튜토리얼

프로그래밍의 수련(修練)
(A Discipline of Programming)

엣져 위베 다익스트라
(Edsger Wybe Dijkstra)

다섯번째: 프로그래밍 언어의 의미 규정 — II

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

4장. 프로그래밍 언어의 의미 규정(意味規定) — II

세미콜런을 도입하기 이전에는, 우리는 하나의 문장으로 이루어진 프로그램만 작성할 수 있었다; 세미콜런의 도움으로 $n(n > 0)$개 문장의 접합(接合; concatenation), 즉 "*S1*; *S2*; . . . ; *Sn*"으로 프로그램을 작성할 수 있다. 중간에 종료하지 않는 것을 제외하였을 때, 이러한 프로그램을 실행하게 되면 먼저 *S1*, 다음에 *S2*, 이런 식으로 *Sn*까지 n개 문장들이 시간적으로 연속됨을 항상 의미한다. 그러나 유클리드 알고리즘을 구현한 판지(cardboard) 게임의 예로부터, 우리는 보다 폭넓은 '그 게임[1]의 규칙'을 기술할 수 있어야만 한다는 것을 알고 있다: 각 게임에는 동작의 연속이 있고 그 동작은 "$x := x - y$" 또는 "$y := y - x$"인데, 이 동작들이 시간적으로 보아 어떤 순서로 번갈아 행해지는지는 물론이고, 심지어 (그 동작들의) 전체 횟수까지 매 게임에 따라서 달라진다; 이것은 조약돌의 첫 위치에 달려 있으며, (바꿔 말하면) 그 시스템의 초기 상태에 달려 있는 것이다. 만약 세미콜런이 주어진 부분들로부터 새로운 전체를 구성하기 위한 유일한 수단이라면 우리는 앞의 상황을 표현할 수 없으며, 따라서 뭔가 새로운 것을 찾아야만 한다.

세미콜런만이 우리가 가지고 있는 연결자(connective)인 한, 구성 메커니즘 중 하나인 $Si(i > 1)$가 활성화되는 유일한 상황은 사전적(辭典的; lexicographic) 순서에서 (Si를) 선행(先行)하는 것들이 적절히 종료하는 것이다. 우리가 필요로 하는 융통성을 달성하기 위해서는, 메커니즘 또는 하위 메커니즘(submechanism)의 활성화가 시스템의 현재 상태에도 함께 의존하게[2] 하는 일이 가능해야만 한다. 이럴 목적으로 우리는—두 단계에 걸쳐서—'가드 명령(guarded command)'이라는 개념을 도입하는데, 그 구문은 다음과 같다:

<가딩 헤드> ::= <부울 표현> → <문장>
<가드 명령> ::= <가딩 헤드> {; <문장>}

여기서 중괄호의 쌍인 '{'와 '}'는 다음과 같이 읽어야 한다: "중괄호에 둘러싸인 것이 0번 혹은 그 이상 나타나는 부분이 뒤따른다".

(가드 헤드를 위한 구문의 다른 대안(代案)은 다음과 같은 것이 될 수도 있었을 것이다:

[1] 여기서의 '게임'이란 판지를 사용하여 유클리드 알고리즘에 의한 두 수의 최대공약수를 구하는 일을 지칭한다.

[2] 즉, Si의 활성화가 $Sj(1 \leq j < i)$가 활성화 된 이후 적절히 종료하는 것뿐만 아니라 시스템의 상태에 따라 달라지거나, 어떤 때는 이런 일, 다른 때는 저런 일을 하는 식이 가능하게 됨을 의미한다.

<문장 목록> ::= <문장> {; <문장>}
<가드 명령> ::= <부울 표현> → <문장 목록>

그러나 이제는 우리와 별 관계 없는 이유 때문에, 나는 가딩 헤드(guarding head)라는 개념을 도입하는 구문을 선호한다.)

여기에서 화살표 앞에 오는 부울 표현(boolean expression)을 '가드(guard)'라고 부른다. 이것은 다음과 같은 착상(着想)이다: 화살표 뒤에 오는 문장 목록(statement list)은 대응하는 가드가 처음에 참인 경우에만 실행되리라는 것이다. 가드 덕분에 우리는 실행이 바람직하지 않거나 혹은 부분 연산(partial operation)이 수반되어서 아예 불가능한 초기 상황 하에서, 문장 목록의 실행을 방지할 수 있게 된다.

가드가 참이라는 것은 한 덩어리로서의 그 가드 명령이 실행되기 위한 초기 필요 조건(necessary condition)이다; 물론 충분 조건은 아닌데, 그 이유는 이런 혹은 저런 방식으로—우리는 그런 방식들 두 가지를 만날 것이다—(실행) 시기 또한 '자신의 차례'여야만 하기 때문이다. 이것이 가드 명령을 문장으로 간주하지 않는 이유이다: 문장이란 자기 차례가 오면 예외 없이 실행되는 것이고, 가드 명령은 문장을 만들기 위한 하나의 구성 블럭(building block)으로 이용할 수 있다.[3] 좀더 정확하게 말하자면, 가드 명령들의 집합으로 문장을 구성하는 두 가지 다른 방법을 제안할 것이다.

잠시 숙고(熟考)해 보면, 가드 명령들의 집합을 고려하는 것이 너무나 당연하다. 우리가 만약 초기 상태가 Q를 만족하면 최종 상태가 R을 만족하는 메커니즘을 구성할 것을 요구받았다고 가정하자. 여기에 더해서, 모든 경우에 이 일을 수행하는 하나의 문장 목록은 우리가 발견할 수 없다고 생각해 보자. (만일 그런 문장 목록이 존재한다면, 우리는 바로 그것만 사용하면 될 것이고 가드 명령은 필요가 없을 것이다.) 그러나 우리는, 가능한 초기 상태의 부분 집합들 각각에 대해 이 일을 달성하는 여러 개의 문장 목록들을 찾을 수 있을지 모른다. 이 문장 목록들 각각에 대해 우리는 그 부분 집합을 규정하는 데 적절한 부울 표현을 가드로서 부착할 수 있고, Q의 참이 적어도 하나의 가드의 참을 함축(含蓄; implication)하는 것과 같이 충분히 포괄적(包括的)인 가드들을 가지게 되었을 때, 우리는 R을 만족하는 상태로 시스템을 옮겨가는 메커니즘(즉 가드가 초기에 참인 가드 명령들 중 하나)을 Q를 만족하는 모든 초기 상태에 대해 가지게 되는 것이다.

이것[4]을 표현하기 위해 우리는 먼저 다음

<가드 명령 집합> ::= <가드 명령> {| <가드 명령>}

을 정의하는데, 여기서 기호 '|'('바(bar)'라고 발음한다)는 순서가 없는 대안(alternative)들 사이의 분리자(separator)로 기능한다. 가드 명령 집합으로부터 문장을 형성하는 방법들 중 한 가지는 그것[5]을 '**if** . . . **fi**' 쌍으로 둘러싸

[3]여기서 사용되는 '문장'의 의미는 이 장의 앞 부분에서 '문장'이라는 용어보다는 '명령'이라는 용어가 적합함을 이야기 할 때의 그 '문장'이 아니라, 프로그래밍 언어 분야에서 현실적으로 통용되는 것에 바탕을 두고 있다.

[4]가드 명령들로 구성되는 문장을 말한다.

[5]가드 명령들의 집합을 지칭한다.

는 것이다. 즉 '문장'이라고 부르는 구문적 범주를 위한 우리의 구문은 다음

<문장> ::= **if** <가드 명령 집합> **fi**

형태를 사용해 확장된다.

이것은 여러 개의 가드 명령들을 하나의 새로운 메커니즘으로 조합할 수 있는 하나의 특별한 방식이다. 우리는 이 메커니즘이 활성화되었을 때 일어나는 동작을 다음과 같이 볼 수 있다. 가드가 참인 가드 명령들 중 하나가 선택되고 그것의 문장 목록이 활성화된다. 우리의 새로운 구조(構造; construct)[6]의 의미에 대한 정형적 정의를 내리려고 진행하기 전에, 세 가지를 언급하는 것이 적절하다.

1. 모든 가드들이 정의된다고 가정한다; 만약 그렇지 않다면, 즉 가드의 평가(評價; evaluation)[7]가 적절하게 종료하는 동작을 일으키지 않는다면, 전체 구조가 적절하게 종료하는 데 실패하게 된다.

2. 일반적으로 우리의 구조는 비결정성(非決定性; nondeterminacy)을 발생시킨다. 즉 두 개 이상의 가드들이 참인 모든 초기 상태에 대해서, (참인 가드들에) 대응하는 문장 목록들 중 어느 것이 활성화되려고 선택될 것인지가 정의되지 않은 채로 남기 때문이다. 만약 어떤 두 가드들이 서로를 배제한다면[8] 비결정성은 수반되지 않는다.

3. 만일 모든 가드가 참이 아닌 초기 상태라면, 우리는 그 대안들 중 어떤 것도, 따라서 한 덩어리로서의 그 구조 역시 다룰 수 없는 초기 상태와 마주친 것이다. 그러한 초기 상태에서의 활성화는 취소(abortion)로 이어질 것이다.[9]

---

[6] **if** . . . **fi** 구조를 가르킨다.

[7] 부울 표현인 가드의 진리 '값을 계산하는 일'을 말한다.

[8] 한 가드가 참이면 다른 가드가 거짓인 식으로, 두 개의 가드들이 동시에 성립하지는 않는 경우를 말한다.

[9] 여기서의 '취소'는 이 장의 앞 부분에서 나온 *abort* 문 혹은 메커니즘이 실행되는 것을 의미한다. 이 문장이 의미하는 바와 관련해서는 함께 생각해 볼 만한 점이 있다. 명백한 것이지만, **if** . . . **fi** 구조 혹은 문장은 오늘날의 일반적인 고급 프로그래밍 언어에 존재하는 다중 선택을 위한 **case** 문(Pascal 계열 언어에서)이나 **switch** 문(C 계열 언어에서)에 해당한다. 실제 고급 프로그래밍 언어에서는 선택을 위한 문장에서 해당하는 경우가 없을 때 아무런 일도 하지 않고 그냥 그 문장의 실행을 마치는 언어가 많다. 즉 이 책에서 다익스트라가 설계한 '미니 언어'의 *skip* 문과 똑같은 효과를 일으킨다. (물론 그렇지 않은 언어도 많이 있다. 예컨대 Pascal, Modula-2 등은 선택 구조에서 다루는 경우 외의 값이 나타나면 오류로 취급한다. 그러나 Algol 68과 요즘 실제로 많이 쓰이고 있는 C 계열 언어들은 그렇지가 않다.) 그러나 다익스트라는 그런 경우 *abort* 문과 똑같은 효과가 일어나도록 정의하고 있다. *skip* 문과 *abort* 문은 완전히 다른 의미를 지니고 있다. 독자들은 어떻게 생각하는가? 어느 것이 옳은 방향이라고 생각하는가? 해역자의 개인적 생각은 다익스트라의 견해가 원론적으로 지당하다고 본다. **if** . . . **fi** 구조를 도입하기 전의 서론 단계에서 다익스트라가 충분히 설명하였다시피, 우리가 선택 구조를 도입하는 이유는 상황과 경우에 따라 어떨 때는 이러한 일을, 다른 때는 저러한 일을 선택적으로 하길 원하기 때문이다. 가드는 그러한 각 경우를 규정하기 위한 것이다. 또한 선택 구조는 여러번 실행되는 것이 아니라, 단 한 번 실행되는 것이다. 한 번 실행되는 선택 구조가 실행되는 순간에 참인 가드가 없다면, 그 선택 구조는 아무런 일을 하지 않는 셈이다. 정말로 아무런 일을 하지 않아도 괜찮다면 그 선택 구조는 아예 없애 버리는 것이 옳을 것이고, 설계시

주의. 만약 우리가 빈 가드 명령 집합도 역시 허용한다면, 문장 '**if fi**'는 따라서 의미 상으로 앞에 나온 문장 '*abort*'와 동등하다.

(다음에 나오는 **if-fi**-구조를 위한 최약 사전 조건의 정형적 정의에서, 우리는 모든 가드들이 전체 함수(total function)인 경우로만 한정할 것이다. 만약 그런 경우가 아니라면, (가드의 부울) 표현 앞에 초기 상태가 모든 가드들의 정의 구역 내에 있어야 한다는 추가의 요구 사항을 **cand**를 사용해 덧붙여야 한다.)

'IF'를 다음 문장

$$\textbf{if } B_1 \to SL_1 \mid B_2 \to SL_2 \mid \ldots \mid B_n \to SL_n \textbf{ fi}$$

의 이름이라고 하면, 임의의 사후 조건 R에 대해서

$$\begin{aligned} \mathrm{wp}(\mathrm{IF},\, R) = {} & (\textbf{E } j : 1 \le j \le n : B_j) \textbf{ and} \\ & (\textbf{A } j : 1 \le j \le n : B_j \Rightarrow \mathrm{wp}(SL_j,\, R)) \end{aligned}$$

이 성립한다. 이 식은 다음과 같이 읽어야 한다: B_j를 참으로 만드는 j가 $1 \le j \le n$ 범위에 적어도 하나가 존재하며, 거기에 더해 B_j를 참으로 만드는 $1 \le j \le n$ 범위 내의 모든 j에 대해서 $\mathrm{wp}(SL_j,\, R)$도 역시 참이 되는 상태 공간의 모든 점에서 $\mathrm{wp}(\mathrm{IF},\, R)$은 참이다. IF 자체의 정의에서 우리가 한 것처럼 '. . .'를 사용하면, 다음

$$\begin{aligned} \mathrm{wp}(\mathrm{IF},\, R) = {} & (B_1 \textbf{ or } B_2 \textbf{ or } \ldots \textbf{ or } B_n) \textbf{ and} \\ & (B_1 \Rightarrow \mathrm{wp}(SL_1,\, R)) \textbf{ and} \\ & (B_2 \Rightarrow \mathrm{wp}(SL_2,\, R)) \textbf{ and } \ldots \textbf{ and} \\ & (B_n \Rightarrow \mathrm{wp}(SL_n,\, R)) \end{aligned}$$

의 대안 형태를 제시할 수도 있다.

이 식들을 이해하는 일은 그리 어렵지 않다. 가드들 중 적어도 하나가 참이어야 한다는 요구 조건은 모든 가드들이 거짓인 경우의 취소(abortion)를 반영한다. 거기에 더해 $\mathrm{wp}(\mathrm{IF},\, R)$을 만족하는 모든 초기 상태에 대해, 우리는 모든 j에 대해 $B_j \Rightarrow \mathrm{wp}(SL_j,\, R)$일 것을 요구한다. B_j가 거짓인 j의 값들에 대해서, 이 함축(implication)은 $\mathrm{wp}(SL_j,\, R)$의 값에 관계없이 참이다. 즉, 그런 j 값들에 대해서는 SL_j가 하는 일이 무엇이든 명백히 상관이 없다. 우리의 구현[10]은 처음에 가드 B_j가 거짓이면 활성화시키려고 SL_j를 선택하지 않음

선택 구조가 꼭 필요하다고 생각되었는데 정작 실행될 때 참인 가드가 하나도 없다는 것은 뭔가 일이 잘못되고 있음을 반증한다고 본다. 그러면 왜 현실의 프로그래밍 언어에서는 참인 가드가 하나도 없을 때 아무런 불평 없이 그냥 아무 일도 하지 않고 넘어가는가? Pascal 계열 언어와 공통의 조상을 가지고 있는 Algol 68의 이러한 설계는 실수라고 본다. 반면에 C는 그 설계 철학이나 염두에 둔 대상 사용자 집단을 고려할 때 당연한 결정이라고 여겨진다. 전문가의 사용을 기본적으로 가정하고 있기에, 프로그램 내의 모든 구성은 전적(全的)으로 프로그래머에게 맡겨져 있고, 선택 구조를 사용함에도 불구하고 참인 가드가 하나도 없는 경우도 프로그래머 나름대로 생각이 있었을 것이라고 간주하는 것이 그 언어의 설계 철학에도 맞는 일이다.

[10] 여기서의 구현이란 다익스트라가 정의하는 '미니 언어'의 수행 의미론(operational seman-

으로써 이 사실을 반영하고 있다. B_j가 참인 j의 값들에 대해서는, 이 함축은 $\mathrm{wp}(SL_j,\, R)$도 역시 참일 때만 참일 수 있다. 우리의 정형적 정의는 j의 모든 값들에 대해서 이 함축의 참을 요구하므로, 우리의 구현은 두 개 이상의 가드들이 참일 때에도 선택에 있어서 완전히 자유롭다.

if-fi-구조는 우리가 가드 명령 집합으로부터 문장을 구성할 수 있는 두 가지 방법들 중 하나일 따름이다. 이 **if-fi**-구조에서는 모든 가드들이 거짓인 상태가 취소에 이르렀다; 두번째 형태에서는 모든 가드들이 거짓인 상태가 적절한 종료에 이르도록 우리는 허용하고, 그 때는 아무런 문장 목록도 활성화되지 않으므로 이 (두번째) 형태는 의미 상으로 공백 문장(empty statement)과 동등해질 것이 그냥 자연스럽다; 그렇다면 참인 가드가 없을 때 적절히 종료하도록 이렇게 허용하는 일의 반대는 하나의 가드라도 참인 동안은 실행이 종료하지 않도록 하는 것이다. 즉, 활성화되면[11] 가드들을 검사한다. 참인 가드가 없다면 실행은 종료한다; 참인 가드들이 있다면 대응되는 문장 목록들 중 하나가 활성화되고, 그것이 종료할 때 이 구현은 가드들을 검사하며 처음부터 다시 시작한다. 이 두번째 구조는 가드 명령 목록을 '**do** . . . **od**' 쌍으로 둘러싸서 나타낸다.

이 **do-od**-구조를 위한 최약 사전 조건의 정형적 정의는 **if-fi**-구조를 위한 것보다 복잡하다; 실제로 전자는 후자를 사용하여 표현된다. 우리는 먼저 그 정형적 정의를 내린 뒤 그 설명을 할 것이다. 'DO'를 다음 문장

$$\textbf{do}\ B_1 \to SL_1 \mid B_2 \to SL_2 \mid \ldots \mid B_n \to SL_n\ \textbf{od}$$

의 이름이라고 하고, 'IF'를 동일한 가드 명령 집합을 '**if** . . . **fi**' 쌍으로 둘러싸서 만든 문장의 이름이라고 하자. 조건 $H_k(R)$이 다음

$$H_0(R) = R\ \textbf{and non}\ (\mathbf{E}\ j : 1 \le j \le n : B_j)$$

식과 $k > 0$에 대해서는 다음

$$H_k(R) = \mathrm{wp}(\mathrm{IF},\, H_{k-1}(R))\ \textbf{or}\ H_0(R)$$

식으로 주어지면, 다음

$$\mathrm{wp}(\mathrm{DO},\, R) = (\mathbf{E}\ k : k \ge 0 : H_k(R))$$

식이 성립한다.

$H_k(R)$의 직관적인 이해는 다음과 같다: 곧 가드 명령을 최대로 k번 선택한 뒤, 시스템을 사후 조건 R을 만족하는 최종 상태에 남겨 두고 **do-od**-구조가 종료하도록 하는 최약 사전 조건이라는 것이다.

$k = 0$에 대해서는 **do-od**-구조가 어떤 가드 명령도 선택하지 않고 종료할

---

tics)적 측면에서 각 문장의 의미를 뜻한다고 볼 수 있다. 즉 각 문장이 어떤 식으로 실행되느냐 하는 것이다.

[11] 가드 명령 집합으로부터 만들어지는 두번째 문장 혹은 구조가 활성화된다는 뜻이다.

것 즉 참인 가드가 존재하지 할 수 없다는 것이 요구되는데, 이것은 두번째 항[12]에 표현되어 있다; 그러면 처음에 R이 참인 것은 명백히 R이 최종적으로 참이 되기 위한 필요하고 충분한 추가 조건이며, 첫번째 항에 표현되어 있다.

$k > 0$에 대해서는 두 가지 경우를 구분해야 한다: 참인 가드가 하나도 없을 수 있으며, 이럴 때는 R이 성립해야만 하고 두번째 항이 도출된다; 혹은 적어도 하나의 가드는 참인데, 이때 일어나는 일은 문장 'IF'가 일단 활성화되는 것처럼 시작한다(참인 가드가 없어서 즉각적으로 취소되지는 않는 초기 상태에서). 그러나 그 실행 후에는, (이제) 하나의 가드 명령은 선택되(어 실행되)었으니까 R을 만족하는 최종 상태에서 종료하는 것을 보장하기 위해 최대 $k-1$번의 추가 선택이 필요한 상태에 도달했음을 확인해야만 한다. 우리 정의에 따르자면 문장 'IF'를 위한 이 사후 조건은 $H_{k-1}(R)$이다.

wp(DO, R)을 정의하는 마지막 줄은 사후 조건 R을 만족하는 최종 상태에서 종료하는 것을 보장하기 위해 최대 k번의 선택이 필요한 k 값이 존재하여야만 한다는 점을 표현한다.

> 주의. 만일 우리가 빈 가드 명령 집합도 허용한다면, 문장 '**do od**'는 결국 앞에 나온 문장 '*skip*'과 의미 상으로 동등해진다.

[12] '**non** (**E** $j : 1 \leq j \leq n : B_j$)'를 말한다.