02장 프로그래밍 언어 구문

2.4 이론적 기초 사항

책에서 살펴봤듯이 어휘 분석기와 구문 분석기는 촘스키 언어 계층의 마지막 두 단계를 형성하는 유한 오토마타와 푸쉬다운 오토마타에 기반한다. 촘스키 계층의 각 단계에서마다 기계는 결정적이거나 비결정적일 수 있다. 결정적 오토마타는 주어진 상황에서 항상 동일한 연산을 수행한다. 비결정적 오토마타는 연산 집합의 아무 연산이나 수행할 수 있다. 비결정적 기계는 각 상황에서의 연산 결정이 기계로 하여금 최종적으로 "예"를 출력하게 해준다면 해당 문자열을 수용한다. 비결정적 유한 오토마타와 결정적 오토마타는 동일한 표현력을 가진다. 정의에 의해 모든 DFA는 퇴화된 NFA이며, 예 2.12의 과정을 통해 모든 NFA에 대해 이와 동일한 언어를 수용하는 DFA를 생성할 수 있다는 것을 알아봤다. 그러나 푸쉬다운 오토마타의 경우에는 그렇지 않다. NPDA는 수용하지만 DPDA는 수용하지 못하는 문맥 자유 언어가 존재한다. 다행히 실제 프로그래밍 언어의 구문을 수용하는 데는 DPDA로도 충분하다. 실제로 어휘 분석기와 구문 분석기는 항상 결정적이다.

【2.4.1】 유한 오토마타

명확히 정의하면 결정적 유한 오토마타(DFA, deterministic finite automaton) M은 (1) 상태의 유한 집합 Q, (2) 입력 기호의 유한 알파벳 Σ, (3) 구별되는 초기 상태 $q_1 \in Q$, (4) 구별되는 최종 상태 집합 $F \subseteq Q$, (5) 현재 상태와 현재 입력 기호에 기반해서 M을 위한 새로운 상태를 선택하는 전이 함수 $\delta: Q \times \Sigma \rightarrow Q$로 구성된다. M은 상태 q_1에서 시작한다. 그리고 δ를 사용해서 상태를 이동해가며 입력 기호를 하나씩 소비한다. 최종 기호가 소비되었을 때 M이 F에 속하는 상태에 있으면 M은 "예"를 출력한다고 해석된다. 그렇

지 않은 경우에는 "아니오"를 출력하는 것으로 해석된다. 입력으로 기호가 아니라 문자열을 받아들이게 δ를 확장하는 것은 어렵지 않다. 이렇게 확장하면 $\delta(q_1, x) \in F$인 경우 M이 문자열 x를 수용한다고 말할 수 있다. 그 다음 $\{x \mid \delta(q_1, x) \in F\}$로 M이 수용하는 언어 L(M)을 정의할 수 있다. 비결정적 유한 오토마타(NFA, nondeterministic finite automaton)에서 전이 함수 δ는 다중 값을 가진다. 즉, 비결정적 유한 오토마타는 입력과 상태가 주어졌을 때 가능한 상태들의 어떤 집합으로도 이동할 수 있다. 또 비결정적 유한 오토마타는 "저절로" 한 상태에서 다른 상태로 이동할 수 있다. 이러한 전이는 입력 기호 ϵ을 취한다고 한다.

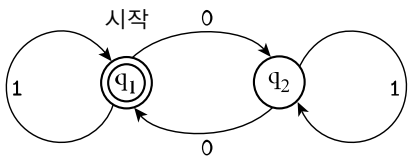

그림 2.32 | 0과 1로 구성되며 0의 개수가 짝수인 모든 문자열을 수용하는 최소 DFA. 책의 그림 2.10을 다시 보인 것이다.

예 2.53
(1*01*0)*1*의 형식 DFA

예를 통해 이러한 정의를 알아볼 수 있다. ©(심화학습에 있는) 그림 2.32(책의 그림 2.10을 다시 출력한 것)의 원과 화살표 오토마타를 보자. 이 그림은 0과 1로 구성되며 0의 개수가 짝수인 문자열을 수용하는 최소 DFA다. $\Sigma=\{0, 1\}$이 기계의 입력 알파벳이다. $Q=\{q1, q2\}$는 상태 집합으로 q1은 초기 상태며 F={q1}은 최종 상태 집합이다. 전이 함수는 $\delta=\{(q_1, 0, q_2), (q_1, 1, q_1), (q_2, 0, q_1), (q_2, 1, q_2)\}$와 같은 트리플 집합으로 나타낼 수 있다. 트리플 (q_i, a, q_j)에서 $\delta(q_i, a) = q_j$다.

예 2.10과 2.12를 통해 임의의 정규식에 의해 생성된 언어를 수용하는 NFA와 임의의 NFA와 동등한 DFA가 존재한다는 것을 알 수 있다. 정규식과 유한 오토마타가 등가의 표현력을 가진다는 것을 보이기 위해 남은 일은 임의의 DFA가 수용하는 언어를 생성하는 정규식이 존재함을 보이는 것이다. 아래에서는 "짝수 개의 0" 예(©심화학습에 있는 그림 2.32)를 통해 이에 필요한 과정을 살펴본다. 모든 정규 언어 구성에 대한 좀 더 형식적이고 일반적인 내용은 표준 오토마타 이론 교과서[HMU01, Sip97]에서 찾아볼 수 있다.

DFA에서 정규식으로

주어진 DFA와 등가의 정규식을 만들기 위해 좀 더 간단한 하위 문제에 대한 해결책 표로부터 연속적으로 좀 더 복잡한 하위 문제에 대한 해결책을 생성하는 동적 프로그래밍 알고리즘을 사용한다. 좀 더 명확히 말해서 전이 함수 δ를 기술하는 간단한 정규식에서 시작한다. 모든 상태 i에 대해 다음을 정의한다.

$$r_{ii}^{0} = a_1 \mid a_2 \mid \ldots \mid a_m \mid \epsilon$$

여기서 $\{a_1 \mid a_2 \mid \ldots \mid a_m\} = \{a \mid \delta(q_i, a) = q_i\}$는 상태 q_i에서 자기 자신으로 돌아오는 "자가 루프"에 레이블을 붙이는 문자들의 집합이다. 이러한 자가 루프가 없는 경우 $r_{ii}^0 = \epsilon$이다.

이와 유사하게 $i \neq j$에 대해 다음과 같이 정의한다.

$$r_{ij}^0 = a_1 \mid a_2 \mid \ldots \mid a_m$$

여기서 $\{a_1 \mid a_2 \mid \ldots \mid a_m\} = \{a \mid \delta(q_i, a) = q_j\}$는 q_i에서 q_j로의 호에 레이블을 붙이는 문자들의 집합이다. 그러한 호가 없으면 r_{ij}^0는 빈 정규식이 된다(차이점에 주의하자. 어떠한 입력도 받아들이지 않음으로써 상태 q_i에 머물 수 있으므로 (이 r_{ii}^0에서는 항상 대안 중 하나지만 $i \neq j$일 때 r_{ij}^0에서는 그렇지 않다).

이러한 r^0 식이 주어지면 동적 프로그래밍 알고리즘은 귀납적으로 좀 더 큰 위첨자를 가지는 수식 r_{ij}^k의 값을 계산한다. k는 q_i에서 q_j로의 경로상에서 제어가 지나갈 수 있는 상태 중 가장 큰 번호를 가지는 상태를 명명한다. 상태는 q_1에서 시작해서 번호가 매겨진다고 가정하므로 $k=0$이면 q_i에서 q_j로 직접 전이해야 하며 그 사이에는 어떠한 상태도 없어야 한다.

예 2.54

2-상태 DFA를 위한 정규식의 재구성

앞서 살펴본 간단한 예 DFA에서 $r_{11}^0 = r_{22}^0 = 1 \mid \epsilon$이며, $r_{12}^0 = r_{21}^0 = 0$이다. $k>0$인 경우 r_{ij}^k식은 일반적으로 다중 문자 문자열을 생성한다. 동적 프로그래밍 알고리즘의 각 단계에서 $r_{ij}^k = r_{ik}^{k-1} \mid r_{ik}^{k-1} \quad r_{kk}^{k-1} * \; r_{kj}^{k-1}$라고 놓는다. 즉, q_i에서 q_j에 k보다 큰 번호를 가지는 상태를 통하지 않고 도달하려면 $k-1$보다 큰 번호를 가지는 상태를 통해 q_i에서 q_j로 가든가(이미 어떻게 해야 할지 아는 경우) 아니면 q_i에서 q_k($k-1$보다 큰 번호를 가지는 상태를 통하지 않고)로 가고 (중간에 $k-1$보다 큰 번호를 가지는 상태는 절대 방문하지 않고) 임의의 횟수만큼 q_k에서 탐색을 한 후 끝으로 (역시 $k-1$보다 큰 번호를 가지는 상태를 방문하지 않고) q_k에서 q_j로 갈 수 있다.

구성 요소가 되는 정규식 중 빈 것이 있으면 가장 바깥쪽 대안의 항을 생략한다. 결국 전체 정답은 $r_{1f_1}^n \mid r_{1f_2}^n \mid \ldots \mid r_{1f_t}^n$가 되며 여기서 $n = \mid Q \mid$는 전체 상태의 개수며 $F = \{qf_1, qf_2, \ldots qf_t\}$는 최종 상태 집합이다. 현재 다루고 있는 예의 첫 번째 귀납 단계는 다음과 같다.

$$r_{11}^1 = (1 \mid \epsilon) \mid (1 \mid \epsilon)\,(1 \mid \epsilon)^*\,(1 \mid \epsilon)$$

$$r_{12}^1 = 0 \mid (1 \mid \epsilon)\,(1 \mid \epsilon)^*\,0$$

$$r_{22}^1 = (1 \mid \epsilon) \mid 0\,(1 \mid \epsilon)^*\,0$$

$$r_{21}^1 = 0 \mid 0\,(1 \mid \epsilon)^*\,(1 \mid \epsilon)$$

두 번째와 마지막 귀납 단계는 다음과 같다.

$$r_{11}^{2} = ((1|\epsilon)|(1|\epsilon)(1|\epsilon)^*(1|\epsilon))|$$
$$(0|(1|\epsilon)(1|\epsilon)^*0$$
$$((1|\epsilon)|0(1|\epsilon)^*0)^*$$
$$(0|0(1|\epsilon)^*(1|\epsilon))$$

F에는 멤버(q_1)가 하나뿐이기 때문에 이 수식이 최종 정답이 된다. 물론 이것이 최소 형태는 아니지만 올바르긴 하다.

【2.4.2】 푸쉬다운 오토마타

결정적 푸쉬다운 오토마타(DPDA, deterministic push-down automaton) N은 DFA와 같이 (1) Q, (2) Σ, (3) q_1, (4) F와 그 외에 (6) 스택 기호들의 유한 알파벳 Γ, (7) 구별되는 초기 스택 기호 $Z_1 \in \Gamma$, (5') 전이 함수 δ: $Q \times \Gamma \times \{\Sigma \cup \{\epsilon\}\} \rightarrow Q \times \Gamma^*$로 구성된다. 여기서 Γ^*는 Γ에 있는 0개 이상의 기호로 구성된 문자열들의 집합이다. N은 상태 q_1에서 빈 스택에 Z_1을 가지고 시작한다. N은 현재 생태 q와 스택 상단 기호 Z를 반복적으로 조사한다. $\delta(q, \epsilon, Z)$가 정의되어 있다면 N은 상태 r로 이동해 Z를 스택에 있는 α로 대체한다. 여기서 $(r, \alpha) = \delta(q, \epsilon, Z)$이다. 이 경우 N은 입력 기호를 소비하지 않는다. $\delta(q, \epsilon, Z)$가 정의되어 있지 않다면 N은 현재 입력 기호 a를 보고 소비한다. 그 후 상태 s로 이동해 Z를 β로 대체한다. 여기서 $(s, \beta) = \delta(q, a, Z)$이다. N은 입력 기호들의 문자열을 소비한 후 상태 F에 있을 때, 그리고 그 때에만 입력 기호의 문자열을 수용하는 것으로 해석된다.

유한 오토마타의 경우와 마찬가지로 비결정적 푸쉬다운 오토마타(NPDA, nondeterministic push-down automaton)는 다중 값을 가지는 전이 함수로 구별된다. NPDA는 상태, 입력, 스택 상단 기호가 주어졌을 때 새로운 상태와 스택 기호 대체를 임의로 선택할 수 있다. $\delta(q, \epsilon, Z)$가 비지 않았다면 N은 현재 입력 기호를 조사하거나 소비하지 않고 새로운 상태와 스택 기호 대체를 선택할 수도 있다. 비결정적 유한 오토마타와 결정적 유한 오토마타는 동일한 표현력을 가지지만 푸쉬다운 오토마타에서는 그렇지 않다. NPDA는 수용하지만 DPDA는 수용하지 못하는 문맥 자유 언어가 존재한다.

CFG와 NPDA의 표현력이 동등하다는 것의 증명은 정규식과 유한 오토마타의 표현력이 동등함을 보이는 것보다 복잡하다. 또 실질적으로 이 증명은 컴파일러 구성에 제한적으로 중요하며 여기서는 다루지 않는다. 임의의 CFL에 대한 NPDA를 생성할 수 있지만 그 NPDA는 어떤 경우 해당 언어에 속하는 문자열을 인식하는 데 지수적 시간을 필요로 할 수 있다(2.3절에서 언급한 $O(n^3)$ 알고리즘은 PDA의 형태를 취하지 않는다). 실제 사용되는 프로그래밍 언어는 모두 (결정적) PDA로 선형 시간에 구문 분석할 수 있는

LL 문법이나 LR 문법을 사용해서 표현할 수 있다.

LL(1) PDA는 매우 간단하다. LL(1) PDA는 현재 입력 토큰과 스택 상단 기호만 기반으로 결정을 내리기 때문에 상태 그림이 자명하다. 한 전이를 제외한 모든 전이가 초기 상태에서 자기 자신으로의 자가 루프다. 입력과 스택에서 $$이 나오면 최종 전이는 초기 상태에서 두 번째 최종 상태로 이동한다. 2.3.3절에서 알아봤듯이 SLR(1) 구문 분석기나 LALR(1) 구문 분석기를 위한 상태 그림은 특유의 유한 상태 기계(CFSM, characteristic finite-state machine)로 훨씬 더 흥미로운 것이다. 완전 LR(1) 구문 분석기도 유사한 기계를 가지지만 보통 경로 특화 미리 보기에 대한 필요성 때문에 좀 더 많은 상태를 가진다.

모든 상태를 수용하는 상태로 정의한다면 스택이 없는 CFSM은 문법의 **독립 접두어**(viable prefix)를 인식하는 DFA라는 것을 알 수 있다. 독립 접두어는 언어에 속하는 문자열의 정규(최우측) 유도에서 문장 형식을 시작할 수 있는 문법 기호로 구성된 모든 문자열로 핸들의 끝부분 이상으로는 확장하지 않는다. 적절한 문법으로부터 LL(1) PDA와 SLR(1) PDA를 구성하는 알고리즘은 2.3.2절과 2.3.3절에 나와 있다.

【2.4.3】 문법과 언어 부류

예 2.55

0^n1^n은 정규 언어가 아니다

2.1.2절에서 살펴봤듯이 어휘 분석기는 임의로 중첩된 구성소를 인식할 수 없다. 이 증명의 핵심은 유한 개의 상태로는 임의의 수의 왼쪽 괄호 기호를 셀 수 없음을 아는 것이다. 예를 들어 언어 0^n1^n를 수용하는 문제를 생각해보자. 이 언어를 수용하는 DFA M이 있다고 가정하자. 또 M이 m개의 상태를 가진다고 가정하자. 이제 M에 m+1개의 0으로 구성되는 문자열을 입력한다고 가정해보자. 비둘기 집의 원리(p개의 비둘기 집에 m개의 물체를 분배할 때 p<m인 경우 어떤 동일한 비둘기 집에는 최소한 두 개의 물체가 있게 된다)에 의해 M은 이 문자열을 어휘 분석하는 동안 어떤 상태 q_i에 두 번 들어가야 한다. 일반성을 유지하면서 M이 j개의 0를 보고 q_i에 들어간 후 다시 k개의 0을 본 후에 q_i에 또 들어간다고 가정하자($j \neq k$). M이 문자열 0^j1^j와 0^k1^k를 수용한다는 것을 알고 있으며 0^j와 0^k를 읽은 후에 M이 정확히 동일한 상태에 있기 때문에 M은 문자열 0^j1^k와 0^k1^j도 수용해야 한다고 연역할 수 있다. 이러한 문자열은 이 언어에 속하지 않기 때문에 모순에 도달하며 M은 존재하지 않는다.

문맥 자유 언어 계열에서는 다양한 구문 분석 알고리즘을 사용해 인식하거나 인식할 수 없는 구성소에 대한 이와 비슷한 정리를 증명할 수 있다. 거의 모든 실제 구문 분석기가 하나의 미리 보기 토큰을 사용하지만 원리상으로는 둘 이상을 사용해 선형 시간에 구문 분석할 수 있는 문법 집합을 확장할 수도 있다. 하향식의 경우 다소 이해하기 쉬운 방식으로 FIRST 집합과 FOLLOW 집합이 토큰 쌍을 포함하게 재정의할 수 있다. 그러나 이렇게 하면 ©(심화학습에 있는) 2.3.4절에서 설명한 즉시 오류 탐지 문제의 좀

더 심각한 버전이 발생한다. 문맥 독립적 FOLLOW 집합을 사용하면 하나 이상의 빈 문자열 생산을 불필요하게 예측한 후까지 구문 오류를 간과할 수 있음을 알아봤다. 문맥 특화 FOLLOW 집합은 이 문제를 해결하지만 하나의 미리 보기 토큰으로 구문 분석할 수 있는 유효한 프로그램의 집합은 변경하지 않는다. k개의 미리 보기 토큰과 스택 상단 기호를 사용해서 예보적으로 구문 분석할 수 있는 모든 문법의 집합으로 LL(k)를 정의하면 k>1인 경우에는 올바르게 구문 분석하기 위해 FOLLOW 집합의 문맥 특화 개념을 사용해야 한다. 문맥 독립적 FOLLOW 집합에 기반한 2.3.2절의 알고리즘은 사실 진정한 LL이라기보다는 SLL(단순 LL, simple LL)이다. k=1인 경우 LL(1)과 SLL(1) 알고리즘은 동일한 집합의 문법을 구문 분석할 수 있다. 그러나 k>1인 경우에는 엄밀하게 LL이 더 강력하다. 상향식 구문 분석기에서 SLR(k), LALR(k), LR(k) 간의 관계는 다소 더 복잡하지만 추가적인 미리 보기 문자는 항상 도움이 된다.

예 2.56

문법 종류 간의 구별

널리 쓰이는 선형 시간 알고리즘이 수용하는 문법 종류 간의 포함 관계는 (심화학습에 있는) 그림 2.33과 같다. LR 종류(접미사 없음)는 G∈LR(k)를 만족시키는 k가 존재하는 모든 문법 G를 포함한다. LL, SLL, SLR, LALR도 유사하게 정의된다.

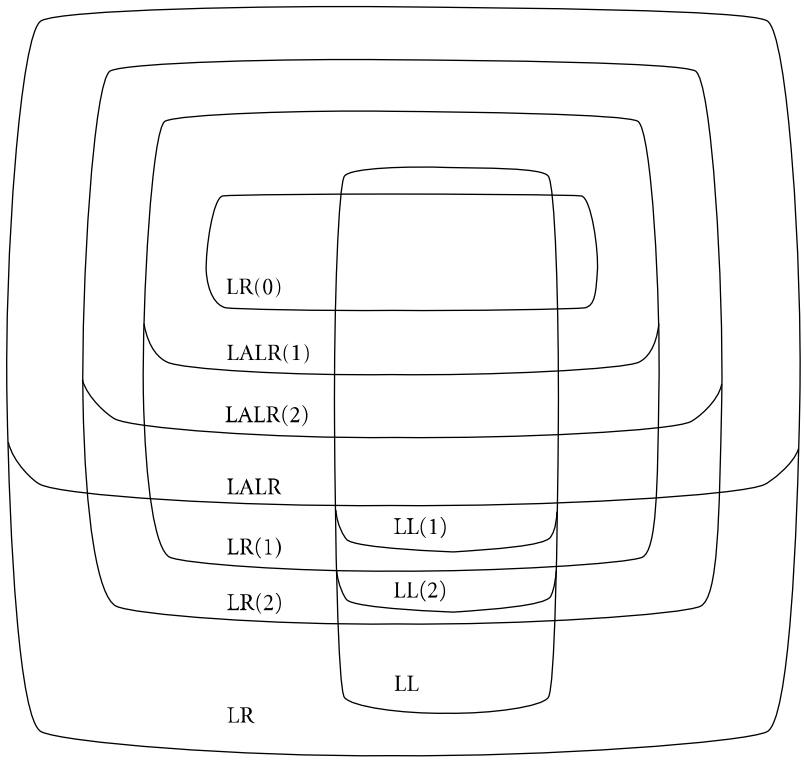

그림 2.33 | 널리 쓰이는 문법 종류 간의 포함 관계. 여기서 보인 포함 관계 외에 k≥2인 경우 SLL(k)는 LL(k)에 포함되지만 그외 모든 것에 대해서는 동일한 관계를 가지며 k≥1인 경우 SLR(k)는 LALR(k)에 포함되지만 그외 모든 것에 대해서는 동일한 관계를 가진다.

이 그림의 모든 부분에서 문법을 찾아볼 수 있다. 예는 ©(심화학습에 있는) 그림 2.34에 나타냈다. 이러한 예들이 해당 문법에 속한다는 증명은 연습문제 2.26에서 다룬다.

LL(2)이지만 SLL은 아닌 문법:
S → a A a | b A b a
A → b | ϵ

SLL(k)이지만 LL(k−1)은 아닌 문법:
S → a^{k-1} b | a^{k}

LR(0)이지만 LL은 아닌 문법:
S → A b
A → A a | a

SLL(1이지만 LALR은 아닌 문법:
S → A a | B b | c C
C → A b | B a
A → D
B → D
D → ϵ

SLL(k)이고 SLR(k)이지만 LR(k −1)은 아닌 문법:
S → A a^{k-1} b | B a^{k-1} c
A → ϵ
B → ϵ

LALR(1)이지만 SLR은 아닌 문법:
S → b A b | A c | a b
A → a

LR(1)이지만 LALR은 아닌 문법:
S → a C a | b C b | a D b | b D a
C → c
D → c

모호하진 않지만 LR은 아닌 문법:
S → a S a | ϵ

그림 2.34 | 그림 2.33의 다양한 영역에 속하는 문법의 예

예 2.57
언어 종류 간의 구별

임의의 문맥 자유 문법 G와 구문 분석 알고리즘 P에 대해 P를 사용해서 G를 구문 분석할 수 있다면 G는 P 문법(예를 들어 LL(1) 문법)이라고 한다. 연장선상에서 임의의 문맥 자유 언어 L에 대해 L을 위한 P 문법(이는 주어진 문법이 아닐 수 있다)이 존재한다면 L은 P 언어라고 한다. 널리 쓰이는 구문 분석 알고리즘이 수용하는 언어 종류 간의 포함

관계는 ©(심화학습에 있는) 그림 2.35와 같다. 이 그림의 어디에서나 언어를 찾아볼 수 있다. 예는 ©(심화학습에 있는) 그림 2.35에 나타냈으며 증명은 연습문제 2.27에서 다룬다.

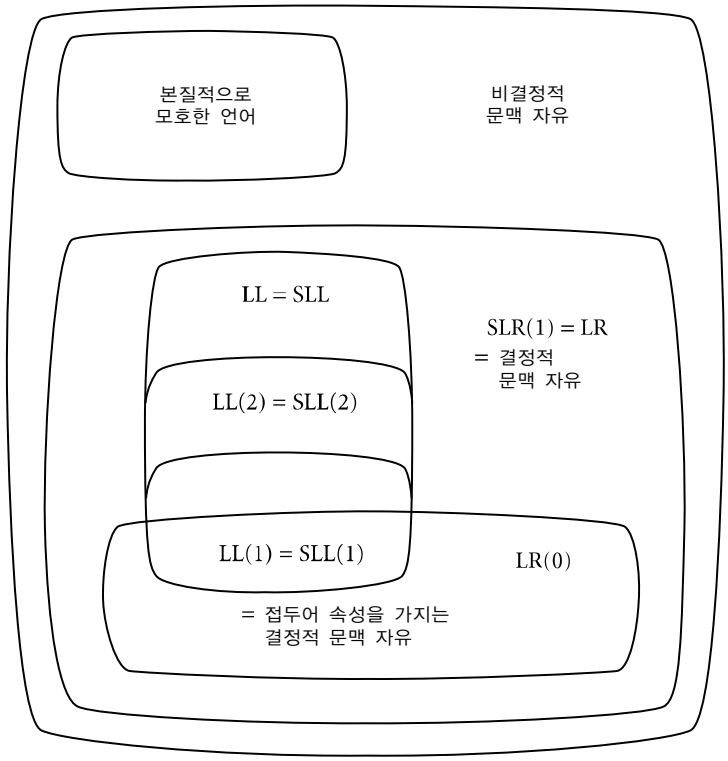

그림 2.35 | 널리 쓰이는 언어 간의 포함 관계

결정적으로 구문 분석할 수 있는 모든 문맥 자유 언어는 SLR(1) 문법을 가진다는 것에 주목하자. 더욱이 결정적으로 구문 분석할 수 있으며 유효한 어떤 문자열도 다른 유효한 문자열을 생성하기 위해 확장될 수 없는(이를 접두어 속성이라고 한다) 언어는 모두 LR(0) 문법을 가진다. 관심의 대상을 파일 끝에 명시적인 `$$` 표시자를 가지는 언어로 제한하면 이러한 언어는 모두 접두어 속성을 가지며 그러므로 LR(0) 문법을 가진다.

언어 종류 간의 관계는 문법 종류 간의 관계만큼 풍부하지 않다. 대부분의 실제 프로그래밍 언어는 널리 쓰이는 구문 분석 알고리즘 중 어떤 것으로도 구문 분석할 수 있다. 물론 이 경우 사용하는 문법이 항상 좋은 것은 아니며 언어가 주어진 종류에 거의(그러나 완전히는 아니게) 속할 때 특수 목적의 "핵(hack)"이 때때로 필요할 수도 있다. 좀 더 강력한 구문 분석 알고리즘(예를 들어 완전 LR)의 주요 장점은 주어진 언어에 대해 좀 더 많은 종류의 문법을 구문 분석할 수 있다는 점이다. 실제로 이러한 융통성은 직관적이고 읽기 쉬우며 의미 동작 루틴의 생성을 용이하게 해주는 문법을 컴파일러 작성자가 쉽게 찾게 해준다.

✓ 확인문제

55. 어휘 분석기의 동작을 나타내는 형식 기계는 무엇인가? 또 구문 분석기의 동작을 나타내는 것은?

56. 실제 어휘 분석기가 형식 기계와 구별되는 세 가지 방법을 기술하라.

57. DFA의 형식 구성 요소는 무엇인가?

58. 주어진 DFA와 동등한 정규식을 만드는 데 사용하는 알고리즘을 간단히 설명하라.

59. NPDA로 구문 분석할 때의 고유한 "빅-O(big-O)" 복잡도는 무엇인가? 이것이 2.3절에서 언급한 $O(n^3)$ 시간보다 안 좋은 이유는 무엇인가?

60. LL(1) PDA에는 얼마나 많은 상태가 존재하는가? SLR(1) PDA의 경우는 어떠한가? 각기 설명하라.

61. CFG의 독립 접두어란 무엇인가?

62. DFA가 임의로 중첩된 구성소를 인식할 수 없다는 증명을 요약하라.

63. LL 구문 분석과 SLL 구문 분석 간의 차이를 설명하라.

64. 모든 LL(1) 문법이 LR(1)인가? 또 LALR(1)인가?

65. 모든 LR 언어가 SLR(1) 문법을 가지는가?

66. 문법 종류 간의 포함 관계가 언어 종류 간의 포함 관계보다 복잡한 이유는 무엇인가?

비결정적 언어:
{an bn c : n ≥ 1} ∪ {an b2n d : n ≥ 1}

본질적으로 모호한 언어:
{ai bj ck : i = j or j = k i, j, k ≥ 1}

LL(k) 문법은 있지만 LL(k-1) 문법은 없는 언어:
{an(b | c | bk d)n : n ≥ 1}

LR(0) 문법은 있지만 LL 문법은 없는 언어:
{an bn : n ≥ 1} ∪ {an cn : n ≥ 1}

그림 2.36 | 그림 2.35의 다양한 영역에 속하는 언어의 예