# 04장 의미 분석

## 4.5 속성을 위한 저장 공간 관리

명시적인 구문 분석 트리를 만들지 않는 컴파일러는 속성을 위한 저장 공간을 할당하고 해제하며 참조할 다른 기법을 필요로 한다. 아래의 두 부속 절에서는 상향식 구문 분석기와 하향식 구문 분석기를 위한 속성 저장 공간 관리를 각기 다룬다. 상향식 구문 분석기의 주요 난제는 아직 나오지 않았으므로 구문 분석 스택에 레코드를 가지지 않는 기호의 상속 속성을 어디에 두느냐는 것이다. 하향식 구문 분석기에서는 이 문제는 발생하지 않지만 이미 구문 분석된 기호들을 위한 공간을 유지하기 위해 약간 더 많은 노력을 들여야 하며, 이 공간을 자동으로 관리할지 동작 루틴 작성자에게 약간의 부담을 지울지도 결정해야 한다.

### 【4.5.1】 상향식 평가

**예 4.16**

동작 루틴과 함께 나타낸 상향식 구문 분석에 대한 스택 변경 내역

©(심화학습에 있는) 그림 4.14에는 그림 4.1의 속성 문법을 사용한 `(1 + 3) * 2`의 구문 분석과 속성 스택의 변경 내역이 나와 있다. 명확성을 위해 구문 분석기와 속성 평가기에 대해 하나로 결합된 스택을 보였으며, CFSM 상태 번호는 생략했다.

이 문법은 S 속성을 가지므로 기호의 속성 평가가 쉽다. `yacc/bison`으로 만든 것과 같이 자동으로 생성된 구문 분석기에서 그림 4.1에 나온 문법의 생산과 연계된 속성 규칙은 관련된 생산이 인식될 때 실행될 동작 루틴을 구성한다. `yacc/bison`의 경우 속성 규칙은 각 생산에서 기호의 속성 레코드를 명명하는 "유사 구조체"를 이용해서 C로 작성할 수 있다. 좌편 기호의 속성들은 유사 구조체 `$$`의 필드들로 접근할 수 있다. 우편 기호의 속성들은 유사 구조체 `$1`, `$2` 등의 필드들로 접근할 수 있다. 예를

들어 ©(심화학습에 있는) 그림 4.14의 변경 내역의 9~10행을 얻으려면 그림 4.1에 나온 문법의 첫 번째 규칙의 동작 루틴 버전(`$$val = $1.val + $3.val`)을 사용해야 한다.

1. (
2. ( 1
3. ( $F_1$
4. ( $T_1$
5. ( $E_1$
6. ( $E_1$ +
7. ( $E_1$ + 3
8. ( $E_1$ + $F_3$
9. ( $E_1$ + $T_3$
10. ( $E_4$
11. ( $E_4$ )
12. $F_4$
13. $T_4$
14. $T_4$ *
15. $T_4$ * 2
16. $T_4$ * $F_2$
17. $T_8$
18. $E_8$

**그림 4.14 | 그림 4.1의 문법을 사용한 (1 +3) * 2의 구문 분석/속성 스택 변경 내역.** 아래첨자는 val 속성을 나타내는 것으로 한 기호의 여러 인스턴스를 구별하기 위한 것은 아니다.

상향식 동작 루틴이 실행될 때 생산 우편의 기호에 대한 속성 레코드는 속성 스택의 상단에 있는 몇 개의 항목에서 찾을 수 있다. 생산 좌편의 기호에 대한 속성 레코드(즉, `$$`)는 아직 스택에 없다. 이 레코드를 초기화하는 것이 동작 루틴의 작업이다. 동작 루틴이 완료되면 구문 분석기는 속성 스택에서 우편 레코드를 팝하고 `$$`로 대체한다. yacc/bison에서는 주어진 생산에 대해 어떠한 동작 루틴도 명시되어 있지 않으면 기본 동작으로 `$1`를 `$$`로 "복사"한다. `$$`는 일단 푸쉬되면 `$1`이 팝되기 전에 점유했던 것과 동일한 위치를 점유하기 때문에 사실상 어떤 작업도 하지 않고 "복사본"을 얻게 된다.

## 상속 속성

**예 4.17**

"묻힌(buried)" 레코드에서 상속 속성 찾기

불행히도 S 속성 문법을 작성하는 것이 항상 쉬운 일은 아니다. 상속 속성이 바람직한 간단한 예는 C나 포트란 방식의 변수 선언에서 찾아볼 수 있다. C나 포트란의 변수 선언에서 유형명은 변수명 리스트보다 앞에 나온다.

```
dec → type id_list
id_list → id
id_list → id_list , id
```

type은 현재 유형에 대한 심볼 테이블 항목으로의 포인터를 포함하는 합성 속성 tp를 가진다고 가정하자. 이상적으로는 이 속성을 id_list에 상속 속성으로 전달해서 새로 선언된 식별자가 나올 때마다 이를 유형 정보를 전부 갖춘 상태로 심볼 테이블에 입력할 수 있게 하는 것이 좋다. 생산 id_list → id를 인식할 때 속성 스택의 최상단 레코드가 id에 대한 속성이라는 것을 알 수 있다. 그러나 이보다 더 많은 정보, 즉 다음 레코드는 type에 대한 속성이어야 한다는 것도 알 수 있다. 심볼 테이블에 위치시킬 새 항목의 유형을 찾기 위해서 안전하게 이 "묻힌" 레코드를 조사할 수 있다. 이 레코드가 현재 생산의 기호에 속하지 않았음에도 불구하고 *id_list* → id 생산에 도달할 수 있는 다른 방법이 없기 때문에 이 레코드의 존재에 의존할 수 있다.

그러면 *id_list* → *id_list* , id에 있는 id는 어떤가? 이번에는 속성 스택의 상단에 있는 세 레코드가 우편 기호 id, ,, *id_list*를 위한 것이다. 그러나 여전히 그 바로 아래에서는 dec가 인식될 수 있게 *id_list*가 전부 갖추어지기를 기다리면서 type에 대한 항목을 찾을 수 있다고 생각할 수 있다. 현재 생산 아래의 유사 구조체에 대해 0이나 음수 색인을 사용하면 다음과 같은 동작 루틴을 작성할 수 있다.

```
dec → type id_list
id_list → id { declare_id($1.name, $0.tp) }
id_list → id_list , id { declare_id($3.name, $0.tp) }
```

속성 스택에서 좀 더 안쪽에 있는 레코드는 $-1, $-2 등으로 접근할 수 있다. 이 문법 조각에서는 *id_list*가 두 군데서 나오는데, 두 경우 모두 속성 스택에서 type 레코드보다 위에 있다는 것이 보장된다. 첫 번째 *id_list*는 우편의 type 다음에 나오며 두 번째 *id_list*는 첫 번째 *id_list*가 산출한 것의 시작 부분이기 때문에 귀납적으로 type 레코드보다 위에 있음을 알 수 있다.

**예 4.18**

**문맥을 필요로 하는 문법 조각**

불행히도 근본적인 기호가 다른 생산에 상속 속성을 필요로 하는 기호가 나오는 문법도 있다. 이러한 경우에도 여전히 상속 속성을 처리할 수 있지만 근본적인 문맥 자유 문법을 수정해야 한다. 수식(과 그 안에 있는 식별자와 연산자)의 의미가 그 수식이 나오는 문맥에 따라 달라지는 펄과 같은 언어에서 이러한 예를 찾아볼 수 있다. 어떤 펄 문맥은 배열이 나올 것으로 예견하며 다른 문맥은 숫자, 문자열, 논리형 등이 나올 것으로 예견한다. 수식을 올바르게 분석하기 위해서는 문맥의 예견을 수식 하위 트리에 상속 속성으로 전달해야 한다. 이 문제를 나타내는 문법 조각은 다음과 같다.

```
stmt  → id := expr
      → ...
      → if expr then stmt
```

```
expr    → ...
```

expr에 대한 생산 내부에서 구문 분석기는 둘러싸는 문맥이 대입인지 아니면 if문의 조건인지 알지 못한다. 둘러싸는 문맥이 조건이라면 수식의 예견되는 유형은 논리형이다. 대입이라면 예견되는 유형은 대입의 좌편에 있는 식별자의 유형이다. 이 식별자는 속성 스택에서 현재 생산보다 두 레코드 아래에서 찾을 수 있다.

## 의미 후크

예 4.19

문맥을 위한 의미 후크

앞서 설명한 여러 경우를 일관적으로 처리하기 위해 문법에 의미 후크나 "표시자" 기호를 추가할 수 있다. 의미 후크는 ϵ을 생성하며 그러므로 문법이 정의한 언어를 변경하지 않는다. 의미 후크의 목적은 상속 속성을 유지하는 것이다.

```
stmt  → id := A expr
      → ...
      → if B expr then stmt
A     → ϵ { $$.tp := $-1.tp }
B     → ϵ { $$.tp := Boolean }
expr  → ... { if $0.tp = Boolean then . . . }
```

예 4.20

LR CFR이 깨지는 의미 후크

의미 후크를 위한 빈 문자열 생산은 동작 루틴을 제공할 수 있기 때문에 의미 후크를 상향식 생산의 중간에 동작 루틴을 삽입하는 일반적인 기술로 생각하기 쉽지만 불행히도 그렇지는 않다. 의미 후크는 구문 분석기가 주어진 생산 안에 있다는 것이 보장되는 곳에서만 사용할 수 있다. 그 외의 위치에서 의미 후크를 사용하면 문법의 "LR성"이 깨지게 되며 구문 분석기 생성기가 수정된 문법을 수용하지 않게 된다. 다음의 예를 보자.

```
1. stmt   → l_val := expr
2.        → id args
3. l_val  → id quals
4. quals  → quals . id
5.        → quals ( expr_list )
6.        → ϵ
7. args   → ( expr_list )
8.        → ϵ
```

이 문법에서 l-값은 선택적인 배열 첨자와 레코드 항목 한정어가 뒤따르는 "한정된(qualified)" 식별자다.[1] 이 언어는 괄호가 프로시저 호출 인자와 배열 첨자 모두를 나타

주1. 일반적으로 프로그래밍 언어에서 l-값은 값이 대입될 수 있는 모든 것(즉, 대입의 좌편에 올 수 있는 모든 것)을 말한다. 저수준 관점에서 보면 기본적으로 l-값은 주소다. r-값은 대입의 우편에 올 수 있는 모든 것이다. 저수준 관점에서 보면 r-값은 주소에 저장될 수 있는 값이다. l-값과 r-값에 대해서는 6.1.2절에서 더 알아본다.

내는 포트란과 에이다의 개념을 따른다고 가정했다. 프로시저 호출의 경우 서브루틴의 심볼 테이블 색인을 상속 속성으로서 인자 목록에 전달해서 이 색인이 인자의 개수와 유형을 검사하는 데 사용될 수 있게 하는 것이 자연스럽다.

```
stmt → id A args
A → ϵ { $$.proc_index := lookup($0.name) }
```

그러나 이를 시도하면 문제에 봉착하게 된다. 왜냐하면 다음과 같은 프로시저 호출

```
foo(1, 2, 3);
```

과 다음과 같은 배열 원소 대입이 모두 동일한 토큰 나열로 시작하기 때문이다.

```
foo(1, 2, 3);
```

오른쪽 괄호 다음의 토큰을 보기 전까지 구문 분석기는 자신이 생산 1에 대해 동작하고 있는지 아니면 생산 2에 대해 동작하고 있는지 알 수 없다. 그러므로 생산 2에서 A의 존재는 이동–환원 충돌을 유발한다. 구문 분석기는 id를 본 후 A를 인식할지 (를 이동시킬지 알지 못한다.

## 왼쪽 모서리

**예 4.21**
꼬리부의 동작 루틴

일반적으로 문맥 자유 문법에서 생산의 우편은 **왼쪽 모서리**와 **꼬리부**로 구성된다. 왼쪽 모서리만으로는 현재 구문 분석 중인 생산이 어느 것인지 알 수 없다. 꼬리부를 봐야 생산을 결정할 수 있다. LL(1) 문법에서 왼쪽 모서리는 항상 비어있다. LR(1) 문법에서 왼쪽 모서리는 우편 전체로까지 구성될 수 있다. 의미 후크는 생산의 꼬리부에는 안전하게 삽입될 수 있지만 왼쪽 모서리에는 안전하게 삽입될 수 없다. yacc/bison은 동작 루틴을 우편에 포함시킬 수 있게 함으로써 이 사실을 명시적으로 나타낸다. yacc/bison은 다음과 같은 생산을

$S \rightarrow \alpha$ { 여기에 코드가 옴 } $\beta$

다음과 같이 자동으로 변환한다.

$S \rightarrow \alpha\ A\ \beta$
$A \rightarrow \epsilon$ { 여기에 코드가 옴 }

여기서 A는 새로운 기호다. 동작 루틴이 꼬리부에 있지 않다면 변환된 문법은 LALR(1)이 아니게 되며 yacc/bison은 오류 메시지를 발생한다.

**예 4.22**
의미 후크 대신 왼쪽 인수 분해

앞서 다룬 프로시저 호출과 배열 첨자 예에서 생산 2의 args 앞에는 의미 후크를 위치시킬 수 없는데, 이는 이 위치가 왼쪽 모서리기 때문이다. 인자를 구문 분석하기 전에 심볼 테이블에서 프로시저명을 검색하고자 한다면 2.3.2절에서 논의한 **왼쪽 인수 분해**와 비슷한 방식을 사용해서 식별자로 시작하는 문장에 대한 생산들을 결합해야 한다.

```
stmt → id A quals assign_opt
A → ϵ { $$.id_index := lookup($0.name) }
quals → quals . id
      → quals ( expr_ist )
      → ϵ
assign_pt → := expr
          → ϵ
```

이러한 변경은 이동-환원 충돌을 제거해 주지만 프로시저 호출 인자와 배열 첨자에 대한 문법 하위 트리 전체를 결합하는 비용이 든다. 수정된 문법을 사용하기 위해서는 두 종류의 구성소 모두에 대해 동작하는 quals에 대한 동작 루틴을 작성해야 하는데, 주로 이 작업이 성가신 일이다. LR-계열 구문 분석기 생성기 사용자들은 가끔 문법 명확성이나 구문 분석 가능성에 대한 소망과 상속 속성을 설정하기 위한 의미 후크의 필요성 사이에 존재하는 긴장 관계를 발견한다.

## 【4.5.2】 하향식 평가

2장에서 논했듯이 하향식 구문 분석기에는 재귀 하강과 테이블 주도형이라는 두 가지 주요 형태가 있다. 재귀 하강 구문 분석기에서의 속성 관리는 거의 자명하다. 기호 foo의 상속 속성은 foo라고 명명된 구문 분석 루틴으로 전달되는 매개변수의 형태를 취하며 합성 속성은 반환 매개변수다. 이러한 합성 속성은 현재 생산의 뒷부분에 있는 기호에 상속 속성으로서 전달되거나 현재 좌편의 합성 속성으로 반환될 수 있다.

자동 생성된 하향식 구문 분석기의 속성 공간 관리는 다소 더 복잡하다. 하향식 구문 분석기는 동작 루틴이 우편의 어느 위치에나 오게 허용하기 때문에 의미 후크를 삽입하기 위해 문법을 수정할 필요가 없다(물론 하향식 문법은 빈 왼쪽 모서리를 가져야 하기 때문에 처음부터 이러한 문법을 작성하기는 더 어려울 수 있다). 구문 분석 스택은 과거가 아니라 미래를 기술하기 때문에 단순히 구문 분석 스택을 미러링하는 속성 스택을 사용할 수 없다. 두 가지 주요 선택 사항은 구문 분석기에 자동 공간 관리를 위한 (좀 더 복잡한) 알고리즘을 갖추게 하거나 동작 루틴이 명시적으로 공간을 관리하게 해야 한다.

### 자동 관리

하향식 구문 분석을 위한 속성 공간의 자동 관리는 상향식 구문 분석에서보다 더 복잡하다. 공간도 더 많이 필요로 한다. 여전히 속성 스택을 사용할 수는 있지만 속성 스택은 (가설적) 구문 분석 트리의 루트와 구문 분석의 현재 지점 사이에 있는 모든 생산의 기호들을 모두 포함해야 한다. 주어진 생산의 모든 우편 기호들은 스택에서 서로 이웃한다. 좌편은 더 깊은(루트에 더 가까운) 생산의 우편에 묻힌다.

**예 4.23**

LL 속성 스택의 연산

©(심화학습에 있는) 그림 4.15는 동작 루틴과 함께 나타낸 상수식을 위한 LL(1) 문법이다. ©(심화학습에 있는) 그림 4.16은 이 문법을 사용해서 `(1 + 3) * 2`에 대한 하향식 속성 스택의 연산을 추적한다. 왼쪽 행은 구문 분석 스택을 나타내며 오른쪽 행은 속성 스택을 나타낸다. 세 개의 전역 포인터가 속성 스택에 색인을 단다. 포인터 하나(추적 내용에서 "화살표 모양의 상자 안의" L)는 현재 생산의 좌편 기호의 속성을 유지하는 속성 스택의 레코드를 식별한다. 두 번째 포인터(추적 내용에서 화살표 모양의 상자 안의 R)은 생산 우편의 첫 번째 기호를 식별한다. L과 R은 동작 루틴이 현재 생산에 있는 기호의 속성을 찾게 해준다. 세 번째 포인터(추적 내용에서 화살표 모양의 상자 안의 N)는 아직 완전히 구문 분석되지 않은 우편 내부의 첫 번째 기호를 식별한다. 이 포인터는 생산이 예측되었을 때 구문 분석기가 L을 올바르게 갱신하게 해준다.

$E \rightarrow T$ { TT.st := T.val }[1] $TT$ { E.val := TT.val }[2]

$TT_1 \rightarrow$ + $T$ { $TT_2$.st := $TT_1$.st + T.val }[3] $TT_2$ { $TT_1$.val := $TT_2$.val }[4]

$TT_1 \rightarrow$ − T { $TT_2$.st := $TT_1$.st − T.val }[5] $TT_2$ { $TT_1$.val := $TT_2$.val }[6]

$TT \rightarrow \epsilon$ { TT.val := TT.st }[7]

$T \rightarrow F$ { FT.st := F.val }[8] $FT$ { T.val := FT.val }[9]

$FT_1 \rightarrow$ * $F$ { $FT_2$.st := $FT_1$.st  F.val }[10] $FT_2$ { $FT_1$.val := $FT_2$.val }[11]

$FT_1 \rightarrow$ / $F$ { $FT_2$.st := $FT_1$.st · F.val }[12] $FT_2$ { $FT_1$.val := $FT_2$.val }[13]

$FT \rightarrow$ { FT.val := FT.st }[14]

$F_1 \rightarrow$ − $F_2$ { F1.val := − F2.val }[15]

$F \rightarrow$ ( E ) { F.val := E.val }[16]

$F \rightarrow$ const { F.val := C.val }[17]

**그림 4.15 | 동작 루틴을 함께 나타낸 상수식을 위한 LL(1) 문법.** 굵은 위 첨자는 ©(심화학습에 있는) 그림 4.16에서 참조하기 위한 것이다.

모든 시점에서 속성 스택은 구문 분석 트리의 루트와 현재 구문 분석 스택의 상단에 있는 기호 사이의 경로상에 있는 모든 생산의 모든 기호를 포함한다. ©(심화학습에 있는) 그림 4.17은 이러한 기호를 ©(심화학습에 있는) 그림 4.16의 생략된 8개 행의 바로 위에서 그래픽적으로 식별한다. 구문 분석 트리에서 왼쪽에 있는 기호들은 이미 회수되었으며 우측에 있는 기호는 아직 할당되지 않은 상태다.

시작 시 속성 스택은 N이 가리키는 목표 기호에 대한 레코드를 포함한다. 예측된 생산의 우편을 구문 분석 스택에 푸쉬할 때 추적 내용에서 콜론으로 나타내는 "생산−끝" 표시자를 추가한다. 이와 동시에 우편 기호에 대한 레코드를 속성 스택에 푸쉬한다(이 레코드는 속성 스택에 추가될 뿐 좌편을 대체하지는 않는다). 이러한 항목을 푸쉬하기 전에 현재 L과 R 포인터를 다른 스택(여기서는 보이지 않음)에 저장한다. 그 다음 L을 이전의 N으로 설정하고 R과 N이 새로 푸쉬된 우편을 가리키게 한다.

| | |
|---|---|
| E$ | ⟨N⟩ E? |
| T1TT2:$ | ⟨L⟩ E? ⟨R⟩ ⟨N⟩ T? TT?,? |
| F8FT9:1TT2:$ | E? ⟨L⟩ T? TT?,? ⟨R⟩ ⟨N⟩ F? FT?,? |
| (E) 16:8FT9:1TT2:$ | E? T? TT?,? ⟨L⟩ F? FT?,? ⟨R⟩ ⟨N⟩ ( E? ) |
| E) 16:8FT9:1TT2:$ | E? T? TT?,? ⟨L⟩ F? FT?,? ⟨R⟩ ( ⟨N⟩ E? ) |
| T1TT2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( ⟨L⟩ E? ) ⟨R⟩ ⟨N⟩ T? TT?,? |
| F8FT9:1TT2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) ⟨L⟩ T? TT?,? ⟨R⟩ ⟨N⟩ F? FT?,? |
| C17:8FT9:1TT2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) T? TT?,? ⟨L⟩ F? FT?,? ⟨R⟩ ⟨N⟩ C1 |
| 17:8FT9:1TT2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) T? TT?,? ⟨L⟩ F? FT?,? ⟨R⟩ C1 ⟨N⟩ |
| :8FT9:1TT2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) T? TT?,? ⟨L⟩ F1 FT?,? ⟨R⟩ C1 ⟨N⟩ |
| 8FT9:1TT2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) ⟨L⟩ T? TT?,? ⟨R⟩ F1 ⟨N⟩ FT?,? |
| FT9:1TT2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) ⟨L⟩ T? TT?,? ⟨R⟩ F1 ⟨N⟩ FT1,? |
| 14:9:1TT2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) T? TT?,? F1 ⟨L⟩ FT1,? ⟨R⟩ ⟨N⟩ |
| :9:1TT2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) T? TT?,? F1 ⟨L⟩ FT1,1 ⟨R⟩ ⟨N⟩ |
| 9:1TT2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) ⟨L⟩ T? TT?,? ⟨R⟩ F1 FT1,1 ⟨N⟩ |
| :1TT2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) ⟨L⟩ T1 TT?,? ⟨R⟩ F1 FT1,1 ⟨N⟩ |
| 1TT2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( ⟨L⟩ E? ) ⟨R⟩ T1 ⟨N⟩ TT?,? |
| TT2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( ⟨L⟩ E? ) ⟨R⟩ T1 ⟨N⟩ TT1,? |
| +T3TT4:2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) T1 ⟨L⟩ TT1,? ⟨R⟩ ⟨N⟩+ T? TT?,? |
| T3TT4:2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) T1 ⟨L⟩ TT1,? ⟨R⟩ + ⟨N⟩ T? TT?,? |
| F8FT9:3TT4:2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) T1 TT1,? + ⟨L⟩ T? TT?,? ⟨R⟩ ⟨N⟩ F? FT?,? |
| C17:8FT9:3TT4:2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) T1 TT1,? + T? TT?,? ⟨L⟩ F? FT?,? ⟨R⟩ ⟨N⟩ C3 |
| 〈 8행 생략 〉 | |
| 3TT4:2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) T1 ⟨L⟩ TT1,? ⟨R⟩+ T3 ⟨N⟩ TT?,? |
| TT4:2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) T1 ⟨L⟩ TT1,? ⟨R⟩ + T3 ⟨N⟩ TT4,? |
| 7:4:2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) T1 TT1,? + T3 ⟨L⟩ TT4,? ⟨R⟩ ⟨N⟩ |
| :4:2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) T1 TT1,? + T3 ⟨L⟩ TT4,4 ⟨R⟩ ⟨N⟩ |
| 4:2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) T1 ⟨L⟩ TT1,? ⟨R⟩+ T3 TT4,4 ⟨N⟩ |
| :2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( E? ) T1 ⟨L⟩ TT1,4 ⟨R⟩+ T3 TT4,4 ⟨N⟩ |
| 2:) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( ⟨L⟩ E? ) ⟨R⟩ T1 TT1,4 ⟨N⟩ |
| :) 16:8FT9:1TT2:$ | E? T? TT?,? F? FT?,? ( ⟨L⟩ E4 ) ⟨R⟩ T1 TT1,4 ⟨N⟩ |
| ) 16:8FT9:1TT2:$ | E? T? TT?,? ⟨L⟩ F? FT?,? ⟨R⟩ ( E4 ⟨N⟩ ) |
| 16:8FT9:1TT2:$ | E? T? TT?,? ⟨L⟩ F? FT?,? ⟨R⟩ ( E4 ) ⟨N⟩ |
| :8FT9:1TT2:$ | E? T? TT?,? ⟨L⟩ F4 FT?,? ⟨R⟩ ( E4 ) ⟨N⟩ |
| 8FT9:1TT2:$ | E? ⟨L⟩ T? TT?,? ⟨R⟩ F4 ⟨N⟩ FT?,? |
| FT9:1TT2:$ | E? ⟨L⟩ T? TT?,? ⟨R⟩ F4 ⟨N⟩ FT4,? |
| *F10FT11:9:1TT2:$ | E? T? TT?,? F4 ⟨L⟩ FT4,? ⟨R⟩ ⟨N⟩ * F? FT?,? |
| F10FT11:9:1TT2:$ | E? T? TT?,? F4 ⟨L⟩ FT4,? ⟨R⟩ * ⟨N⟩ F? FT?,? |
| C17:10FT11:9:1TT2:$ | E? T? TT?,? F4 FT4,? * ⟨L⟩ F? FT?,? ⟨R⟩ ⟨N⟩C2 |
| 17:10FT11:9:1TT2:$ | E? T? TT?,? F4 FT4,? * ⟨L⟩ F? FT?,? ⟨R⟩ C2 ⟨N⟩ |
| :10FT11:9:1TT2:$ | E? T? TT?,? F4 FT4,? * ⟨L⟩ F2 FT?,? ⟨R⟩ C2 ⟨N⟩ |
| 10FT11:9:1TT2:$ | E? T? TT?,? F4 ⟨L⟩ FT4,? ⟨R⟩ * F2 ⟨N⟩ FT?,? |
| FT11:9:1TT2:$ | E? T? TT?,? F4 ⟨L⟩ FT4,? ⟨R⟩ * F2 ⟨N⟩ FT8,? |
| 〈 6행 생략 〉 | |
| 1TT2:$ | ⟨L⟩ E? ⟨R⟩ T8 ⟨N⟩ TT?,? |
| TT2:$ | ⟨L⟩ E? ⟨R⟩ T8 ⟨N⟩ TT8,? |
| 7:2:$ | E? T8 ⟨L⟩ TT8,? ⟨R⟩ ⟨N⟩ |
| :2:$ | E? T8 ⟨L⟩ TT8,8 ⟨R⟩ ⟨N⟩ |
| 2:$ | ⟨L⟩ E? ⟨R⟩ T8 TT8,8 ⟨N⟩ |
| :$ | ⟨L⟩ E8 ⟨R⟩ T8 TT8,8 ⟨N⟩ |
| $ | E8 ⟨N⟩ |

**그림 4.16** | 그림 4.15의 문법(과 동작 루틴 번호)을 사용해서 (1+3) * 2를 구문 분석할 때의 구문 분석 스택(왼쪽)과 속성 스택(오른쪽)의 추적 내용. 속성 스택의 첨자는 속성 값을 나타낸다. 두 개의 속성을 가지는 기호의 경우에는 st가 먼저 나온다.

구문 분석 스택의 상단에 동작 기호(추적 내용에서 굵고 작은 숫자로 나타낸 것)가 나오면 이를 팝하고 대응되는 동작 루틴을 실행한다. 구문 분석 스택의 상단에서 단말이 일치되면 이를 팝하고 속성 스택에서 N을 한 레코드 앞으로 이동시킨다. 구문 분석 스택의 상단에 생산-끝 표시자가 나오면 이를 팝하고 N을 L이 현재 가리키는 레코드 뒤에 오는 속성 레코드로 설정한 후 속성 스택의 R에서부터 그 앞쪽의 모든 것을 팝하고 가장 최근에 저장된 L과 R의 값을 복구한다.

이 추적 내용이 길고 지루하지만 그 복잡도는 동작 루틴 작성자로부터 완전히 감춰져 있다는 점이 매우 중요하다. 일단 공간 관리 루틴이 하향식 구문 분석기 생성기를 위한 드라이버와 통합되면 컴파일러 작성자가 보는 모든 것은 ©(심화학습에 있는) 그림 4.15의 문법이다. ©(심화학습에 있는) 그림 4.14와 ©(심화학습에 있는) 그림 4.16를 비교할 때 생산 동작 루틴의 환원과 실행이 LR 추적 내용에서는 단일 단계로 보이지만 LL 추적 내용에서는 별개로 보임으로써 실제보다 더 복잡해 보인다는 점에도 주목해야 한다.

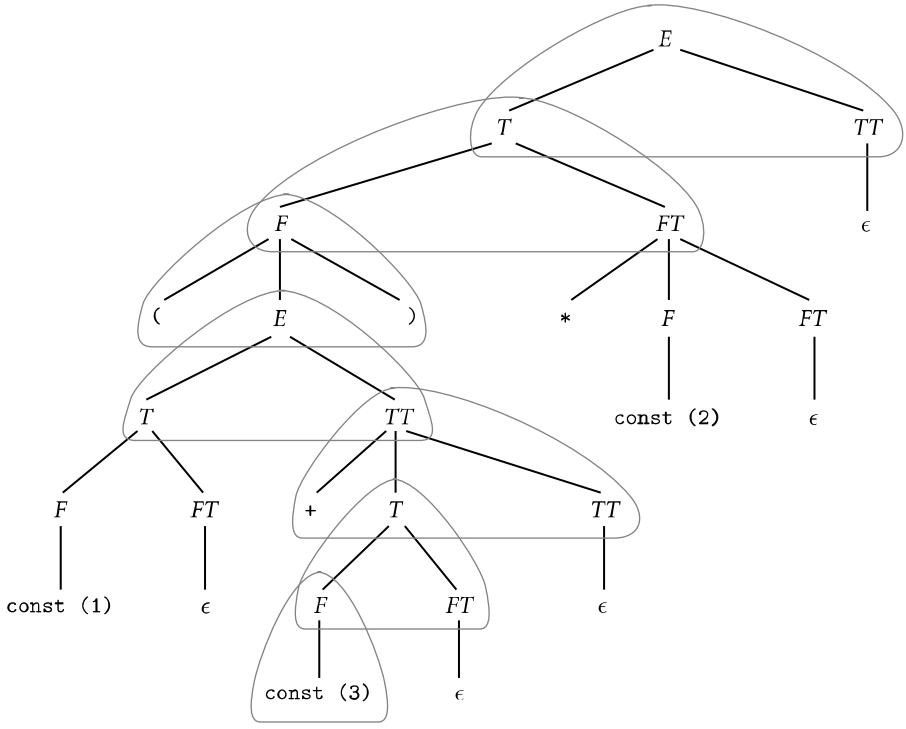


**그림 4.17 |** (그림 4.15의 문법을 사용해서) (1+3) * 2을 구문 분석하는 동안 방금 3을 구문 분석하려는 시점에 현재 속성 스택에 있는 기호와 함께 나타낸 생산. 그림 4.16에서 이 시점은 8개의 생략된 행 바로 위와 일치한다.

## 임시 관리

하향식 문법을 위한 자동 공간 관리의 한 가지 단점은 컴파일러 작성자가 복사 루틴을 명시해야 하는 빈도수다. 그림 4.9나 ©(심화학습에 있는) 그림 4.15의 17개 동작 루틴 중 12개는 단순히 한 곳에서 다른 곳으로 정보를 옮긴다. 이러한 루틴을 실행하는 데 필요한 시간은 대규모 레코드 대신 포인터를 복사함으로써 최소화할 수 있지만 컴파일러 작성자에게는 복사본이 여전히 성가신 것일 수 있다.

**예 4.24**

**의미 스택의 임시 관리**

다른 방법으로 정보가 생성되거나 소비될 때에만 임시 의미 스택을 푸쉬하거나 팝하면서 동작 루틴 내부에서 공간을 명시적으로 관리하는 것이 있다. 이 기술을 사용하면 그림 4.9의 동작 루틴을 ©(심화학습에 있는) 그림 4.18과 같이 좀 더 간단한 버전으로 대체할 수 있다. 변수 cur_tok은 가장 최근에 일치된 토큰의 합성 속성을 포함한다고 가정한다. 의미 스택은 구문 트리 노드에 대한 포인터를 포함한다. push_leaf 루틴은 명시된 상수에 대한 노드를 생성하고 이 노드에 대한 포인터를 의미 스택에 푸쉬한다. un_op 루틴은 최상단 포인터를 스택에서 팝하고 이를 명시된 단항 연산자에 대해 새로 생성된 노드의 자식 노드로 삼은 후 이 자식 노드에 대한 포인터를 다시 스택에 푸쉬한다. bin_op 루틴은 의미 스택에서 두 개의 상단 포인터를 팝하고 명시된 이진 연산자에 대해 새로 생성된 노드에 대한 포인터를 푸쉬한다. E의 구문 분석이 완료되면 산출물을 나타내는 구문 트리에 대한 포인터는 의미 스택의 최상단 레코드에서 찾을 수 있다.

```
E → T TT
TT → + T { bin op("+" } TT
TT → - T { bin op("-" } TT
TT → ϵ
T → F FT
FT → * F { bin op("×" } FT
FT → / F { bin op("÷" } FT
FT →
F → - F { un op("+/-" }
F → ( E )
F → const { push_leaf(cur_tok.val) }
```

**그림 4.18 | 구문 트리를 생성하기 위한 LL(1) 문법 속성 공간의 임시 관리**

임시 공간 관리의 장점은 분명 더 작은 수의 규칙과 왼쪽 문맥을 나타내기 위해 사용하는 상속 속성의 제거다. 단점은 컴파일러 작성자가 언제나 현재의 의미 스택 상태를 알고 적절한 시점에 푸쉬하고 팝할 것을 기억해야 한다는 점이다.

**예 4.25**

**속성 스택을 이용한 리스트 처리**

한 단계 더 나아간 다른 임시 의미 스택의 장점은 동작 루틴이 임의의 수의 레코드를 푸쉬하고 팝할 수 있다는 점이다. 자동 공간 관리를 사용하면 한 루틴이 볼 수 있는 레코드의 수가 현재 생산에 있는 기호의 개수로 제한된다. 차이는 리스트를 생성하는 생산의 경우에 특히 분명해진다. ©(심화학습에 있는) 4.5.1절에서는 C와 포트란 방식의 선언을 위한 SLR(1) 문법을 보았다. C와 포트란 방식의 선언에서는 유형명이 식별자 리스트 앞에 나온다. 아래는 변수가 유형보다 먼저 나오는 파스칼과 에이다 방식의 언어를 위한 LL(1) 문법 일부다.

```
dec → id_list : type
id_list → id id_list_tail
id_list_tail   → , id_list
               → ϵ
```

비-L 속성 흐름으로 재정렬하지 않는 한(연습문제 4.24를 보자) id_list를 상속 속성으로 선언된 유형에 전달하는 것은 불가능하다. 대신 끝부분에 유형이 나왔을 때 식별자 리스트를 모아서 한꺼번에 심볼 테이블에 입력해야 한다. 속성을 위한 자동 공간 관리를 사용하는 경우 동작 루틴은 다음과 같이 될 수 있다.

```
dec → id_list : type { declare_vars(id_list.chain, type.tp) }
id_list → id id_list_tail { id_list.chain := append(id.name, id_list_tail.chain) }
id_list_tail → , id_list { id_list_tail.chain := id_list.chain }
             → ϵ { id_list_tail.chain := nil }
```

**예 4.26**

**의미 스택을 사용한 리스트 처리**

임시 저장 공간 관리를 사용하면 연결 리스트가 필요치 않다.

```
dec → { push(marker) }
      id_list : type
      { pop(tp)
        pop(name)
        while name ≠ marker
            declare_var(name, tp)
            pop(name) }
id_list → id { push(cur_tok.name) } id_list_tail
id_list_tail → , id_list
             → ϵ
```

하향식 구문 분석기에서는 자동 속성 공간 관리와 임시 속성 공간 관리 중 어떤 하나가 절대적으로 우월하지는 않다. 임시 접근 방법에서는 많은 복사 규칙과 상속 속성이 필요치 않으며, 결과적으로 다소 더 높은 시간과 공간 효율성을 가진다. 의미 스택에 리스트를 포함시킬 수도 있다. 한편 임시 접근 방법에서는 의미 스택에서 적절히 팝하고 푸쉬하기 위해 동작 루틴을 작성하는 프로그래머가 스택에 무엇이 왜 있는지를 지속적으로 알고 있어야 한다. 결국 선택은 주로 취향의 문제로 남는다.

## ✓ 확인문제

17. 상향식 구문 분석기에서 합성 속성을 위한 공간 관리를 어떻게 하는지 설명하라.

18. 상향식 구문 분석기에서 상속 속성을 위한 공간 관리를 어떻게 하는지 설명하라.

19. 왼쪽 모서리와 꼬리부를 정의하라.

20. 상향식 구문분석기에서 동작 루틴이 생산의 우편에 포함될 수 있는 상황은 언제인가? 마찬가지로 문법을 비-LR로 만들지 않고 표시자 기호가 우편에 포함될 수 있는 상황은 언제인가?

21. 하향식 구문 분석기에서 자동 속성 공간 관리와 임시 속성 공간 관리 사이의 트레이드오프를 요약하라.

22. 하향식 구문 분석의 임의의 한 시점에 자동으로 관리되는 속성 스택에서 속성 레코드를 가지는 것은 어떤 기호인가?