LR(k) 서브 스트링 인식과 완성

김상헌[U] 박용관 유재우
숭실대학교 컴퓨터학과
{shkim, psiver}@ss.ssu.ac.kr, cwyoo@comp.ssu.ac.kr

LR(K) Substring Recognition and Completion.

Sang-Heon Kim[U] Yong-Kwan Park Chae-Woo Yoo
Dept. of Computing, Soongsil University

요 약

편집 환경에서 입력되는 구문은 완전한 문장으로 입력되기보다는 문장의 일부가 부분적으로 입력되면서 점진적으로 프로그램을 완성하게 된다. 본 논문에서는 부분적인 문장의 입력을 분석하여 문장의 부족한 부분을 예측하여 서브 스트링에 대한 파스트리를 완성할 수 있는 방법을 제시한다.

1. 서브 스트링

파싱 과정에서 파서는 구문 입력으로부터 트리를 생성해야 한다. 트리가 하향식으로 구성되면 파서는 입력 문장의 빠진 부분에 대하여 정확히 위치 고정자(placeholder)를 삽입할 수 있다. LR 파서의 경우 트리는 상향식으로 구성되며 트리의 노드는 축약(reduction)이 일어날 때만 구성되므로 이러한 작업이 상당히 어렵다. 파싱 과정에서 LR 스택의 각 비단말기호는 해당 부분의 부분적인 트리에 대한 포인터를 가진다. 따라서 입력이 불완전할 경우, 파싱 스택은 완전히 축약되지 못하여 여러 개의 비단말기호를 가진 문장형을 이루게 되며, 이들 각각에는 트리의 부분들이 연결되어 있다. 그러나, 이들 부분들이 서로 연결되어 있지 못하므로 표준의 LR 파서에서는 더 이상의 축약이 일어나지 못한다. 따라서 빠진 부분에 대한 서브 트리의 루트가 생성될 수 없으므로 이미 구성된 트리의 부분들은 서로 연결될 수가 없다. 예를 들어

if i > j then x =

라는 불완전한 입력이 처리될 때 입력의 끝 부분에서 입력의 끝을 나타내는 토큰 "$"가 입력되면 불완전한 문장이므로 구문의 오류가 보고된다. 그러나, 이러한 입력에 대해 파서는 입력된 정보를 이용한 [그림 1] 과 같은 부분적인 정보를 포함하는 파스 트리를 구성하는 것이 바람직하다.

편집기의 입력은 프로그램의 편집 위치에 따라 문장의 왼쪽 혹은 오른쪽이 부분적으로만 존재할 수 있다. 프로그램이 부분적으로 입력되는 경우 이러한 부분적인 입력 문장에 대해 적당한 파스 트리를 구성할 수 있어야 한다. 이런 입력을 처리할 수 있다면 편집기에서 여러 곳에서 부분적인 변경이 발생하고, 이 변경이 완전한 문장을 완성하지 못하더라고 부분적인 구문에 대한 파스 트리의 구성이 가능할 것이다.

프로그램의 입력이 끝나게 되면 부분적인 구문에 대해서는 구문의 오류를 보고하기보다는 문맥이 완성되지 않았음을 나타내는 상태정보를 제공하는 것이 바람직하다.

서브 스트링을 인식하고 불완전한 문장을 완성하기 위해서는 입력된 서브 스트링이 유효한지를 확인해야 하며, 유효한 경우에 한해 부족한 부분을 채우도록 한다. 이러한 과정을 수행하기 위하여 본 논문에서는 서브 스트링의 인식과정과 문장의 완성 과정을 수행하는 알고리즘을 구성하였다.

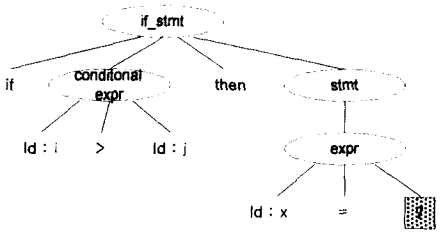

그림 1. 서브 스트링에 대한 파스 트리

2. 서브 스트링의 인식과 완성

LR 파싱표를 이용하여 서브스트링을 인식하는 방법은 기본적으로 모든 가능한 prefix에 대하여 파서의 동작 상태를 따라 인식 가능한 형태의 파스 트리를 구하는 것이다.

서브 스트링의 인식을 위해서는 스택의 움직임을 따르지만 푸시다운 오토마타가 파스 경로에 따라 다양한 형태의 동작을 가지도록 하여야 한다. 따라서 스택의 여러 가지 동작 과정을 모두 표현할 수 있는 방법이 필요하다. 이는 Graph Structured Stack을 이용하여 해결할 수 있다. 서브 스트링의 파싱이 성공적으로 끝나면 [그림 1]와 같이 불완전 노드를 포함하는 파스 트리가 구성된다.

각 트리의 노드는 LR 머신의 상태로 표시되어 있다. 루트노드에서 단말노드까지의 모든 경로는 입력된 부분 문맥의 가능한 해석으로 LR 파서의 스택의 top 부분에 해당하며, 이 경우 경로의 root에 해당되는 노드의 상태는 스택의 top에 해당하게 된다. 부분 문맥의 인식을 위한 알고리즘은 다음과 같다.

[알고리즘 1] 서브 스트링 인식

서브 스트링을 인식하기 위해 LR파서의 동작을 reduce 항목에 대해 다음과 같이 수정한다.

reduce A := $\alpha\beta$ 일 때

case $|\alpha\beta| + 1 > |$ stack$|$: process as normal LR

case $|\beta| = |$stack$|$: produce node for α as placeholder;

check next_state on goto(A);

reduce A;

case $|\alpha\beta| = |$stack$|$: check next_state on goto(A);

reduce A; □

[알고리즘 2] Nontermial/Goto 테이블의 구성

입력 : LR 파싱 테이블

출력 : Goto 테이블 N × { state }

방법 : 파싱 테이블의 모든 비단말 기호에 해당하는 열에서 상태 번호를 모두 구한다.

[예 1]

G_1이

E → E + T | T

T → T * F | F

F → (E) | id

와 같이 정의될 때 이에 대한 Nonterminal/Goto 테이블은 [표 1]과 같이 구성된다.

표 1. Nontermianl/Goto 테이블

Non-Terminal	Goto states after reduction
E	1 8
T	2 9
F	3 10

파싱 과정에서 여러 개의 경로가 나올 경우 각각의 트리는 별개로 구성한다. 트리가 구성될 수 있으면 유효한 서브스트링으로 간주된다. 트리가 구성될 수 없다면 해당 입력을 무시하고 트리의 단말에서 루트까지의 모든 경로를 삭제한다. 부분 문맥의 모든 입력이 이루어지고 유효한 파스 경로가 하나 이상 생성된다면 주어진 부분 문맥은 유효한 부분 문맥이다.

3. 부분 문장형의 구성

서브 스트링의 입력이 완전한 축약이 이루어지지 못한 상태에서 끝나게 되면 파서는 문장이 완성되지 못하였으므로 "error" 상태가 된다. 주어진 서브 스트링에 대하여 유도 가능한 문장형을 생성하기 위해서는 "accept" 상태가 유도될 수 있도록 추가적인 축약이 이루어지도록 해야 한다. 이러한 환원은 생성 규칙의 오른쪽 부분이 빠진 불완전한 것이다. 따라서 트리를 구성할 때 빠진 부분에 필요한 기호를 알 수 있다면 주어진 서브 스트링에 대한 트리를 구성할 수 있게 된다. 이를 해결하기 위한 기본적인 방법은 파싱표 상에서

`action[z, $] = error`

항목을

`action[z, $] = reduce ~P`

형태의 축약 항목으로 바꾸는 것이다. 여기서 $는 입력 끝을 나타내며 ~P는 생성규칙 P에 대하여 불완전한 기호를 가진 축약임을 나타낸다. 만일 입력 끝 기호가 입력에서 삭제되지 않도록 하면 이러한 방법은 서브 스트링에 대한 트리가 완성될 때까지 추가적인 축약을 수행하게 된다.

이를 위해서 불완전한 축약의 표현과 트리 구성 동작의 표현, 그리고 파싱표의 구성에서 별도의 축약 항목을 계산하기 위한 방법이 필요하다.

action 함수 : $S \times (X \cup \{\$\})$

여기서, S는 상태 번호, X는 단말기호 또는 비단말기호, $는 입력 끝기호이다. 입력이 끝나게 되면 구문이 완성되지 않았음을 알리고 필요한 추가 심볼을 계산할 수 있도록 추가적인 축약이 이루어지도록 해야 한다. 이러한 축약은 생성 규칙의 오른쪽의 일부가 빠진 것이므로 불완전한 것이다. 이를 해결하기 위하여 파싱 테이블에서 action[S,$] = error 형태의 항목을 action[S,$] = ~P 형태의 축약 항목으로 바꾼다. 만약 입력 끝 기호가 입력에서 삭제되지 않는다고 가정하면 이러한 방식은 "accept"상태에 도달할 때까지 추가적인 축약을 수행하게 한다.

일반적으로 하나의 생성 규칙

$$X \rightarrow X_1 \ldots X_n$$

에 대해

$$X \rightarrow X_1 \ldots X_{n\,1}$$

$$X \rightarrow X_1 \ldots X_{n\,2}$$

.....

$$X \rightarrow X_1 X_2$$

$$X \rightarrow X_1$$

와 같은 모든 불완전한 형태가 고려되어야 한다. 이러한 불완전한 축약은 생성 규칙의 오른쪽의 길이를 나타내는 정보가 주어지면 원래의 생성 규칙에 의해 구별될 수 있으며 위치 고정자를 갖는 트리 생성 동작은 완전한 부 트리를 대신할 위치고정자의 수를 첨가하면 된다. 모든 불완전한 축약은 일반적인 축약과 트리 생성 규칙에서 근거한 것이므로 위치 고정자들을 갖는 정확한 트리를 구성할 수 있다.

문맥 자유 문법 G가 주어지면 LR 파싱 테이블을 구성하는 과정에서 각 LR 상태 S는 LR 아이템의 집합을 갖는다. 각 아이템은 다음 중의 한가지 형태이다.

$A \rightarrow B \bullet C$	shift 아이템
$A \rightarrow D \bullet$	reduce 아이템
$A \rightarrow \bullet B$	closure 아이템

여기서 $B, C \in (N \cup T)^+$, $D \in (N \cup T)^*$ 이며 $A \rightarrow BC$, $A \rightarrow D$, $A \rightarrow B \in P$이다.

입력의 끝에서 불완전한 축약은 다음과 같은 방법으로 결정될 수 있다. 만약 어떤 상태에서 $A \rightarrow D \bullet$ 와 같은 형태로 오직 하나의 reduce 항목이 있다면, 입력 끝 기호 "$" 가 적절한 LOOKAHEAD가 아니더라도 이 항목이 축약 규칙으로 사용된다. 이러한 상태에서는

$D = d_1 \ldots d_j$ 일 때

action[S,$] = reduce A → D

handle_length[S] = j

placeholder_no[S] = 0

이 된다. 따라서 이 경우에는 표준의 축약과 트리 구성 동작이 수행된다.

만약 상태 S 가 A → B.C 와 같이 reduce 항목이 없이 오직 하나의 shift 항목만이 존재할 경우, 이 항목이 불완전한 축약에 사용된다.

$B = b_1 b_2 \ldots b_j$,

$C = c_1 c_2 \ldots c_k$,

$tree(A) = X(tree(b_{i_1}), \ldots, tree(b_{i_m}), tree(c_{l_1}), \ldots, tree(c_{l_n}))$

이면 트리 생성 규칙은

action[S,$] = reduce A → BC

handle_length[S] = j

placeholder_no[S] = m

으로 정해진다.

만약 LR 상태가 closure 항목만을 가지면 이는 초기 상태로서 특별한 경우이다. 초기 상태의 경우에는 완전한 empty 입력에 대해서는 아무런 축약도 일어날 수 없으므로 파싱표의 action[S_0,$] = error 항목은 변경되지 않으며 대신 empty 입력에 대한 분석은 하지 않아야 한다. 이러한 방식으로 생성된 불완전한 축약이 첨가된 LR 파서는 불완전한 입력에 대해서도 정확한 트리를 생성할 수 있다.

두 가지 이상의 축약이 가능한 경우에는 각 상태의 core인 축약 규칙 만을 적용한다. 이는 core에서 유도된 다른 축약 항목도 역시 해당 트리에서 유도될 수 있으므로 서브 스트링의 완성에서 가능한 보편적인 기호만을 제시하는 것이 이후의 입력에 대해서 불필요한 분석 과정을 생략할 수 있다.

[알고리즘 3] 불완전 축약을 위한 파싱 테이블 확장

입력 : 문법 G와 canonical collection C={I_1, I_2, …, I_n}

출력 : n개의 state를 갖는 ACTION 함수로 구성된 확장형 파싱표

방법 :

(1) LR 파싱 테이블을 구성한다.

(2) ACTION[S_i, $] = "error" 항목을 다음과 같이 수정한다.

(2.1) 만일[A→ α • β]의 형태가 I_i에 있으면,

ACTION[S_i, $]="reduce ~A→ α β"

(2.2) ACTION[S_0, $] = "error" □

4. 실험 및 평가

서브 스트링 "* id)" 의 파싱과정은 [표 2]와 같다. 초기 상태를 결정하기 위해서 [표 3]의 파싱 테이블이 참조된다. 파싱표에서 첫번째 입력인 '*'에 해당 하는 열에서 모든 시프트 동작을 찾는다. 축약 동작은 이전의 입력에 적용되는 것이므로 여기서는 시프트 동작만을 선택하여 그 상태 집합을 초기 상태로 한다. 예와 같은 경우에서는 '*'에 대한 시프트 동작은 모두 상태7로 전이된다. 따라서 초기 상태는 7이 된다.

이 상태을 시작 상태로 하여 파서의 동작을 수행한다. 입력이 계속됨에 따라 스택의 내용은 계속 늘어나게 된다. 이러한 동작은 'id'가 입력될 때 까지 계속된다. 상태 10에서 다음 입력 기호인 ')'에 대한 동작은 축약이 일어난다.

순서 3, 4에서 생성되는 파스 포리스트는 [그림 3]과 같다. 이 때 축약에 적용된 생성 규칙인 "T → T * F"를 만족하기에는 스택의 내용이 부족하다. 따라서 부족한 기호인 'T'에 해당하는 기호노드를 생성하고 축약을 수행한다. [그림 3]에서 생성된 노드 T는 회색으로 표시되어 있다.

축약 후 상태는 goto 표를 참조하여 T에 대한 전이 상태를 결정한다. 이 예에서는 2와 9의 두 가지 상태가

결정된다. 이 상태들에 대해 각각 파싱을 수행하기 위해 스택을 두 가지 상태 (순서 5.1, 5.2)로 분리된다.

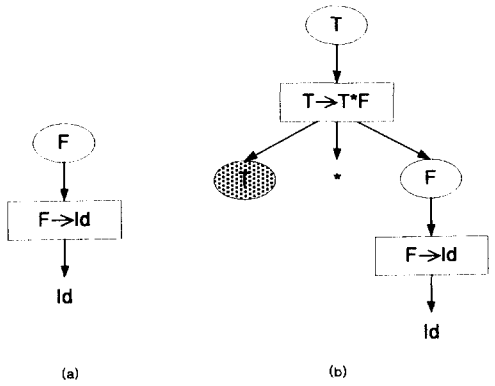

그림 3. 서브스트링 "* id)" 의
파스 포리스트 구성(I)

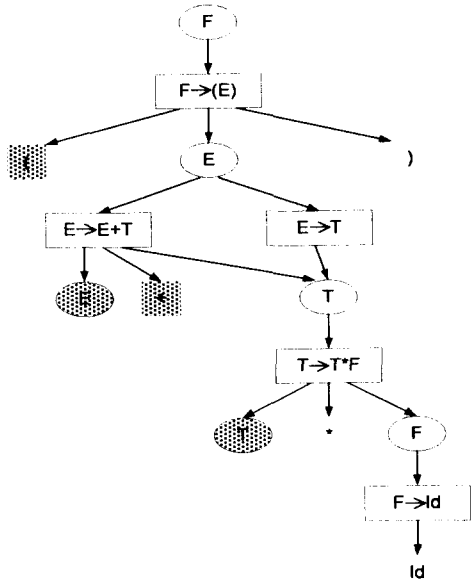

그림 4. 서브스트링 "* id)" 의
파스 포리스트 구성(III)

표 2. 서브 스트링 "* id)" 의 파싱 단계

순서	STACK		ACTION
1	($$,	*id)$$	shift 7
2	($$*5,	)	shift 5
3	($$*5id7,	id)$$)	reduce F → id
4	($$*5F10,	)$$)	reduce T → T * F
		)$$)	makenode(T)
			check_goto(T)
			duplicate 5.1, 5.2
5.1	($$T2,		reduce E → T
		)$$)	check_goto(E)
			duplicate 6.1.1, 6.1.2
5.2	($$T9,		reduce E → E + T
		)$$)	makenode(E)
			makesymbol("+")
			check_goto(E)
			duplicate 6.2.1, 6.2.2
			merge 6.2.1 to 6.1.1
			merge 6.2.2 to 6.1.2
6.1.1	($$E1,		error : discard
6.1.2	($$E8,	)$$)	shift 11
7	($$E8)11,	)$$)	reduce F → (E)
		$$)	makesymbol("(")
			check_goto(E)
8	($$F,		p-accept
		$$)	

표 3. 불완전 축약 규칙을 가지는
확장된 파싱표

	Action								
	id	+	*	(	)	F	E	T	$
0	s5			s4		s3	s1	s2	
1		s6							acc
2		r2	t7		r2				r2
3		r4	r4		r4				r4
4	s5			s4		s3	s8	s2	r~5
5		r6	r6		r6				r6
6	s5			s4		s3		s9	r~1
7	s5			s4		s10			r~3
8		s6			s11				r~5
9		r1	s7		r1				r1
10		r3	r3		r3				r3
11		r5	r5		r5				r5

파싱 순서 5.1은 역시 축약을 수행한 후 다음 상태를 [표 1]을 참조하여 결정한다. 이 때에도 두 가지 상태가 주어지므로 다시 이 상태를 두 가지 파싱 순서로 분리한다.(순서 6.1.1, 6.1.2) 파싱 순서 5.2는 축약을 수행하기 위해 필요한 노드를 생성한다. 파싱 순서 5.1과 5.2에 의

해 생성된 파스 결과는 [그림 4]와 같다.

5.2의 파싱 결과에서도 두 가지 상태가 주어진다. 5.1과 마찬가지로 파싱 순서를 분리한다. 분리한 결과는 기존의 파서의 상태가 비교하여 동일한 상태를 가진 파서는 병합되게 된다.

주어진 두 가지 상태에서 파서가 오류를 나타내는 것은 해당 파서의 상태를 제외한다. 제외되지 않는 파서 순서는 계속 파싱을 진행하여 최종적으로 하나의 기호 'F'로 축약된다. 입력된 서브 스트링은 'F'를 루트 노드로 하는 파스 포리스트 구성하였으며, 문장을 완성하기 위한 기호에 대한 노드를 포함하고 있다. 완성된 구조는 [그림 4]와 같다.

서브 스트링 "(id* "의 파싱과정은 [표 4]와 같다. 순서 1에서 순서 5 까지의 과정은 일반적인 파싱 과정을 따른다. 순서 6에서 적용된 동작 규칙 reduce ~T → T * F 는 생성규칙의 RHS에 해당하는 모든 기호가 스택에 입력되지 않은 불완전한 축약이다. 따라서 입력되지 않은 기호인 "F" 노드를 생성한 후 축약이 이루어진다.

모든 기호가 입력된 상태에서는 단일한 노드로 축약될 때까지 파싱과정을 수행한다. 순서 8에서는 불완전한 축약이 이루어지므로 기호 노드 ")"가 생성되어 완성된다.

표4. 서브 스트링 "(id * " 의 파싱 단계

Sequence	Configuration		Action
1	($$,	(id*$$	shift 4
2	($$(4,	)	shift 5
3	($$(4id5,	id*$$)	reduce F → id
4	($$(4F3,	*$$)	reduce T → F
5	($$(4T2,	*$$)	shift 7
6	($$(4T2*7,	*$$)	reduce ~T → T * F
		$$)	makenode(F)
7	($$(4T2,		reduce E → T
8	($$(4E8,	$$)	reduce ~F → (E)
		$$)	makesymbol(")")
9	($$F		p-accept
		$$)	

5. 결론

서브 스트링의 인식은 대화식 프로그래밍 환경을 구성하기 위한 필수적인 기반 요소이다. 기존의 대화식 환경 구성 방법에서 제시된 점진적 파싱 알고리즘은 완성된 문장의 부분적인 변경만을 다루었으나 이는 실제 프로그램 개발 단계에서 제한적으로만 적용될 수 있다. 본 논문에서 제안된 서브 스트링인식 기법은 기존의 제한된 분석 방법을 프로그램의 초기 입력 단계부터 적용할 수 있으므로 알고리즘의 적용 범위를 확장할 수 있다.

참고 문헌

[1] Aho, A.V., Sethi R., and Ullman J. D., *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[2] Bates, J. and Lavie A.,"Recognizing Substring of LR(K) Languages in Linear Time", *ACM TOPLAS*, Vol.16 ,No.3, pp.1051-1077, 1994.

[3] Grune D., Jacobs C., *Parsing Techniques: A Practical Guide*, Ellis Horwood Limited, 1998.

[4] Jalili F. and Gallier J., "Building Friendly Parsers", *Proceedings of 9th ACM POPL*, pp.196-206, 1982.

[5] Larchéveque J.M., "Optimal Incremental Parsing", *ACM TOPLAS*, Vol.17 ,No.1, pp.1-15, 1995.

[6] Nozohoor-Farshi R., "GLR parsing for ε-grammars.", *Generalized LR Parsing*, pp.61-75, Kluwer Academic Pubkishers, 1991.

[7] Reckers J. and Koorn W., Substring parsing for arbitrary context-free grammars. *ACM SIGPLAN Notices*,, 26(5), pp.59-66, 1991.

[8] Snelting G., "How to build LR parsers which accept incomplete input", *ACM SIGPLAN Notices*, vol. 25, no. 4, pp.83-89, 1990.

[9] Wagner A., and Graham S., "Incremental analysis of real programing languages.", *Proceedings of ACM PLDI '97*, 1997.

[10] 송후봉, 유재우, "구문지향편집기에서의 점진적 파싱 알고리즘", *한국정보과학회논문지*, Vol.18, No.4, pp.388-398, 1991.

[11] 김상헌 외, "효율적인 노드 재사용을 위한 점진적 파싱 알고리즘," *한국정보과학회 '98 추계학술발표논문집*, 1998.