CS206A: Data Structures

2017-09-01

카이스트 전산학부 정지원(Otfried Cheong) 著
부경대학교 컴퓨터공학과 김태원 譯

2022년 6월 23일

계산기 구현하기

토큰화와 파싱

계산기는 어떻게 (단지 문자열에 불과한) 산술 표현식에서 표현식의 값을 취할 수 있는가?

이 물음은 프로그래밍 언어에 대해 인터프리터나 컴파일러 안에서 일어나는 일에 관한 아주 단순한 판본이다. (웹브라우저에 내장된 자바스크립트 인터프리터, 엑셀, 메이플, 매트랩의 매크로 언어나 파이썬의 인터프리터 자체를 예로 들 수 있다)

일반적으로 이 절차는 세 단계로 구성된다. **어휘 분석**(lexical analysis), **구문 분석**(syntactic analysis), **의미 분석**(semantic analysis).

어휘 분석

어휘 분석(**토큰화**(tokenization))은 (문자열 또는 텍스트 파일과 같은) 입력 텍스트를 최소 의미 단위로 쪼갠다. 이들 최소 의미 단위는 **토큰**이라고 부르며 어휘 분석기 또한 **토큰화 장치**(tokenizers)라고 부른다.

계산기를 위해서는 네 가지 유형의 토큰이 필요하다.

- 수 (실수, 예컨대 $3, 71, 23.45$),
- 식별자 (변수 이름, 가령 `i`, `j`, `a123`),
- 연산자 (`+`, `-`, `*`, `/`, $\cdots$),
- 입력 종결을 표기하기 위한 정지 토큰

토큰화 장치는 표현식의 의미나 구문에 관해 아무것도 모른다. 토큰화 장치는 (스페이스, 탭 문자, 개행과 같은) 화이트스페이스(white space)를 이용해 토큰이 끝나거나 시작하는 곳을 결정한다. 화이트스프에스가 표현식에서 아무런 의미를 지니지 않기에, 화이트스페이스에 대해서는 아무 토큰도 생성되지 않는다. (이와 유사하게 파이썬 인터프리터에서 주석은 토큰화 장치로 인해 제거된다)

가령 `123456`과 `123 456`은 다른 토큰 열(token sequence)을 내놓고 `a12`와 `a 12` 또한 그렇다. 반면 `1+2`와 `1 + 2`는 모두 같은 토큰 열을 내놓는다. (즉 수 1.0, 기호 +, 수 2.0을 내놓는다).

가령 아래 표현식이 주어졌을 때

```
(abc12+27 * 23.0(12abc34
```

토큰화 장치는 다음 열을 생산한다.

```
Token: Symbol(()
Token: Identifier(abc12)
Token: Symbol(+)
Token: Number(27.0000)
Token: Symbol(*)
Token: Number(23.0000)
Token: Symbol(()
Token: Number(12.0000)
Token: Identifier(abc34)
Token: Stop
```

문자열이 무의미하더라도 (아니면 적어도 의미 있는 표현식이 아니더라도) 토큰화 장치는 이를 자신의 규칙에 따라 토큰으로 쪼갤 적에 어려움이 없다는 점에 다시금 주의하라. 토큰화 장치는 문자열의 구문이나 의미에 관한 아무 이해를 지니지 않는다.

구문 분석

토큰화 장치로 인해 생산된 토큰 열은 구문 분석 (**파싱**(parsing)) 중에 검증되고 의미 있는 구조로 정돈된다. 파싱 과정에서는 구문 오류가 감지된다. 구문 분석기는 **파서**(paser)라고 부르기도 한다.

구문 분석은 종종 **구문 트리**(syntax tree) 또는 **파스 트리**(parse tree)라고 부르는 트리 구조로 토큰을 정돈한다. 우리의 경우 트리는 입력 표현식의 구조를 나타낸다. 가령 아래 표현식은

```
( 1 - 2 ) * 3.0 + 4 / a12
```

그림 1의 구문 트리로 파스된다. 파싱은 상이한 연산자의 우선 순위를 다루는 책무를 지닌다. `*`와 `/` 결합은 `+`와 `-` 결합보다 강하다.

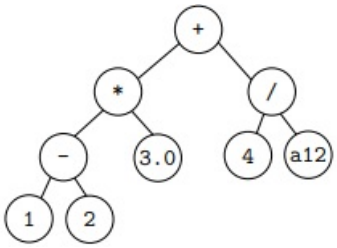

그림 1: 구문 트리

의미 분석

의미 분석은 파서에 의해 생산된 파스 트리에 대한 **의미**(meaning) 배정을 가리키는 일반적인 용어다. 우리의 계산기에는 의미 분석이 조금밖에 없다. 우리는 단순히 표현식의 값을 계산할 따름이다.

컴파일러에서는 타입 확인, 변수 영역 확인을 비롯한 의미론 분석이 이루어지며, 최종적으로 컴파일러가 코드를 생성된다.

재귀적 감소 파싱

우리가 파서를 만들기 위해 사용하는 기법은 **재귀적 감소 파싱**(recursive descent parsing)이라고 부른다. 우선 우리가 파스하고자 하는 표현식의 구문 도표를 그려 본다. 그림 2를 보자.

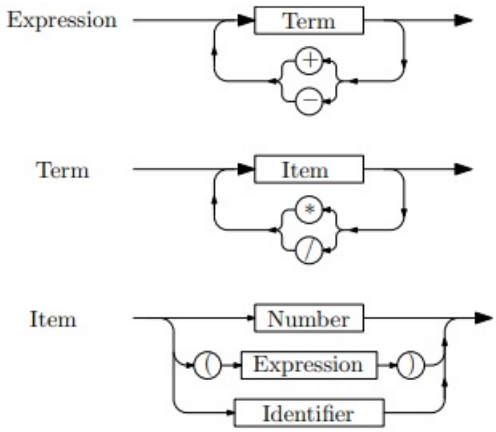

그림 2: 간단한 표현식의 구문 도표

구문 도표는 하나의 표현식에 나타날 수 있는 세 가지 상이한 구문 요소를 설명한다. 다음과 같다. **표현식**(expression), **항**(Term), **인수**(factor). "표현식"은 "항"의 열로서 덧셈과 뺄셈에 의해 분리된다. "항"은 "항목"(Items)의 열로서 곱셈과 나눗셈에 의해 분리된다. 마지막으로 "항목"은 수이거나, 식별자(즉, 변수 이름)이거나, 괄호로 묶인 임의의 표현식이다.

주어진 표현식을 파스하기 위해 해야 할 일이라고는 구문 도표상의 화살표를 따라가는 것이 전부다. 각 단계별로 분기를 결정하기 위해 토큰 열에서 다음 토큰을 확인하는 것만으로 충분하다.

구문 도표를 지니는 이상, 구현은 아주 쉽다. 이제 각 구문 요소마다 하나의 메소드를 작성한다. `parse_expression`, `parse_term`, `parse_item`. 각 메소드는 각자 한 가지 유형의 구문 요소를 파스하는 책무을 지닌다. 또 요소의 값을 (한 숫자를) 반환하며 파싱 오류가 나타날 경우 예외를 던진다.

아래는 일례로 `parse_term`을 위한 코드다. (`calculator1.py`에서 예제 코드를 보라)

```
1   def parse expression(tok):
2       result = parse_term(tok)
3       t = tok[0]
4       while t.isSymbol("+") or t.isSymbol("-"):
5           tok.pop(0)
6           rhs = parse_term(tok)
7           if t.isSymbol("+"): result = result + rhs
8           else: result = result - rhs
9           t = tok[0]
10      return result
```

파서 확장하기

위 단순한 구문 도표는 단항(單項, unary) 뺄셈 연산자를 포함하지 않는다. 또 `-a`나 `-(3+5)` 같은 음수나 음의 표현식을 작성할 수 없다. 이에 우리는 한발짝 나아가 지수를 위해 `^` 연산자를 추가하고자 한다.

우리는 여기서 조금 주의를 기울여야 한다. $-$와 $/$가 **좌결합**인 반면 (이는 $a-b-c-d=((a-b)-c)-d$를 의미한다) 지수 연산은 **우결합**이다. (수학에서는 $2^{3^2}=2^9=512$이며, 이와 유사하게 우리는 `2^3^2=2^(3^2)=2^9=512`를 취하고자 한다)

그림 3의 구문 도표는 단항 덧셈 및 뺄셈과 우결합 지수 연산자를 올바르게 구현한다. 이 구문 도표의 구현을 `calculator2.py`의 예제 코드에서 확인할 수 있다.

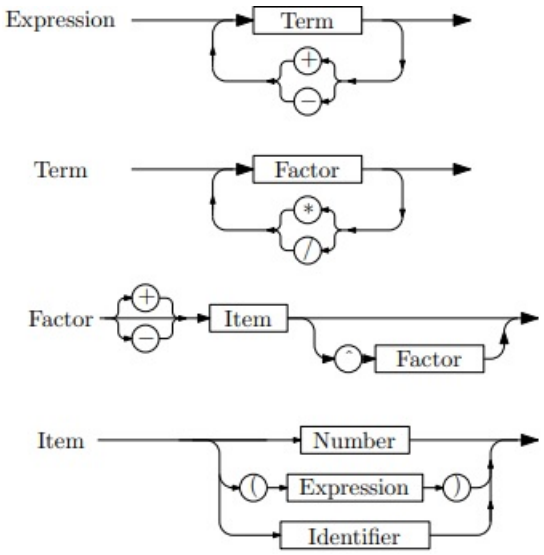

그림 3: 지수 및 단항 연산이 가능한 확장 구문 도표