

Problems on Basic Data Structures

Koscom Algorithm Lecture

신승우

Wednesday 5th September, 2018

- 1 Huffman Encoding/Decoding
- 2 Expression Parsing
 - Abstract Syntax Tree
 - Recursive Descent Algorithm

Huffman Encoding

Huffman Encoding은 무손실¹ 압축 알고리즘 중 하나입니다. 압축 알고리즘의 구현은 곧 압축 알고리즘 및 복원 알고리즘의 구현을 말합니다. 여기서는 binary tree를 이용한 문자열 압축 알고리즘을 살펴보겠습니다.

¹Jpeg나 mp3 같은 것은 손실압축의 예시입니다.

Huffman Encoding의 아이디어

예를 들어서, 문자열 abbcccccddddeeeeeeffffff가 있다고 합시다. 이 때, 원칙적으로 문자열을 인코딩하기 위해서는 문자당 5비트가 필요하므로² 21글자를 인코딩하기 위하여 110비트가 필요합니다.

그런데, 사실 구분 가능하기만 하다면 모든 문자에 같은 크기의 비트를 할당할 필요가 없습니다. 만약, 더 많이 나오는 글자에 적은 수의 비트를 할당하고, 적게 나오는 글자에 더 긴 비트를 할당한다면, 글자를 인코딩할 때 필요한 비트 수를 줄일 수 있을 것입니다. 그렇다면 어떻게 각 글자를 인코딩하는 것을 만들 수 있을까요?

²현실에서는 더 필요하지만, 여기서는 단순히 영어 소문자만 인코딩한다고 생각합니다.

Huffman Coding의 과정

Huffman Coding은 priority queue 하나와 binary tree 하나를 이용하여 이루어집니다.

- 먼저, 압축하기 위해서 주어진 문자열을 분석하여 빈도를 우선순위로 하여, 나오는 모든 알파벳을 우선순위 큐에 push합니다.
- 큐에 하나의 원소만 남을 때까지 다음 과정을 반복합니다.
 - 원소 2개를 pop 합니다. 이 때, 우선순위가 낮은³ 원소부터 pop해야 합니다.
 - 원소 2개를 children으로, root로는 두 원소의 우선순위를 합한 정수를 가지는 binary tree를 만듭니다.
 - 위에서 만든 binary tree를 priority는 root에 저장된 정수와 같게 하여 priority queue에 push합니다.
- 큐의 마지막 원소를 pop합니다. 이 때 나오는 원소는 binary tree여야 합니다.
- 위에서 pop한 트리를 이용하여, 각 leaf마다 주소를 할당합니다. 주소의 할당은 root의 주소에, 오른쪽 child는 0을, 왼쪽 child는 1을 붙여 할당합니다.
- leaf에 저장된 알파벳과, 할당된 주소를 이용해서 원 string을 encode 하고 그 후 트리와 같이 리턴합니다.

Huffman Coding의 과정 예시

위 알고리즘을 따라서 abbccccc를 인코딩하는 과정의 일부를 살펴보겠습니다.

Table: Huffman Encoding의 예시

queue	x, y	binary tree
(a,1), (b,2), (c,4)	(a,1), (b,2)	$t = \text{Tree}(3, [(a,1), (b,2)])$
(t, 4), (c,3)	queue 원소 다	$s = \text{Tree}(7, [(c,3), (t,4)])$
(s, 7)		* is binary!

What is Parsing?

Parsing

특정 형식에 맞는 문자열을 원하는 데이터구조 형태로 만드는 것

- 특정 형식에 맞는 문자열 : Formal Grammar
- 원하는 데이터구조 : Abstract Syntax Tree(AST)

수식의 Grammar

수식의 grammar를 살펴보면 다음과 같다.

- 1) `expr -> part (binary part)*`
- 2) `part -> num \ | "(" expr ")" | unary part ;`
- 3) `binary -> "^" \ | "*" \ | "/" \ | "+" \ | "-" ;`
- 4) `unary -> "-" ;`
- 5) `num -> r"[a-zA-Z]+" | r"[1-9][0-9]*\.[0-9]*" ;`

1)에서 `(binary part)*`는 `binary part`의 유한한 반복을 뜻한다. 0번 반복하더라도 상관없다.

Syntax Check using Grammar

위 문법에 기반하여 $x * (y + z)$ 가 수식의 문법에 맞는 문장인지 알아보자.

Table: $x * (y + z)$ 체크

current string	rule	result
$x*(y+z)$	1	is x part?, is * binary?, is $(y+z)$ part?
x	2, 3	x is num! / num is part!
*	3	* is binary!
$(y+z)$	3	is $y+z$ expr?
y, z	same to x	y,z is num! / num is part!
+	3	+ is binary!

Syntax Check using Grammar

이번에는 $x * (y +$ 가 수식의 문법에 맞는 문장인지 알아보자.

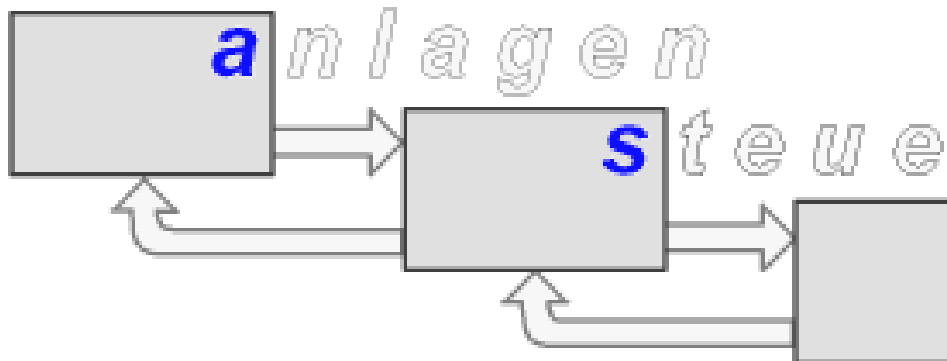
Table: $x * (y +$ 체크

current string	rule	result
$x * (y +$	1	is x part?, is $*$ binary?, is $(y +$ part?
x	2, 3	x is num! / num is part!
$*$	3	$*$ is binary!
$(y +$	3	is $y +$ expr?
y	same to x	y is num! / num is part!
$+$	3	$+$ is binary!
eos	None	expected part, return eos : SyntaxError

이처럼, 위 문법에서의 규칙들을 순차적으로 적용하는 것으로 특정 문자열이 그 문법에 맞게 작성되었는지를 알 수 있다. 이럴 때 그 특정 문자열은 그 문법에 의해서 생성되었다고 한다.

Abstract Syntax Tree

구문의 문법 구조를 반영하여 트리 형태로 나타낸 것을 abstract syntax tree라고 한다. 예를 들어서, 위 $x * (y + z)$ 의 경우 다음과 같은 트리로 생각할 수 있다.



이때까지 문법과 그 문법을 이용하여 파싱한 결과물이 무엇인지 간략하게 살펴보았다. 이제 본격적으로 어떤 식으로 구현하는지 살펴 보고자 한다.

먼저, 일반적인 파서의 구조를 살펴보자. 일반적으로 파서는 두 가지 함수로 이루어져 있다.

- tokenizer : string to tokens
- parser : tokens to AST

여기서, token이란 최소한의 의미를 가지는 문자열을 말한다. 문법에서 "으로 둘러싸인 문자열이나 그 문자열이 나타내는 정규표현식과 일치하는 문자열을 뜻한다.

Implementing Tokenizer

Tokenizer의 경우, 다음의 과정을 통해서 만들 수 있다.

- 문법에서 token의 패턴을 추출한다.
- input string에 대해서, input string == "" 가 될 때까지
 - 각 패턴에 대해서 check re.match(pattern, input string)
 - 만약 맞으면 input string에서 pattern과 일치하는 부분을 yield
 - 일치하는 부분을 제외한 나머지를 input string으로 업데이트
 - 만약 모든 패턴에 대해서 맞지 않으면 SyntaxError 리턴

Shunting-Yard Algorithm

이제 얻어진 token들을 이용하여 abstract syntax tree를 만드는 알고리즘을 생각해 보자. 먼저, 스택 2개를 생각한다.

- operand stack
- operation stack

이 때, operation stack에서는 필요한 operand들이 파싱이 끝날 때까지 operation을 pop하지 않는다. 또한, operation들 중 더 우선순위가 높은 operation이 항상 스택의 위에 오도록 유지한다. 예를 들어서, operation stack에 /가 있을 때, +가 들어오기 전에 /를 pop하고 필요한 처치를 한다. 즉, 그 때 operand stack의 가장 위에 있는 token 2개를 pop⁴ 하여 Tree('/', tok1, tok2) 형태로 만드는 것이다. 만약 그 때 operand stack에 2개의 token이 없다면 에러를 리턴한다. operator가 스택에서 pop될 때마다 tree가 하나씩 생성되며, 이를 다시 operand stack에 push한다. 파싱이 되는 예시 2개와, 되지 않는 예시 1개를 들어서 살펴보고 이를 구현해보겠다.

⁴만약 operation이 unary라면 1개. operation에 맞는 갯수를 리턴하면 된다.

Table: $x * y + z$ 파싱 예제

tokens	operand	op	action
x, *, y, +, z			operand.push(x)
, y, +, z	x		compare(, None)
, y, +, z	x	*	operation.push(*)
y, +, z	x	*	operand.push(y)
+, z	y, x	*	compare(*, +)
+, z	Tree(*, [x, y])		operator.pop()
z	Tree(*, [x, y])	+	operation.push(+)
	z, Tree(*, [x, y])	+	operand.push(z)
	Tree(+, [z, Tree(*, [x, y])])		operation.pop()

Table: $x * (y + z)$ 파싱 예제

tokens	operand	op	action
x, *, (, y, +, z,)			operand.push(x)
, (, y, +, z,)	x		operation.push()
(, y, +, z,)	x	*	parse(find_match(token
	Tree(+, [y,z]) x	*	operand.push(parse(..
	Tree(+, [y,z]) x	*	operator.pop()
	Tree(*, [Tree(+, [y,z]), x])		operator.pop()

여기서 find_match 함수를 사용하는데, 이는 tokens에서 어떤 index의 괄호와 쌍을 이루는 괄호를 찾는 것이다. 이를 통해서 괄호 안의 식을 우선적으로 처리할 수 있다.

Table: $y+$ 파싱 예제

tokens	operand	op	action
y, +			operand.push(y)
+	y		operator.push(+)
	y	+	operator.pop()
	y	+	operand.pop();operand.pop()
	y	+	raise SyntaxError

Implementing Parser

위에서 알고리즘의 개요를 살펴보았다. 이제 본 알고리즘을 구현해볼 것이다. 본격적인 구현 전에, 필요한 변수들과 함수들을 구현하자.

- precedence : operator들 간 우선순위를 저장한 변수
- find_match 함수 : 맞는 괄호 찾아주기
- compare 함수 : operator 간 우선순위 비교

이후, 스택 두 개를 만들어 위 알고리즘을 구현한다.

Implementing Parser : 실습

파싱하고자 하는 expression의 문법은 다음과 같습니다.

- 1) `expr -> part (binary part)*;`
- 2) `part -> num | "(" expr ")";`
- 3) `binary -> "+" | "-" | "*" | "/" | "^";`
- 4) `num -> r"[a-zA-Z]+" | r"[1-9][0-9]*\.[0-9]*;`

실습에서는

- operator 순서는 $^$, $(*, /)$, $(+, -)$ 순입니다.
- **unary**는 고려하지 않습니다.
- parser은 문법에 맞게 syntax tree 형태로 식을 파싱하여 반환하면 됩니다.