프로그래밍의 수련(修練)
(A Discipline of Programming)

엣져 위베 다익스트라
(Edsger Wybe Dijkstra)

네번째: 프로그래밍 언어의 의미 규정

해역(解譯)
김도형
성신여자대학교 컴퓨터정보학부
dkim@cs.sungshin.ac.kr
URL: http://cs.sungshin.ac.kr/∼dkim

4장. 프로그래밍 언어의 의미 규정(意味規定) — I

앞 장(章)에서 우리는 한 메커니즘 S의 '술어 변환자(述語變換子)', 즉 임의의 사후 조건(事後條件) R에 대해서 활성화(活性化)시키면 R을 만족하는 최종 상태에 메커니즘을 도달시키고 적절히 정지하는 동작을 일으키는 초기 상태를 뜻하며 'wp(S, R)'로 표시되는 최약 사전 조건(最弱事前條件)을 유도하는 법을 말해주는 규칙을 안다면, 그 메커니즘 S의 의미를 충분히 잘 알고 있는 셈이라는 입장을 취했다. 문제는 다음과 같은 것이다: 주어진 S와 R로부터 어떻게 wp(S, R)을 유도하는가?

일단은 하나의 특정한 메커니즘 S에 대한 이야기는 그저 그 정도로 한다. 명확하게 정의(定義)된(well-defined) 어떤 프로그래밍 언어로 쓰여진 프로그램은 하나의 메커니즘이고, 이것은 만약 그것의 상응(相應)하는 술어 변환자를 안다면 곧 그 자체를 충분히 잘 안다고 할 수 있는 그러한 메커니즘이다. 한편 프로그래밍 언어란 그것을 사용하여 많은 상이(相異)한 프로그램을 표현할 수 있을 때만 유용한 것이며, 우리는 그러한 모든 프로그램에 대해 해당하는 술어 변환자를 틀림없이 알고 싶어할 것이다.

그러한 프로그램은 어떤 것이든지 잘 정의된 프로그래밍 언어로 작성된 텍스트(text)에[1] 의해 정의되며, 그 텍스트가 따라서 우리의 출발점이 되어야 할 것이다.[2] 그러자 불현듯 그러한 프로그램 텍스트의 두 가지 완전히 상이한 역할을 깨닫게 된다! 우리가 프로그램이 자동적으로 수행되기를 바라거나 특정한 계산이 실행되기를 바랄 때면, 그 프로그램 텍스트는 기계에 의해 해석되어야만 한다. 다른 한편으로 프로그램 텍스트는 상응하는 술어 변환자를 어떻게 구성할 수 있는가를, 혹은 우리의 관심을 끄는 임의의 사후 조건 R에 대한 $\mathrm{wp}(S, R)$을 유도하는 술어 변환을 어떻게 하면 달성할 수 있는가를 우리에게[3] 말해 주어야만 한다. 이러한 관찰로부터, 우리와[4] 관련하여 '명확하게 정의된

[1] 'text'에 대한 좋은 번역 용어를 발견할 수 없어서 그냥 이렇게 했다. 프로그램에서 순수하게 알고리즘을 서술하는 부분을 지칭하는 것으로 보인다.

[2] 첫번째 단락에서 언급되었지만 우리가 궁극적으로 알고자 하는 것은 제시된 메커니즘과 사후 조건에 대한 술어 변환자를 유도하는 방법이다. 그리고 이 목표를 위해서 이 장에서 우리가 하고자 하는 것은, 제목에도 나와 있지만 결국 우리가 다룰 메커니즘은 프로그램이고 그것은 프로그래밍 언어로 작성되므로, 프로그램이라는 메커니즘의 술어 변환자를 알려면(곧 그 의미를 알려면) 그것을 형성하는 도구인 프로그래밍 언어의 의미를 알아야 하며 따라서 프로그래밍 언어의 요소들을 하나씩 분석하여 그 의미를 규정하는(곧 해당하는 술어 변환자를 알아내는) 것이다. 이 문장은 프로그램의 의미를 파악하기 위한 출발점은 그것의 텍스트를 구성하고 있는 요소들의 분석이 될 것이라는 뜻이다.

[3] 앞의 문장에 나오는 '기계'와 이 문장의 '우리' 즉 '인간'을 대비하고 있다. 앞에서도 나왔지만 '기계'—'공학적 고려 사항'—'효율(效率)'이 한 축(軸)이고, '인간'—'과학적(또는 수학적) 고려 사항'—'정합성(整合性)'이 또 다른 한 축을 이루고 있다.

[4] 즉 '인간'과.

프로그래밍 언어'라고 말할 때 우리가 의미하는 바가 무엇인지를 알 수 있다. 하나의 특정한 메커니즘(프로그램)의 의미는 그것의 술어 변환자에 의해 주어지는데 비해서, 어떤 프로그래밍 언어의 의미 규정은 그 언어로 작성된 각 프로그램과 해당 술어 변환자를 연계(連繫)시킬 수 있는 규칙들의 집합으로 주어진다. 이러한 관점에서 우리는 프로그램을 술어 변환자를 위한 '하나의 코드'라고 간주할 수 있다.

만약 원한다면, 프로그래밍 언어를 설계한다는 문제를 여기에서부터[5] 접근할 수도 있다. 그러한 접근 방법에서—상당히 형식화(形式化)된—출발점은 술어 변환자를 구성하는 규칙들은 그것들을 적용하여 구성되는 어떤 것이든지 앞 장 '의미의 규정'에 나와 있는 성질 1부터 4까지를 만족하는 술어 변환자여야 한다는 것이다. 왜냐하면 만약 그렇지 못할 경우, 우리는 술어들을 더 이상 사후 조건이나 대응하는 최약 사전 조건으로 해석될 수 없는 식으로 다루고 있을 뿐이기 때문이다.

그 요구되는 성질들을[6] 만족하는 두 개의 매우 간단한 술어 변환자가 즉각적으로 떠오른다.

우선 항등 변환(恒等變換; identity transformation), 즉 임의의 사후 조건 R에 대해 $\mathrm{wp}(S, R) = R$인 메커니즘 S가 있다.[7] 이 메커니즘은 모든 프로그래머들에게 알려져 있고 또 애용(愛用)되고 있는 것이다: 프로그래머들은 그것을 '공백문(empty statement)'으로 알고 있고, 프로그램 텍스트 내에서 구문적으로 보아 문장(statement)이 필요한 곳에 아무 것도 쓰지 않음으로써 그것

[5]원문에는 "$\cdots$ one can approach the problem of programming language design from out of that corner."로 되어 있다. 이 문장에서 '여기(that corner)'란 앞 단락에서 서술한, 프로그래밍 언어는 그 언어로 쓰여진 프로그램에 대해 술어 변환자를 대응시킬 수 있는 규칙에 의해 의미가 규정된다는 인식을 지칭한다.

[6]성질 1부터 4까지를 말한다.

[7]여기까지의 이야기를 잠깐 정리해 보도록 하자. 어떤 구체적 메커니즘 하나에 대한 의미는 그것의 술어 변환자를 알아냄으로써 파악한다. 프로그래밍 언어의 경우는 상황이 다소 다르다. 모름지기 '유용한(의미 있는?)' 프로그래밍 언어라면 그것을 사용하여 기술(記述)할 수 있는 프로그램의 개수가 무한(無限)이어야 한다. 따라서 하나의 프로그래밍 언어로 만들어질 수 있는 무한개의 메커니즘 각각에 대해 상응하는 술어 변환자를 우리가 알아낸다면, 우리는 그 언어의 의미를 파악했다고 말할 수 있을 것이다. 그렇지만 무한개의 메커니즘을 각각 '독립적인' 개체로 취급하여 그 술어 변환자를 알아내려고 접근하는 방식은 역시 그 메커니즘의 개수만큼 '무한한' 노력을 요구하게 된다. 결국 그 무한한 개수의 프로그램들이 '유한(有限)한' 개수의 언어 구성 요소로부터 만들어지게 된다는 점을 이용하는 것이 바른 방향일 것이다. (주지(周知)하다시피 언어 구성 요소의 개수가 무한한 언어는 학습이 불가능하고 따라서 현실적인 의미가 없다.) 즉, 프로그램 텍스트가 그 프로그램에 대응하는 술어 변환자를 구하기 위한 '코드'가 되는 것이다.

프로그램을 구성하는 언어 구성 요소의 기본적 단위부터 그 의미(곧 그것의 술어 변환자)를 알아내고, 그 기본적 단위들이 결합되어 보다 큰 단위, 궁극적으로는 복잡한 프로그램을 만들어내는 규칙과 그러한 과정에 대응하는 술어 변환자의 유도 규칙을 알아내는 것이 프로그래밍 언어의 의미를 파악하는 길이 될 것이다.

그러면 술어 변환자가 무엇인지 다시 생각해 보자. 결국 그것은 하나의 함수(函數)이다. 주어진 메커니즘에 대해서, 임의의 사후 조건을 인자(因子)로 받아 대응하는 최약 사전 조건을 돌려주는 함수인 것이다. 일반적으로 말해서, 언어 구성 요소들 중 가장 기본적인 단위에 대응하는 술어 변환자 역시 가장 기본적인 함수의 형태를 취할 것이다. 가장 기본적인 함수에는 어떤 것이 있는가? 항등 함수(恒等函數; identity function)나 상수 함수(常數函數; constant function) 등이 될 것이다. 지금 막 항등 함수에 해당하는 술어 변환자와 그것에 상응하는 메커니즘을 이야기하려고 하고 있다.

을 나타낸다. 이것은 별로 좋은 관습이 아니므로(컴파일러는 있어야 할 문장을 보지 못함으로써 그것[8]을 '알게'될 따름이다)[9] 우리는 그것에 예컨대 '*skip*' 같은 이름을 부여하겠다. '*skip*'이라고 명명(命名)된 문장의 의미는 따라서 다음과 같이 주어진다:

$$\text{임의의 사후 조건 } R\text{에 대해서, } \mathrm{wp}(skip, R) = R$$

(모든 사람들이 그러하듯이, 나는 '문장'이라는 용어를 사용할 것이다. 왜냐하면 그것은 전문 용어로서 자리를 굳혔기 때문이다. 사람들이 '명령(command)'이 아마도 더 적합한 용어일 것이라고 제안했을 때는, 이미 너무 늦었었다!)[10]

> 주의. '아무것도 쓰지 않는 것(nothing)'이 공백문의 의미를 잘 표현하고 있는데, 그것을 위해 '*skip*'이라는 별도의 이름을 도입하는 것은 문자의 낭비라고 생각하는 사람들은 영(zero)이라는 개념을 위해 '0'이라는 문자를 도입한 덕분으로 십진수 체계 자체가 가능해졌음을 깨달아야 한다.

앞으로 나아가기 전에 다음과 같은 점을 지적할 기회를 놓치고 싶지 않다: 이럭저럭 하는 동안에 우리는 하나의 프로그래밍 언어를 (이미) 설계한 것이다! 분명히 그 언어는 다소 초보적인 것이다: 이 언어는 하나의 문장만 가진 언어가 되는데, 오직 하나의 메커니즘만 정의할 수 있고 그 메커니즘이 우리를 위해 해줄 수 있는 유일한 일이라고는 '상태를 있는 그대로 남겨두는 것(leaving things as they are)'(혹은 '아무 일도 하지 않는 것(doing nothing)'. 그런데 부정(negation) 표현때문에 이것은 언어를 안전하지 못하게 사용하는 것이다; 다음 단락을 보라)이다.[11]

그 다음 간단한 술어 변환자로는 사후 조건 R에 전혀 의존하지 않는, 상수(常數)의 최약 사전 조건을 유도하는 것이 있다. 우리는 상수 술어로 T와 F 두 가지를 가지고 있다. 모든 R에 대해 $\mathrm{wp}(S, R) = T$인 메커니즘 S는 기적 배제의 법칙을 위배할 것이므로 존재할 수 없다;[12] 그러나 모든 R에 대해 $\mathrm{wp}(S, R) = F$인 메커니즘 S는 모든 필요한 성질을 만족하는 술어 변환자를 가진다. 우리는 또한 이 문장에도 예컨대 '*abort*'같은 이름을 부여할 것이다. '*abort*'라고 명명된 문장의 의미는 따라서 다음과 같이 주어진다:

---

[8] 즉 공백문을 말한다.

[9] 원문은 "a compiler only 'sees' it by *not* seeing a statement that should be there"로 되어 있다. 영어의 'see'가 '보다'라는 뜻과 '알다'라는 뜻을 다 가지고 있음을 이용하여 서로 다르게 번역하였다. 그렇지 않으면 번역 문장이 너무 부자연스러워지기 때문이다.

[10] '문장(文章)'이란 상황이 어떠하다고 표현하는 진술(陳述)이다. 프로그래밍 언어에서의 문장이란 사실 그 내용으로 볼 때, 이러한 일을 하라고 지시하는 '명령(命令)'에 보다 가깝다.

[11] 다음 단락에서 보충 설명을 하겠지만, 우리말에서는 '아무 일도 하지 않는 것'이 '상태를 있는 그대로 남겨두는 것'과 동일한 뜻이지만, 그 영어 표현인 'do nothing'은 혼란의 여지가 있다.

[12] '모든 R'이라고 했으므로, R 자리에 F를 대입시키면 기적 배제의 법칙에 위배됨을 보일 수 있다.

임의의 사후 조건 R에 대해서, $\mathrm{wp}(abort, R) = F$

이 술어 변환자는 '상태를 있는 그대로 남겨둔다'는 뜻에서의 '아무 것도 하지 않는 것'조차 하지 못 한다; 그것은 실제로 어떤 일도 하지 못 한다.[13] 만약 우리가 $R = T$로 취한다고 하면, 즉 최종 상태에 대해 존재하는 것 이상의 아무런 추가적인 요구 사항을 부과하지 않는다고 할지라도, (그 최종 상태에) 대응하는 초기 상태가 존재하지 않는다. 따라서 '*abort*'라는 이름의 메커니즘은 호출되었을 때, 어떤 최종 상태에도 도달하지 못 한다: 그것을 활성화시키려는 시도는 실패(失敗)의 징후(徵候)로 해석되는 것이다. (나중에 특별 경우로서 '*skip*' 및 '*abort*'와 의미상 동일한 것을 포괄(包括)하는 문장의 틀을 제시할텐데, 지금으로서는(그리고 나중에도!) 우리에게 중요하지 않다.[14])

이제 우리는 (여전히 매우 초보적인!) 두 문장으로 된 프로그래밍 언어를 가지게 되었는데, 이것으로 우리는 아무 것도 하지 않는 것과 항상 실패하는 것 이렇게 두 가지 메커니즘을 정의할 수 있다. 1960년에 그 유명한 '알고리즘적 언어 ALGOL 60에 대한 보고서(Report on the Algorithmic Language ALGOL 60)'[15]가 출판된 이래, 자존심이 강한 컴퓨터 과학자라면 이 단계에 이르러 'BNF'('Backus-Naur Form'의 준말)라고 부르는 표기 기법으로 여태까지 개발한 자신의 언어의 구문(構文)에 대한 형식적 정의를 내리지 않는 이가 없다. 즉:

$$\text{<문장>} ::= \mathit{skip} \mid \mathit{abort}$$

(다음과 같이 읽는다: "'문장'이라고 부르는 구문적 범주(範疇)의 요소(이것이 특이한 괄호 '<'와 '>'가 나타내는 것이다)는 '*skip*' 혹은(이것이 수직 막대(vertical bar) '|'가 나타내는 것이다) '*abort*'로서 정의된다(이것이 '::='이 나타내는 것이다)." 대단하다![16] 그러나 걱정하지 말라; 표기 기법으로서 BNF

[13] 이 부분의 원서 표현을 일단 그대로 옮기겠다: "This one cannot even 'do nothing' in the sense of 'leaving things as they are'; it really cannot do a thing." 영어 표현에서 'do nothing'과 'do not do anything'은 동일한 것으로 취급된다. 그러나 여기에서는 이 두 표현의 의미가 다르다. 전자(前者)는 '아무 일도 하지 않아서 상태를 있는 그대로 남겨둔다'는 뜻이고, 후자(後者)는 '어떤 일도 하지 못 한다'는 뜻이다. 이런 혼란의 우려 때문에 원서에서는 'do nothing' 대신에 'leaving things as they are'를 사용하고, 앞 단락에서 굳이 부연 설명한 것이다.

[14] 여기서 말하는 문장의 틀(framework of statements)은 선택 구조와 반복 구조를 구성하는 데 사용되는 '가드 명령 집합(guarded command set)'을 말한다. 이 장의 뒷부분에서 나올 것이다.

[15] 데이빗 겔런터(David Gelernter)에 따르면(사실 다른 많은 사람들도 동의하는 바이지만), 1963년에 새롭게 개정된 ALGOL 60 보고서(페터 나우어(Peter Naur)가 편집하여 CACM에 발표된 'Revised Report on the Algorithmic Language ALGOL 60'을 말한다)는 여러 가지 이유에서 컴퓨터 과학의 가장 중요한(물론 이에 대해서는 논쟁의 여지가 있다고 스스로 인정하지만) 논문이라고 한다. 적어도 그 논문이 이 분야에서 가장 명쾌한 저술의 전범(典範)을 보이고 있다는 점에는 대다수의 사람들이 동의하는 바일 것이다.

[16] 물론 스스로 비꼬는 말이다. :-) 너무나 간단한 언어이기에, BNF와 같은 형식적 표기 체계의 도움을 받을 필요가 없을 정도이기 때문이다.

의 보다 인상적(印象的)인 적용은 조만간 나타날 것이다!)

확실하게 보다 흥미로운 술어 변환자의 한 부류는 치환(置換), 즉 사후 조건 R을 위한 형식적 표현(formal expression)에 있는 한 변수의 모든 존재를 (동일한) '다른 어떤 것'으로 대치하는 것에 바탕을 두고 있다. 만약 술어 R에서 변수 x의 모든 존재가 어떤 표현(E)으로 대치(代置)된다면, 우리는 이러한 변환의 결과를 $R_{E\to x}$로 표시한다. 이제 주어진 x와 E에 대해서, 모든 사후 조건 R에 대해 $\mathrm{wp}(S,R) = R_{E\to x}$인 메커니즘을 생각할 수 있다. 여기서 x는 우리 상태 공간의 한 '좌표 변수(coordinate variable)'이고 E는 적절한 타입을 가진 표현이다.

> 주의. 치환에 의한 이러한 변환은 앞 장의 성질 1부터 4까지를 만족한다. 우리는 이것을 보이려 하지 않을 것이고, 독자가 이 사실을 뻔한 것으로 간주하든지 아니면 심오(深奧)한 수학적 결과로 간주하든지 그 취향에 맡길 것이다.

위의 패턴은 술어 변환자들의 온전한 부류, 즉 메커니즘의 온전한 부류 하나를 추가한다. 이것은 '배정문(assignment statement)'이라고 부르는 문장에 의해 표시되는데, 이 문장은 세 가지를 명시해야만 한다:

1. 대치할 변수의 확인;
2. 치환이 술어 변환을 위한 대응 규칙이라는 사실;
3. 사후 조건에서 대치되는 변수의 모든 존재를 대치할 표현.

만약 변수 x를 표현 (E)로 대치하려면, 그러한 문장을 쓰는 통상적 방식은 다음과 같다:

$$x := E$$

(여기에서 소위 배정 연산자 ':='는 '~는 ~이 된다' 라고 읽어야 한다). 이 메커니즘은 다음과 같이

$$\text{임의의 사후 조건 } R\text{에 대해서, } \mathrm{wp}(\text{“}x := E\text{”}, R) = R_{E\to x}$$

정의하여 요약할 수 있는데, 이것은 우리가 그렇게 원한다면 어떤 좌표 변수 x와 적절한 타입의 표현 E에 대한 배정문의 의미 정의로 볼 수 있다.

우리가 하는대로 BNF의 사용을 즐기면, 형식적 구문을 다음과 같이 확장할 수 있다:

<문장> ::= *skip* | *abort* | <배정문>
<배정문> ::= <변수> := <표현>

여기에서 마지막 줄은 "'배정문'이라고 부르는 구문적 범주의 요소는 '변수'라고 부르는 구문적 범주의 요소 뒤에 배정 연산자 ':='가 나오고 그 뒤를 '표현'이라고 부르는 구문적 범주의 요소가 따르는 것으로 정의된다"고 읽어야 한다.

앞으로 나아가기 전에, 배정문의 의미에 대한 우리의 형식적 정의가 배정문에 대한 우리의 직관적(直觀的) 이해—만약 우리가 그런 것을 가지고 있다면!—를 정말로 포착하고 있음을 확인하는 것이 현명할 듯 싶다. 두 개의 정수 좌표 변수 'a'와 'b'를 가진 상태 공간을 생각해 보자. 그러면

$$\mathrm{wp}(\text{“}a := 7\text{”}, a = 7) = \{7 = 7\}$$

이고, 우변에 있는 부울(boolean) 표현은 a와 b의 모든 값, 즉 상태 공간에 있는 모든 점들에 대해서 참이므로, 우리는 다음

$$\mathrm{wp}(\text{“}a := 7\text{”}, a = 7) = T$$

과 같이 단순화시킬 수 있다. 즉, 모든 초기 상태는 배정문 '$a := 7$'이 '$a = 7$'의 참을 확립하리라는 점을 보장할 것이다. 유사하게,

$$\mathrm{wp}(\text{“}a := 7\text{”}, a = 6) = \{7 = 6\}$$

이고, 그 부울 표현은 a와 b의 모든 값에 대해서 거짓이므로, 우리는 다음

$$\mathrm{wp}(\text{“}a := 7\text{”}, a = 6) = F$$

이라고 알게 된다. 이것은 배정문 '$a := 7$'이 '$a = 6$'의 참을 확립한다는 점을 우리가 보장할 수 있는 초기 상태가 존재하지 않는다는 것을 의미한다. (이것은 모든 초기 상태가 '$a = 7$'의 참을, 따라서 '$a \neq 7$'의 종국적(終局的)인 거짓을 확립할 것이라는 우리의 앞의 결과와 일치한다.) 또한 다음

$$\mathrm{wp}(\text{“}a := 7\text{”}, b = b0) = \{b = b0\}$$

이 성립하는데, 즉 만약 우리가 배정 '$a := 7$' 다음에 변수 b가 어떤 값 $b0$을 가지는 것을 보장하고자 한다면, b는 이미 초기 상태에서 그 값을 가지고 있어야 한다는 것이다. 달리 말하면, 'a' 이외의 어떤 변수도 건드리지 않으므로 그

변수들은 가지고 있던 값을 유지하게 된다; 배정 '$a := 7$'은 상태 공간에서 현재 시스템 상태에 대응하는 점을 '$a = 7$'이 궁극적으로 성립하도록 a-축에 평행(平行)하게 이동시킨다.

우리는 표현 E를 위해 상수를 선택하는 대신, 초기 상태의 함수를 가질 수도 있었다. 이 점은 다음 예들이 보여준다:

$$\begin{aligned}
&\mathrm{wp}(\text{“}a := 2 * b + 1\text{”},\ a = 13) = \{2 * b + 1 = 13\} = \{b = 6\} \\
&\mathrm{wp}(\text{“}a := a + 1\text{”},\ a > 10) = \{a + 1 > 10\} = \{a > 9\} \\
&\mathrm{wp}(\text{“}a := a - b\text{”},\ a > b) = \{a - b > b\} = \{a > 2 * b\}
\end{aligned}$$

만약 E가 초기 상태의 부분 함수(partial function), 즉 그 정의 구역 바깥에 있는 초기 상태를 가지고 그 함수의 평가를 시도하면 적절히 종료하는 동작에 이르지 못할 함수가 되는 것을 허용한다면, 약간 복잡해지는 부분이 있다; 만약 우리가 그 상황에도 마찬가지로 대응하기를 원한다면, 배정 연산자의 의미 정의를 보다 강화하여 다음

$$\mathrm{wp}(\text{“}x := E\text{”}, R) = \{D(E)\ \mathbf{cand}\ R_{E \to x}\}$$

과 같이 써야만 한다. 여기서 술어 $D(E)$는 'E의 정의 구역 내에 있는'을 의미한다; 부울 표현 '*B1* **cand** *B2*'(소위 '조건적 공접(conditional conjunction)')는 두 개의 피연산자가 다 정의되었을 때는 '*B1* **and** *B2*'와 동일한 값을 가지는데, *B2*가 정의되느냐는 점에 관계없이 *B1*이 '거짓'일 때는 '거짓'이라는 값을 가지도록 또한 정의된다.[17] 조건 $D(E)$는 T이거나 혹은 우리가 E의 정의 구역 바깥의 초기 상태에서는 배정문이 절대 활성화되지 않도록 조치하기 때문에, 대개 명시적으로 언급되지는 않는다.

많은 프로그래머들이 좋아하는 배정문의 자연스런 확장은 소위 '병행 배정(concurrent assignment)'이다. 여기에서는 여러 개의 상이한[18] 변수들을 한꺼번에 치환할 수 있다; 병행 배정문은 배정 연산자의 좌변에 치환될 상이한 변수들(서로 쉼표에 의해 분리된)의 목록과 그 우변에 같은 개수의 표현들(역시 서로 쉼표에 의해 분리된)의 목록으로 표시된다. 따라서 다음

x1, *x2* := *E1*, *E2*
x1, *x2*, *x3* := *E1*, *E2*, *E3*

과 같이 적는 것이 허용된다. 예를 들어 주어진 *x1*, *x2*, *E1*, *E2*에 대해서,

[17] 익히 알려져 있다시피, 부울 표현의 '단락-회로 평가(short-circuit evaluation)'를 지원하는 C와 같은 언어에서의 공접 연산자('&&')나 Ada의 'and then' 연산자가 여기서의 '**cand**'와 동일한 의미를 지니고 있다.

[18] 같은 변수에 대한 병행 배정은 결국 마지막에 실행되는(순차적 구현의 경우) 혹은 비결정적으로 마지막 순서가 된 배정만 효과를 내게 되기 때문에, '병행' 배정의 의미가 없다.

$$x1, x2 := E1, E2$$

가 의미상으로

$$x2, x1 := E2, E1$$

과 동등한 것과 같이, 좌변의 목록에서 i-번째 변수가 우변의 목록의 i-번째 표현으로 대치된다는 것을 유의하라. 병행 배정은 우리가

$$x, y := y, x$$

에 의해서 두 변수 x와 y의 값을 바꾸라고 지시할 수 있게 해 주는데, 이 연산은 병행 배정이 아니었다면 묘사하기 커찮아지는 것이다. 앞의 연산이 용이하게 구현되며, 일부 지나친 명세(overspecification)를 우리가 피할 수 있도록 해준다는 이 사실이 그것이 인기 있는 이유이다.[19] 만약 그 목록이 길어지면,

[19] 약간의 사족을 붙이면 다음과 같다. 두 변수 x와 y의 값을 바꾸고자 하면, 병행 배정이 지원되지 않는 경우의 일반적인 방법은 주지하디시피 임수 변수(예컨대 x, y와 같은 타입인 t)를 사용하여 다음과 같이 하는 것이다:

$$t := x;\ x := y;\ y := t$$

그러나 변수 t의 사용이 불가피함에도 불구하고, 그것의 사용에 마음이 편치않은 사람들도 많이 있다—메모리 셀(cell)의 특성 때문에 어쩔 수 없이 한 변수의 값을 저장해 두어야 하는 다른 변수가 '임시로' 필요한 것일 뿐이다.

게다가 컴퓨터의 하드웨어 상에서는 두 메모리 셀의 내용을 다른 메모리 셀의 도움 없이 한꺼번에 바꿔치는 것이 가능하기에 더욱 그렇다. 이것은 곧 우리가 두 변수 값의 교환에 대한 과정을 프로그래밍 언어 상에서는 불필요할 정도로 자세하게 서술하고 있다고 표현할 수도 있을 것이다.

물론 프로그래밍 언어 상에서도 두 변수 값의 교환을 임시 변수의 도움을 받지 않고 수행할 수도 있다. 널리 알려진 기법은 두 가지 정도인데, 하나는 덧셈과 뺄셈을 적절히 이용하는 것이고:

$$x := x + y;\ y := x - y;\ x := x - y$$

다른 것은 배타적 이접(exclusive OR; XOR)을 이용하는 것이다:

$$x := x \ \mathbf{xor}\ y;\ y := x \ \mathbf{xor}\ y;\ x := x \ \mathbf{xor}\ y$$

그러나 이러한 방법은 프로그램을 대단히 읽기 어렵게 만들므로 바람직하지 않다.

결과 프로그램은 읽기가 매우 어려워진다.

진짜 BNF 중독자라면 배정문에 대한 두 대안(alternative)[20] 형태를 제공하여 자신의 구문을 확장할 것이다. 즉:

<배정문> ::= <변수> := <표현> | <변수>, <배정문>, <표현>

'배정문'이라는 이름의 구문적 단위를 위한 대안 형태들 중 하나(즉 두번째 것)가 그 구성 요소들 중 하나로 '배정문'이라는 이름의 동일한 구문적 단위, 곧 우리가 (지금) 정의하고 있는 구문적 단위를 다시 포함하고 있기 때문에, 이것은 이른바 '재귀적(再歸的) 정의(recursive definition)'이다! 처음 보면 그러한 순환적(cyclic) 정의는 놀라우나, 자세히 살펴보면 적어도 구문적 관점에서는 그 정의에 아무 문제도 없다는 것을 우리는 확신할 수 있다. 예를 들어, (앞의 정의에 있는) 처음 대안에 따르면

$$x2 := E1$$

은 배정문의 하나의 실례(實例; instance)이므로, 다음 식

$$x1, x2 := E1, E2$$

은 다음 형태

$$x1, \text{<배정문>}, E2$$

의 파싱(parsing)을 허용한다. 따라서 이 형태는 두번째 대안에 따라 역시 배정문이 된다. 그러나 의미적(semantic) 관점에서 보면, 이 형태는 *E2*가 *x2* 대신에 *x1*과 연결되어야 한다고 암시하므로 끔찍한 일이다.

'*skip*'과 '*abort*'만 가진 두 문장으로 이루어진 언어에 비하면, 배정문을 가진 우리의 언어는 훨씬 풍요해졌다: '배정문'이라는 구문적 단위의 상이한 실체들의 숫자에는 더 이상 상한(上限)이 없다. 허나 아직도 우리 목적을 위해서는 명백히 부족하다; 우리는 보다 복잡한 프로그램과 보다 정교한 메커니즘을 구성할 수 있는 능력을 필요로 한다. 잠재적으로 정교한 메커니즘의 구성을 위해서 우리는 다음

<메커니즘> ::= <기본적 메커니즘> | <<메커니즘>들의 적절한 합성>

[20] 형식 언어 이론에서, 문법 규칙의 동일한 좌변에 대한 여러 개의 우변을 각각 '대안'이라고 부르는 것은 잘 알려진 사실이다.

에 의해 재귀적으로 묘사될 수 있는 양식(樣式; pattern)을 따른다. 이 양식이 어떤 쓸모가 있으려면, 두 개의 조건이 충족되어야만 한다: 일단 시작할 '기본적 메커니즘'을 가져야만 하고, 둘째로 '적절히 합성(合成)하는' 방법을 알아야만 한다. 여태까지 도입된 문장들은 기본적 메커니즘으로 취할 수 있으며, 이 장의 나머지 부분은 주어진 것들로부터 새로운 메커니즘을 적절하게 합성하는 일에 관계한다. 이 새로운 메커니즘은 다시 보다 큰 합성 메커니즘의 일부로 작동할 수 있는 것이다.

하나의 객체가 여러 부분으로 이루어졌을 경우에는 언제나, 우리는 그 결과 객체를 두 가지 방식으로 볼 수 있다. 그것을 대체로 마술(혹은 믿음 혹은 가정)에 의해 정해진 성질을 가진, '쪼개지지 않은 하나의 통일체'로 보는 것이 그 하나이다; 이 시각에서는 그 성질만이 관심사이며, 객체가 무슨 부분들로 어떻게 구성되었는지는 관심이 없다. 이 관점에서, 동일한 성질을 가진 두 메커니즘은 어떤 것들이든지 동등하다. 그 객체가 서술된 성질을 왜 가지는지 우리가 이해할 수 있는 '합성 객체'로 그것을 보는 것이 다른 방식이다. 그렇게 되면 우리는 그 (객체의) 부분들을 오직 그 성질만이 중요한 쪼개지지 않은 '작은' 통일체들로 간주한다. 후자의 관점은 우리가 '합성(composition)'으로 의미하는 바를 분명하게 하여 준다. 합성이란 부분들의 성질로부터 전체의 성질이 어떻게 나오는가를 정의하여야만 한다.

이러한 일반적 언급을 한 뒤 우리는 구체적 메커니즘으로 돌아올텐데, 우리가 고려하는 그것의 성질은 그 술어 변환자에 의해 파악된다. 보다 구체적으로, 술어 변환자가 알려진 두 메커니즘 *S1*과 *S2*가 주어졌을 때, 이 두 술어 변환자로부터 새로운 술어 변환자를 유도하는 규칙을 우리가 생각해 낼 수 있는가? 만약 그렇다면, 우리는 이 결과 술어 변환자를 부분 *S1*과 *S2*로부터 특정한 방식으로 구성된 합성 객체의 성질을 기술하는 것으로 간주할 수 있다.

두 개의 주어진 함수로부터 새로운 함수를 하나 유도하는 가장 간단한 방법은 소위 '함수 합성(functional composition)', 즉 한 함수의 (결과) 값을 다른 함수에게 인자로 제공하는 것이다.[21] 이러한 술어 변환자에 대응하는 합성 객체는 '*S1*; *S2*'로 나타내는 것이 관례이며, 우리는 다음과 같이

$$\mathrm{wp}(\text{“}S1;\ S2\text{”},\ R) = \mathrm{wp}(S1,\ \mathrm{wp}(S2,\ R))$$

정의하는데, 우리가 그러고자 한다면 이것을 세미콜런(semicolon)의 의미 정의로 볼 수 있다.

[21] 앞에 나왔던 각주에서 이미 언급했다시피, 술어 변환자는 함수로 볼 수 있다. 우리는 지금 프로그래밍 언어를 설계하고 있다. 앞 장에서 나왔던 성질 1부터 4까지를 만족하여 술어 변환자라고 부를 수 있는, 곧 우리가 그 의미를 확실하게 파악할 수 있는 구조들을 하나씩 언어에 편입시키고 있다. 가장 기본적인 구조에 해당하는 '*skip*', '*abort*', 배정문 등을 도입하였고, 이제 기본적인 구조로부터 적절히 합성하여 이루어진 보다 복잡한 구조를 추가시키려고 한다.

구조의 의미를 포착하는 술어 변환자가 함수라는 것으로부터, 기본적 구조를 합성하여 이루어지는 합성 구조에 대응하는 술어 변환자의 첫번째 후보는 기본적 구조에 대응하는 술어 변환자의 함수 합성으로 만들어진 것이 될 것이다. 함수 합성은 함수들로부터 새로운 함수를 이끌어내는 가장 기본적인 조작이기 때문이다.

주의. *S1*과 *S2*를 위한 술어 변환자가 앞 장의 성질 1부터 4까지를 만족한다는 사실로부터, 위와 같이 정의된 '*S1*; *S2*'를 위한 술어 변환자 역시 이 네 가지 성질을 가진다는 점을 우리는 유도할 수 있다. 예컨대, *S1*과 *S2*에 대해 기적 배제의 법칙이 성립하므로:

$$\mathrm{wp}(S1,\, F) = F \text{이고 } \mathrm{wp}(S2,\, F) = F$$

위의 정의에서 R 대신에 F를 치환함으로써,

$$\begin{aligned}\mathrm{wp}(\text{“}S1;\ S2\text{”},\, F) &= \mathrm{wp}(S1,\, \mathrm{wp}(S2,\, F)) \\ &= \mathrm{wp}(S1,\, F) \\ &= F\end{aligned}$$

다른 세 가지 성질도 역시 성립한다는 증명은 독자를 위한 연습 문제로 남겨둔다.

진행하기 전에, 세미콜런의 의미에 대한 우리의 형식적 정의가 그것에 대한 우리의 직관적 이해(만약 우리가 그런 것을 가지고 있다면!), 즉 합성 메커니즘 '*S1*; *S2*'는 "먼저 *S1*을 활성화시키고, 그 동작이 종료했을 때 *S2*를 활성화시킨다"는 규칙에 의해 구현될 수 있다는 것을 포착하고 있음을 스스로 확인해 볼 것이다. 기실(其實), $\mathrm{wp}(\text{“}S1;\ S2\text{”},\, R)$의 정의에서 우리는 R—이 합성 메커니즘에 대한 사후 조건—을 *S2*를 위한 술어 변환자에게 사후 조건으로 제공하며, 이것은 '*S1*; *S2*'의 전체 동작이 *S2*의 동작과 더불어 끝맺을 수 있다는 점을 반영한다; *S2*에 대해 상응하는 최약 사전 조건, 즉 $\mathrm{wp}(S2,\, R)$은 *S1*을 위한 술어 변환자에게 사후 조건으로 제공되므로 곧 우리는 *S2*를 위한 초기 상태와 *S1*을 위한 최종 상태를 명백하게 동일시하는 것이다. 그런데 이러한 사항은 시간적으로 *S1*의 동작 다음에 *S2*의 활성화가 뒤따를 때 정확히 반영된다.

그냥 확실하게 하기 위해 예를 하나 생각해 보자. '*S1*; *S2*'를 다음

$$\text{‘}a := a + b;\ b := a * b\text{’}$$

이라고 하고, 사후 조건을 어떤 술어 $R(a, b)$라고 하자. 그 경우,

$$\begin{aligned}\mathrm{wp}(S2,\, R(a,b)) &= \mathrm{wp}(\text{“}b := a * b\text{”},\, R(a,b)) \\ &= R(a, a * b)\end{aligned}$$

이고

$$\begin{aligned}\mathrm{wp}(\text{“}S1;\ S2\text{”},\, R(a,b)) &= \mathrm{wp}(S1,\, \mathrm{wp}(S2,\, R(a,b))) \\ &= \mathrm{wp}(S1,\, R(a, a * b))\end{aligned}$$

$$= \mathrm{wp}(\text{“}a := a + b\text{”},\ R(a, a * b))$$
$$= R(a + b, (a + b) * b)$$

이다. 즉, 처음에 $a + b$와 $(a + b) * b$ 사이에 R 관계가 성립한다면, 우리는 a와 b의 최종 값들 사이에 동일한 관계를 보장할 수 있다.

마지막으로, 함수 합성은 결합 법칙이 성립하므로 우리가 '*S1*; *S2*; *S3*'을 '[*S1*; *S2*]; *S3*'으로 파싱하든지 '*S1*; [*S2*; *S3*]'으로 파싱하든지 상관이 없다. 즉, 우리는 정말로 세미콜런을 일종의 접합(concatenation) 기호로 간주할 수 있으며, 우리가 '*S1*; *S2*; *S3*; . . . ; *Sn*' 형태의 문장들 목록을 적을 때 아무런 모호성도 존재하지 않으므로, 기회가 주어지면 자유로이 그렇게 할 것이다.

연습 문제

다음 두 식

'*x1* := *E1*; *x2* := *E2*'와 '*x2* := *E2*; *x1* := *E1*'

은, 만약 변수 *x1*이 표현 *E2*에 나타나지 않고 동시에 변수 *x2* 역시 표현 *E1*에 나타나지 않는다면 의미상 동등하다는 것을 증명하라. 사실, 그렇게 되면 위의 두 식은 병행 배정 '*x1*, *x2* := *E1*, *E2*'와 의미상 동등하다. (이 동등성이 병행 배정을 장려하는 논거의 하나이다; 그것을 사용하면 우리가 지나친 순차적(順次的) 명세를 피할 수 있게 해 주며, 더욱이 병행 배정에서는 두 표현 *E1*과 *E2*가 병행하여 평가될 수도 있다는 것이 명백한데, 이 사실은 어떤 구현 기법에 있어서는 흥미를 가질 수 있다. 그 이외에, 우리는 어쩌면 보다 흥미로울 '*x1*, *x2* := *E1*, *E2*'가 의미상 '*x1* := *E1*; *x2* := *E2*'와도 동등하지 않고 '*x2* := *E2*; *x1* := *E1*'과도 동등하지 않을 가능성을 가지고 있다.