02장 프로그래밍 언어 구문

【2.3.4】 구문 오류

예 2.41

C의 구문 오류(다시 보기)

책에서는 C의 간단한 예를 이용해서 구문 오류 복구의 문제를 설명했다.

```
A = B : C + D;
```

컴파일러는 B 직후에 구문 오류를 바로 찾아내지만 이 지점에서 컴파일 작업을 중단할 수는 없다. 컴파일러는 프로그램의 나머지 부분에서 추가적인 오류를 계속 찾아야 한다. 이를 허용하기 위해서는 잘못된 연속 오류를 알리지도 않고 실제 오류를 놓치지도 않으면서 구문 분석을 지속할 수 있게 해주는 방법으로 입력 프로그램이나 구문 분석기의 상태, 또는 둘 모두를 수정해야 한다. 아래에서 설명할 기술들은 컴파일러가 추가적인 구문 오류를 검색할 수 있게 해준다. 4장에서는 컴파일러가 부가적인 정적 의미 오류까지 검색하게 해주는 추가적인 기술을 살펴본다.

패닉 모드

가장 간단한 형태의 구문 오류 복구는 패닉 모드라는 기술일 것이다. 패닉 모드에서는 입력에서 오류가 없는 지점을 명확히 나타내는 "안전한 기호"로 구성된 작은 집합을 정의한다. 오류가 발생하면 패닉 모드 복구 알고리즘은 안전한 기호를 찾을 때까지 입력 토큰을 삭제한 후 구문 분석기를 그 기호가 나타날 수 있는 문맥으로 돌려보낸다. 앞서 살펴본 예에서 패닉 모드 복구를 사용하는 재귀 하강 구문 분석기는 세미콜론을 찾을 때까지 입력 토큰을 삭제하고 stmt 내에서 호출된 모든 서브루틴으로부터 복귀한 후 stmt의 몸체 자체를 다시 시작할 수 있다.

불행히도 패닉 모드는 다소 극단적이다. 구문 분석을 재개하기 위한 정적인 "안전한" 기호 집합으로 자기 자신을 제한함으로써 패닉 모드는 그러한 기호를 찾는 동안 상당

한 양의 입력을 삭제할 수 있다. 설상가상으로 삭제된 토큰 중 일부가 언어에서 대규모의 구성소를 시작하는 "시작" 기호였다면(예를 들어 `begin`, `procedure`, `while`) 사용자는 구성소의 끝에 도달할 때 잘못된 연속 오류를 보게 될 것이 거의 확실하다.

예 2.42

패닉 모드의 문제

모듈라 2로 작성한 다음과 같은 코드 일부를 생각해보자.

```
IF a b THEN x;
ELSE y;
END;
```

패닉 모드 복구 알고리즘은 첫 행의 `b`에서 오류를 발견하면 세미콜론까지 건너뛸 가능성이 높으며 그러므로 `THEN`을 지나치기 쉽다. 구문 분석기는 2행의 `ELSE`를 보고 잘못된 오류 메시지를 생성한다. 구문 분석기는 또 3행의 `END`을 보고 둘러싸는 구조(예를 들어 서브루틴 전체)의 끝에 도달했다고 생각하고 뒤로 이어지는 행들에 대해 추가적인 연속 오류를 발생할 수도 있다. 패닉 모드는 베이직이나 포트란과 같이 많은 수의 "시작" 기호를 가지지 않는 상대적으로 "구조화되지 않은" 언어에서만 제대로 동작하는 경향이 있다.

단계 수준 복구

다른 문맥에 다른 집합의 "안전한" 기호를 사용해서 복구의 질을 개선할 수 있다. 이러한 개선을 포함하는 구문 분석기는 단계 수준 복구를 구현했다고 한다. 예를 들어 수식에서 오류를 발견하면 단계 수준 복구 알고리즘은 수식 뒤에 나올 수 있는 어떤 것에 도달할 때까지 입력 토큰을 삭제할 수 있다. 이처럼 좀 더 지역적인 복구는 문장에서 잘못된 수식의 뒤에 나오는 부분을 조사할 수 있게 해주기 때문에 무조건 현재 문장의 끝으로 돌아가야 하는 방법보다 낫다.

예 2.43

재귀 하강의 단계 수준 복구

파스칼을 발명한 니클라우스 워드는 1976년에 재귀 하강을 위한 훌륭한 단계 수준 구현을 발표했다[Wir76, 5.9절]. 그의 알고리즘의 가장 간단한 버전은 2.3.1절의 끝부분에서 정의한 `FIRST`와 `FOLLOW` 집합에 의존한다. 비단말 `foo`에 대한 구문 분석 루틴이 코드의 첫 부분에서 오류를 발견하면 이 루틴이 진행 중인 경우에는 `FIRST(foo)`의 원소를 찾을 때까지, 복귀 중인 경우에는 `FOLLOW(foo)`의 원소를 찾을 때까지 입력 토큰을 삭제한다.

```
procedure foo
    if (input_token ∉ FIRST(foo)) and (ε ∉ FIRST(foo))
       report_error              -- 사용자를 위한 메시지 출력
       repeat
          delete token
       until input_token ∈ (FIRST(foo) ∪ FOLLOW(foo) ∪ {$$})
    case input_token of
```

```
        ... : ...
        ... : ...                -- 유효한 시작 토큰
        ... : ...
        otherwise return         -- 오류나 foo → ϵ(//입실론입니다)
```

report_error 루틴이 구문 분석을 종료하지 않는다는 점에 주의하자. 이 루틴은 단순히 메시지를 출력하고 복귀한다. 알고리즘을 완료하기 위해 다른 무언가가 나올 때 예견되는 토큰을 효율적으로 삽입하고 오류를 출력한 후에 복귀하게 match 루틴을 변경해야 한다.

```
procedure match(expected)
    if input_token = expected
       consume input_token
    else
       report error
```

끝으로 코드를 단순화하기 위해 다양한 비단말 서브루틴의 공통적인 접두어는 다음과 같이 오류 검사 서브루틴으로 이동시킬 수 있다.

```
procedure check_for_error(first_set, follow_set)
    if (input_token ∉ first_set) and (ϵ ∉ first_set)
       report_error
       repeat
          delete_token
       until input_token ∈ first_set ∪ follow_set ∪ {$$}
```

문맥 특화 미리 보기

앞서 설명한 복구 알고리즘은 간단하긴 하지만 $foo \rightarrow \epsilon$ 와 같은 경우 실제로 오류를 바로 출력해야 하는 때에 하나 이상의 빈 문자열 생산을 예측하는 좋지 않은 경향이 있다. 이러한 약점을 **즉시 오류 탐지 문제**라고 한다. 즉시 오류 탐지 문제는 FOLLOW(foo)가 문맥 독립적이라는 사실에서 기인한다. FOLLOW(foo)는 현재 프로그램의 현재 문맥에서는 아닐지라도 어떤 유효한 프로그램에서 foo 뒤에 올 수 있는 모든 토큰을 포함한다(이는 2.3.3절에서 언급한대로 SLR과 LALR 구문 분석기 간 구별의 기본이 되는 원리와 기본적으로 동일하다).

예 2.44

연속 구문 오류

예로서 다음과 같이 계산기 언어로 작성한 잘못된 코드를 생각해보자.

```
Y := (A * X X*X) + (B * X*X) + (C * X) + D
```

사람은 프로그래머가 다항식의 x^3항에서 *를 실수로 넣지 않았다는 것을 매우 쉽게 알 수 있다. 그러나 복구 알고리즘은 그렇게 똑똑하지 않다. 재귀 하강 구문 분석기의

복구 알고리즘은 다음과 같은 루틴 내부에 있을 때 입력의 다음 식별자 (X)를 보게 된다.

```
program
stmt_list
stmt
expr
term
factor
expr
term
factor_tail
factor_tail
```

일부 프로그램(예를 들어 `A := B  C := D`)에서는 id가 factor_tail의 뒤에 올 수 있기 때문에 가장 안쪽의 구문 분석 루틴은 factor_tail → ϵ을 예측하고 단순히 복귀한다. 이 시점에서 바깥쪽의 factor_tail과 안쪽의 term은 모두 각자 코드의 끝부분에 있으며 역시 복귀한다. 다음으로 좀 더 안쪽의 expr은 term_tail을 호출하며 특정 프로그램에서 id는 term_tail의 뒤에 올 수 있기 때문에 term_tail은 또 다시 빈 문자열 생산을 예측한다. 이로 인해 안쪽의 expr은 자신의 코드 끝부분에 남아 복귀하게 된다. 그리고 factor가 오른쪽 괄호를 예견하면서 match를 호출할 때에서야 비로소 오류를 발견하게 된다. 그 후 입력이 다음과 같이 변환되면서 많은 수의 연속 오류가 발생한다.

```
Y := (A * X)
X := X
B := X*X
C := X
```

예 2.45

문맥 특화 미리 보기로 연속 오류 줄이기

부적절한 빈 문자열 생산을 피하기 위해서 워드는 **문맥 특화 FOLLOW 집합**이라는 개념을 도입했다. 문맥 특화 FOLLOW 집합은 명시적인 매개변수로서 모든 비단말 서브루틴에 전달된다. 앞서 살펴본 예에서는 생산 *stmt* → id := *expr*의 일부로 호출되는 처음의 바깥쪽 expr을 위한 FOLLOW 집합의 일부로서 id가 전달되지만 생산 *factor* → (*expr*)의 일부로 호출되는 안쪽의 두 번째 expr에는 id가 전달되지 않는다. term과 factor_tail에 대한 중첩된 호출은 결국 오른쪽 괄호가 유일한 멤버인 FOLLOW 집합과 함께 호출된다. factor_tail에 대한 안쪽의 호출은 id가 FIRST(factor_tail)에 없다는 것을 발견하면 복귀하기 전에 오른쪽 괄호가 나올 때까지 토큰을 삭제한다. 최종적인 결과는 단일 오류 메시지와 다음과 같은 입력 변환이다.

```
Y := (A * X) + (B * X*X) + (C * X) + D
```

물론 이것도 여전히 "올바른" 해석은 아니지만 이전보다는 훨씬 낫다.

예 2.46

완전한 단계 수준 복구를 사용한 재귀 하강

워드의 단계 수준 복구의 마지막 버전은 연속 오류를 피하기 위해 추가적인 휴리스틱을 하나 사용한다. 이 휴리스틱은 정적으로 정의된 "시작" 기호 집합(예를 들어 `begin`, `procedure`, `(` 등)의 원소를 삭제하지 않는다는 것이다. 이러한 기호들은 프로그램의 뒷부분에서 일치되는 토큰을 필요로 하는 경향이 있다. 입력을 삭제하는 도중 시작 기호를 발견하면 오류가 있는 구성소의 나머지 부분에 대한 삭제를 중단한다. 그리고 호출하는 루틴이 시작 기호를 받아들이지 않을 것을 알면서도 단순히 복귀한다. 문맥 특화 FOLLOW 집합과 시작 기호를 가지는 단계 수준 복구는 다음과 같다.

```
procedure check_for error(first_set, follow_set)
    if input_token ∉ first set
       report_error
       repeat
          delete_token
       until input_token ∈ first_set ∪ follow_set ∪ starter_set ∪ {$$}

procedure expr(follow_set)
    check_for_error(FIRST(expr), follow_set)
    case input_token of
       ...:...
       ...:...        유효한 시작 토큰
       ...:...
       otherwise return
```

재귀 하강의 예외 기반 복구

워드의 기술 대신 사용할 수 있는 매력적인 방법은 에이다, 모듈라 3, C++, 자바, C#, ML 등과 같은 다수의 현대 언어에서 지원하는 예외 처리 기법에 의존한다. 언어의 모든 비단말에 대한 복구를 구현(이는 다소 지루한 작업이다)하는 대신 예외 기반 접근 방법은 오류가 발생할 때 벗어날 소규모 문맥 집합을 식별한다. 예를 들어 가장 가까운 수식이나 문장으로 벗어나게 할 수 있다. 극단적으로 "벗어날" 곳을 하나만 두면 패닉 모드 복구를 구현하게 된다.

이러한 방식의 오류 복구는 8.5.3절에서 예외 처리를 다룰 때 좀 더 자세히 살펴본다. 기본적인 아이디어는 오류 복구를 구현하고자 하는 코드의 블록에 예외 처리기(특수한 구문 구성소)를 추가하는 것이다. (다수의 프로시저 호출만큼 중첩될) 오류가 탐지되면 구문 오류 예외를 발생시킨다("`raise`"는 예외를 지원하는 언어들의 고유 명령어다). 그 후 언어 구현은 예외 처리기가 있는 가장 최근의 문맥까지 스택을 풀고 예외 처리기가 추가되어 있는 블록의 나머지 부분 대신 이 예외 처리기를 실행한다. 단계 수준(또는 패닉

모드) 복구에서 처리기는 구문 분석을 다시 시작할 수 있는 토큰을 발견할 때까지 입력 토큰을 삭제할 수 있다. 처리기가 삭제할 수 없는 시작 기호를 발견하면 프로시저 호출 체인상의 다음 처리기로 팝하면서 예외를 다시 발생시킨다.

2.3.1절에서 살펴봤듯이 ANTLR 구문 분석기 생성기는 입력으로 CFG를 취해 사람이 읽을 수 있는 재귀 하강 구문 분석기를 생성한다. 컴파일러 작성자는 예외 처리 기법을 가지는 자바, C#, C++를 생성할 수 있다. ANTLR로 생성한 구문 분석기는 구문 오류를 발견하면 `MismatchedTokenException`이나 `NoViableAltException`을 던진다. 기본적으로 ANTLR은 모든 비단말 서브루틴에 이러한 예외에 대한 처리기를 포함한다. 처리기는 오류 메시지를 출력하고 비단말의 FOLLOW 집합에 속하는 무언가를 발견할 때까지 토큰을 삭제한 후 복귀한다. 컴파일러 작성자는 원하는 경우 생산 단위 기반으로 다른 처리기를 정의할 수 있다.

오류 생산

일반적으로 오류 복구 기술은 가능한 한 언어 독립적인 것이 바람직하다. 특정 언어를 위해 사람이 직접 작성한 재귀 하강 구문 분석기에서도 오류 복구를 `check_for_error`와 `match` 서브루틴에 캡슐화하는 것이 좋다. 그러나 때로는 한 언어에 크게 특화시킴으로써 훨씬 더 나은 복구 효과를 얻을 수도 있다.

예 2.47

"; else"에 대한 오류 생산

대부분의 언어는 프로그래머가 예측 가능한 방식으로 틀리기 쉬운 직관적이지 않은 규칙을 몇 개 포함한다. 예를 들어 파스칼에서 세미콜론은 문장을 분리하는 데 사용하지만 많은 프로그래머는 이를 **문장 종결자**로 생각한다. 파스칼에서는 빈 문장도 허용되기 때문에 대부분의 경우 이러한 오해는 중요하지 않다. 예를 들어 다음과 같은 코드를 보자.

```
begin
    x := (-b + sqrt(b*b -4*a*c)) / (2*a);
    writeln(x);
end;
```

컴파일러는 세 개의 문장으로 `begin...end` 블록을 구문 분석하며 이때 세 번째 문장이 빈 것으로 간주된다. 반면 다음과 같은 코드를 보자.

```
if d <> 0 then
    a := n/d;
else
    a := n;
end;
```

파스칼에서 `if...then...else` 구성소의 `then` 부분은 단일 문장으로 구성되어야

하기 때문에 컴파일러는 위 코드를 구문 분석할 때 오류를 발생하게 된다. 파스칼에서 세미콜론은 절대로 `else` 직전에 나올 수 없지만 프로그래머는 항상 그 위치에 세미콜론을 놓는다. 대부분의 파스칼 컴파일러 작성자는 이 문제를 올바르게 처리하기 위해 일반적인 복구나 수정 알고리즘을 조정하기보다는 문법을 수정한다. 수정된 문법은 세미콜론을 허용하는 추가적인 생산을 포함하지만 세미콜론이 해당 위치에 올 수 없다는 것을 사용자에게 알려주기 위해 의미 분석기가 경고 메시지를 출력하게 한다. C가 발전함에 따라 발생하게 된 "시대 착오"를 처리하기 위해 C 컴파일러도 이와 유사한 오류 생산을 사용한다. 구문 분석기는 여전히 초기 버전의 C에서만 유효했던 구문을 받아들이지만 경고 메시지를 발생한다.

테이블 주도형 LL 구문 분석기의 오류 복구

테이블 주도형 하향식 구문 분석기는 재귀 하강 구문 분석기와 유사하기 때문에 이를 위한 단계 수준 복구를 구현하는 것은 어렵지 않다. 구문 분석 테이블에서 오류 항목이 나올 때마다 단순하게 정적으로 정의된 시작 기호(`$$` 포함) 집합의 멤버나 구문 분석 스택의 최상단에 있는 비단말의 FIRST 집합이나 FOLLOW 집합의 멤버가 나올 때까지 입력 토큰을 삭제하면 된다.[1] FIRST 집합의 멤버를 찾으면 드라이버의 주 루프를 재개한다. FOLLOW 집합이나 시작 기호 집합의 멤버를 찾으면 우선 구문 분석 스택에서 비단말을 팝한다. 구문 분석 테이블이 아니라 `match`에서 오류가 발생하면 단순히 해당 토큰을 구문 분석 스택에서 팝한다.

그러나 이보다 더 나은 복구를 수행할 수 있다! 테이블 주도형 하향식 구문 분석기에서는 쉽게 접근할 수 있는 전체 구문 분석 스택을 가지기 때문에(재귀 하강에서는 제어 흐름과 프로시저 호출 순서에 숨겨져 있다) 구문 분석을 계속할 수 있게 해주는 삽입과 삭제의 가능한 모든 조합을 열거하는 것이 가능하다. 적절한 기준이 주어지면 여러 대안을 평가해 그 중 어떤 면에서 "최고"인 것을 선택할 수 있다.

완벽한 오류 복구(실제로 오류 수정)를 위해서는 프로그래머의 마음을 읽어야 하기 때문에 여러 "정정"을 평가하는 실제 기술은 휴리스틱에 의존해야 한다. 대부분의 컴파일러는 단순성을 위해 (1) 의미 정보를 필요로 하지 않고 (2) 구문 분석기나 입력 스트림을 "백업"할 필요가 없으며(즉, 오류가 탐지되기 전의 어떤 상태로 돌아갈 필요가 없으며) (3) 토큰의 철자나 토큰 간의 경계를 수정하지 않는 휴리스틱만을 사용한다. 피셔, 밀튼, 퀴링은 1980년에 이러한 제약 사항을 따르는 매우 훌륭한 알고리즘을 발표했다 [FMQ80, FL88]. 처음 기술된 대로 이 알고리즘은 삭제 없이 입력 스트림에 적절한 토큰을 삽입함으로써 항상 프로그램을 수정할 수 있는 언어에만 적용 가능했다. 그러나

주1. 이 설명은 전역 FOLLOW 집합을 사용한 것이다. 문맥 특화 미리 보기를 사용하고자 않다면 스택의 아랫부분을 미리 볼 수 있다. 토큰은 스택의 두 번째 기호 A의 FIRST 집합에 속하거나 스택 상단으로부터 세 번째 기호 B의 FIRST 집합인 A → ϵ의 예측을 초래하거나 네 번째 기호의 FISRT 집합인 B → ϵ의 예측을 초래하면(이런 식으로 다섯 번째, 여섯 번째 등으로 확장 가능) 수용 가능하다.

삭제와 치환을 포함하게 이 알고리즘을 확장하는 것은 비교적 쉽다. 우선은 삽입만 하는 알고리즘을 살펴본다. 삭제도 지원하는 버전은 삭제를 서브루틴으로서 사용한다. 치환은 다루지 않는다.[2]

FMQ 오류 수정 알고리즘을 사용하려면 컴파일러 작성자는 모든 토큰 t에 삽입 비용 C(t)와 삭제 비용 D(t)를 할당해야 한다(입력이 끝나는 곳은 변경할 수 없으므로 C($$) = D($$) = ∞이 된다). 오류가 발생하면 이 알고리즘은 구문 분석기가 실제 입력의 토큰을 하나 더 소비하게 해주는 삽입과 삭제의 최소 비용 조합을 선택한다. 구문 분석기의 상태는 절대 변경되지 않으며 입력만 수정된다(수정 알고리즘은 스택 기호를 팝하는 대신 입력 스트림에 수정 산출물을 푸쉬한다).

재귀 하강 구문 분석기의 단계 수준 복구에서와 마찬가지로 FMQ 알고리즘도 즉시 오류 탐지 문제를 해결해야 하며, 이에는 몇 가지 방법이 존재한다. 지역 FOLLOW 집합을 유지하는 "완전 LL" 구문 분석기를 사용할 수 있다. 또 빈 문자열 생산을 예측할 때 스택 아래에 있는 것이 입력 토큰을 수용하게 해주는지 알아보기 위해 스택을 조사할 수도 있다. 첫 번째 방법은 구문 분석기의 크기와 복잡도를 크게 증가시킨다. 두 번째 방법은 비선형 시간의 구문 분석 알고리즘을 초래한다. 다행히도 세 번째 방법이 있다. match 루틴이 다른 실제 입력 토큰을 수용할 때까지 스택의 모든 변경 사항(과 의미 분석기의 동작 루틴에 대한 모든 호출)을 임시 버퍼에 저장할 수 있다. 실제 토큰을 수용하기 전에 오류를 발견하면 스택 변경을 취소하고 버퍼에 저장한 동작 루틴에 대한 호출을 취소한다. 그 다음 완전 LL 구문 분석기가 오류를 인식했을 시점에 오류를 인식한 것처럼 한다.

이제 삽입만으로 수정하는 작업을 생각해보자. 우선 삽입 비용의 개념을 명백한 방식으로 문자열에 확장해보자. w = $a_1a_2...a_n$이라면 $C(w) = \sum_{i=1}^{n} C(ai)$이 된다. 이 비용 함수 C를 사용해서 테이블 S와 E의 쌍을 만들 수 있다. S 테이블은 1차원으로 문법 기호에 의해 색인된다. 임의의 기호 X에 대해 S(X)는 X로부터 유도할 수 있는 최소 비용의 단말 문자열이다. 즉, 아래와 같다.

S(X) = w ⟺ X ⇒* w이며 X ⇒* x인 ∀x, C(w) ≤ C(x)

물론 토큰 a에 대해 S(a)=a∀다.

E 테이블은 2차원으로 기호/토큰 쌍으로 색인된다. 임의의 기호 X와 토큰 a에 대해 E(X, a)는 X에 있는 a의 최저 비용 접두어, 즉 X ⇒* wax인 최저 비용 토큰 문자열 w다. X가 a를 포함하는 문자열을 산출할 수 없다면 E(X, a)는 삽입 비용이 ∞인 특수

주2. 치환은 항상 삭제/삽입 쌍으로 표현할 수 있지만 이상적으로는 이를 특별하게 고려하는 것이 좋다. 예를 들어 왼쪽 대괄호나 왼쪽 괄호는 연속 오류를 발견할 가능성이 높은 입력 뒷부분의 무언가와 일치되어야 하기 때문에 왼쪽 대괄호를 삭제하거나 왼쪽 괄호를 삽입할 때는 주의가 필요할 수 있다. 그러나 왼쪽 괄호를 왼쪽 대괄호로 치환하는 것은 어떤 면에서 좀 더 그럴듯하며, 이는 특히 여러 프로그래밍 언어의 배열 첨자 구문 간의 차이가 주어졌을 때 더욱 그러하다.

기호 ??로 정의된다. X=a이거나 X $\Rightarrow^*$ ax면 E(X, a)=ϵ이 되며 C(ϵ)=0이다.

```
function find_insertion(a : token) : string
    -- 구문 분석 스택은 기호 Xn,...X2, X1으로 구성되며 스택의 상단은
    -- Xn이라고 가정한다.
    ins := ??
    prefix := ϵ
    for i in n..1
       if C(prefix) ≥ C(ins)
          -- 더 나은 삽입이 불가능
          return ins
       if C(prefix . E(Xi, a)) < C(ins)
          -- 더 나은 삽입을 발견
          ins := prefix . E(Xi, a)
       prefix := prefix . S(Xi)
    return ins
```

그림 2.30 | 구문 분석기가 입력 토큰 a를 수용하게 해주는 최소 비용 삽입을 찾기 위한 함수의 초안. 여기서 마침표 문자(.)는 문자열 연결을 의미한다.

예 2.48

FMQ의 삽입 유일 수정

주어진 오류를 수정하는 최소 비용 삽입을 찾기 위해 ©(심화학습에 있는) 그림 2.30과 같은 find_insertion 함수를 실행한다. 이 함수는 우선 스택 상단의 기호로부터 입력 토큰을 유도하게 해주는 최소 비용 삽입을 고려한다(이러한 삽입이 없을 수도 있다). 그 후 (입력 스트림에 최소 비용 산출물을 삽입함으로써) 스택의 상단 기호를 "삭제"하고 스택의 두 번째 기호로부터 입력 토큰을 유도하는 것을 고려한다. 이런 방식으로 기호의 산출물을 삽입하는 비용이 이제까지 발견한 최저 비용 수정의 비용을 초과할 때까지 스택의 기호로부터 입력 토큰을 유도하는 방법을 고려해간다. 스택의 하단에 도달할 때까지 유한 비용 수정을 발견하지 못한다면 해당 오류는 삽입만으로는 수정할 수 없는 것이다.

예 2.49

삭제를 지원하는 FMQ

좀 더 양질의 수정을 생성하고 삽입만으로 수정할 수 없는 언어를 처리하기 위해 삭제를 고려해야 한다. 삽입 비용 벡터 C로 했던 것과 마찬가지로 명백한 방법으로 확장 비용 벡터 D를 토큰 문자열에 확장할 수 있다. 그 후 ©(심화학습에 있는) 그림 2.31과 같이 두 번째 루프에 find_insertion에 대한 호출을 삽입한다. 이 루프는 추가적인 토큰을 삭제하는 비용이 이제까지 발견한 최소 비용 수정의 비용을 초과할 때까지 반복적으로 남은 입력에 대해 find_insertion을 호출하면서 점차 더 많은 토큰의 삭제를 고려한다. 이 검색은 절대로 실패하지 않는다. 파일 끝 토큰을 수용하게 해주는 삽입과 삭제 조합을 찾는 것은 언제나 가능하다. 이 알고리즘은 임의의 수의 토큰을 삭제하는 선택 사항을 고려(하고 거부)할 필요가 있을 수 있기 때문에 어휘 분석기는 입력 스트림에서 임의의 거리를 미리 보고 다시 돌아올 수 있어야 한다.

```
function find_repair : string, int
    -- 구문 분석 스택은 Xn, . . . X2, X1으로 구성되며
    -- 스택 상단은 Xn이라고 가정
    -- 또 입력 스트림은 토큰 a1, a2, a3, ...로 구성된다고 가정
i := 0    -- 삭제를 고려 중인 토큰의 수
best_ins := ??
best_del := 0
loop
    cur_ins := find_insertion(ai+1)
    if C(cur_ins) + D(a1...ai ) < C(best_ins) + D(a1...abest_del)
       -- 좀 더 나은 수정을 발견
       best_ins := cur_ins
       best_del := i
    i +:= 1
    if D(a1...ai ) > C(best_ins) + D(a1...abest del)
       -- 좀 더 나은 수정이 불가능
       return (best_ins, best_del)
```

그림 2.31 | 구문 분석기가 하나 이상의 입력 토큰을 수용하게 해주는 삽입과 삭제의 최소비용 조합을 찾기 위한 함수의 초안

FMQ 알고리즘은 몇 가지 바람직한 특징을 가진다. 우선 간단하며 효율적이다(문법의 크기가 유한한 경우 상수 시간에 적절한 수정을 선택할 수 있다는 것을 증명할 수 있다). 또 임의의 입력 문자열을 수정할 수 있다. FMQ 알고리즘의 결정은 구문 분석기로 하여금 앞으로 진행할 수 있게 해주는 더 낮은 비용의 수정이 없다는 면에서 지역적으로 최적이다. 이 알고리즘은 테이블 주도형이며, 그러므로 완전히 자동이다. 끝으로 토큰의 삽입 비용과 삭제 비용을 수정함으로써 "좀 더 있을 법한" 수정을 선호하게 개정할 수 있다. 알기 쉬운 휴리스틱으로 다음과 같은 것들이 있다.

- 주로 삭제 비용이 삽입 비용보다 크다.
- 일상적인 연산자(예를 들어 곱셈)의 비용은 문법에서 동일한 위치에 있는 일상적이지 않은 연산자(예를 들어 나머지 연산)의 비용보다 작아야 한다.
- 시작 기호(예를 들어 begin, if, ()의 비용은 그에 대응되는 종료 기호(end, fi,))의 비용보다 커야 한다.
- "공해" 기호(콤마, 세미콜론, do)의 비용은 매우 작아야 한다.

상향식 구문 분석기의 오류 복구

지역적 최소 비용 수정은 상향식 구문 분석기에서도 가능하지만 하향식에서와 같이 쉽지는 않다. 하향식 구문 분석기의 장점은 구문 분석 스택의 내용이 모호성 없이 오류의 문맥을 식별하고 향후에 예견되는 구성소를 명시한다는 것이다. 이와 달리 상향식 구문 분석기의 스택은 가능한 문맥 집합을 기술하고 명시적으로는 미래에 대해서 어떤 것도 알려주지 않는다.

실제로 대부분의 상향식 구문 분석기는 패닉 모드나 단계 수준 복구에 의존하는 경향이 있다. 직관적으로 보면 오류가 발생할 때 구문 분석 스택의 상단에 있는 몇 개의 상태는 잘못된 구성소의 이동된 접두어를 나타낸다. 복구는 이러한 상태들을 스택에서 팝하고 입력 토큰 스트림에서 해당 구성소의 나머지 부분을 삭제하고 잘못된 구성소를 나타내기 위해 가짜 비단말을 이동시킨 후 구문 분석기를 다시 시작할 것이다.

유닉스의 yacc/bison은 상향식 단계 수준 복구의 전형적인 예를 제공한다. 보통의 언어 토큰 외에 yacc/bison은 컴파일러 작성자가 문법 생산 우편의 어떤 곳에나 특수 토큰, error를 포함할 수 있게 해준다. 이런 문법으로부터 만든 구문 분석기는 구문 오류를 탐지하면 다음과 같은 작업을 수행한다.

1. 함수 yyerror를 호출한다. 이 함수는 컴파일러 작성자가 제공해야 한다. 보통 yyerror는 단순히 yacc/bison이 인자로 전달하는 메시지(예를 들어 "구문 분석 오류")를 출력한다.
2. error 토큰을 이동시킬 수 있는 상태를 찾을 때까지 구문 분석 스택에서 상태를 팝한다(그러한 상태가 없는 경우 구문 분석기는 종료된다).
3. error 토큰을 삽입한 후 이동시킨다.
4. 구문 분석기는 (error 후의) 새로운 문맥의 유효한 미리 보기 문자를 찾을 때까지 입력 스트림으로부터 토큰을 삭제한다.
5. 더 이상의 오류 보고를 일시적으로 비활성화한다.
6. 구문 분석을 재개한다.

error 토큰을 포함하는 생산과 연관된 의미 동작 루틴이 있는 경우 이러한 루틴은 다른 동작 루틴과 같은 방식으로 실행된다. 예를 들어 추가적인 오류 메시지를 출력하거나 심볼 테이블을 수정하거나 의미 처리를 수정하거나 대화형 도구(yacc/bison은 일괄 방식 컴파일러 외의 도구를 만드는 데에도 사용할 수 있다)에서 사용자에게 추가 입력을 위한 프롬프트를 띄우거나 코드 생성을 비활성화하는 등의 작업을 수행할 수 있다. 더 이상의 구문 오류를 비활성화하는 이유는 연속적인 오류를 무릅쓰기 전에 구문 분석을 재개할 수용 가능한 문맥을 실제로 찾게 보장하기 위함이다. yacc/bison은 이러한 실제 입력 토큰을 성공적으로 이동시킨 후 오류 보고를 다시 활성화한다. 의미

동작 루틴은 필요한 경우 내장 루틴 yyerrorok를 호출함으로써 좀 더 빨리 오류 메시지를 재활성화할 수 있다.

예 2.50

yacc/bison의 패닉 모드

그림 2.24의 상향식 문법을 사용해서 앞서 살펴본 계산기 언어에 대한 yacc/bison 구문 분석기를 만들 수 있다. 패닉 모드 복구를 위해 가장 가까운 문장으로 돌아갈 수 있다.

```
stmt → error
        {printf("현재 문장의 끝에서 구문 분석을 재개\n");}
```

중괄호 안에 작성된 의미 루틴은 구문 분석기가 *stmt* → error를 인식할 때 실행된다.[3] 구문 분석은 문장 뒤에 나올 수 있는 다음 토큰(계산기 언어의 경우 다음의 id, read, write, $$)에서 재개한다.

예 2.51

문장 종결자를 사용한 패닉 모드

오류 복구의 관점에서 볼 때 계산기 언어의 약점은 현재의 잘못된 문장도 추가적인 id를 포함할 수 있다는 것이다. 이러한 id 중 하나에서 구문 분석을 재개하면 바로 다른 오류를 보게 된다. 여러 실제 토큰을 이동시킬 때까지 오류 메시지를 비활성화함으로써 이러한 오류를 피할 수 있다. 모든 문장이 세미콜론으로 끝나는 언어에서는 다음과 같이 좀 더 안전하게 작성할 수 있다.

```
stmt → error ;
        {printf("현재 문장의 끝에서 구문 분석을 재개\n");}
```

예 2.52

yacc/bison의 단계 수준 복구

위의 두 가지 예에서는 error 기호를 우편의 시작 부분이 위치시켰지만 반드시 이렇게 할 필요는 없다. 예를 들어 괄호로 묶인 수식의 경우 오른쪽 괄호 다음부터 구문 분석의 재개를 시도할 수 있다. 오류가 이러한 수식 내부에서 발생하지 않는 한 오류를 발견할 때마다 현재 문장을 포기하게 결정할 수 있다. 그러면 다음과 같은 생산을 추가할 수 있다.

```
factor → ( error )
        {printf("parsing resumed at end of
                parenthesized expression\n");}
```

위 생산을 추가하면 그림 2.25의 CFSM의 상태 8에서는 error를 이동시키고 일부 토큰을 삭제한 후)를 이동시키고 factor를 인식하고 한 단계 위의 둘러싸는 수식의 구문 분석을 재개할 수 있다. 물론 잘못된 수식이 중첩된 괄호를 포함한다면 구문 분석기가 이들을 모두 건너뛰지 않을 수 있고 이로 인해 여전히 연속 오류가 발생할 수 있다.

주3. 이 구문은 yacc/bison이 실제로 받아들이는 것과 다르지만 이전 내용과의 일관성을 위해 사용했다.

yacc/bison은 LALR 구문 분석기를 생성하기 때문에 자동으로 문맥 특화 미리 보기를 채택하며, 대개 즉시 오류 탐지 문제를 겪지 않는다(완전 LR 구문 분석기는 약간 더 낫다). SLR 구문 분석기의 좋은 오류 복구 알고리즘에는 하향식 구문 분석기에서 사용했던 것과 동일한 수단이 필요하다. 명확히 말해서 shift 루틴이 실제 입력 토큰을 받아들일 때까지 모든 스택 변경 사항을 버퍼링하고 의미 동작 루틴을 호출한다. 실제 토큰을 받아들이기 전에 오류를 발견하면 스택 변경을 취소하고 의미 루틴에 대한 버퍼링된 호출을 취소한다. 그 다음 완전 LR 구문 분석기와 동일한 시점에 오류를 인식한 것처럼 동작한다.

✓ 확인문제

44. 구문 오류 복구는 왜 중요한가?
45. 연속 오류란 무엇인가?
46. 패닉 모드란 무엇인가? 패닉 모드의 주요 약점은 무엇인가?
47. 단계 수준 복구가 패닉 모드에 비해 가지는 장점은 무엇인가?
48. 즉시 오류 탐지 문제란 무엇이며 어떻게 해결할 수 있는가?
49. 문맥 특화 FOLLOW 집합이 유용할 수 있는 상황을 두 가지 설명하라.
50. 재귀 하강 구문 분석기에서 오류 복구를 위한 워드의 기법을 간략히 설명하라. 이 기법을 예외 기반 복구와 비교해보자.
51. 오류 생산이란 무엇인가? 양질의 범용 오류 복구 알고리즘을 포함하는 구문 분석기가 여전히 이러한 생산을 사용해서 이득을 볼 수 있는 이유는 무엇인가?
52. FMQ 알고리즘을 간단히 설명하라. 이 알고리즘은 어떤 면에서 최적인가?
53. 하향식 구문 분석기에서보다 상향식 구문 분석기에서 오류 복구가 더 어려운 이유는 무엇인가?
54. yacc/bison에서 사용하는 오류 복구 기법을 설명하라.