

상향식 파싱 (Bottom-Up Parsing)

상향식 파싱은 구문 트리를 단말(leaves)에서 시작하여 루트(start symbol) 방향으로 구성하는 방식입니다. 이는 문법 규칙을 거꾸로 적용하여 입력 문자열을 시작 기호로 환원(reduce)하는 과정으로 이해할 수 있습니다. 마치 문법으로부터 생성된 가장 오른쪽 유도(rightmost derivation) 과정을 역추적하는 것과 같습니다.



생성자: Son Tom

상향식 파싱의 특징

장점

대부분의 결정적 문맥 자유 문법(deterministic CFG)을 처리 할 수 있으며, 실제 프로그래밍 언어(C, Java 등)의 파서 생성에 사용됩니다.

하향식 파싱과의 비교

하향식 파싱(Top-down parsing)과 비교했을 때, 상향식 방식은 더 넓은 문맥을 고려할 수 있어 더 많은 문법을 인식할 수 있다는 장점이 있습니다.

기본 연산 (Basic Operations)

상향식 파서는 두 가지 기본 연산을 사용하여 파싱을 수행합니다:



Shift (시프트)

다음 입력 토큰을 파싱 스택의 맨 위로 옮기는(push) 연산입니다. 일반적으로 다음 리듀스를 수행할 수 있을 때까지 입력 토큰을 스택으로 시프트합니다.



Reduce (리듀스)

스택의 맨 위에 있는 부분 문자열 (substring)이 어떤 문법 규칙의 우변(RHS, Right-Hand Side)과 일치할 때, 그 부분 문자열을 해당 규칙의 좌변(LHS, Left-Hand Side) 비단말(nonterminal)로 대체하는 연산입니다. 즉, $A \rightarrow \alpha$ 형태의 규칙에서 스택의 맨 위 α 를 비단말 A 로 환원합니다. 리듀스를 수행하기 전에 항상 스택의 맨 위에 있는 "핸들(Handle)"을 찾습니다.



Accept (엑셉트)

확장된 문법(Augmented Grammar)의 시작 규칙인 $S' \rightarrow S$ 로 리듀스가 완료되고 입력 문자열이 모두 처리되었을 때 파싱이 성공적으로 끝났음을 나타내는 연산입니다.

핸들 (Handle)

핸들은 상향식 파싱에서 중요한 개념입니다. 핸들은 어떤 문법 규칙 $A \rightarrow \alpha$ 의 우변 α 와 일치하는 부분 문자열로서, 이를 해당 규칙의 좌변 A 로 리듀스할 수 있는 후보가 됩니다. 상향식 파서는 일반적으로 입력 토큰을 왼쪽에서 오른쪽으로 스캔하여 핸들을 찾고, 이를 해당 좌변으로 대체하는 과정을 반복합니다. 시프트-리듀스 파서에서는 핸들이 항상 스택의 맨 위(stack top)에 나타나며, 스택의 중간에는 나타나지 않습니다.

핸들 예시

예를 들어, 문법 $E \rightarrow E + n \mid n$ 과 입력 문자열 $n + n$ 에서:

1 첫 번째 핸들

첫 번째 n 은 규칙 $E \rightarrow n$ 의 우변에 해당하므로 핸들입니다. 파싱 스택에서 n 은 E 로 리듀스됩니다.

2 두 번째 핸들

리듀스 후 스택에 $E + n$ 이 있을 때, $E + n$ 은 규칙 $E \rightarrow E + n$ 의 우변에 해당하므로 핸들입니다. 파싱 스택에서 $E + n$ 은 E 로 리듀스됩니다.



시프트-리듀스 파싱 (Shift-Reduce Parsing)

상향식 파싱은 시프트-리듀스 파싱이라고도 불립니다. 시프트-리듀스 파서는 문법 기호(단말 또는 비단말) 스택을 사용하여 동작합니다. 입력 토큰은 스택으로 시프트 되며, 스택의 맨 위에 핸들이 나타나면 해당 핸들을 좌변으로 리듀스합니다. 파싱 동작은 일반적으로 파싱 테이블에 의해 결정됩니다.

충돌 (Conflicts)

결정적(deterministic)인 파싱 과정에서는 각 단계에서 수행할 동작(시프트 또는 리듀스, 혹은 어떤 규칙으로 리듀스할지)이 명확해야 합니다. 그러나 주어진 상태에서 유효한 파싱을 이끌 수 있는 동작이 하나 이상일 경우 충돌이 발생합니다.

시프트-리듀스 충돌 (Shift-reduce conflicts)

특정 상태에서 시프트를 수행하는 것도 가능하고 리듀스를 수행하는 것도 가능한 경우 발생합니다. LR(0) 파싱에서는 상태가 리듀스 항목($A \rightarrow \alpha .$)과 시프트 항목($A \rightarrow \alpha . B \beta$ 또는 $A \rightarrow \alpha . b \beta$ 형태의 항목)을 동시에 포함할 때 발생합니다.

리듀스-리듀스 충돌 (Reduce-reduce conflicts)

특정 상태에서 두 개 이상의 서로 다른 규칙으로 리듀스를 수행하는 것이 모두 가능한 경우 발생합니다. LR(0) 파싱에서는 상태가 두 개 이상의 리듀스 항목($A \rightarrow \alpha .$, $B \rightarrow \beta .$)을 동시에 포함할 때 발생합니다.

이러한 충돌은 파서가 어느 동작을 선택해야 할지 결정할 수 없게 만듭니다. 따라서 충돌이 없는 문법(unambiguous grammar)이거나, 충돌을 해결할 수 있는 파싱 알고리즘이 필요합니다.

전처리: 확장 문법 (Augmented Grammar)



상향식 파싱을 위해 문법에 전처리 과정이 필요할 수 있습니다. 문맥 자유 문법 $G = (VN, VT, P, S)$ 가 주어지면, 새로운 시작 기호 S' 와 규칙 $S' \rightarrow S$ 를 추가하여 확장 문법 $G' = (V'N, VT, P', S')$ 를 구성합니다. 여기서 $V'N = VN \cup \{S'\}$ 이고 $P' = P \cup \{S' \rightarrow S\}$ 입니다. 이 확장 문법은 원본 문법과 동일한 언어 $L(G) = L(G')$ 를 인식 합니다. 새로운 시작 기호 S' 를 추가하는 것은 파싱이 성공적으로 완료되었음을 나타내는 유일한 리듀스 ($S' \rightarrow S$)를 정의하기 위함입니다.

LR 파싱 (LR Parsing)

LR 파싱은 문맥 자유 문법이 생성하는 언어 $L(G)$ 에 속하는 터미널 문자열 w 가 주어졌을 때, w 가 그 언어에 속하는지 확인하고, 속한다면 w 의 가장 오른쪽 유도 과정을 역으로 구성하는 파싱 방법입니다. 모든 문맥 자유 문법에 대해 가능한 것은 아니지만, LR 파싱은 LL(k) 파싱이 인식하는 문법의 상위 집합을 파싱할 수 있습니다. 또한 우수한 구문 오류 감지 능력을 가지고 있습니다.

LR(k)의 의미

LR(k)에서 각 문자는 다음을 의미합니다:



L

입력 문자열을 왼쪽에서 오른쪽으로 스캔합니다 (Left-to-right scan).



R

가장 오른쪽 유도를 역으로 구성합니다 (Rightmost derivation in reverse).



k

파싱 결정을 내릴 때 사용되는 선행 입력 기호(lookahead token)의 개수를 나타냅니다. k=0이면 선행 기호를 사용하지 않고, k=1이면 다음 하나의 선행 기호를 사용합니다.

LR 파서의 구조

LR 파서의 구조는 스택과 파싱 테이블로 구성됩니다.

스택 (Stack)

파서가 처리한 문법 기호와 상태를 저장합니다. 스택의 각 상태는 그 아래 스택에 포함된 정보를 요약합니다. 스택의 맨 위에 있는 상태와 현재 입력 기호(lookahead symbol)를 기반으로 파싱 동작(action)을 결정합니다.

파싱 테이블 (Parsing table)

상태와 현재 입력 기호(또는 비단말)를 인덱스로 하여 다음 동작(action) 또는 이동할 상태(goto)를 알려주는 테이블입니다. 파싱 테이블은 액션 테이블과 고투 테이블로 구성됩니다. 이 테이블은 LR(0) 항목을 사용하여 DFA(결정적 유한 오토마타)를 구축하여 생성할 수 있습니다. 각 상태는 지금까지 무엇을 보았고 앞으로 무엇을 볼 것으로 예상하는지를 요약합니다.

파싱 동작 결정

파싱 동작은 스택 기반 결정과 입력 선행 기호 결정을 모두 사용합니다. 스택의 내용과 다음 입력 심볼을 이용하여 다음 동작을 결정합니다. 예를 들어, 예시에서 스택에 \$와 상태 0이 있고 입력에 n이 있을 때 시프트 동작을 수행합니다. 스택에 \$와 E 상태 1이 있고 입력에 \$가 있을 때 억셉트 동작을 수행합니다.

LR(0) 파싱

LR(0) 파싱은 가장 기본적인 LR 파싱 형태입니다. 선행 입력 기호(lookahead)를 고려하지 않고 파싱 테이블을 구축하고 파싱 결정을 내립니다. DFA의 상태는 스택의 상태만으로 결정되며, 상태를 구축할 때 선행 입력 기호는 무시됩니다. 이것이 LR(0)이라고 불리는 이유입니다.



LR(0) 항목 (LR(0) Item)

문법 규칙의 우변에 점(.)으로 표시된 구별된 위치를 가진 생산 규칙입니다. 점의 위치는 우변의 어느 정도까지 인식되었는지를 나타냅니다.

$A \rightarrow \alpha . \beta$

α 까지 인식했고 앞으로 β 를 볼 것으로 예상하는 중간 단계 항목입니다.

$A \rightarrow . \alpha$

α 전체를 아직 인식하지 못한 초기 항목입니다.

$A \rightarrow \alpha .$

α 전체를 인식했고 이 규칙($A \rightarrow \alpha$)으로 리듀스할 준비가 된 완료 항목(reduce item)입니다.

LR(0) 항목의 DFA

LR(0) 항목들은 DFA의 상태로 사용될 수 있습니다. 이 DFA는 파싱 스택과 시프트-리듀스 파서의 진행 상황에 대한 정보를 유지합니다.

시작 상태

확장 문법의 시작 규칙 $S' \rightarrow .S$ 항목을 포함하는 상태입니다.

전이

어떤 상태에서 $A \rightarrow X . YZ$ 형태의 항목이 있고 Y 가 다음 기호일 때, 점을 Y 뒤로 옮겨 $A \rightarrow XY . Z$ 항목을 포함하는 새로운 상태로 전이합니다. 이 새로운 상태를 만들 때는 해당 항목의 동치 집합 (equivalent LR(0) items) 전체를 포함하도록 합니다.

최종 상태

$A \rightarrow \alpha .$ 형태의 항목(점이 우변의 가장 오른쪽에 있는 항목)을 포함하는 상태는 최종 상태(final state)이며, 이는 리듀스를 의미합니다.

동치 항목은 $S \rightarrow X.YZ$ 와 $Y \rightarrow .WQ$ 처럼, 점 뒤에 비단말이 오고 그 비단말을 시작 기호로 하는 다른 규칙의 초기 항목이 있을 때 발생합니다.

LR(0) 파싱 알고리즘

파싱 스택에 종료 마커 \$와 초기 상태 0을 푸시합니다. 현재 DFA 상태에 따라 다음과 같은 동작을 선택합니다:



Shift

현재 상태가 시프트 항목만 포함하는 경우 (예: $A \rightarrow \alpha . B \beta$), 현재 입력 토큰을 스택에 시프트하고 스택에 새로운 상태를 푸시합니다.



Reduce

현재 상태가 하나의 리듀스 항목만 포함하는 경우 (예: $A \rightarrow \alpha .$), 규칙 $A \rightarrow \alpha$ 를 사용하여 리듀스를 수행합니다. $S' \rightarrow S$ 규칙으로의 리듀스는 억셉트를 의미합니다 (입력이 비어있을 때). 리듀스를 수행할 때는 스택에서 α 에 해당하는 기호와 관련 상태들을 팝(pop)하고, 이전 상태에서 A 기호에 대한 고투(goto) 테이블을 찾아 새로운 상태를 스택에 푸시합니다.



Conflict

시프트 항목과 리듀스 항목이 같거나, 두 개 이상의 리듀스 항목이 같이 있는 상태에서는 충돌이 발생하며 LR(0) 파싱이 불가능합니다.

LR(0) 파싱은 파싱 결정을 내릴 때 선행 입력 토큰을 고려하지 않습니다.

LR(0) 파싱 예시

예를 들어, 문법 $G' = (\{A, S'\}, \{(,), a\}, P, S')$ 와 입력 ((a))에 대한 LR(0) 파싱 과정은 다음과 같이 스택과 입력, 동작을 추적하며 진행됩니다:

스택	입력	동작
\$ 0	((a)) \$	shift
\$ 0 (3	(a)) \$	shift
\$ 0 (3 (3	a)) \$	shift
\$ 0 (3 (3 a 2)) \$	Reduce A → a
\$ 0 (3 (3 A 4)) \$	shift

이후 과정도 계속 진행됩니다.

LR(0) 파싱의 한계



LR(0) 파서는 충돌이 자주 발생하여 매우 제한적인 문법만 처리할 수 있다는 단점이 있습니다. 자료에서 예시로 제시된 몇몇 문법은 LR(0)에서 시프트-리द스 충돌을 일으킵니다.

SLR(1) 파싱 (SLR(1) Parsing)

SLR(1) (Simple LR(1)) 파싱은 LR(0) 파싱의 확장입니다. LR(0) 항목으로 구성된 동일한 DFA를 사용하지만, 파싱 동작을 결정할 때 다음 입력 토큰(lookahead, $k=1$)을 추가적으로 고려합니다. 특히 리듀스 동작을 결정할 때 선행 입력 기호를 사용합니다.

SLR(1) 파싱 알고리즘

LR(0) 파싱과 유사하지만, 리듀스 결정을 내릴 때 선행 입력 토큰을 고려합니다.



Shift

시프트 항목($A \rightarrow \alpha . B \beta$)이 있고 다음 입력 토큰이 B 이면 시프트를 수행합니다.



Reduce

리듀스 항목($A \rightarrow \alpha .$)이 있고 다음 입력 토큰이 해당 규칙 좌변 비단말 A 의 Follow 집합(Follow(A))에 포함되어 있으면 리듀스를 수행합니다. Follow 집합은 문법에서 비단말 A 뒤에 올 수 있는 단말들의 집합입니다. $S' \rightarrow S$ 리듀스는 입력이 비어있을 때 억셉트입니다.

SLR(1) 파싱 테이블

SLR(1) 파싱 테이블은 상태와 입력 기호(\$ 포함)를 인덱스로 가지며, 각 칸에는 시프트(s), 리द스(r), 억셉트(acc), 에러 중 하나의 동작이 표기됩니다. 하나의 상태에서 시프트와 리�스가 모두 가능할 수 있지만, 선행 입력 기호에 따라 결정되기 때문에 충돌이 해결될 수 있습니다.

State	n	+	\$	E
0	s2			1
1		s3	acc	
2		r(2)	r(2)	
3	s4			
4		r(1)	r(1)	

(규칙 번호: (1) $E \rightarrow E + n$, (2) $E \rightarrow n$)

SLR(1) 파싱 예시

문법 $G' = (\{E, E'\}, \{+, n\}, P, E')$ 와 입력 $n + n + n$ 에 대한 SLR(1) 파싱 과정이 자료에 제시되어 있습니다. 이 문법에서 $\text{Follow}(E') = \{\$\}$, $\text{Follow}(E) = \{\$, +\}$ 입니다.

스택	입력	동작
\$ 0	$n + n + n \$$	shift
\$ 0 n 2	$+ n + n \$$	reduce $E \rightarrow n$
\$ 0 E 1	$+ n + n \$$	shift

파싱 과정은 계속 진행됩니다.

SLR(1) 파싱의 특징

SLR(1) 파싱은 LR(0)보다 강력하며 대부분의 실용적인 언어 구조를 처리할 수 있습니다. 그러나 여전히 일부 문법에서는 충돌이 발생하여 파싱이 불가능한 경우가 있습니다. SLR(1) 파싱에서 충돌이 발생하는 상황은 자료에 예시로 제시된 문법에서 확인할 수 있습니다. 예를 들어, 문법 $S \rightarrow (S) S \mid \epsilon$ 는 SLR(1)에서 충돌이 발생합니다.

General LR(1) 파싱 (Canonical LR(1) Parsing)

LR(0)나 SLR(1)의 한계를 극복하기 위한 더 강력한 파싱 기법으로 LR(1) 파싱(Canonical LR(1))과 LALR(1) 파싱이 있습니다.
General LR(1) 파싱은 LR 파싱 알고리즘 중 가장 강력합니다.

LR(1) 항목 (LR(1) Item)

LR(0) 항목에 선행 입력 토큰(lookahead token) 하나가 추가된 쌍입니다. $[A \rightarrow \alpha . \beta, a]$ 형태로 표시되며, 여기서 $A \rightarrow \alpha \beta$ 는 문법 규칙이고 a 는 선행 입력 토큰입니다. 이 항목은 스택의 맨 위에 α 가 있고 다음 입력 토큰이 a 일 때, 만약 β 가 문법적으로 ϵ 으로 유도된다면 $A \rightarrow \alpha \beta$ 규칙을 사용하여 리듀스할 수 있음을 나타냅니다.

LR(1) 항목의 DFA

LR(1) 항목들은 DFA의 상태로 사용됩니다. 이 DFA는 LR(0) 항목의 DFA를 구축하는 것과 유사하지만, 선행 입력 기호를 포함하여 상태를 구축하고 전이를 결정합니다.

시작 상태

시작 상태는 $[S' \rightarrow . S, \$]$ 항목을 포함하는 상태입니다.

전이

$[A \rightarrow \alpha . B C \beta, b]$ 항목이 있는 상태에서 비단말 B에 대한 전이는 $[A \rightarrow \alpha B . C \beta, b]$ 항목을 포함하는 새로운 상태로 이동합니다.

동치 항목

$[A \rightarrow \alpha . B \beta, b]$ 항목이 있는 상태에서 점 뒤에 비단말 B가 오고 $B \rightarrow \gamma$ 라는 규칙이 있다면, 이 상태는 $[B \rightarrow . \gamma, a]$ 형태의 모든 항목과 동치입니다. 여기서 선행 입력 a는 FIRST(βb) 집합에 속하는 모든 단말입니다. 이는 β 뒤에 b가 오거나, β 가 ϵ 으로 유도되고 b가 올 때 B 뒤에 올 수 있는 기호들입니다.

리듀스

$[A \rightarrow \beta . , b]$ 형태의 항목(점이 우변의 가장 오른쪽에 있는 항목)을 포함하고, 다음 입력 토큰이 b일 때 리듀스를 수행할 수 있습니다.

LR(0)와 LR(1) DFA 비교



LR(0) 항목의 DFA와 LR(1) 항목의 DFA는 그 구조가 유사하지만, LR(1) DFA는 각 항목에 선행 입력 정보가 포함되어 있기 때문에 상태의 개수가 훨씬 많아질 수 있습니다. 예시 문법 $A \rightarrow (A) \mid a$ 에 대한 LR(0) DFA와 LR(1) DFA는 형태는 비슷하지만, 각 상태에 포함된 항목의 선행 입력 기호에서 차이를 보입니다.

General LR(1) 파싱 알고리즘

SLR(1) 파싱과 유사하지만, 파싱 결정을 내릴 때 LR(1) 항목에 명시된 선행 입력 토큰을 사용합니다.



Shift

[$A \rightarrow \alpha . X \beta, b$] 항목이 있고 다음 입력 토큰이 X 이면
(X 는 단말) 시프트를 수행합니다.



Reduce

[$A \rightarrow \alpha . , b$] 항목이 있고 다음 입력 토큰이 b 이면 규칙
 $A \rightarrow \alpha$ 를 사용하여 리듀스를 수행합니다. $S' \rightarrow S$ 리듀스
는 입력이 비어있을 때 억셉트입니다.

General LR(1) 파싱 예시

문법 $G' = (\{A, S'\}, \{(,), a\}, P, S')$ 와 입력 ((a))에 대한 General LR(1) 파싱 과정이 자료에 제시되어 있습니다. 규칙 번호는 (1) $A \rightarrow (A)$, (2) $A \rightarrow a$ 입니다.

State	(a)	\$	A
0	s2	s3			1
1				acc	
2	s5	s6			4
3			r(2)		
4			s7		

테이블은 계속 이어집니다.

General LR(1) 파싱의 특징

LR(1) 파싱은 SLR(1)보다 강력하지만, DFA의 상태 수가 많아 파싱 테이블 크기가 매우 커지고 복잡하다는 단점이 있습니다.

LALR(1) 파싱 (LALR(1) Parsing)

LALR(1) (Lookahead LR(1)) 파싱은 General LR(1) 파싱을 변형한 것입니다. General LR(1)의 DFA에서 LR(0) 항목 부분은 같고 선행 입력 기호만 다른 상태들을 병합(merge)하여 상태 수를 줄입니다. 이를 통해 General LR(1)의 대부분의 장점을 유지하면서 SLR(1)의 효율성(테이블 크기)을 확보합니다.

LR 파싱의 계층

LR 파싱 기법들의 강력함(처리할 수 있는 문법의 범위) 순서는 다음과 같습니다:

LR(0) < SLR(1) < LALR(1) < General LR(1)

LR(0)

선행 기호 없이 스택 상태(LR(0) 항목) 기반으로 DFA 구축 및 동작 결정. 매우 제한적.



SLR(1)

LR(0) DFA 사용 + 리듀스 시 선행 기호(Follow 집합) 고려. LR(0)보다 강력하지만 여전히 충돌 발생 가능.

General LR(1)

스택 상태와 선행 기호(LR(1) 항목) 모두를 포함하여 DFA 구축. 가장 강력하지만 복잡하고 무거움.

LL 파싱 vs LR 파싱

하향식 파싱 방식인 LL 파서와 상향식 파싱 방식인 LR 파서는 다음과 같은 차이가 있습니다:

처리 가능한 문법 범위

LR 파서가 LL 파서보다 더 많은 문맥 자유 문법을 인식할 수 있습니다. 상향식 방식은 입력 문자열 전체를 읽으면서 어떤 생산 규칙의 결과였는지 역으로 추적하기 때문에 더 넓은 문맥을 고려할 수 있습니다. 반면, 하향식 파서는 선행 기호를 토대로 규칙을 미리 예측하며, 좌측 재귀(Left recursion)나 좌측 팩토링(Left factoring)과 같은 전처리가 필요하여 제약이 많습니다.

구현 및 사용

LR 파서는 파싱 테이블과 스택을 기반으로 동작하며, 대부분의 결정적 문법을 처리할 수 있어 실제 프로그래밍 언어의 파서 생성에 널리 사용됩니다.

요약

상향식 파싱은 입력 문자열을 시작 기호로 환원하며 구문 트리를 단말에서 루트로 구성하는 방식입니다. 시프트와 리듀스 연산을 사용하며, 핸들 개념이 중요합니다. 파싱 과정에서 시프트-리듀스 또는 리듀스-리듀스 충돌이 발생할 수 있으며, 이를 해결하기 위해 다양한 LR 파싱 기법이 개발되었습니다.

요약 (계속)

LR 파싱은 스택과 파싱 테이블을 사용하며, 파싱 테이블은 LR 항목으로 구성된 DFA를 통해 생성됩니다. LR(0)은 선행 기호를 고려하지 않아 제약이 많고, SLR(1)은 LR(0) DFA에 선행 기호(Follow 집합)를 추가로 활용하며, General LR(1)은 LR(1) 항목(LR(0) 항목 + 선행 기호) 기반의 DFA를 구축하여 가장 강력한 파싱 성능을 제공합니다. LALR(1)은 LR(1)의 효율적인 변형입니다. LR 파싱은 LL 파싱보다 강력하며 실제 컴파일러에서 널리 사용됩니다.

상향식 파싱의 기본 - 쌓고(Shift), 줄인다(Reduce)!

상향식 파싱의 기본 개념은 입력 토큰을 하나씩 스택에 쌓고(Shift), 문법 규칙에 맞는 패턴이 스택 상단에 나타나면 그것을 해당 규칙의 좌변 심볼로 줄이는(Reduce) 과정입니다.

레고 조립 비유

Shift 연산

개별 블록(토큰)들을 작업대(스택)에 하나씩 쌓는 과정입니다. 마치 레고 블록을 하나씩 작업대에 올려놓는 것과 같습니다.

Handle

설명서의 한 단계에 해당하는 블록 묶음 (예: 창문 부품)을 의미합니다. 스택 상단에 있는 이 패턴은 하나의 규칙으로 줄일 수 있는 후보입니다.

Reduce 연산

작업대의 블록 묶음(Handle)을 하나의 완성된 부품(Non-terminal)으로 조립 (Reduce)하는 과정입니다. 여러 개의 레고 블록을 조립하여 하나의 의미 있는 부품으로 만드는 것과 같습니다.

간단한 파싱 과정 애니메이션

문법 $E \rightarrow E + n \mid n$ 과 입력 문자열 $n + n$ 에 대한 파싱 과정을 살펴봅시다:

스택	입력	동작	설명
\$	$n + n \$$	shift	첫 번째 토큰 n 을 스택으로 시프트합니다.
\$ n	$+ n \$$	reduce $E \rightarrow n$	스택 맨 위 n 은 $E \rightarrow n$ 규칙에 맞는 핸들이므로 E 로 리듀스합니다.
\$ E	$+ n \$$	shift	다음 토큰 $+$ 를 스택으로 시프트합니다.
\$ E +	$n \$$	shift	다음 토큰 n 을 스택으로 시프트합니다.
\$ E + n	\$	reduce $E \rightarrow E + n$	스택 맨 위 $E + n$ 은 $E \rightarrow E + n$ 규칙에 맞는 핸들이므로 E 로 리듀스합니다.
\$ E	\$	accept	입력이 모두 소비되고 스택에 시작 기호만 남았으므로 파싱 성공!

파서의 나침반, LR(0) 파싱과 DFA

LR(0) 파싱에서는 문법 규칙에 점(.)을 추가한 LR(0) 항목을 사용하여 DFA를 구축합니다. 이 DFA는 파서가 현재 어떤 상태에 있는지, 그리고 다음에 어떤 동작을 수행해야 하는지를 결정하는 나침반 역할을 합니다.

DFA 상태 생성 과정 시연

문법 $S' \rightarrow S, S \rightarrow (S)S \mid \epsilon$ 에 대한 LR(0) DFA 생성 과정을 단계별로 살펴봅시다:

1 상태 0 생성

시작 규칙 $S' \rightarrow .S$ 를 놓고, 클로저(Closure) 개념을 적용합니다. "책갈피 뒤에 비단말 S 가 왔네요. 그럼 S 로 시작하는 모든 규칙($(.S)S, .\epsilon$)도 함께 고려해야 합니다. 이것이 상태 0의 완전한 모습입니다."

2 상태 전이

상태 0에서 각 기호($S, ()$)를 읽었을 때 책갈피를 옮기고(GOTO), 그 결과로 만들어진 새 아이템의 클로저를 계산하여 새로운 상태(상태 1, 상태 2)가 탄생합니다.

미니 실습

간단한 문법 $A \rightarrow aA \mid b$ 를 제시하고, 학습자가 직접 상태 0과 상태 1을 유도해봅시다.

상태 0

$A' \rightarrow .A$ (시작 규칙)

$A \rightarrow .aA$ (A로 시작하는 규칙)

$A \rightarrow .b$ (A로 시작하는 규칙)

상태 1(A에 대한 전이)

$A' \rightarrow A.$ (A를 읽은 후)

상태 2(a에 대한 전이)

$A \rightarrow a.A$ (a를 읽은 후)

$A \rightarrow .aA$ (A로 시작하는 규칙)

$A \rightarrow .b$ (A로 시작하는 규칙)

충돌 해결사 등장! SLR(1) 파싱

LR(0) 파싱에서는 충돌 문제가 자주 발생합니다. 특히 한 상태에서 시프트와 리द스 가 모두 가능하거나, 여러 규칙으로 리द스가 가능할 때 파서는 어떤 동작을 선택해야 할지 결정할 수 없습니다. SLR(1)은 이러한 충돌 문제를 해결하기 위해 등장했습니다.

충돌 상황 재현

문법 $E' \rightarrow E, E \rightarrow E+n \mid n$ 의 DFA 상태 1을 살펴봅시다:

상태 1의 항목들

$E' \rightarrow E.$ (리듀스 항목)

$E \rightarrow E.+n$ (시프트 항목)

문제 상황

이 상태에는 리듀스 항목($E' \rightarrow E.$)과 시프트 항목($E \rightarrow E.+n$)이 공존합니다. 파서는 지금 리듀스를 해야 할까요, 시프트를 해야 할까요? LR(0)는 다음 입력을 보지 않으니 결정할 수 없습니다!

Follow 집합 소개

Follow(A)는 "비단말 A의 뒤를 따라올 수 있는 터미널들의 목록, 즉 '컨닝 페이퍼'"라고 생각할 수 있습니다. 이 정보를 활용하면 리듀스 결정을 더 정확하게 내릴 수 있습니다.



Follow(E')

{\$} - E' 는 시작 기호이므로 입력의 끝(\$)만 따라올 수 있습니다.



Follow(E)

{+, \$} - E 뒤에는 +가 올 수 있고($E \rightarrow E+n$ 규칙에서), E '의 일부로서 \$도 올 수 있습니다.

해결 과정 시각화

SLR(1)은 리듀스를 할지 말지 망설여질 때, 이 컨닝 페이퍼(Follow 집합)를 봅니다. '만약 다음 입력 토큰이 컨닝 페이퍼(Follow(E'))에 있다면 리듀스를 승인!' 하는 방식입니다.

상태	다음 입력	결정	이유
상태 1 ($E' \rightarrow E.$ 및 $E \rightarrow E.+n$)	+	시프트	+는 Follow(E')에 없으므로 리듀스 $E' \rightarrow E$ 를 하지 않고, $E \rightarrow E+n$ 규칙에 따라 시프트를 선택
상태 1 ($E' \rightarrow E.$ 및 $E \rightarrow E.+n$)	\$	리듀스	\$는 Follow(E')에 있으므로 리듀스 $E' \rightarrow E$ 를 수행

더 강력한 눈, LR(1) 파싱

SLR(1)도 모든 충돌을 해결할 수 있는 것은 아닙니다. 특히 리듀스-리듀스 충돌이 발생하는 경우, Follow 집합만으로는 충분하지 않을 수 있습니다. 이를 해결하기 위해 더 강력한 LR(1) 파싱이 등장했습니다.

LR(1) Item 소개

[$A \rightarrow \alpha\beta, a$] 아이템은 "책갈피(.)뿐만 아니라, 이 규칙을 적용할 수 있는 특정 '상황(lookahead a)'까지 함께 기록하는 정밀한 메모"라고 할 수 있습니다. 이는 LR(0) 항목에 선행 입력 토큰(lookahead token) 하나가 추가된 형태입니다.



LR(0) 항목

$A \rightarrow \alpha\beta$ - 단순히 어디까지 파싱했는지만 표시



LR(1) 항목

$[A \rightarrow \alpha\beta, a]$ - 파싱 위치와 함께, 이 규칙을 적용할 수 있는 다음 입력 토큰 a도 명시

SLR(1)의 한계와 LR(1)의 해결책

리듀스-리듀스 충돌이 발생하는 예제를 살펴봅시다:

SLR(1)의 문제점

상태 2에서 $S \rightarrow id.$ 와 $V \rightarrow id.$ 규칙이 모두 리듀스를 원합니다. 다음 입력
이 $:=$ 일 때, $\text{Follow}(S)$ 와 $\text{Follow}(V)$ 모두 $:=$ 을 포함할 수 있어 SLR(1)은 여전히 혼란스럽습니다.



LR(1)의 해결책

LR(1)은 애초에 상태를 만들 때부터 lookahead를 고려합니다. 상태 2는 $[S \rightarrow id., \$]$ 와 $[V \rightarrow id., :=]$ 로 나뉘어 만들어집니다. 따라서 다음 입력이 $\$$ 이면 첫 번째 규칙으로, $:=$ 이면 두 번째 규칙으로 리듀스하면 되니 전혀 헷갈리지 않습니다!

최종 점검: 파싱 테이블과 실전 트레이싱

지금까지 배운 내용을 종합하여, 파싱 테이블을 구축하고 실제 입력 문자열에 대한 파싱 과정을 추적해봅시다.

테이블 구축 가이드

DFA와 Follow/lookahead 정보를 바탕으로 파싱 테이블의 Action 부분(shift, reduce)과 Goto 부분을 채우는 규칙을 명확한 순서로 제시합니다:

1

Action 테이블 구축

상태 i에서 단말 a에 대한 액션 결정:

- 상태 i에 $[A \rightarrow \alpha.a\beta, b]$ 형태의 항목이 있으면, a에 대해 시프트 j를 설정 (j는 $GOTO(i,a)$ 의 결과 상태)
- 상태 i에 $[A \rightarrow \alpha., a]$ 형태의 항목이 있으면, a에 대해 리듀스 $A \rightarrow \alpha$ 를 설정
- 상태 i에 $[S' \rightarrow S., \$]$ 항목이 있으면, \$에 대해 양셉트를 설정

2

Goto 테이블 구축

상태 i에서 비단말 A에 대한 고투 결정:

- $GOTO(i,A) = j$ 이면, 고투 테이블의 $[i,A]$ 위치에 j를 설정