratosthenes를 하스켈로 만들면 특히 깔끔하고 간결하다.

보초에 대한 마지막 보기로서, 참조표 lookup table 를 키 key 와 값 value 으로 이루어진 순서쌍의 리스트로 표현한다고 생각해 보자. 키의 타입이 같기 타입 equality type 이기만 하다면, 주어진 키와 관련된 모든 값을 리스트로 돌려주는 *find* 함수를 다음과 같이 정의할 수 있다.

$$\begin{aligned} \text{find} &:: \text{Eq } a \Rightarrow a \rightarrow [(a, b)] \rightarrow [b] \\ \text{find } k t &= [v \mid (k', v) \leftarrow t, k == k'] \end{aligned}$$

예컨대,

```
> find 'b' [( 'a', 1), ('b', 2), ('c', 3), ('b', 4)]
[2, 4]
```

따라하기

다음은 보초를 사용하는 리스트 조건제시식으로 정의한 함수들을 *guard.hs*에 정의한 화면이다.

```

guard.hs (K:WMyDoc) - GVIM
파일(F) 편집(E) 도구(I) 문법(S) 버퍼(B) 창(W) 도움말(H)
Factors :: Int -> [Int]
Factors n = [x | x <- [1..n], n `mod` x == 0]

Prime :: Int -> Bool
Prime n = Factors n == [1..n]

Primes :: Int -> [Int]
Primes n = [x | x <- [2..n], prime x]

Find :: Eq a => a -> [(a,b)] -> [b]
Find k t = [v | (k',v) <- t, k == k']

"guard.hs" 11L, 267C 저장 했습니다      5,20      모두

```

다음은 *guard.hs*를 WinHugs에서 불러들여 실행한 화면이다.

```

WinHugs
File Edit Actions Browse Help
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-2005
World Wide Web: http://haskell.org/hugs
Bugs: http://hackage.haskell.org/trac/hugs
Version: Sep 2006

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Main> factors 15
[1,3,5,15]
Main> factors 7
[1,7]
Main> prime 15
False
Main> prime 7
True
Main> primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
Main> find 'b' [('a',1), ('b',2), ('c',3), ('b',4)]
[2,4]
Main>

```

따라하기 끝

5.3 zip 함수

라이브러리 함수 *zip*은 두 개의 리스트로부터 원소를 하나씩 빼내 와서 순서쌍을 만드는데, 이를 리스트 중 하나 혹은 둘 다로부터 더 이상 빼낼 원소가 없을 때까지 계속한다. 예컨대,

```
> zip ['a', 'b', 'c'] [1, 2, 3, 4]
[('a', 1), ('b', 2), ('c', 3)]
```

zip 함수는 리스트 조건제시식으로 프로그래밍할 때 쓸모있는 경우가 많다. 예컨대, 리스트에서 서로 인접하는 원소들로 이루어진 모든 순서쌍을 돌려주는 *pairs* 함수를 다음과 같이 정의할 수 있다.

```
pairs    :: [a] → [(a, a)]
pairs xs = zip xs (tail xs)
```

예컨대,

```
> pairs [1, 2, 3, 4]
[(1, 2), (2, 3), (3, 4)]
```

이제 *pairs* 함수를 써서, 아무 순서 타입 ordered type 의 원소로 이루어진 리스트가 정렬되어 있는지 검사하는 함수를 다음과 같이 모든 인접하는 원소들이 바른 순서로 되어 있는지 검사함으로써 간단히 정의할 수 있다.

```
sorted    :: Ord a ⇒ [a] → Bool
sorted xs = and [x ≤ y | (x, y) ← pairs xs]
```

예컨대,

```
> sorted [1, 2, 3, 4]
True
```

```
> sorted [1, 3, 2, 4]
False
```

prime 함수와 마찬가지로, *sorted* 함수는 모든 인접한 원소들의 순서쌍을 다 만들어 내지 않고도 [1, 3, 2, 4]와 같은 리스트가 정렬되어 있지 않음을 판단할 수 있다. 왜냐하면 바르지 않은 순서로 되어 있는 순서쌍이 발견되자마자 *False* 를 그 결과로 돌려주기 때문이다. 위의 보기에서는 (3, 2) 가 바로 그러한 순서쌍이다.

또한, *zip* 함수를 써서, 어떤 값이 리스트에 나타나는 모든 위치를 리스트로 돌려주는 함수를 각각의 원소를 그 위치와 짹지운 다음 원하는 값에 대한 위치만을 골라 내는 방법으로 다음과 같이 정의할 수 있다.

```
positions    :: Eq a ⇒ a → [a] → [Int]
positions x xs = [i | (x', i) ← zip xs [0 .. n], x == x']
    where n = length xs - 1
```

5.4 글줄 string 조건제시식 comprehension

여태까지는 글줄을 하스켈의 기본 개념으로 다루었다. 하지만 실제로 글줄은 그렇게 기본적인 개념이 아니며, 글자의 리스트로 이루어져 있다. 예컨대, "abc"::String 은 단지 ['a', 'b', 'c']::[Char]를 줄여 쓴 것일 뿐이다. 글줄은 좀 유달라 보이긴 하지만 리스트의 한 종류일 뿐이므로, 리스트를 다루는 여러모양 함수는 어떤 것인든 다음과 같이 글줄과 함께 쓸 수 있다.

```
> "abcde" !! 2  
'c'  
  
> take 3 "abcde"  
"abc"  
  
> length "abcde"  
5  
  
> zip "abc" [1,2,3,4]  
[('a',1),('b',2),('c',3)]
```

같은 이유로, 글줄을 다루는 함수를 정의하는 데 리스트 조건제시식을 쓸 수 있다. 예컨대, 소문자의 개수를 세는 함수 및 특정한 글자가 글줄에 몇 번이나 나타나는지 세는 함수를 다음과 같이 정의할 수 있다.

```
lowers    :: String → Int  
lowers xs = length [x | x ← xs, isLower x]  
count     :: Char → String → Int  
count x xs = length [x' | x' ← xs, x == x']
```

예컨대,

```
> lowers "Haskell"
6

> count 's' "Mississippi"
4
```

5.5 카이사르 암호 Caesar cipher

좀더 큰 보기를 다루는 것으로서 이 장을 마무리하고자 한다. 글줄을 원하지 않는 사람들이 읽지 못하도록 암호화하는 문제를 생각해 보라. 이러한 암호화 방법으로 잘 알려진 것으로는 율리우스 카이사르Julius Caesar의 이름을 딴 카이사르 암호가 있다. 글줄을 암호화하기 위해 카이사르는 단순히 글줄에 있는 각각의 글자를 알파벳 순서로 다음 세 번째 글자로 바꿨으며, 끝 알파벳의 다음 글자는 한바퀴 돌아 첫 알파벳으로 했다. 예컨대,

```
"haskell is fun"
```

위 글줄은 다음과 같은 암호로 바뀐다.

```
"kdvnhoor lv ixq"
```

더 일반적으로는, 카이사르가 사용한 치우침 인자shift factor 3 대신 1에서 25 까지의 수 중 아무 수나 골라 쓸 수 있으며, 이에 따라 25 가지 서로 다른 방식의 암호화가 가능하다. 예컨대, 10을 치우침 인자로 쓰면 원래 글줄은 다음과 같이 암호화된다.

```
"rkcuovv sc pex"
```

앞으로 이 절에서는 카이사르 암호를 구현하는 방법에 대해 살펴보고, 또 글자의 빈도에 대한 정보를 이용하여 카이사르 암호를 쉽게 깰 수 있는 방법에 대해 알아보겠다.

암호화 encoding 와 해독 decoding

간단한 설명을 위해, 줄글에서 소문자만을 암호화하고, 그 외에 대문자나 구두점은 그대로 두겠다. 우선 'a'에서 'z' 사이의 소문자를 그에 대응하는 0에서 25 사이의 정수로 바꾸는 *let2int* 함수의 정의와, 그 반대 역할을 하는 *int2let* 함수의 정의를 살펴보자.

```
let2int :: Char → Int
let2int c = ord c - ord 'a'
int2let :: Int → Char
int2let n = chr (ord 'a' + n)
```

(라이브러리 함수 *ord* :: *Char* → *Int* 와 *chr* :: *Int* → *Char* 는 각각 글자를 정수로 표현된 그 유니코드 값으로, 또 그 반대로 바꿔준다.)

옮긴이 주: *chr* 과 *ord* 는 표준 라이브러리 *Char* 모듈에 있는 함수이므로, 이 함수들을 쓰려면 Hugs에서 :load *Char* 로 불러오거나 (GHC의 대화식 환경인 ghci에서는 load 대신 module 명령어를 써야 한다.) 스크립트 앞부분에서 **import Char** 로 불러들어야 한다.

예컨대,

```
> let2int 'a'
0
> int2let 0
'a'
```

위 두 함수를 써서, 치우침 인자를 소문자에 적용하는 *shift* 함수를 아래와 같이 정의할 수 있다. 이 함수는 글자를 수로 바꾼 뒤 치우침 인자를 더한 값을 26으로 나눈 나머지를 구하고 (26으로 나눈 나머지를 구하는 이유는 맨 끝 알파벳 다음 글자를 한바퀴 돌아 다시 첫 알파벳이 되도록 하기 위해서다.) 그 나머지 정수값을 다시 소문자로 바꾼다.

```
shift :: Int → Char → Char
shift n c | isLower c = int2let ((let2int c + n) `mod` 26)
| otherwise = c
```

이 함수가 양과 음의 치우침 인자를 모두 받을 수 있다는 점과 소문자만을 바꾼다는 점에 주목하라. 예컨대,

```
> shift 3 'a'
'd'
```

```
> shift 3 'z'
'c'
```

```
> shift (-3) 'c'
'z'
```

```
> shift 3 ''
'',
```

글줄 조건제시식에 *shift* 함수를 쓰면, 주어진 치우침 인자에 맞게 줄글을 암호화하는 함수를 다음과 같이 쉽게 정의할 수 있다.

```
encode :: Int → String → String
encode n xs = [shift n x | x ← xs]
```

암호를 해독하는 함수는 따로 필요없다. 왜냐하면, 음의 치우침 인자를 주기만 하면 간단히 해독할 수 있기 때문이다. 예컨대,

```
> encode 3 "haskell is fun"
"kdvnhoorlvixq"
```

```
> encode (-3) "kdvnhoorlvixq"
"haskeill is fun"
```

빈도표 frequency table

카이사르 암호를 깨는 비법은 영어로 된 글에 어떤 글자들을 다른 글자들보다 더 많이 나타난다는 사실을 알아차리는 데 있다. 많은 양의 글을 분석하여 영어 알파벳이 나타나는 빈도를 백분율로 어림잡은 다음 표를 얻을 수 있다.

```
table :: [Float]
table = [8.2, 1.5, 2.8, 4.3, 12.7, 2.2, 2.0, 6.1, 7.0, 0.2, 0.8, 4.0, 2.4,
         6.7, 7.5, 1.9, 0.1, 6.0, 6.3, 9.1, 2.8, 1.0, 2.4, 0.2, 2.0, 0.1]
```

예컨대, 'e' 가 12.7%의 빈도로 가장 많이 나타나고, 'q' 와 'z' 는 0.1%로 가장 드물게 나타난다. (단어별로 빈도표를 만드는 것도 유용할 것이다.) 이에 따라, 먼저 어떤 수의 다른 수에 대한 백분율을 떠돌이 소수점 수로 돌려주는 함수를 다음과 같이 정의하자.

```
percent :: Int → Int → Float
percent n m = (fromIntegral n / fromIntegral m) * 100
```

(라이브러리 함수 `fromIntegral :: Int → Float`는 정수를 그에 해당하는 떠돌이 소수점 수로 바꾼다.) 글줄 조건제시식에 `percent` 함수 및 앞에서 정의한 `lowers` 와 `count` 함수를 써서 아무 글줄에 대한 빈도표를 돌려주는 함수를 다음과 같이 정의할 수 있다.

```
freqs :: String → [Float]
freqs xs = [percent (count x xs) n | x ← ['a' .. 'z']]
           where n = lowers xs
```

예컨대,

```
> freqs "abbcccdddddeeeee"
[6.7, 13.3, 20.0, 26.7, 33.3, 0.0, 0.0, 0.0, ..., 0.0]
```

즉, 글자 'a' 는 6.7%의 빈도, 글자 'b' 가 13.3%의 빈도 등등으로 나타난다. `freqs` 에 지역 정의 `n = lowers xs` 가 있기 때문에, 글줄 조건제시식에서 이 수를 스물여섯 번 쓰더라도 인자로 넘어온 글줄에 있는 소문자의 개수 계산은 딱 한 번만 하게 된다.

암호 깨기

관찰된 빈도 리스트 os 와 기대 빈도 리스트 es 를 비교하는 표준적인 방법으로는, 다음 덧셈식으로 정의되는 카이제곱 통계량 chi-square statistics 이 있다. 식에 나타나는 n 은 두 리스트의 길이를 나타내며 xs_i 는 0부터 세어 i 번째에 있는 xs 리스트의 원소를 나타낸다.

$$\sum_{i=0}^{n-1} \frac{(os_i - es_i)^2}{es_i}$$

카이제곱 통계량에 대한 자세한 내용을 관심있게 다루지는 않겠지만, 중요한 것은 위 식의 값이 작을수록 두 빈도 리스트가 더 가깝게 일치한다는 점이다. 라이브러리 함수 `zip` 과 리스트 조건제시식을 쓰면, 위 수식을 다음과 같은 함수 정의로 쉽게 바꿀 수 있다.

```
chisqr      :: [Float] → [Float] → Float
chisqr os es = sum [(o - e)↑2) / e | (o, e) ← zip os es]
```

그 다음, 리스트의 원소를 전체적으로 n 만큼 원쪽으로 밀어 끝에서는 한바퀴 돌려 첫 위치로 밀어내는 함수를 다음과 같이 정의할 수 있다. 단, n 은 0보다 크거나 같고 리스트의 길이보다 작은 정수라고 가정한다.

```
rotate      :: Int → [a] → [a]
rotate n xs = drop n xs ++ take n xs
```

예컨대,

```
> rotate 3 [1, 2, 3, 4, 5]
[4, 5, 1, 2, 3]
```

이제 암호화할 때 쓰인 치우침 인자가 알려지지 않은 암호문을 보고 그것을 해독하기 위해 치우침 인자를 알아내고 싶다고 치자. 이 때, 암호문의 관찰 빈도표를 한칸씩 밀어 가며 회전시켜 얻은 표들의 기대 빈도표에 대한 카이제곱 chi-square 값이 가장 작은 위치를 찾아 그것을 치우침 인자로 쓰면 암호를 깰 수 있을 것이다.

예컨대, $table' = freqs "kdvnhoo lv ixq"$ 라 할 때

$[chisqr (rotate n table') table | n \leftarrow [0..25]]$

위 식은 다음 값을 돌려주는데

$[1408.8, 640.3, 612.4, 202.6, 1439.8, 4247.2, 651.3, \dots, 626.7]$

이 중 가장 작은 값인 202.6는 리스트에서 세 번째 위치에 나타난다. 따라서, 암호화에 쓰인 치우침 인자는 아마도 3일 것이라고 생각할 수 있다. 이 장에서 앞서 정의한 *positions* 함수를 써서 지금까지 살펴본 과정을 다음과 같이 구현할 수 있다.

```
crack    :: String → String
crack xs = encode (‐factor) xs
  where
    factor = head (positions (minimum chitab) chitab)
    chitab = [chisqr (rotate n table') table | n ← [0..25]]
    table' = freqs xs
```

예컨대,

```
> crack "kdvnhoo lv ixq"
"haskell is fun"

> crack "vscd mywzbzboroxcsyxc kbo ecopev"
"list comprehensions are useful"
```

일반적으로, *crack* 함수를 써서 대부분의 카이사르 암호문을 해독할 수 있다. 하지만 짧은 글줄이나 쓰인 글자의 빈도가 특이한 경우, 해독이 실패할 수도 있음을 알아 두라. 예컨대,

```
> crack (encode 3 "haskell")
"piasmmt"

> crack (encode 3 "boxing wizards jump quickly")
"wjisdib rduvmyn ephk lpdxfgt"
```

따라하기

카이사르 암호를 정의한 *cipher.lhs*를 불러들여 실습한 화면이다.

```

WinHugs
File Edit Actions Browse Help
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-2005
World Wide Web: http://haskell.org/hugs
Bugs: http://hackage.haskell.org/trac/hugs
Version: Sep 2006

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type ?: for help
Main> crack "kdvnhoi lv ixq"
"haskeill is fun"
Main> crack "vscd mywzbzoroxcsyxc kbo ecopev"
"list comprehensions are useful"
Main> crack (encode 3 "haskell")
"piaasmt"
Main> crack (encode 3 "boxing wizards jump quickly")
"wjsdib rduvmyn ephk lpdxfgt"
Main>

```

crack 뿐 아니라 이 앞서 *crack*을 정의하는 데 쓰인 다른 함수들도 *transmit.lhs*를 불러들여 실습할 수 있다.
따라하기 끝

옮긴이 주: 하스켈 스크립트에는 확장자가 .hs 인 파일과 .lhs 인 파일의 두 종류가 있다. 지금까지 이 책에서는 확장자가 .hs 인 하스켈 스크립트만을 살펴보았다. 확장자가 .lhs 인 파일은 문학적 하스켈 스크립트 literate haskell script 라 부른다. 하스켈 스크립트와 문학적 하스켈 스크립트의 차이점은 주석을 다루는 방식에 있다. 하스켈 스크립트는 기본적으로 함수나 데이터 타입을 정의하며 컴퓨터가 알아볼 수 있는 프로그램을 작성하다가, 예외적으로 사람들이 읽을 주석을 작성하려면 한줄 주석이나 포개지는 주석 기호를 써서 따로 표시해야 한다. 반면 문학적 하스켈 스크립트는 마치 문학작품을 쓰듯 기본적으로 사람이 읽을 주석을 작성하다가 필요한 곳에 따로 특별한 표시를 해서 함수나 데이터 타입을 정의한다. 곧이어 한눈에 다시보기를 통해 문학적 하스켈 스크립트인 *transmit.lhs*를 살펴보면 그 차이점을 확실히 느낄 수 있을 것이다. 앞으로 각 장마다 나타나는 흥미로운 보기들은 대부분 문학적 하스켈 스크립트로 작성되어 있으며, *transmit.lhs*를 포함한 이런 문학적 하스켈 스크립트들을 이 책(한글판)의 홈페이지에서 내려받을 수 있다.

스크립트 *cipher.lhs* 한눈에 다시보기

```

1 Programming in Haskell 5.5절 카이사르 암호 보기,
2 Graham Hutton, Cambridge University Press, 2007.
3
4
5 > import Char
6
7 암호화와 앤드
8 -----
9
10 > let2int          :: Char -> Int
11 > let2int c         =  ord c - ord 'a'
12 >
13 > int2let          :: Int -> Char
14 > int2let n         =  chr (ord 'a' + n)
15 >
16 > shift             :: Int -> Char -> Char
17 > shift n c | isLower c = int2let ((let2int c + n) `mod` 26)
18 >           | otherwise   =  c
19 >
20 > encode            :: Int -> String -> String
21 > encode n xs       =  [shift n x | x <- xs]
22
23 빈도 분석
24 -----
25
26 > table             :: [Float]
27 > table              =  [8.2, 1.5, 2.8, 4.3, 12.7, 2.2, 2.0,
28 >                      6.1, 7.0, 0.2, 0.8, 4.0, 2.4, 6.7,
29 >                      7.5, 1.9, 0.1, 6.0, 6.3, 9.1, 2.8,
30 >                      1.0, 2.4, 0.2, 2.0, 0.1]
31 >
32 > lowers             :: String -> Int
33 > lowers xs          =  length [x | x <- xs, isLower x]
34 >
35 > count              :: Char -> String -> Int
36 > count x xs         =  length [x' | x' <- xs, x == x']
37 >
38 > percent            :: Int -> Int -> Float
39 > percent n m        =  (fromIntegral n / fromIntegral m) * 100
40 >
41 > freqs              :: String -> [Float]
42 > freqs xs           =  [percent (count x xs) n | x <- ['a'..'z']]
43 >                      where n = lowers xs
44
45 암호 해독
46 -----
47
48 > chisqr              :: [Float] -> [Float] -> Float
49 > chisqr os es        =  sum [(o - e) ^ 2] / e | (o,e) <- zip os es]
50 >
51 > rotate              :: Int -> [a] -> [a]
52 > rotate n xs          =  drop n xs ++ take n xs
53 >
54 > positions           :: Eq a => a -> [a] -> [Int]
55 > positions x xs      =  [i | (x',i) <- zip xs [0..], x == x']
56 >
57 > crack               :: String -> String

```

```
58 > crack xs = encode (-factor) xs
59 >
60 >     where
61 >         factor = head (positions (minimum chitab) chitab)
62 >         chitab = [chisqr (rotate n table') table | n <- [0..25]]
63 >         table' = freqs xs
```

책에서 앞서 설명하지 않은 것들 중 *cipher.lhs* 에서 눈여겨 볼 점들은 다음과 같다.

1. 줄번호 1-4, 줄번호 6-9 등과 같이 아무 표시도 없는 줄은 주석이다.
2. 줄번호 5, 줄번호 10-21 등과 같이 >로 시작하는 줄은 프로그램이다. > 다음에 한 칸을 비우고 함수를 정의한다는 것을 눈여겨 보라.
3. 줄번호 5에서 **import** *Char* 로 모듈을 불러들였기에 줄번호 11과 줄번호 14에서 *Char* 모듈의 함수 *ord* 와 *chr* 를 쓸 수 있다.

5.6 살펴보기

조건제시법 comprehension 이라는 용어는 집합론의 “내포 공리” axiom of comprehension 에서 온 낱말인데, 내포 공리란 어떤 조건을 만족하는 값을 모두 골라 집합을 만들 수 있다는 생각을 염밀하게 나타낸 것이다. 리스트 조건 제시식을 더 기본적인 언어의 기능으로 바꾸어 나타낸 염밀한 형식적 의미가 하스켈 보고서(25)에 나와 있다. 카이사르 암호를 비롯한 여러 암호화 기법을 다루는 인기있는 책으로는 코드북(29)이 있다.

5.7 연습문제

1. 1부터 100까지 제곱의 합 $1^2 + 2^2 + \dots + 100^2$ 을 구하는 식을 리스트 조건제시식을 써서 만들어 보라.
2. *length* 함수와 비슷한 방법으로 똑같은 원소의 리스트를 만들어 내는 라이브러리 함수 *replicate* :: $\text{Int} \rightarrow a \rightarrow [a]$ 를 조건제시식을 이용해 정의해 보라. 예컨대,

```
> replicate 3 True
[True, True, True]
```

3. 양의 정수로 이루어진 순서있는 3 짝 (x, y, z) 가 $x^2 + y^2 = z^2$ 를 만족할 때 피타고라스 3 짝이라 부른다. 리스트 조건제시식을 써서, 각각의 성분이 주어진 수 이하인 모든 피타고라스 3 짝의 리스트를 만들어 내는 *pyths* :: $\text{Int} \rightarrow [(Int, Int, Int)]$ 함수를 정의하라. 예컨대,

```
> pyths 10
[(3, 4, 5), (4, 3, 5), (6, 8, 10), (8, 6, 10)]
```

4. 양의 정수가 그 자신을 제외한 인수들의 합과 같은 수를 완전수라 한다. 리스트 조건제시식과 *factors* 함수를 써서, 주어진 수 이하의 모든 완전수를 구하는 함수 *perfects* :: $\text{Int} \rightarrow [\text{Int}]$ 를 정의하라. 예컨대,

```
> perfects 500
[6, 28, 496]
```

5. 두 개의 생성원으로 이루어진 하나의 조건제시식 $[(x, y) \mid x \leftarrow [1, 2, 3], y \leftarrow [4, 5, 6]]$ 를 각각 하나의 생성원으로 이루어진 두 개의 조건제시식으로 바꿔 써 보라

귀띔: 라이브러리 함수 *concat* 을 이용하고 하나의 조건제시식을 다른 조건제시식에 포개어 쓰라.

6. *positions* 함수를 *find* 함수를 써서 다시 정의해 보라.

7. 길이 n 인 정수 리스트 xs 와 ys 의 스칼라곱은 두 리스트에서 같은 위치에 있는 원소들의 곱을 모두 더한 것으로 정의한다.

$$\sum_{i=0}^{n-1} (xs_i * ys_i)$$

chisqr 함수와 비슷한 방법으로, 리스트 조건제시식을 써서 두 리스트의 스칼라곱을 구하는 함수 *scalarproduct*:: $[Int] \rightarrow [Int] \rightarrow Int$ 를 정의하라. 예컨대,

```
> scalarproduct [1,2,3] [4,5,6]  
32
```

8. 카이사르 암호 프로그램이 대문자도 다룰 수 있도록 고쳐 보라.

제 6 장

되도는 함수 recursive function

이 장에서는 되돌기 recursion 가 무엇인지 알아본다. 되돌기는 하스켈에서 계산을 되풀이하는 기본적인 방법이다. 먼저 정수를 되돌며 계산하는 함수를 살펴보고, 마찬가지로 이러한 방식을 리스트를 되돌며 계산하는 경우로도 넓혀 나갈 수 있음을 보인다. 또한 여러 개의 함수 인자가 있을 때, 그리고 함수들이 서로 부르며 되도는 경우를 살펴보고, 마지막으로 되도는 함수를 어떻게 정의해 나가는 것이 좋은지 알려주는 도움말로 이 장을 맺는다.

옮긴이 주: ‘되돌기’를 ‘재귀(再歸)’라는 말로 쓰기도 했다. 이 책에서 한때 많이 쓰던 ‘재귀(再歸)’라는 낱말을 두고 ‘되돌기’로 옮기는 이유는 다음과 같다.

첫째, ‘되돌기’가 ‘재귀(再歸)’보다 쉬운 말이기 때문이다. 국어사전과 옥편을 찾아 보면 여기에는 아무도 의견이 없을 것이다.

둘째, 우리나라에서 함수형 언어를 제대로 다루는 최초의 번역서이자 컴퓨터 과학 분야에서 고전 중의 고전으로 널리 알려진 “컴퓨터 프로그램의 구조와 해석”(Structure and Interpretation of Computer Programs)에서도 recursion 을 ‘되돌기’로 옮기고 있으며, 그 책이 현재 서울대학교 컴퓨터 공학과 및 KAIST 전산학과의 강의 교재로 쓰이고 있다. 따라서 이러한 국내 함수형 프로그램 언어 관련 서적 및 관련 교과목에서 쓰는 용어와 일관성을 유지하기 위해서도 ‘되돌기’로 옮기는 것이 바람직하다고 생각한다.

셋째, 함수형 언어에서는 되도는 방법이 정말 자기 자신을 이용해 정의하는 것 말고는 없기 때문이다. 명령형 언어에서는 반복문 loop 을 기본으로 사용하며, 자기 자신을 이용한 함수 정의를 상당히 예외적인 것으로 생각하기 때문에 ‘재귀(再歸)’라는 어려운 말을 쓸 일밀의 이유가 있을 수 있다. 하지만 함수형 언어에서는 말 그대로 되도는 방법은 되도는 함수를 이용하는 것밖에 없으므로 개념상으로도 딱 떨어지는 말인 것이다.

6.1 기본 개념

앞선 장에서 살펴 본 것처럼, 많은 함수들은 이미 주어진 다른 함수를 써서 자연스레 정의할 수 있다. 예컨대, 음이 아닌 정수의 사다리곱 factorial 을 구하는 함수를 다음과 같이 1부터 n 까지의 정수의 곱을 계산하는 라이브러리 함수 product 를 써서 정의할 수 있다.

```
factorial :: Int → Int
factorial n = product [1 .. n]
```

하스켈에서는 함수를 함수 자신을 써서 정의할 수 있으며, 이러한 함수를 ‘되도는’ 함수라 부른다. 예컨대, 이러한 방식으로 사다리곱을 구하는 factorial 함수를 아래와 같이 정의할 수 있다.

```
factorial 0      = 1
factorial (n + 1) = (n + 1) * factorial n
```

첫째 등식은 0의 사다리곱 factorial 이 1임을 나타내며, 이를 밑바탕(이 되는) 경우 base case 라 부른다. 둘째 등식은 양의 정수의 사다리곱은 그 양의 정수와 그 바로 앞 수의 사다리곱의 곱임을 나타내며, 이를 되도는 경우 recursive case 이라 부른다. 예컨대, 위 정의에 따라 3의 사다리곱을 다음과 같이 계산할 수 있다.

```
factorial 3
=      { factorial 을 적용 }
3 * factorial 2
=      { factorial 을 적용 }
3 * (2 * factorial 1)
=      { factorial 을 적용 }
3 * (2 * (1 * factorial 0))
=      { factorial 을 적용 }
3 * (2 * (1 * 1))
=      { * 를 적용 }
```

6

factorial 함수를 그 자신을 써서 정의하기는 했지만, 이 함수가 계산을 끝없이 되풀이^{loop} 하지 않는다는 것에 주목하라. 좀더 자세히 설명하면, *factorial* 함수를 한 번 부를 때마다 정수 인자 값이 0이 되기 전까지 하나씩 줄어들어, 결국 0이 되면 되돌기를 끝내고 곱셈을 하게 된다. 0의 사다리곱이 1인 이유는 1이 곱셈의 항등원이기 때문이다. 다시 말해 임의의 정수 x 에 대해서 $1 * x = x$ 과 $x * 1 = x$ 이 항상 성립한다.

factorial 함수의 경우는 처음에 라이브러리 함수를 써서 정의했던 것이 되돌기로 정의하는 것보다 더 간단하다. 하지만 이 책에서 앞으로 살펴볼 여러 함수들은 되돌기로 정의하는 것이 간단하고 자연스럽다. 예컨대, 하스켈의 많은 라이브러리 함수들이 이런 식으로 정의되어 있다. 또한, 13장에서 살펴보겠지만, 되돌기로 정의한 함수들은 귀납법이라는 강력한 기법을 써서 그 함수들의 성질을 증명할 수 있다.

정수를 계산하며 되도는 또 하나의 보기로 위에서 썼던 곱셈 연산자 $*$ 를 살펴보자. 효율상의 이유로 이 연산자는 하스켈에서 기본 함수로 제공된다. 하지만, 음이 아닌 정수에 대한 곱셈은 다음과 같이 두 인자 중 하나에 대한 되돌기로 정의할 수도 있다.

$$\begin{array}{ll} (*) & :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\ m * 0 & = 0 \\ m * (n + 1) & = m + (m * n) \end{array}$$

예컨대,

$$\begin{aligned} & 4 * 3 \\ = & \{ * \text{를 적용 } \} \\ = & 4 + (4 * 2) \\ = & \{ * \text{를 적용 } \} \\ = & 4 + (4 + (4 * 1)) \\ = & \{ * \text{를 적용 } \} \\ = & 4 + (4 + (4 + (4 * 0))) \\ = & \{ * \text{를 적용 } \} \\ = & 4 + (4 + (4 + 0)) \\ = & \{ + \text{를 적용 } \} \\ & 12 \end{aligned}$$

* 연산자의 되도는 정의는 바로 곱셈이란 덧셈의 반복으로 바꿔 생각할 수 있다는 것을 형식화된 등식으로 나타낸 것이다.

6.2 리스트로 되돌기

정수를 다루는 함수만 되돌기로 정의할 수 있는 것이 아니라, 리스트를 다루는 함수 또한 되돌기로 정의할 수 있다. 예컨대, 바로 앞 절에서 쓰인 라이브러리 함수 *product*를 다음과 같이 정의할 수 있다.

$$\begin{aligned} \text{product} & :: \text{Num } a \Rightarrow [a] \rightarrow a \\ \text{product } [] & = 1 \\ \text{product } (n : ns) & = n * \text{product } ns \end{aligned}$$

첫째 등식은 빈 리스트 곱이 1임을 나타내며, 이는 1이 곱셈의 항등원이므로 적절한 정의이다. 둘째 등식은 비어 있지 않은 리스트의 곱이란, 첫째 수와 그 나머지 수들로 이루어진 리스트의 곱한 결과를 곱한 것임을 나타낸다. 예컨대,

```

product [2,3,4]
=   { product 를 적용 }
=   2 * product [3,4]
=   { product 를 적용 }
=   2 * (3 * product [4])
=   { product 를 적용 }
=   2 * (3 * (4 * product []))
=   { product 를 적용 }
=   2 * (3 * (4 * 1))
=   { * 를 적용 }
24

```

하스켈에서 리스트는 사실 콘스^{cons} 연산자를 써서 한번에 하나의 원소씩 붙여 나감으로써 만들어진다는 것을 기억하라. 따라서, $[2,3,4]$ 는 단지 $2 : (3 : (4 : []))$ 를 줄여 쓴 것일 뿐이다. 리스트를 다루며 되도는 또 다른 간단한 보기로서, 라이브러리 함수 *length*를 *product*와 같은 방식으로 정의할 수 있다.

```

length      :: [a] → Int
length []    = 0
length (_ : xs) = 1 + length xs

```

이는 빈 리스트의 길이는 0이며, 비어 있지 않은 리스트의 길이는 첫 원소를 제외한 나머지 리스트의 길이보다 하나 더 큰 수임을 나타낸다. 되도는 경우에 쓰인 아무거나 패턴 $_$ 에 주목하라. 이는 리스트의 길이가 리스트 원소들의 값에 의존하지 않음을 드러낸다.

이번에는 리스트의 순서를 거꾸로 뒤집는 라이브러리 함수 *reverse*를 살펴보자. 이 함수는 다음과 같이 되돌기로 정의할 수 있다.

```

reverse      :: [a] → [a]
reverse []    = []
reverse (x : xs) = reverse xs ++ [x]

```

이는 빈 리스트를 뒤집으면 그냥 그대로 빈 리스트이며, 비어 있지 않은 리스트를 뒤집으려면 첫 원소를 제외한 나머지 리스트를 뒤집고 그 끝에 첫 원소로 이루어진 한원소 리스트를 이어붙이면 된다는 것을 나타낸다. 예컨대,

$$\begin{aligned}
 & \text{reverse } [1, 2, 3] \\
 = & \quad \{ \text{ reverse 를 적용 } \} \\
 & \text{reverse } [2, 3] ++ [1] \\
 = & \quad \{ \text{ reverse 를 적용 } \} \\
 & (\text{reverse } [3] ++ [2]) ++ [1] \\
 = & \quad \{ \text{ reverse 를 적용 } \} \\
 & ((\text{reverse } []) ++ [3]) ++ [2]) ++ [1] \\
 = & \quad \{ \text{ reverse 를 적용 } \} \\
 & ((([] ++ [3]) ++ [2]) ++ [1]) \\
 = & \quad \{ ++ 를 적용 \} \\
 & [3, 2, 1]
 \end{aligned}$$

이어서, 위 *reverse*의 정의에 쓰인 이어붙이기 연산자 $++$ 또한 첫번째 인자에 대해 되도록 함수로 다음과 같이 정의할 수 있다.

$$\begin{aligned}
 (++) & :: [a] \rightarrow [a] \rightarrow [a] \\
 [] & ++ ys = ys \\
 (x : xs) & ++ ys = x : (xs ++ ys)
 \end{aligned}$$

예컨대,

$$\begin{aligned}
 & [1, 2, 3] ++ [4, 5] \\
 = & \quad \{ ++ 를 적용 \} \\
 & 1 : ([2, 3] ++ [4, 5]) \\
 = & \quad \{ ++ 를 적용 \} \\
 & 1 : (2 : ([3] ++ [4, 5])) \\
 = & \quad \{ ++ 를 적용 \} \\
 & 1 : (2 : (3 : ([] ++ [4, 5]))) \\
 = & \quad \{ ++ 를 적용 \} \\
 & 1 : (2 : (3 : [4, 5])) \\
 = & \quad \{ \text{리스트 표기} \} \\
 & [1, 2, 3, 4, 5]
 \end{aligned}$$

#+ 연산자의 되도는 정의는 바로 두 리스트를 이어붙이려면 첫째 리스트의 원소를 남김없이 다 복사한 후 둘째 리스트를 끝에다 붙이면 된다는 생각을 형식화된 등식으로 나타낸 것이다.

정렬된 리스트에 대해 되도는 두 가지 보기로 살펴보는 것으로 이 절을 맺겠다. 첫째 보기로, 임의의 정렬된 리스트에 새로운 원소를 하나 끼워넣어 또 다른 정렬된 리스트를 만드는 함수를 다음과 같이 정의할 수 있다.

$$\begin{aligned} \text{insert} &:: \text{Ord } a \Rightarrow a \rightarrow [a] \rightarrow [a] \\ \text{insert } x [] &= [x] \\ \text{insert } x (y : ys) \mid x \leq y &= x : y : ys \\ &\quad | \text{ otherwise } = y : \text{insert } x ys \end{aligned}$$

즉, 빈 리스트에 새 원소를 끼워넣으면 한원소 리스트가 만들어지고, 비어 있지 않은 리스트에 끼워넣는 경우는 새로운 원소 x 와 리스트의 첫 원소 y 의 크기 순서에 따라 달라진다. 더 자세히는, $x \leq y$ 이면 새로운 원소 x 를 원래 리스트의 앞에 붙여넣기만 하면 되고, 그렇지 않으면^{otherwise} 원래 리스트의 첫 원소 y 를 그대로 새로운 리스트의 첫 원소로 두고 나머지 리스트에 x 를 끼워넣기를 계속한다. 예컨대,

$$\begin{aligned} &\text{insert } 3 [1, 2, 4, 5] \\ &= \{ \text{insert 를 적용 } \} \\ &= 1 : \text{insert } 3 [2, 4, 5] \\ &= \{ \text{insert 를 적용 } \} \\ &= 1 : 2 : \text{insert } 3 [4, 5] \\ &= \{ \text{insert 를 적용 } \} \\ &= 1 : 2 : 3 : [4, 5] \\ &= \{ \text{리스트 표기 } \} \\ &= [1, 2, 3, 4, 5] \end{aligned}$$

insert 함수를 써서 삽입정렬 insertion sort 을 하는 함수를 정의할 수 있다. 삽입정렬이란 다음과 같이 빈 리스트의 경우는 이미 정렬되어 있고,

6.3. 인자가 여럿일 때

비어 있지 않은 리스트의 경우는 리스트의 첫 원소를 그 나머지 리스트를 삽입정렬하여 결과로 얻은 리스트에 끼워넣는다.

$$\begin{aligned}isort &:: \text{Ord } a \Rightarrow [a] \rightarrow [a] \\isort [] &= [] \\isort (x : xs) &= insert x (isort xs)\end{aligned}$$

예컨대,

$$\begin{aligned}&isort [3, 1, 2, 4] \\&= \{ \text{isort 를 적용 } \} \\&\quad insert 3 (insert 2 (insert 1 (insert 4 []))) \\&= \{ \text{insert 를 적용 } \} \\&\quad insert 3 (insert 2 (insert 1 [4])) \\&= \{ \text{insert 를 적용 } \} \\&\quad insert 3 (insert 2 [1, 4]) \\&= \{ \text{insert 를 적용 } \} \\&\quad insert 3 [1, 2, 4] \\&= \{ \text{insert 를 적용 } \} \\&\quad [1, 2, 3, 4]\end{aligned}$$

6.3 인자가 여럿일 때

인자가 여럿인 함수도 한꺼번에 하나 이상의 인자에 대해 되돌도록 정의할 수 있다. 예컨대, 두 리스트를 받아 하나의 순서쌍 리스트를 만들어 내는 라이브러리 함수 `zip` 을 다음과 같이 정의할 수 있다.

$$\begin{aligned}zip &:: [a] \rightarrow [b] \rightarrow [(a, b)] \\zip [] &- = [] \\zip _ &[] = [] \\zip (x : xs) (y : ys) &= (x, y) : zip xs ys\end{aligned}$$

예컨대,

$$\begin{aligned}
 & \text{zip } ['a', 'b', 'c'] [1, 2, 3, 4] \\
 = & \quad \{ \text{zip 을 적용 } \} \\
 = & \quad ('a', 1) : \text{zip } ['b', 'c'] [2, 3, 4] \\
 = & \quad \{ \text{zip 을 적용 } \} \\
 = & \quad ('a', 1) : ('b', 2) : \text{zip } ['c'] [3, 4] \\
 = & \quad \{ \text{zip 을 적용 } \} \\
 = & \quad ('a', 1) : ('b', 2) : ('c', 3) : \text{zip } [] [4] \\
 = & \quad \{ \text{zip 을 적용 } \} \\
 = & \quad ('a', 1) : ('b', 2) : ('c', 3) : [] \\
 = & \quad \{ \text{리스트 끝기 } \} \\
 & [('a', 1), ('b', 2), ('c', 3)]
 \end{aligned}$$

인자로 오는 두 리스트 중 어느 쪽이 비어있을지 모르므로, `zip` 함수의 정의에 두 개의 밑바탕 경우가 필요함에 주목하라. 여러 인자로 되도는 또 다른 보기로서, 리스트의 처음부터 주어진 개수 만큼의 원소를 버리는 라이브러리 함수 `drop`을 다음과 같이 정의할 수 있다.

$$\begin{array}{lll}
 \text{drop} & :: \text{Int} \rightarrow [\text{a}] \rightarrow [\text{a}] \\
 \text{drop } 0 & xs & = xs \\
 \text{drop } (n + 1) [] & & = [] \\
 \text{drop } (n + 1) (_ : xs) & & = \text{drop } n xs
 \end{array}$$

다시금, 두 개의 밑바탕 경우가 필요하다. 하나는 0 개의 원소를 버리려 하는 경우고, 다른 하나는 하나 이상의 원소를 빼 리스트에서 버리려 하는 경우이다.

6.4 여러 갈래로 되돌기

여러 갈래로 되돌도록 함수를 정의할 수도 있는데, 이는 함수 정의에서 한 번 이상 자기 자신을 적용하는 것을 뜻한다. 예컨대, 피보나치 수열을 $0, 1, 1, 2, 3, 5, 8, 13, \dots$ 을 생각해 보라. 첫째와 둘째 수가 0과 1이며, 그 다음에 나타나는 각 수는 그 앞의 두 수를 더하여 얻을 수 있다. 하스켈에서, $n \geq 0$ 인 임의의 정수가 주어졌을 때 n 번째 피보나치 수를 구하는 함수를 되돌기를 두 번 써서 다음과 같이 정의할 수 있다.

```
fibonacci      :: Int → Int
fibonacci 0    = 0
fibonacci 1    = 1
fibonacci (n + 2) = fibonacci n + fibonacci (n + 1)
```

또 다른 보기로는, 퀵정렬(quicksort)이라 불리는 리스트를 정렬하는 잘 알려진 방법이 있는데, 1 장에서 다음과 같이 구현할 수 있다는 것을 살펴본 바 있다.

```
qsort          :: Ord a ⇒ [a] → [a]
qsort []        = []
qsort (x : xs) = qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [a | a ← xs, a ≤ x]
    larger  = [b | b ← xs, b > x]
```

즉, 빈 리스트의 경우는 이미 정렬되어 있고, 비어 있지 않은 리스트의 경우는 그 첫 원소를 나머지 리스트 중에서 그보다 작거나 같은 원소들만 모아 정렬한 리스트와 그보다 큰 원소들만 모아 정렬한 리스트 사이에 놓음으로써 정렬할 수 있다.

6.5 서로 부르며 되돌기 mutual recursion

함수를 서로 부르며 되돌도록 정의할 수도 있는데, 이는 두 개 이상의 함수들을 모두 각각 다른 함수들을 써서 정하는 것을 뜻한다. 예컨대 라이브러리 함수 *even* 과 *odd*의 경우를 살펴보자. 효율상의 이유로, 이 함수들은 2로 나눈 나머지를 이용해 정의하는 것이 보통이다. 하지만, 음이 아닌 정수에 대해서 이 함수들을 서로 부르며 되돌기로 정의할 수도 있다.

```
even      :: Int → Bool
even 0    = True
even (n + 1) = odd n

odd      :: Int → Bool
odd 0    = False
odd (n + 1) = even n
```

즉, 0은 짝수이지만 홀수는 아니며, 양의 정수의 경우에는 그 바로 앞 수가 짝수이면 홀수이고 그 바로 앞 수가 홀수이면 짝수이다. 예컨대,

```
even 4
=   { even 을 적용 }
odd 3
=   { odd 를 적용 }
even 2
=   { even 을 적용 }
odd 1
=   { odd 를 적용 }
even 0
=   { even 을 적용 }
True
```

마찬가지로, 리스트에서 (0부터 세어서) 홀수 번째 원소나 짝수 번째 원소만 골라내는 함수를 다음과 같이 정의할 수 있다.

```


$$\begin{aligned} \textit{evens} &:: [a] \rightarrow [a] \\ \textit{evens} [] &= [] \\ \textit{evens} (x : xs) &= x : \textit{odds} xs \\ \textit{odds} &:: [a] \rightarrow [a] \\ \textit{odds} [] &= [] \\ \textit{odds} (_ : xs) &= \textit{evens} xs \end{aligned}$$

```

예컨대,

```


$$\begin{aligned} &\textit{evens} "abcde" \\ &= \{ \textit{evens} 를 적용 \} \\ & 'a' : \textit{odds} "bcde" \\ &= \{ \textit{odds} 를 적용 \} \\ & 'a' : \textit{evens} "cde" \\ &= \{ \textit{evens} 를 적용 \} \\ & 'a' : 'c' : \textit{odds} "de" \\ &= \{ \textit{odds} 를 적용 \} \\ & 'a' : 'c' : \textit{evens} "e" \\ &= \{ \textit{evens} 를 적용 \} \\ & 'a' : 'c' : 'e' : \textit{odds} [] \\ &= \{ \textit{odds} 를 적용 \} \\ & 'a' : 'c' : 'e' : [] \\ &= \{ \text{리스트 표기} \} \\ & "ace" \end{aligned}$$

```

하스켈에서, 사실 문자열은 문자들의 리스트로 이루어짐을 기억하라. 따라서, "abcde" 는 ['a', 'b', 'c', 'd', 'e'] 의 줄인 표현일 뿐이다.

6.6 되도는 함수 정의를 위한 도움말

되도는 함수를 정의하는 것은 마치 자전거를 타는 것과 같다. 다른 사람이 하는 것을 보면 쉬워 보이지만, 처음으로 직접 해 보면 절대로 할 수 없을 정도로 어렵다는 생각이 들고, 연습을 통해 간단하고 자연스럽게 할 수 있게 된다. 이 절에서는 일반적으로 함수를 정의할 때와 되도는 함수를 정의할 때 특히 도움이 될 만한, 함수를 정의하는 다섯 걸음을 몇 가지 보기로 살펴보기로 하겠다.

보기 - product

간단한 첫 보기로서, 이 장에서 앞서 정의를 소개한 라이브러리 함수인 *product*를 살펴본다. 그럼 리스트에 있는 수들의 곱을 구하는 *product*를 어떻게 만들어 나가는지 그 과정을 한 걸음씩 살펴보자.

첫째 걸음: 타입을 정의하라

함수를 정의할 때 타입에 대해 생각하는 것은 대단히 많은 도움이 된다. 따라서 함수를 정의하기 전에 함수의 타입을 정의하는 것이 좋은 습관이다. 이 경우, 다음과 같은 타입으로 시작해 보자.

product :: $[Int] \rightarrow Int$

이는 *product* 가 정수 리스트를 받아 하나의 정수를 낸다는 뜻이다. 위와 같이, 간단한 타입으로 시작해서 나중에 혹 적절한 타입이 있다면 다시 정의하거나 일반화하는 것이 좋을 때가 많다.

둘째 걸음: 모든 경우를 나누어 늘어놓으라

대부분의 인자 타입이 그러하듯이, 몇 가지 표준적인 경우를 고려해야 한다. 리스트는 표준적인 경우가 빈 리스트와 비어 있지 않은 리스트이다. 따라서, 패턴 매칭을 이용해 다음과 같이 함수의 뼈대를 놓을 수 있다.

```
product []      =
product (n : ns) =
```

음이 아닌 정수는 표준적인 경우를 0과 $n + 1$ 로, 논리값은 *False*와 *True*로, 이와 같이 나눈다. 타입과 마찬가지로, 경우를 나누는 것도 나중에 고쳐 나둬야 할지도 모르지만, 표준적인 경우부터 나누어 놓고 시작하는 것이 좋다.

셋째 걸음: 간단한 경우부터 정의하라

정의에 따라, 0개의 정수의 곱은 1이다. 이는 1이 곱셈의 항등원이기 때문이다. 따라서 빈 리스트의 경우는 다음과 같이 간단히 정의할 수 있다.

$$\begin{aligned} \text{product} [] &= 1 \\ \text{product} (n : ns) &= \end{aligned}$$

위와 같이 간단한 경우는 쉽게 정의할 수 있을 때가 많다.

넷째 걸음: 나머지 경우를 정의하라

비어 있지 않은 리스트의 곱은 어떻게 계산해야 할까? 이번 걸음에서는 함수 자신 (*product*), 함수 인자 (*n* and *ns*), 또 관련된 타입의 라이브러리 함수 (+, -, *) 등) 등 쓸 수 있는 재료가 무엇인지 먼저 생각해 보는 것이 좋다. 이 경우는 첫째 정수와 나머지 정수 리스트의 곱을 간단히 곱하기만 하면 된다.

$$\begin{aligned} \text{product} [] &= 1 \\ \text{product} (n : ns) &= n * \text{product} ns \end{aligned}$$

다섯째 걸음: 일반화하고 간단히 하라

일단 위의 과정을 거쳐 함수를 정의하고 나면, 그 함수를 다음과 같이 정수 뿐 아니라 아무 수치 타입에 대한 함수로 일반화할 수 있다는 것이 분명해진다.

$$\text{product} :: \text{Num } a \Rightarrow [a] \rightarrow a$$

간단히 하는 것에 있어서는, 7장에서 앞으로 살펴보겠지만 *product* 정의에 쓰인 되돌기 유형은 라이브러리 함수 *foldr*로 요약할 수 있으므로,

이를 이용해 다음과 같이 *product* 를 하나의 등식으로 다시 정의할 수 있다.

$$\text{product} = \text{foldr } (*) \ 1$$

따라서, 최종적인 *product* 의 정의는 다음과 같다.

$$\begin{aligned}\text{product} &:: \text{Num } a \Rightarrow [a] \rightarrow a \\ \text{product} &= \text{foldr } (*) \ 1\end{aligned}$$

위 정의는, 효율상의 이유로 *foldr* 대신 관련된 다른 라이브러리 함수인 *foldl* 를 쓴다는 점을 빼면, 부록 A 의 표준 서막에 있는 정의와 똑같다. *foldl* 에 대해서는 7 장에서 다룬다.

보기 - drop

더 내용있는 보기로서, 앞서 정의한 라이브러리 함수 *drop* 을 다섯 걸음으로 어떻게 정의하는지 보이고자 한다. *drop* 은 어떤 리스트에서 주어진 개수만큼의 원소를 버리는 함수이다.

첫째 걸음: 타입을 정의하라

우선, 정수와 임의의 타입 *a* 로 이루어진 리스트를 받아 같은 타입의 리스트를 내는 다음과 같은 타입으로 시작하자.

$$\text{drop} :: \text{Int} \rightarrow [a] \rightarrow [a]$$

위 타입을 정의하기 위해 내린 결정이 네 가지가 있다는 것에 주목하라. 간단한 타입으로 시작하려고 더 일반적인 수치 타입 대신 정수를 썼고, 유연한 함수가 되게 하려고 인자들을 순서쌍으로 받는 대신 커리했고 (3.6 절을 보라), 읽기 쉽게 하려고 정수 인자를 리스트 인자보다 앞에 오도록 했으며 (*drop n xs* 는 영어 문장 “drop *n* elements from *xs*” 와 비슷하므로 자연스럽다), 마지막으로는 일반적인 함수를 만들기 위해 리스트의 원소 타입을 여러모양 타입으로 정했다.

옮긴이 주: 프로그래밍 언어가 서구권에서 만들어졌기 때문에 그들의 문장 구조를 어느 정도 반영할 수 밖에 없다. 프로그래밍 언어 뿐 아니라 고대 논리학에서 현대 형식논리에 이르기까지 수학과 논리학 전반에 걸쳐 나타나는 현상이다. 이는 우리 입장에서는 또 하나의 어려움으로 다가오며 단순히 우리말로 옮기는 것으로는 극복할 수 없고 영어 혹은 그와 유사한 서구권의 언어에 대한 기본적인 이해가 필요하다는 점에서 더욱 안타깝게 생각한다. 우리말에서는 이러한 순서가 자연스럽지 않다. 오히려 “ xs 에서 n 개의 원소를 버리라 ($drop$)”는 표현이 자연스럽다. 그래서 만약 우리나라 사람들이 프로그래밍 언어를 만들기 시작했다면 아마도 $xs n drop$ 이런 순서로 배열하지 않았을까 생각한다. 실제로 Forth 같은 후위 표현 postfix 프로그래밍 언어도 있는데, 우리들에게는 이러한 배열의 언어가 더 자연스럽게 읽힐 수 있다고 생각한다. 혹시 우리말 바탕의 새로운 프로그래밍 언어를 만들어 보고자 하는 사람이 있다면, 후위 표현의 함수형 언어로 설계하는 것은 어떨까 제안해 본다.

둘째 걸음: 모든 경우를 나누어 늘어놓으라

정수 인자는 두 가지 표준적인 경우가 있고 (0 과 $n + 1$) 리스트 인자도 두 가지 표준적인 경우가 있으므로 ($[]$ 과 $x : xs$), 함수 패턴 매칭을 써서 총 네 가지 경우로 나누어 다음과 같이 함수 정의의 뼈대를 놓을 수 있다.

$$\begin{array}{lll} drop\ 0 & [] & = \\ drop\ 0 & (x : xs) & = \\ drop\ (n + 1)\ [] & = \\ drop\ (n + 1)\ (x : xs) & = \end{array}$$

셋째 걸음: 간단한 경우부터 정의하라

정의상 어떤 리스트에서든 0개의 원소를 버리는 것은 원래 리스트와 같으므로, 첫 두 경우는 다음과 같이 쉽게 정의할 수 있다.

$$\begin{array}{lll} drop\ 0 & [] & = [] \\ drop\ 0 & (x : xs) & = (x : xs) \\ drop\ (n + 1)\ [] & = \\ drop\ (n + 1)\ (x : xs) & = \end{array}$$

빈 리스트에서 하나 이상의 원소를 버리려는 것은 잘못되었으므로, 세 번째 경우를 생략하여 그러한 상황이 발생했을 경우 잘못이 나도록 할 수도 있다. 하지만 실제로는, 잘못을 내는 것을 피하고 그러한 경우 빈 리스트를 돌려주도록 다음과 같이 정의한다.

$$\begin{aligned} \text{drop } 0 & \quad [] = [] \\ \text{drop } 0 & \quad (x : xs) = (x : xs) \\ \text{drop } (n + 1) & [] = [] \\ \text{drop } (n + 1) & (x : xs) = \end{aligned}$$

넷째 걸음: 나머지 경우를 정의하라

비어 있지 않은 리스트에서 하나 이상의 원소를 어떻게 버릴 수 있을까? 첫 원소를 제외한 나머지 리스트에서 원래 주어진 수보다 하나 작은 개수만큼의 원소를 버리는 것으로 다음과 같이 쉽게 정의할 수 있다.

$$\begin{aligned} \text{drop } 0 & \quad [] = [] \\ \text{drop } 0 & \quad (x : xs) = (x : xs) \\ \text{drop } (n + 1) & [] = [] \\ \text{drop } (n + 1) & (x : xs) = \text{drop } n xs \end{aligned}$$

다섯째 걸음: 일반화하고 간단히 하라

drop 함수는 어떤 종류의 정수에 적용하든 상관 없으므로, 그 타입을 아무 정수같은 타입을 받도록 다음과 같이 일반화할 수 있으며, 이 때 Int 와 Integer 는 표준적인 인스턴스이다.

$$\text{drop} :: \text{Integral } b \Rightarrow b \rightarrow [a] \rightarrow [a]$$

하지만 효율상의 이유로, 3.9 절에서 나오듯, 실제로는 표준 서막에서 이런 일반화를 하지 않는다. 정의를 간단히 하는 것으로는, *drop* 의 첫 두 등식을 어떤 리스트에서 0개의 원소를 버리면 원래 리스트와 똑같다는 것을 나타내는 하나의 등식으로 다음과 같이 묶어낼 수 있다.

$$\begin{aligned} \text{drop } 0 & \quad xs = xs \\ \text{drop } (n + 1) & [] = [] \\ \text{drop } (n + 1) & (x : xs) = \text{drop } n xs \end{aligned}$$

또한, 마지막 등식의 변수 x 는 등식의 몸체에 쓰이지 않으므로 아무거나 패턴 $_$ 로 다음과 같이 바꿀 수 있다.

$$\begin{aligned} \text{drop } 0 & \quad xs &= xs \\ \text{drop } (n + 1) [] & = [] \\ \text{drop } (n + 1) (_ : xs) & = \text{drop } n xs \end{aligned}$$

마찬가지로 둘째 등식의 n 을 $_$ 로 바꿀 수는 없나 하고 생각할 수도 있겠지만, 이는 올바른 정의가 아닌데, 그 이유는 $n + k$ 패턴에서 n 은 반드시 변수여야만 하기 때문이다. 이러한 제약을 피해 둘째 등식의 $n + 1$ 패턴을 통째로 $_$ 로 바꿀 수도 있지만, 이렇게 하면 함수의 동작이 달라진다. 예컨대, $\text{drop } (-1) []$ 을 계산하면 원래는 잘못이 나와야 하는데 이렇게 바꾸고 나면 빈 리스트가 나온다. 그 이유는 $_$ 는 아무 정수나 들어맞지만 $n + 1$ 은 1 이상의 정수만 들어맞기 때문이다.

결론적으로, 위 drop 함수의 최종적인 정의는 다음과 같으며, 이는 부록 A의 표준 서막에 있는 정의와 똑같다.

$$\begin{aligned} \text{drop} & \quad :: \text{Int} \rightarrow [a] \rightarrow [a] \\ \text{drop } 0 & \quad xs &= xs \\ \text{drop } (n + 1) [] & = [] \\ \text{drop } (n + 1) (_ : xs) & = \text{drop } n xs \end{aligned}$$

보기 - init

마지막 보기로서, 비어 있지 않은 리스트에서 마지막 원소를 버리는 라이브러리 함수 $init$ 을 어떻게 만들어 나가는지 살펴보자.

첫째 걸음: 타입을 정의하라

$init$ 이 임의의 타입 a 로 이루어진 리스트를 받아 같은 타입의 리스트를 내도록 다음과 같은 타입으로 시작하자.

$$init :: [a] \rightarrow [a]$$

둘째 걸음: 모든 경우를 나누어 늘어놓으라

빈 리스트는 *init*에 적당치 않은 인자이므로, 한 경우에 대한 패턴 매칭만을 써서 다음과 같은 뼈대를 정의할 수 있다.

$$\text{init } (x : xs) =$$

셋째 걸음: 간단한 경우부터 정의하라

앞서 살펴본 두 보기에서는 이 셋째 걸음이 쉽고 간단했는데, *init*의 경우에는 그보다 약간 더 생각을 해야 한다. 정의상, 원소 한개짜리 리스트에서 마지막 원소를 버리면 빈 리스트가 되므로 다음과 같이 보초를 써서 간단한 경우를 처리할 수 있다.

$$\begin{aligned} \text{init } (x : xs) &| \text{null } xs = [] \\ &| \text{otherwise} = \end{aligned}$$

라이브러리 함수 *null*은 리스트가 비었는지 검사하는 함수임을 기억하라.

넷째 걸음: 나머지 경우를 정의하라

두 개 이상의 원소로 이루어진 리스트에서 어떻게 마지막 원소를 버릴 수 있겠는가? 첫 원소는 그대로 두고 나머지 리스트의 마지막 원소를 버리도록 다음과 같이 간단히 정의할 수 있다.

$$\begin{aligned} \text{init } (x : xs) &| \text{null } xs = [] \\ &| \text{otherwise} = x : \text{init } xs \end{aligned}$$

다섯째 걸음: 일반화하고 간단히 하라

*init*의 타입은 이미 가장 일반적이지만, 정의 자체는 보초 대신에 패턴 매칭을 써서 간단히 할 수 있으며, 그 중 첫 등식은 변수 대신 아무거나 패턴 wildcard pattern 을 써서 정의할 수 있다.

$$\begin{aligned} init &:: [a] \rightarrow [a] \\ init [-] &= [] \\ init (x : xs) &= x : init xs \end{aligned}$$

이전의 보기와 마찬가지로, 이 정의는 표준 서막에 있는 것과 똑같다.

6.7 살펴보기

이 장에서 보인 되도는 정의들은 명확함을 추구했으나, 12 장과 13 장에 나오는 기술을 쓰면 그 중 여러 함수의 효율을 개선할 수 있다. 함수를 정의하는 다섯 걸음의 과정은 (10)을 바탕으로 한 것이다.

6.8 연습문제

1. 곱셈 연산자 *를 되돌기로 정의한 것과 같은 방식으로 음이 아닌 정수에 대한 거듭제곱 연산자 ↑를 정의하고, 그 정의를 이용해 $2 \uparrow 3$ 의 계산 과정을 보이라.
2. 이 장에 주어진 함수 정의를 이용해 $\text{length } [1, 2, 3]$, $\text{drop } 3 [1, 2, 3, 4, 5]$, $\text{init } [1, 2, 3]$ 의 계산 과정을 보이라.
3. 표준 서막의 정의를 보지 않고 다음 라이브러리 함수들을 되돌기로 정의하라.
 - 리스트에 있는 논리값들이 모두 *True* 인지 검사한다.

$\text{and} :: [\text{Bool}] \rightarrow \text{Bool}$

- 리스트의 리스트를 이어붙인다.

$\text{concat} :: [[a]] \rightarrow [a]$

- n 개의 같은 원소로 이루어진 리스트를 만든다.

$\text{replicate} :: \text{Int} \rightarrow a \rightarrow [a]$

- 리스트의 n 번째 원소를 고른다.

$(!!) :: [a] \rightarrow \text{Int} \rightarrow a$

- 어떤 값이 리스트의 원소인지 검사한다.

$\text{elem} :: \text{Eq } a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}$

참고: 사실 표준 서막에서 대부분의 위 함수들은 되돌기가 드러나 보이게 정의하기보다는, 다른 라이브러리 함수를 써서 정의한다.

4. 되도는 함수 $\text{merge} :: \text{Ord } a \Rightarrow [a] \rightarrow [a] \rightarrow [a]$ 를 정의하라. merge 는 두 정렬된 리스트를 합쳐 하나의 정렬된 리스트를 만든다. 예컨대,

```
> merge [2,5,6] [1,3,4]
[1,2,3,4,5,6]
```

참고: *insert* 나 *sort* 와 같이 정렬된 리스트를 다루는 다른 함수를 쓰지 않고 드러나 보이는 되돌기로 *merge*를 정의해야 한다.

5. *merge* 를 써서 합병정렬 merge sort 을 하는 $m\text{sort} :: \text{Ord } a \Rightarrow [a] \rightarrow [a]$ 함수를 되돌기로 정의하라. 합병정렬이란, 빈 리스트는 이미 정렬되었다고 정의하며, 비어 있지 않은 리스트의 경우에는 리스트를 반쪽으로 나눈 두 리스트를 각각 정렬한 후 하나의 정렬된 리스트로 합치는 것으로 정의한다.
귀띔: 우선 하나의 리스트를 두 개의 리스트로 반토막내는 $\text{halve} :: [a] \rightarrow ([a], [a])$ 함수부터 정의하라. 이때, 반토막난 두 리스트의 길이는 많아야 하나만큼 차이나도록 한다.
6. 리스트에 있는 수의 합을 구하는 함수, 주어진 수 만큼의 원소를 리스트에서 취하는 함수, 비어 있지 않은 리스트에서 마지막 원소를 고르는 함수를 각각 다섯 걸음의 과정으로 정의해 보라.

제 7 장

함수를 주고받는 higher-order 함수

이 장에서는 함수를 주고받는 함수에 대해 알아보겠다. 함수를 주고받는 함수를 이용하면, 프로그램을 짤 때 흔히 나타나는 반복적인 유형을 요약하여 함수로 나타낼 수 있다. 함수를 주고받는 함수란 무엇인지 그리고 왜 쓸모있는지 알아보는 것으로 시작하여, 표준 서막에서 리스트를 다루는 함수를 주고받는 함수들에 대해 알아보고, 함수의 합성에 대해 살펴본 후, 글줄 전송기 string transmitter 를 만들어 보는 것으로 끝맺는다.

7.1 기본 개념

앞서 여러 장에 걸쳐 살펴 본 바와 같이, 하스켈에서는 인자가 여럿인 함수를 주로 커리한다 currying 는 개념을 이용해 정의한다. 즉, 함수가 함수를 돌려 줄 수 있다는 사실을 이용해 인자를 한 번에 하나씩 취하는 것이다. 예컨대,

```
add      :: Int → Int → Int
add x y = x y
```

위와 같은 정의가 뜻하는 바는 다음과 같으며,

```
add :: Int → (Int → Int)
add = λx → (λy → x + y)
```

이는 *add* 함수가 정수 *x*를 받아 함수를 돌려주는데, 그 함수는 다시 정수 *y*를 받아 두 정수의 합 $x + y$ 를 돌려준다는 뜻이다. 또한, 하스켈에서는 함수가 그 인자로 함수를 취하는 것도 가능하다. 예컨대, 어떤 함수와 어떤 값을 받아 그 받은 함수를 그 받은 값에 두 번 적용하는 함수를 다음과 같이 정의할 수 있다.

```
twice    :: (a → a) → a → a
twice f x = f (f x)
```

예컨대,

```
> twice (*2) 3
12
```

```
> twice reverse [1,2,3]
[1,2,3]
```

그리고, *twice* 가 커리된 curried 함수이므로 인자 하나만을 부분 적용하여 다른 쓸모있는 함수를 만들어 낼 수도 있다. 예컨대, 주어진 수의 네 배인 수를 계산하는 함수를 *twice (*2)* 와 같이 만들 수 있으며, (유한한 길이의) 리스트의 순서를 두 번 뒤집으면 원래의 리스트와 같다는 사실을 *twice reverse = id* 와 같은 등식으로 나타낼 수 있다. 앞의 등식에서 *id* 는 항등함수 identity function 로서 그 정의는 *id x = x* 이다.

함수를 주고받는다 higher-order 는 엄밀한 의미는 함수가 함수를 인자로 받거나 돌려주는 것이다. 하지만, 실제로, 커리되었다 curried 라는 용어가 이미 함수를 돌려준다는 뜻을 대표해 쓰이고 있으므로, 함수를 주고받는다 higher-order 는 용어는 보통 함수를 인자로 받는다는 뜻으로 쓰는 경우가 많다. 이 장에서도 그 두 번째 뜻인 함수를 인자로 받는 함수를 주제로 다룬다.

함수를 주고받는 함수를 쓰면 프로그래밍할 때 흔히 나타나는 유형을 함수로 요약할 수 있어 언어 자체로 그러한 유형을 표현할 수 있게 되기 때문에 하스켈의 표현 능력이 크게 향상된다. 더 일반적으로, 함수를 주고받는 함수로 “전문 영역 언어”domain specific language를 하스켈 안에서 정의할 수 있다. 예컨대, 이 장에서 리스트를 조작하기 위한 간단한 언어를 살펴 볼 것이며, 8장과 9장에서는 문법 분석기parser와 대화식 프로그램interactive program을 만드는 데 같은 원리를 적용할 것이다.

7.2 리스트 다루기

표준 서막에는 리스트를 다루는 쓸모있는 함수를 주고받는 함수들이 여럿 정의되어 있다. 예컨대, 어떤 함수를 리스트의 모든 원소에 각각 적용시키는 *map* 함수를 다음과 같이 조건제시식을 써서 정의할 수 있다.

$$\begin{aligned} \text{map} &:: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\ \text{map } f \text{ xs} &= [f x \mid x \leftarrow xs] \end{aligned}$$

즉, *map f xs*는 리스트 *xs*의 모든 원소 *x*에 대한 *f x* 값을 리스트로 돌려준다. 예컨대,

```
> map (+1) [1,3,5,7]
[2,4,6,8]
```

```
> :load Char
Char> map isDigit ['a', '1', 'b', '2']
[False, True, False, True]
```

```
> map reverse ["abc", "def", "ghi"]
["cba", "fed", "ihg"]
```

옮긴이 주: *isDigit* 함수는 *Char* 모듈에 있으므로, *isDigit* 함수를 쓰려면 Hugs에서 *:load Char* 명령어로 불러들여야 하며 (GHC의 대화식 환경인 ghci에서는 load 대신 module 명령어를 써야 한다.), 스크립트 안에서 *isDigit* 함수를 쓰려면 스크립트 맨 앞부분에 **import Char**라고 써서 *Char* 모듈을 불러들여야 한다.

map 함수에 대해 더 알아야 할 것이 서너 가지 있다. 우선 첫째로, 다른 대부분의 리스트를 다루는 함수를 주고받는 함수와 마찬가지로, *map* 함수는 아무 타입의 리스트에나 적용할 수 있는 여러모양 polymorphic 함수다. 둘째로, 자기 자신에 적용함으로써 포개진 nested 리스트를 다룰 수 있다. 예컨대, *map (map (+1))* 함수는 정수 리스트의 리스트에 있는 모든 정수를 다음과 같이 1만큼씩 증가시킨다.

$$\begin{aligned} & \text{map } (\text{map } (+1)) [[1, 2, 3], [4, 5]] \\ = & \quad \{ \text{ 바깥쪽 map 을 적용 } \} \\ & [\text{map } (+1) [1, 2, 3], \text{map } (+1) [4, 5]] \\ = & \quad \{ \text{ map 을 적용 } \} \\ & [[2, 3, 4], [5, 6]] \end{aligned}$$

그리고, 마지막으로, *map* 함수를 되돌기 recursion로 정의할 수 있다.

$$\begin{aligned} \text{map } f [] &= [] \\ \text{map } f [x : xs] &= f x : \text{map } f xs \end{aligned}$$

즉, 어떤 함수를 리스트의 모든 원소에 적용하려면, 빈 리스트인 경우는 빈 리스트를 돌려주고, 비어 있지 않은 리스트인 경우에는 일단 함수를 첫 원소에 적용하고 그 다음 나머지 모든 원소에 같은 방법으로 함수를 적용해 나가면 된다는 뜻이다. 리스트 조건제시식을 이용한 원래 *map*의 정의가 더 간단하지만, (13장에서 살펴볼) 논리적 증명에는 되도록는 recursive 정의가 더 알맞다.

또 하나의 쓸모 있는 라이브러리 함수인 *filter*는 리스트의 원소들 중 어떤 술어 predicate를 만족하는 원소만을 고르는데, 이 때 술어 predicate(혹은 성질 property)란 논리값을 돌려주는 함수이다.

map 과 마찬가지로 *filter* 함수도 리스트 조건제시식으로 간단히 정의할 수 있다.

$$\begin{aligned} \text{filter} &:: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \\ \text{filter } p \text{ } xs &= [x \mid x \leftarrow xs, p \ x] \end{aligned}$$

즉, *filter p xs* 는 리스트 *xs* 의 모든 원소 *x*에 대해 *p x* 가 *True* 인 모든 *x*의 리스트를 돌려준다. 예컨대,

> *filter even [1..10]*

[2, 4, 6, 8, 10]

> *filter (>5) [1..10]*

[6, 7, 8, 9, 10]

> *filter (≠ ,) "abc def ghi"*

"abcdefghi"

map 과 마찬가지로, *filter* 함수는 아무 타입의 리스트에나 적용할 수 있으며 논증에 알맞게 되돌기로 정의할 수도 있다.

$$\begin{aligned} \text{filter } p \text{ } [] &= [] \\ \text{filter } p \text{ } (x : xs) \mid p \ x &= x : \text{filter } p \text{ } xs \\ \mid \text{otherwise} &= \text{filter } p \text{ } xs \end{aligned}$$

즉, 어떤 조건을 만족하는 모든 원소를 찾으려면, 빈 리스트인 경우는 빈 리스트를 돌려주고, 비어 있지 않은 리스트의 경우에는 첫 원소가 조건을 만족하는지 아닌지에 따라 그 결과가 달라진다. 첫 원소가 조건을 만족하면 첫 원소를 그대로 남기고 나머지 리스트의 원소들을 걸러 나가고, 조건을 만족하지 않으면 첫 원소를 버리고 그냥 나머지 리스트의 원소들만 걸러 나간다.

*map*과 *filter* 함수는 프로그램에서 같이 쓰는 경우가 많은데, *filter*로 리스트에서 알맞은 원소들만을 골라 *map*으로 골라낸 각각의 원소들을 변환하는 식으로 쓰곤 한다. 예컨대, 주어진 리스트에서 짝수들만의 제곱의 합을 구하는 함수를 다음과 같이 정의할 수 있다.

```
sumsqreven    :: [Int] → Int
sumsqreven ns = sum (map (↑2) (filter even ns))
```

표준 서막에 있는 다음 몇 가지 함수를 주고받는 함수들을 보기를 곁들여 간단히 설명하는 것으로 이 절을 마무리한다.

- 모든 원소가 주어진 조건을 만족하는지 검사한다.

```
> all even [2,4,6,8]
True
```

- 주어진 조건을 만족하는 원소가 하나라도 있는지 검사한다.

```
> any odd [2,4,6,8]
False
```

- 주어진 조건을 만족하는 원소들을 리스트 처음부터 만족하는 동안 계속해서 고른다.

```
> takeWhile isLower "abc def"
"abc"
```

- 주어진 조건을 만족하는 원소들을 리스트 처음부터 만족하는 동안 계속해서 버린다.

```
> dropWhile isLower "abc def"
" def"
```

7.3 foldr 함수

리스트를 인자로 받는 여러 함수들은 다음과 같이 간단한 형태의 리스트에 대한 되돌기로 정의할 수 있다.

$$\begin{aligned} f [] &= v \\ f (x : xs) &= x \oplus f xs \end{aligned}$$

즉, 위 함수는 빈 리스트를 v 라는 값에 대응시키고, 비어 있지 않은 리스트는 \oplus 연산자를 리스트의 첫 원소와 나머지 리스트를 되돌며 recursively 처리한 결과에 적용한 것에 대응시킨다. 예컨대, 익숙한 여러 라이브러리 함수들을 이러한 형태로 다음과 같이 정의할 수 있다.

$$\begin{aligned} sum [] &= 0 \\ sum (x : xs) &= x + sum xs \\ product [] &= 1 \\ product (x : xs) &= x * product xs \\ or [] &= False \\ or (x : xs) &= x \vee or xs \\ and [] &= True \\ and (x : xs) &= x \wedge and xs \end{aligned}$$

함수를 주고받는 라이브러리 함수 $foldr$ (“오른쪽으로부터 접어들어온다”fold right의 줄임)은 리스트에 대한 함수 정의에 쓰이는 이러한 형태의 되돌기를 연산자 \oplus 와 값 v 를 인자로 받는 함수로서 요약한다. 예컨대, $foldr$ 함수를 써서 위의 네 정의를 다음과 같이 바꿔 쓸 수 있다.

$$\begin{aligned} sum &= foldr (+) 0 \\ product &= foldr (*) 1 \\ or &= foldr (\vee) False \\ and &= foldr (\wedge) True \end{aligned}$$

(연산자를 인자로 넘기려면 괄호로 감싸야 한다는 것을 기억하라.) 이 새로운 정의에다가도 $sum xs = foldr (+) 0 xs$ 와 같이 리스트 인자를 드러나 보이게 쓸 수도 있지만, 부분 적용식으로 인자를 보이지 않게 하는 것이 더 간결하기 때문에 더 좋다.

foldr 함수 자체는 다음과 같이 되돌기로 정의할 수 있다.

$$\begin{aligned} \text{foldr} &:: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\ \text{foldr } f \ v [] &= v \\ \text{foldr } f \ v (x : xs) &= f x (\text{foldr } f \ v xs) \end{aligned}$$

즉, *foldr f v* 이라는 함수는 빈 리스트를 값 *v*에 대응시키며, 비어 있지 않은 리스트를 *f*를 첫 원소에 적용한 값과 나머지 리스트를 되돌며 처리한 값에 적용한 것에 대응시킨다는 것이다. 하지만, 실제로, *foldr f v*를 되돌기로 생각하기보다는, 리스트에서 콘스^{cons} 연산자를 함수 *f*로 바꿔치고 빈 리스트를 값 *v*로 바꿔친 것으로 간단히 생각하는 것이 가장 이해하기 편하다. 예컨대,

$$1 : (2 : (3 : []))$$

함수 *foldr (+) 0*를 위 리스트에 적용하면 다음과 같은 결과를 얻는데,

$$1 + (2 + (3 + 0))$$

이는 :와 []를 각각 +와 0로 바꾸어 쓴 것과 같다. 즉, *sum = foldr (+) 0*라는 정의는 리스트에 있는 수의 합을 구하는 것이 곧 각 콘스를 덧셈으로 그리고 빈 리스트를 0으로 바꿔 쓰는 것과 같다는 뜻이 된다.

*foldr*은 간단한 형태의 되돌기를 요약할 뿐이지만, 처음 봤을 때 예상했던 것보다 훨씬 더 많은 함수를 정의할 수 있다. 우선 첫째로, 라이브러리 함수 *length*의 다음과 같은 정의를 기억하라.

$$\begin{aligned} \text{length} &:: [a] \rightarrow \text{Int} \\ \text{length} [] &= 0 \\ \text{length } (-, xs) &= 1 + \text{length } xs \end{aligned}$$

예컨대,

$1 : (2 : (3 : []))$

length 를 위 리스트에 적용하면 아래와 같은 결과를 얻는다.

$1 + (1 + (1 + 0))$

즉, 리스트의 길이를 계산한다는 것은 곧 각 콘스를 그 둘째 인자마다 1을 더하는 함수로 바꾸고 빈 리스트를 0 으로 바꾸는 것과 같다. 따라서, *length* 의 정의를 다음과 같이 *foldr* 을 이용해 다시 쓸 수 있다.

$\text{length} = \text{foldr} (\lambda n \rightarrow 1 + n) 0$

이제 다음과 같이 드러나 보이는 되돌기로 정의된 리스트의 순서를 뒤집는 라이브러리 함수를 생각해 보자.

reverse	$:: [a] \rightarrow [a]$
$\text{reverse} []$	$= []$
$\text{reverse} (x : xs)$	$= \text{reverse} xs ++ [x]$

예컨대,

$1 : (2 : (3 : []))$

reverse 를 위 리스트에 적용하면 다음과 같은 결과를 얻는다.

$(([] ++ [3]) ++ [2]) ++ [1]$

위의 정의나 보기를 보고서 *reverse* 를 어떻게 *foldr* 로 정의할지 바로 알아차리기는 힘들 수도 있다. 하지만, 리스트의 처음이 아닌 끝에 새로운 원소를 덧붙이는 함수 $\text{snoc } x xs = xs ++ [x]$ 을 정의하고 나면 (“*snoc*” 은 콘스^{cons} 를 거꾸로 쓴 것),

reverse 를 다음과 같이 다시 정의할 수 있으며,

$$\begin{aligned} \text{reverse} [] &= [] \\ \text{reverse} (x : xs) &= \text{snoc } x (\text{reverse } xs) \end{aligned}$$

다시 쓴 정의로부터 다음과 같이 *foldr* 을 이용한 정의를 곧바로 이끌어 낼 수 있다.

$$\text{reverse} = \text{foldr snoc } []$$

즉, 리스트를 뒤집는다는 것은 곧 각 콘스^{cons} 를 *snoc* 으로 바꾸고 빈 리스트는 그냥 빈 리스트로 그대로 두는 것으로 정의할 수 있다. *snoc* 의 정의에 쓰이는 이어붙이기 연산자 ++ 또한 *foldr* 을 써서 다음과 같이 아주 간결하게 정의할 수 있다.

$$\begin{aligned} \text{-- } (\text{++ } ys) &= \text{foldr } (:) ys \text{ 임을 뜻함} \\ (\text{++}) &= \text{flip } (\text{foldr } (:)) \end{aligned}$$

즉, 어떤 리스트의 끝에 ys 를 이어붙인다는 것은 곧 그 리스트의 각 콘스는 그대로 두고 마지막의 빈 리스트만 ys 로 바꾸는 것과 같다느 뜻이다.

foldr 이 오른쪽으로부터 접어들어간다는 말은 넘겨주는 연산자가 오른쪽부터 (괄호로) 묶인다는 사실을 반영하는 것임을 언급하며 이 절을 마치고자 한다. 예컨대, $\text{foldr } (+) 0 [1, 2, 3]$ 을 계산하여 $1 + (2 + (3 + 0))$ 의 결과가 나온다고 할 때, 괄호가 쳐진 것을 보면 덧셈이 오른쪽부터 묶이고 있다는 것을 알 수 있다. 더 일반적으로는, *foldr* 의 동작을 다음과 같이 정리할 수 있다.

$$\text{foldr } (\oplus) v [x_0, x_1, \dots, x_n] = x_0 \oplus (x_1 \oplus (\dots (x_n \oplus v) \dots))$$

7.4 foldl 함수

왼쪽부터 (괄호로) 묶이는 연산을 하는 리스트에 대한 되도는 함수를 정의하는 것도 가능하다. 예컨대, sum 함수를 이런 식으로 최종 결과를 쌓아올리는 인자 v 를 하나 더 받는 보조 함수 sum' 을 써서 다음과 같이 다시 정의할 수 있다.

```
sum = sum' 0
where
  sum' v []      = v
  sum' v (x : xs) = sum' (v + x) xs
```

예컨대,

$$\begin{aligned} & sum [1, 2, 3] \\ &= \{ sum 을 적용 \} \\ &= sum' 0 [1, 2, 3] \\ &= \{ sum' 를 적용 \} \\ &= sum' (0 + 1) [2, 3] \\ &= \{ sum' 를 적용 \} \\ &= sum' ((0 + 1) + 2) [3] \\ &= \{ sum' 를 적용 \} \\ &= sum' (((0 + 1) + 2) + 3) [] \\ &= \{ sum' 를 적용 \} \\ &= (((0 + 1) + 2) + 3) \\ &= \{ + 를 적용 \} \\ &= 6 \end{aligned}$$

위 계산에서는 덧셈이 왼쪽부터 괄호로 묶이고 있음을 알 수 있다. 하지만, 실제로, 이런 경우에는 어느 쪽부터 괄호로 묶든 그 결과값이 같은데, 이는 덧셈이 아무 쪽으로나 묶어도 결과가 같은 결합법칙을 만족하기 때문이다. 즉, $x + (y + z) = (x + y) + z$ 가 모든 수 x, y, z 에 대해 성립한다.

위의 sum 에 대한 보기일 일반화하여 생각해 보면, 많은 리스트에 대한 함수들이 다음과 같이 간단한 되도는 형태로 정의된다는 것을 알 수 있다.

$$\begin{array}{l} f\ v\ [] = v \\ f\ v\ (x : xs) = f\ (v \oplus x)\ xs \end{array}$$

즉, 위 함수는 빈 리스트를 쌓개 accumulator 값 v 로 바꾸고, 비어 있지 않은 리스트를 새 쌓개 값과 첫 원소를 제외한 나머지 리스트를 되돌며 처리한 것에 대응시키는데, 이 때 새 쌓개는 현재의 쌓개와 첫 원소에 \oplus 연산자를 적용하여 얻은 값이다. 함수를 주고받는 라이브러리 함수 $foldl$ (“왼쪽으로부터 접어들어간다” fold left 의 줄임)은 위와 같은 형태의 되돌기를 \oplus 연산자와 쌓개 v 를 인자로 하는 함수로서 요약한다. 예컨대, $foldl$ 을 써서 앞서 살펴본 sum 의 정의를 다음과 같이 더 간결하게 고쳐 쓸 수 있다.

$$sum = foldl\ (+)\ 0$$

마찬가지로, 다음과 같이 정의할 수 있다.

$$\begin{array}{l} product = foldl\ (*)\ 1 \\ or = foldl\ (\vee)\ False \\ and = foldl\ (\wedge)\ True \end{array}$$

앞절에서 살펴본 $foldr$ 을 이용한 다른 보기들도 다음과 같이 알맞은 연산을 적용하면 $foldl$ 로 다시 정의할 수 있다.

$$\begin{array}{l} length = foldl\ (\lambda n\ _{-} \rightarrow n + 1)\ 0 \\ reverse = foldl\ (\lambda xs\ x \rightarrow x : xs)\ [] \\ append = foldl\ (\lambda ys\ y \rightarrow ys ++ [y]) \end{array}$$

예컨대, 이 새로운 정의로 다음과 같이 계산할 수 있다.

$$\begin{array}{ll} length\ [1, 2, 3] & = ((0 + 1) + 1) + 1 \\ reverse\ [1, 2, 3] & = 3 : (2 : (1 : [])) \\ [1, 2, 3] ++ [4, 5] & = ([1, 2, 3] ++ [4]) ++ [5] \end{array}$$

위의 보기처럼 함수를 *foldr*로 정의할 수도 있고 *foldl*로 정의할 수 있을 때, 둘 중 어느 정의가 더 나은지 대개 그 효율성에 기반해 판단하는데, 이를 위해서는 12장에서 다룬 하스켈이 바탕으로 삼는 계산 원리에 대한 신중한 고찰이 필요하다.

foldl 함수 자체는 다음과 같이 되돌기로 정의할 수 있다.

$$\begin{aligned} \text{foldl} &:: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a \\ \text{foldl } f \ v [] &= v \\ \text{foldl } f \ v (x : xs) &= \text{foldl } f (f \ v x) xs \end{aligned}$$

하지만, *foldr*의 경우와 마찬가지로, *foldl*의 동작을 되돌기로 생각하기보다는, 왼쪽부터 묶이는 \oplus 연산자를 써서 나타낸 다음과 같은 등식으로 생각하는 것이 가장 이해하기 편하다.

$$\text{foldl } (\oplus) \ v [x_0, x_1, \dots, x_n] = (\dots((v \oplus x_0) \oplus x_1) \dots \oplus x_n)$$

7.5 함수 합성 연산자

두 함수의 합성함수를 하나의 함수로서 돌려주는, 함수를 주고받는 라이브러리 연산자 \circ 를 다음과 같이 정의할 수 있다.

$$\begin{aligned} (\circ) &:: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c) \\ f \circ g &= \lambda x \rightarrow f(g x) \end{aligned}$$

즉, $f \circ g$ (“ f 와 g 의 합성”이라고 읽음)라는 함수는 인자 x 를 받으면 함수 g 를 이 인자에 적용하고, 그 결과에 함수 f 를 적용한다. 이 연산자를 $(f \circ g) x = f(g x)$ 와 같이 정의할 수도 있다. 그러나, 인자 x 를 람다식을 써서 함수 몸체 안으로 돌리는 것이 함수 합성 연산자가 그 결과로 함수를 돌려준다는 것을 드러나 보이게 한다는 점에서는 더 좋다.

함수 합성 연산자를 쓰면 포개진 함수 적용에서 괄호를 없애고, 그 처음 넘어온 인자를 드러나 보이게 참조하지 않아도 되도록 간단히 할 수 있다. 예컨대,

$$\begin{aligned} \text{odd } n &= \neg (\text{even } n) \\ \text{twice } f \ x &= f (f \ x) \\ \text{sumsqreven } ns &= \text{sum} (\text{map} (\uparrow 2) (\text{filter even} \ ns)) \end{aligned}$$

함수 합성 연산자를 써서 위 정의를 다음과 같이 더 간단한 정의로 바꿔 쓸 수 있다.

$$\begin{aligned} \text{odd} &= \neg \circ \text{even} \\ \text{twice } f &= f \circ f \\ \text{sumsqreven} &= \text{sum} \circ \text{map} (\uparrow 2) \circ \text{filter even} \end{aligned}$$

마지막 정의는 함수 합성 연산자가 뭇이는 순서에 상관하지 않음을 이용하고 있다. 즉, $f \circ (g \circ h) = (f \circ g) \circ h$ 가 알맞은 타입의 모든 함수 f, g, h 에 대해 성립한다. 따라서, 세 개 이상의 함수를 합성할 때는 괄호로 어느 것이 먼저 뭇이는지 드러내 보일 필요가 없는데, 이는 어느 것이 먼저 뭇이든 그 결과에는 영향을 미치지 않기 때문이다.

함수 합성 연산에도 항등원 identity 이 있으니, 이는 바로 다음과 같은 항등함수 identity function 이다.

$$\begin{aligned} id &:: a \rightarrow a \\ id &= \lambda x \rightarrow x \end{aligned}$$

즉, id 는 받은 인자를 그냥 그대로 돌려주는 함수로서, 모든 함수 f 에 대해 $id \circ f = f$ 와 $f \circ id = f$ 의 성질을 만족한다. 항등함수는 프로그램에 대한 논리적 증명에 쓸모 있는 경우가 많으며, 여러 함수를 합성하고자 할 때 알맞은 처음값으로도 쓰인다. 예컨대, 리스트의 모든 함수를 합성하는 함수를 다음과 같이 정의할 수 있다.

$$\begin{aligned} \text{compose} &:: [a \rightarrow a] \rightarrow (a \rightarrow a) \\ \text{compose} &= \text{foldr} (\circ) id \end{aligned}$$

7.6 글줄 string 전송기

더 큰 보기로 살펴보는 것으로 이 장을 끝맺고자 한다. 글줄을 0과 1로 전송하는 시늉을 내는 문제를 생각해보자. 더 정확히 말하자면, 글줄을 0과 1의 리스트로 바꾸는 함수와 함께 그 반대의 역할을 하는 함수를 정의하는 방법을 알아보자는 것이다.

이진수 binary number

사람들은 열 손가락을 가지고 있기 때문에 대개 10을 밑으로 하는 십진 decimal 표현을 가장 익숙하게 느낀다. 십진수 decimal number 란 0부터 9까지 아홉 개의 숫자를 한줄로 늘어놓은 것으로서, 가장 오른쪽의 숫자가 1만큼의 가중치를 가지며, 왼쪽으로 갈수록 숫자의 가중치가 열 배씩 늘어난다. 예컨대, 십진수 2345는 앞서 설명한 바와 같이 다음과 같은 뜻이다.

$$2345 = (1000 * 2) + (100 * 3) + (10 * 4) + (1 * 5)$$

즉, 십진수 2345는 각각 1000, 100, 10, 1의 가중치를 가지는 숫자 2, 3, 4, 5를 그 가중치와 곱한 합을 나타내며, 이를 계산하면 그 값은 2345이다.

한편, 컴퓨터에서는 더 기본적인 2를 밑으로 하는 이진 binary 표현을 쓰는 것이 대개 더 편리하다. 이진수 binary number 란 이진 숫자 binary digit 혹은 비트 bits 라 불리는 0과 1을 한줄로 늘어놓은 것으로서, 왼쪽으로 갈수록 비트의 가중치가 두 배씩 늘어난다. 예컨대, 이진수 1101은 다음과 같이 이해할 수 있다.

$$1101 = (8 * 1) + (4 * 1) + (2 * 0) + (1 * 1)$$

즉, 이진수 1101은 각각 가중치가 8, 4, 2, 1인 비트 1, 1, 0, 1을 그 가중치와 곱한 합을 나타내며, 이를 계산하면 그 값은 13이다.

이제부터 글줄 전송기를 만드는 동안 함수 정의를 간소화하기 위해 이진수를 쓸 때 낮은 자리부터 거꾸로 쓰도록 하겠다.

예컨대, 1101 을 이제부터 1011 이라고 쓰고, 오른쪽으로 갈수록 비트의 가중치가 2 배 증가하는 것으로 한다.

$$1011 = (1 * 1) + (2 * 0) + (4 * 1) + (8 * 1)$$

진법 변환 base conversion

함수의 타입에 더 의미를 부여하기 위해 다음과 같이 비트 타입을 정수 타입의 다른이름 synonym 으로서 정의한다.

type Bit = Int

비트의 리스트로 표현된 이진수는 다음과 같이 가중합 weighted sum 을 계산함으로써 간단히 정수로 바꿀 수 있다.

```
bin2int      :: [Bit] → Int
bin2int bits = sum [w * b | (w, b) ← zip weights bits]
  where weights = iterate (*2) 1
```

함수를 주고받는 라이브러리 함수 iterate 는 다음과 같이 주어진 함수를 어떤 값에 적용하는 회수를 하나씩 늘려나가는 리스트를 만들어낸다.

$$\text{iterate } f x = [x, f x, f(f x), f(f(f x)), \dots]$$

따라서 bin2int 의 정의에 쓰인 식 iterate (*2) 1 는 가중치 리스트 [1, 2, 4, 8, ...] 를 만들어낸다. 이 리스트는 표기상으로 볼 때는 무한하지만, 12 장에서 살펴볼 느긋한 계산법 하에서는 문맥상 꼭 필요한 만큼의 원소만 – 이 경우는 비트 리스트와 zip 하는 만큼만 – 실제로 만들어지게 된다. 예컨대,

> bin2int [1, 0, 1, 1]

13

하지만 약간의 대수적인 성질을 이용하면 $bin2int$ 를 더 간단히 정의할 수 있다. 임의의 4 비트로 이진수 $[a, b, c, d]$ 가 있다고 하자. 여기에 $bin2int$ 를 적용하면 다음과 같은 가중합을 얻는데,

$$(1 * a) + (2 * b) + (4 * c) + (8 * d)$$

이를 다음과 같이 바꾸어 쓸 수 있다.

$$\begin{aligned} & (1 * a) + (2 * b) + (4 * c) + (8 * d) \\ = & \quad \{ 1 * a \text{ 를 간단히 } \} \\ = & \quad a + (2 * b) + (4 * c) + (8 * d) \\ = & \quad \{ 2 \text{ 로 묶어내기 } \} \\ = & \quad a + 2 * (b + (2 * c) + (4 * d)) \\ = & \quad \{ 2 \text{ 로 묶어내기 } \} \\ = & \quad a + 2 * (b + 2 * (c + (2 * d))) \\ = & \quad \{ d \text{ 를 늘리기 } \} \\ = & \quad a + 2 * (b + 2 * (c + 2 * (d + 2 * 0))) \end{aligned}$$

마지막 결과는 비트 리스트 $[a, b, c, d]$ 를 정수로 바꾸는 것은 곧 각 콘스를 첫째 인자에 둘째 인자의 두 배를 더하는 함수로 바꾸고, 빈 리스트를 0 으로 바꾸는 것과 같다는 것을 나타낸다. 더 일반적으로, $bin2int$ 를 $foldr$ 을 써서 다음과 같이 다시 쓸 수 있다.

$$bin2int = foldr (\lambda x y \rightarrow x + 2 * y) 0$$

이제 그 반대로 음이 아닌 정수를 이진 리스트로 바꾸는 방법을 생각해 보자. 정수를 계속해서 2로 0이 될 때까지 나누면서 그 나머지를 취함으로써 이진 리스트를 만들어 낼 수 있다. 예컨대, 정수 13 으로 시작해서 다음과 같이 하면 된다.

$$\begin{aligned} 13 \text{ 나누기 } 2 &= 6 \text{ 나머지 } 1 \\ 6 \text{ 나누기 } 2 &= 3 \text{ 나머지 } 0 \\ 3 \text{ 나누기 } 2 &= 1 \text{ 나머지 } 1 \\ 1 \text{ 나누기 } 2 &= 0 \text{ 나머지 } 1 \end{aligned}$$

나머지를 한줄로 늘어놓은 1011이 바로 정수 13의 이진 표현이다. 이같은 과정을 다음과 같이 되돌기로 쉽게 구현할 수 있다.

```
int2bin    :: Int → [Bit]
int2bin 0 = []
int2bin n = n `mod` 2 : int2bin (n `div` 2)
```

예컨대,

```
> int2bin 13
[1, 0, 1, 1]
```

만들어내는 이진수가 8비트로 항상 같은 길이를 갖도록 다음의 *make8* 함수를 써서 이진수를 자르거나 늘여 정확히 8비트가 되도록 알맞게 조절한다.

```
make8      :: [Bit] → [Bit]
make8 bits = take 8 (bits ++ repeat 0)
```

라이브러리 함수 *repeat*는 어떤 값이 무한히 많이 반복되는 리스트를 만드는데, 역시 느긋한 계산법으로 인해 문맥상 필요한 만큼의 원소만 실제로 만들낸다. 예컨대 :

```
> make8 [1, 0, 1, 1]
[1, 0, 1, 1, 0, 0, 0, 0]
```

전송 transmission

이제 글줄에서 각각의 글자를 유니코드 Unicode 번호로 바꾸고, 그 번호를 8비트 이진수로 바꾼 뒤, 그것들을 모두 이어붙여 하나의 비트 리스트를 만들어 내는 방식으로, 글줄을 비트 리스트로 부호화 encode 하는 함수를 정의할 수 있다. 함수를 주고받는 함수인 *map*과 함수 합성 연산자를 써서 그러한 변환을 다음과 같이 구현할 수 있다.

```
encode :: String → [Bit]
encode = concat ∘ map (make8 ∘ int2bin ∘ ord)
```

예컨대,

```
> encode "abc"
[1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
```

*encode*로 부호화된 비트 리스트를 복호화^{decode}하기 위해, 우선 그런 비트 리스트를 8비트씩 이진수로 잘라내는 *chop8* 함수를 다음과 같이 정의한다.

```
chop8      :: [Bit] → [[Bit]]
chop8 []   = []
chop8 bits = take 8 bits : chop8 (drop 8 bits)
```

이제 비트 리스트를 글자로 복호화하는 함수를, 리스트를 8비트씩 잘라내고 그 잘라낸 각 조각에 해당하는 이진수를 유니코드 번호로 바꾸고, 또 그 번호를 글자로 바꾸는 방식으로, 다음과 같이 쉽게 정의할 수 있다.

```
decode :: [Bit] → String
decode = map (chr ∘ bin2int) ∘ chop8
```

예컨대,

```
> decode [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
"abc"
```

옮긴이 주: 앞의 *isDigit*과 마찬가지로 *ord*와 *chr*도 *Char* 모듈에 있는 함수이므로 스크립트 앞부분에 *import Char*로 *Char* 모듈을 불러와야 한다.

마지막으로, 항등함수 identity function 로 본뜬 (정보 손실이 없는) 완전한 통신로 communication channel 를 이용해 글줄을 비트로 전송하는 것을 시늉내는 *transmit* 함수를 다음과 같이 정의할 수 있다.

```
transmit :: String → String
transmit = decode ∘ channel ∘ encode
channel :: [Bit] → [Bit]
channel = id
```

예컨대,

```
> transmit "higher-order functions are easy"
"higher-order functions are easy"
```

따라하기

글줄 전송기를 정의한 *transmit.lhs* 를 불러들여 실습한 화면이다.

The screenshot shows the WinHugs Haskell interpreter window. The title bar says "WinHugs". The menu bar includes "File", "Edit", "Actions", "Browse", and "Help". The toolbar has icons for file operations like Open, Save, and Print. The main text area displays the following:

```
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-2005
World Wide Web: http://haskell.org/hugs
Bugs: http://hackage.haskell.org/trac/hugs
Version: Sep 2006

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type ?: for help
Main> :browse
module Main where
bin2int :: [Bit] -> Int
int2bin :: Int -> [Bit]
make8 :: [Bit] -> [Bit]
encode :: String -> [Bit]
chop8 :: [Bit] -> [[Bit]]
decode :: [Bit] -> String
transmit :: String -> String
channel :: [Bit] -> [Bit]
Main> transmit "higher-order functions are easy"
"higher-order functions are easy"
Main>
```

transmit 뿐 아니라 이 앞서 *transmit* 을 정의하는 데 쓰인 다른 함수들도 *transmit.lhs* 를 불러들여 실습할 수 있다.

:browse 명령으로 현재 불러들인 스크립트에 어떤 함수들이 정의되어 있는지 알아볼 수 있음을 눈여겨 보라.

따라하기 끝

스크립트 *transmit.lhs* 한눈에 다시보기

```

1 Programming in Haskell 7.6절 글줄 전송 기 보기,
2 Graham Hutton, Cambridge University Press, 2007.
3
4 > import Char
5
6 진법 변환
7 -----
8
9 > type Bit          = Int
10 >
11 > bin2int           :: [Bit] -> Int
12 > bin2int           = foldr (\x y -> x + 2*y) 0
13 >
14 > int2bin           :: Int -> [Bit]
15 > int2bin 0          = []
16 > int2bin n          = n `mod` 2 : int2bin (n `div` 2)
17
18 전송
19 -----
20
21 > make8              :: [Bit] -> [Bit]
22 > make8 bits         = take 8 (bits ++ repeat 0)
23 >
24 > encode              :: String -> [Bit]
25 > encode              = concat . map (make8 . int2bin . ord)
26 >
27 > chop8               :: [Bit] -> [[Bit]]
28 > chop8 []            = []
29 > chop8 bits          = take 8 bits : chop8 (drop 8 bits)
30 >
31 > decode              :: [Bit] -> String
32 > decode              = map (chr . bin2int) . chop8
33 >
34 > transmit             :: String -> String
35 > transmit             = decode . channel . encode
36 >
37 > channel              :: [Bit] -> [Bit]
38 > channel              = id
39

```

*transmit.lhs*에서 눈여겨 볼 점은 다음과 같다.

1. *transmit.lhs*는 주석이 기본인 문학적 하스켈 스크립트이다.
2. 줄번호 9에서 **type** 예약어는 새로운 타입을 정의하는 것이 아니라 기존 타입에 새로운 이름 혹은 별명을 붙이는 데 쓰인다. 즉 9째 줄을 지우고 이후의 모든 *Bit*를 *Int*로 바꿔 써도 프로그램은 똑같이 동작한다. 다만 *Bit*로 별명을 붙이는 것이 사람이 읽기 편하다.
3. 책에서는 합성 함수 연산자를 \circ 로 나타내지만 일반 텍스트 편집기나 WinHugs에서는 .로 나타난다. (줄 번호 25의 *encode*, 줄번호 32의 *decode*, 줄번호 35의 *transmit* 정의에 함수 합성 연산자를 쓰고 있다.) 이런 특수 기호 입력에 대해서는 부록 B를 참조하라.
4. *foldr*, *map* 그리고 함수 합성 연산자(\circ) 같은 함수를 주고받는 함수를 이용하여 다른 함수를 간결하게 정의하고 있다. (줄번호 12 *bin2int* 정의에서 *foldr*을, 줄번호 25의 *encode*와 줄번호 32의 *decode* 정의에서 *map*을 쓰고 있다.)

7.7 살펴보기

함수를 주고받는 함수를 컴퓨터 음악computer music, 금융 계약financial contract, 그래픽 이미지graphical image, 하드웨어 기술hardware description, 논리 프로그램logic program, 아름 출력기pretty printer 등을 구성하는 데 한층 더 활용하는 것을 The Fun of Programming (7)에서 찾아볼 수 있다. *foldr*에 대한 더 깊이있는 강좌로는 (14)를 보라.

7.8 연습문제

1. 리스트 조건제시식 $[f x \mid x \leftarrow xs, p x]$ 를 함수를 주고받는 함수인 *map* 과 *filter* 를 써서 나타내 보라.
2. 표준 서막의 정의를 보지 말고, 함수를 주고받는 함수들인 *all*, *any*, *takeWhile*, *dropWhile* 을 정의하라.
3. *map f* 와 *filter p* 함수를 *foldr* 로써 다시 정의하라.
4. *foldl* 을 써서 십진수를 정수로 바꾸는 *dec2int* :: $[Int] \rightarrow Int$ 함수를 정의하라. 예컨대,

```
> dec2int [2,3,4,5]
2345
```

5. 다음 정의가 그릇된 까닭을 설명하라.

```
sumsqreven = compose [sum, map (\x) (\y) x * y^2, filter even]
```

6. 표준 서막을 보지 말고 순서쌍을 인자로 받는 함수를 커리된 함수로 바꾸는 함수를 주고받는 라이브러리 함수 *curry* 와, 반대로 두 개의 인자를 받는 커리된 함수를 순서쌍을 인자로 받는 함수로 바꾸는 *uncurry* 함수를 정의하라.

귀띔: 일단 두 함수의 태입부터 적어 보라.

7. 리스트를 만들어내는 간단한 되돌기 형태를 요약하는, 함수를 주고받는 함수인 *unfold*를 다음과 같이 정의할 수 있다.

$$\begin{aligned} \textit{unfold } p \ h \ t \ x \mid p \ x &= [] \\ \mid \text{otherwise} &= h \ x : \textit{unfold } p \ h \ t \ (t \ x) \end{aligned}$$

이는 *unfold p h t* 가 조건 *p*를 만족할 경우에는 빈 리스트를 만들어내며, 그렇지 않을 경우에는 비어있지 않은 리스트를 만들어 내는데, 그 첫 원소는 *h*를 적용함으로써 얻고, 그 나머지 리스트는 또 다른 함수 인자인 *t*를 적용하여 마찬가지 방법으로 되돌며 만들어 낸다는 것을 뜻한다. 예컨대, *int2bin* 함수를 *unfold*를 써서 다음과 같이 더 간결하게 다시 정의할 수 있다.

$$\textit{int2bin} = \textit{unfold} \ ((== 0)) \ ('mod\cdot 2) \ ('div\cdot 2)$$

chop8, *map f*, *iterate f* 와 같은 함수들을 *unfold*를 써서 다시 정의하라.

8. 홀짝 비트 parity bit 를 이용해 간단한 전송 잘못을 알아낼 수 있도록 글줄 전송기 프로그램을 고쳐 보라. 즉, 부호화하면서 각 8비트 이진수마다 하나의 홀짝 비트를 덧붙이되, 그 값은 1이 홀수 개일 때는 1로 그렇지 않을 경우는 0으로 하라. 그 다음, 복호화하면서 9비트 이진수를 읽을 때마다 홀짝 비트가 맞는지 검사하여 맞으면 홀짝 비트를 버리고 틀리면 홀짝 틀림 parity error 을 내도록 하라.

귀띔: 라이브러리 함수 *error::String → a* 는 계산을 그만두고 인자로 주어진 글줄을 잘못 글귀 error message 로 보여준다.

9. 위 연습문제에서 새로 만든 글줄 전송기 프로그램을 첫 비트를 항상 읽는 불안한 통신로에서 시험해 보라. 이러한 통신로는 비트 리스트에 *tail* 함수를 적용함으로써 본뜰 수 있다.

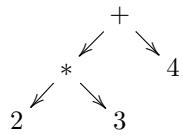
제 8 장

함수형 문법 분석기 functional parser

이 장에서는 하스켈로 간단한 문법 분석기parser 프로그램을 만드는 방법에 대해 알아보겠다. 먼저 문법 분석기란 무엇이며 어떤 때 쓸모있는지 설명하는 것으로 시작하여, 문법 분석기를 함수로서 이해하는 것이 자연스럽다는 것을 살펴보고, 몇 개의 기본 문법 분석기 및 문법 분석기들을 여러 가지 방법으로 엮어내는 함수를 주고받는 함수를 정의한 다음, 그를 바탕으로 산술식 문법 분석기를 만들어 보는 것으로 끝맺는다.

8.1 문법 분석기parser

문법 분석기란 여러 글자로 이루어진 글줄을 받아, 그 문법 구조를 명확히 드러내는 모양의 나무를 만들어 내는 프로그램이다. 예컨대, 산술식 문법 분석기는 $2 * 3 + 4$ 라는 글줄로부터 숫자는 이파리에 그리고 연산자는 마디에 나타나는 다음과 같은 나무를 만들어 내야 한다.



이 나무의 구조를 보면 + 와 * 가 두 개의 인자를 받는 연산자이며, + 보다 * 가 더 우선순위가 높다는 것을 분명히 알 수 있다.

문법 분석기는 컴퓨터 과학에서 중요한 주제다. 왜냐하면 실생활에서 쓰는 대부분의 프로그램이 문법 분석기를 써서 그 입력을 전처리^{pre-process}하기 때문이다. 예컨대, 계산기 프로그램은 계산 전에 수식 문법을 분석하고, Hugs 시스템은 실행 전에 하스켈 문법을 분석하며, 웹 브라우저^{web browser}는 내용을 화면에 보여주기 전에 하이퍼텍스트 문서^{hypertext document}의 문법을 분석한다. 각각의 경우에 있어, 입력의 구조를 명확히 드러나게 하면 그 다음 과정이 훨씬 간단해진다. 예컨대, 위 보기와 같이 일단 수식 문법을 분석하여 나무 구조로 바꾸고 나면 식을 계산하는 것은 쉽고 간단하다.

8.2 문법 분석기 탑

하스켈에서 문법 분석기란 곧 글줄을 받아 나무를 만들어 내는 함수라고 보는 것이 자연스럽다. 따라서, 나무를 나타내는 *Tree*라는 적당한 탑입이 있다고 하면, 문법 분석기라는 개념을 탑입이 *String* \rightarrow *Tree*인 함수로서 나타낼 수 있고, 이를 다음과 같이 선언함으로써 *Parser*로 줄여 쓸 수 있다.

```
type Parser = String → Tree
```

하지만, 일반적으로 문법 분석기가 인자로 넘겨받은 글줄을 끝까지 다 처리하지는 못하는 경우도 있다. 예컨대, 숫자를 처리하는 문법 분석기를 숫자 다음 낱말이 오는 글줄에 적용하는 경우에 그러하다. 따라서, 문법 분석기가 인자로 넘겨받은 글줄 중에서 아직 처리하지 못한 나머지 부분을 같이 돌려주도록 문법 분석기의 탑입을 다음과 같이 일반화할 수 있다.

```
type Parser = String → (Tree, String)
```

비슷한 이유로, 문법 분석기가 항상 성공하는 것만은 아니다. 예컨대, 숫자를 처리하는 문법 분석기를 낱말로 이루어진 글줄에 적용하는 경우가 그러하다. 이를 다루기 위해, 문법 분석기의 타입이 다음과 같이 결과의 리스트를 돌려주도록 하며, 이 때 관례상 빈 리스트는 실패를 나타내고 한원소 리스트는 성공을 나타낸다.

```
type Parser = String → [(Tree, String)]
```

리스트를 돌려주게 되면, 넘겨받은 글줄을 여러가지로 문법 분석이 가능할 때 하나 이상의 결과를 돌려줄 수 있는 가능성도 열어 놓을 수 있다. 하지만, 간단한 설명을 위해 이 책에서는 문법 분석기가 많아야 하나의 결과만 돌려주는 것으로 간주한다.

마지막으로, 여러 가지 서로 다른 문법 분석기들은 서로 다른 종류의 나무를 돌려주는 경우가 많고, 또 더 일반적으로는 아무 종류의 값이나 돌려 줄 수도 있다. 예컨대, 숫자를 처리하는 문법 분석기가 어떤 정수값을 돌려 주게 할 수도 있다. 따라서, 어떤 결과값을 어떤 특정한 Tree 타입으로 하기보다는 다음과 같이 Parser 타입의 인자로 요약하는 것이 더 쓸모있을 것이다.

```
type Parser a = String → [(a, String)]
```

요컨대, 위 선언은 문법 분석기 타입이란 들어오는 글줄을 받아 리스트를 만들어 내는 함수이며, 그 만들어진 리스트에는 문법을 분석한 결과값과 내보내는 글줄의 순서쌍이 원소로 나타난다.

좀 색다르게는, 문법 분석기 타입을 다음과 같이 세우스 박사^{Dr. Seuss} 식의 운문으로서 낭송해 볼 수도 있겠다!

*A parser for things
Is a function from strings
To list of pairs
Of things and strings*

옮긴이 주: 옮긴이들은 운문을 옮기는 문학적 재주까지는 없으므로 이것은 그대로 영문으로 두겠다. 참고로, 세우스 박사(필명)는 어린이들을 위한 동화를 쓴 미국 작가로 특히 컴퓨터 과학자들을 비롯한 이공학도들이 좋아한다. 지적 욕구가 결여된 사람들의 뻔한 수다거리에는 그다지 관심이 없고 지적으로 흥미롭고 인류 발전에 진정 유익한 과학, 기술, 역사, 철학 등의 주제에 주로 집중하며, 결코 죄대없이 또래들이나 주변 사람들이 따르는 것에 휩쓸려 다니지 않고 뚜렷한 주관을 가지고 자신과 관심사가 비슷한 성향의 사람들과 주로 어울리려 하는 사람들을 일컫는 ‘nerd’라는 영어 낱말이 이 사람의 작품에서 유래했다고 한다. 최근에는 세우스 박사의 “Horton Hears a Who”(2008) 가 애니메이션으로 영화화되었다.

8.3 기본 문법 분석기

이제 다른 모든 문법 분석기를 정의하는 구성 요소 building block로 쓰일 세 가지 기본 문법 분석기를 정의하겠다. 역자 주: 모나드와 클래스 인스턴스 선언에 익숙하지 않은 독자는 문법 분석기 실습을 위해 책 홈페이지에서 소스코드를 내려받을 것을 권한다. 이 장에서는 모나드와 클래스 인스턴스 선언에 대한 배경지식 없이도 문법분석기의 핵심 개념을 이해하고 실습해 보는 것이 목적이므로 책에서는 개념을 중심으로 간단히 설명한다. 그래서 이 장의 뒷부분 살펴보기에도 언급하듯 실제 문법분석기 라이브러리 소스코드와 책의 정의가 약간 다른 것이 있으므로 책의 정의만 스크립트에 그대로 따라 쳐넣어서는 보기가 동작하지 않는다.

우선 첫째로, 문법 분석기 $return v$ 는 들어오는 글자를 하나도 처리하지 않고 다음과 같이 결과값 v 로 언제나 성공하기만 한다.

```
return :: a → Parser a
return v = λinp → [(v, inp)]
```

위 함수를 $return v \quad inp = [(v, inp)]$ 와 같이 정의해도 같은 뜻이다. 하지만, 위 정의와 같이 둘째 인자 inp 를 람다식을 써서 몸체 안으로 돌리는 것이, 그 타입 $a \rightarrow Parser a$ 에서 볼 수 있듯, 하나의 인자를 받아 문법 분석기를 돌려주는 함수임을 드러나 보이게 한다는 점에서는 더 좋다.

return v 가 항상 성공하는 반면, 그와 맞짝을 이루는 문법 분석기 *failure* 는 다음과 같이 그 들어오는 글줄의 내용에 관계없이 무조건 실패한다.

$$\begin{aligned} \textit{failure} &:: \textit{Parser } a \\ \textit{failure} &= \lambda \textit{inp} \rightarrow [] \end{aligned}$$

마지막으로 살펴 볼 기본 문법 분석기는 *item* 으로서, 빈 글줄이 들어오는 경우에는 실패하고, 그 밖의 다른 경우에는 첫 번째 글자의 값을 돌려주며 성공하도록 다음과 같이 정의할 수 있다.

$$\begin{aligned} \textit{item} &:: \textit{Parser Char} \\ \textit{item} &= \lambda \textit{inp} \rightarrow \textbf{case } \textit{inp} \textbf{ of} \\ &\quad [] \rightarrow [] \\ &\quad (x : xs) \rightarrow [(x, xs)] \end{aligned}$$

위와 같이 하스켈의 **case** 식을 써서 정의 몸체 안에서 패턴 매칭을 할 수 있으며, 위 보기에서는 두 개의 가능한 결과들 중에 적당한 것을 고르기 위해 글줄 *inp* 를 두 가지 패턴에 맞춰 보고 있다. 비록 이 책에서 많이 사용하지는 않지만, **case** 식은 때때로 쓸모가 있다.

문법 분석기를 함수로 나타냈으므로 보통의 함수 적용으로 글줄에 적용할 수도 있지만, 구체적으로 어떤 구조인지 숨기고 단지 문법 분석기라는 것만을 요약할 수 있도록 다음과 같은 함수를 나름대로 정의하겠다.

$$\begin{aligned} \textit{parse} &:: \textit{Parser } a \rightarrow \textit{String} \rightarrow [(a, \textit{String})] \\ \textit{parse } p \textit{ inp} &= p \textit{ inp} \end{aligned}$$

parse 를 써서, 앞서 정의한 세 가지 기본 문법 분석기가 어떻게 동작하는지 아래 몇 가지 보기 통해 알아보는 것으로 이 절을 끝맺겠다.

```
> parse (return 1) "abc"
[(1, "abc")]
```

```
> parse failure "abc"
[]
```

```
> parse item ""
[]
```

```
> parse item "abc"
[('a', "bc")]
```

8.4 순서대로 엮기^{sequencing}

문법 분석기를 엮는 가장 간단한 방법은 아마도 하나의 문법 분석기 다음에 또 하나의 문법 분석기를 순서대로 적용하는 것으로, 이 때 첫째 분석기가 처리하고 남아 내보내는 글줄이 둘째 문법 분석기가 읽어들이는 글줄로 넘어간다. 그렇다면, 두 분석기가 내놓는 각각의 결과값은 어떻게 처리해야 할까? 한 가지 방법으로, 문법 분석기를 차례대로 엮는 연산자가 두 문법 분석기의 결과값을 순서쌍으로 묶어내도록 다음과 같은 타입으로 만들 수도 있다.

$$\text{Parser } a \rightarrow \text{Parser } b \rightarrow \text{Parser } (a, b)$$

하지만, 실제로 문법 분석기를 순서대로 엮을 때 앞선 문법 분석기의 결과값을 처리할 수 있는 함수의 형태로 엮는 것이 편리하므로, 다음과 같은 순서대로 엮기 연산자 $\gg=$ (“하고 나서”^{then}라고 읽음)를 쓴다.

$$\begin{aligned} (\gg=) &:: \text{Parser } a \rightarrow (a \rightarrow \text{Parser } b) \rightarrow \text{Parser } b \\ p \gg= f &= \lambda \text{inp} \rightarrow \text{case } \text{parse } p \text{ inp } \text{ of} \\ &\quad [] \qquad \rightarrow [] \\ &\quad [(v, \text{out})] \rightarrow \text{parse } (f v) \text{ out} \end{aligned}$$

이는, 문법 분석기 p 를 입력 글줄 inp 에 적용했을 때 실패하면 전체 문법 분석기 $p \gg= f$ 도 실패하며, 그렇지 않고 p 가 성공했다면 그 결과값에 함수 f 를 적용하여 얻은 새로운 문법 분석기를 얻고, 이렇게 얻은 둘째 문법 분석기를 첫째 문법기가 분석하고 남은 출력 글줄 out 에 적용함으로써 최종 결과를 얻음을 나타낸다. 이런 방식으로, 첫째 문법 분석기가 내는 결과값을 둘째 문법 분석기가 직접 받아 처리할 수 있게 된다.

$\gg=$ 로 엮어 만든 문법 분석기는 대개 다음과 같은 구조로 되어 있다.

$$\begin{aligned} p1 &\gg= \lambda v1 \rightarrow \\ p2 &\gg= \lambda v2 \rightarrow \\ &\dots \\ pn &\gg= \lambda vn \rightarrow \\ &return (f v1 v2 \dots vn) \end{aligned}$$

즉, 문법 분석기 $p1$ 을 적용한 결과값을 $v1$, 문법 분석기 $p2$ 를 적용한 결과값을 $v2, \dots$, 그리고 문법 분석기 pn 을 적용한 결과값을 vn 라고 하면, 이 결과값들에 함수 f 를 적용해 얻는 하나의 결과값을 돌려준다는 것을 나타낸다. 하스켈에서는 이런 문법 분석기를 좀더 편리하게 작성할 수 있도록 돋는 아래와 같은 특별한 문법이 있다.

$$\begin{aligned} \mathbf{do} \quad &v1 \leftarrow p1 \\ &v2 \leftarrow p2 \\ &\dots \\ &vn \leftarrow pn \\ &return (f v1 v2 \dots vn) \end{aligned}$$

리스트 조건제시식에서와 마찬가지로, $vi \leftarrow pi$ 와 같은 식을 생성원 generator이라 부른다. 생성원 $vi \leftarrow pi$ 로부터 나오는 결과값이 필요치 않다면, 생성원을 그냥 pi 로 줄여 써도 된다. 문법 분석기를 순서대로 엮는 **do** 표현을 쓸 때도 들여쓰기 규칙을 따라야 한다는 것에 유의하라. 즉, 순서에 따라 차례대로 나타나는 각각의 문법 분석기는 반드시 같은 열 column에서 시작해야 한다.

예컨대, 글자 셋을 읽어들여, 둘째 글자는 버리고, 첫째와 셋째 글자로 이루어진 순서쌍을 돌려주는 문법 분석기를 다음과 같이 정의할 수 있다.

```
p :: Parser (Char, Char)
p = do x ← item
       item
       y ← item
       return (x, y)
```

p 는 정의에 순서대로 나타나는 모든 문법 분석기가 차례대로 성공할 때만 성공하므로, 이 문법 분석기를 만족하는 글줄은 최소한 세 글자 이상이어야만 한다.

```
> parse p "abcdef"
[((‘a’, ‘c’), "def")]
```

```
> parse p "ab"
[]
```

8.5 선택 choice

두 문법 분석기를 엮는 또 하나의 자연스러운 방법은, 첫째 문법 분석기를 입력 글줄에 적용해 봄에 실패하면 그 대신 둘째 문법 분석기를 입력 글줄에 적용하는 것이다. 그런 선택 연산자 $+++$ (“하거나 아니면”or else이라고 읽음)를 다음과 같이 정의할 수 있다.

```
(++) :: Parser a → Parser a → Parser a
p +++ q = λinp → case parse p inp of
                    [] → parse q inp
                    [(v, out)] → [(v, out)]
```

예컨대,

```
> parse (item +++ return ‘d’) "abc"
[(‘a’, "bc")]
```

```
> parse (failure +++ return ‘d’) "abc"
[(‘d’, "abc")]
```

```
> parse (failure +++ failure) "abc"
[]
```

8.6 간단한 문법 분석기 이끌어내기

세 가지 기본 문법 분석기와 순서대로 엮기 및 선택 연산자를 함께 쓰면, 여러 가지 쓸모있는 간단한 문법 분석기를 만들 수 있다.

우선 첫째로, 조건 p 를 만족하는 글자 하나를 처리하는 문법 분석기를 다음과 같이 정의할 수 있다.

```
sat   :: (Char → Bool) → Parser Char
sat p = do x ← item
      if p x then return x else failure
```

sat 함수와 함께 표준 라이브러리 *Char* 모듈에서 알맞은 조건 함수 *predicate* 을 찾아 쓰면, 숫자 하나, 대문자, 소문자, 알파벳, 알파벳이나 숫자 및 주어진 특정 글자를 나타내는 문법 분석기를 다음과 같이 정의할 수 있다.

```
digit  :: Parser Char
digit = sat isDigit
lower  :: Parser Char
lower = sat isLower
upper  :: Parser Char
upper = sat isUpper
```

```
letter    :: Parser Char
letter   = sat isAlpha
alphanum :: Parser Char
alphanum = sat isAlhpaNum
char      :: Char → Parser Char
char x   = sat (== x)
```

☞ **Warning:** 주: 7장에서처럼 스크립트에서 **import Char** 로 모듈을 불러와야 한다.

예컨대,

```
> parse digit "123"
[('1', "23")]

> parse digit "abc"
[]

> parse (char 'a') "abc"
[('a', "bc")]

> parse (char 'a') "123"
[]
```

다음으로는, 주어진 글줄 *xs* 와 일치하는 글줄을 처리하여 그 결과값으로 그 글줄 자체를 돌려주는 문법 분석기 *string xs* 를 다음과 같이 *char* 를 써서 정의할 수 있다.

```
string      :: String → Parser String
string []    = return []
string (x : xs) = do char x
                      string xs
                      return (x : xs)
```

string 을 되돌기로 정의했으며, 전체 입력 글줄을 다 처리해야만 *string* 이 성공한다는 것에 주목하라. 밑바탕 경우인 빈 글줄은 언제나 문법 분석에 성공한다. 되도는 경우인 비어 있지 않은 글줄의 경우에는, 첫째 글자를 문법 분석하고 나서 나머지 글자들을 분석한 다음 전체 글줄을 그 결과값으로 돌려주면 된다. 예컨대,

```
> parse (string "abc") "abcdef"
[("abc", "def")]

> parse (string "abc") "ab1234"
[]
```

따라하기

다음은 *Parsing.lhs*를 WinHugs에서 불러와 간단한 문법 분석기인 *digit*, *char*, *string*로 실습해 본 화면이다.

```
WinHugs
File Edit Actions Browse Help
File | Open | Save | Recent | < | > | Stop | Run | Help | Exit

Type ?: for help
Parsing> :type parse
parse :: Parser a -> String -> [(a,String)]
Parsing> :type digit
digit :: Parser Char
Parsing> :type parse digit
parse digit :: String -> [(Char,String)]
Parsing> parse digit "123"
[('1','23')]
Parsing> parse digit "abc"
[]
Parsing> :type char
char :: Char -> Parser Char
Parsing> :type parse (char 'a')
parse (char 'a') :: String -> [(Char,String)]
Parsing> parse (char 'a') "abc"
[('a','bc')]
Parsing> parse (char 'a') "123"
[]
Parsing> :type string
string :: String -> Parser String
Parsing> :type parse (string "abc")
parse (string "abc") :: String -> [[(Char),String]]
Parsing> parse (string "abc") "abcdef"
[("abc", "def")]
Parsing> parse (string "abc") "ab1234"
[]
Parsing>
```

*Parsing.lhs*는 함수형 문법분석기 라이브러리로 이 책에서 다루는 여러 문법 분석기를 정의하고 있다. 여태까지 따라하기에서 스크립트를 불러왔을 때와는 달리 (마치 표준 라이브러리 *Char* 모듈을 *:load* 명령으로 불러왔을 때처럼) 길잡이 앞에 모듈 이름이 *Main*이 아닌 *Parsing*으로 나타나는 것을 눈여겨 보라. 곧 한눈에 다시보기로 함수형 문법분석기 라이브러리를 살펴보면 왜 이렇게 되는지 더 잘 이해할 수 있을 것이다.

따라하기 끝

다음으로 정의할 두 문법 분석기는 *many p* 와 *many1 p* 로서, 문법 분석기 *p*를 실패해서 더 이상 적용할 수 없을 때까지 여러 번 적용하여, 성공적으로 적용하여 얻은 각각의 결과값을 리스트로 묶어 돌려준다. 반복을 나타내는 이 두 가지 간단한 문법 분석기의 차이점은, *many* 는 한 번도 성공하지 않아도 되지만 *many1* 은 최소한 한 번은 성공해야 한다는 점이다. 이 두 문법 분석기의 정의는 다음과 같다.

```
many    :: Parser a → Parser [a]
many p  = many1 p +++ return []
many1   :: Parser a → Parser [a]
many1 p = do v ← p
             vs ← many p
             return (v : vs)
```

many 와 *many1* 을 서로 되돌며 정의하고 있음에 주목하라. 더 자세히 말하자면, *many p* 의 정의는 *p* 를 한번 이상 적용하거나 아니면 아예 한번도 적용하지 않아도 됨을 나타내며, *many1 p* 의 정의는 *p* 을 한번 적용한 다음에는 0 번 이상만 적용하면 됨을 나타낸다. 예컨대,

```
> parse (many digit) "123abc"
[("123", "abc")]

> parse (many digit) "abcdef"
[("", "abcdef")]

> parse (many1 digit) "abcdef"
[]
```

many 와 *many1* 을 써서, 소문자 다음에 알파벳 및 숫자가 0 개 이상의 글자가 뒤따르는 식별자 identifier(변수 이름), 하나 이상의 숫자로 이루어진 자연수, 그리고 빈칸, 탭 및 줄바꿈 글자가 0 개 이상 모인 공백을 나타내는 문법 분석기를 각각 다음과 같이 정의할 수 있다.

```

ident :: Parser String
ident = do x ← lower
          xs ← many alphanum
          return (x : xs)

nat   :: Parser Int
nat   = do xs ← many1 digit
          return (read xs)

space :: Parser ()
space = do many (sat isSpace)
           return ()

```

예컨대,

```

> parse ident "abc def"
[("abc", " def")]

> parse nat "123 abc"
[(123, " abc")]

> parse space "    abc"
[(((), "abc"))]

```

space 가 처리하는 글자들이 무엇이었는지 자세한 내용은 보통 중요하지 않기에, 의미없는 결과값을 나타내는 빈 순서쌍 () 을 돌려준다.

따라하기

다음은 *Parsing.lhs*를 WinHugs에서 불러와 *many* 와 *many1* 을 써서 만든 간단한 문법 분석기를 실습해 본 화면이다.

```

WinHugs
File Edit Actions Browse Help
Open Save Print Run Stop Help
Parsing> :type parse
parse :: Parser a -> String -> [(a,String)]
Parsing> :type many
many :: Parser a -> Parser [a]
Parsing> :type many1
many1 :: Parser a -> Parser [a]
Parsing> :type parse (many digit)
parse (many digit) :: String -> [[[Char],String]]
Parsing> :type parse (many1 digit)
parse (many1 digit) :: String -> [[[Char],String]]
Parsing> parse (many digit) "123abc"
[("123","abc")]
Parsing> parse (many digit) "abcdef"
[("", "abcdef")]
Parsing> parse (many1 digit) "abcdef"
[]
Parsing> parse ident "abc def"
[("abc", " def")]
Parsing> parse nat "123 abc"
[(123, " abc")]
Parsing> parse space "    abc"
[("", "abc")]
Parsing>

```

*Parsing.lhs*는 함수형 문법분석기 라이브러리로 이 책에서 다루는 여러 문법 분석기를 정의하고 있다. 여태까지 따라하기에서 스크립트를 불러왔을 때와는 달리 (마치 표준 라이브러리 *Char* 모듈을 *:load* 명령으로 불러왔을 때처럼) 길잡이 앞에 모듈 이름이 *Main* 이 아닌 *Parsing* 으로 나타나는 것을 눈여겨 보라. 곧 한눈에 다시보기로 함수형 문법분석기 라이브러리를 살펴보면 왜 이렇게 되는지 더 잘 이해할 수 있을 것이다.

따라하기 끝

8.7 빈칸 처리

실생활에서 쓰는 대부분의 문법 분석기는 입력 글줄에 나타나는 기본 토큰^{token} 주위에 공백이 얼마든지 와도 괜찮다. 예컨대, 글줄 “1+2” 와 “1 + 2” 는 Hugs에서 똑같은 것으로 문법 분석된다. 이런 식으로 공백을 처리하기 위해, 어떤 토큰을 처리하는 문법 분석기를 받아 그 토큰 앞뒤로 오는 공백 글자를 무시하는 간단한 문법 분석기를 아래와 같이 새로 정의할 수 있다.

```

token :: Parser a → Parser a
token p = do space
            v ← p
            space
            return v

```

이제, 식별자^{identifier}, 자연수, 특수글자 주위의 공백을 무시하는 문법 분석기를 다음과 같이 *token* 을 써서 간단히 정의할 수 있다.

```

identifier :: Parser String
identifier = token ident

```

```

natural :: Parser String
natural = token nat
symbol :: String → Parser String
symbol xs = token (string xs)

```

예컨대, 토큰 주위의 공백을 무시하며 비어 있지 않은 자연수 리스트를 처리하는 문법 분석기를 다음과 같이 정의할 수 있다.

```

p :: Parser [Int]
p = do symbol "[" 
       n ← natural
       ns ← many (do symbol ","
                     natural)
       symbol "]"
       return (n : ns)

```

이 정의가 나타내는 바는, 이 문법 분석기가 처리할 수 있는 리스트는 대괄호를 여는 것으로 시작하여 하나의 자연수가 오고, 그 뒤로 쉼표와 자연수가 연달아 0 번 이상 나타날 수 있으며, 마지막으로 대괄호를 닫음으로써 끝난다. *p* 는 정확히 이러한 형태를 따르는 완전한 리스트를 나타내는 글줄을 받았을 때만 성공한다.

```

> parse p "[1, 2, 3] "
[[1, 2, 3], """]

> parse p "[1,2,]"
[]

```

스크립트 *Parsing.lhs* 한눈에 다시보기

```
1 Programming in Haskell 8장 합수형 문법분석기 랙이브 허튼,  
2 Cambridge University Press, 2007.  
3  
4  
5 > module Parsing where  
6 >  
7 > import Char  
8 > import Monad  
9 >  
10 > infixr 5 +++  
11  
12 문법분석기 모나드  
13 -----  
14  
15 > newtype Parser a = P (String -> [(a,String)])  
16 >  
17 > instance Monad Parser where  
18 >   return v      = P (\inp -> [(v,inp)])  
19 >   p >>= f     = P (\inp -> case parse p inp of  
20 >                   []          -> []  
21 >                   [(v,out)] -> parse (f v) out)  
22 >  
23 > instance MonadPlus Parser where  
24 >   mzero        = P (\inp -> [])  
25 >   p `mplus` q   = P (\inp -> case parse p inp of  
26 >                   []          -> parse q inp  
27 >                   [(v,out)] -> [(v,out)])  
28
```

```

29  기본 문법분석기
30 -----
31
32 > failure      :: Parser a
33 > failure      = mzero
34 >
35 > item          :: Parser Char
36 > item          = P (\inp -> case inp of
37 >                   []      -> []
38 >                   (x:xs) -> [(x,xs)])
39 >
40 > parse         :: Parser a -> String -> [(a,String)]
41 > parse (P p) inp = p inp
42
43 선택
44 -----
45
46 > (+++)        :: Parser a -> Parser a -> Parser a
47 > p +++ q      = p `mplus` q
48
49 간단한 문법분석기 이끌어내기
50 -----
51
52 > sat           :: (Char -> Bool) -> Parser Char
53 > sat p         = do x <- item
54 >                  if p x then return x else failure
55 >
56 > digit         :: Parser Char
57 > digit         = sat isDigit
58 >
59 > lower          :: Parser Char
60 > lower          = sat isLower
61 >
62 > upper          :: Parser Char
63 > upper          = sat isUpper
64 >
65 > letter         :: Parser Char
66 > letter         = sat isAlpha
67 >
68 > alphanum       :: Parser Char
69 > alphanum       = sat isAlphaNum
70 >
71 > char            :: Char -> Parser Char
72 > char x         = sat (== x)
73 >
74 > string          :: String -> Parser String
75 > string []       = return []
76 > string (x:xs)   = do char x
77 >                      string xs
78 >                      return (x:xs)
79 >
80 > many            :: Parser a -> Parser [a]
81 > many p          = many1 p +++ return []
82 >
83 > many1           :: Parser a -> Parser [a]
84 > many1 p          = do v <- p
85 >                      vs <- many p
86 >                      return (v:vs)
87 >

```

```
88 > ident          :: Parser String
89 > ident          = do x  <- lower
90 >                  xs <- many alphanum
91 >                  return (x:xs)
92 >
93 > nat            :: Parser Int
94 > nat            = do xs <- many1 digit
95 >                  return (read xs)
96 >
97 > int            :: Parser Int
98 > int            = do char '-'
99 >                  n <- nat
100 >                 return (-n)
101 >                 +++ nat
102 >
103 > space          :: Parser ()
104 > space          = do many (sat isSpace)
105 >                  return ()
```

```
107  빈칸 처리
108  -----
109
110 > token          :: Parser a -> Parser a
111 > token p        =  do space
112 >                  v <- p
113 >                  space
114 >                  return v
115 >
116 > identifier      :: Parser String
117 > identifier      =  token ident
118 >
119 > natural         :: Parser Int
120 > natural         =  token nat
121 >
122 > integer          :: Parser Int
123 > integer          =  token int
124 >
125 > symbol           :: String -> Parser String
126 > symbol xs       =  token (string xs)
127
```

이 책에서 설명하지 않은 내용 중 *Parsing.lhs*에서 눈여겨 볼 점은 다음과 같다.

1. 줄번호 5에서 **module Parsing where**라는 표현으로 이 스크립트가 정의하는 모듈 이름을 *Parsing*으로 붙였기 때문에 WinHugs에서 불러들였을 때 길잡이 앞에 모듈 이름이 *Parsing*이라고 나타난 것이다. 우리는 지금까지 모듈 이름을 생략하고 WinHugs에서 스크립트를 불러들였는데, 이는 **module Main where**라고 한 것과 같은 효과다.
2. 줄번호 10에서 **infixr 5 +++** 은 **+++** 연산자가 오른쪽에서부터 묶이며 우선 순위 단계가 5라는 것을 지정 한다.
3. 줄번호 8, 줄번호 15-27은 모나드와 관련된 내용이다. 문법 분석기를 모나드로 정의했기 때문에 **do** 표현을 쓸 수 있는 것이다. 모나드에 대해서는 10.6 절에서 좀더 자세히 다룰 것이다.
4. 줄번호 15에서 **newtype**은 항수가 1인 생성자 하나만을 갖는 타입을 좀더 효율적으로 처리하기 위한 하스켈의 기능이다. 자세한 내용은 하스켈 보고서(25)에서 찾아볼 수 있다.

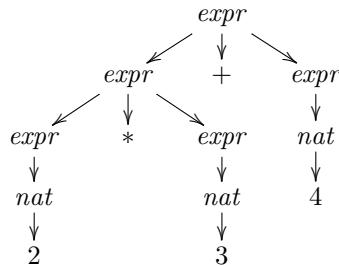
8.8 산술식 문법 분석하기

더 큰 보기장을 만들어 보는 것으로서 이 장을 맺겠다. 자연수 및 자연수를 덧셈, 곱셈, 괄호로 엮어 만들 수 있는 간단한 형태의 산술식을 생각해 보자. 덧셈과 곱셈은 오른쪽으로부터 묶이며 곱셈이 덧셈보다 우선순위가 높다고 가정한다. 예컨대, $2 + 3 + 4$ 는 $2 + (3 + 4)$ 를 뜻하며, $2 * 3 + 4$ 는 $(2 * 3) + 4$ 를 뜻한다.

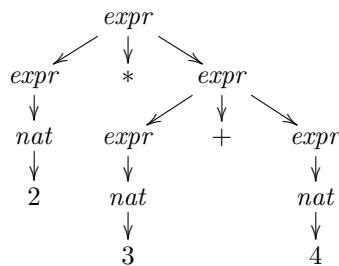
이 언어의 문법 구조를 문법^{grammar}이라는 수학적 개념, 곧 언어를 이루는 글자를 어떻게 만드는지 정의하는 규칙들로 형식화할 수 있다. 예컨대, 이런 산술식을 나타내는 문법을 다음 두 규칙으로 정의할 수 있다.

$$\begin{aligned} \text{expr} &::= \text{expr} + \text{expr} \mid \text{expr} * \text{expr} \mid (\text{expr}) \mid \text{nat} \\ \text{nat} &::= 0 \mid 1 \mid 2 \mid \dots \end{aligned}$$

첫째 규칙은 식이란 두 식의 덧셈식 및 곱셈식, 괄호로 묶인 식, 또는 자연수로 이루어짐을 것을 나타낸다. 그 다음 둘째 규칙은 자연수란 0, 1, 2 등의 수들 중 하나라는 것을 나타낸다. 예컨대, 이 문법으로 만들 수 있는 식 $2 * 3 + 4$ 를 아래와 같은 문법 분석 나무 parse tree 로 나타낼 수 있다. 문법 분석 나무에서는 아래와 같이 식에 나타나는 토큰이 이파리에 나타나며, 식을 만들 때 적용한 문법 규칙이 나무가 가지치는 마디 구조로 나타나게 된다.



이 나무의 구조는 $2 * 3 + 4$ 가 두 식의 덧셈으로 이루어진다는 것을 분명히 드러내는데, 이 때 덧셈식에 나타나는 첫째 식은 수 2와 3이라는 또 다른 두 식의 곱셈식이며, 덧셈식에 나타나는 둘째 식은 수 4이다. 하지만, 이 문법에 따르면 이 보기로부터 $2 * (3 + 4)$ 에 해당하는 다음과 같은 잘못된 문법 분석 나무도 얻을 수 있다.

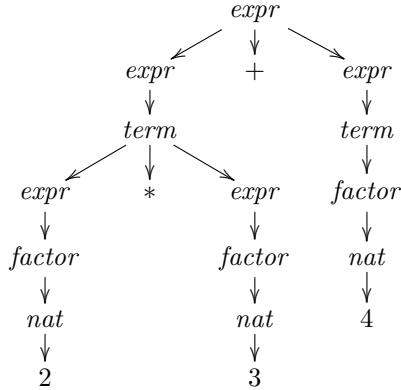


우리가 정의한 문법의 문제점은 바로 곱셈이 덧셈보다 우선순위가 높다 점을 고려하지 않았다는 것이다. 하지만, 각각의 우선 순위마다 별도의 규칙이 있도록 문법을 아래와 같이 고침으로써 이러한 문제점을 쉽게 해결할 수 있다. 이 때, 덧셈이 가장 낮은 우선순위를 가지며, 곱셈이 중간 우선 순위, 괄호와 수는 가장 우선순위가 높다.

```

expr ::= expr + expr | term
term ::= term * term | factor
factor ::= (expr) | nat
nat   ::= 0 | 1 | 2 |...
  
```

이 새로운 문법에 따르면, 식 $2 * 3 + 4$ 는 이제 $(2 * 3) + 4$ 를 나타내는 올바른 문법 분석 나무는 단 하나 뿐이다.

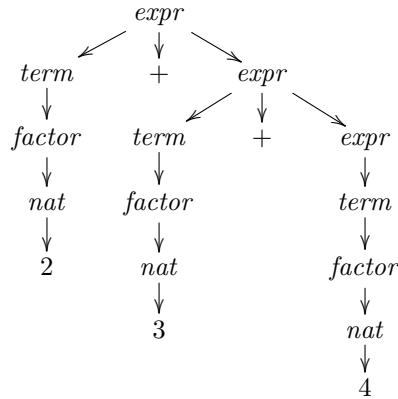


이제 우선 순위에 대한 문제는 처리했지만, 아직 덧셈 및 곱셈이 오른쪽으로부터 묶인다는 것까지는 고려하지 못했다. 예컨대, 식 $2 + 3 + 4$ 가 지금의 문법으로는 각각 $(2 + 3) + 4$ 와 $2 + (3 + 4)$ 에 해당하는 두 가지 문법 분석 나무로 문법 분석이 가능하다. 하지만, 덧셈과 곱셈에 대한 문법 규칙을 양쪽 다가 아닌 오른쪽 인자에 대해서만 되돌도록 다음과 같이 고침으로써 이 문제점 또한 쉽게 해결할 수 있다.

```

expr ::= term + expr | term
term ::= factor * term | factor
  
```

이 새로운 규칙에 따르면, 식 $2 + 3 + 4$ 는 이제 $2 + (3 + 4)$ 에 해당하는 올바른 문법 분석 나무는 단 하나 뿐이다.



우리 산술식 문법은 이제 모든 문법에 맞는 well-formed 식마다 올바른 문법 분석 나무는 단 하나 뿐이라는 점에서 모호성이 없다 unambiguous.

이제 산술식 문법 자체를 마지막으로 한 번만 더 고칠텐데, 이번에는 문법을 간단히 해 보기로 하자. 예컨대, $expr ::= term + expr \mid term$ 라는 규칙을 살펴보면, 식이 항^{term}과 식의 덧셈식이거나 아니면 항으로만 이루어진다는 것을 나타내고 있다. 달리 말하면, 식은 항상 항으로 시작하며, 그 뒤로 덧셈 기호와 다른 식이 올 수도 있고, 아니면 아무것도 오지 않을 수도 있다. 따라서, 식을 나타내는 규칙을 $expr ::= term (+expr \mid \epsilon)$ 로 간단히 할 수 있으며, 이 때 기호 ϵ 기호는 빈 글줄을 나타낸다. 항을 나타내는 식도 마찬가지 방법으로 간단히 하면, 마지막으로 다음과 같은 산술식 문법을 얻는다.

```

expr  ::= term  (+expr | ε)
term  ::= factor (*term | ε)
factor ::= (expr) | nat
nat   ::= 0 | 1 | 2 | ...
  
```

이제 이 문법을 산술식을 처리하는 문법 분석기로 바꿔 쓰는 것은 쉽고 간단하다. 그냥 각각의 규칙을 앞서 만든 간단한 문법 분석기를 이용해 써내려가기만 하면 된다.

여기서는 문법 분석기가 문법 분석 나무를 만들어내는 대신 식을 나타내는 글줄을 처리함과 동시에 그 식의 값까지 정수로 구해 내도록 하였다.

```
expr  :: Parser Int
expr  = do t ← term
          do symbol "+"
              e ← expr
              return (t + e)
          +++ return t

term  :: Parser Int
term  = do f ← factor
          do symbol "*"
              t ← term
              return (f * t)
          +++ return f

factor :: Parser Int
factor = do symbol "("
            e ← expr
            symbol ")"
            return e
        +++ natural
```

예컨대, 문법 분석기 *expr* 은 일단 어떤 항을 문법 분석하여 그 값 *t* 를 구한 다음, 덧셈 기호와 식을 문법 분석하여 그 식의 값인 *e* 를 구해 *t + e* 의 값을 돌려주거나, 아니면 그 뒤로 아무것도 오지 않을 경우를 분석하여 그냥 *t* 값을 그대로 돌려준다. 문법 분석기 *term* 과 *factor* 도 비슷한 방법으로 이해할 수 있다.

마지막으로, *expr* 을 써서 산술식의 정수값을 구하는 *eval*::*String* → *Int* 함수를 아래와 같이 정의할 수 있다. 못다 처리한 입력이나 잘못된 입력이 올 때는 잘못 글귀를 화면에 나타내고 프로그램을 끝내도록 라이브러리 함수 *error*::*String* → *a* 를 쓴다.

```
eval   :: String → Int
eval xs = case parse expr xs of
            [(n, [])] → n
            [(_ , out)] → error ("못 다 처리한 입력" ++ out)
            []           → error "잘못된 입력"
```

옮긴이 주: GHC로 프로그램을 돌린다면 잘못 글귀를 한글 대신 알파벳으로 써야 한다. Hugs는 한글을 지역화 설정(locale)에 맞게 출력해 주지만 GHC에서는 한글을 출력하려면 인코딩을 따로 처리해 주어야 한다.

예컨대,

> eval "2*3+4"

10

> eval "2*(3+4)"

14

> eval "2 * (3 + 4)"

14

> eval "2*3-4"

Error : 못다 처리한 입력 - 4

> eval "-1"

Error : 잘못된 입력

스크립트 *parser.lhs* 한눈에 다시보기

```
1 Programming in Haskell 8.8절 산술식 문법분석기 보기,
2 Graham Hutton, Cambridge University Press, 2007.
3
4
5 간단한 산술식 문법 분석기
6 -----
7
8 > import Parsing
9
10 > expr      :: Parser Int
11 > expr      =  do t <- term
12 >           do symbol "+"
13 >           e <- expr
14 >           return (t+e)
15 >           +++ return t
16 >
17 > term      :: Parser Int
18 > term      =  do f <- factor
19 >           do symbol "*"
20 >           t <- term
21 >           return (f * t)
22 >           +++ return f
23 >
24 > factor    :: Parser Int
25 > factor    =  do symbol "("
26 >           e <- expr
27 >           symbol ")"
28 >           return e
29 >           +++ natural
30 >
31 > eval      :: String -> Int
32 > eval xs   =  case (parse expr xs) of
33 >           [(n,[])]  -> n
34 >           [(_ ,out)] -> error ("unused input " ++ out)
35 >           []        -> error "invalid input"
36
```

이 스크립트에서 눈여겨 볼 점은 다음과 같다.

1. 줄번호 8에서 **import** *Parsing* 으로 문법 분석기 라이브러리 모듈인 *Parsing.lhs*를 불러들이고 있다. 지금까지는 *Char* 모듈과 같은 표준라이브러리 모듈만 불러들였지만, 이와 같이 우리가 새로 정의한 라이브러리도 얼마든지 불러들일 수 있다. 이렇게 *Parsing* 모듈을 불러들여 기본적인 문법 분석기를 쓰기 위해선 *Parsing.lhs*를 이 스크립트 파일(*parser.lhs*)과 같은 디렉토리에 두고 WinHugs에서 *parser.lhs*를 불러오면 된다.
2. 하스켈은 들여쓰기로 단지 프로그램 소스 코드를 아름답고 읽기 쉽게 나타내는 것 뿐 아니라 들여쓰기에 따라 프로그램의 뜻이 달라지므로 주의해야 한다. 줄번호 15나 줄번호 22에서 안쪽 **do** 와 같이 들여쓴 선택 연산자 **+++**를 줄번호 29에서와 같이 앞당겨 쓰면 바깥쪽 선택 연산자가 바깥쪽 **do**에 걸리게 되어 전혀 다른 잘못된 동작을 하게 된다.

8.9 살펴보기

더 복잡한 문법 분석기를 만들어내는 데 쓸 수 있는, 이 장에서 나온 간단한 문법 분석기들을 포함하는 라이브러리를 이 책의 웹사이트에서 얻을 수 있다. 문법 분석기가 모나드라는 속성을 갖기 때문에 생기는 기술적인 이유로, 라이브러리에 있는 기본 정의 몇 개는 이 책에 나타난 것과 조금씩 다르다. 더 자세한 내용은 이 장의 바탕이 되는 (16;17)를 참조하라. 문법에 관한 더 많은 내용을 (27)에서 찾아볼 수 있고, 하스켈로 문법 분석기를 만드는 더 고급스런 기법들을 (22;9)에서 찾아볼 수 있다. 문법 분석기 타입을 운문 형식으로 읽은 것은 Fritz Ruehr(세우스 박사의 본명)를 따라해본 것이다.

8.10 연습문제

1. 라이브러리 파일에서는 정수를 위한 문법 분석기 $int :: Parser\ Int$ 를 정의하고 있다. 이 정의를 보지 않고, int 를 정의하라.

귀띔: 정수는 자연수이거나 자연수 앞에 음의 부호가 붙을 수 있다.

2. -- 기호로 시작하여 그 줄의 끝까지 이어지는 하스켈 한줄 주석을 처리하는 문법 분석기 $comment :: Parser ()$ 를 정의하라. 참고로, 각 줄의 끝에는 제어 글자 '\n' 가 온다.

3. 두 번째 산술식 문법에 따라 식 $2 + 3 + 4$ 가 가질 수 있는 두 가지 문법 분석 나무를 그려 보라.

4. 세 번째 산술식 문법에 따라 식 $2 + 3$, $2 * 3 * 4$, $(2 + 3) + 4$ 의 문법 분석 나무를 그려 보라.

5. 산술식 문법을 마지막으로 간단히 합에 따라 만들어지는 문법 분석기는 그 효율에 큰 영향을 받게 되는데, 어째서 그러한지 설명해 보라.

귀띔: 마지막으로 문법을 간단히 하지 않았더라면 수 하나로만 이루어진 식을 문법 분석할 때 어떻게 될지 생각해 보라.

6. 산술식 문법 분석기를 다음 문법에 따라 뺄셈과 나눗셈을 처리할 수 있도록 확장하라.

$$\begin{aligned}expr &::= term \ (\text{+} expr \mid \text{-} expr \mid \epsilon) \\term &::= factor \ (\text{*} expr \mid \text{/} expr \mid \epsilon)\end{aligned}$$

7. 산술식 문법 분석기의 문법을 확장해 거듭제곱을 처리할 수 있게 하라. 거듭제곱 연산자는 오른쪽으로부터 묶이며, 곱셈과 나눗셈보다 우선순위가 높지만, 괄호와 숫자보다는 우선순위가 낮다. 예컨대, $2 \uparrow 3 * 4$ 는 $(2 \uparrow 3) * 4$ 와 같은 뜻이다.

귀띔: 우선 순위가 한 단계 더 새로 나타났으므로, 이를 표현할 새로운 문법 규칙이 필요하다.

8. 자연수 및 원쪽으로부터 묶이는 랜섬 연산자로 만들 수 있는 식들을 생각해 보라.

- (a) 이러한 식을 나타내는 문법을 있는 그대로 정의해 보라.
- (b) 정의한 문법을 문법 분석기 $expr :: Parser Int$ 로 바꿔 나타내 보라.
- (c) 이렇게 만든 문법 분석기에는 문제가 있다. 그 문제점은 무엇인가?
- (d) 문법 분석기의 문제점을 고쳐 보라.

귀띔: 반복을 나타내는 간단한 문법 분석기 *many* 및 표준 라이브러리 함수 *foldl*을 써서, 문법 분석기를 다시 정의해 보라.

제 9 장

대화식interactive 프로그램

이 장에서는 하스켈로 대화식 프로그램을 어떻게 작성할 수 있는지 알아본다. 먼저 대화식 프로그램이란 무엇인지 설명하고, 어떻게 그런 프로그램을 함수로서 자연스럽게 나타낼 수 있는지 살펴본 다음, 몇 개의 기본적인 대화식 프로그램 및 대화식 프로그램을 엮어내는 함수를 주고받는 함수^{higher order function}를 정의하고, 계산기 및 생명 게임^{game of life}을 만들어 보는 것으로 이 장을 끝맺는다.

9.1 대화interaction

프로그램이 도는 동안 사용자와 대화^{interact} 하지 않는 프로그램을 배치^{batch} 프로그램이라 한다. 컴퓨터를 처음 사용하기 시작하던 시절에는 대부분의 프로그램이 배치 프로그램이었는데, 이는 컴퓨터가 최대한 많은 일을 처리 할 수 있도록 사용자로부터 격리된 상태에서 프로그램을 돌려야 했기 때문이다. 예컨대, 경로 탐색^{route-planning} 프로그램은 출발 지점과 목표 지점을 입력으로 받아 조용히 많은 양의 계산을 한 다음 추천 경로를 출력할 것이다.

이 책에서는 지금까지 하스켈로 배치 프로그램을 어떻게 작성하는지 알아보았다. 하스켈에서는 그런 경로탐색 프로그램뿐 아니라 일반적으로 모든 프로그램을 순수 함수 pure function, 곧 입력을 모두 드러나 보이는 인자로 받고 출력을 모두 드러나 보이는 결과값으로 내는 함수로 나타낸다. 예컨대, 경로 탐색기 route planner 를 두 지점을 순서쌍으로 받아 그 두 지점 사이의 경로를 내는, 타입이 (*Point, Point*) → *Route* 인 함수로 나타낼 수 있다.

반면, 대화식 interactive 프로그램은 프로그램이 도는 동안 사용자로부터 따로 입력을 받을 수도 있으며, 사용자에게 따로 출력을 내줄 수도 있다. 오늘날에는 대부분의 컴퓨터 프로그램이 사용자와 대화함으로써 더 유연하게 동작하는 대화식 프로그램이다. 예컨대, 계산기 프로그램 사용자는 글쇠판으로 수식을 대화식으로 입력할 수 있고, 그 식의 값을 바로 화면에서 볼 수 있다.

언뜻 보기에도, 순수 함수로는 대화식 프로그램을 나타낼 수 있을 것 같지가 않다. 왜냐하면 그런 프로그램은 그 속성상 프로그램이 도는 동안 따로 입력을 받거나 따로 출력을 내는 등의 결따르는 반응 side effect 을 필요로 하기 때문이다. 예컨대, 계산기 프로그램을 대체 어떻게 인자를 받아 결과를 내는 순수 함수로서 나타낼 수 있겠는가?

여러 해 동안 어떻게 결따르는 반응을 순수 함수의 개념과 결합할 것인지의 문제를 놓고 여러 사람들이 많은 방법을 제시했다. 앞으로 이 장에서는 몇 개의 간단한 연산이 있는 새로운 타입을 바탕으로 결따르는 반응을 나타내는, 하스켈이 채택한 방법을 소개하겠다. 앞으로 보게 되겠지만, 이 방법은 바로 앞 장에서 문법 분석기를 만들 때 썼던 접근 방법과 공통점이 많다.

9.2 입출력 타입

하스켈에서는, 대화식 프로그램을 현재 “세계 world 의 상태”를 인자로 받아 바뀐 세계 modified world 를 그 결과값으로 내는 순수 함수로 나타내는데, 이 때 바뀐 세계에서 프로그램이 돌며 일으키는 모든 결따르는 반응을 반영하게 된다. 즉, 현재 세계의 상태를 나타내는 *World* 타입만

알맞게 주어지면, 대화식 프로그램이라는 개념을 타입이 $World \rightarrow World$ 인 함수로 나타낼 수 있으며, 이 타입을 다음과 같이 IO (입출력 input/output 을 나타냄)라 줄여 부를 수 있다.

```
type IO = World → World
```

하지만, 일반적으로 대화식 프로그램은 곁따르는 반응을 일으킬 뿐 아니라 결과값도 돌려줄 수도 있다. 예컨대, 글쇠판으로부터 글자를 읽어들이는 프로그램은 어떤 글자를 읽었는지 돌려주어야 할 것이다. 따라서, 대화식 프로그램을 나타내는 타입이 결과값을 포함하도록 일반화해야 하므로, 아래와 같이 IO 타입이 결과값 타입을 인자로 받도록 하면 된다.

```
type IO a = World → (a, World)
```

IO 타입의 식을 동작action 이라 부른다. 예컨대, $IO Char$ 는 글자를 돌려주는 동작을 나타내는 타입이며, $IO ()$ 는 빈 순서쌍을 의미없는 dummy 결과값으로 돌려주는 타입을 나타내는 동작이다. $IO ()$ 타입의 동작은 사실상 결과값 없이 곁따르는 반응만 있는 동작으로 볼 수 있는데, 대화식 프로그램을 작성할 때 흔히 나타난다.

대화식 프로그램이 결과값을 돌려주는 것 말고도 인자값을 더 받아야 할 수도 있다. 하지만 이를 나타내기 위한 동작 타입을 따로 일반화할 필요는 없는데, 왜냐하면 동작을 커리함으로써 이러한 개념을 이미 나타낼 수 있기 때문이다. 예컨대, 글자를 받아 정수를 돌려주는 대화식 프로그램은 $Char \rightarrow IO Int$ 타입으로 나타낼 수 있으며, 이는 커리된 함수 타입인 $Char \rightarrow World \rightarrow (Int, World)$ 를 줄여 쓴 것일 뿐이다.

여기까지 설명을 듣고 나면, 동작action 을 프로그래밍 한답시고 온 세계의 상태를 한꺼번에 인자로 넘기는 것이 있을법한 일인지 의구심이 얼마든지 들 수 있다. 하지만 실제 Hugs 와 같은 하스켈 시스템에서는 동작action 을 여기에서 묘사한 것보다 훨씬 더 효율적으로 구현하며, 동작이 어떻게 돌아가는지 그 원리를 익히는 데는 이런 추상적인 관점에서 이해하는 것으로도 충분하다.

9.3 기본 동작 basic action

이제 대화식 프로그램을 구성하는 세 가지 기본 동작에 대해 알아보자. 우선 첫째로, *getChar*라는 동작은 글쇠판으로부터 글자를 읽어들여 화면에 그 글자를 그대로 보여주고 결과값으로 또한 그 글자를 돌려준다.

```
getChar :: IO Char
getChar = ...
```

실제 *getChar*의 정의는 Hugs 시스템 안에 내장된 상태로 기본 동작으로 제공되며, 하스켈 언어 자체 내에서는 정의할 수 없다. 글쇠판으로부터 읽어들일 글자가 없으면, *getChar*는 글쇠판을 쳐서 글자가 들어올 때까지 기다린다.

이와 맞짝을 이루는 동작 *putChar c*는 글자 *c*를 화면에 나타내고 결과값으로는 아무것도 돌려줄 필요가 없으므로 빈 순서쌍을 돌려준다.

```
putChar :: Char → IO ()
putChar c = ...
```

마지막으로 살펴볼 기본 동작 *return v*는 사용자와 아무런 대화 없이 그냥 결과값 *v*를 돌려주기만 한다.

```
return :: a → IO a
return v = λworld → (v, world)
```

return 함수는 곁따르는 반응이 없는 순수한^{pure} 식의 세계로부터 곁따르는 반응이 있어 불순한^{impure} 동작의 세계로 건너갈 수 있도록 다리를 놓아 주는 함수이다. 하지만, 결정적으로, 다시 돌아올 수 있는 다리는 없다. 즉, 한번 불순한 세계로 갔으면 영원히 불순한 상태가 되며, 거기서 벗어날 길은 전혀 없다! 그렇다면, 이런 불순함이 프로그램 전체로 급속히 퍼져나가지 않을까 걱정될 수도 있지만, 실제로 그렇게 되지는 않는다. 대부분의 하스켈 프로그램에서는, 함수들 중 절대 다수가 대화^{interaction} 와는 관계가 없으며, 프로그램의 가장 바깥쪽 층에서 적은 수의 대화식 함수들로만 대화를 처리하기 때문이다.

Hugs에서 동작을 실행하면 겉따르는 반응을 처리한 후 그 결과값은 버린다는 것을 언급하면서 이 절을 마무리하고자 한다. 예컨대, `putChar 'a'`를 실행하면, 화면에 글자 '`a`'를 쓰며 이 동작의 결과값은 버리고, 실행이 끝난다.

```
> putChar 'a'  
'a'
```

9.4 순서대로 엮기^{sequencing}

문법 분석기의 경우와 마찬가지로, 두 동작을 엮는 자연스런 방법은 한 동작 다음 다른 동작을 순서대로 실행하며 첫째 동작에서 돌려주는 바뀐 세계의 상태를 둘째 동작의 현재 세계 상태로 넘겨주는 것이다. 이렇게 순서대로 엮는 연산자 $\gg=$ (“하고 나서”^{then}라고 읽음)를 다음과 같이 정의한다.

$$\begin{aligned} (\gg=) &:: IO a \rightarrow (a \rightarrow IO b) \rightarrow IO b \\ f \gg= g &= \lambda world \rightarrow \text{case } f \text{ world of } (v, world') \rightarrow g v world' \end{aligned}$$

이는, 동작 f 를 현재 세계의 상태에 적용하고, 그 결과로 얻은 바뀐 세계의 상태에 함수 g 를 적용하여 바뀐 상태를 둘째 동작에 넘겨주는 방법으로 최종 결과값을 얻는다는 것을 나타낸다. 그러나, 실제로는, 앞장에서 살펴본 문법 분석기의 경우와 마찬가지로, $\gg=$ 로 엮어 만든 동작을 **do** 표현을 써서 더 보기 좋은 형태로 작성한다.

예컨대, **do** 표현으로 간단한 동작 `getChar`를 글쇠판으로부터 글자를 읽어들이고, 읽은 글자를 화면에 보여주고, 그 글자를 또한 결과값으로 돌려주는 세 가지 동작으로 나누어 정의할 수 있다.

```
getChar :: IO Char  
getChar = do x ← getCh  
            putChar x  
            return x
```

글쇠판으로부터 글자를 읽어들일 때 화면에 글자를 보여주지 않는 *getCh* 동작은 표준 서막에는 없지만 이전 버전의 Hugs에서 추가로 제공하는 함수로서, 스크립트에 다음과 같은 특별한 형태의 줄을 써넣고 나면 사용할 수 있다.

```
primitive getCh :: IO Char
```

하지만 최신 버전의 Hugs 또는 GHC에서는 *getCh*를 더이상 지원하지 않으므로 대신에 다음과 같이 *getCh*를 정의할 수 있다.

```
import System.IO
getCh :: IO Char
getCh = do e ← hGetEcho stdin
          b ← hGetBuffering stdin
          hSetEcho stdin False
          hSetBuffering stdin NoBuffering
          c ← getChar
          hSetEcho stdin e
          hSetBuffering stdin b
          return c
```

문제는 유닉스나 리눅스에서 GHC를 쓸 때만 위 코드가 제대로 돌아가며, Hugs나 윈도우즈 GHC에서는 이렇게 정의해도 제대로 동작하지 않는다.

옮긴이 주: Hugs나 윈도우즈 GHC에서는 버퍼링 설정을 변경하는 *System.IO* 모듈의 라이브러리 함수 *hSetBuffering*이 동작하지 않는다. (우리말 판으로 옮기는 작업이 진행되는 동안에 나온 GHC 6.10.3판 이후부터는 윈도우즈에서도 *hSetBuffering*이 제대로 동작하도록 고쳐졌다고 한다.)

윈도우즈 GHC에서는 다음과 같이 *getCh*를 구현할 수 있다.

```
{-# LANGUAGE ForeignFunctionInterface #-}
import Monad
import Char
import Foreign.C
getCh = liftM (chr ∘ fromEnum) c_getch
foreign import ccall unsafe "conio.h getch" c_getch :: IO CInt
```

옮긴이 주: 이 윈도우즈용 *getCh*는 윈도우즈에서만 동작한다. 참고로 이와 같이 Haskell과 C 등의 다른 언어에 사이에 서로 함수를 불러 쓸 수 있도록 하는 기능을 외부 함수 인터페이스(Foreign Function Interface)라 하며, 하스켈로 작성되지 않은 기존의 라이브러리나 프로그램을 하스켈과 연동할 때 쓴다.

따라서 *getCh*를 쓰는 9.6 절 계산기 프로그램은 Hugs 대신 ghci로 실습해야만 한다. 그 다음에 나오는 9.7 절 생명 게임 보기는 *getCh*를 쓰지는 않으므로 유닉스나 리눅스에서는 Hugs로 실습할 수 있다. 하지만 생명 게임 역시 계산기와 마찬가지로 윈도우즈 명령줄에서는 작동하지 않는 ANSI 코드를 쓰므로 윈도우즈에서는 생명 게임 역시 GHC로 실습하는 것이 좋다.

9.5 간단한 동작 이끌어내기

이제 세 가지 기본 동작과 함께 순서대로 엮는 연산자를 써서 쓸모있는 간단한 동작을 여럿 만들 수 있다. 우선 첫째로, 글쇠판으로부터 글줄을 읽어들이는 *getLine* 동작을 정의하자.

```
getLine :: IO String
getLine = do x ← getChar
            if x == '\n' then
                return []
            else
                do xs ← getLine
                return (x : xs)
```

(기호 '\n'은 줄바꿈 글자를 나타낸다.) 그와 맞짝을 이루는 동작으로 *putStr*과 *putStrLn*를 정의한다. 이 두 동작은 모두 인자로 받은 글줄을 화면에 쓰며, *putStrLn*은 글줄을 쓰고 난 다음 줄바꿈을 한다.

```
putStr      :: String → IO ()
putStr []   = return ()
putStr (x : xs) = do putChar x
                      putStr xs
putStrLn    :: String → IO ()
putStrLn xs = do putStr xs
                  putChar '\n'
```

예컨대, 글쇠판으로부터 들어오는 글줄을 기다리고 있다가 글줄이 들어오면 그 길이를 화면에 보여주는 동작을, 다음과 같이 앞서 살펴본 세 가지 간단한 동작으로 정의할 수 있다.

```
strlen :: IO ()
strlen = do putStrLn "글 줄을 쳐 넣으시오 : "
            xs ← getLine
            putStrLn "이 글 줄은 "
            putStrLn (show (length xs))
            putStrLn "글자입니다"
```

옮긴이 주: 이 *strlen* 보기로 GHC로 실습하려면 화면에 나타낼 글귀를 한글 대신 알파벳으로 써야 한다. Hugs는 한글을 지역화 설정(locale)에 맞게 출력해 주지만 GHC에서는 소스코드로부터 한글을 출력할 때 인코딩을 따로 처리해 줘야 한다.

예컨대,

```
> strlen
글 줄을 쳐 넣으시오: abcde
이 글 줄은 5 글자입니다
```

지금까지는 이미 표준 라이브러리에 있는 간단한 동작들을 정의해 보았다. 앞으로 이 장에서 살펴볼 다른 보기들을 만드는 데 쓰일, 그 밖의 여러가지 쓸모있는 간단한 동작들을 더 정의할 것이다. 우선 첫째로, 빼 소리를 내는 동작과 화면을 말끔히 지우는 동작을 다음과 같이 각각의 동작에 알맞은 제어 글자를 출력함으로써 정의한다.

```

beep :: IO ()
beep = putStrLn "\BEL"
cls  :: IO ()
cls  = putStrLn "\ESC[2J"

```

관례상, 화면에 나타나는 글자의 위치를 두 양의 정수로 이루어지는 순서쌍 (x, y) 로 나타내며, 이 때 $(1, 1)$ 이 왼쪽 위 모퉁이를 나타낸다. 이런 화면 위치를 나타내는 타입을 다음과 같이 정의한다.

```
type Pos = (Int, Int)
```

커서는 다음 글자가 어디에 나타날지 표시하는데, 이제 알맞은 제어 글자를 써서 이 커서를 원하는 위치로 옮기는 함수를 정의할 수 있다.

```

goto      :: Pos → IO ()
goto (x, y) = putStrLn ("\ESC[" ++ show y ++ ";" ++ show x ++ "H")

```

옮긴이 주: *cls* 나 *goto* 같이 ANSI 제어 글자로 구현한 함수는 대개의 유닉스 터미널이나 DOS 호환 명령줄에서는 동작하지만, 경우에 따라 동작하지 않는 환경이 있을 수도 있다. 특히 WinHugs는 텍스트 기반 환경이 아니라 제대로 동작하지 않는다. 따라서 이 장에서 WinHugs로는 제대로 실습할 수 없다. 뿐만 아니라 윈도우즈 XP 등 NT 계열 커널의 OS에서는 명령줄에서 ANSI 코드가 잘 동작하지 않으므로, 윈도우즈 XP 등에서는 *cls*나 *goto*를 앞서 정의한 *getCh*처럼 윈도우즈 콘솔 API를 이용한 외부 함수 호출 등 다른 방법으로 구현해야 한다. 윈도우즈 환경을 위한 계산기와 생명 게임 실습을 위한 스크립트는 이 책(우리말 판)의 홈페이지에서 내려받을 수 있다. 이 책의 본문은 ANSI 코드를 사용하는 유닉스나 리눅스 터미널을 기준으로 하고 있으므로 리눅스나 유닉스 사용자는 책 본문에 나오는 정의를 그대로 이용할 수 있고 마찬가지로 이 책(우리말 판) 홈페이지에서 실습을 위한 함수가 미리 정의된 스크립트를 내려받을 수 있다. 윈도우즈 환경의 사용자를 위해서 9.6 절과 9.7 절의 끝부분에 나오는 스크립트 한눈에 살펴보기에서는 윈도우즈 환경의 스크립트를 기준으로 설명하였다.

다음으로는, 화면에서 원하는 위치에 글자를 출력하는 함수를 정의하자.

```

writeat      :: Pos → String → IO ()
writeat p xs = do goto p
                  putStrLn xs

```

마지막으로, 리스트에 나타나는 순서대로 주어진 동작을 실행하며 그 각각의 결과는 버리고 의미없는 값을 돌려주는 *seqn* 함수를 정의하자.

```
seqn      :: [IO a] → IO ()
seqn []    = return ()
seqn (a : as) = do a
                     seqn as
```

예컨대, *seqn* 과 리스트 조건제시식을 써서 *putStr* 를 다음과 같이 더 간결하게 다시 정의할 수 있다.

```
putStr xs = seqn [putChar x | x ← xs]
```

9.6 계산기

앞장의 마지막 부분에서 산술식 문법 분석기를 만들어 보았다. 그것을 확장하여 사용자가 대화식으로 산술식을 글쇠판으로 쳐서 입력하고 그 식의 결과를 화면에서 바로 볼 수 있는 간단한 계산기를 만들어 보자.

우리가 만들 계산기는 정수 및 정수를 덧셈, 뺄셈, 곱셈, 나눗셈, 괄호로 엮어 만들 수 있는 모든 식을 처리할 수 있어야 한다. 이 장에서는 대화식 프로그램을 중점적으로 다루고자 하므로 문법 분석 및 식의 값을 구하는 것에 대해서는 자세히 다루지 않으며, 그런 문법 분석기 *expr :: Parser Int* 은 이미 만들어져 있다고 가정하겠다. 앞 장의 연습문제를 풀었다면 그런 문법 분석기를 만들 수 있을 것이다.

먼저 계산기 사용자 인터페이스부터 만들어 보자. 우선 첫째로, 계산기 기판을 다음과 같은 글줄 리스트로 정의한다.

```

box :: [String]
box = ["+-----+",
       "|           |",
       "|-----+---+---+",
       "| q | c | d | = |",
       "|-----+---+---+",
       "| 1 | 2 | 3 | + |",
       "|-----+---+---+",
       "| 4 | 5 | 6 | - |",
       "|-----+---+---+",
       "| 7 | 8 | 9 | * |",
       "|-----+---+---+",
       "| 0 | ( | ) | / |",
       "|-----+---+---+"]

```

사용자는 계산기의 첫 네 단추인 q,c,d,=를 눌러 프로그램을 그만두거나`quit`, 화면을 말끔히 지우거나`clear`, 글자를 지우거나`delete`, 식의 값을 구할 수 있다. 나머지 열여섯 개의 단추로는 사용자가 계산기에 식을 입력할 수 있다.

또한, 계산기에 쓰이는 단추들을 글자 리스트로 정의하는데, 여기에는 기판 자체에 나타난 스무 개의 표준적인 단추와 함께, 좀더 편하게 계산기를 쓰도록 돋는 확장 글쇠도 몇 개 들어 있다. 확장 글쇠로는 Q,C,D, 빈칸`space`, 나옴`escape`, 뒤로`backspace`, 지움`delete`, 줄바꿈`newline`을 쓴다.

```

buttons :: [Char]
buttons = standard ++ extra
where
    standard = "qcd=123+456-789*0()/"
    extra = "QCD \ESC\BS\DEL\n"

```

옮긴이 주: 원도우즈에서는 줄바꿈 글자 입력이 유닉스나 리눅스와 다르므로 <code>extra</code> = <code>"QCD \ESC\BS\DEL\n\r"</code> 와 같이 글자 ' <code>\r</code> ' 을 하나 더 추가해야 한다.

계산기 기판을 화면의 왼쪽 위에 보여주는 동작은 다음과 같이 리스트 조건제시식으로 정의할 수 있다.

```
showbox :: IO ()
showbox = seqn [ writeat (1, y) xs | (y, xs) ← zip [1..13] box]
```

사용자 인터페이스의 마지막 부분은 계산기 화면에 입력 글줄을 보여주는 함수로서, 아래와 같이 먼저 화면을 말끔히 지운 다음 입력 글줄의 끝 열세 글자만 보여준다. 이러한 방식으로, 사용자가 입력 글줄에서 글자 하나를 지우면 자동으로 화면에서도 지워지며, 사용자가 열세 글자를 넘게 치면 글자를 더 칠 때마다 화면이 왼쪽으로 이동하게 된다.

```
display    :: String → IO ()
display xs = do writeat (3, 2) "
                           "
                           writeat (3, 2) (reverse (take 13 (reverse xs)))
```

calc 는 계산기의 동작 자체를 나타내는 함수로서, 현재 입력 글줄을 화면에 보여준 다음, 글쇠판으로부터 글자를 화면에 보여주지 않고 읽어들인다. 이렇게 읽어들인 글자가 입력 가능한 단추일 경우에는 글자를 처리^{process}하고, 그렇지 않다면 삑 소리를 내서 잘못된 글자가 들어왔다는 것을 알려준 다음 현재 입력 글줄 그대로 계산기 동작을 계속한다.

```
calc    :: String → IO ()
calc xs = do display xs
            c ← getCh
            if elem c buttons then
                process c xs
            else
                do beep
                calc xs
```

process 는 입력 가능한 글자와 현재 입력 글줄을 인자로 받아, 그 받은 글자에 알맞은 동작을 실행하는 다음과 같은 함수다.

```

process          :: Char → String → IO ()
process c xs
| elem c "qQ\ESC"    = quit
| elem c "dD\BS\DEL" = delete xs
| elem c "=\\n"       = eval xs
| elem c "cC"        = clear
| otherwise           = press c xs

```

이제 가능한 다섯 가지 동작을 차례대로 하나씩 살펴보자.

- *quit* 은 커서를 계산기 기판 아래로 옮기고 프로그램을 끝낸다.

```

quit :: IO ()
quit = goto (1, 14)

```

- *delete* 는 현재 입력 글줄이 비어 있으면 아무 효과가 없고, 그렇지 않으면 입력 글줄에서 마지막 글자를 버린다.

```

delete   :: String → IO ()
delete "" = calc ""
delete xs = calc (init xs)

```

- *eval* 은 현재 입력 글줄을 문법 분석하고 계산하여 그 결과를 화면에 나타내는데, 문법 분석이 실패하면 빽 소리를 낸다.

```

eval   :: String → IO ()
eval xs = case parse expr xs of
            [(n, "")] → calc (show n)
            _           → do beep
                           calc xs

```

- *clear* 는 입력 글줄을 비어 있는 초기 상태로 되돌려 놓는다.

```

clear :: IO ()
clear = calc ""

```

- 그 외에 다른 글자들이 들어오면 현재 입력 글줄의 끝에 이어붙인다.

```
press      :: Char → String → IO ()  
press c xs = calc (xs ++ [c])
```

마지막으로 계산기를 실행하는 함수를 다음과 같이 화면을 말끔히 지우고, 계산기를 화면에 나타내고, 빈 입력 글줄로 시작하도록 정의하면 된다.

```
run :: IO ()  
run = do cls  
        showbox  
        clear
```

스크립트 *calculatorWin32.lhs* 한눈에 다시보기

```
1 Programming in Haskell 9.6 절 계산기 보기,
2 Graham Hutton, Cambridge University Press, 2007.
3
4 Note: the definition for getCh in this example works with the
5 Glasgow Haskell Compiler, but may not work with some Haskell
6 systems, such as Hugs. Moreover, the use of control characters
7 may not work on some systems, such as WinHugs.
8
9 Note by Ahn, Ki Yung: This code works on GHC versions 6.10.1
10 on MS Windows command line using the code extracted from
11 ansi-terminal-0.5.0 package. You will need Win32ANSI.hs
12 in addition to Parsing.lhs. To run this code you should invoke
13 the compiler first to compile Win32ANSI.o and then the run ghci
14 as follows:
15 C:\> ghc -c Win32ANSI.hs
16 C:\> ghci calculatorWin32.lhs
17 ...
18
19 > {-# LANGUAGE ForeignFunctionInterface #-}
20 > import Parsing
21 > import Win32ANSI (setCursorPosition)
22 >
23 > import Monad
24 > import Char
25 >
26 > import System.IO
27 > import System.Cmd (system)
28 >
29 > import Foreign.C
```

30

```
31 수식 문법분석기
32 -----
33
34 > expr      :: Parser Int
35 > expr      = do t <- term
36   >           do symbol "+"
37   >           e <- expr
38   >           return (t + e)
39   >           +++ do symbol "-"
40   >           e <- expr
41   >           return (t - e)
42   >           +++ return t
43 >
44 > term      :: Parser Int
45 > term      = do f <- factor
46   >           do symbol "*"
47   >           t <- term
48   >           return (f * t)
49   >           +++ do symbol "/"
50   >           t <- term
51   >           return (f `div` t)
52   >           +++ return f
53 >
54 > factor    :: Parser Int
55 > factor    = do symbol "("
56   >           e <- expr
57   >           symbol ")"
58   >           return e
59   >           +++ integer
60
```

```
61  간단한 동작 예를 아래에
62 -----
63
64 > getch      :: IO Char
65 > getch      = liftM (chr . fromEnum) c_getch
66 > foreign import ccall unsafe "conio.h getch" c_getch :: IO CInt
67 >
68 > beep       :: IO ()
69 > beep      = do putStrLn "\BEL"
70 >                  hFlush stdout
71 >
72 > cls        :: IO ()
73 > cls      = do system("cls")
74 >                  return ()
75 >
76 > type Pos   = (Int,Int)
77 >
78 > goto       :: Pos -> IO ()
79 > goto (x,y) = setCursorPosition (y-1) (x-1)
80 >
81 > writeat    :: Pos -> String -> IO ()
82 > writeat p xs = do goto p
83 >                  putStrLn xs
84 >                  hFlush stdout
85 >
86 > seqn       :: [IO a] -> IO ()
87 > seqn []     = return ()
88 > seqn (a:as) = do a
89 >                  seqn as
90
```

```

91 계산기
92 -----
93
94 > box      :: [String]
95 > box      = ["-----+",
96 >           "|       |",
97 >           "+-----+",
98 >           "| q | c | d | = |",
99 >           "+-----+",
100 >          "| 1 | 2 | 3 | + |",
101 >          "+-----+",
102 >          "| 4 | 5 | 6 | - |",
103 >          "+-----+",
104 >          "| 7 | 8 | 9 | * |",
105 >          "+-----+",
106 >          "| 0 | ( | ) | / |",
107 >          "+-----+"]
108 >
109 > buttons   :: String
110 > buttons   = standard ++ extra
111 >           where
112 >             standard = "qcd=123+456-789*0()/"
113 >             extra   = "QCD \ESC\BS\DEL\n\r"
114 >
115 >
116 > showbox    :: IO ()
117 > showbox    = seqn [writeat (1,y) xs | (y,xs) <- zip [1..13] box]
118 >
119 > display xs = do writeat (3,2) "
120 >                   writeat (3,2) (reverse (take 13 (reverse xs)))
121 >
122 > calc       :: String -> IO ()
123 > calc xs    = do display xs
124 >           c <- getch
125 >           if elem c buttons then
126 >             process c xs
127 >           else
128 >             do beep
129 >             calc xs
130 >
131

```

```
132
133 > process          :: Char -> String -> IO ()
134 > process c xs
135 > | elem c "qQ\ESC" = quit
136 > | elem c "dD\BS\DEL" = delete xs
137 > | elem c "=\n\r" = eval xs
138 > | elem c "cC" = clear
139 > | otherwise = press c xs
140 >
141 > quit           :: IO ()
142 > quit           = goto (1,14)
143 >
144 > delete          :: String -> IO ()
145 > delete ""      = calc ""
146 > delete xs      = calc (init xs)
147 >
148 > eval            :: String -> IO ()
149 > eval xs         = case parse expr xs of
150 >                   [(n,"")] -> calc (show n)
151 >                   _           -> do beep
152 >                           calc xs
153 >
154 > clear           :: IO ()
155 > clear           = calc ""
156 >
157 > press            :: Char -> String -> IO ()
158 > press c xs      = calc (xs ++ [c])
159 >
160 > run              :: IO ()
161 > run              = do cls
162 >                     showbox
163 >                     clear
164
```

따라하기

다음은 *run* 함수로 계산기를 실행하기 직전 화면이다.

```
C:\WINDOWS\system32\cmd.exe - ghci calculatorWin32.lhs
K:\MyDoc\chap09>dir
  K 드라이브의 볼륨: kyagrd
  볼륨 일련 번호: 125D-0985

  K:\MyDoc\chap09 디렉터리

  2009-01-01  09:16    <DIR>      .
  2009-01-01  09:10    <DIR>      ..
  2008-12-29  03:28          6,302 calculatorWin32.lhs
  2008-12-16  22:03          8,105 Win32ANSI.hs
  2008-12-16  02:45          4,325 Parsing.lhs
                           3개 파일          18,732 바이트
                           2개 디렉터리  42,337,214,464 바이트 남음

K:\MyDoc\chap09>ghc -c Win32ANSI.hs

K:\MyDoc\chap09>dir
  K 드라이브의 볼륨: kyagrd
  볼륨 일련 번호: 125D-0985

  K:\MyDoc\chap09 디렉터리

  2009-01-01  09:17    <DIR>      .
  2009-01-01  09:10    <DIR>      ..
  2008-12-29  03:28          6,302 calculatorWin32.lhs
  2009-01-01  09:17          2,079 Win32ANSI.hi
  2008-12-16  22:03          8,105 Win32ANSI.hs
  2008-12-16  02:45          4,325 Parsing.lhs
  2009-01-01  09:17          51,097 Win32ANSI.o
                           5개 파일          71,908 바이트
                           2개 디렉터리  42,337,148,928 바이트 남음

K:\MyDoc\chap09>ghci calculatorWin32.lhs
GHCi, version 6.10.1: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
[1 of 3] Compiling Parsing      < Parsing.lhs, interpreted >
[3 of 3] Compiling Main         < calculatorWin32.lhs, interpreted >
Ok, modules loaded: Main, Win32ANSI, Parsing.
*Main> run_
```

위 윈도우즈 명령줄 (cmd.exe)에서 일어나고 있는 일은 다음과 같다.

1. 먼저 필요한 세 스크립트 파일 *calculatorWin32.lhs*, *Win32ANSI.hs*, *Parsing.lhs*가 현재 디렉토리에 있는지 확인한다.
2. *Win32ANSI.hs*로부터 확장자가 .o인 목적 파일을 얻기 위해 ghc에 -c 옵션을 주어 컴파일한다.
3. 계산기를 정의하는 *calculatorWin32.lhs*를 ghci로 불러들여 *run* 함수로 이제 막 실행하려 하고 있다.

다음은 ghci에서 *run* 함수 실행 직후 나타나는 계산기 실행 화면이다.

글쇠판으로 화면에 나타난 숫자 및 기호를 입력하면 대화식으로 동작하는 계산기 프로그램을 시험해 볼 수 있다.
옮긴이 주: 유닉스나 리눅스에서는 *calculator.lhs* 와 *Parsing.lhs* 두 스크립트만 있으면 되고, ghci에서 *calculator.lhs* 만 불러오면 되는 등 실습이 훨씬 간단하다. 유닉스 / 리눅스용 스크립트 역시 이 책(한글판) 홈페이지에 내려받을 수 있다.

윈도우즈에서 실습이 다소 복잡해지는 이유는 앞서 옮긴이 주에서 이미 언급한 바와 같이 윈도우즈 명령줄(cmd.exe)이 ANSI 코드를 지원하는 터미널이 아니기 때문에 직접 *Win32ANSI.hs*라는 외부 함수 호출 관련 모듈을 작성하여 따로 컴파일하는 과정이 필요하기 때문이다. 또한 Hugs나 윈도우즈에서는 GHC에서는 표준 입출력 버퍼링 관련 표준 라이브러리가 제대로 구현되어 있지 않다.

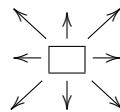
따라하기 끝

9.7 생명 게임 game of life

생명 게임을 다루는 좀더 큰 보기 하나를 더 만들어 보는 것으로 이 장을 맺겠다. 이 게임은 간단한 진화 시스템^{evolutionary system}의 모형으로, 이차원 판에서 진행된다. 아래 보기처럼, 판 위의 네모칸에는 아무것도 없거나 살아있는 세포가 들어 있다.

			○
○			○
	○		○

판 위의 네모칸 주위에는 맞붙어 있는 이웃이 여덟 있다.



일관성을 유지하기 위해, 판의 가장자리에 있는 네모칸 주변에도 맞붙어 있는 이웃이 여덟 있는 것으로 간주하며, 이 때 위-아래 모서리가 맞붙어 있고 왼쪽-오른쪽 모서리가 맞붙어 있는 것으로 생각한다. 즉, 이 판을 실제로는 토러스, 곧 삼차원에서 도넛 모양 도형의 껍데기라고 생각할 수 있다.

판의 초기 상태에 다음 규칙을 모든 네모칸에 동시에 적용하면 다음 세대^{generation}를 얻는다.

- 살아 있는 세포 주위에 맞붙어 있는 이웃에 살아 있는 세포가 둘 또는 셋 있을 경우에는 세포가 계속 살고, 그렇지 않으면 세포가 죽어 빈칸이 된다.
- 빈칸 주변에 맞붙은 이웃에 살아있는 세포가 셋 있을 경우에는 그곳에서 새로운 세포가 탄생하고, 그렇지 않으면 계속 빈 상태로 있다.

예컨대, 이 규칙들을 앞의 판에 적용하면 다음과 같이 된다.

	○		
		○	○
	○	○	

새로 얻은 판에 이러한 과정을 되풀이하면 무한히 많은 세대로 이루어진 수열^{sequence}이 만들어진다. 초기 조건을 잘 주면 흥미로운 성질이 나타나는 수열을 여러가지 만들어낼 수 있다. 예컨대, 앞에서 살펴본 세포 배열은 글라이더^{glider}라 부르며, 계속해서 다음 세대로 진행함에 따라 대각선 아래 방향으로 움직인다.

그 단순성에도 불구하고, 생명 게임은 사실 계산적으로 완전한데, 이는 어떤 계산 과정이든 생명 게임으로 알맞게 부호화하여 시늉낼 수 있음을 뜻한다. 앞으로 이 절에서는 하스켈로 생명 게임을 어떻게 만들 수 있는지 알아보겠다.

우선, 판의 크기를 쉽게 바꿀 수 있도록, 판의 너비와 높이를 나타내는 정수값을 두 개 정의하자.

```
width :: Int  
width = 5  
height :: Int  
height = 5
```

판은 살아있는 세포가 있는 위치를 (x, y) 로 나타낸 리스트로 나타내며, 이 때 위치 표현은 계산기 보기에서와 똑같은 방식의 좌표를 따른다.

```
type Board = [Pos]
```

예컨대, 보기로 살펴본 판의 초기 조건은 다음과 같이 나타낼 수 있다.

```
glider :: Board  
glider = [(4, 2), (2, 3), (4, 3), (3, 4), (4, 4)]
```

판을 이렇게 나타내 놓고 보면, 살아있는 세포를 화면에 보여주는 함수나 어떤 위치에 세포가 살아있는지 혹은 빙칸인지 알아보는 함수를 쉽게 정의할 수 있다.

```
showcells :: Board → IO ()  
showcells b = seqn [writeat p "0" | p ← b]  
isAlive :: Board → Pos → Bool  
isAlive b p = elem p b  
isEmpty :: Board → Pos → Bool  
isEmpty b p = ¬ (isAlive b p)
```

다음으로는, 맞붙은 이웃의 위치를 돌려주는 함수를 정의하자.

```
neighbs      :: Pos → [Pos]
neighbs (x, y) = map wrap [(x - 1, y - 1), (x, y - 1),
                                         (x + 1, y - 1), (x - 1, y),
                                         (x + 1, y), (x - 1, y + 1),
                                         (x, y + 1), (x + 1, y + 1)]
```

wrap 은 판의 가장자리가 뒤로 밀려 맞붙어 있다는 사실을 반영하는 보조함수로, 주어진 위치의 각 성분에서 1씩 빼고 나서 판의 너비와 높이로 나눈 나머지를 구해 다시 각 성분에 1씩 더해 주도록 다음과 같이 정의한다.

```
wrap      :: Pos → Pos
wrap (x, y) = (((x - 1) `mod` width) + 1,
                     ((y - 1) `mod` height) + 1)
```

어떤 위치가 주어졌을 때 맞붙은 이웃에 살아있는 세포가 몇이나 있는지 계산하는 함수를, 맞붙은 이웃을 구하고, 그 중 살아있는 세포가 들어있는 곳만 골라, 그 개수를 세도록 다음과 같이 함수 합성 연산자를 써서 정의할 수 있다.

```
liveneighbs :: Board → Pos → Int
liveneighbs b = length ∘ filter (isAlive b) ∘ neighbs
```

이 함수를 이용하면, 판 위의 세포들 중 맞붙어 이웃하는 세포들이 둘 혹은 셋 있는 세포들의 위치, 곧 다음 세대에서 살아남을 세포들의 위치를 구하는 것은 쉽고 간단하다.

```
survivors :: Board → [Pos]
survivors b = [p | p ← b, elem (liveneighbs b p) [2, 3]]
```

다음으로는, 판 위의 빈칸들 중 이웃하는 세포들이 셋 있는 빈칸의 위치, 곧 새로운 세포가 태어나는 위치를 아래와 같이 구할 수 있다.

```

births :: Board → [Pos]
births b = [(x, y) | x ← [1..width],
                           y ← [1..height],
                           isEmpty b (x, y),
                           liveneighbs b (x, y) == 3]

```

하지만, 위 정의에서는 판 위의 모든 위치를 다 알아보고 있다. 커다란 판에서 더 좋은 효율을 낼 수 있는 더 나은 방법은 바로 살아있는 세포 주변만 알아보는 것인데, 왜냐하면 그러한 곳에서만 세포가 새로 탄생할 수 있기 때문이다. 이 방법에 따라 *births* 함수를 다시 정의하면 다음과 같다.

```

births b = [p | p ← rmdups (concat (map neighbs b)),
                           isEmpty b p,
                           liveneighbs b p == 3]

```

*rmdups*는 리스트에서 중복된 원소를 버리는 보조 함수로, 세포가 새로 태어날 수 있는 곳을 한번씩만 살펴보도록 한다.

```

rmdups          :: Eq a ⇒ [a] → [a]
rmdups []      = []
rmdups (x : xs) = x : rmdups (filter ( $\neq$  x) xs)

```

이제, 살아남은 세포 리스트에 새로 태어난 세포 리스트를 이어붙이기만 하면 다음 세대의 판을 얻을 수 있다.

```

nextgen :: Board → Board
nextgen b = survivors b  $\uparrow\downarrow$  births b

```

마지막으로, 생명 게임 자체를 구현하는 *life* 함수를 화면을 말끔히 지우고, 지금 판에 살아있는 세포를 보여준 후, 잠깐 기다린 다음, 새로운 세대로 이같은 과정을 계속하도록 다음과 같이 정의하면 된다.

```
life   :: Board → IO ()
life b = do cls
           showcells b
           wait 5000
           life (nextgen b)
```

wait 는 생명 게임이 적당한 속도로 진행되도록 게임의 속도를 늦춰주는 함수로, 다음과 같이 의미없는 동작을 주어진 횟수만큼 실행하도록 정의한다.

```
wait   :: Int → IO ()
wait n = seqn [return () | _ ← [1..n]]
```

글라이더 보기지를 *life* 함수로 한번 돌려 보고, 또 직접 몇 가지 형태의 초기 조건을 만들어서 어떤 성질이 나타나는지 시험해 보면 재미날 것이다.

스크립트 *lifeWin32.lhs* 한눈에 다시보기

```

1 Programming in Haskell 9.7 절 생명 게임 보기,
2 Graham Hutton, Cambridge University Press, 2007.
3
4 Note: the control characters used in this example may not work
5 on some Haskell systems, such as WinHugs.
6
7 Note by Ahn, Ki Yung: This code works on GHC versions 6.10.1
8 on MS Windows command line using the code extracted from
9 ansi-terminal-0.5.0 package. You will need Win32ANSI.hs
10 in addition to Parsing.lhs. To run this code you should invoke
11 the compiler first to compile Win32ANSI.o and then the run ghci
12 as follows:
13   C:\> ghc -c Win32ANSI.hs
14   C:\> ghci lifeWin32.lhs
15   ...
16
17 > import Win32ANSI (setCursorPosition)
18 > import System.IO
19 > import System.Cmd (system)
20
21 간단한 동작 이끌어내기
22 -----
23
24 > cls          :: IO ()
25 > cls          = do system("cls")
26 >                   return ()
27 >
28 > type Pos     = (Int,Int)
29 >
30 > goto         :: Pos -> IO ()
31 > goto (x,y)   = setCursorPosition (y-1) (x-1)
32 >
33 > writeat      :: Pos -> String -> IO ()
34 > writeat p xs = do goto p
35 >                   putStr xs
36 >
37 > seqn         :: [IO a] -> IO ()
38 > seqn []       = return ()
39 > seqn (a:as)   = do a
40 >                   seqn as
41
42 생명 게임
43 -----
44
45 > width        :: Int
46 > width        = 5
47 >
48 > height       :: Int
49 > height       = 5
50 >
51 > type Board    = [Pos]
52 >
53 > glider       :: Board
54 > glider       = [(4,2),(2,3),(4,3),(3,4),(4,4)]
55 >
56 > showcells    :: Board -> IO ()
57 > showcells b   = seqn [writeat p "0" | p <- b]
```

```

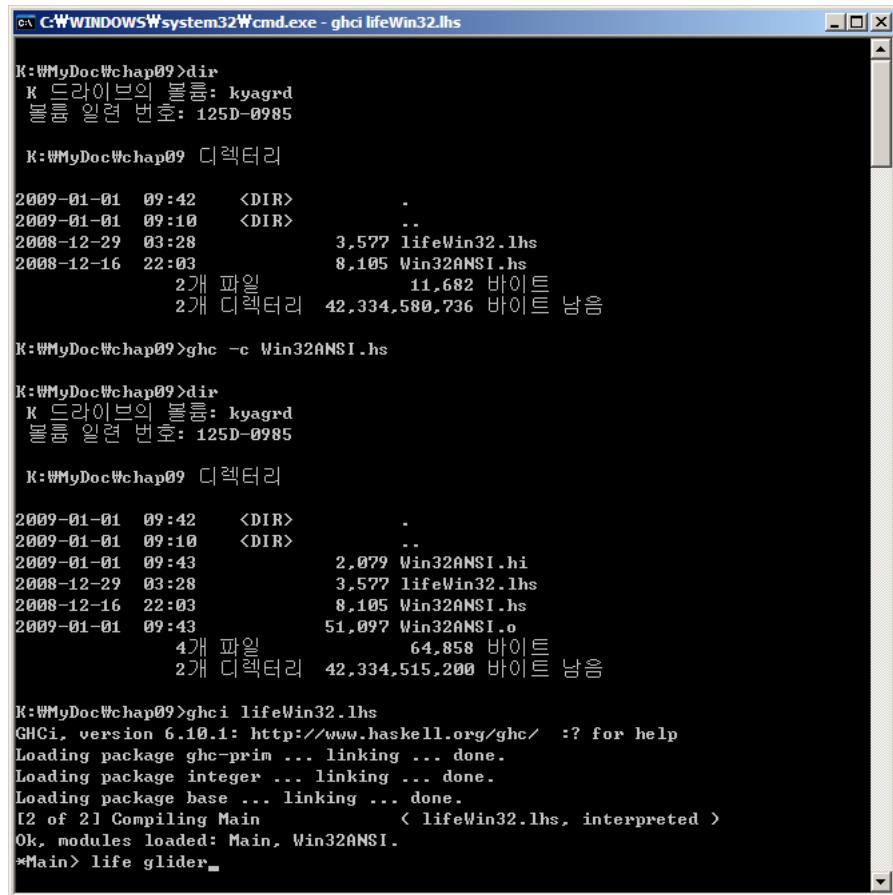
58 >
59 > isAlive      :: Board -> Pos -> Bool
60 > isAlive b p = elem p b
61 >
62 > isEmpty      :: Board -> Pos -> Bool
63 > isEmpty b p = not (isAlive b p)
64 >
65 > neighbs     :: Pos -> [Pos]
66 > neighbs (x,y) = map wrap [(x-1,y-1), (x,y-1),
67 >                                (x+1,y-1), (x-1,y),
68 >                                (x+1,y) , (x-1,y+1),
69 >                                (x,y+1) , (x+1,y+1)]
70 >
71 > wrap         :: Pos -> Pos
72 > wrap (x,y)   = (((x-1) `mod` width) + 1, ((y-1) `mod` height + 1))
73 >
74 > liveneighbs :: Board -> Pos -> Int
75 > liveneighbs b = length . filter (isAlive b) . neighbs
76 >
77 > survivors    :: Board -> [Pos]
78 > survivors b  = [p | p <- b, elem (liveneighbs b p) [2,3]]
79 >
80 > births b     = [p | p <- rmdups (concat (map neighbs b)),
81 >                      isEmpty b p,
82 >                      liveneighbs b p == 3]
83 >
84 > rmdups       :: Eq a => [a] -> [a]
85 > rmdups []     = []
86 > rmdups (x:xs) = x : rmdups (filter (/= x) xs)
87 >
88 > nextgen      :: Board -> Board
89 > nextgen b    = survivors b ++ births b
90 >
91 > life         :: Board -> IO ()
92 > life b       = do cls
93 >                  showcells b
94 >                  wait 5000
95 >                  life (nextgen b)
96 >
97 > wait         :: Int -> IO ()
98 > wait n       = seqn [return () | _ <- [1..n]]
99

```

a

따라하기

다음은 *life* 함수로 생명 게임을 실행하기 직전 화면이다.



```

C:\WINDOWS\system32\cmd.exe - ghci lifeWin32.lhs

K:\MyDoc\chap09>dir
  K 드라이브의 블록: kyagrd
  블록 일련 번호: 125D-0985

  K:\MyDoc\chap09 디렉토리

  2009-01-01  09:42    <DIR>      .
  2009-01-01  09:10    <DIR>      ..
  2008-12-29  03:28        3,577 lifeWin32.lhs
  2008-12-16  22:03        8,105 Win32ANSI.hs
                           2개 파일           11,682 바이트
                           2개 디렉토리   42,334,580,736 바이트 남음

K:\MyDoc\chap09>ghc -c Win32ANSI.hs

K:\MyDoc\chap09>dir
  K 드라이브의 블록: kyagrd
  블록 일련 번호: 125D-0985

  K:\MyDoc\chap09 디렉토리

  2009-01-01  09:42    <DIR>      .
  2009-01-01  09:10    <DIR>      ..
  2009-01-01  09:43        2,079 Win32ANSI.hi
  2008-12-29  03:28        3,577 lifeWin32.lhs
  2008-12-16  22:03        8,105 Win32ANSI.hs
  2009-01-01  09:43        51,097 Win32ANSI.o
                           4개 파일           64,858 바이트
                           2개 디렉토리   42,334,515,200 바이트 남음

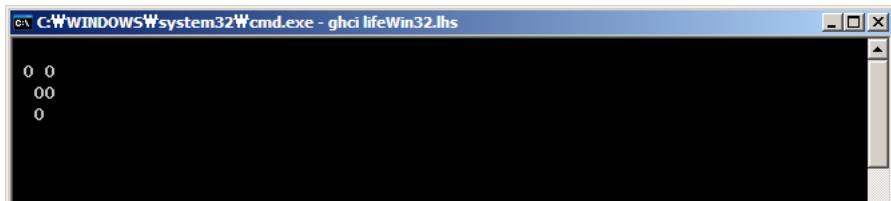
K:\MyDoc\chap09>ghci lifeWin32.lhs
GHCi, version 6.10.1: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
[2 of 2] Compiling Main             < lifeWin32.lhs, interpreted >
Ok, modules loaded: Main, Win32ANSI.
*Main> life glider_

```

위 윈도우즈 명령줄 (cmd.exe)에서 일어나고 있는 일은 다음과 같다.

- 먼저 필요한 두 스크립트 파일 *lifeWin32.lhs*, *Win23ANSI.hs*가 현재 디렉토리에 있는지 확인한다.
- Win23ANSI.hs*로부터 확장자가 .o인 목적 파일을 얻기 위해 ghc에 -c 옵션을 주어 컴파일한다.
- 생명 게임을 정의하는 *lifeWin32.lhs*를 ghci로 불러들인 후, *life* 함수로 *glider*를 초기 판으로 하여 이제 막 생명 게임을 시작하려 하고 있다.

다음은 ghci에서 *life* 함수로 *glider* 를 초기 판으로 하여 생명 게임을 진행하는 화면이다.



위 화면은 계속해서 변한다. 생명 게임을 끝내려면 글쇠판으로 Ctrl-C를 입력하여 빠져나갈 수 있다.

옮긴이 주: 유닉스나 리눅스에서는 *life.lhs* 스크립트 하나만 ghci에서 불러들이기만 하면 바로 실습할 수 있어 훨씬 간단하다. 유닉스/리눅스용 스크립트 역시 이 책(한글판) 홈페이지에 내려받을 수 있으며, 유닉스/리눅스에서 생명 게임을 실습할 때는 Hugs를 사용할 것을 권한다.

윈도우즈에서 실습이 다소 복잡해지는 이유는 앞서 옮긴이 주에서 이미 언급한 바와 같이 윈도우즈 명령줄(cmd.exe)이 ANSI 코드를 지원하는 터미널이 아니기 때문에 직접 *Win32ANSI.hs*라는 외부 함수 호출 관련 모듈을 작성하여 따로 컴파일하는 과정이 필요하기 때문이다.

따라하기 끝

9.8 살펴보기

파일 입출력 및 예외 사건exceptional event 처리 등 다른 형태의 결파르는 반응을 어떻게 IO 타입으로써 나타내는지 하스켈 보고서(25)에서 다루고 있다. 입출력 및 다른 형태의 결파르는 반응의 정확한 의미를 (24)에서 식으로 염밀하게 정리하고 있다. 그래픽 환경에서 대화를 처리하는 다양한 라이브러리들을 하스켈 홈페이지 <http://www.haskell.org/>에서 찾을 수 있다. 생명 게임은 존 콘웨이John Conway가 만들었으며, 마틴 가드너Martin Gardner가 1970년 10월 판 *Scientific American*에 소개함으로써 유명해졌다.

9.9 연습문제

1. `getLine` 과 다른 점은 똑같은데 지움 ^{delete} 글쇠로 글자를 지울 수도 있는 `readLine :: IO String` 동작을 정의하라.
귀띔: 지움 글자는 '\DEL'이며, 커서 `cursor` 를 한 칸 뒤로 옮기는 제어 글줄은 "\ESC[1D" 이다.
2. 계산기 프로그램에 올바르지 않은 식을 입력했을 때 그냥 빼 소리만 내는 대신, 문법 분석기가 아직 못다 쓴 입력을 결과값으로 돌려주는 것을 이용해 대략 어디쯤에서 잘못되었는지 알려주도록 고쳐 보라.
3. 어떤 시스템에서는 생명 게임을 돌릴 때 매 세대마다 화면을 말끔히 지우느라 깜빡거림이 있을 수 있다. 실제로 바뀌는 부분만 화면에 반영함으로써 그런 깜빡거림이 없어지도록 생명 게임 프로그램을 고쳐 보라.
4. 사용자가 대화식으로 생명 게임 판을 만들고 그 내용을 수정할 수 있는 편집기를 만들어 보라.
5. 하스켈 홈페이지 <http://www.haskell.org/> 에 있는 그래픽 라이브러리 중 하나를 써서, 계산기 및 생명 게임 프로그램을 그래픽 판으로 만들어 보라.
6. 님^{Nim}이라는 다섯 줄의 별을 늘어놓은 판에서 하는 놀이는 다음과 같은 상태로 시작한다.

```
1 : * * * * *
2 : * * * *
3 : * * *
4 : * *
5 : *
```

두 사람이 번갈아 가며 어떤 한 줄에서 하나 이상의 별을 가져가다가, 마지막으로 별을 다 가져가는 사람이 이기는 놀이다. 님 놀이를 하스켈로 만들어 보라.

귀띔: 판을 각 줄에 있는 별의 개수를 포함하는 리스트로 나타내면, 처음 시작하는 판은 [5, 4, 3, 2, 1] 로 나타낼 수 있다.

제 10 장

타입과 클래스 선언

이 장에서는 하스켈에서 새로운 타입과 클래스를 선언하는 방법에 대해 알아본다. 먼저 타입을 선언하는 두 가지 방법을 알아보고, 되도는 타입을 살펴본다. 그리고는 늘 참 검사기 tautology checker 와 추상 기계 abstract machine 를 만들어 보고, 클래스와 그 인스턴스를 어떻게 선언하는지 알아보는 것으로 이 장을 맺는다.

10.1 타입 선언

새로운 타입을 선언하는 가장 간단한 방법은 하스켈에서 **type** 예약어를 사용하는 방법으로서, 이미 있는 타입에 다 새로운 이름을 붙이는 것이다. 예컨대, 표준 라이브러리에 있는 다음 선언은 *String* 타입이 글자로 이루어진 리스트를 나타내는 [*Char*] 타입의 다른 이름일 뿐이라는 것을 나타낸다.

```
type String = [Char]
```

위 보기와 같이, 새로운 타입의 이름은 대문자로 시작해야 한다. 타입 선언은 겹겹이 포개져 있을 수도 있는데, 이는 위와 같이 선언된 타입으로 또 다른 타입을 선언할 수 있다는 뜻이다. 예컨대, 앞 장에서는 다음과 같이 판^{board} 을 나타내는 타입을 위치^{position} 을 나타내는 타입을 써서 정의한 바 있다.

```
type Board = [Pos]
type Pos = (Int, Int)
```

하지만, 기술적인 이유로, 자기 자신을 써서 되돌기로 타입 선언을 할 수 없다. 예컨대, 다음과 같은 선언은 잘못된 것으로 간주한다.

```
type Tree = (Int, [Tree])
```

위에서 나타내고자 하는 바는, 나무^{tree} 가 정수와 부분나무^{subtree} 리스트로 이루어진 순서쌍이라는 것이다. 이러한 선언은 부분나무^{subtree} 가 없는 빈 리스트일 때가 되돌기의 밑바탕 경우^{base case} 가 되는 지극히 합리적인 선언이지만, 하스켈에서는 이러한 되도는 타입 선언을 허용하지 않는다. 되도는 타입을 선언해야 할 필요가 있을 때는 **data** 예약어를 사용하는 더 강력한 방법을 써야 하는데, 이에 대해 조만간 살펴보게 될 것이다.

다른 타입을 인자로 받는^{parameterised} 타입을 선언할 수도 있다. 예컨대, 바로 전 두 장에서 아래와 같이 문법 분석기와 대화식 프로그램의 타입을 선언하면서 타입 인자^{type parameter} 를 썼다.

```
type Parser a = String → [(a, String)]
type IO a = World → (a, World)
```

타입 인자를 여러 개 있는 선언도 가능하다. 예컨대, 어떤 타입의 키를 또 다른 타입의 값과 엮는 참조표^{lookup table} 타입을 다음과 같이 (*key*, *value*) 순서쌍으로 이루어진 리스트로 정의할 수 있다.

```
type Assoc k v = [(k, v)]
```

이 타입을 써서 주어진 키와 엮인 첫번째 값을 돌려주는 함수를 다음과 같이 정의할 수 있다.

```
find      :: Eq k ⇒ k → Assoc k v → v
find k t = head [v | (k', v) ← t, k == k']
```

10.2 데이터 선언

이미 있던 타입에 새 이름을 붙이는 것이 아니라 완전히 새로운 타입을 선언하려면, 하스켈의 **data** 예약어를 사용하여 그 새로운 타입에는 어떠한 값이 있을 수 있는지 일일이 늘어놓는 방법으로 할 수 있다. 예컨대, 표준 라이브러리에 있는 다음 선언은 *Bool* 타입이 *True* 와 *False*라는 두 개의 새로운 값들로 이루어진다는 것을 나타낸다.

```
data Bool = False | True
```

위와 같은 선언에서 | 기호를 “이거나”로 읽을 수 있으며, 각각의 새 값을 생성자constructor라고 부른다. 새로 선언하는 타입의 이름과 마찬가지로, 생성자도 대문자로 시작해야 한다. 또한, 같은 이름의 생성자를 여러 타입에 쓸 수 없다.

새로운 타입과 생성자에 붙인 이름들은 Hugs 시스템에서 고유한 뜻을 갖지 않는다는 점에 주목하라. 예컨대, 위 선언 대신에 **data A = B | C**라고 선언하더라도 마찬가지로 제대로 된 선언인데, 이는 각각의 이름이 정확히 무엇인지는 의미가 없고 한 번도 쓰인 적이 없는 이름이기만 하면 되기 때문이다. 프로그래머가 새로운 타입을 다루는 함수를 작성함으로써, 비로소 *Bool*, *False*, *True*와 같은 이름이 뜻을 갖게 된다.

새로운 타입의 값들은 기본 타입의 값들과 똑같이 쓰인다. 좀 더 자세히 말하자면, 새로운 값을 함수에 인자로 넘길 수 있고, 함수의 결과값으로 돌려줄 수도 있고, 리스트와 같은 데이터 구조에 넣어 놓을 수도 있으며, 패턴에 쓰일 수도 있다. 예컨대, 다음과 같은 움직임을 선언했을 때,

```
data Move = Left | Right | Up | Down
```

어떤 위치에 움직임을 적용하는 함수 및 어떤 위치에 일련의 움직임을 적용하는 함수, 그리고 움직임의 방향을 뒤집는 함수를 다음과 같이 정의할 수 있다. (위치는 화면 왼쪽 위를 기준으로 나타낸다는 것을 기억하라.)

```

move          :: Move → Pos → Pos
move Left (x, y) = (x - 1, y)
move Right (x, y) = (x + 1, y)
move Up (x, y) = (x, y - 1)
move Down (x, y) = (x, y + 1)
moves         :: [Move] → Pos → Pos
moves []      p = p
moves (m : ms) p = moves ms (move m p)
flip          :: Move → Move
flip Left    = Right
flip Right   = Left
flip Up      = Down
flip Down   = Up

```

데이터 선언에 나타나는 생성자는 인자를 가질 수 있다. 예컨대, 반지름으로 나타낸 원과 가로 세로로 나타낸 직사각형을 포함하는 도형의 타입을 다음과 같이 선언할 수 있다.

```
data Shape = Circle Float | Rect Float Float
```

즉, *Shape* 타입은 *Circle r* 과 같은 형태 (*r*은 떠돌이 소수점 수) 및 *Rect x y*와 같은 형태의 (*x*와 *y*는 떠돌이 소수점 수) 값을 갖는다. 이 생성자들로 주어진 변의 길이에 맞는 정사각형을 만들어 낸다거나 주어진 도형의 넓이를 계산하는 등, 도형을 다루는 함수들을 다음과 같이 정의할 수 있다.

```

square        :: Float → Shape
square n      = Rect n n
area          :: Shape → Float
area (Circle r) = pi * r ↑ 2
area (Rect x y) = x * y

```

Circle 나 *Rect* 같은 생성자는 인자를 받으므로, 실제로는 알맞은 개수의 *Float* 타입의 인자를 받아 *Shape* 타입의 결과를 내는 생성자 함수^{constructor function}이다. 이를 Hugs에서 다음과 같이 확인할 수 있다.

```
> : type Circle
Circle :: Float → Shape

> : type Rect
Rect :: Float → Float → Shape
```

생성자 함수는 보통 함수와 달리 등식으로 정의하는 것이 아니라 단지 그 자체로 데이터를 만드는 용도로만 쓰인다. 예컨대, 식 *negate* 1.0은 *negate*의 정의를 적용해 -1.0 이라는 값을 구할 수 있는 반면, 식 *Circle* 1.0은 *Circle*을 정의하는 등식이 없으므로 이미 계산이 끝난 상태라 더 이상 간단히 할 수 없다. 1.0이 단순히 데이터인 것과 마찬가지로 식 *Circle* 1.0도 단순히 데이터일 뿐이다.

데이터 선언에도 당연히 타입 인자를 쓸 수 있다. 예컨대, 표준 라이브러리에서 다음과 같은 타입을 정의한다.

```
data Maybe a = Nothing | Just a
```

이는 *Maybe a* 타입이 *Nothing* 또는 *a*가 *a* 타입일 때 *Just x* 형태의 값을 갖는다는 것을 나타낸다. *Maybe a* 타입이란 *a* 타입의 값을 구하다 실패할지도 모르는 것으로 생각할 수 있다. *Just*는 값을 구하는 데 성공한 경우를 나타내고 *Nothing*은 실패한 경우를 나타낸다. 예컨대, 이 타입을 써서 라이브러리 함수 *div*와 *head*의 안전한 판, 곧 대응되는 값이 없는 인자를 받았을 때 잘못을 내는 대신 *Nothing*을 돌려주는 함수를 각각 다음과 같이 정의할 수 있다.

```
safediv      :: Int → Int → Maybe Int
safediv _ 0  = Nothing
safediv m n = Just (m `div` n)
safehead     :: [a] → Maybe a
safehead []  = Nothing
safehead xs = Just (head xs)
```

10.3 되도는 타입

data 예약어로 되도는 타입을 선언할 수 있다. 간단한 첫 보기로서, 다음 되도는 타입을 살펴보자.

```
data Nat = Zero | Succ Nat
```

이는 *Nat* 타입이 *Zero* 또는 *n* 이 *Nat* 타입의 값일 때 *Succ n* 형태의 값을 갖는다는 것을 나타낸다. 따라서 이 선언으로 *Zero*로부터 시작해 바로 전의 값에 생성자 함수 *Succ*를 계속 적용해 나감으로써 다음과 같이 무한히 많은 일련의 값을 만들어 나갈 수 있다.

```
Zero
Succ Zero
Succ (Succ Zero)
Succ (Succ (Succ Zero))
...
...
```

타입의 이름으로부터도 짐작할 수 있듯, *Nat* 타입의 값들은 자연수(음이 아닌 정수)를 나타내는 것으로 생각할 수 있다. *Zero*는 자연수 0를 나타내며, *Succ*는 그런 수에 1+을 해서 바로 다음 수를 구하는 함수를 나타낸다. 예컨대, *Succ (Succ (Succ Zero))*는 $1 + (1 + (1 + 0)) = 3$ 를 나타낸다. 더 형식적으로는, 다음과 같은 변환 함수를 정의할 수 있다.

```
nat2int      :: Nat → Int
nat2int Zero    = 0
nat2int (Succ n) = 1 + nat2int n

int2nat      :: Int → Nat
int2nat 0      = Zero
int2nat (n + 1) = Succ (int2nat n)
```

예컨대, 이 함수들로 두 자연수의 합을 구하는 함수를 작성할 수 있는데, 다음과 같이 두 수를 우선 정수로 변환해 더한 결과를 다시 자연수로 변환하면 된다.

```
add      :: Nat → Nat → Nat
add m n = int2nat (nat2int m + nat2int n)
```

하지만, 그런 변환 함수 없이도 *add* 를 되돌기로 다음과 같이 정의할 수 있으며, 변환 과정이 없으므로 더 효율적이다.

```
add Zero    n = n
add (Succ m) n = Succ (add m n)
```

이 정의는 두 자연수를 더하려면 첫째 수에서 *Succ* 생성자가 다 없어질 때까지 복사한 다음 둘째 수를 *Zero* 위치에 바꿔 넣으면 된다는 생각을 엄밀하게 식으로서 나타낸 것이다. 예컨대, $2 + 1 = 3$ 의 계산 과정은 다음과 같다.

$$\begin{aligned} & \text{add } (\text{Succ } (\text{Succ } \text{Zero})) \text{ (Succ Zero)} \\ = & \quad \{ \text{add 를 적용 } \} \\ & \text{Succ } (\text{add } (\text{Succ } \text{Zero}) \text{ (Succ Zero)}) \\ = & \quad \{ \text{add 를 적용 } \} \\ & \text{Succ } (\text{Succ } (\text{add Zero } (\text{Succ Zero}))) \\ = & \quad \{ \text{add 를 적용 } \} \\ & \text{Succ } (\text{Succ } (\text{Succ Zero})) \end{aligned}$$

또 다른 보기로서, **data** 선언으로 하스켈에서 제공하는 리스트 타입과 마찬가지로 임의의 타입을 인자로 받을 수 있는 그런 리스트 타입을 다음과 같이 직접 만들어 볼 수 있다.

```
data List a = Nil | Cons a (List a)
```

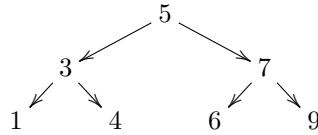
이는 *List a* 타입이 빈 리스트를 나타내는 *Nil* 또는 어떤 원소 $x :: a$ 와 리스트 $xs :: List a$ 가 주어졌을 때 비어 있지 않은 리스트를 나타내는 *Cons x xs* 형태의 값을 갖는다는 뜻이다. 원래 하스켈에서 제공하는 리스트를 다루는 라이브러리 함수를 우리가 정의한 위 타입에 대해 직접 만들어 볼 수 있다. 예컨대, 다음은 리스트의 길이를 구하는 함수 정의다.

```

len          :: List a → Int
len Nil      = 0
len (Cons _ xs) = 1 + len xs

```

리스트는 컴퓨터 작업에서 가장 많이 쓰는 데이터 구조이긴 하지만, 데이터를 두 갈래로 뻗는 구조, 곧 아래 그림과 같은 두갈래나무^{binary tree}에 저장하는 것이 쓸모있을 때가 많다.



이 보기에서 숫자 1, 4, 6, 9는 나무 가장자리의 이파리^{leaf}에, 숫자 5, 3, 7은 안쪽 마디^{node}에 나타난다. 이런 나무를 나타내기에 알맞은 타입을 다음과 같이 되돌기로 선언할 수 있다.

```
data Tree = Leaf Int | Node Tree Int Tree
```

정의를 간단히 하고자 나무에 들어가는 값을 정수로 고정했지만, 이 정의를 타입 인자를 갖도록 쉽게 일반화할 수 있다. 이 타입을 써서 위 그림의 나무를 다음과 같이 나타낼 수 있다.

```

t :: Tree
t = Node (Node (Leaf 1) 3 (Leaf 4)) 5 (Node (Leaf 6) 7 (Leaf 9))

```

이제 이런 나무에 대한 함수를 몇 가지 살펴보자. 우선 첫째로, 어떤 수가 나무에 나타나는지 알아보는 함수를 다음과 같이 정의한다.

```

occurs          :: Int → Tree → Bool
occurs m (Leaf n) = m == n
occurs m (Node l n r) = m == n ∨ occurs m l ∨ occurs m r

```

이는 나무에 정수가 나타난다는 것은 잎의 경우는 잎에 있는 값과 정수가 일치할 때이며, 마디의 경우는 마디에 있는 값과 같거나 왼쪽 부분나무에 나타나거나 오른쪽 부분나무에 나타남을 뜻한다. 느긋한 계산법으로는, 마디의 경우에 첫째나 둘째 조건이 참이면 나머지 조건을 계산할 필요도 없이 그 결과로 *True*를 돌려준다는 것에 주목하라. 하지만 최악의 경우에는 나무 전체를 다 돌아야 한다. 구체적으로는, 주어진 정수가 나무에 나타나지 않을 때 그러하다.

이번에는 나무를 리스트로 펼치는 함수를 살펴보자.

$$\begin{aligned} flatten &:: Tree \rightarrow [Int] \\ flatten(Leaf\ n) &= [n] \\ flatten(Node\ l\ n\ r) &= flatten\ l + [n] + flatten\ r \end{aligned}$$

이 함수를 나무에 적용한 결과가 정렬된 리스트일 때, 그 나무를 검색 나무^{search tree}라 부른다. 예컨대, 앞서 보기로 살펴본 나무가 바로 검색 나무이며, 이를 다음과 같이 확인해 볼 수 있다.

$$flatten\ t = [1, 3, 4, 5, 6, 7, 9]$$

검색 나무는, 주어진 정수가 나무에 나타나는지 알아볼 때, 마디에서 두 부분나무 중 어느 나무에 그 정수가 나타날 수 있는지 항상 알 수 있다는 중요한 성질이 있다. 더 자세히 설명하자면, 주어진 정수가 마디에 나타나는 값보다 작다면 왼쪽 부분나무에 나타날 수 있고, 마디에 나타나는 값보다 크다면 오른쪽 부분나무에 나타날 수 있다. 따라서 검색 나무에 대해서는 *occurs* 함수를 다음과 같이 고쳐 쓸 수 있다.

$$\begin{aligned} occurs\ m(Leaf\ n) &= m == n \\ occurs\ m(Node\ l\ n\ r) &\\ | m == n &= True \\ | m < n &= occurs\ m\ l \\ | otherwise &= occurs\ m\ r \end{aligned}$$

이 정의는 원래 정의보다 더 효율적이다. 왜냐하면 두 갈래 중 한쪽 갈래로만 찾아나가므로 전체 나무를 다 뒤져야 할 가능성성이 없기 때문이다.

컴퓨터 과학에서는 다양한 문제를 다루므로, 자연히 다양한 형태의 나무를 쓴다는 것을 이야기하며 이 절을 마무리하고자 한다. 예컨대, 이파리에만 데이터가 있는 나무, 마디에만 데이터가 있는 나무, (앞서 살펴본 것처럼) 이파리와 마디에 모두 데이터가 있는 나무, 또 마디에 고정된 개수가 아닌 여러개의 부분나무 리스트를 갖는 나무 타입 등을 다음과 같이 정의할 수 있다.

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
data Tree a = Leaf | Node (Tree a) a (Tree a)
data Tree a b = Leaf a | Node (Tree a b) b (Tree a b)
data Tree a = Node a [Tree a]
```

어떤 형태의 나무가 가장 알맞은지는 상황에 따라 다르다. 마지막 보기에서는 마디의 부분나무 리스트가 빈 경우가 이파리의 역할을 대신할 수 있으므로 이파리를 만드는 생성자가 없다는 것에 주목하라.

10.4 늘 참tautology 검사기

실제로 나무 타입을 어떤 종류의 언어를 나타내는 데 쓰는 경우가 많다. 이 장에서는 그러한 타입과 관련한 더 큰 보기로서 논리 명제logical proposition 가 늘 참tautology 인지 알아보는 함수를 만드는 과정을 살펴보겠다.

명제를 나타내는 언어를 생각해 보자. 논리값(False, True) 및 변수(A, B, \dots, Z)로부터 시작해, 논리 부정negation (\neg), 논리곱conjunction (\wedge), 조건implication (\Rightarrow), 그리고 괄호를 써서 더 큰 명제를 만들어 나갈 수 있다. 예컨대, 다음은 모두 명제다.

$$\begin{aligned} & A \wedge \neg A \\ & (A \wedge B) \Rightarrow A \\ & A \Rightarrow (A \wedge B) \\ & (A \wedge (A \Rightarrow B)) \Rightarrow B \end{aligned}$$

논리 연산자logical operator 가 뜻하는 바를 다음과 같은 진리표truth table 로 정의할 수 있다. 진리표에는 가능한 모든 인자의 조합에 대한 결과값이 나와 있다.

		A	B	$A \wedge B$	A	B	$A \Rightarrow B$
		F	F	F	F	F	T
		F	T	F	F	T	T
		T	F	F	T	F	F
		T	T	T	T	T	T

예컨대, 논리곱을 정의하는 진리표는 $A \wedge B$ 의 값이 A 와 B 가 모두 *True* 일 때만 *True*이고 그렇지 않으면 *False*임을 나타낸다. (이런 진리표가 차지하는 지면을 아끼고자 논리값을 F 와 T 로 줄여 썼다.) 위의 정의를 바탕으로 임의의 식에 대한 진리표를 만들 수 있다. 예컨대, 앞서 보기로 살펴본 네 명제에 대한 진리표는 다음과 같다.

		A	B	$(A \wedge B) \Rightarrow A$
		F	F	T
		F	T	T
		T	F	T
		T	T	T

		A	B	$(A \wedge (A \Rightarrow B)) \Rightarrow B$
		F	F	T
		F	T	T
		T	F	T
		T	T	T

위 진리표들 중 둘째와 넷째 명제는 항상 그 값이 *True* 이므로 늘 참이지만, 첫째와 셋째 명제는 적어도 한 경우에는 그 값이 *False* 이므로 늘 참이 아니다.

명제가 늘 참인지 알아보는 함수를 만들기 위해 가장 먼저 할 일은, 명제를 나타내는 타입을 선언하는 것이다. 가능한 다섯 가지 명제의 형태에 대해 생성자가 각각 하나씩 있도록 명제 타입을 다음과 같이 선언할 수 있다.

```
data Prop = Const Bool
| Var Char
| Not Prop
| And Prop Prop
| Imply Prop Prop
```

생성자를 선언하는 데 따로 괄호를 치지 않아도 된다는 것에 주목하라. 한 묶음으로 나타내야 할 때는 하스켈에서 쓰는 보통의 괄호를 쓰면 된다. 예컨대, 앞의 보기에서 있는 네 명제를 다음과 같이 나타낼 수 있다.

```
p1 :: Prop
p1 = And (Var 'A') (Not (Var 'B'))
p2 :: Prop
p2 = Imply (And (Var 'A') (Var 'B')) (Var 'A')
p3 :: Prop
p3 = Imply (Var 'A') (And (Var 'A') (Var 'B'))
p4 :: Prop
p4 = Imply (And (Var 'A') (Imply (Var 'A') (Var 'B'))) (Var 'B')
```

명제의 논리값을 계산하려면, 각각의 변수가 어떤 값인지 알아야 한다. 각각의 변수를 그 값으로 바꿔치기^{substitution} 위해서, 이 장의 앞부분에서 소개한 *Assoc* 타입으로 변수 이름에 논리값을 엮어 놓은 참조표를 쓰겠다.

```
type Subst = Assoc Char Bool
```

예컨대, 바꿔치기표^{substitution}(또는 바꿔치기 참조표) $[('A', False), ('B', True)]$ 는 A 를 $False$ 로, B 를 $True$ 로 바꾼다는 뜻이다. 이제 주어진 바꿔치기표에 따라 명제의 값을 구하는 함수를 다섯 가지 형태의 명제에 대한 패턴 매칭으로 다음과 같이 정의할 수 있다.

```
eval          :: Subst → Prop → Bool
eval _ (Const b) = b
eval s (Var x)  = find x s
eval s (Not p)   = ¬ (eval s p)
eval s (And p q) = eval s p ∧ eval s q
eval s (Imply p q) = eval s p ≤ eval s q
```

예컨대, 상수 (논리값) 명제의 값은 상수 (논리값) 그 자체이며, 변수 명제의 값은 바꿔치기표를 찾아봄으로써 얻을 수 있고, 논리곱 명제의 값은 인자로 오는 두 명제의 값을 구해 그 논리곱을 취함으로써 얻을 수 있다. 논리 조건 연산자 \Rightarrow 는 논리값에 대한 크기 비교 연산자 \leq 로써 정의한 것에 주목하라.

주어진 명제가 늘 참인지 알아보기 위해서 명제에 나타나는 변수들에 대한 모든 가능한 바꿔치기표를 만들고자 한다. 우선, 명제에 나타나는 변수를 모두 찾는 다음과 같은 함수를 정의한다.

```
vars      :: Prop → [Char]
vars (Const _) = []
vars (Var x)  = [x]
vars (Not p)   = vars p
vars (And p q) = vars p ++ vars q
vars (Imply p q) = vars p ++ vars q
```

예컨대, $\text{vars } p \vartheta = [\text{'A'}, \text{'B'}, \text{'A'}]$ 이다. 이 함수는 중복된 변수를 없애지 않음에 주목하라. 중복을 없애는 것은 나중에 따로 할 것이다.

바꿔치기표를 만들어 내기 위해 해야 할 중요한 일은 바로 주어진 길이를 갖는 모든 논리값 리스트를 만들어내는 것이다. 이러한 일을 하는 $\text{bools} :: \text{Int} \rightarrow [[\text{Bool}]]$ 함수를 어떻게 정의하면 좋을지 알아보자. 예컨대, bools 함수는 다음과 같이 3 개의 논리값이 가질 수 있는 가능한 모든 경우인 8 개의 리스트를 돌려준다.

```
> bools 3
[[False, False, False],
 [False, False, True],
 [False, True, False],
 [False, True, True],
 [True, False, False],
 [True, False, True],
 [True, True, False],
 [True, True, True]]
```

리스트들이 만들어지는 순서는 중요하지 않다. 이런 일을 하는 함수를 만드는 한 가지 방법은 각각의 리스트를 이진수에 대응시키는 것이다. 즉, 0과 1을 각각 False 와 True 로 해석한다. 예컨대, 리스트 $[\text{True}, \text{False}, \text{True}]$ 는 이진수 101에 해당한다.

이렇게 해석하면 *bools*는 그저 0부터 알맞은 범위까지 이진수를 헤아려 올라가는 함수일 뿐이다. 위와 같은 생각을 바탕으로 *bools*를 다음과 같이, 7장에서 음이 아닌 정수를 비트 리스트로 표현된 이진수로 변환하는 데 쓰인 *int2bin* :: *Int* → [*Bin*] 함수를 이용해 정의할 수 있다.

```
bools    :: Int → [[Bool]]
bools n = map (map conv ∘ make n ∘ int2bin) [0..limit]
where
  limit = (2↑n) - 1
  make n bs = take n (bs ++ repeat 0)
  conv 0 = False
  conv 1 = True
```

하지만 구하고자 하는 리스트들이 갖는 구조에 대해 잘 생각해 보면 *bools*를 더 간단히 정의할 수 있는 방법이 있다. 예컨대, *bools 3*을 잘 관찰해 보면, *bools 2*가 두 벌 나타남을 알 수 있는데, 한 벌에는 *False*가 그 앞에 모두 붙어 있고 다른 한 벌에는 *True*가 그 앞에 모두 붙어 있다.

<i>False</i>	<i>False</i>	<i>False</i>
<i>False</i>	<i>False</i>	<i>True</i>
<i>False</i>	<i>True</i>	<i>False</i>
<i>False</i>	<i>True</i>	<i>True</i>
<i>True</i>	<i>False</i>	<i>False</i>
<i>True</i>	<i>False</i>	<i>True</i>
<i>True</i>	<i>True</i>	<i>False</i>
<i>True</i>	<i>True</i>	<i>True</i>

이러한 규칙을 알고 나면, *bools*를 되돌기로 정의하는 것이 자연스럽다. 밑바탕 경우인 *bools 0*의 경우에는 0 개의 논리값을 갖는 모든 리스트, 곧 빈 리스트 한 개만 돌려주면 된다. 되도는 경우인 *bools (n+1)*의 경우에는 *bools n* 가 만들어내는 리스트를 두 벌 취해, 첫째 한 벌에는 *False*를 그 앞에 모두 붙이고, 둘째 한 벌에는 *True*를 그 앞에 모두 붙인 다음, 이어붙이면 된다.

```

bools      :: Int → [[Bool]]
bools 0    = [[]]
bools (n + 1) = map (False:) bss ++ map (True:) bss
                where bss = bools n
```

이제 *bools* 를 써서 주어진 명제에 대한 모든 바꿔치기표를 만들어 내는 함수를 정의하는 것은 간단하다. 다음과 같이 명제에 나타나는 모든 변수를 찾아 변수 리스트를 만들고 거기서 중복된 변수가 없도록 한 다음 (중복된 변수를 없애는 데는 9장의 *rmdups* 함수를 쓴다), 변수 개수만큼의 논리값이 가질 수 있는 가능한 모든 조합의 리스트를 만들어 그것을 변수 리스트와 맞물리게 여미면 *zip* 된다.

```

substs   :: Prop → [Subst]
substs p = map (zip vs) (bools (length vs))
           where vs = rmdups (vars p)
```

예컨대,

```

> substs p2
[[(‘A’, False), (‘B’, False)],
 [(‘A’, False), (‘B’, True)],
 [(‘A’, True), (‘B’, False)],
 [(‘A’, True), (‘B’, True)]]
```

마지막으로, 주어진 명제가 늘 참인지 알아보는 함수를, 다음과 같이 모든 가능한 바꿔치기표에 대해 주어진 명제가 참인지 검사함으로써 정의한다.

```

isTaut  :: Prop → Bool
isTaut p = and [eval s p | s ← substs p]
```

예컨대,

```
> isTaut p1
False
```

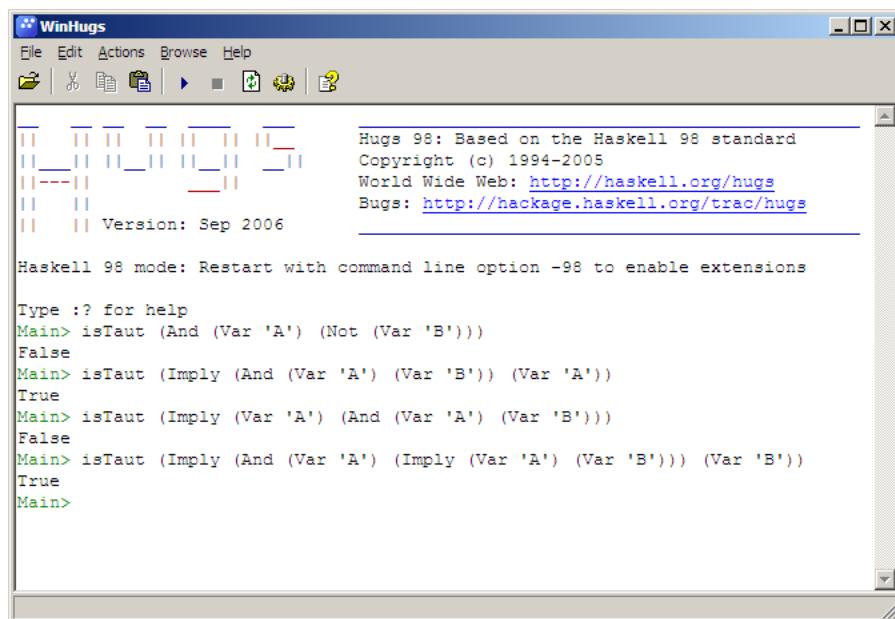
```
> isTaut p2
True
```

```
> isTaut p3
False
```

```
> isTaut p4
True
```

따라하기

다음은 *tautology.lhs* 스크립트를 불러와 실행해 본 화면이다.



The screenshot shows the WinHugs Haskell interpreter window. The title bar says "WinHugs". The menu bar includes "File", "Edit", "Actions", "Browse", and "Help". Below the menu is a toolbar with icons for file operations. The main window has two panes. The left pane contains the Haskell code being run:

```
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-2005
World Wide Web: http://haskell.org/hugs
Bugs: http://hackage.haskell.org/trac/hugs
Version: Sep 2006

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type ?: for help
Main> isTaut (And (Var 'A') (Not (Var 'B')))
False
Main> isTaut (Imply (And (Var 'A') (Var 'B')) (Var 'A'))
True
Main> isTaut (Imply (Var 'A') (And (Var 'A') (Var 'B')))
False
Main> isTaut (Imply (And (Var 'A') (Imply (Var 'A') (Var 'B')))) (Var 'B')
True
Main>
```

위에서부터 차례대로 책에 나온 $p1, p2, p3, p4$ 가 늘 참인지 알아본 것이다.

따라하기 끝

스크립트 *tautology.lhs* 한눈에 다시보기

```

1 Programming in Haskell 10.4절 늘 참 검사기 보기,
2 Graham Hutton, Cambridge University Press, 2007.
3
4
5 늘 참 검사기
6 -----
7
8 > data Prop          = Const Bool
9   | Var Char
10  | Not Prop
11  | And Prop Prop
12  | Imply Prop Prop
13
14 > type Subst        = Assoc Char Bool
15
16 > type Assoc k v    = [(k,v)]
17
18 > find               :: Eq k => k -> Assoc k v -> v
19 > find k t            = head [v | (k',v) <- t, k == k']
20
21 > eval                :: Subst -> Prop -> Bool
22 > eval _ (Const b)     = b
23 > eval s (Var x)       = find x s
24 > eval s (Not p)       = not (eval s p)
25 > eval s (And p q)     = eval s p && eval s q
26 > eval s (Imply p q)   = eval s p <= eval s q
27
28 > vars                :: Prop -> [Char]
29 > vars (Const _)       = []
30 > vars (Var x)         = [x]
31 > vars (Not p)         = vars p
32 > vars (And p q)       = vars p ++ vars q
33 > vars (Imply p q)     = vars p ++ vars q
34
35 > bools               :: Int -> [[Bool]]
36 > bools 0              = [[]]
37 > bools (n+1)          = map (False:) bss ++ map (True:) bss
38 >                         where bss = bools n
39
40 > rmdups               :: Eq a => [a] -> [a]
41 > rmdups []             = []
42 > rmdups (x:xs)         = x : rmdups (filter (/= x) xs)
43
44 > substs               :: Prop -> [Subst]
45 > substs p              = map (zip vs) (bools (length vs))
46 >                         where vs = rmdups (vars p)
47
48 > isTaut                :: Prop -> Bool
49 > isTaut p              = and [eval s p | s <- substs p]
50

```

*tautology.lhs*에서 눈여겨 볼 점들은 다음과 같다.

- 줄번호 8-13에서는 되도는 타입인 *Prop*을 *data* 예약어를 써서 선언했다. *Prop*이 되도는 타입이라는 것은 줄번호 10,11,13의 *Not*, *And*, *Imply* 생성자의 인자 타입에 *Prop*이 나타나는 것으로 보면 알 수 있다.

2. 줄번호 14에서 **type** 예약어로 정의한 *Subst*는 줄번호 10에서 정의한 *Assoc k v* 타입을 이용한다.
3. 줄번호 16에서 **type**으로 정의한 *Assoc k v*는 다른 타입을 *k, v*의 위치에 인자로 받는 타입이다. 이 때 *Assoc*을 타입을 만들어낸다는 뜻에서 타입 생성자**type constructor**라고 부르기도 한다.

10.5 추상 기계 abstract machine

두 번째의 더 큰 보기로서, 정수를 바탕으로 덧셈 연산자를 써서 만들어 나가는 간단한 산술식 타입과 함께, 그런 식을 계산해 정수값을 구하는 다음과 같은 함수를 생각해 보자.

```
data Expr = Val Int | Add Expr Expr
value      :: Expr → Int
value (Val n) = n
value (Add x y) = value x + value y
```

예컨대, 식 $(2 + 3) + 4$ 를 계산하면 다음과 같다.

$$\begin{aligned}
 & \text{value (Add (Add (Val 2) (Val 3)) (Val 4))} \\
 = & \quad \{ \text{value 를 적용 } \} \\
 = & \quad \text{value (Add (Val 2) (Val 3))} + \text{value (Val 4)} \\
 = & \quad \{ \text{첫째 value 를 적용 } \} \\
 & \quad (\text{value (Val 2)} + \text{value (Val 3)}) + \text{value (Val 4)} \\
 = & \quad \{ \text{첫째 value 를 적용 } \} \\
 & \quad (2 + \text{value (Val 3)}) + \text{value (Val 4)} \\
 = & \quad \{ \text{첫째 value 를 적용 } \} \\
 & \quad (2 + 3) + \text{value (Val 4)} \\
 = & \quad \{ \text{첫째 + 를 적용 } \} \\
 & \quad 5 + \text{value (Val 4)} \\
 = & \quad \{ \text{value 를 적용 } \} \\
 & \quad 5 + 4 \\
 = & \quad \{ \text{+ 를 적용 } \} \\
 9
 \end{aligned}$$

value 함수 정의에는, 덧셈식의 왼쪽 인자를 오른쪽 인자보다 먼저 계산해야 한다거나, 또는 더 일반적으로, 계산 과정 중 어느 시점에서도 그 다음 걸음에 무엇을 해야 할지 정해 놓지 않았다. 다만, 계산 순서를 하스켈에서 하는 대로 맡기고 있을 때이다. 하지만, 필요하다면 식을 계산하는 추상 기계로써 그러한 계산의 흐름을 드러나 보이게 정의할 수 있다. 이러한 추상 기계는 식의 값을 구하는 과정을 한 걸음 한 걸음 조목조목 드러낸다.

이를 위해, 추상 기계가 지금 하고 있는 계산을 끝마친 다음 수행할 일련의 연산으로 이루어지는 (흐름) 제어 스택 control stack 탑을 먼저 다음과 같이 정의하자:

```
type Cont = [Op]
data Op = EVAL Expr | ADD Int
```

두 연산이 각각 뜻하는 바는 곧 설명하겠다. 이제 주어진 제어 스택을 놓고 식의 값을 구하는 함수를 다음과 같이 정의하자.

```
eval          :: Expr → Cont → Int
eval (Val n) c = exec c n
eval (Add x y) c = eval x (EVAL y : c)
```

즉, 정수식이라면 이미 값을 완전히 구했으므로 제어 스택을 실행하면 된다. 덧셈식이라면, $EVAL y$ 연산을 원래 제어 스택의 맨 위에 올려놓고 첫째 인자 x 의 값을 구하는데, $EVAL y$ 연산을 스택의 맨 위에 올려놓는 것은 첫째 인자의 값을 구한 다음 둘째 인자 y 의 값을 구해야 한다는 것을 나타내기 위함이다. 다음으로는, 주어진 정수 인자를 놓고 제어 스택을 실행하는 함수를 아래와 같이 정의하자.

```
exec          :: Cont → Int → Int
exec [] n     = n
exec (EVAL y : c) n = eval y (ADD n : c)
exec (ADD n : c) m = exec c (n + m)
```

즉, 빈 스택이면 주어진 정수 인자를 그대로 실행 결과로 돌려준다. 스택의 맨 위에 $EVAL y$ 연산이 나타나면, $ADD n$ 연산을 나머지 스택의 맨 위에 올려놓고 식 y 의 값을 구하는데, $ADD n$ 을 나머지 스택의 맨 위에 올려놓는 것은 식 y 의 값을 구한 다음 지금 주어진 정수 인자 n 과 합을 구해야 한다는 것을 나타내기 위함이다. 그리고, 스택의 맨 위에 $ADD n$ 연산이 나타나면, 덧셈식의 두 인자값을 다 구한 것이므로, 두 정수의 합을 새로이 정수 인자로 주고 나머지 스택을 마저 실행하면 된다.

마지막으로, 빈 제어 스택을 놓고 *eval* 함수를 써서 주어진 식의 값을 정수로 구하는 함수를 다음과 같이 정의한다.

$$\begin{aligned} \text{value } e :: \text{Expr} &\rightarrow \text{Int} \\ \text{value } e &= \text{eval } e [] \end{aligned}$$

추상 기계를 두 개의 서로 되도는 함수 *eval*과 *exec*로써 정의했는데, 이는 추상 기계가 두 가지 상태, 곧 식의 구조 및 제어 스택에 따라 계산의 진행 과정이 결정된다는 것을 반영한다. 식 $(2 + 3) + 4$ 의 값을 구하는 아래 보기지를 통해 이 추상 기계가 어떻게 동작하는지 볼 수 있다.

$$\begin{aligned} &\text{value } (\text{Add } (\text{Add } (\text{Val } 2) (\text{Val } 3)) (\text{Val } 4)) \\ &= \quad \{ \text{ value 를 적용 } \} \\ &\quad \text{eval } (\text{Add } (\text{Add } (\text{Val } 2) (\text{Val } 3)) (\text{Val } 4)) [] \\ &= \quad \{ \text{ eval 을 적용 } \} \\ &\quad \text{eval } (\text{Add } (\text{Val } 2) (\text{Val } 3)) [\text{EVAL } (\text{Val } 4)] \\ &= \quad \{ \text{ eval 을 적용 } \} \\ &\quad \text{eval } (\text{Val } 2) [\text{EVAL } (\text{Val } 3), \text{EVAL } (\text{Val } 4)] \\ &= \quad \{ \text{ eval 을 적용 } \} \\ &\quad \text{exec } [\text{EVAL } (\text{Val } 3), \text{EVAL } (\text{Val } 4)] 2 \\ &= \quad \{ \text{ exec 를 적용 } \} \\ &\quad \text{eval } (\text{Val } 3) [\text{ADD } 2, \text{EVAL } (\text{Val } 4)] \\ &= \quad \{ \text{ eval 을 적용 } \} \\ &\quad \text{exec } [\text{ADD } 2, \text{EVAL } (\text{Val } 4)] 3 \\ &= \quad \{ \text{ exec 를 적용 } \} \\ &\quad \text{exec } [\text{EVAL } (\text{Val } 4)] 5 \\ &= \quad \{ \text{ exec 를 적용 } \} \\ &\quad \text{eval } (\text{Val } 4) [\text{ADD } 5] \\ &= \quad \{ \text{ eval 을 적용 } \} \\ &\quad \text{exec } [\text{ADD } 5] 4 \\ &= \quad \{ \text{ exec 를 적용 } \} \\ &\quad \text{exec } [] 9 \\ &= \quad \{ \text{ exec 를 적용 } \} \\ &\quad 9 \end{aligned}$$

eval 함수가 식의 가장 왼쪽에 있는 정수를 찾아내려가며, 앞으로 계산해야 할 식의 오른쪽 부분에 대한 발자취를 제어 스택에 남기고 있는 것에 주목하라. 그 다음에는, *exec* 함수가 그 발자취를 거슬러 올라가 흐름 제어를 다시 *eval* 함수에 넘겨주어 덧셈을 알맞게 수행할 수 있도록 하고 있다.

스크립트 *machine.lhs* 한눈에 다시보기

```

1 Programming in Haskell 10.5절 추상 기계 보기,
2 Graham Hutton, Cambridge University Press, 2007.
3
4 추상 기계
5 -----
6
7 > data Expr          = Val Int | Add Expr Expr
8 >
9 > type Cont          = [Op]
10 >
11 > data Op             = EVAL Expr | ADD Int
12 >
13 > eval                :: Expr -> Cont -> Int
14 > eval (Val n) c       = exec c n
15 > eval (Add x y) c     = eval x (EVAL y : c)
16 >
17 > exec                :: Cont -> Int -> Int
18 > exec []              n = n
19 > exec (EVAL y : c) n   = eval y (ADD n : c)
20 > exec (ADD n : c) m   = exec c (n+m)
21 >
22 > value                :: Expr -> Int
23 > value e               = eval e []
24

```

1. 줄번호 9에서 `type` 예약어로 정의한 *Cont*는 앞으로 수행할 연산들을 나타내는 리스트 타입인 `[Op]`의 별명(다른 이름)이다.
2. 줄번호 9의 *Cont* 타입 정의에서 줄번호 11에서 정의하는 *Op*를 이용한다. 이와 같이 하스켈에서는 함수를 정의할 때 뒤에 정의하는 함수를 이용할 수 있는 것과 마찬가지로 타입을 정의할 때 뒤에 정의하는 타입을 이용할 수 있다.

10.6 클래스와 인스턴스 선언

지금까지 타입 선언에 대해 알아보았으니, 이제 클래스에 대해 알아보기로 하자. 하스켈에서는 **class** 키워드를 써서 새로운 클래스를 선언할 수 있다. 예컨대, 같기 타입^{equality type}을 나타내는 *Eq* 클래스를 표준 라이브러리에서 다음과 같이 정의한다.

```
class Eq a where
  (==), (≠) :: a → a → Bool
  x ≠ y      = ¬ (x == y)
```

이 클래스 선언이 나타내는 바는, 어떤 타입이 *Eq* 클래스의 인스턴스가 되기 위해서는 그 타입끼리 같은지^{equality}(==) 같지 않은지^{inequality}(≠) 알아보는 연산이 가능해야 한다는 것이다. 실제로는, 클래스 선언에 ≠ 가 없을 때를 위한 정의^{default definition} 가 이미 있기 때문에, == 만 정의해도 이 클래스의 인스턴스를 선언할 수 있다. 예컨대, 다음과 같이 *Bool* 타입을 같기 타입으로 만들 수 있다.

```
instance Eq Bool where
  False == False = True
  True == True = True
  _    == _    = False
```

기술적인 이유로, **data** 키워드로 정의한 타입들만 클래스 인스턴스로 선언할 수 있다. 인스턴스를 선언할 때 필요하다면 없을 때 정의^{default definition} 를 옆을^{override} 수 있음에 주목하라. 예컨대, 어떤 같기 타입은 두 값이 서로 다른지 알아보는 방법으로, 그냥 두 값이 같은지 비교한 다음 그것이 참이 아님을 검사하는 것보다 더 효율적이고 알맞은 방법이 있을 수 있다.

기존의 클래스에 메서드를 덧붙임으로써 새 클래스를 만들 수도 있다. 예컨대, 전순서를 이루는 값들의 타입을 분류하는 *Ord* 클래스를 표준 라이브러리에서 다음과 같이 *Eq* 클래스에 메서드를 덧붙여 정의한다.

```
class Eq a ⇒ Ord a where
  (<), (≤), (>) , (≥) :: a → a → Bool
  min, max :: a → a → a
  min x y | x ≤ y = x
            | otherwise = y
  max x y | x ≥ y = x
            | otherwise = y
```

이는, 어떤 타입이 *Ord*의 인스턴스라면 반드시 *Eq*의 인스턴스이기도 하며 추가로 여섯 개의 연산이 가능해야 한다는 뜻이다. *min*과 *max*는 없을때 정의가 이미 있기 때문에, 네 개의 비교 연산만 정의하면 (예컨대, 아래와 같이 *Bool*의 경우처럼) 같기 타입을 순서 타입^{ordered type}으로 선언할 수 있다.

```
instance Ord Bool where
  False < True = True
  _     < _     = False
  b     ≤ c     = (b < c) ∨ (b == c)
  b     > c     = c < b
  b     ≥ c     = c ≤ b
```

유도된^{derived} 인스턴스

새로이 선언하는 타입을 하스켈에서 기본적으로 제공하는 몇 가지 클래스의 인스턴스로 만드는 것이 적당한 경우가 많다. 하스켈에서는 **deriving**이라는 키워드로써 간단히 새로운 타입을 자동으로 *Eq*, *Ord*, *Show*, 및 *Read* 클래스의 인스턴스로 유도^{derive} 할 수 있다. 예컨대, *Bool* 타입을 실제로 표준 라이브러리에서는 다음과 같이 정의한다.

```
data Bool = False | True deriving (Eq, Ord, Show, Read)
```

위와 같이 선언하고 나면, 논리값에다 이 네 클래스의 모든 멤버 함수^{member function}를 적용할 수 있다. 예컨대,

```
> False == False
True

> False < True
True

> show False
"False"

> read "False" :: Bool
False
```

옮긴이 주: 멤버 함수^{member function} 는 메서드^{method} 와 같은 뜻으로 쓰는 낱말이다.

마지막 보기의 :: 를 써서 타입을 지정한 까닭은 문맥으로부터 결과값의 타입을 유추할 수 없기 때문이다. 순서 타입을 나타내는 *Ord* 클래스를 유도할 때는 타입 선언에 나타나는 순서 그대로 생성자들의 순서를 매긴다는 점에 주목하라. 즉, 위 *Bool* 타입의 선언에서 *False* 가 *True* 앞에 오므로, 그 순서는 *False < True* 로 나타난다.

생성자가 인자를 가지는 경우는, 그 인자 타입 또한 지금 유도하는 모든 클래스의 인스턴스여야만 한다. 예컨대, 이 장에서 앞서 선언한 다음 두 타입을 다시 살펴보자.

```
data Shape    = Circle Float | Rect Float Float
data Maybe a = Nothing | Just a
```

Shape 이 같기 타입이 되도록 유도하려면, *Float* 도 같기 타입이어야 하는데 실제로 그러하다. 마찬가지로, *Maybe a* 를 같기 타입이 되도록 유도하려면 인자 타입 *a* 도 역시 같기 타입이어야 하며, 이것이 바로 이 인자에 걸리는 클래스 제약이다. 리스트나 순서쌍의 경우와 마찬가지로, 생성자로 이루어진 값들은 기본적으로 사전식^{lexicographical} 순서를 따른다. 예컨대, *Shape* 을 순서 타입이 되도록 유도했다면 순서 비교가 다음과 같이 된다.

```
> Rect 1.0 4.0 < Rect 2.0 3.0
True
```

```
> Rect 1.0 4.0 < Rect 1.0 3.0
False
```

따라하기

다음은 *Shape* 을 정의하고 *Ord* 의 인스턴스로 유도하여 *deriving.hs*에 저장한 화면이다.

```

Km deriving.hs (K:WMyDoc) - GVIM
파일(F) 편집(E) 도구(I) 문법(S) 버퍼(B) 창(W) 도움말(H)
[File Edit Tools Syntax Buffer Window Help]
data Shape = Circle Float | Rect Float Float deriving (Eq, Ord)
[REDACTED]
"deriving.hs" 2L, 67C 저장 했습니다 2,0-1 모두

```

Shape 을 순서 타입을 나타내는 *Ord* 클래스의 인스턴스로 유도하려면 *Shape* 이 같기 타입을 나타내는 *Eq* 의 인스턴스이어야만 하므로 *Eq* 로도 같이 유도해야만 한다.

또, 다음과 같이 *Shape* 을 *Eq* 의 인스턴스로 따로 정의했다면 *Ord* 만 유도할 수 있으며, 이렇게 *Eq* 클래스의 인스턴스를 정의하면 바로 앞에서 *Eq* 와 *Ord* 를 동시에 유도한 것과 같은 효과다.

```

Km deriving.hs (K:WMyDoc) - GVIM
파일(F) 편집(E) 도구(I) 문법(S) 버퍼(B) 창(W) 도움말(H)
[File Edit Tools Syntax Buffer Window Help]
data Shape = Circle Float | Rect Float Float deriving Ord
instance Eq Shape where
    (Circle x) == (Circle y) = x == y
    (Rect a b) == (Rect c d) = a == c && b == d
    _ == _ = False
"deriving.hs" 6L, 206C 저장 했습니다 2,0-1 모두

```

다음은 *deriving.hs* 를 WinHugs에서 불러들여 실습한 결과다.

```

WinHugs
File Edit Actions Browse Help
Type :? for help
Main> :info Shape
-- type constructor
data Shape

-- constructors:
Circle :: Float -> Shape
Rect :: Float -> Float -> Shape

-- instances:
instance Eq Shape
instance Ord Shape

Main> Rect 1.0 4.0 < Rect 2.0 3.0
True
Main> Rect 1.0 4.0 < Rect 1.0 3.0
False
Main> Circle 4.0 < Rect 1.0 3.0
True
Main> Circle 4.0 < Circle 1.0
False
Main>

```

따라하기 끝

모나드 타입^{monadic type}

앞 두 장의 주제였던 문법 분석기^{parser} 와 대화식 프로그램^{interactive program} 을 다시금 살펴보는 것으로써 이 장을 맺겠다. 두 장에서 각각 *return* 과 $\gg=$ 함수를 정의했던 것이 기억날 것이다. 문법 분석기에서는 다음과 같이,

```
return :: a → Parser a
( $\gg=$ ) :: Parser a → (a → Parser b) → Parser b
```

그리고 대화식 프로그램에서는 다음과 같이 정의하였다.

```
return :: a → IO a
( $\gg=$ ) :: IO a → (a → IO b) → IO b
```

함수들의 이름이 같고 대응하는 함수들의 타입이 비슷한 것은 우연이 아니다. 더 구체적으로 설명하자면, *Parser* 와 *IO* 의 이러한 공통적인 속성을 임의의 다른 타입을 인자로 받는^{parameterised} 타입으로 일반화시키면 모나드^{monad} 라는 개념에 해당하는데, 이를 하스켈에서는 다음과 같은 클래스 선언으로 나타낼 수 있다.

```
class Monad m where
    return :: a → m a
    ( $\gg=$ ) :: m a → (a → m b) → m b
```

즉, 모나드란 다른 타입을 인자로 받는 타입으로서, 그러한 타입 *m* 에 위 클래스 선언에 나타난 대로의 타입을 갖는 함수 *return* 과 $\gg=$ 이 적용 가능해야 함을 뜻한다. *m* 이 그냥 타입이 아니라 다른 타입을 인자로 받는^{parameterised} 타입이라는 것은 두 멤버 함수의 타입으로부터 미루어 짐작할 수 있다. 이러한 클래스 선언을 바탕으로, 다음과 같이 각각의 경우에 알맞는 두 멤버 함수를 정의함으로써, 문법 분석기와 대화식 프로그램을 모나드 타입을 나타내는 클래스의 인스턴스로 만들 수 있다.

```
instance Monad Parser where
    return v = ...
    p ≫ f = ...

instance Monad IO where
    return v = ...
    f ≫ g = ...
```

이와 같은 인스턴스 선언을 함으로써, 문법 분석기와 대화식 프로그램을 순서대로 엮는 sequence 데 **do** 표현을 쓸 수 있게 된다. 더 일반적으로는, 하스켈에서 다음과 같은 형태의 임의의 모나드 타입의 식을

$$\begin{aligned} e1 &\gg \lambda v_1 \rightarrow \\ e2 &\gg \lambda v_2 \rightarrow \\ &\dots \\ e_n &\gg \lambda v_n \rightarrow \\ &return (f v_1 v_2 \dots v_n) \end{aligned}$$

아래와 같이 바꿔 쓸 수 있다.

$$\begin{aligned} \mathbf{do} \quad v_1 &\leftarrow e_1 \\ v_2 &\leftarrow e_2 \\ &\dots \\ v_n &\leftarrow e_n \\ &return (f v_1 v_2 \dots v_n) \end{aligned}$$

10.7 살펴보기

추상 기계 보기는 (19)에서 따왔으며, 추상 기계 보기에 쓰인 제어 스택 타입은 되도록 타입의 값을 가로지르며 돌아다니는 지퍼 zipper 라는 데이터 구조(12)의 한 특별한 경우이다. 모나드 monad 라는 낱말은 원래 카테고리 이론 category theory(23)이라는 수학의 한 분야에서 왔다. 함수형 프로그래밍에서 모나드 이론이 어떻게 나타나고 그것을 어떻게 응용하는지에 대한 더 자세한 내용을 (31;24)에서 찾아볼 수 있다. 이 장에서 살펴본 새로운 타입과 클래스를 선언하는 기본적인 방법 말고도, Hugs 시스템에서는 더 발전된 실험적인 타입 기능들도 선보이고 있다. 더 자세한 내용은 다음 홈페이지를 참조하라: <http://www.haskell.org/hugs>

10.8 연습문제

1. *add* 함수를 써서 자연수에 대한 곱셈 함수 $mult :: Nat \rightarrow Nat \rightarrow Nat$ 를 되돌기로 정의하라.
2. 부록 A 에는 실려 있지 않지만, 표준 라이브러리에서는 다음 타입을 정의하고 있으며

```
data Ordering = LT | EQ | GT
```

그와 함께 다음 함수도 정의하는데

```
compare :: Ord a ⇒ a → a → Ordering
```

이는 순서 타입의 한 값이 같은 타입의 다른 값보다 더 작은지 (LT), 같은지 (EQ), 아니면 더 큰지 (GT) 알아보는 함수다. 이 함수를 써서, 겹색 나무에 대한 $occurs :: Int \rightarrow Tree \rightarrow Bool$ 함수를 다시 정의해 보라. 왜 새로운 정의가 원래 것보다 더 효율적인가?

3. 다음은 두갈래나무 binary tree 를 나타내는 타입이다.

```
data Tree = Leaf Int | Node Tree Tree
```

왼쪽과 오른쪽 부분나무에 있는 이파리 개수가 하나 이상 차이나지 않을 때, 이런 나무를 균형잡혔다 말한다. 물론, 이파리는 그 자체로 균형잡힌 것으로 본다. 나무가 균형잡혔는지 알아보는 함수 $balanced :: Tree \rightarrow Bool$ 를 정의하라.

귀띔: 우선 나무에 잎이 몇 개 있는지 세는 함수를 정의하라.

4. 비어 있지 않은 정수 리스트로부터 균형잡힌 나무를 만들어내는 함수 $balance :: [Int] \rightarrow Tree$ 를 정의하라
귀띔: 우선 하나의 리스트를 두 개의 리스트로 반토막내는 함수부터 정의해 보라. 이때, 반토막난 두 리스트의 길이는 많아야 하나만큼 차이나도록 한다.
5. 늘 참 검사기가 논리합 logical disjunction (\vee) 과 동치 equivalence (\Leftrightarrow) 를 포함하는 명제를 처리할 수 있도록 고쳐 보라.

6. *isTaut* 함수와 함께 앞선 두 장에 있는 문법 분석기 및 대화식 프로그램을 작성하는 라이브러리를 써서, 사용자가 쓰기 편한 문법으로 명제를 글쇠판으로 입력할 수 있는 대화식 늘 참 검사기를 정의해 보라.

귀띔: 8장의 산술식 문법분석기를 고쳐 명제 문법분석기를 만들어 보라.

7. 추상 기계가 곱셈을 처리할 수 있도록 고쳐 보라.

8. 다음 인스턴스 선언을 마저 채워넣어 보라.

```
instance Monad Maybe where
```

...

```
instance Monad [] where
```

...

여기서 []는 [a]에서 타입 인자를 빼고 쓴 것으로 리스트 타입을 나타낸다.

귀띔: 우선 각 인스턴스에서 *return*과 $\gg=$ 의 타입부터 적어 보라.

제 11 장

카운트다운 문제 countdown problem

이 장에서는 주어진 조건에 맞는 식을 만드는 숫자 놀이인 카운트다운 문제를 하스켈로 어떻게 풀 수 있는지 알아본다. 먼저 놀이 규칙을 하스켈 프로그램의 식으로 나타내 보는 것으로 시작하여, 간단하게 문제를 풀긴 하지만 효율이 떨어지는 프로그램을 선보인 다음, 두 단계에 걸쳐 프로그램의 효율을 개선한다.

11.1 소개

영국 텔레비전에서 1982년부터 방영한 카운트다운이라는 인기있는 퀴즈 프로그램 진행 중에 바로 우리가 카운트다운 문제라고 부르는 숫자 게임을 한다. 이 문제의 핵심을 다음과 같이 정리할 수 있다.

여러 개의 수들과 목표로 하는 결과값이 주어지면, 주어진 여러 개의 수들로부터 원하는 숫자를 골라 덧셈, 뺄셈, 곱셈, 나눗셈 및 괄호를 써서 식을 만들어, 그 만들어진 식의 계산 결과가 목표로 하는 결과값과 같도록 해 보라.

주어진 숫자들은 많아야 한 번씩만 식에 나타나야 하며, 계산 중간값을 포함한 계산에 쓰이는 모든 수들은 반드시 양의 정수(1, 2, 3, ...)여야 한다. 음수나, 0, 또는 $2/3$ 와 같은 분수가 나와서는 안된다.

예컨대, 1, 3, 7, 10, 25, 50로부터 765를 결과값으로 계산하는 문제를 생각해 보자. 이 때, $(1 + 50) * (25 - 10)$ 가 이 문제의 답으로 가능한 식의 하나임을 다음과 같이 간단한 계산을 통해 확인할 수 있다.

$$\begin{aligned} & (1 + 50) * (25 - 10) \\ = & \quad \{ +\text{를 적용 } \} \\ & 51 * (25 - 10) \\ = & \quad \{ -\text{를 적용 } \} \\ & 51 * 15 \\ = & \quad \{ *\text{를 적용 } \} \\ & 765 \end{aligned}$$

사실, 보기로 든 이 문제는 780 개의 서로 다른 답이 있음을 보일 수 있다. 한편, 같은 숫자들로부터 831을 결과로 계산하는 문제는 답이 없음을 보일 수 있다.

텔레비전 퀴즈 프로그램에 나오는 카운트다운 문제는 사람들이 놀이하기에 알맞도록 하는 몇 가지 규칙이 더 있다. 그 규칙들을 구체적으로 말하자면, 항상 1...10, 1...10, 25, 50, 75, 100 범위에 있는 여섯 개의 숫자로부터 100...999 범위에 있는 결과값을 30초 안에 계산해야 한다. 컴퓨터로 이 문제를 풀 때는 그런 제약이 없는 것이 더 자연스럽기 때문에, 이 장에서 작성하는 프로그램에는 그러한 추가 규칙을 강제하거나 사용하지 않겠다. 하지만, 양의 정수만 나타나야 한다는 조건은 그대로 둔다는 것에 주목하라. 만일 자연수 전체나 유리수로 수의 범위를 확장한다면, 문제의 계산 복잡도 computational complexity 를 근본적으로 바꿔버리고 만다.

11.2 문제를 수식으로 정리하기

우선 네 가지 수치 연산을 나타내는 타입을 다음과 같이 정의하자.

```
data Op = Add | Sub | Mul | Div
```

이 타입을 써서 연산자를 두 양의 정수에 적용했을 때 그 결과도 양의 정수인지 알아보는 함수 *valid* 와 실제로 그러한 연산을 수행하는 함수 *apply* 를 다음과 같이 정의할 수 있다.

```
valid      :: Op → Int → Int → Bool
valid Add _ _ = True
valid Sub x y = x > y
valid Mul _ _ = True
valid Div x y = x `mod` y == 0

apply      :: Op → Int → Int → Int
apply Add x y = x + y
apply Sub x y = x - y
apply Mul x y = x * y
apply Div x y = x `div` y
```

예컨대, 적용식 *Sub 2 3* 는 $2 - 3$ 이 음수값을 가지므로 마땅치 않으며^{invalid}, *Div 2 3* 는 $2/3$ 가 유리수 값을 가지므로 마땅치 않다^{invalid}. 다음으로는, 수식을 나타내는 타입을 아래와 같이 정의하자. 수식은 하나의 정수값으로 또는 두 식에 연산을 적용함으로써 만들 수 있다.

```
data Expr = Val Int | App Op Expr Expr
```

이 타입을 써서 식이 가지는 값을 리스트로 돌려주되, 계산 결과가 양의 자연수일 때만 식의 값을 돌려주는 *eval* 함수를 다음과 같이 정의할 수 있다.

```

values      :: Expr → [Int]
values (Val n) = [n]
values (App o l r) = values l ++ values r

eval       :: Expr → [Int]
eval (Val n) = [n | n > 0]
eval (App o l r) = [apply o x y | x ← eval l,
                           y ← eval r,
                           valid o x y]

```

리스트를 돌려줌으로써 *eval* 함수가 값을 구하다가 실패하는 경우를 처리할 수 있음에 주목하라. 관례대로 한원소 리스트는 성공을 나타내며 빈 리스트는 실패를 나타낸다. 예컨대, $2 + 3$ 과 $2 - 3$ 의 값을 구하면 다음과 같다.

```
> eval (App Add (Val 2) (Val 3))
[5]
```

```
> eval (App Sub (Val 2) (Val 3))
[]
```

eval 이 실패하는 경우를 *Maybe* 타입으로 처리할 수도 있지만, 조건제시법 표현으로 *eval* 함수를 간편하게 정의할 수 있기에 리스트 타입을 썼다.

이제, 특정 조건을 만족하는 모든 리스트를 돌려주는 쓸모가 많은 조합론적 함수를 몇 개 정의하자. *subs* 함수는 주어진 리스트의 모든 부분수열^{subsequence}을 돌려주는데, 각각의 원소를 빼거나 더하는 모든 가능한 방법의 조합으로 부분수열을 만들어 낸다. *interleave*는 어떤 원소를 주어진 리스트에 끼워넣는 가능한 모든 경우를 돌려준다. *perms*는 주어진 리스트의 모든 순열^{permutation}, 곧 원소들의 순서를 바꾸는 모든 가능한 경우를 돌려준다.

<i>subs</i>	$:: [a] \rightarrow [[a]]$
<i>subs []</i>	$= [[]]$
<i>subs (x : xs)</i>	$= yss \uplus map (x:) yss$ where $yss = subs xs$
<i>interleave</i>	$:: a \rightarrow [a] \rightarrow [[a]]$
<i>interleave x []</i>	$= [[x]]$
<i>interleave x (y : ys)</i>	$= (x : y : ys) : map (y:) (interleave x ys)$
<i>perms</i>	$:: [a] \rightarrow [[a]]$
<i>perms []</i>	$= [[]]$
<i>perms (x : xs)</i>	$= concat (map (interleave x) (perms xs))$

예컨대,

```
> subs [1,2,3]
[[], [3], [2], [2,3], [1], [1,3], [1,2], [1,2,3]]

> interleave 1 [2,3,4]
[[1,2,3,4], [2,1,3,4], [2,3,1,4], [2,3,4,1]]

> perms [1,2,3]
[[1,2,3], [2,1,3], [2,3,1], [1,3,2], [3,1,2], [3,2,1]]
```

다음으로는, 리스트에서 0개 이상의 수들을 아무 순서로나 골라내 만들 수 있는 수열을 모두 구하는 *choices* 함수를, 다음과 같이 모든 부분수열 subsequences 의 순열들을 permutation 모두 구함으로써 간단하게 정의할 수 있다.

<i>choices</i>	$:: [a] \rightarrow [[a]]$
<i>choices xs</i>	$= concat (map perms (subs xs))$

예컨대,

```
> choices [1,2,3]
[[], [3], [2], [2,3], [3,2], [1], [1,3], [3,1], [1,2], [2,1],
 [1,2,3], [2,1,3], [2,3,1], [1,3,2], [3,1,2], [3,2,1]]
```

이제 드디어, 어떤 카운트다운 문제 하나를 풀었다는 사실이 뜻하는 바를 식으로서 나타내는 *solution* 함수를 다음과 같이 정의할 수 있다.

```
solution      :: Expr → [Int] → Int → Bool
solution e ns n = elem (values e) (choices ns) ∧ eval e == [n]
```

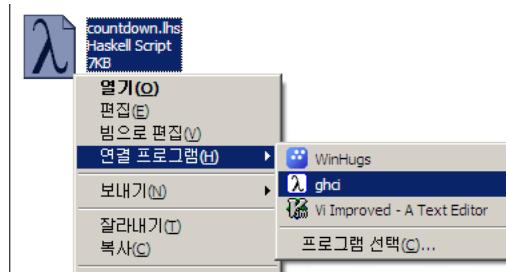
이는, 어떤 식이 주어진 수의 리스트로부터 결과값을 계산하는 해답 *solution* 이 되려면, 식에 나타나는 수들을 주어진 수의 리스트로부터 골랐어야 하며 식의 값을 계산하면 결과값이 되어야 한다는 뜻이다. 예컨대, $e :: Expr$ 가 식 $(1 + 50) * (25 - 10)$ 를 나타낸다고 하면, 다음과 같이 해답인지 확인해 볼 수 있다.

```
> solution e [1, 3, 7, 10, 25, 50] 765
True
```

한 리스트를 다른 리스트로부터 골라 만들었는지 알아보는 *isChoice* 함수를 직접 만들어 쓰면, 골라 만들 수 있는 모든 리스트를 다 생성하는 *choices* 로써 간접적으로 그렇게 하는 것보다 *solution* 의 효율을 개선할 수 있다. 하지만 지금 단계에서는 효율을 중요하게 보지 않고 있고, 이 장에서 다른 함수들을 정의할 때 *choices* 를 어차피 또 쓰게 된다.

따라하기

다음과 같이 *countdown.lhs* 스크립트 파일에 오른 딸깍 right click 하여 팝업 메뉴에서 연결 프로그램을 선택하여 ghci 를 불러들일 수 있다.



이 스크립트에는 카운트다운 문제를 풀기 위한 여러 함수들이 정의되어 있다.
이 장의 끝에서 *countdown.lhs* 를 한눈에 다시 살펴볼 것이다.

다음은 ghci에서 *countdown.lhs*를 불러들여 실행해 본 화면이다.

```

C:\ghc\ghc-6.10.1\bin\ghci.exe
GHCi, version 6.10.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
[1 of 1] Compiling Main           < K:\MyDoc\countdown.lhs, interpreted >
Ok, modules loaded: Main.
*Main> :type eval
eval :: Expr -> [Int]
*Main> eval (App Add (Val 2) (Val 3))
Loading package syb ... linking ... done.
Loading package array-0.2.0.0 ... linking ... done.
Loading package bytestring-0.9.1.4 ... linking ... done.
Loading package Win32-2.2.0.0 ... linking ... done.
Loading package filepath-1.1.0.1 ... linking ... done.
Loading package old-locale-1.0.0.1 ... linking ... done.
Loading package old-time-1.0.0.1 ... linking ... done.
Loading package directory-1.0.0.2 ... linking ... done.
Loading package process-1.0.1.0 ... linking ... done.
Loading package random-1.0.0.1 ... linking ... done.
Loading package haskell98 ... linking ... done.
[5]
*Main> eval (App Sub (Val 2) (Val 3))
[]
[6]
*Main> eval (App Mul (Val 2) (Val 3))
[6]
*Main> eval (App Div (Val 2) (Val 3))
[]
[7]
*Main> subs [1..3]
[[1],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]
*Main> interleave 1 [2,3,4]
[[1,2,3,4],[2,1,3,4],[2,3,1,4],[2,3,4,1]]
*Main> perms [1..2..3]
[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]
*Main> choices [1..2..3]
[[1],[3],[2],[2,3],[3,2],[1],[1,3],[3,1],[1,2],[2,1],[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,2,1]]
*Main> solution (App Mul (App Add (Val 1) (Val 50)) (App Sub (Val 25) (Val 10)))
[1,3,7,10,25,50] 765
True
*Main> solution (App Mul (App Add (Val 1) (Val 50)) (App Sub (Val 25) (Val 10)))
[1,3,7,10,25,50] 831
False
*Main>

```

따라하기 끝

11.3 짐승같이 무식한 brute-force 풀이

카운트다운 문제를 푸는 첫 번째로 방법으로, 주어진 수들로부터 만들 수 있는 모든 가능한 식을 다 만들어 내는 방식으로 짐승같이 무식하게 brute force 해답을 구할 수 있다. 우선, 주어진 하나의 리스트를 두 개의 비어 있지 않은 리스트로 나누되, 그 두 개의 리스트를 이어붙이면 원래 주어진 리스트가 되도록 하는 모든 가능한 경우를 다 돌려주는 *split* 함수를 다음과 같이 정의하자.

```

split      :: [a] → [[a], [a]]
split []    = []
split [_]   = []
split (x : xs) = ([x], xs) : [(x : ls, rs) | (ls, rs) ← split xs]

```

예컨대,

```

> split [1, 2, 3, 4]
[[1], [2, 3, 4]), ([1, 2], [3, 4]), ([1, 2, 3], [4])

```

exprs 은 프로그램에서 핵심적인 역할을 하는 함수로, 식에 나타난 값들이 주어진 리스트와 똑같은 모든 가능한 식을 다 돌려준다. 이를 *split* 을 써서 다음과 같이 정의할 수 있다.

```

exprs     :: [Int] → [Expr]
exprs []  = []
exprs [n] = [Val n]
exprs ns  = [e | (ls, rs) ← split ns,
              l ← exprs ls,
              r ← exprs rs,
              e ← combine l r]

```

이는, 주어진 수들이 없는 빈 리스트로는 아무 식도 만들어낼 수 없으며, 한원소 리스트로는 그 수 자체로 이루어지는 하나의 식만을 만들 수 있음을 나타낸다. 나머지 경우, 곧 두 개 이상의 수로 이루어진 리스트의 경우에는, 일단 리스트를 모든 가능한 방법으로 나눈 다음, 그 나누어진 리스트로부터 만들 수 있는 모든 가능한 식의 쌍을 아래와 같이 정의된 별도의 함수를 써서 네 가지 수치 연산자로 묶는다.

```

combine   :: Expr → Expr → [Expr]
combine l r = [App o l r | o ← ops]
ops       :: [Op]
ops = [Add, Sub, Mul, Div]

```

마지막으로, 주어진 카운트다운 문제의 해답이 될 수 있는 모든 식을 구하는 *solutions* 함수를, 일단 주어진 수로부터 골라 만들 수 있는 모든 식을 다 생성한 다음 그 중 값을 계산했을 때 목표로 하는 결과값과 같은 식만으로 고름으로써 다음과 같이 정의할 수 있다.

```
solutions      :: [Int] → Int → [Expr]
solutions ns n = [e | ns' ← choices ns,
                     e ← exprs ns',
                     eval e == [n]]
```

이 장에서 작성한 프로그램을 Hugs로 시험해 보기에는 성능에 한계가 있으므로, 대신에 글래스고우 하스켈 컴파일러 Glasgow Haskell Compiler 를 썼다. 예컨대, 1.2GHz 펜티엄 M 노트북에서 GHC 6.4.1 판을 사용하여 *solutions* [1,3,7,10,25,50] 765 를 돌리면 첫 해답을 0.36초만에 찾고 모든 780 개의 해답을 43.98초만에 돌려주며, 결과값을 831로 바꿔서 주면 빈 해답 리스트를 44.42초만에 돌려준다.

11.4 생성^{generation} 하면서 계산^{evaluation} 하기

solutions 함수는 주어진 수들로 만들 수 있는 모든 가능한 식을 생성하지만, 뺄셈과 나눗셈의 결과가 항상 양의 자연수가 아니므로 이들 중 실제로 많은 식은 값을 구하지 못하고 실패하게 된다. 예컨대, 1,3,7,10,25,50로 만들 수 있는 모든 가능한 33,665,406 개의 식들 중 4,672,540 개의 식만 제대로 된 값을 가짐을 보일 수 있으며, 비율로는 14%도 채 되지 않는다.

이러한 관찰을 바탕으로, 카운트다운 문제를 푸는 두 번째 방법에서는 식의 생성과 함께 그 값의 계산을 한데 묶어 두 가지 일을 한꺼번에 함으로써 짐승처럼 무식한 프로그램의 효율을 개선한다. 이러한 방식을 쓰면 값이 제대로 나오지 않는 식을 더 일찍 걸러낼 수 있을 뿐 아니라, 그런 제대로 된 값을 갖지 않는 식을 포함하는 다른 식들을 더 이상 만들지 않게 되어 더욱 효율이 개선된다. 우선 제대로 된 값을 갖는 식과 그 값을 순서쌍으로 묶은 *Result* 타입을 정의하자.

```
type Result = (Expr, Int)
```

이 타입을 써서 식에 나타나는 값들이 주어진 리스트와 일치하는 그런 식들과 그 식의 값으로 이루어진 순서쌍을 모두 찾아 돌려주는 *results* 함수를 다음과 같이 정의하자.

```
results      :: [Int] → [Result]
results []   = []
results [n]  = [(Val n, n) | n > 0]
results ns  = [res | (ls, rs) ← split ns,
                  lx ← results ls,
                  ry ← results rs,
                  res ← combine' lx ry]
```

이는, 빈 리스트로부터는 아무것도 만들어내지 못하며, 한원소 리스트로부터는 그 원소가 양의 정수일 때에 한해 그 수 자체로 이루어지는 식과 그 값의 순서쌍 하나를 얻게 된다는 것을 나타낸다. 나머지 경우, 곧 두 개 이상의 수로 이루어진 리스트의 경우에는 일단 리스트를 모든 가능한 방법으로 나눈 다음, 그 나누어진 리스트로부터 만들 수 있는 모든 가능한 식의 쌍을 다음과 같은 함수를 써서 네 가지 수치 연산자로 묶었을 때 제대로 된 값을 갖는 것만을 취한다.

```
combine'       :: Result → Result → [Result]
combine' (l, x) (r, y) = [(App o l r, apply o x y) | o ← ops,
                           valid o x y]
```

이제 *results* 함수를 써서 주어진 카운트다운 문제의 해답으로 가능한 모든 식을 구하는 새로운 *solutions'* 함수를, 우선 주어진 수들로부터 골라내 만들 수 있는 모든 수열을 생성하여, 그 생성한 수열로부터 만들 수 있는 모든 식들 중 그 값이 구하고자 하는 결과값과 같은 것만을 취하도록 다음과 같이 정의할 수 있다.

```
solutions'     :: [Int] → Int → [Expr]
solutions' ns n = [e | ns' ← choices ns
                      (e, m) ← results ns',
                      m == n]
```

성능을 비교하자면, $solutions'$ [1, 3, 7, 10, 25, 50] 765 를 돌렸을 때 첫 해답을 0.04초만에 찾고 ($solutions$ 보다 9 배 빠르다) 모든 해답을 3.47초만에 돌려주며 (12 배 빠르다), 구하고자 하는 결과값을 831로 바꾸면 빈 리스트를 3.28초만에 돌려준다 (13 배 빠르다).

11.5 대수적 성질을 이용하기

$solutions'$ 함수는 주어진 수들로부터 값이 제대로 나오는 모든 가능한 식을 다 만들어 내지만, 수치 연산의 대수적 성질 때문에 실제로 이들 중 많은 식들이 본질상 서로 같게 된다. 예컨대, 덧셈 연산은 인자의 순서에 영향을 받지 않으므로 식 $2+3$ 과 $3+2$ 는 본질상 같으며, 어떤 수를 1로 나누면 그 수 그대로이므로 $2/1$ 과 2 도 본질상 같다.

이러한 관찰을 바탕으로, 카운트다운 문제를 푸는 마지막 방법에서는 대수적 성질을 이용해 두 번째 프로그램이 생성하던 식의 개수를 줄임으로써 효율을 개선한다. 구체적으로는, 다음 다섯 가지 교환법칙 commutativity 및 항등원 identity 의 성질을 이용한다.

$$\begin{aligned}x + y &= y + x \\x * y &= y * x \\x * 1 &= x \\1 * x &= x \\x / 1 &= x\end{aligned}$$

먼저, 어떤 연산을 두 양의 자연수에 적용했을 때 그 결과 또한 양의 정수인지 알아보는 *valid* 함수의 정의를 다시 살펴보는 것으로 시작해 보자.

```
valid      :: Op → Int → Int → Bool
valid Add _ _ = True
valid Sub x y = x > y
valid Mul _ _ = True
valid Div x y = x `mod` y == 0
```

위 정의를 아래와 같이 덧셈과 곱셈의 인자들이 반드시 수의 크기 순서대로 오도록 간단히 고쳐 ($x \leq y$) 교환법칙을 이용할 수 있으며, 또한 곱셈과 나눗셈의 인자들이 단위값이 되지 않도록 간단히 고쳐 ($\neq 1$) 항등원의 성질을 이용할 수 있다.

$$\begin{aligned} valid &:: Op \rightarrow Int \rightarrow Int \rightarrow Bool \\ valid(Add\ x\ y) &= x \leq y \\ valid(Sub\ x\ y) &= x > y \\ valid(Mul\ x\ y) &= x \neq 1 \wedge y \neq 1 \wedge x \leq y \\ valid(Div\ x\ y) &= y \neq 1 \wedge x \cdot mod^y == 0 \end{aligned}$$

예컨대, 이 새로운 정의에 따르면 $Add\ 3\ 2$ 을 이제는 제대로 된 식으로 치지 않는데, 왜냐하면 덧셈의 교환법칙에 따라 $Add\ 2\ 3$ 와 본질상 같은 식이기 때문이다. 또한 $Div\ 2\ 1$ 도 이제는 제대로 된 식으로 치지 않는데, 왜냐하면 나눗셈 항등원의 성질에 따라 본질상 수 2와 같기 때문이다.

새로 정의한 *valid* 를 써서 *solutions'* 함수와 같은 방법으로 카운트다운 문제를 푸는 함수를 *solutions''* 라고 부르기로 하자. 이 새 함수를 쓰면 문제를 풀기 위해 생성하는 식과 해답의 개수가 크게 줄어든다. 예컨대, *solutions''* [1, 3, 7, 10, 25, 50] 765 를 돌리면 식을 254,655 개밖에 생성하지 않으며 그로부터 49개의 해답을 얻는데, 이를 *solutions'* 의 경우와 견주어 보면 생성하는 식의 개수는 5%, 해답의 개수는 6% 가량으로 줄어든 것이다.

성능을 비교하자면, 이제 *solutions''* [1, 3, 7, 10, 25, 50] 765 를 돌렸을 때 첫 해답을 0.002초만에 찾고 (*solutions'* 보다 두 배 빠르다) 모든 해답을 0.44초만에 찾으며 (7 배 빠르다), 구하고자 하는 결과값을 831 로 바꾸면 또한 빈 리스트를 0.44초만에 돌려준다 (7 배 빠르다). 더 일반적으로는, 마지막으로 만든 이 *solutions''* 프로그램으로 텔레비전에서 나올 수 있는 형태의 카운트다운 문제의 모든 해답을 대개 1초 안에 얻을 수 있었다.

스크립트 *countdown.lhs* 한눈에 다시보기

```

1 Programming in Haskell 11장 카운트다운 문제 보기,
2 Graham Hutton, Cambridge University Press, 2007.
3
4
5 > import CPUTime
6 > import Numeric
7
8 식
9 --
10
11 > data Op          = Add | Sub | Mul | Div
12 >
13 > valid           :: Op -> Int -> Int -> Bool
14 > valid Add _ _  = True
15 > valid Sub x y = x > y
16 > valid Mul _ _ = True
17 > valid Div x y = x `mod` y == 0
18 >
19 > apply            :: Op -> Int -> Int -> Int
20 > apply Add x y   = x + y
21 > apply Sub x y   = x - y
22 > apply Mul x y   = x * y
23 > apply Div x y   = x `div` y
24 >
25 > data Expr        = Val Int | App Op Expr Expr
26 >
27 > values           :: Expr -> [Int]
28 > values (Val n)   = [n]
29 > values (App _ l r) = values l ++ values r
30 >
31 > eval              :: Expr -> [Int]
32 > eval (Val n)     = [n | n > 0]
33 > eval (App o l r) = [apply o x y | x <- eval l
34 >                           , y <- eval r
35 >                           , valid o x y]
36

```

```

37
38 조합론적 함수
39 -----
40
41 > subs          :: [a] -> [[a]]
42 > subs []        = []
43 > subs (x:xs)   = yss ++ map (x:) yss
44 >                  where yss = subs xs
45 >
46 > interleave     :: a -> [a] -> [[a]]
47 > interleave x [] = [[x]]
48 > interleave x (y:ys) = (x:y:ys) : map (y:) (interleave x ys)
49 >
50 > perms          :: [a] -> [[a]]
51 > perms []        = []
52 > perms (x:xs)   = concat (map (interleave x) (perms xs))
53 >
54 > choices         :: [a] -> [[a]]
55 > choices xs      = concat (map perms (subs xs))
56
57 문제를 수식으로 정의
58 -----
59
60 > solution        :: Expr -> [Int] -> Int -> Bool
61 > solution e ns n = elem (values e) (choices ns) && eval e == [n]
62
63 짐승같이 무식한 풀이
64 -----
65
66 > split           :: [a] -> [([a],[a])]
67 > split []         = []
68 > split [_]        = []
69 > split (x:xs)    = ([x],xs) : [(x:ls,rs) | (ls,rs) <- split xs]
70 >
71 > exprs           :: [Int] -> [Expr]
72 > exprs []         = []
73 > exprs [n]        = [Val n]
74 > exprs ns        = [e | (ls,rs) <- split ns
75 >                   , l      <- exprs ls
76 >                   , r      <- exprs rs
77 >                   , e      <- combine l r]
78 >
79 > combine          :: Expr -> Expr -> [Expr]
80 > combine l r     = [App o l r | o <- ops]
81 >
82 > ops              :: [Op]
83 > ops              = [Add,Sub,Mul,Div]
84 >
85 > solutions        :: [Int] -> Int -> [Expr]
86 > solutions ns n = [e | ns' <- choices ns
87 >                   , e      <- exprs ns'
88 >                   , eval e == [n]]
89
90 생성하면서 계산하기
91 -----
92
93 > type Result      = (Expr,Int)
94 >
95 > results          :: [Int] -> [Result]

```

11.5. 대수적 성질을 이용하기

```
96 > results []          = []
97 > results [n]          = [(Val n,n) | n > 0]
98 > results ns           = [res | (ls,rs) <- split ns
99 >                      , lx      <- results ls
100 >                     , ry      <- results rs
101 >                     , res     <- combine' lx ry]
102 >
103 > combine'             :: Result -> Result -> [Result]
104 > combine' (l,x) (r,y) = [(App o l r, apply o x y) | o <- ops
105 >                           , valid o x y]
106 >
107 > solutions'            :: [Int] -> Int -> [Expr]
108 > solutions' ns n      = [e | ns'   <- choices ns
109 >                           , (e,m) <- results ns'
110 >                           , m == n]
111
112 대수적 성질을 이용하기
113 -----
114
115 > valid'                :: Op -> Int -> Int -> Bool
116 > valid' Add x y        = x <= y
117 > valid' Sub x y        = x > y
118 > valid' Mul x y       = x /= 1 && y /= 1 && x <= y
119 > valid' Div x y       = y /= 1 && x `mod` y == 0
120 >
121 > results'               :: [Int] -> [Result]
122 > results' []            = []
123 > results' [n]            = [(Val n,n) | n > 0]
124 > results' ns           = [res | (ls,rs) <- split ns
125 >                           , lx      <- results' ls
126 >                           , ry      <- results' rs
127 >                           , res     <- combine' lx ry]
128 >
129 > combine''              :: Result -> Result -> [Result]
130 > combine'' (l,x) (r,y) = [(App o l r, apply o x y) | o <- ops
131 >                           , valid' o x y]
132 >
133 > solutions''            :: [Int] -> Int -> [Expr]
134 > solutions'' ns n      = [e | ns'   <- choices ns
135 >                           , (e,m) <- results' ns'
136 >                           , m == n]
137
138 검사를 위한 대화식 프로그램
139 -----
140
141 > instance Show Op where
142 >   show Add = "+"
143 >   show Sub = "-"
144 >   show Mul = "*"
145 >   show Div = "/"
146 >
147 > instance Show Expr where
148 >   show (Val n)      = show n
149 >   show (App o l r)  = bracket l ++ show o ++ bracket r
150 >                           where
151 >                             bracket (Val n) = show n
152 >                             bracket e      = "(" ++ show e ++ ")"
153 >
154 > showtime    :: Integer -> String
```

```
155 > showtime t  =  showFFloat (Just 3)
156 >                      (fromIntegral t / (10^12)) " seconds"
157 >
158
```

```
159  
160 > display    :: [Expr] -> IO ()  
161 > display es  = do t0 <- getCPUTime  
162 >           if null es then  
163 >             do t1 <- getCPUTime  
164 >               putStrLn "\nThere are no solutions, verified in "  
165 >               putStrLn (showtime (t1 - t0))  
166 >  
167 >           else  
168 >             do t1 <- getCPUTime  
169 >               putStrLn "\nOne possible solution is "  
170 >               putStrLn (show (head es))  
171 >               putStrLn ", found in "  
172 >               putStrLn (showtime (t1 - t0))  
173 >               putStrLn "\n\nPress return to continue searching.."  
174 >               getLine  
175 >               putStrLn "\n"  
176 >               t2 <- getCPUTime  
177 >               if null (tail es) then  
178 >                 putStrLn "There are no more solutions"  
179 >               else  
180 >                 do sequence [print e | e <- tail es]  
181 >                   putStrLn "\nThere were "  
182 >                   putStrLn (show (length es))  
183 >                   putStrLn " solutions in total, found in "  
184 >                   t3 <- getCPUTime  
185 >                   putStrLn (showtime ((t1 - t0) + (t3 - t2)))  
186 >                   putStrLn ".\n\n"  
187 > main      :: IO ()  
188 > main      = do putStrLn "\nCOUNTDOWN NUMBERS GAME SOLVER"  
189 >           putStrLn "-----\n"  
190 >           putStrLn "Enter the given numbers : "  
191 >           ns <- readLn  
192 >           putStrLn "Enter the target number : "  
193 >           n  <- readLn  
194 >           display (solutions' ns n)  
195
```

따라하기

다음은 GHC 대화식 환경인 ghci로 실습한 화면이다. 이와 같이 ghci를 윈도우즈 명령줄(cmd.exe)에서 직접 실행할 수도 있다.



```

C:\WINDOWS\system32\cmd.exe - ghci countdown.lhs

K:\MyDoc>dir countdown*
  K 드라이브의 블록: kyagrd
  블록 일련 번호: 125D-0985

  K:\MyDoc 디렉터리

2009-01-01  08:04           6,905 countdown.lhs
                           1개 파일           6,905 바이트
                           0개 디렉터리 42,341,871,616 바이트 남음

K:\MyDoc>ghci countdown.lhs
GHCi, version 6.10.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
[1 of 1] Compiling Main             < countdown.lhs, interpreted >
Ok, modules loaded: Main.
*Main> main
Loading package syb ... linking ... done.
Loading package array-0.2.0.0 ... linking ... done.
Loading package bytestring-0.9.1.4 ... linking ... done.
Loading package Win32-2.2.0.0 ... linking ... done.
Loading package filepath-1.1.0.1 ... linking ... done.
Loading package old-locale-1.0.0.1 ... linking ... done.
Loading package old-time-1.0.0.1 ... linking ... done.
Loading package directory-1.0.0.2 ... linking ... done.
Loading package process-1.0.1.0 ... linking ... done.
Loading package random-1.0.0.1 ... linking ... done.
Loading package haskel198 ... linking ... done.

COUNTDOWN NUMBERS GAME SOLVER
-----

Enter the given numbers :
[1,3,7,10,25,50]
Enter the target number :
831

There are no solutions, verified in 8.016 seconds.

*Main> =

```

8초나 걸려 답이 없음을 알았다.

다음은 GHC 컴파일러인 ghc로 실습한 화면이다. 이와 같이 (별도의 통합 환경에서 ghc를 불러 쓰도록 설정하지 않은 이상) ghc는 윈도우즈 명령줄(cmd.exe)에서 직접 실행해야 한다.

The screenshot shows a Windows command prompt window titled 'cmd C:\WINDOWS\system32\cmd.exe'. The user is in the directory 'K:\MyDoc'. They first run 'dir countdown*' to see the files: 'countdown.lhs' (6,905 bytes), 'countdown.exe' (1,334,250 bytes), 'countdown.o' (42,469 bytes), 'countdown.hi' (1,355 bytes), and 'countdown.exe.manifest' (506 bytes). Then, they run 'ghc --make countdown.lhs' to compile the source file into an executable. After compilation, they run 'countdown.exe' which outputs 'COUNTDOWN NUMBERS GAME SOLVER'. It prompts for given numbers (1,3,7,10,25,50) and a target number (831), then states there are no solutions found in 0.391 seconds. Finally, they type 'K:\MyDoc>' followed by a carriage return.

0.4초 이내로 답이 없음을 알았다. 일반적으로 ghc로 컴파일하는 것이 ghci 대화식 환경에서 바로 실행하는 것보다 성능이 더 좋다.

따라하기 끝

11.6 살펴보기

카운트다운의 원조는 프랑스 텔레비전 프로그램 “Des Chiffres et des Letters”이며, 카운트다운 문제 자체는 어린이들 산수 놀이인 “krypto” 및 “four fours” 등과 연관이 있다. 이 장은 (15)를 기초로 썼으며, 그 논문에서는 이 장에서 만든 세 프로그램의 정확성^{correctness}도 증명하였다. Bird와 Mu는 카운트다운 문제를 푸는 여러 가지 더 좋은 방법(2)을 연구했다. *subs*, *interleave*, *perms* 함수 정의는 Bird와 Wadler의 책(3)에서 가져왔다.

11.7 연습문제

1. 라이브러리 함수 `concat` 과 `map` 을 쓰지 않고 리스트 조건제시식으로 조합론적 함수인 `choices` 를 다시 정의해 보라.
2. 조합론적 함수인 `perms` 나 `subs` 를 쓰지 않고 한 리스트를 다른 리스트로부터 골라내 만들었는지 알아보는 함수 $isChoice :: Eq a \Rightarrow [a] \rightarrow [a] \rightarrow Bool$ 를 되돌기로 정의하라.
귀띔: 우선, 리스트에서 처음으로 나타나는 어떤 주어진 값 하나를 버리는 함수를 정의하는 것으로 시작해 보라.
3. `split` 함수를 빈 리스트를 포함하는 순서쌍도 돌려주도록 일반화하면, `solutions` 함수의 동작이 어떻게 달라지겠는가?
4. `choices`, `exprs` 및 `eval` 함수를 써서, 1,3,7,19,25,50로부터 만들 수 있는 식이 모두 33,665,406 개이고, 그 중 4,672,540 개만이 제대로 된 값을 가진다는 것을 확인해 보라.
5. 앞 문제와 마찬가지 방법으로, 카운트다운 문제에서 다루는 수치 영역 numeric domain 을 임의의 정수로 일반화하면 제대로 된 값이 나오는 식의 개수가 10,839,369 개로 늘어난다는 것을 확인해 보라.
귀띔: `valid` 함수의 정의를 고치면 된다.
6. 이 장에서 작성한 마지막 프로그램을
 - (a) 식에 거듭제곱을 쓸 수 있도록,
 - (b) 정확히 맞는 해답이 없을 때는 구하고자 하는 결과값에 가장 가까운 값을 갖는 식을 찾아주도록,
 - (c) 결과를 간단한 것부터 적절히 정렬해 보여주도록고쳐 보라.

제 12 장

느긋한 계산법 lazy evaluation

이 장에서는 하스켈에서 식을 계산하는 방법인 느긋한 계산법을 살펴본다. 우선 값을 구하는 계산^{evaluation}이라는 개념을 되돌아 보는 것으로 시작해, 여러 가지 계산 방법과 그 성질을 살펴보고, 무한 구조와 조립식 프로그래밍^{modular programming}을 다룬 후, 프로그램 쓰는 메모리를 아낄 수 있는 특별한 형태의 함수 적용에 대해 알아보는 것으로 이 장을 맺는다.

12.1 소개

이 책에서 살펴본 바와 같이, 하스켈에서는 함수를 인자에 적용하는 것을 계산 원리의 기본으로 삼는다. 예컨대, 하나 더 큰 정수를 계산하는 함수를 다음과 같이 정의했다면,

```
inc   :: Int → Int
inc n = n + 1
```

식 $inc(2 * 3)$ 는 다음과 같이 계산할 수 있다.

$$\begin{aligned}
 & inc(2 * 3) \\
 = & \{ inc\text{를 적용 } \} \\
 = & (2 * 3) + 1 \\
 = & \{ *\text{를 적용 } \} \\
 = & 6 + 1 \\
 = & \{ +\text{를 적용 } \} \\
 = & 7
 \end{aligned}$$

함수를 적용하는 순서를 바꾸어도 최종 결과값은 변함이 없다는 사실은, 위와 같은 간단한 보기뿐 아니라 하스켈에서 일반적으로 함수를 적용할 때 성립하는 중요한 성질이다. 더 격식을 차려 이야기하자면, 하스켈에서 같은 식을 어떤 다른 두 가지 방법으로 계산하더라도, 계산이 둘 다 끝나기만 한다면 항상 같은 결과를 낸다. 나중에 이 장에서 계산의 끝남에 대해 다시 살펴보겠다.

계산의 기본 방법을 저장된 값을 바꿔 나가는 것으로 삼는 대부분의 명령형 언어 imperative language에서는 위와 같은 성질이 성립하지 않음에 주목하라. 예컨대, 변수 n 의 값과 변수의 값을 1로 바꾼 값을 더하는 명령식 $n + (n := 1)$ 의 경우를 생각해 보자. 처음 n 값은 0이라고 가정하자. 덧셈의 왼쪽을 먼저 계산하면 이 식을 다음과 같이 계산할 수 있으며,

$$\begin{aligned}
 & n + (n := 1) \\
 = & \{ n\text{을 적용 } \} \\
 = & 0 + (n := 1) \\
 = & \{ :=\text{를 적용 } \} \\
 = & 0 + 1 \\
 = & \{ +\text{를 적용 } \} \\
 = & 1
 \end{aligned}$$

오른쪽부터 계산하면 다음과 같이 계산할 수 있다.

$$\begin{aligned}
 & n + (n := 1) \\
 = & \{ :=\text{를 적용 } \} \\
 = & n + 1 \\
 = & \{ n\text{을 적용 } \} \\
 = & 1 + 1 \\
 = & \{ +\text{를 적용 } \} \\
 = & 2
 \end{aligned}$$

각각의 경우 최종 결과값이 다르다. 이 보기에서도 드러나듯이, 명령형 언어에서는 덮어쓰기가 정확히 언제 일어나느냐에 따라 계산 결과가 달라질 수 있다는 보편적인 문제가 있다. 반면 하스켈에서는 언제 함수가 인자에 적용되든 절대로 계산의 결과값에는 영향을 미치지 않는다. 그럼에도 불구하고, 계산 순서와 그 특성이 실제로 중요한 문제임을 이 장에서 보일 것이다.

12.2 계산 방식 evaluation strategy

함수와 하나 혹은 그 이상의 인자가 연달아 나타나는 식은 함수를 적용함으로써 “줄일” reduce 수 있으며, 이러한 식을 줄일 수 있는 식 *reducible expression*, 혹은 간단히 줄 식 *redex* 이라 부른다. 앞 문장에서 식을 “줄인다”고 큰 따옴표로 강조한 이유는, 식을 줄였을 때 실제로 식의 크기가 줄어드는 경우가 많기는 하지만 항상 식의 크기가 줄어드는 것은 아니기 때문이다.

예컨대, 정수 쌍을 받아 그 곱을 돌려주는 *mult* 함수를 다음과 같이 정의하고,

$$\begin{aligned} \text{mult} &:: (\text{Int}, \text{Int}) \rightarrow \text{Int} \\ \text{mult } (x, y) &= x * y \end{aligned}$$

식 *mult* ($1 + 2, 2 + 3$)을 살펴보자. 이 식에는 줄 식 *redex* 이 세 개 있는데, + 함수가 두 개의 인자에 적용된 부분식 *subexpression* $1 + 2$ 와 $2 + 3$, 그리고 전체 식 *mult* ($1 + 2, 2 + 3$) 그 자체도 줄 식이다. 이를 각각 줄이면 *mult* ($3, 2 + 3$), *mult* ($1 + 2, 5$) 및 $(1 + 2) * (2 + 3)$ 를 얻는다.

그렇다면 식의 값을 계산할 때 어떤 순서로 줄여 나가야 할까? 흔히 쓰는 방법의 하나로 안쪽부터 계산법 *innermost evaluation* 이 있는데, 다른 어떤 줄 식도 포함하지 않는 가장 안쪽부터 줄여 나가는 방법이다.

하나 이상의 가장 안쪽 줄 식이 있을 때는 관례상 왼쪽부터 줄여 나간다.

예컨대, 식 $\text{mult}(1+2, 2+3)$ 에서 $1+2$ 와 $2+3$ 은 둘 다 다른 줄 식을 포함하지 않는 가장 안쪽 줄 식이며, $1+2$ 가 그 중 가장 왼쪽에 있는 식이다. 왼쪽부터 계산법을 계속 적용해 나가면, 우리가 살펴보고자 하는 식의 값을 다음과 같이 구할 수 있다.

$$\begin{aligned} & \text{mult}(1+2, 2+3) \\ = & \quad \{ \text{첫째 } + \text{를 적용 } \} \\ & \text{mult}(3, 2+3) \\ = & \quad \{ \text{+를 적용 } \} \\ & \text{mult}(3, 5) \\ = & \quad \{ \text{mult 를 적용 } \} \\ & 3 * 5 \\ = & \quad \{ * \text{를 적용 } \} \\ & 15 \end{aligned}$$

인자를 함수에 어떻게 넘기는지 그 특징으로 왼쪽부터 계산법을 설명할 수도 있다. 이 계산 방법을 쓰면 인자의 값을 완전히 다 계산한 다음 함수에 넘기게 된다. 즉, 함수에 인자를 값으로 넘긴다. 예컨대, $\text{mult}(1+2, 2+3)$ 의 값을 왼쪽부터 계산해 나간 위 보기에서도 식 $1+2$ 와 $2+3$ 의 값을 먼저 계산한 다음 mult 함수에 인자로 넘긴다. 가장 안쪽의 줄 식 중에서도 가장 왼쪽부터 줄여 나가므로, 첫째 인자값을 둘째 인자값보다 항상 먼저 구하게 된다.

식의 값을 계산할 때 흔히 쓰는 또 다른 방법으로는, 다른 어떤 줄 식에도 포함되지 않는 가장 바깥쪽 줄 식부터 줄여 나가는 방법이 있다. 그러한 줄 식이 여러 개 있을 때는 마찬가지로 왼쪽 것부터 계산한다. 왼쪽부터 계산법과 맞짝을 이루는 이 계산 방법은 당연히 바깥쪽부터 계산법 outermost evaluation 이라 부른다.

예컨대, 식 $\text{mult}(1+2, 2+3)$ 자체는 다른 다른 어떤 줄 식에도 포함되지 않으므로 가장 바깥쪽 줄 식이다. 바깥쪽부터 계산법을 계속 적용해 나가면 우리가 살펴보고자 하는 식의 값을 다음과 같이 구할 수 있다.

$$\begin{aligned}
 & mult(1+2, 2+3) \\
 &= \{ \text{mult 를 적용 } \} \\
 & (1+2) * (2+3) \\
 &= \{ \text{첫째 } + \text{를 적용 } \} \\
 & 3 * (2+3) \\
 &= \{ \text{+ 를 적용 } \} \\
 & 3 * 5 \\
 &= \{ * \text{를 적용 } \} \\
 & 15
 \end{aligned}$$

인자를 함수에 어떻게 넘기는지 그 특징으로 바깥쪽부터 계산법을 설명하자면, 인자의 값을 구하기 전에 함수에 인자를 넘기는 방법의 계산이다. 이 때문에, 바깥쪽부터 계산할 때는 인자를 이름으로 넘긴다고 표현한다. 예컨대, $mult(1+2, 2+3)$ 의 값을 바깥쪽부터 계산해 나간 위 보기에서도 먼저 $mult$ 함수를 그 값을 아직 다 계산하지 않은 $1+2$ 와 $2+3$ 에 적용한 다음 그 두 식을 계산한다.

하지만 하스켈에서 기본적으로 제공하는 많은 함수들의 경우 바깥쪽부터 계산한다 하더라도 함수를 적용하기 전에 인자의 값을 계산해야 한다는 점에 주목하라. 예컨대, 위의 계산에서도 드러나듯이, 하스켈에서 기본적으로 제공하는 $+$ 나 $*$ 같은 산술 연산은 그 두 인자의 값을 숫자로 계산하기 전까지는 적용할 수 없다. 이러한 성질의 함수를 깬깐하다^{strict} 고 일컫는데, 깬깐한 함수에 대해서는 이 장의 끝부분에서 더 자세히 다루겠다.

람다식 lambda expression

이번에는 $mult$ 를 한 번에 하나의 인자를 받는 커리된 함수로 정의해 보자. 람다식으로 커리됨을 드러나 보이게 다음과 같이 정의할 수 있다.

$$\begin{aligned}
 mult & :: Int \rightarrow Int \rightarrow Int \\
 mult x & = \lambda y \rightarrow x * y
 \end{aligned}$$

그리고는, 예를 들면 다음 보기와 같이 안쪽부터 계산할 수 있다.

$$\begin{aligned}
 & \text{mult} (1 + 2) (2 + 3) \\
 = & \quad \{ \text{첫째 } + \text{를 적용 } \} \\
 = & \quad \text{mult} 3 (2 + 3) \\
 = & \quad \{ \text{mult 를 적용 } \} \\
 = & \quad (\lambda y \rightarrow 3 * y) (2 + 3) \\
 = & \quad \{ + \text{를 적용 } \} \\
 = & \quad (\lambda y \rightarrow 3 * y) 5 \\
 = & \quad \{ \lambda y \rightarrow 3 * y \text{ 를 적용 } \} \\
 = & \quad 3 * 5 \\
 = & \quad \{ * \text{를 적용 } \}
 \end{aligned}$$

앞 절에서 한꺼번에 인자가 쌍으로 넘어간 것과는 달리, 여기서는 두 인자가 한 번에 하나씩 *mult* 함수 몸체로 넘어가 변수를 바꿔치게 되며, 이는 커리된 함수라는 점에서 예상할 수 있는 대로이다. 계산 과정이 이렇게 되는 이유는 *mult* 3 이 식 *mult* 3 (2+3) 안에서 가장 왼쪽에 있는 가장 안쪽 줄 식이기 때문이다. *mult* 3을 줄이면 위 계산 과정의 둘째 단계에 있는 람다식 ($\lambda y \rightarrow 3 * y$)를 얻게 되고, 이 람다식은 다음에 올 둘째 인자를 기다린다.

하스켈에서는 람다식 안에서는 식을 줄이지 못한다. 람다식 안에서 줄이지 못하도록 한 논리적 배경은 함수를 그 안을 들여다 볼 수 없는 상자로 바라보자는 것이다. 더 격식을 차려서 말하자면, 함수를 가지고 할 수 있는 연산은 인자에 적용하는 것밖에 없다. 함수 몸체를 줄이는 것은 인자를 적용하고 난 다음에야 가능하다. 예컨대, 람다식 $\lambda x \rightarrow 1 + 2$ 는 비록 $1 + 2$ 가 줄 식임에도 불구하고 완전히 값이 계산된 것으로 본다. 하지만, 일단 인자가 넘어간 다음에는 줄 식의 값을 계산할 수 있다.

$$\begin{aligned}
 & (\lambda x \rightarrow 1 + 2) 0 \\
 = & \quad \{ \lambda x \rightarrow 1 + 2 \text{ 를 적용 } \} \\
 = & \quad 1 + 2 \\
 = & \quad \{ + \text{를 적용 } \} \\
 = & \quad 3
 \end{aligned}$$

람다식 안을 제외한 안쪽부터 계산법과 바깥쪽부터 계산법을 보통 인자 먼저 계산법 call-by-value evaluation

그리고 함수 먼저 계산법 call-by-name evaluation 이라 부른다. 다음 두 절에서는 두 계산법으로 계산이 어떻게 끝나는지 그리고 끝까지 줄이는 데는 몇 걸음이나 드는지, 그 중요한 특징을 비교해 볼 것이다.

12.3 끝남 termination

다음 되도는 정의를 살펴보자.

$$\begin{aligned} \text{inf} &:: \text{Int} \\ \text{inf} &= 1 + \text{inf} \end{aligned}$$

이는 정수 inf (“무한”infinity 의 줄임)를 그 자신의 바로 다음 수 successor로 정의한 것이다. inf 를 계산하는 데 어떤 방법을 쓰더라도 다음과 같이 식이 점점 더 커지기만 하므로, 계산이 끝나지 않는다.

$$\begin{aligned} \text{inf} &= \{ \text{inf} \text{ 를 적용 } \} \\ &= 1 + \text{inf} \\ &= \{ \text{inf} \text{ 를 적용 } \} \\ &= 1 + (1 + \text{inf}) \\ &= \{ \text{inf} \text{ 를 적용 } \} \\ &= 1 + (1 + (1 + \text{inf})) \\ &= \{ \text{inf} \text{ 를 적용 } \} \\ &\dots \end{aligned}$$

실제로 inf 를 계산하려고 Hugs에서 돌려 보면, 금방 메모리를 다 써버려서 잘못 글귀 error message 가 난다.

이번에는, 식 $\text{fst}(0, \text{inf})$ 를 살펴보자. fst 는 순서쌍의 첫째 성분을 취하는 라이브러리 함수이며, 그 정의는 $\text{fst}(x, y) = x$ 이다. 인자 먼저 계산법으로 이 식을 계산하려 하면, 앞의 경우와 마찬가지로 다음과 같이 계산이 끝나지 않는다.

$$\begin{aligned} & fst(0, inf) \\ = & \quad \{ inf \text{ 를 적용 } \} \\ = & \quad fst(0, 1 + inf) \\ = & \quad \{ inf \text{ 를 적용 } \} \\ = & \quad fst(0, 1 + (1 + inf)) \\ = & \quad \{ inf \text{ 를 적용 } \} \\ = & \quad fst(0, 1 + (1 + (1 + inf))) \\ = & \quad \{ inf \text{ 를 적용 } \} \\ & \dots \end{aligned}$$

반면, 함수 먼저 계산법으로는 다음과 같이 fst 의 정의부터 적용하여 한 걸음 만에 결과값을 얻게 되므로, 계산이 끝나지 않는 식 inf 의 값을 계산하지 않아도 된다.

$$\begin{aligned} & fst(0, inf) \\ = & \quad \{ fst \text{ 를 적용 } \} \\ = & \quad 0 \end{aligned}$$

이 간단한 보기를 통해, 함수 먼저 계산을 하면 인자 먼저 계산으로는 끝나지 않는 경우에도 결과값을 구할 수 있음을 볼 수 있다. 일반적으로 다음의 중요한 성질이 성립한다: 어떤 식의 계산이 끝나는 과정이 존재할 때, 함수 먼저 계산으로도 항상 그 식의 계산을 끝낼 수 있고 그 결과값도 같다.

요컨대, 되도록 계산을 끝내고 싶다면 인자 먼저 계산보다는 함수 먼저 계산이 더 낫다.

12.4 줄이기^{reduction} 횟수

이제 다음 정의를 살펴보자.

$\text{square} :: \text{Int} \rightarrow \text{Int}$
 $\text{square } n = n * n$

다음의 보기와 같이 인자 먼저 계산을 할 수 있다.

$$\begin{aligned}
 & \text{square } (1 + 2) \\
 = & \quad \{ + \text{를 적용 } \} \\
 & \text{square } 3 \\
 = & \quad \{ \text{square 를 적용 } \} \\
 & 3 * 3 \\
 = & \quad \{ * \text{를 적용 } \} \\
 & 9
 \end{aligned}$$

한편, 같은 식을 함수 먼저 계산을 하면 다음과 같이 한 걸음이 더 필요한데, 이는 *square* 함수를 적용했을 때 $1 + 2$ 가 중복되어 두 번 그 값을 구하기 때문이다.

$$\begin{aligned}
 & \text{square } (1 + 2) \\
 = & \quad \{ \text{square 를 적용 } \} \\
 & (1 + 2) * (1 + 2) \\
 = & \quad \{ \text{첫째 } + \text{를 적용 } \} \\
 & 3 * (1 + 2) \\
 = & \quad \{ + \text{를 적용 } \} \\
 & 3 * 3 \\
 = & \quad \{ + \text{를 적용 } \} \\
 & 9
 \end{aligned}$$

이 보기 통해 함수 먼저 계산이 인자 먼저 계산보다 더 많은 걸음으로 값을 구하게 되는 경우가 있음을 알 수 있다. 구체적으로 말하자면, 인자가 함수 몸체에서 여러 번 쓰일 때 그러하다. 일반적으로, 다음의 성질이 성립한다: 인자 먼저 계산에서는 인자의 값을 딱 한 번만 구하지만, 함수 먼저 계산에서는 여러 번 구하게 될 수도 있다. 다행히, 함수 먼저 계산에서 나타나는 이와 같은 효율상의 문제는, 인자로 넘어온 식을 실제로 복사하지 않고 가리키개로 식을 공유함으로써 쉽게 해결할 수 있다. 인자가 함수 몸체에서 여러 번 쓰이면, 식의 원본은 하나만 두고 여러 곳에서 그 식을 가리키게 하면 된다. 이렇게 되면, 인자를 줄여 나갈 때 그 인자를 가리키고 있는 각각의 가리키개들이 줄여 나간 결과를 공유할 수 있다. 예컨대, 이러한 방법으로 다음과 같이 계산할 수 있다.

$$\begin{aligned}
 & \text{square } (1 + 2) \\
 = & \quad \{ \text{ square 를 적용 } \} \\
 & \begin{array}{c} \bullet * \bullet \\ \swarrow \quad \searrow \\ 1 + 2 \end{array} \\
 = & \quad \{ + \text{ 를 적용 } \} \\
 & \begin{array}{c} \bullet * \bullet \\ \swarrow \quad \searrow \\ 3 \end{array} \\
 = & \quad \{ * \text{ 를 적용 } \} \\
 \end{aligned}$$

9

이는 첫 걸음에서 정의가 $\text{square } n = n * n$ 인 함수를 적용할 때 인자식 $1 + 2$ 를 원본 그대로 한 벌만 두고 두 곳에서 가리키도록 한다. 이렇게 하면 둘째 걸음에서 식 $1 + 2$ 를 줄였을 때 두 가리키개가 그 결과를 공유하게 된다.

함수 먼저 계산을 하면서 공유하는 이러한 계산 방법을 느긋한 계산법이라 부른다. 바로 이 계산 방법을 쓰기 때문에, 하스켈을 느긋한 계산하는 프로그래밍 언어 ^{lazy programming language}라 일컫는다. 느긋한 계산법은 인자를 넘기는 계산에 바탕을 두고 있으므로 최대한 많은 경우에 계산이 끝남을 보장한다. 또한, 느긋한 계산법은 계산 결과를 공유하기 때문에 인자 먼저 계산에 비해 결코 걸음을 낭비하는 법이 없다. 왜 “느긋하다”라는 표현을 쓰는지 곧 설명하겠다.

12.5 무한 구조 infinite structure

함수 먼저 계산, 또한 그에 바탕을 둔 느긋한 계산법의 또 다른 특징은, 처음 보면 말도 안되는 이야기로 들릴지 모르겠는데, 무한 구조를 다루는 프로그램을 짤 수 있다는 점이다. 이러한 개념은 이 장에서 앞서 살펴본 간단한 보기에도 이미 드러나 있는데, 바로 $\text{fst } (0, \text{inf})$ 를 계산할 때 무한 구조로 정의된 inf 가 $1 + (1 + (1 + \dots))$ 와 같이 늘어나는 것을 피한 바 있다. 무한한 리스트를 다룰 때 나타나는 성질을 살펴보면 더 흥미롭다. 예컨대, 다음의 되도는 정의를 살펴보자.

```
ones :: [Int]
ones = 1 : ones
```

즉, *ones*라는 리스트는 그 자신 앞에 1을 하나 덧붙인 것이다. *inf*와 마찬가지로, 어떤 방법으로도 *ones*의 값을 구하는 계산은 끝나지 않는다.

```
ones
= { ones 를 적용 }
  1 : ones
= { ones 를 적용 }
  1 : (1 : ones)
= { ones 를 적용 }
  1 : (1 : (1 : ones))
= { ones 를 적용 }
...
...
```

실제로 Hugs에서 *ones* 리스트의 값을 구하려 하면, 사용자가 작동을 강제로 멈추지 않는 한 다음과 같이 리스트를 끝없이 만들어 나간다.

```
> ones
[1, 1, 1, 1, 1, 1, 1, 1, ...]
```

이제 이 1이 무한이 반복되는 이 리스트의 첫 원소를 취하고자 하는 식 *head ones*를 살펴보자. 인자 먼저 계산법을 쓰면, 이 식의 계산 역시 끝나지 않는다.

```
head ones
= { ones 를 적용 }
  head (1 : ones)
= { ones 를 적용 }
  head (1 : (1 : ones))
= { ones 를 적용 }
  head (1 : (1 : (1 : ones)))
= { ones 를 적용 }
...
...
```

반면, 느긋한 계산법으로는 (이 보기에서는 식을 공유할 필요가 없으므로

함수 먼저 계산법으로도 마찬가지다) 두 걸음 만에 계산이 끝난다.

$$\begin{aligned}
 & \text{head } \textit{ones} \\
 = & \quad \{ \text{ } \textit{ones} \text{ } \text{를 적용 } \} \\
 = & \quad \text{head } (1 : \textit{ones}) \\
 = & \quad \{ \text{ } \text{head } \text{를 적용 } \} \\
 1
 \end{aligned}$$

이런 계산 과정이 나타나는 까닭은 느긋한 계산법은 말 그대로 계산을 느긋하게, 다시 말해 결과값을 구하기 위해 꼭 필요한 경우에만 인자를 계산하기 때문이다. 예컨대, 리스트의 첫 원소를 취하려면 리스트의 나머지 부분은 필요가 없으므로 `head (1 : ones)`에서 무한한 리스트인 `ones`를 더 계산해 들어가지 않는다. 일반적으로 다음 성질이 성립한다: 느긋한 계산법에서는 문맥상 필요한 꼭 그만큼만 식의 값을 계산한다.

방금 살펴본 것처럼 느긋한 계산법에서 `ones`는 문자 그대로 그렇게 무한한 리스트라기보다는, 문맥상 필요한 만큼만의 값을 구해 놓고 앞으로도 무한히 계속 늘어날 가능성이 있는 리스트라는 것을 알 수 있다. 이러한 개념을 리스트뿐 아니라, 하스켈에서 어떤 데이터 구조에도 마찬가지로 적용할 수 있다. 예컨대, 이 장 마지막 연습문제에서 무한한 나무를 다루겠다.

12.6 모듈 방식 modular 프로그래밍

느긋한 계산법을 쓰면 계산을 할 때 데이터에서 (흐름) 제어^{control} 부분을 따로 떼어놓을 수 있다. 예컨대, 세 개의 1로 이루어진 리스트를 1이 무한히 반복되는 리스트에서 (데이터) 처음 세 원소만 골라내어 (제어) 만들 수 있다.

```
> take 3 ones
[1, 1, 1]
```

표준 서막에 있는 *take*의 정의는 다음과 같다.

$$\begin{array}{lll} take\ 0 & = [] \\ take\ (n+1)\ [] & = [] \\ take\ (n+1)\ (x:xs) & = x : take\ n\ xs \end{array}$$

위 정의에 따라 느긋한 계산법으로 계산하면 앞서 이야기한 것처럼 1이 세 개 있는 리스트를 다음과 같이 얻게 된다.

$$\begin{aligned} & take\ 3\ ones \\ = & \{ ones \text{ 를 적용 } \} \\ & take\ 3\ (1:ones) \\ = & \{ take \text{ 를 적용 } \} \\ & 1:take\ 2\ ones \\ = & \{ ones \text{ 를 적용 } \} \\ & 1:take\ 2\ (1:ones) \\ = & \{ take \text{ 를 적용 } \} \\ & 1:1:take\ 1\ ones \\ = & \{ ones \text{ 를 적용 } \} \\ & 1:1:1:take\ 0\ ones \\ = & \{ take \text{ 를 적용 } \} \\ & 1:1:1:[] \\ = & \{ \text{리스트 표기 } \} \\ & [1,1,1] \end{aligned}$$

이와 같이 데이터와 제어 두 부분이 번갈아 식을 줄여 나가면서 제어 부분에서 필요한 만큼만 데이터를 계산한다. 느긋한 계산법이 아니라면, 제어와 데이터 부분을 다음과 같이 n 개의 똑같은 원소로 이루어진 리스트를 만들어내는 하나의 함수로 한데 섞어 놓아야 한다.

$$\begin{array}{ll} replicate & :: Int \rightarrow a \rightarrow [a] \\ replicate\ 0\ _ & = [] \\ replicate\ (n+1)\ x & = x : replicate\ n\ x \end{array}$$

논리적으로 독립적인 부분을 따로 떼어 놓아 조립할 수 있도록 하는 것은 프로그램을 짜는 데 있어 매우 중요한 원리 중 하나이며, 데이터와 (흐름) 제어를 따로 떼어놓을 수 있다는 것은 느긋한 계산법의 매우 중요한 장점 중 하나이다.

느긋한 계산법을 쓰더라도 무한한 리스트를 다룰 때는 여전히 끝나지 않는 계산이 나타나지 않도록 주의를 기울여야 한다는 것을 알라. 예컨대, 다음과 같은 식은

filter (≤ 5) [1 ..]

(위에서 $[n ..]$ 는 n 부터 시작하여 무한히 계속되는 정수 리스트를 나타낸다) 정수 1,2,3,4,5 를 만들어 낸 후 언제까지고 계속해서 돈다. 왜냐하면 *filter* (≤ 5) 함수는 무한한 리스트로부터 계속해서 원소를 가져와 5 이하인 수가 또 없는지 찾는 헛수고를 반복하기 때문이다. 반면, 다음 식은

takeWhile (≤ 5) [1 ..]

마찬가지 정수들을 만들어 내고 난 다음 계산을 끝낸다. 왜냐하면 *takeWhile* (≤ 5) 함수는 5보다 큰 원소를 하더라도 찾으면 계산을 그만두기 때문이다.

소수에 관한 보기를 살펴보는 것을 끝으로 이 절을 마무리하겠다. 5장에서 주어진 한계까지 소수를 만들어내는 함수를 정의한 바 있다. 여기서는 소수를 앞에서부터 일부분만을 생성하는 것이 아니라, 모든 소수로 이루어진 무한 수열을 만들어 내는 간단한 절차를 소개하겠다.

1. 무한 수열 2,3,4,5,6,... 를 적는다.
2. 첫번째 수 p 를 소수로 표시한다.
3. 나머지 p 의 배수들을 수열에서 지운다.
4. 걸음 1로 돌아간다.

첫째와 셋째 걸음을 마치는 데는 무한한 양의 일을 해야 하므로, 실제로는 계산이 이러한 걸음을 걸음을 왔다갔다하면서 일어난다. 위 절차를 처음부터 몇 번 반복하는 과정을 아래와 같이 나타낼 수 있다.

2	3	<u>4</u>	5	<u>6</u>	7	<u>8</u>	9	<u>10</u>	11	<u>12</u>	13	<u>14</u>	<u>15</u>	...
3		5		7		9			11		13		<u>15</u>	...
	5			7					11		13			...
			7						11		13			...
					7				11		13			...
						11				13				...
							11			13				...
								13						...

각 줄은 위 절차를 한 번 수행한 것을 나타내는데, 첫 줄은 제일 처음 적어 내려간 수열이고 (결음 1), 각 줄에서 첫 수는 소수임을 강조하기 위해 (결음 2) 굵게 표시하였으며, 그 배수들은 지워서 다음 번에 쓰지 않을 것임을 나타내기 위해 (결음 3) 밑줄로 표시하였다. 이런 식으로 각 줄의 첫 수만을 모아 수열을 만들면 매번 밑줄 친 소수의 배수들이 걸러져 나가므로 굵게 표시된 소수만 남은 다음과 같은 무한 수열을 얻는다.

2, 3, 5, 7, 11, 13, ...

위와 같이 소수를 만들어 내는 방법을, 처음 발견한 그리스 수학자의 이름을 따서, “에라토스테네스의 체”^{sieve of Eratosthenes}라 부른다. 에라토스테네스의 체를 있는 그대로 하스켈로 옮기면 다음과 같다.

```
primes :: [Int]
primes = sieve [2..]
sieve      :: [Int] → [Int]
sieve (p : xs) = p : sieve [x | x ← xs, x `mod` p ≠ 0]
```

즉, 무한한 리스트 [2..]로 시작해서 (결음 1), 여기에 *sieve* 함수를 적용하여 첫 원소 *p*를 소수로서 골라내고 (결음 2), 나머지 리스트에서 *p*의 배수를 모두 걸러낸 리스트에 (결음 3) *sieve* 함수 자신을 적용함으로써 (결음 4) 되돈다. 이 프로그램은 정말로 다음과 같은 무한한 소수 리스트를 만들어 내는데, 이는 느긋한 계산법을 따르기 때문이다.

```
> primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 53, 61, 67, ...]
```

소수를 만들어 내는 과정에서 유한한 개수만 만들어 내야 한다는 제약을 없애 자유롭게 함으로써, 우리는 조립식 프로그램, 즉 경우에 따라 제어 부분을 달리 조립할 수 있는 프로그램을 만들었다. 예컨대, 10 개의 소수가 필요하거나 10보다 작은 소수가 필요한 경우에는 다음과 같이 필요한 소수들을 얻도록 조립할 수 있다.

```
> take 10 primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

```
> takeWhile (<10) primes
[2, 3, 5, 7]
```

12.7 깊은 적용 strict application

하스켈에서는 기본적으로 느긋한 계산법을 쓰지만, 특별한 $\$!$ 연산자를 써서 함수를 깊은 적용할 수 있으며, 이는 때때로 쓸모가 있다. 대강 설명하자면, $f \$! x$ 와 같은 모양의 식은 x 의 최상위 단계를 먼저 계산하고 난 다음에 함수 f 를 적용한다는 점을 빼고는 보통 함수 적용 $f x$ 와 똑같다.

예컨대, *Int* 나 *Bool* 같은 기본 타입 인자라면 최상위 단계를 계산한 것이 곧 인자값을 완전히 계산한 것이다. 반면, (*Int*, *Bool*) 과 같은 순서쌍 타입의 경우, (각 성분의 값은 구하지 않고) 순서쌍임을 알 수 있을 때까지만 계산하고 더 이상 계산하지 않는다. 마찬가지로 리스트 타입의 경우에도 빈 리스트나 두 식을 콘스($:$)로 엮은 값을 얻을 때까지만 계산한다.

더 격식을 차려 말하자면, $f \$! x$ 와 같은 형태의 식은, 인자 x 를 보통의 느긋한 계산법으로 계산하여 그 값이 부정 undefined value 이 아님을 확인하고 나서야 줄일 수 있는 식이 되어, 보통 적용식 $f x$ 처럼 줄인다.

하스켈에서 부정(정의되지 않은 값)이란 *error* 함수의 결과 혹은 부분적으로라도 어떤 값도 내지 않고 되돌며 끝나지 않는 계산(예를 들면 $inf = inf$ 와 같은 정의)을 말한다.

예컨대, 정의가 $\text{square } n = n * n$ 인 제곱 함수가 포함된 식 $\text{square } \$! (1 + 2)$ 의 값을 구하려면 마치 인자 먼저 call-by value 계산할 때처럼 인자식 $1 + 2$ 를 먼저 계산해 그 값 3을 구한 다음 square 함수를 적용한다.

$$\begin{aligned} & \text{square } \$! (1 + 2) \\ = & \quad \{ + \text{를 적용 } \} \\ & \text{square } \$! 3 \\ = & \quad \{ \$! \text{를 적용 } \} \\ & \text{square } 3 \\ = & \quad \{ \text{square} \text{를 적용 } \} \\ & 3 * 3 \\ = & \quad \{ * \text{를 적용 } \} \\ & 9 \end{aligned}$$

인자를 여러 개 받는 커리된 함수의 경우, 원하는 인자만 골라 깐깐하게 적용하여 그 인자들의 최상위 단계의 계산만 강제하는 것도 얼마든지 가능하다. 예컨대, 커리된 함수 f 가 인자를 두 개 받는다면, $f x y$ 와 같은 모양의 식에 대해 다음과 같이 세 가지 서로 다른 방법으로 깐깐한 적용을 할 수 있다.

$$\begin{aligned} (f \$! x) y & \quad -- x \text{의 최상위 단계 계산을 강제} \\ (f x) \$! y & \quad -- y \text{의 최상위 단계 계산을 강제} \\ (f \$! x) \$! y & \quad -- x \text{와 } y \text{의 최상위 단계 계산을 강제} \end{aligned}$$

하스켈에서는 주로 프로그램이 쓰는 메모리를 아끼기 위해 깐깐한 함수 적용을 한다. 예컨대, 쌓개 accumulator를 써서 여러 정수의 합을 구하는 다음과 같은 sumwith 함수를 살펴보자.

$$\begin{aligned} \text{sumwith} & :: \text{Int} \rightarrow [\text{Int}] \rightarrow \text{Int} \\ \text{sumwith } v [] & = v \\ \text{sumwith } v (x : xs) & = \text{sumwith } (v + x) xs \end{aligned}$$

느긋한 계산법으로 다음과 같이 계산할 수 있다.

```

sumwith 0 [1,2,3]
=   { sumwith 를 적용 }
  sumwith (0 + 1) [2,3]
=   { sumwith 를 적용 }
  sumwith ((0 + 1) + 2) [3]
=   { sumwith 를 적용 }
  sumwith (((0 + 1) + 2) + 3) []
=   { sumwith 를 적용 }
  (((0 + 1) + 2) + 3)
=   { 첫째 + 를 적용 }
  ((1 + 2) + 3)
=   { 첫째 + 를 적용 }
  3 + 3
=   { + 를 적용 }
  6

```

실제 덧셈은 한번도 하지 않은 채, 덧셈식 $((0 + 1) + 2) + 3$ 부터 통째로 먼저 만든다는 점에 주목하라. 더 일반적으로는, *sumwith* 함수는 인자로 넘겨받은 원래 리스트의 길이에 비례하는 크기의 덧셈식을 만들게 된다. 예컨대, Hugs로 *sumwith* 0 [1..10000]의 값을 구하려 하면 금방 메모리를 다 써 버리고 잘못 글귀가 난다. 그래서 실제로 이런 계산을 할 때는 덧셈이 나타나는 족족 계산함으로써 메모리 낭비를 피하는 것이 좋다.

간간한 적용으로 쌓개 값 계산을 강제하도록 *sumwith*를 다음과 같이 바꿔 정의하면 원하는 결과를 얻을 수 있다.

```

sumwith v []      = v
sumwith v (x : xs) = (sumwith $! (v + x)) xs

```

이번에는 다음과 같이 계산된다.

```

sumwith 0 [1,2,3]
=   { sumwith 를 적용 }
  (sumwith $! (0 + 1)) [2,3]
=   { $! 를 적용 }
  sumwith 1 [2,3]
=   { sumwith 를 적용 }
  (sumwith $! (1 + 2)) [3]
=   { $! 를 적용 }
  sumwith 3 [3]
=   { sumwith 를 적용 }
  (sumwith $! (3 + 3)) []
=   { $! 를 적용 }
  sumwith 6 []
=   { sumwith 를 적용 }
  6

```

이번에는 깐깐한 적용을 하느라 지난 번보다 더 많은 걸음이 들지만, 커다란 덧셈식을 만들지 않고 덧셈이 나오는 족족 계산한다. 예컨대, Hugs로 $\text{sumwith } 0 [1..10000]$ 의 계산하면, 이번에는 지난 번처럼 잘못이 나지 않고 결과값이 제대로 나온다.

위 보기일 일반화하여 다음과 같이 함수를 돌려주는 라이브러리 함수 foldl 의 깐깐한 판인 foldl' 를 만들어 볼 수도 있다. foldl' 함수는 나머지 리스트를 갖고 계산하기에 앞서 쌓개 값부터 먼저 계산하도록 강제한다.

$$\begin{aligned}\text{foldl}' &:: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a \\ \text{foldl}' f v [] &= v \\ \text{foldl}' f v (x : xs) &= ((\text{foldl}' f) \$! (f v x)) xs\end{aligned}$$

예컨대, 위 함수를 써서 $\text{sumwith} = \text{foldl}' (+)$ 와 같이 정의할 수 있다. 하지만 깐깐한 적용 strict application 이 결코 하스켈 프로그램의 메모리 낭비를 저절로 해결해 주는 만병통치약이 아님을 명심해야 한다. 깐깐한 적용 strict application 은 비교적 간단해 보이는 경우라도 느긋한 계산법이 어떻게 동작하는지 신중히 고려해야 하는 전문적인 주제다.

12.8 살펴보기

여러 순서의 계산법과 그 성질에 대한 더 자세한 내용은 (28)에서 찾아볼 수 있으며, 느긋한 계산법을 써서 조립식 프로그래밍을 하는 더 많은 보기일 유명한 논문 “함수형 프로그램이 왜 중요한가?” Why Functional Programming Matters (13)에서 찾아볼 수 있다. 느긋한 계산법의 엄밀한 형식적 의미는 (21)에서 찾아볼 수 있으며, 느긋한 계산법을 효율적으로 구현하는 포괄적인 강좌를 (26)에서 찾아볼 수 있다.

12.9 연습문제

1. 다음 식들에서 줄 식을 모두 찾은 다음, 각각의 줄 식 중 어느 것이 가장 안쪽에 있고, 또 어느 것이 가장 바깥에 있는지, 혹은 둘 중 어느 경우도 아닌지, 아니면 둘 다인지 알아보라.

$$\begin{aligned} & 1 + (2 * 3) \\ & (1 + 2) * (2 + 3) \\ & fst(1 + 2, 2 + 3) \\ & (\lambda x \rightarrow 1 + x)(2 * 3) \end{aligned}$$

2. 식 $fst(1 + 2, 2 + 3)$ 의 값을 구하는 데 있어, 바깥쪽부터 계산법이 안쪽부터 계산법보다 어째서 더 나은지 보이라.
3. 함수 정의 $mult = \lambda x \rightarrow (\lambda y \rightarrow x * y)$ 를 이용해, $mult 3 4$ 의 값을 구하는 과정을 어떻게 네 걸음으로 따로 분리해 낼 수 있는지 보이라.
4. 리스트 조건제시식으로 무한한 피보나치 수열

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots$$

를 만들어 내는 식 $fibs :: [Integer]$ 를 다음의 간단한 절차로써 정의하라.

- (a) 첫 두 수는 0과 1이다.
- (b) 그 다음에 오는 수는 바로 앞 두 수의 합이다.
- (c) 바로 위 둘째 걸음으로 돌아가라.

귀띔: 라이브러리 함수 zip 과 $tail$ 을 이용하라. 피보나치 수는 금방 커지기 때문에, 위에서 임의 크기 정수 타입인 $Integer$ 를 쓴 것에 주목하라.

5. $fibs$ 를 써서 (0부터 세어) n 번째 피보나치 수를 구하는 함수 $fib :: Int \rightarrow Integer$ 를 정의하고, 10000 보다 큰 첫 피보나치 수를 구하는 식을 만들어 보라.

6. 다음 라이브러리 함수들의 적절한 판을

```

repeat   :: a → [a]
repeat x = xs where xs = x : xs
take      :: Int → [a] → [a]
take 0    _     = []
take (n + 1) []  = []
take (n + 1) (x : xs) = x : take n xs
replicate :: Int → a → [a]
replicate n = take n ∘ repeat
  
```

아래와 같은 두갈래나무 타입에 대해 정의하라.

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

제 13 장

프로그램에 대한 논리적 증명

이 마지막 장에서는 하스켈 프로그램의 성질을 논리적으로 증명하는 방법을 알아보겠다. 등식 바탕의 논증에 대한 개념을 들이켜 보는 것으로 시작해, 등식 바탕의 논증을 하스켈이 어떻게 적용할지 살펴보고, 증명에 있어 중요한 기술인 귀납법^{induction}을 소개하고, 귀납법을 써서 이어붙이기 연산자^{append operator}를 어떻게 없애는지 알아본 후, 간단한 번역기^{compiler}의 정확성을 증명하는 것으로 이 장을 끝맺는다.

13.1 등식 바탕의 논증 equational reasoning

덧셈의 교환법칙과 곱셈의 교환법칙이 성립하며, 덧셈에 대한 곱셈의 분배법칙^{distributivity}이 왼쪽 오른쪽 양쪽으로 성립한다는 등의 다음과 같은 수의 대수적 성질을 학교에서 배웠을 것이다:

$$\begin{array}{ll} xy & = yx \\ x + (y + z) & = (x + y) + z \\ x(y + z) & = xy + xz \\ (x + y)z & = xz + yz \end{array}$$

예컨대, 위의 성질을 이용하여 $(x + a)(x + b)$ 와 같은 형태의 곱셈식을 풀면 $x^2 + (a + b)x + ab$ 와 같은 형태의 덧셈식이 됨을 다음과 같이 보일 수 있다.

$$\begin{aligned} & (x + a)(x + b) \\ &= \{ \text{왼쪽 분배법칙} \} \\ & (x + a)x + (x + a)b \\ &= \{ \text{오른쪽 분배법칙} \} \\ & xx + ax + xb + ab \\ &= \{ \text{제곱으로 나타내기} \} \\ & x^2 + ax + xb + ab \\ &= \{ \text{교환법칙} \} \\ & x^2 + ax + bx + ab \\ &= \{ \text{오른쪽 분배법칙} \} \\ & x^2 + (a + b)x + ab \end{aligned}$$

이러한 계산을 할 때는 은근슬쩍 결합법칙을 써먹는다는 것에 주목하라. 위의 경우에는 덧셈의 결합법칙을 써서 덧셈이 연이어 나타날 때 괄호를 생략하였다.

대수적 성질은 그 자체로 흥미롭기도 하지만, 계산이 어떻게 이루어지는가 하는 측면에서도 중요하다. 예컨대, 식 $x(y + z)$ 의 값을 구하려면 연산을 두 번 (곱셈 한 번과 덧셈 한 번) 해야 하지만, 값이 같은 식 $xy + xz$ 의 값을 구하려면 연산을 세 번 (곱셈 두 번과 덧셈 한 번) 해야 한다. 따라서 이 두 식이 대수적으로는 같은 값을 갖지만, 효율을 고려하면 먼저 식이 나중 식보다 낫다.

13.2 하스켈 프로그램에 대한 논증

하스켈에서도 똑같은 방식의 등식을 바탕으로 한 논증이 가능하다. 예컨대, 이러한 맥락에서 등식 $x * y = y * x$ 가 뜻하는 바는 식 x 와 y 가 어떤 식이든 간에 같은 수치 타입이기만 하다면 식 $x * y$ 와 $y * x$ 를 계산한 결과로 항상 같은 수치값이 나온다는 것이다. 이러한 성질을 나타내는 데 하스켈에서 제공하는 같기 연산자 $==$ 를 쓰지 않음에 주목하라. 이를테면 $x * y == y * x$ 라고 하지 않는 이유는, 하스켈 프로그램의 성질을 증명하는 언어로 하스켈 그 자신이 아닌 수학을 쓰기 때문이다. 만일 증명을 위한 언어로 하스켈 그 자신을 쓴다면 뭔가 돌고 도는 논증이 되기 때문이다.

하스켈에 대해 논증할 때, 덧셈이나 곱셈과 같은 하스켈에서 기본적으로 제공하는 연산자의 성질 뿐만 아니라 프로그래머가 함수를 정의할 때 쓴 등식들도 증명에 이용한다. 예컨대, 정수의 두 배 값을 구하는 다음 함수를 살펴보자.

```
double :: Int → Int  
double x = x + x
```

위에 나타난 등식은 함수 정의로 생각할 수 있을 뿐 아니라, 이 함수에 대해 논증할 때 쓰이는 함수의 성질로도 생각할 수 있다. 더 구체적으로 말하자면, 위 등식은 임의의 식 x 에 대해 $double x$ 를 $x + x$ 로 언제든지 바꿔 쓸 수 있고, 반대로 $x + x$ 를 $double x$ 로 바꿔 쓸 수 있다는 논리적인 성질을 나타낸다. 이런 식으로, 프로그램에 대해 논증할 때는 함수 정의를 왼쪽에서 오른쪽으로 적용함은 물론 오른쪽에서 왼쪽으로 역 적용하는 양방향으로 쓸 수 있다.

하지만, 여러 등식으로 정의한 함수에 대해 논증할 때는 주의가 필요하다. 예컨대, 다음과 같이 두 개의 등식을 써서 정의한, 정수가 0인지 알아보는 함수를 생각해 보자.

```
isZero :: Int → Bool  
isZero 0 = True  
isZero n = False
```

첫 등식 $\text{isZero } 0 = \text{True}$ 는 언제든지 양방향으로 쓸 수 있는 논리적 성질로 볼 수 있다. 하지만 둘째 등식 $\text{isZero } n = \text{False}$ 의 경우는 그렇지 못하다. 더 자세히 설명하자면, 이는 등식이 나타나는 순서를 고려해야 하기 때문에, $n = 0$ 일 때는 첫째 등식이 적용되므로, $n \neq 0$ 일 경우에만 $\text{isZero } n$ 을 False 로 바꿔 쓸 수 있다. 반대로, $\text{isZero } n = \text{False}$ 를 역작용^{unapply}하여 False 를 $\text{isZero } n$ 로 바꿔 쓰는 것도 같은 이유로 $n \neq 0$ 일 경우만 가능하다.

일반적으로, 함수를 여러 개의 등식으로 정의했을 때 각각의 등식을 다른 등식과 독립적인 논리적 성질로 볼 수 없으며, 그 순서를 고려하여 일련의 등식에 나타나는 패턴 매칭이 어떻게 일어나는지 살펴보아야 한다. 그렇기 때문에, 나타나는 순서에 영향을 받지 않는 등식으로써 함수를 정의하는 것이 더 좋다. 예컨대, 위 정의를 보초를 이용해 다음과 같이 고쳐 쓰면

$$\begin{aligned}\text{isZero } 0 &= \text{True} \\ \text{isZero } n \mid n \neq 0 &= \text{False}\end{aligned}$$

이제는 보초 $n \neq 0$ 를 통과할 때만 $\text{isZero } n$ 을 False 로 바꿔 쓰거나 반대로 False 를 $\text{isZero } n$ 로 바꿔 쓸 수 있다는 것이 명확히 드러난다. 어느 패턴을 먼저 맞추든 그 순서에 관계없는 패턴들을 서로 겹치지 않는 disjoint, non-overlapping 고 한다. 프로그램에 대한 논증 과정을 간단히 하기 위해서는 가능한 한 패턴을 서로 겹치지 않도록 정의하는 것이 좋은 방법이다. 예컨대, 부록 A에서 대부분의 표준 라이브러리 함수들을 이런 방법으로 정의한다.

13.3 간단한 보기들

하스켈에서 등식 바탕의 논증을 보여 주는 간단한 보기로서, 리스트를 뒤집는 라이브러리 함수의 정의를 다시 살펴보자.

$$\begin{aligned}\text{reverse} &:: [a] \rightarrow [a] \\ \text{reverse } [] &= [] \\ \text{reverse } (x : xs) &= \text{reverse } xs ++ [x]\end{aligned}$$

이 정의를 써서 *reverse* 가 한원소 리스트에 대해서는 아무 일도 않는다는 성질, 곧 어떤 원소 x 에 대해서도 $\text{reverse } [x] = [x]$ 가 성립함을 다음과 같이 보일 수 있다.

$$\begin{aligned} & \text{reverse } [x] \\ &= \{ \text{리스트 표기 } \} \\ &\quad \text{reverse } (x : []) \\ &= \{ \text{reverse 를 적용 } \} \\ &\quad \text{reverse } [] \uplus [x] \\ &= \{ \text{reverse 를 적용 } \} \\ &\quad [] \uplus [x] \\ &= \{ \uplus \text{를 적용 } \} \\ &\quad [x] \end{aligned}$$

따라서 프로그램에 $\text{reverse } [x]$ 형태의 식이 나타나면 언제든지 $[x]$ 로 바꿔 쓸 수 있다. 이 때, 프로그램의 의미는 변하지 않지만 *reverse* 함수 적용이 없어지므로 프로그램의 효율은 변한다.

등식 바탕의 논증에서 경우를 나눠 분석해야 할 때가 흔히 있다. 예컨대, 다음과 같은 논리 부정 함수를 생각해 보자.

$$\begin{aligned} \neg &:: \text{Bool} \rightarrow \text{Bool} \\ \neg \text{False} &= \text{True} \\ \neg \text{True} &= \text{False} \end{aligned}$$

이 함수는 패턴 매칭으로 정의했기 때문에, \neg 함수의 성질은 대개 그 인자에 대한 경우를 나눠 분석함으로써 보일 수 있다. 예컨대, \neg 이 그 자신의 역함수라는 성질, 곧 모든 논리값 b 에 대해 $\neg(\neg b) = b$ 가 성립함을 b 가 가질 수 있는 두 값에 대해 각각의 경우를 분석함으로써 보일 수 있다. 예컨대, $b = \text{False}$ 인 경우에 다음과 같이 증명할 수 있으며, $b = \text{True}$ 인 경우에도 비슷하게 증명할 수 있다.

$$\begin{aligned} & \neg(\neg \text{False}) \\ &= \{ \text{안쪽 } \neg \text{을 적용 } \} \\ &\quad \neg \text{True} \\ &= \{ \neg \text{을 적용 } \} \\ &\quad \text{False} \end{aligned}$$

13.4 자연수에 대한 귀납법

쓸만한 함수형 프로그램은 대개 되돌기를 어떤 형태로든 쓰게 마련이다. 그러한 프로그램에 대한 논리적 증명을 할 때는 보통 귀납법이라는 간단하지만 강력한 기술을 쓴다. 우선 가장 간단한 되도는 타입으로 잘 알려진, 이름하여 자연수 타입을 살펴보자.

```
data Nat = Zero | Succ Nat
```

이 타입 선언은 *Zero* 가 *Nat* 타입이며 (밑바탕 경우), *n* 이 *Nat* 타입의 값이라면 *Succ n* 도 *Nat* 타입의 값임을 (되도는 경우) 나타낸다. 뻔히 드러나 보이지는 않지만, 이 선언은 또한 *Nat* 타입의 생성자^{constructor} 는 *Zero* 와 *Succ* 뿐이라는 사실도 나타낸다. 따라서 *Nat* 타입의 값들을 다음과 같이 늘어놓을 수 있다.

```
Zero
Succ Zero
Succ (Succ Zero)
Succ (Succ (Succ Zero))
...
...
```

이 장에서는, 논의를 간단히 하고자 *Zero* 및 *Succ* 를 유한 번 적용해 얻을 수 있는 유한한 자연수만 다룬다. 더 자세히 말하자면, *inf* = *Succ inf* 로 정의할 수 있는 무한은 고려하지 않는다. 마찬가지로, 이 장에서 다루는 다른 되도는 타입들의 경우도 유한한 값만 다룬겠다.

이제 어떤 성질 *p* 가 모든 (유한한) 자연수에 대해 성립함을 보이려 한다 치자. 귀납 원리에 따르면, 밑바탕 경우인 *Zero* 일 때 *p* 가 성립하며, 귀납하는 경우^{inductive case} 인 *Succ* 를 적용했을 때도 *p* 가 보존됨을 보이기만 하면 된다. 귀납하는 경우를 더 정확히 설명하자면, 성질 *p* 가 임의의 자연수 *n* 에 대해 성립한다는 귀납 가정^{induction hypothesis} 아래 *Succ n* 에 대해서도 성립함을 보여야 한다.

어째서 귀납법을 쓰면 성질 p 가 모든 자연수에 대해 성립함을 보일 수 있게 되는 것일까? 예컨대, 귀납법으로 터 $Succ$ ($Succ$ Zero)에 대해 p 가 성립한다는 사실은 어떻게 이끌어낼 수 있을까? $Zero$ 에 대해 p 가 성립한다는 밑바탕 경우로부터 시작해, $n = Zero$ 로 놓고 귀납하는 경우를 한번 적용하면 $Succ$ Zero에 대해 p 가 성립함을 알 수 있으며, $n = Succ$ Zero로 놓고 귀납하는 경우를 한번 더 적용하면 $Succ$ ($Succ$ Zero)에 대해서도 p 가 성립함을 알 수 있다. 이와 같은 방법으로 어떤 자연수에 대해서든 p 가 성립함을 알 수 있다.

“도미노 효과”^{domino effect}에 빗대 귀납법을 설명하면 이해하기 쉽다. 도미노가 줄줄이 늘어서 있을 때, 첫째 도미노가 넘어질 것이라는 사실과, 또한 어떤 도미노가 넘어지면 그 바로 다음 도미노도 넘어진다는 사실을 알고 있다고 하자. 그렇다면, 첫째 도미노가 넘어지는 것으로부터 시작해 이웃하는 도미노가 하나하나 뒤따라 넘어진다는 원리를 계속해서 적용해 나가면, 모든 도미노가 다 넘어지는 것이 당연하다. $Zero$ 에 대해 어떤 성질이 성립하며 (첫째 도미노가 넘어지며), $Succ$ 을 적용해도 그 성질이 계속해서 성립한다면 (어떤 도미노가 넘어지면 그 다음 도미노도 넘어진다면), 결과적으로 모든 자연수에 대해 그 성질이 성립함(모든 도미노가 넘어짐)을 알 수 있다.

구체적인 보기로써, 두 자연수를 더하는 함수의 다음과 같은 되도는 함수 정의를 살펴보자.

$$\begin{aligned} add & :: Nat \rightarrow Nat \rightarrow Nat \\ add\ Zero\ m & = m \\ add\ (Succ\ n)\ m & = Succ\ (add\ n\ m) \end{aligned}$$

첫째 등식을 보면 모든 자연수 m 에 대해 $add\ Zero\ m = m$ 가 성립함을 대번에 알 수 있다. 이제 그와 맞짝을 이루는 성질인 $add\ n\ Zero = n$ 도 모든 자연수 n 에 대해 성립함을 보이자. 이 성질을 줄여서 p 라 부르고, n 에 대한 귀납법으로 증명하겠다. 밑바탕 경우인 p Zero, 곧 $add\ Zero\ Zero = Zero$ 임을 보이는 것은 다음처럼 간단하다.

$$\begin{aligned} & add\ Zero\ Zero \\ = & \{ \text{add 를 적용 } \} \\ & Zero \end{aligned}$$

귀납하는 경우, p 가 임의의 자연수 n 에 대해 성립하면 $p (Succ n)$ 에 대해서도 성립함을 보여야 한다. 즉, 귀납 가정 $add n Zero = n$ 이 성립하면 $add (succ n) Zero = Succ n$ 가 성립함을 다음과 같은 계산과정으로 보일 수 있다,

$$\begin{aligned} & add (Succ n) Zero \\ &= \{ add 를 적용 \} \\ & Succ (add n Zero) \\ &= \{ 귀납 가정 \} \\ & Succ n \quad \square \end{aligned}$$

귀납법으로 증명할 때 보통 여러 번의 계산과정을 보이게 되므로 증명이 끝났다는 것을 잘 드러나 보이게 하는 것이 좋다. 이를 위해, 정사각형 \square 를 위에서처럼 오른쪽 여백에다 쓰겠다.

또 하나의 보기로서, 자연수의 덧셈에 대해 교환법칙이 성립함을 보이자. 즉, $add x (add y z) = add (add x y) z$ 가 모든 자연수 x, y, z 에 대해 성립함을 보이자는 뜻이다. 그런데, 이 세 인자 중 어느 것에 대해 귀납법을 써야 할까? add 함수를 첫째 인자에 대한 패턴 매칭으로 정의했으므로, 교환법칙을 나타내는 등식에서 x 가 첫째 인자로 두 번 쓰이므로, 첫째 인자로 한 번만 쓰이는 y 나 아예 첫째 인자로 쓰이지도 않는 z 보다는, x 에 대해 귀납법을 시도하는 것이 자연스럽다는 것에 주목하라. x 에 대한 귀납법으로 add 의 교환법칙을 다음과 같이 증명할 수 있다.

밑바탕 경우:

$$\begin{aligned} & add Zero (add y z) \\ &= \{ 바깥쪽 add 를 적용 \} \\ & add y z \\ &= \{ add 를 역적용 \} \\ & add (add Zero y) z \end{aligned}$$

귀납하는 경우:

$$\begin{aligned}
 & add(Succ x)(add y z) \\
 = & \quad \{ \text{바깥쪽 } add \text{ 를 적용 } \} \\
 & Succ(add x (add y z)) \\
 = & \quad \{ \text{귀납 가정} \} \\
 & Succ(add (add x y) z) \\
 = & \quad \{ \text{바깥쪽 } add \text{ 를 역적용 } \} \\
 & add(Succ(add x y)) z \\
 = & \quad \{ \text{안쪽 } add \text{ 를 역적용 } \} \\
 & add(add(Succ x) y) z \quad \square
 \end{aligned}$$

두 경우 모두 정의를 적용하여 시작해서 정의를 역적용하며 끝맺는다. 귀납법을 쓰는 증명에서 이런 식의 계산과정을 유도하는 것을 흔히 볼 수 있는데, 아마도 처음 볼 때는 마지막 부분이 뭔가 이상아름하게 보일 것이다. 특히, 어떤 정의를 역적용할지 알아내는데 예지력이 필요한 것처럼 보이기 때문이다. 하지만 실제로 증명을 하다 보면, 이러한 계산과정을 유도하다가 막힐 때 반대로 원하는 최종 결과에 초점을 맞추고 거꾸로 계산 과정을 거슬러 올라가다 보면 그 막힌 곳에서 다시 만나는 식으로 증명을 진행할 수 있는 경우가 많다.

예컨대, 위에서 귀납 가정을 귀납하는 경우에 적용해 $Succ(add(add x y) z)$ 까지 계산을 진행하고 나면, 마땅히 적용할 정의가 없어 그 다음이 막막해 보인다. 하지만, 얻고자 하는 식 $add(add(Succ x) y) z$ 에 초점을 맞춰 생각하면 그저 안쪽 add 와 바깥쪽 add 를 적용함으로써 막혔던 바로 그 식을 얻을 수 있다. 따라서 막힌 부분부터 거꾸로 (적용 대신 역적용으로) 진행하면 원하는 계산과정을 얻는다.

지금까지 되도는 타입 Nat 에 대한 귀납법만 살펴보았지만, 같은 원리를 하스켈에서 기본적으로 제공하는 정수에도 적용할 수 있다. 더 구체적으로는, 어떤 성질 p 가 $n \geq 0$ 인 모든 정수 n 에 대해 성립함을 보이려면, 밑바탕 경우인 0 일 때 p 를 만족하고 귀납하는 경우인 p 가 n 보다 작거나 같은 모든 정수에 대해 만족하면 $n+1$ 에 대해서도 만족함을 보이면 된다.

예컨대, n 개의 같은 원소로 이루어진 리스트를 만들어 내는 라이브러리 함수 $replicate$ 의 다음 정의를 살펴보자.

```

replicate      :: Int → a → [a]
replicate 0    _ = []
replicate (n + 1) x = x : replicate n x

```

위 정의로부터 이 함수가 정말로 길이 n 인 리스트를 만들어 낸다는 사실, 곧 $\text{length}(\text{replicate } n x) = n$ 임을 보이려면 다음과 같이 $n \geq 0$ 인 n 에 대한 귀납법을 쓰면 쉽다.

밑바탕 경우:

$$\begin{aligned}
& \text{length}(\text{replicate } 0 x) \\
= & \quad \{ \text{replicate 를 적용 } \} \\
& \text{length} [] \\
= & \quad \{ \text{length 를 적용 } \} \\
& 0
\end{aligned}$$

귀납하는 경우:

$$\begin{aligned}
& \text{length}(\text{replicate } (n + 1) x) \\
= & \quad \{ \text{replicate 를 적용 } \} \\
& \text{length}(x : \text{replicate } n x) \\
= & \quad \{ \text{length 를 적용 } \} \\
& 1 + \text{length}(\text{replicate } n x) \\
= & \quad \{ \text{귀납 가정 } \} \\
& 1 + n \\
= & \quad \{ + \text{의 교환법칙 } \} \\
& n + 1
\end{aligned}
\quad \square$$

13.5 리스트에 대한 귀납법

귀납법은 자연수에만 쓰는 것이 아니라, 리스트를 포함한 그 밖에 모든 다른 되도는 타입에 대해 논증할 때도 쓴다. 자연수를 0부터 시작하여 그 다음 수를 구하는 함수를 계속 적용함으로써 되돌기로 만들어 나가듯, 리스트도 마찬가지로 빈 리스트로부터 시작해서 콘스^{cons} 연산자를 적용함으로써 만들어 나간다.

어떤 성질 p 가 모든 리스트에 대해 성립함을 보이고 싶다고 하자. 그러면 리스트에 대한 귀납 원리에 따라, 밑바탕 경우인 빈 리스트 []에 대해 성립하며 귀납하는 경우인 p 가 임의의 xs 에 대해 만족하면 $x:xs$ 에 대해서도 성립함을 보이기만 하면 된다. 물론, 이 때 원소 x 와 리스트 xs 모두 알맞은 타입을 가져야 한다.

보기로서, 이 장에서 앞서 정의한 *reverse* 함수의 역함수가 자기 자신이라는 성질, 곧 $\text{reverse}(\text{reverse } xs) = xs$ 임을 xs 에 대한 귀납법으로 증명해 보자. 밑바탕 경우는 다음과 같이 *reverse*의 정의를 적용하여 간단히 보일 수 있다.

$$\begin{aligned} & \text{reverse}(\text{reverse}[]) \\ = & \quad \{ \text{안쪽 reverse 를 적용 } \} \\ & \text{reverse}[] \\ = & \quad \{ \text{reverse 를 적용 } \} \\ & [] \end{aligned}$$

귀납하는 경우, $\text{reverse}(\text{reverse } xs) = xs$ 라는 가정 하에 $\text{reverse}(\text{reverse}(x:xs)) = x:xs$ 가 성립함을 다음과 같이 보일 수 있다.

$$\begin{aligned} & \text{reverse}(\text{reverse}(x:xs)) \\ = & \quad \{ \text{안쪽 reverse 를 적용 } \} \\ & \text{reverse}(\text{reverse } xs ++ [x]) \\ = & \quad \{ \text{분배법칙 - 아래를 보라 } \} \\ & \text{reverse}[x] ++ \text{reverse}(\text{reverse } xs) \\ = & \quad \{ \text{한원소 리스트 - 아래를 보라 } \} \\ & [x] ++ \text{reverse}(\text{reverse } xs) \\ = & \quad \{ \text{귀납 가정 } \} \\ & [x] ++ xs \\ = & \quad \{ ++ 를 적용 \} \\ & x:xs \end{aligned}$$

reverse 함수의 두 가지 성질을 이용하여 이 계산 과정의 진행을 도왔다. 우리가 앞서 보인 *reverse*가 한원소 리스트를 그대로 가져간다는 성질 $\text{reverse}[x] = [x]$, 그리고 이어붙이기 연산자에 대해 *reverse*가 분배될 때 두 인자 리스트의 순서가 뒤바뀌며 분배된다는 다음의 새로운 성질을 함께 이용했다.

$$\text{reverse}(xs ++ ys) = \text{reverse } ys ++ \text{reverse } xs$$

전문용어로는 이를 반변^{contravariant} 분배라 부른다. 이어붙이기 연산자 ++ 를 첫째 인자에 대한 패턴 매칭으로 정의했으므로 xs 에 대한 귀납법으로 이 성질에 대한 증명을 시도하는 것이 자연스럽다.

밑바탕 경우:

$$\begin{aligned} & \text{reverse} ([] \text{ ++ } ys) \\ = & \quad \{ \text{ ++ 를 적용 } \} \\ & \text{reverse } ys \\ = & \quad \{ \text{ ++ 의 성질 - 연습문제 5를 보라 } \} \\ & \text{reverse } ys \text{ ++ } [] \\ = & \quad \{ \text{ reverse 역작용 } \} \\ & \text{reverse } ys \text{ ++ reverse } [] \end{aligned}$$

귀납하는 경우:

$$\begin{aligned} & \text{reverse} ((x : xs) \text{ ++ } ys) \\ = & \quad \{ \text{ ++ 를 적용 } \} \\ & \text{reverse} (x : (xs \text{ ++ } ys)) \\ = & \quad \{ \text{ reverse 를 적용 } \} \\ & \text{reverse} (xs \text{ ++ } ys) \text{ ++ } [x] \\ = & \quad \{ \text{ 귀납 가정 } \} \\ & (\text{reverse } ys \text{ ++ reverse } xs) \text{ ++ } [x] \\ = & \quad \{ \text{ ++ 의 결합법칙 } \} \\ & \text{reverse } ys \text{ ++ } (\text{reverse } xs \text{ ++ } [x]) \\ = & \quad \{ \text{ 둘째 reverse 역작용 } \} \\ & \text{reverse } ys \text{ ++ reverse } (x : xs) \end{aligned}$$

이 계산 과정에서 쓰인 ++ 의 결합법칙은 앞서 add 의 결합법칙이 성립함을 보인 것과 마찬가지 방식으로 귀납법으로 증명할 수 있다 (이 장의 연습문제 5를 보라).

13.6 이어붙이기 연산자 없애기

많은 되도는 함수는 리스트를 이어붙이는 연산 ++ 로 정의하는 것이 자연스럽기는 하지만, 이 연산을 되돌며 계속 반복하는 것은 효율을 상당히 떨어뜨린다. 이 절에서는 귀납법으로 이러한 이어붙이기 연산을 없애고, 따라서 함수의 효율을 개선하는 방법을 알아본다. 그 첫 보기로서, 다음 함수 정의를 생각해 보자.

```
reverse      :: [a] → [a]
reverse []    = []
reverse (x : xs) = reverse xs ++ [x]
```

reverse 는 얼마나 효율적일까? 우선 첫째로, *xs* 와 *ys* 가 이미 완전히 계산된 값이라고 가정했을 때, *xs* ++ *ys* 의 값을 구하려면 *xs* 의 길이보다 하나 더 많은 횟수만큼 식을 줄여야 한다는 것을 쉽게 알 수 있다. 따라서, ++ 는 첫째 인자의 길이에 대한 선형 시간복잡도라고 말한다. 다음으로는, *xs* 의 길이가 길이가 n 일 때 *reverse xs* 를 계산하는 데는 1부터 $n + 1$ 까지의 정수의 합, 곧 $(n + 1)(n + 2)/2$ 번 식을 줄여야 한다는 것을 알 수 있다. 이 장의 첫 부분에서 보인 것처럼 $(n + 1)(n + 2)/2$ 를 괄호를 풀어 계산하면 $(n^2 + 3n + 2)/2$ 를 얻는다. 따라서 *reverse* 는 인자의 길이에 대해 2 차 시간복잡도라고 말한다.

2 차 시간복잡도는 좋지 않다. 예컨대, 만 개의 원소로 이루어진 리스트를 뒤집으려면 식을 대략 오천만번 정도 줄여야 한다. 하지만, 다행히 귀납법으로 *reverse* 의 정의에 나타나는 이어붙이기를 쉽게 없앨 수 있고, 따라서 효율을 높일 수 있다.

효율을 높이는 요령은 *reverse* 와 ++ 가 하는 일을 한꺼번에 하는 더 일반적인 함수를 정의하는 것이다. 구체적으로는, 다음 정의를 만족하는 되도는 함수 *reverse'* 를 정의해야 한다.

```
reverse' xs ys = reverse xs ++ ys
```

이는 $reverse'$ 를 두 리스트에 적용하면 첫째 리스트를 뒤집어 둘째 리스트에 이어붙인 리스트가 나온다는 것을 나타낸다. 이러한 함수를 정의할 수 있다면, $reverse$ 자체를 $reverse\ xs = reverse'\ xs\ []$ 로 다시 정의할 수 있으며, 이는 빈 리스트가 이어붙이기 연산의 항등원임을 이용한 것이다.

$reverse'$ 의 정의를 먼저 찾고 나서 그 정의가 위 등식을 만족함을 보이기보다는, 위 등식을 그러한 정의를 만들어 나가는 원동력으로 삼아 함수 정의를 만들어 나갈 수 있다. 구체적으로는, 위 등식을 xs 에 대한 귀납법으로 증명함으로써 정의를 만들어 나간다. 밑바탕 경우에서 $reverse'\ []\ ys$ 에 대한 정의를 나타내는 등식을 얻으며, 귀납하는 경우에서 $reverse'\ (x : xs)\ ys$ 에 대한 정의를 얻는다.

밑바탕 경우:

$$\begin{aligned} & reverse'\ []\ ys \\ = & \quad \{ reverse'\ 가 만족해야 할 성질 \} \\ & reverse\ [] \mathbin{+} ys \\ = & \quad \{ reverse\ 적용 \} \\ & [] \mathbin{+} ys \\ = & \quad \{ \mathbin{+} 적용 \} \\ & ys \end{aligned}$$

귀납하는 경우:

$$\begin{aligned} & reverse'\ (x : xs)\ ys \\ = & \quad \{ reverse'\ 가 만족해야 할 성질 \} \\ & reverse\ (x : xs) \mathbin{+} ys \\ = & \quad \{ reverse\ 를 적용 \} \\ & (reverse\ xs \mathbin{+} [x]) \mathbin{+} ys \\ = & \quad \{ \mathbin{+} 의 결합법칙 \} \\ & reverse\ xs \mathbin{+} ([x] \mathbin{+} ys) \\ = & \quad \{ 귀납\ 가정 \} \\ & reverse'\ xs\ ([x] \mathbin{+} ys) \\ = & \quad \{ \mathbin{+} 적용 \} \\ & reverse'\ xs \mathbin{+} (x : ys) \end{aligned}$$

□

위 증명으로부터 다음 정의를 유도할 수 있으며

$$\begin{aligned} \text{reverse}' &:: [a] \rightarrow [a] \rightarrow [a] \\ \text{reverse}' [] ys &= ys \\ \text{reverse}' (x : xs) ys &= \text{reverse}' xs (x : ys) \end{aligned}$$

이 정의는 이미 귀납법으로 보인 바와 같이 $\text{reverse}' xs ys = \text{reverse} xs ++ ys$ 를 만족한다. $\text{reverse}'$ 의 정의에 원래 reverse 함수나 이어붙이기 연산을 쓰지 않았다는 것에 주목하라. 따라서, 이제 reverse 자체를 다음과 같이 다시 정의할 수 있다.

$$\begin{aligned} \text{reverse} &:: [a] \rightarrow [a] \\ \text{reverse} xs &= \text{reverse}' xs [] \end{aligned}$$

예컨대, 다음과 같이 계산할 수 있다.

$$\begin{aligned} &\text{reverse} [1, 2, 3] \\ &= \{ \text{reverse} \text{ 를 적용 } \} \\ &\quad \text{reverse}' [1, 2, 3] [] \\ &= \{ \text{reverse}' \text{ 를 적용 } \} \\ &\quad \text{reverse}' [2, 3] (1 : []) \\ &= \{ \text{reverse}' \text{ 를 적용 } \} \\ &\quad \text{reverse}' [3] (2 : (1 : [])) \\ &= \{ \text{reverse}' \text{ 를 적용 } \} \\ &\quad \text{reverse}' [] (3 : (2 : (1 : []))) \\ &= \{ \text{reverse}' \text{ 를 적용 } \} \\ &\quad (3 : (2 : (1 : []))) \end{aligned}$$

최종 결과를 쌓아나가는 인자를 하나 더 두어 리스트를 뒤집은 것이다. 새로 정의한 reverse 는 원래 판보다 이해하기가 약간 더 어려울지는 몰라도 효율은 훨씬 좋다. 더 자세히 이야기하자면, xs 의 길이가 n 일 때 새로운 정의를 쓰면 식 $\text{reverse} xs$ 를 $n + 2$ 번만 줄이면 그 값을 구할 수 있다. 따라서 새로 정의한 reverse 는 그 인자의 길이에 대해 시간복잡도가 선형이다. 이를테면 만 개의 원소로 이루어진 리스트를 뒤집는 대는 대략 만 걸음이 든다는 이야기인데, 원래 정의로 계산했을 때 오천만 걸음 정도 들던 것과 견주어 볼 때 효율이 엄청나게 나아진다.

이미 7장과 12장에서 함수를 주고받는^{higher-order} 라이브러리 함수인 *foldl*를 살펴보면서, 쌍개를 써서 함수의 효율을 어떻게 개선하는지 알아본 바 있다. 예컨대, *reverse*를 쌍개를 쓰도록 하려면 $\text{reverse} = \text{foldl} (\lambda xs x \rightarrow x : xs) []$ 로 정의하면 간단하다. 하지만, 마찬가지 동작을 하는 함수 정의를 귀납법으로는 어떻게 만들 수 있는지 알아보는 것은 여전히 유익하다.

이어붙이기 연산을 없애는 또 하나의 보기로서, 다음과 같은 두갈래나무 타입을 살펴보자. 이 보기 통해 알게 되겠지만, 나무같은 구조에도 역시 귀납법을 쓸 수 있다.

```
data Tree = Leaf Int | Node Tree Tree
flatten :: [Tree] → [Int]
flatten (Leaf n) = [n]
flatten (Node l r) = flatten l ++ flatten r
```

flatten 함수는 이어붙이기 연산을 쓰기 때문에 효율적이지 않다. 이제, *reverse*에서와 같은 요령으로 더 효율을 개선해 보자. 즉, *flatten* 함수와 ++ 가 하는 일을 한꺼번에 하는 더 일반적인 함수, 곧 다음 성질을 만족하는 *flatten'* 함수를 정의해야 한다.

```
flatten' t ns = flatten t ++ ns
```

모든 나무에 대해서 어떤 성질이 성립함을 보이려면, *Tree* 타입에 대한 귀납 원리에 따라, *Leaf n* 형태의 모든 나무에 대해 성립하며, 또한 임의의 두 나무 *l*과 *r*이 그 성질을 만족하면 *Node l r*에 대해서도 성립함을 보이기만 하면 된다. 이러한 원리에 따라 위 성질을 만족하는 *flatten'*의 정의를 다음과 같이 만들어 나갈 수 있다.

밑바탕 경우:

$$\begin{aligned} & \text{flatten}' (\text{Leaf } n) ns \\ = & \quad \{ \text{ flatten}' \text{의 성질 } \} \\ = & \quad \text{flatten} (\text{Leaf } n) ++ ns \\ = & \quad \{ \text{ flatten } \text{을 적용 } \} \\ = & \quad [n] ++ ns \\ = & \quad \{ \text{ ++ 를 적용 } \} \\ & n : ns \end{aligned}$$

귀납하는 경우:

$$\begin{aligned}
 & \text{flatten}' (\text{Node } l r) ns \\
 = & \quad \{ \text{ flatten}' \text{의 성질, flatten 적용 } \} \\
 & (\text{flatten } l + \text{flatten } r) + ns \\
 = & \quad \{ + \text{ 결합법칙 } \} \\
 & \text{flatten } l + (\text{flatten } r + ns) \\
 = & \quad \{ l \text{에 귀납 가정 적용 } \} \\
 & \text{flatten}' l (\text{flatten } r + ns) \\
 = & \quad \{ r \text{에 귀납 가정 적용 } \} \\
 & \text{flatten}' l (\text{flatten}' r ns)
 \end{aligned}$$

이를 종합하면 다음 정의를 얻으며,

$$\begin{aligned}
 \text{flatten}' & :: \text{Tree} \rightarrow [\text{Int}] \rightarrow [\text{Int}] \\
 \text{flatten}' (\text{Leaf } n) \ ns & = n : ns \\
 \text{flatten}' (\text{Node } l r) \ ns & = \text{flatten}' l (\text{flatten}' r ns)
 \end{aligned}$$

이 정의는 *flatten'*이 만족해야 할 성질을 모두 만족한다. 따라서 *flatten* 함수를 다음과 같이 다시 정의할 수 있다.

$$\begin{aligned}
 \text{flatten} & :: \text{Tree} \rightarrow [\text{Int}] \\
 \text{flatten } t & = \text{flatten}' t []
 \end{aligned}$$

이번에도, 새로 정의한 *flatten* 함수가 원래 판보다 이해하기가 약간 더 어려울지는 몰라도, 이어붙이기 연산 대신 인자를 하나 더 써서 최종 결과를 쌓아나가므로 효율이 훨씬 좋다.

13.7 번역기 정확성 compiler correctness

더 큰 보기로서 이 장을 끝맺고자 한다. 10장에서 살펴본 정수와 덧셈 연산자로 이루어진 산술식 타입 및 그런 식을 직접 정수값으로 계산하는 (여기서는 *eval*이라 부르는) 함수를 다시 살펴보자.

```
data Expr = Val Int | Add Expr Expr
eval          :: Expr → Int
eval (Val n)   = n
eval (Add x y) = eval x + eval y
```

이러한 식을 스택^{stack}으로 돌리는 부호^{code}를 써서 간접적으로 계산할 수도 있다. 여기서 스택은 그냥 정수 리스트로, 부호는 스택에 넣기^{push} 연산과 더하기^{add} 연산으로 이루어진다.

```
type Stack = [Int]
type Code  = [Op]
data Op    = PUSH Int | ADD deriving Show
```

이러한 부호가 뜻하는 바는, 처음 상태로 주어진 스택으로부터 코드를 돌려 마지막 상태의 스택을 얻는 다음과 같은 함수로써 나타낸다.

```
exec          :: Code → Stack → Stack
exec []        s      = s
exec (PUSH n : c) s  = exec c (n : s)
exec (ADD : c)   (m : n : s) = exec c (n + m : s)
```

즉, 넣기^{push} 연산으로 스택의 맨 위에 새로 정수를 올려놓고, 더하기^{add} 연산으로 스택의 맨 위에 있는 두 정수의 합을 구해 원래 두 정수 대신 올려놓는다. 산술식을 부호로 번역하는 함수를 이러한 연산으로 쉽게 만들 수 있다. 정수값은 그냥 그 값을 스택에 넣는 것으로 번역하고, 덧셈식은 먼저 두 인자 *x*와 *y*를 번역하고 스택 맨 위에 나타나는 두 정수를 더하는 것으로 번역하면 된다.

$$\begin{aligned} \text{comp} &:: \text{Expr} \rightarrow \text{Code} \\ \text{comp } (\text{Val } n) &= [\text{PUSH } n] \\ \text{comp } (\text{Add } x \ y) &= \text{comp } x + \text{comp } y + [\text{ADD}] \end{aligned}$$

예컨대, $e :: \text{Expr}$ 가 식 $(2+3)+4$ 를 나타낼 때 앞서 정의한 세 함수들이 다음과 같이 동작함을 확인할 수 있다.

```
> eval e  
9
```

```
> comp e  
[PUSH 2, PUSH 3, ADD, PUSH 4, ADD]
```

```
> exec (comp e) []  
[9]
```

위 보기와 일반화하여, 산술식 번역기의 정확성을 다음 등식으로 표현할 수 있다.

$$\text{exec } (\text{comp } e) [] = [\text{eval } e]$$

즉, 식을 번역한 결과로 나온 코드를 빈 스택으로부터 돌려 얻는 결과는, 식을 직접 계산한 결과를 한원소 스택으로 만든 것과 같다라는 뜻이다. 위 등식을 증명하려면, 임의의 초기 스택으로부터 시작하도록 다음과 같이 일반화해야 함을 곧 알게 될 것이다.

$$\text{exec } (\text{comp } e) s = \text{eval } e : s$$

따라하기

다음은 *compiler.lhs* 스크립트를 불러와 실행해 본 화면이다.

The screenshot shows the WinHugs Haskell interpreter window. The title bar says "WinHugs". The menu bar includes "File", "Edit", "Actions", "Browse", and "Help". The toolbar has icons for file operations like Open, Save, and Print. The main pane displays the following text:

```

Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-2005
World Wide Web: http://haskell.org/hugs
Bugs: http://hackage.haskell.org/trac/hugs

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type ?: for help
Main> eval ((Val 2 `Add` Val 3) `Add` Val 4)
9
Main> comp ((Val 2 `Add` Val 3) `Add` Val 4)
[PUSH 2,PUSH 3,ADD,PUSH 4,ADD]
Main> exec (comp ((Val 2 `Add` Val 3) `Add` Val 4)) []
[9]
Main> comp ((Val 2 `Add` Val 3) `Add` Val 4) `exec` []
[9]
Main>

```

알아보기 편하라고 $Add (Add (Val 2) (Val 3)) (Val 4)$ 대신 역따옴표로 Add 를 감싸 두 인자 사이에 써서 마치 덧셈 연산자와 같이 읽기 편하도록 $(Val 2 `Add` Val 3) `Add` Val 4$ 로 표현한 것을 눈여겨 보라. Add 와 같은 생성자도 함수와 마찬가지로 역따옴표로 감싸면 두 인자 사이에 쓸 수 있게 된다.

따라하기 끝 *Expr* 타입에 대한 귀납법은 생성자의 이름이 다르다는 것만 빼면 바로 앞 절의 *Tree* 타입에 대한 귀납법과 같다. *Expr* 타입에 대한 귀납법으로 이 번역기의 정확성을 다음과 같이 보일 수 있다.

밑바탕 경우:

$$\begin{aligned}
 & \text{exec (comp (Val n)) } s \\
 = & \quad \{ \text{comp 를 적용 } \} \\
 & \text{exec [PUSH } n \text{] } s \\
 = & \quad \{ \text{exec 를 적용 } \} \\
 & n : s \\
 = & \quad \{ \text{eval 역작용 } \} \\
 & \text{eval (Val n)} : s
 \end{aligned}$$

귀납하는 경우:

$$\begin{aligned}
 & \text{exec (comp (Add } x y)) s \\
 = & \quad \{ \text{ comp 를 적용 } \} \\
 & \text{exec (comp } x ++ \text{comp } y ++ [\text{ADD}]) s \\
 = & \quad \{ ++ \text{의 결합법칙 } \} \\
 & \text{exec (comp } x ++ (\text{comp } y ++ [\text{ADD}])) s \\
 = & \quad \{ \text{ 분배법칙 (아래 설명 참조) } \} \\
 & \text{exec (comp } y ++ [\text{ADD}]) (\text{exec (comp } x) s) \\
 = & \quad \{ x \text{에 대한 귀납 가정 } \} \\
 & \text{exec (comp } y ++ [\text{ADD}]) (\text{eval } x : s) \\
 = & \quad \{ \text{ 분배법칙 다시 적용 } \} \\
 & \text{exec } [\text{ADD}] (\text{exec (comp } y) (\text{eval } x : s)) \\
 = & \quad \{ y \text{에 대한 귀납 가정 } \} \\
 & \text{exec } [\text{ADD}] (\text{eval } y : \text{eval } x : s) \\
 = & \quad \{ \text{ exec 를 적용 } \} \\
 & \text{eval } y + \text{eval } x : s \\
 = & \quad \{ \text{ eval 역적용 } \} \\
 & \text{eval (Add } x y) : s
 \end{aligned}$$

코드를 돌린 결과로 임의의 스택이 나올 수 있도록 일반화하지 않았더라면 두 번째 귀납 가정을 적용할 수 없었을 것이다. 왜냐하면 그 지점에서 스택이 비어 있지 않기 때문이다. 귀납하는 경우에 쓰인 분배법칙은 두 조각의 코드를 하나로 이어붙여 실행하는 것과 따로따로 차례대로 계산한 결과가 같음을 나타낸다.

$$\text{exec (c ++ d) } s = \text{exec d } (\text{exec c } s)$$

이 성질은 부호 c 에 대한 귀납법으로 증명하는데, 귀납하는 경우에는 부호의 첫 연산이 스택에 넣기^{push} 일 때와 더하기^{add} 일 때로 각각 나누어 생각해야 한다.

밑바탕 경우:

$$\begin{aligned}
 & \text{exec } ([] ++ d) s \\
 = & \quad \{ ++ \text{를 적용 } \} \\
 & \text{exec } d s \\
 = & \quad \{ \text{ exec 역적용 } \} \\
 & \text{exec } d (\text{exec } [] s)
 \end{aligned}$$

귀납하는 경우:

$$\begin{aligned}
 & \text{exec } ((PUSH\ n : c) \uparrow\downarrow d) s \\
 = & \quad \{ \uparrow\downarrow 를 적용 \} \\
 & \text{exec } (PUSH\ n : (c \uparrow\downarrow d)) s \\
 = & \quad \{ \text{exec 를 적용} \} \\
 & \text{exec } (c \uparrow\downarrow d) (n : s) \\
 = & \quad \{ \text{귀납 가정} \} \\
 & \text{exec } d (\text{exec } c (n : s)) \\
 = & \quad \{ \text{exec 역작용} \} \\
 & \text{exec } d (\text{exec } (PUSH\ n : c) s)
 \end{aligned}$$

귀납하는 경우:

$$\begin{aligned}
 & \text{exec } ((ADD : c) \uparrow\downarrow d) s \\
 = & \quad \{ \uparrow\downarrow 를 적용 \} \\
 & \text{exec } (ADD : (c \uparrow\downarrow d)) s \\
 = & \quad \{ s 가 m : n : s' 의 형태라고 가정 \} \\
 & \text{exec } (ADD : (c \uparrow\downarrow d)) (m : n : s') \\
 = & \quad \{ \text{exec 를 적용} \} \\
 & \text{exec } (c \uparrow\downarrow d) (n + m : s') \\
 = & \quad \{ \text{귀납 가정} \} \\
 & \text{exec } d (\text{exec } c (n + m : s')) \\
 = & \quad \{ \text{exec 역작용} \} \\
 & \text{exec } d (\text{exec } (ADD : c) (m : n : s'))
 \end{aligned}$$

둘째 귀납하는 경우에서 생각하지 않은 스택 상태는 바로 스택이 모자라는 stack underflow 잘못이 날 때다. 실제로는 이런 일이 결코 일어나지 않는데, 왜냐하면 번역기의 구조상 더하기 연산을 하기 전에 스택에 항상 적어도 두 개의 정수가 놓이게 되어 있기 때문이다.

한편, 앞 절에서와 같이 이어붙이기 연산을 없애는 기법을 적용하면, 분배법칙 증명이나 스택 모자람에 대한 문제를 한꺼번에 해결할 수 있다. 더 구체적으로는, 다음 성질을 만족하는 더 일반적인 comp' 함수를 정의하면 된다.

$$\text{comp}' e c = \text{comp } e \uparrow\downarrow c$$

e 에 대한 귀납법으로 다음과 같은 함수 정의를 만들어낼 수 있으며,

$$\begin{aligned} \text{comp}' &:: \text{Expr} \rightarrow \text{Code} \rightarrow \text{Code} \\ \text{comp}' (\text{Val } n) c &= \text{PUSH } n : c \\ \text{comp}' (\text{Add } x y) c &= \text{comp}' x (\text{comp}' y (\text{ADD} : c)) \end{aligned}$$

이를 이용해 원래 번역기를 $\text{comp } e = \text{comp}' e []$ 로 다시 정의할 수 있으며, 이 새로 정의한 번역기가 원래 프로그램의 의미 semantics 를 따름을 다음 등식으로 나타낼 수 있다.

$$\text{exec} (\text{comp}' e c) s = \text{exec } c (\text{eval } e : s)$$

이는 식을 번역한 부호에 또 다른 임의의 부호를 덧붙여 돌린 결과는 그 식을 직접 계산한 결과를 원래 스택의 맨 위에 올려놓고 덧붙이려던 부호를 돌린 것과 같다는 것을 나타낸다. 위 등식을 식 e 에 대한 귀납법으로 증명하면 다음과 같다.

밑바탕 경우:

$$\begin{aligned} &\text{exec} (\text{comp}' (\text{Val } n) c) s \\ &= \{ \text{comp}' \text{를 적용 } \} \\ &\text{exec} (\text{PUSH } n : c) s \\ &= \{ \text{exec} \text{를 적용 } \} \\ &\text{exec } c (n : s) \\ &= \{ \text{eval} \text{을 역적용 } \} \\ &\text{exec } c (\text{eval} (\text{Val } n) : s) \end{aligned}$$

귀납하는 경우:

$$\begin{aligned} &\text{exec} (\text{comp}' (\text{Add } x y) c) s \\ &= \{ \text{comp}' \text{를 적용 } \} \\ &\text{exec} (\text{comp}' x (\text{comp}' y (\text{ADD} : c))) s \\ &= \{ x \text{에 대한 귀납 가정 } \} \\ &\text{exec} (\text{comp}' y (\text{ADD} : c)) (\text{eval } x : s) \\ &= \{ y \text{에 대한 귀납 가정 } \} \\ &\text{exec} (\text{ADD} : c) (\text{eval } y : \text{eval } x : s) \\ &= \{ \text{exec} \text{를 적용 } \} \\ &\text{exec } c ((\text{eval } x + \text{eval } y) : s) \\ &= \{ \text{eval} \text{을 역적용 } \} \\ &\text{exec } c (\text{eval} (\text{Add } x y) : s) \end{aligned}$$

여기서 $s = c = []$ 로 놓고 식을 간단히 하면, 원래 증명하려 했던 번역기 정확성 $\text{exec}(\text{comp } e) [] = [\text{eval } e]$ 를 얻는다. 정확성 증명에서 스택이 모자라는 경우를 따지지 않아도 된다는 것 말고도, 쌍개를 쓰는 번역기 판은 좋은 점이 두 가지 더 있다. 우선 첫째로, $\#$ 를 사용하지 않기 때문에 효율이 더 좋다. 그리고 둘째로, 증명의 길이가 원래 두 부분으로 나눠 증명한 것을 합친 길이의 반도 되지 않는다. 형식적 증명에서 흔히 그러하듯, 증명하고자 하는 결과를 적절히 일변화하면 증명 과정이 훨씬 간단해진다. 수학은 간단한 증명으로 효율적인 프로그램 개발을 돋는 아주 좋은 도구다!

스크립트 *compiler.lhs* 한눈에 다시보기

```

1 Programming in Haskell 13.7절 번역기 보기,
2 Graham Hutton, Cambridge University Press, 2007.
3
4 간단한 산술식 번역기
5 -----
6
7 > data Expr          = Val Int | Add Expr Expr
8 >
9 > eval               :: Expr -> Int
10 > eval (Val n)       = n
11 > eval (Add x y)    = eval x + eval y
12 >
13 > type Stack         = [Int]
14 >
15 > type Code          = [Op]
16 >
17 > data Op             = PUSH Int | ADD deriving Show
18 >
19 > exec               :: Code -> Stack -> Stack
20 > exec []           s      = s
21 > exec (PUSH n : c) s    = exec c (n:s)
22 > exec (ADD      : c) (m:n:s) = exec c (n+m:s)
23 >
24 > comp'              :: Expr -> Code -> Code
25 > comp' (Val n)     c      = PUSH n : c
26 > comp' (Add x y)   c      = comp' x (comp' y (ADD : c))
27 >
28 > comp               :: Expr -> Code
29 > comp e             = comp' e []
30

```

13.8 살펴보기

함수형 프로그램의 논리적 증명은 그 자체로 책 한 권을 충분히 쓰고도 남을 만한 주제로, 여기서는 수박 겉핥기만 했을 뿐이다. 모든 경우에 대해 다 정의되지 않은 값이나 무한한 구조에 대한 논증을 포함하는 더 심도 있는 주제는 (5;8)을, 관계를 중심으로 표현하며 인자를 생략하는 point-free 방법에 대해서는 (1)을, 프로그램 성질 자동 검사는 (4)를, 효과 effect에 대한 증명은 (24)를, 그리고 귀납법을 피하는 기법은 (14)를 참조하라. 번역기에 대한 보기는 (18)에서 가져왔으며, “이어붙이기를 없앤다”는 표현은 (30)에서 유래했다.

13.9 연습문제

1. 부록 A에 있는 표준 라이브러리 함수들 중 서로 겹치는 패턴으로 정의된 것을 찾아보라.
2. n 에 대한 귀납법으로 $\text{add } n (\text{Succ } m) = \text{Succ} (\text{add } n m)$ 임을 보이라.
3. 위 성질과 $\text{add } n \text{ Zero} = n$ 를 써서, 덧셈의 교환법칙 $\text{add } n m = \text{add } m n$ 를 n 에 대한 귀납법으로 증명하라.
4. 리스트의 모든 원소가 어떤 조건 predicate 을 만족하는지 알아보는 다음과 같은 라이브러리 함수 정의를 써서

$$\begin{aligned} \text{all } p [] &= \text{True} \\ \text{all } p (x : xs) &= p x \wedge \text{all } p xs \end{aligned}$$

replicate 가 올바르게 정의되었다는 것을 먼저 보이라. *replicate* 함수가 만들어내는 리스트의 원소들이 모두 같은 값이라는 성질, 곧 $\text{all } (= x) (\text{replicate } n x)$ 가 성립함을 $n \geq 0$ 인 정수 n 에 대한 귀납법으로 보이면 된다.

귀띔: 증명하고자 하는 성질이 항상 *True* 값을 가진다는 것을 보이라.

5. 다음 정의에 대해

$$\begin{aligned} [] \mathbin{++} ys &= ys \\ (x : xs) \mathbin{++} ys &= x : (xs \mathbin{++} ys) \end{aligned}$$

아래 두 성질이 성립함을 xs 에 대한 귀납법으로 보이라.

$$\begin{aligned} xs \mathbin{++} [] &= xs \\ xs \mathbin{++} (ys \mathbin{++} zs) &= (xs \mathbin{++} ys) \mathbin{++} zs \end{aligned}$$

귀띔: *add* 함수에 대한 증명과 비슷하다.

6. 등식 $\text{reverse} (\text{reverse } xs) = xs$ 를 보조 정리 $\text{reverse} (xs \mathbin{++} [x]) = x : \text{reverse } xs$ 하나만 써서 증명할 수도 있으며, 이 보조 정리 역시 xs 에 대한 귀납법으로 증명 가능하다. 이 장에서는 세 가지 보조 정리를 이용해 증명했는데, 어떤 점에서 그런 방법이 더 좋은가?

7. 다음 정의를 가지고

$$\begin{aligned} \text{map } f [] &= [] \\ \text{map } f (x : xs) &= f x : \text{map } f xs \\ (f \circ g) x &= f (g x) \end{aligned}$$

xs 에 대한 귀납법으로 $\text{map } f (\text{map } g xs) = \text{map } (f \circ g) xs$ 임을 보이라.

8. 위 ++ 의 정의와 다음의 정의를 함께 고려하여

$$\begin{aligned} \text{take } 0 &-&= [] \\ \text{take } (n + 1) [] &-&= [] \\ \text{take } (n + 1) (x : xs) &=& x : \text{take } n xs \\ \text{drop } 0 &-&= xs \\ \text{drop } (n + 1) [] &-&= [] \\ \text{drop } (n + 1) (x : xs) &=& \text{drop } n xs \end{aligned}$$

$\text{take } n xs \text{ ++ drop } n xs = xs$ 임을 $n \geq 0$ 인 정수 n 과 리스트 xs 에 대한 귀납법을 같이 써서 증명하라.

귀띔: take 와 drop 의 정의에 나타나는 각각의 패턴에 대해 한 가지씩, 세 가지 경우로 나누어 생각하라.

9. 다음과 같은 나무 타입 선언을 갖고

```
data Tree = Leaf Int | Node Tree Tree
```

이파리^{leaf} 가 마디^{node}보다 항상 한개씩 더 많다는 것을 나무에 대한 귀납법으로 보이라.

귀띔: 우선 이파리 개수를 세는 함수와 마디 개수를 세는 함수를 정의하라.

10. 등식 $\text{comp}' e c = \text{comp } e \text{ ++ } c$ 를 만족하는 comp' 함수의 정의를 c 에 대한 귀납법을 써서 만들어 나가 보라.

부록 A

표준 서막 standard prelude

이 부록에는 표준 서막에서 가장 많이 쓰는 클래스 및 함수 정의를 실었다. 정의를 간단하고 명료하게 보여주기 위해, 하스켈 보고서 Haskell Report (25)에 나온 정의를 간단히 하거나 좀 다르게 바꿔 정의한 것들도 있다.

A.1 클래스 class

같기 타입 equality type

```
class Eq a where
  (==), (≠) :: a → a → Bool
  x ≠ y      = ¬(x == y)
```

순서 타입 ordered type

```
class Eq a ⇒ Ord a where
  (<), (≤), (>) , (≥) :: a → a → Bool
  min, max :: a → a → a
  min x y | x ≤ y = x
            | otherwise = y
  max x y | x ≤ y = y
            | otherwise = x
```

보이는 타입 Showable type

```
class Show a where
  show :: a → String
```

읽히는 타입 Readable type

```
class Read a where
  read :: String → a
```

수치 타입 Numeric types

```
class (Eq a, Show a) ⇒ Num a where
  (+), (-), (*) :: a → a → a
  negate, abs, signum :: a → a
```

정수같은 타입 Integral type

```
class Num a ⇒ Integral a where
  div, mod :: a → a → a
```

분수같은 타입 Fractional types

```
class Num a ⇒ Fractional a where
  (/) :: a → a → a
  recip :: a → a
  recip n = 1 / n
```

모나드 타입 Monadic type

```
class Monad m where
  return :: a → m a
  (≫=) :: m a → (a → m b) → m b
```

A.2 논리값 logical value

타입 선언 type declaration

```
data Bool = False | True
deriving (Eq, Ord, Show, Read)
```

논리곱 logical conjunction

```
( ∧ ) :: Bool → Bool → Bool
False ∧ _ = False
True ∧ b = b
```

논리합 logical disjunction

```
( ∨ ) :: Bool → Bool → Bool
False ∨ b = b
True ∨ _ = True
```

논리 부정 logical negation

```
¬ :: Bool → Bool
¬ False = True
¬ True = False
```

항상 성립하는 보초

```
otherwise :: Bool
otherwise = True
```

A.3 글자^{character} 와 글줄^{string}

이 절에 있는 함수들은 표준 서막이 아니라 *Char* 모듈에서 정의한다.

타입 선언^{type declaration}

```
data Char =      -- 내용 생략
           deriving (Eq, Ord, Show, Read)
```

```
type String = [Char]
```

글자가 소문자인지 알아본다

```
isLower :: Char → Bool
isLower c = c ≥ 'a' ∧ c ≤ 'z'
```

글자가 대문자인지 알아본다

```
isUpper :: Char → Bool
isUpper c = c ≥ 'A' ∧ c ≤ 'Z'
```

글자가 알파벳인지 알아본다

```
isAlpha :: Char → Bool
isAlpha c = isLower c ∨ isUpper c
```

글자가 숫자인지 알아본다

```
isDigit :: Char → Bool
isDigit c = c ≥ '0' ∧ c ≤ '9'
```

글자가 알파벳 혹은 숫자인지 알아본다

```
isAlphaNum :: Char → Bool
isAlphaNum c = isAlpha c ∨ isDigit c
```

글자가 공백인지 알아본다

```
isSpace :: Char → Bool
isSpace c = elem c " \t\n"
```

글자를 유니코드 번호로 바꾼다

```
ord :: Char → Int
ord c = -- 내용 생략
```

유니코드 번호를 글자로 바꾼다

```
chr :: Int → Char
chr n = -- 내용 생략
```

숫자 글자를 정수로 바꾼다

```
digitToInt :: Char → Int
digitToInt c | isDigit c = ord c - ord '0'
```

정수를 숫자 글자로 바꾼다

```
intToDigit :: Int → Char
intToDigit n
| n ≥ 0 ∧ n ≤ 9 = chr (ord '0' + n)
```

글자를 소문자로 바꾼다

```
toLowerCase          :: Char → Char
toLowerCase c | isUpper c = chr (ord c − ord 'A' + ord 'a')
| otherwise = c
```

글자를 대문자로 바꾼다

```
toUpperCase        :: Char → Char
toUpperCase c | isLower c = chr (ord c − ord 'a' + ord 'A')
| otherwise = c
```

A.4 수number

타입 선언 type declaration

```
data Int      =      -- 내용 생략
                  deriving (Eq, Ord, Show, Read, Num, Integral)
data Integer =      -- 내용 생략
                  deriving (Eq, Ord, Show, Read, Num, Integral)
data Float    =      -- 내용 생략
                  deriving (Eq, Ord, Show, Read, Num, Fractional)
```

정수가 짝수인지 알아본다

```
even   :: Integral a ⇒ a → Bool
even n = n `mod` 2 == 0
```

정수가 홀수인지 알아본다

```
odd   :: Integral a ⇒ a → Bool
odd =  $\neg \circ$  even
```

거듭제곱

```
(↑)           :: (Num a, Integral b) ⇒ a → b → a
_↑ 0          = 1
x ↑ (n + 1) = x * (x ↑ n)
```

A.5 순서쌍 tuples

타입 선언 type declaration

```
data ()      =      -- 내용 생략
               deriving (Eq, Ord, Show, Read)
data (a, b)  =      -- 내용 생략
               deriving (Eq, Ord, Show, Read)
data (a, b, c) =      -- 내용 생략
               deriving (Eq, Ord, Show, Read)
:
:
```

순서쌍의 첫째 성분을 고른다

```
fst       :: (a, b) → a
fst (x, _) = x
```

순서쌍의 둘째 성분을 고른다

```
snd       :: (a, b) → b
snd (_, y) = y
```

A.6 애마도 Maybe

타입 선언 type declaration

```
data Maybe a = Nothing | Just a
    deriving (Eq, Ord, Show, Read)
```

A.7 리스트 list

타입 선언 type declaration

```
data [a] = [] | a : [a]
    deriving (Eq, Ord, Show, Read)
```

빈 리스트인지 알아본다

```
null      :: [a] → Bool
null []    = True
null (_ : _) = False
```

리스트의 원소인지 알아본다

```
elem      :: Eq a ⇒ a → [a] → Bool
elem x xs = any (== x) xs
```

리스트에 있는 논리값이 모두 True 인지 알아본다

```
and :: [Bool] → Bool
and = foldr (Λ) True
```

리스트에 있는 논리값 중 True 인 것이 있는지 알아본다

```
or :: [Bool] → Bool
or = foldr (∨) False
```

모든 원소가 조건 predicate 을 만족하는지 알아본다

$$\begin{aligned} all &:: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool} \\ all p &= \text{and} \circ \text{map } p \end{aligned}$$

조건 predicate 을 만족하는 원소가 있는지 알아본다

$$\begin{aligned} any &:: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool} \\ any p &= \text{or} \circ \text{map } p \end{aligned}$$

비어있지 않은 리스트에서 첫번째 원소를 고른다

$$\begin{aligned} head &:: [a] \rightarrow a \\ head (x : _) &= x \end{aligned}$$

비어있지 않은 리스트에서 마지막 원소를 고른다

$$\begin{aligned} last &:: [a] \rightarrow a \\ last [x] &= x \\ last (_ : xs) &= last xs \end{aligned}$$

비어있지 않은 리스트에서 n 번째 원소를 고른다

$$\begin{aligned} (!!)&:: [a] \rightarrow \text{Int} \rightarrow a \\ (x : _) !! 0 &= x \\ (_ : xs) !! (n + 1) &= xs !! n \end{aligned}$$

리스트에서 첫 n 개의 원소를 고른다

$$\begin{aligned} take &:: \text{Int} \rightarrow [a] \rightarrow [a] \\ take 0 _&= [] \\ take (n + 1) [] &= [] \\ take (n + 1) (x : xs) &= x : take n xs \end{aligned}$$

리스트에서 조건을 만족하는 모든 원소를 고른다

$$\begin{aligned} filter &:: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \\ filter p xs &= [x \mid x \leftarrow xs, p x] \end{aligned}$$

리스트에서 주어진 조건을 만족하는 한 계속 원소를 고른다

```
takeWhile      :: (a → Bool) → [a] → [a]
takeWhile [] = []
takeWhile p (x : xs)
| p x        = x : takeWhile p xs
| otherwise   = []
```

비어있지 않은 리스트에서 첫째 원소를 버린다

```
tail          :: [a] → [a]
tail (_ : xs) = xs
```

비어있지 않은 리스트에서 마지막 원소를 버린다

```
init          :: [a] → [a]
init []       = []
init (x : xs) = x : init xs
```

리스트에서 첫 n 개의 원소를 버린다

```
drop          :: Int → [a] → [a]
drop 0 xs     = xs
drop (n + 1) [] = []
drop (n + 1) (_ : xs) = drop n xs
```

리스트에서 주어진 조건을 만족하는 한 계속 원소를 버린다

```
dropWhile     :: (a → Bool) → [a] → [a]
dropWhile [] = []
dropWhile p (x : xs)
| p x        = dropWhile p xs
| otherwise   = x : xs
```

n 번째 원소 위치에서 리스트를 나눈다

```
splitAt       :: Int → [a] → ([a], [a])
splitAt n xs = (take n xs, drop n xs)
```

주어진 조건에 따라 리스트를 나눈다

```
span      :: (a → Bool) → [a] → ([a], [a])
span p xs = (takeWhile p xs, dropWhile p xs)
```

리스트를 오른쪽으로부터 묶이는 연산자로 처리한다

```
foldr      :: (a → b → b) → b → [a] → b
foldr _ v []     = v
foldr f v (x : xs) = f x (foldr f v xs)
```

비어있지 않은 리스트를 오른쪽으로부터 묶는 연산자로 처리한다

```
foldr1     :: (a → a → a) → [a] → a
foldr1 _ [x]   = x
foldr1 f (x : xs) = f x (foldr1 f xs)
```

리스트를 왼쪽으로부터 묶이는 연산자로 처리한다

```
foldl      :: (a → b → a) → a → [b] → a
foldl _ v []     = v
foldl f v (x : xs) = foldl f (f v x) xs
```

비어 있지 않은 리스트를 왼쪽으로부터 묶이는 연산자로 처리한다

```
foldl1     :: (a → a → a) → [a] → a
foldl1 f (x : xs) = foldl f x xs
```

똑같은 원소로 이루어진 무한한 리스트를 만든다

```
repeat    :: a → [a]
repeat x = xs where xs = x : xs
```

n 개의 똑같은 원소로 이루어진 리스트를 만든다

```
replicate  :: Int → a → [a]
replicate n = take n ∘ repeat
```

함수를 어떤 값에 계속 적용해 나가며 무한한 리스트를 만들어낸다

$$\begin{aligned} iterate &:: (a \rightarrow a) \rightarrow a \rightarrow [a] \\ iterate f x &= x : iterate f (f x) \end{aligned}$$

리스트의 순서쌍으로부터 순서쌍 리스트를 만들어낸다

$$\begin{aligned} zip &:: [a] \rightarrow [b] \rightarrow [(a, b)] \\ zip [] &= [] \\ zip _ &= [] \\ zip (x : xs) (y : ys) &= (x, y) : zip xs ys \end{aligned}$$

리스트의 길이를 구한다

$$\begin{aligned} length &:: [a] \rightarrow Int \\ length &= foldl (\lambda n _ \rightarrow n + 1) 0 \end{aligned}$$

리스트에 있는 수의 합을 구한다

$$\begin{aligned} sum &:: Num a \Rightarrow [a] \rightarrow a \\ sum &= foldl (+) 0 \end{aligned}$$

리스트에 있는 수의 곱을 구한다

$$\begin{aligned} product &:: Num a \Rightarrow [a] \rightarrow a \\ product &= foldl (*) 1 \end{aligned}$$

비어 있지 않은 리스트에서 가장 작은 값을 구한다

$$\begin{aligned} minimum &:: Ord a \Rightarrow [a] \rightarrow a \\ minimum &= foldl1 min \end{aligned}$$

비어 있지 않은 리스트에서 가장 큰 값을 구한다

$$\begin{aligned} maximum &:: Ord a \Rightarrow [a] \rightarrow a \\ maximum &= foldl1 max \end{aligned}$$

두 리스트를 이어붙인다

$$\begin{aligned} (+\!\!+\!) &:: [a] \rightarrow [a] \rightarrow [a] \\ [] +\!\!+ y &= ys \\ (x : xs) +\!\!+ ys &= x : (xs +\!\!+ ys) \end{aligned}$$

리스트로 이루어진 리스트를 이어붙인다

$$\begin{aligned} concat &:: [[a]] \rightarrow [a] \\ concat &= foldr (+\!\!+) [] \end{aligned}$$

리스트를 뒤집는다

$$\begin{aligned} reverse &:: [a] \rightarrow [a] \\ reverse &= foldl (\lambda xs x \rightarrow x : xs) [] \end{aligned}$$

주어진 함수를 리스트의 모든 원소에 적용한다

$$\begin{aligned} map &:: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\ map f xs &= [f x \mid x \leftarrow xs] \end{aligned}$$

A.8 힘수 function

타입 선언 type declaration

data $a \rightarrow b =$ -- 내용 생략

항등 힘수 identity function

$$\begin{aligned} id &:: a \rightarrow a \\ id &= \lambda x \rightarrow x \end{aligned}$$

함수 합성 function composition

$(\circ) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$
 $f \circ g = \lambda x \rightarrow f(g x)$

상수 함수 constant function

$const :: a \rightarrow (b \rightarrow a)$
 $const x = \lambda_- \rightarrow x$

간간한 적용 strict application

$(\$!) :: (a \rightarrow b) \rightarrow a \rightarrow b$
 $f \$! x = \quad \text{-- 내용 생략}$

순서쌍을 받는 함수를 커리된 함수로 바꾼다

$curry :: ((a, b) \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c)$
 $curry f = \lambda x y \rightarrow f(x, y)$

커리된 함수를 순서쌍을 받는 함수로 바꾼다

$uncurry :: (a \rightarrow b \rightarrow c) \rightarrow ((a, b) \rightarrow c)$
 $uncurry f = \lambda(x, y) \rightarrow f x y$

A.9 입출력 input/output

타입 선언 type declaration

`data IO a = \quad \text{-- 내용 생략}`

글쇠판으로부터 글자를 읽어들인다

`getChar :: IO Char`
`getChar = \quad \text{-- 내용 생략}`

글쇠판에서 글줄을 읽어들인다

```
getLine :: IO String
getLine = do x ← getChar
            if x == '\n' then
                return ""
            else do xs ← getLine
                    return (x : xs)
```

글쇠판으로부터 값을 읽어들인다

```
readLn :: Read a ⇒ IO a
readLn = do xs ← getLine
            return (read xs)
```

글자를 화면에 나타낸다

```
putChar    :: Char → IO ()
putChar c = -- 내용 생략
```

글줄을 화면에 나타낸다

```
putStr      :: String → IO ()
putStr ""   = return ()
putStr (x : xs) = do putChar x
                      putStr xs
```

글줄을 화면에 나타내고 다음 줄로 넘어간다

```
putStrLn    :: String → IO ()
putStrLn xs = do putStr xs
                  putChar '\n'
```

값을 화면에 나타낸다

```
print :: Show a ⇒ a → IO ()
print = putStrLn ∘ show
```

잘못 글귀를 화면에 나타내고 프로그램을 끝낸다

```
error    :: String → a
error xs = -- 내용 생략
```

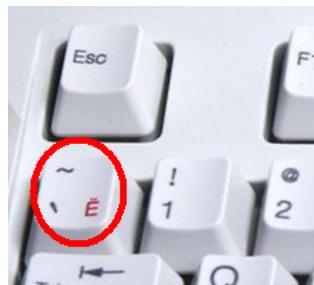
부록 B

기호표 및 Hugs 명령어

이 부록에서는 책에서 쓰인 각 하스켈 특수 기호의 뜻과 그 기호를 보통의 글쇠판으로는 어떻게 치는지 정리해 놓은 표를 보여준다.

기호	뜻	치기
\rightarrow	대응	\rightarrow
\Rightarrow	클래스 제약	$=>$
\geq	크거나 같다	\geq
\leq	작거나 같다	\leq
\neq	같지 않다	$/=$
\wedge	논리곱	$\&\&$
\vee	논리합	$ $
\neg	논리 부정	not
\uparrow	거듭제곱	\wedge
\circ	함수 합성	.
λ	요약	\
$\#$	이어붙이기	$++$
\leftarrow	뽑아내다	$<-$
\gg	순서	$>>=$
$+++$	선택	$+++$

옮긴이 주: 역따옴표^{backquote} 글쇠는 이렇게 생겼습니다.



(이 그림은 위키미디어 공용 Wikimedia Commons에서 가져왔습니다.)

참고로 역따옴표 글쇠가 그림과는 다른 곳에 있는 글쇠판도 있습니다.

Hugs 명령

명령	뜻
:load 파일	스크립트 파일을 불러들인다
:load	표준 서막을 제외한 나머지 스크립트를 모두 잊는다
:also 파일	스크립트 파일을 추가로 불러들인다
:reload	현재 스크립트를 다시 불러들인다
:module 모듈	모듈을 이용할 수 있도록 불러들인다
:edit 파일	스크립트 파일을 편집한다
:edit	현재 스크립트를 편집한다
:type 식	식의 타입을 보여준다
:browse 모듈	모듈에 나온 정의를 나열한다
:info 이템	어떤 이템(함수, 타입, 클래스 등)에 대한 정보 출력
:find 정의	정의를 포함하는 모듈을 편집한다
:names 정규식	접근 가능한 이름 중 정규식과 맞는 것 나열
:names	접근 가능한 모든 이름을 나열한다
:set 옵션	명령줄 옵션을 설정한다
:set	명령줄 옵션 도움말을 보여준다
:main	메인 함수 실행
:main 인자	지정한 인자로 메인 함수 실행
:gc	메모리 재활용을 강제한다
:cd 디렉토리	디렉토리를 바꾼다
:version	Hugs의 판을 보여준다
:?	모든 명령어에 대한 도움말을 보여준다
:quit	Hugs를 끝낸다

부록 C

번역 용례

우리말로 옮길 때 “컴퓨터 프로그램의 구조와 해석”(SICP 우리말 판)의 번역 용례에 있는 낱말들은 대체로 그 번역을 따랐다. 또한 ROPAS 번역 용례¹, “컴퓨터 프로그래밍의 예술”(The Art of Computer Programming 우리말 판), 통계학회용어집, 문화관광부 발행 국어순화용어자료집(1997) 전산기 순화 용어², 동아대학교 조장우 교수의 컴파일러 과목 강의노트³, 김중태문화원 한글컴퓨터낱말표⁴ 등을 참고하였다.

그 중에 아직 번역 용례가 드물다 싶거나 헛갈릴 여지가 있다고 생각되는 것들을 다음 표에 늘어놓았다.

영어	우리말	비고
accumulator	쌓개	
arity	항수	
associates	(괄호로) 묶이는	associates to the left 를 “왼쪽으로부터 (괄호로) 묶이는”과 같이 옮겼다.
base case	밑바탕(이 되는) 경우	
brute-force	짐승같이 무식한	
built-in to ~	~ 자체 내(에서)	
comprehension	조건제시식	‘내포’ ⁵ 라는 논리학 용어도 있으나, 의무교육 수학 교과과정에서 배우는 ‘조건제시법’(comprehension notation)과 일관성 있도록 골랐다.
default	없을때, 없을 때를 위한	예컨대, default definition 을 ‘없을때 정의’로 옮겼다.
derive	이끌어내다	
dual	맞짝	맞서는 짝이라는 뜻이다.
error	실수, 잘못	error 함수로 프로그램 자체에서 프로그램이 잘못되었다고 하는 것은 잘못으로, 문법이나 타입 error처럼 실수로 잘못된 프로그램을 작성해서 하스켈 구현에서 받아들이지 않는 경우는 실수로 번역하였다. SICP 에도 두 가지 다 용례로 나와 있는데 아마 이와 마찬가지로 했을 것이라 추측한다. parity error 는 문화관광부 전산기 용어 순화안에 따라 ‘홀짝 틀림’으로 옮겼다.
evaluation strategy	계산 방식	ROPAS 번역 용례
formal	(엄밀한) 형식적	
guard	보초	이것으로 표준화되었으면 한다.
guarded	보초선	이것으로 표준화되었으면 한다.
factorial	사다리곱	SICP 우리말 판
higher-order	함수를 주고받는	SICP 우리말 판에는 ‘차수 높은’으로 옮겼으나 ROPAS 번역 용례가 더 알아듣기 쉬운 것 같아 이를 골랐다. 하지만 단한 차례, higher-order functional language 의 경우는 ‘차수 높은’으로 옮겼다.
instance	인스턴스	
invalid	그릇되다	
message	글귀	
modular	모듈 방식	SICP 우리말 판에서 골랐으며, 덧붙여 짜 맞추듯이 한다는 뜻으로 ‘맞춤식(조립식)’도 소개해 놓았다. ‘따로따로 포장해서’라는 ROPAS 번역 용례도 있다.

¹<http://ropas.snu.ac.kr/lib/term/>

²<http://wiki.kldp.org/KoreanDoc/html/Translation-KLDP/compword.txt>

³<http://web.donga.ac.kr/jwjo/>

⁴<http://www.dal.kr/chair/cm/cm0701.html>

⁵comprehension, connotatoin, intention

mutual recursion	서로 부르며 되돌기	
nested	포개지는	
n-tuple	순서있는 n 짹	통계학회용어집
overloaded	여러의미	여러모양(polymorphic) 과 대구를 이루기 위해 골랐다
partial application	부분 적용식	
predicate	조건, 술어, 성질	'성질'로 옮길 수 있는 property와 동의어로 쓰이고 있으므로, 처음으로 predicate이라는 단어가 소개되는 곳을 제외한 그 이후로는 '조건'으로 옮겼다.
processing	조작	
prompt	길잡이	네이버 영어사전
prototype	시제품	
primitive (8 장)	간단한	basic 을 이미 '기본'으로 옮겼으며, 기본적인(basic) 것을 이용해서 간단한(primitive) 것을 끌어내는(derive) 상황이기 때문에 '기초', '원시' 등으로 옮길 수 없으므로 '간단'을 골랐다. 8 장 이외의 다른 곳에서는 SICP 와 마찬가지로 '기본(적인)'과 같이 옮겼다.
recursive case	되도는 경우	
redex	줄 식	
script	스크립트	
section	잘린식	수학용어로는 자름면, 단면 등이 있으나 적절한 번역인지 확실치 않아 나름대로 옮겼다.
sequencing	순서대로 엮기	
side effect	곁따르는 반응	ROPAS 번역 용례 '함께오는 반응'에서 실마리를 얻었다.
singleton	한원소	통계학회용어집: 한원소집합 singleton set
strict application	깐깐한 (함수) 적용	SICP 우리말 판
survey article	모두풀이 논문	
tautology	늘 참	항진이 더 널리 쓰이나 '늘 참'도 두 글자인데다 알아듣기 쉽기 때문에 이를 골랐다.
termniation	끝남	
tuple	순서쌍, 순서짝	통계학회용어집
underflow	모자람	stack underflow를 스택 모자람으로 옮겼다.

참고문헌

- [1] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.
- [2] Richard Bird and Shin-Cheng Mu. *Countdown: A Case Study in Origami Programming*. University of Oxford, 2005.
- [3] Richard Bird and Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988.
- [4] Koen Claessen and John Hughes. Quick Check: A Lightweight Tool for Random Testing of Haskell Programs. In Proceedings of the Fifth ACM SIGPLAN International Conferenceon Functional Programming, Montreal, Canada, September 2000.
- [5] Nils Anders Danielsson and Patrik Jansson. Chasing Bottoms: A Case Study in Program Verification in the Presense of Partial and Infinite Values. In Proceedings of the 7th International Conference on Mathematics of Program Construction, volume 3125 of Lecture Notes in Computer Science, Stirling, Scotland, Springer, 2004.
- [6] Karl-Filip Faxén. A Static Semantics for Haskell. In Graham Hutton, editor, *Journal of Functional Programming*, Special Double Issue on Haskell, 12(4&5). Cambridge University Press, 2002.
- [7] Jeremy Gibbons and Oege de Moor, eds. *The Fun of Programming*. Palgrave, 2003.
- [8] Jeremy Gibbons and Graham Hutton. Proof Methods for Corecursive Programs. *Fundamenta Informaticae*: Special Issue on Program Transformation, 66(4): 353–366, 2005.
- [9] Andy Gill and Simon Marlow. Happy: A Parser Generator for Haskell. Available on the web from: <http://www.haskell.org/happy>.
- [10] Hugh Glaser, Pieter Hartel, and Paul Garratt. Programming by Numbers: A Programming Method for Novices. *The Computer Journal*, 43(4), 2000.
- [11] Paul Hudak. Conception, Evolution and Application of Functional Programming Languages. *Communications of the ACM*, 21(3): 359–411, 1989.
- [12] Gerard Huet. The Zipper. *Journal of Functional Programming*, 7(5): 549–554, September 1997.
- [13] John Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2): 98–107, 1989.
- [14] Graham Hutton. A Tutorial on the Universality and Expressiveness of Fold. *Journal of Functional Programming*, 9(4): 355–372, 1999.
- [15] Graham Hutton. The Countdown Problem. *Journal of Functional Programming*, 12(6): 609–616, 2002.
- [16] Graham Hutton and Erik Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.
- [17] Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8(4): 437–444, 1998.
- [18] Graham Hutton and Joel Wright. Compiling Exceptions Correctly. In Proceedings of the 7th International Conference on Mathematics of Program Construction, vol.3125 of Lecture Notes in Computer Science, Stirling, Scotland. Springer, 2004.

-
- [19] Graham Hutton and Joel Wright. Calculating an Exceptional Machine. In Hans-Wolfgang Loidl, ed., Trends in Functional Programming, vol.5:Intellect, February 2006. Selected papers from the Fifth Symposium on Trends in Functional Programming, Munich, November 2004.
 - [20] Mark P. Jones. Typing Haskell in Haskell. In Proceedings of the 1999 Haskell Workshop, Paris, France, 1999.
 - [21] John Launchbury. A Natural Semantics for Lazy Evaluation. In Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 144–154, Charleston, South Carolina, January 1993.
 - [22] Daan Leijen. Parsec: A Parsing Library for Haskell. Available on the web from: <http://www.haskell.org/parsec>.
 - [23] Saunders MacLane. Categories for the Working Mathematician. Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971.
 - [24] Simon Peyton Jones. Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-Language Calls in Haskell. In Tony Hoare, Manfred Broy, and Ralf Steinbruggen, eds., Engineering Theories of Software Construction, pages 47–96. IOS Press, 2001. Presented at the 2000 Marktoberdorf Summer School.
 - [25] Simon Peyton Jones. Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, 2003. Also available on the web from <http://www.haskell.org/definition>.
 - [26] Simon Peyton Jones and David Lester. Implementing Functional Languages: A Tutorial. Prentice Hall, 1992.
 - [27] V. J. Rayward-Smith. A First Course in Formal Language Theory, vol.234 of Pitman Research Notes in Math. Blackwell Scientific Publications, 1983.
 - [28] John C. Reynolds. Theories of Programming Languages. Cambridge University Press, 1998.
 - [29] Simon Singh. The Code Book: The Secret History of Codes and Code Breaking. Fourth Estate, 1999.
 - [30] Philip Wadler. The Concatenate Vanishes. University of Glasgow, 1989.
 - [31] Philip Wadler. Monads for Functional Programming. In Manfred Broy, ed., Proceedings of the Marktoberdorf Summer School on Program Design Calculi. Springer-Verlag, 1992.

찾아보기

!!¹, 51, 349
(), 74, 193, 208, 347
*, 49, 342
+, 49, 92, 342
-, 49, 92, 342
/, 95, 342
::, 105, 348
:::, 65
<, 86, 342
==, 84, 341, 356
>, 86, 342
[], 72, 185, 275, 348
\$!, 308, 354
abs, 92, 100, 342
all, 162, 349
and, 112, 163, 348
any, 162, 349
Bool, 65, 68, 343
case, 185
Char, 69, 344
chr, 124, 345
class, 264
concat, 116, 353
const, 109, 354
curry, 354
data, 244
deriving, 265
digitToInt, 345
div, 49, 53, 94, 174, 342
drop, 51, 143, 149, 350
dropWhile, 162, 350
elem, 348
Eq, 83, 341
error, 204, 355
even, 99, 145, 346
False, 65, 68
filter, 161, 349
Float, 71, 126, 346
foldl, 149, 167, 311, 351
foldl1, 351
foldr, 149, 163, 178, 351
foldr1, 351
FP, 35
Fractional, 95, 342
fromIntegral, 126
fst, 105, 347
getCh, 215
getChar, 213, 214, 354
getLine, 216, 355
GHC, 42, 280
head, 51, 106, 349
Hugs, 42, 54
명령어, 61, 357
id, 158, 353
if, 67, 100
init, 152, 350
instance, 264
Int, 70, 346
Integer, 70, 346
Integral, 94, 151, 342
intToDigit, 345
IO, 212, 354
isAlphaNum, 189
isAlpha, 189, 344
isAlphaNum, 345
isDigit, 98, 189, 345
isLower, 189, 344
isSpace, 193, 345
isUpper, 189, 344
ISWIM, 35
iterate, 172, 352
length, 52, 80, 117, 139, 164, 168, 352
map, 159, 353
max, 86, 342
maximum, 352
Maybe, 246, 348
min, 86, 342
minimum, 352
Miranda, 35
ML, 35
mod, 94, 117, 125, 174, 307, 342
Monad, 268, 343
negate, 92, 342

null, 106, 153, 348
 Num, 37, 82, 92, 342
 odd, 145, 170, 346
 or, 163, 348
 Ord, 39, 86, 121, 141, 342
 ord, 124, 345
 otherwise, 101, 344
 print, 355
 product, 59, 147, 163, 352
 putChar, 213, 216, 355
 putStr, 216, 355
 putStrLn, 216, 355
 Read, 90, 342
 read, 90, 342
 readLn, 355
 recip, 95, 99, 342
 repeat, 174, 351
 replicate, 305, 351
 return, 184, 213, 268, 343
 reverse, 52, 139, 165, 168, 317, 326, 353
 Show, 88, 342
 show, 88, 342
 signum, 92, 101, 342
 snd, 105, 347
 span, 351
 splitAt, 99, 350
 String, 70, 344
 sum, 52, 163, 167, 168, 352
 tail, 51, 106, 349, 350
 take, 51, 305, 349
 takeWhile, 162, 350
 toLower, 346
 toUpper, 346
 True, 65, 68
 type, 242
 uncurry, 354
 unfold, 180
 where, 38
 zip, 142, 352
 값 구하기 evaluation, λ 계산(법)
 같다, 같기, $=$
 같지 않다, \neq
 거듭제곱 exponentiation, \uparrow
 결따르는 반응 side effect, 34, 211
 계산(법) evaluation, 49, 66, 204, 275, 293
 느긋한 lazy, 34, 73, 103, 118, 172, 302
 바깥쪽부터 outermost, 296
 안쪽부터 outermost, 295
 인자 먼저 call-by-value, 298
 최상위 top-level, 308
 함수 먼저 call-by-name, 298
 곱셈 multiplication, λ *, product
 공백, λ 빈칸
 공유 sharing, 301
 교환법칙 commutativity, 282
 곱셈, 282, 315
 덧셈, 282, 321
 귀납(법) induction, 35, 314
 가정 hypothesis, 319
 나무 tree, 329
 리스트 list, 323
 수 number, 320, 322
 식 expression, 333
 클래스 고우 히스켈 컴파일러, λ GHC
 글자, λ Char
 글줄 string, λ String
 기호표 symbol table, 356
 끝남 termination, 30, 34, 299, 304
 나눗셈 division, λ div, /
 나머지 remainder, λ mod
 논리 logical
 논리값 value, λ Bool
 논리곱 conjunction, λ \wedge , and
 논리합 disjunction, λ \vee , or
 를 참 tautology, 251
 명제 proposition, 251
 부정 negation, λ \neg
 연산자 operator, λ \wedge , \vee , \neg
 (바로) 다음 수 successor, 111, 247, 299
 덧셈 addition, λ +, sum
 도미노 효과 domino effect, 320
 되돌기 recursion, 33, 37, 135
 서로 부르며 mutual, 145
 여러 갈래로 multiple, 144
 두 인자 사이에 infix, λ 역따옴표, 59
 두갈래나무 binary tree, 249
 들여쓰기 규칙 layout rule, 62, 187
 등식 바탕의 논증 equational reasoning, 34, 314
 등호, λ ==
 람다 션법 lambda calculus, 35, 108
 리스트 list, 32, 36, 51, 72, 114
 길이 length, λ length, 72
 무한 infinite, 34, 73, 172, 303
 빈 empty, λ []
 원소 element, 72, 105
 정렬 sort, 38, 121, 141, 144, 156
 한원소 singleton, 72, 183
 리스P Lisp, 35
 마무리 오차 rounding error, 71
 모나드 monad, λ Monad
 무한 infinity, 299, 319
 문법 분석 나무 parse tree, 201
 문법 분석기 parser, 181
 미란다 Miranda, 35
 n 번째 원소 고르기, λ !!
 보기

- 계산기 calculator, 219
 글줄 전송기 string transmitter, 171
 늘 참 검사기 tautology checker, 251
 부분수열 subsequence, 275
 빨리 뒤집기 fast reverse, 326
 사다리곱 factorial, 63, 136
 산술식 문법분석기, 200
 삽입정렬 insertion sort, 141
 생명 게임 game of life, 230
 소수 prime number, 118, 306
 순열 permutation, 275
 에라토스테네스의 체, 307
 진법 변환 base conversion, 172
 추상 기계 abstract machine, 260
 카운트다운 문제, 272
 카이사르 암호 Caesar cipher, 123
 카이제곱 chi-square, 127
 쾌정렬 quicksort, 39, 144
 피보나치 수열, 144, 312
 보초 guards, 34, 101, 117, 153
 부호 sign, signum
 부호 반전 negation, negate
 분배법칙 distributivity, 314, 324, 334
 빙칸, 194
 뺄셈 subtraction, $-$
 짹 소리를 내다, 217
- 사전식 순서 lexicographic order, 87, 266
 생성원 generator, \leftarrow , 114
 성질 predicate, 조건
 세계의 상태 state of the world, 211
 세우스 박사 Dr. Seuss, 183
 수 number, 342
 떠돌이 소수점, Float
 소수 prime, 118, 306
 십진수 decimal number, 171
 완전수 perfect, 133
 이진수 binary, 171
 자연수 natural, 195, 201, 319
 정수, $\text{Int}, \text{Integer}$
 순서대로 엮기 sequencing, \gg
 순서쌍(순서있는 n 짝) tuple, 74, 347
 빈, \emptyset
 성분 component, 74, 105
 순서쌍(순서있는 2 짝) pair, 74, 115, 120, 347
 순서있는 3 짝 triple, 74, 347
 형수 arity, 74
- 술어 predicate, 조건
 스칼라곱 scalar product, 134
 스크립트 script, 54
 스택 stack, 331
 (흐름) 제어 스택 control stack, 261
 스택 모자람 stack underflow, 335
 식 expression, 29
 논리식 logical, 251
 람다 lambda, λ
 불순한 impure, 213
 산술식 arithmetic, 200
 순수한 pure, 213
- 조건식 conditional, if
 줄일 수 있는 reducible, 251, 295
 식별자 identifier(변수 이름), 61, 192
- (바로) 앞 수 predecessor, 107, 136, 145
 엠엘 ML, 35
 여러모양 polymorphic, 80
 여러의미 overloading, 82
 역따옴표 backquote, 357
 연수 reciprocation, recip
 연산자 operator, 49, 111, 163, 167, 274
 예약어 keyword, 33, 61
 예외 exception 처리, 240
 우선순위 priority, 50, 54, 200
 유니코드 Unicode, 124, 174, 345
 이름짓기 규칙 naming requirements, 61, 243, 244
 이어붙이기
 여러 리스트, concat
 연산자, +
 입출력 input/output, IO
- 잘린식 section, 111
 잘못 글귀, error , 53
 전문 영역 언어 domain specific language, 34, 159
 절대값 absolute value, abs
 정렬 sort
 삽입정렬 insertion sort, 141
 퀵정렬 quicksort, 39, 144
 합병정렬 merge sort, 156
 제어 글자 control character, 69, 217
 조건 (함수) predicate, 160, 189
 조건 condition, 100
 조건식이 동강나다 dangling else, 101
 조건제시법 comprehension
 집합 set, 114, 132
 조건제시식 comprehension
 글줄 string, 125
 리스트 list, 33, 114
 주석 comment, 62
 줄 식 redex, 295
 줄임 reduction, 295
 람다식 안에서 under lambda, 298
 지퍼 zipper, 269
 진리표 truth table, 251
 참조표 lookup table, 119, 243
 카테고리 이론 category theory, 269
 커서 cursor, 241
 커서 읽기기, 218
 콘스 cons, :
 클래스 class, 83, 264, 341
 메서드 method, 83
 없을때 정의 default definition, 264
 유도된 derived 인스턴스, 265
 인스턴스 instance, 82, 264
 제약 constraint, 82, 266
- 타입 type, 33, 37, 66

- 변수 variable, 81
 생성자 constructor, 244
 선언 declaration, 242, 244
 실수 error, 66
 안전 safe, 66
 유추 inference, 33, 37, 66
 인자 parameter, 243, 246
- ~ 타입 type
 같기 equality, Eq
 기본 basic, 68
 나무 tree, 249, 313, 329
 되도는 recursive, 247
 리스트 list, 72, 248, 348
 모나드 monadic, Monad
 보이는 showable, Show
 분수같은 타입 fractional, Fractional
 수치 numeric, Num
 순서 ordered, Ord
 순서쌍 tuple, 74
 아마도 maybe, Maybe
 여러모양 polymorphic, 80
 여러의미 overloaded, 82
 읽히는 readable, Read
 정수같은 integral, Integral
 타입을 인자로 받는 parameterised, 243, 246, 268
 함수 function, 75
 토큰 token, 194
- 파일 file 처리, 240
 패턴 pattern
 $n + k$, 107, 136, 143
 리스트 list, 105, 138
 서로 겹치지 않는 disjoint, 317
 순서쌍 tuple, 105
 아무거나 wildcard, -
 정수 integer, n + k
 패턴 매칭 pattern matching, 34
 표준 서막 standard prelude, 49, 341
 프로그래밍 programming
 대화식 interactive, 42, 211
 명령형 imperative, 31, 294
 모듈 방식 modular, 34, 304
 배치 batch, 210
 함수형 functional, 30
- 함수 function, 28, 54
 깐깐한 strict, 297, 308
 되도는 recursive, 34, 135
 모두 정의된 total, 76
 상수 constant, const
 생성자 constructor, 246
 여러모양 polymorphic, 33, 80, 122, 160
 여러의미 overloaded, 33, 82
 이름없는 nameless, λ
 일부만 정의된 partial, 76
 조합론적 combinatorial, 275
 커리된 curried, 77, 108, 157, 297, 309
 함수를 주고받는 higher-order, 34, 157
 합성 composite, ○
- 항등 identity, id
 합성 composition, ○
 항등원 identity, 282
 (함수) 합성, 170
 곱셈, 137, 138, 282
 나눗셈, 282
 덧셈, 37
 이어붙이기 (+), 327
 화면을 말끔히 지우다, 223