C++ 연산자 오버로딩을 활용한
새로운 연산자 기호 프로그래밍 방법

팀명: 최강정컴

202365536 장영실

202465536 난설헌

202565536 우리애

지도교수: 덴이수

제출일: 2025년 5월 15일

부산대학교

목차

제1장 서론

프로그래밍 언어를 처음 배울 때 프로그래머 대부분을 당황하게 했던 것은 아마 연산자가 아닐까 생각한다. BASIC이나 C의 대입 연산자(assignment operator) =는 수학에서 말하는 "같다"라는 의미가 아니기 때문이다. BASIC에서는 `Let` 문(대입문)에 대한 표기법의 일부로서 =를 사용하지만[3] 사실 `Let` 명령어를 생략할 수 있으므로 연산 기호 =는 C의 대입 연산자와 다름없이 사용된다. 이는 = 기호에 새로운 의미를 부여했기 때문인데, 이렇게 특정한 기능을 수행하는 기호(혹은 기호의 나열)를 연산자(operator)라고 한다.[1]

1.1 연구 배경 및 문제점

그런데 어떤 프로그래밍 언어에서 연산자를 새로 만들 수 있을까? 사실 특정 프로그래밍 언어를 사용한다면 그 언어에서 제공하는 연산자를 사용해야 하며, 제공하고 있는 원시 연산자 집합이 전부인 경우가 많다. Haskell[10]이나 Scheme[14]과 같은 일부 언어는 연산자를 확장하는 것을 허용하고 있지만, 연산자를 새로 정의하는 기능은 주요 언어에서 채택되지 않고 있다. 왜냐하면, 새로운 연산자를 정의하지 않아도 함수를 통해 새로운 연산을 정의할 수 있기 때문이다. 그러나 함수보다는 연산자 표기가 간단한 경우가 많다.

Haskell은 지연 함수형 언어로서 Scheme과 달리 타입을 지원하며 언어의 최신 기능을 다수 지원하고 있다. Haskell 로고는 그림 1.1과 같다.

그림 1.1: Haskell 로고

대신 주요 언어에서는 이미 존재하는 연산자의 의미를 확장할 수 있도록 허용하고 있다.

[1] 사실 연산자는 피연산자에 적용한 후 값을 반환할 수 있어야 한다. 그런 의미에서 BASIC의 대입 연산 기호는 진정한 연산자라고 간주하지 않는 견해도 있다.

이를 연산자 오버로딩(operator overloading)이라고 부르는데, 연산자 오버로딩은 연산자 겹지정이라고 부르기도 한다. 연산자 오버로딩을 이용하면 새로 정의한 사용자 정의 타입을 원시 자료형처럼 사용할 수 있다. 예컨대 사용자 정의 자료형 `Matrix`를 정의했다면, `Matrix`형 객체 `m1`, `m2`에 대하여 `m1 * m2` 등을 행렬 곱으로 정의할 수 있다. `Matrix`는 원시 자료형이 아니지만, 연산자 오버로딩을 통해 이미 존재하는 곱셈 연산자 `*`를 `Matrix` 객체에 대해 정의할 수 있으며, 이를 통해 새로운 자료형 `Matrix`를 마치 원시 자료형인 것처럼 사용할 수 있다.

그런데도 새로운 연산자에 대한 요구는 계속 발생할 수 있다. 기본 자료형인 `int`나 `double` 등에 대해서는 기본 연산자로 충분할지 모르지만 조금 복잡한 사용자 정의 자료형에 대해서는 더 복잡한 연산자가 필요한 경우가 많기 때문이다. 일례로 벡터나 행렬 연산에 강한 MATLAB의 경우에는 원소별 곱 연산자 `.*`가 정의되어 있다[2]. Haskell의 모나드 연산자(monadic operator) 중 바인드(bind) 연산자 >==는 순차 적용을 강제함으로써 순수한 함수형 방법으로 입출력 연산을 구현할 수 있도록 하고 있다[8]. Prolog의 연산자 -->는 한정절 문법(definite clause grammar)을 전통적인 문맥 무관 문법(context free grammar)처럼 작성할 수 있도록 한다[11]. 하지만 연산자 오버로딩만 지원하는 언어에서는 새로운 연산자를 정의할 수 없으므로 앞서 언급한 새로운 연산자는 별도의 함수나 메소드로 정의할 수밖에 없다.

1.2 연구 목표

이 논문에서는 연산자 오버로딩만 지원하는 언어에서 새로운 연산자를 구현하는 방법에 대해 논한다. 사실 완전히 새로운 연산자를 구현하는 것은 아니고 연산자 오버로딩만을 이용하여 제한적으로나마 새로운 연산자를 프로그래밍 기법으로 구현하는 방법이다. 이 방법은 정적 타입 검사(static type checking)를 수행하는 C++와 같은 언어는 물론, "크게 베어 물기 원칙"을 통해 어휘 분석 및 구문 분석을 수행하는 대부분의 언어에 적용할 수 있다. 어휘 분석기의 특징을 이용하기는 하지만 컴파일러나 인터프리터를 수정할 필요가 없으며, 프로그래밍을 통해 연산자를 정의할 수 있으므로 활용도가 높을 것으로 생각된다.

제2장 관련 연구

2.1 사용자 정의 연산자

Scheme은 모든 연산자를 함수로 간주하기 때문에 당연히 새로운 연산자를 정의할 수 있다. Scheme에서 연산자 정의란 함수 정의와 다를 바가 없다. 다만 연산자 이름에 사용할 수 없는 문자를 사용해서는 안 된다. 예컨대 다음과 같이 문자열 접합 함수를 정의할 수 있다.

```
1 (define (++ s1 s2)
2   (string-append s1 s2))
```

하지만 코드의 1행에서 ++ 대신 +"을 사용하면 오류가 발생한다. 큰따옴표는 문자열 리터럴을 나타내는 용도로 한정되어 있기 때문이다.

Prolog[4]는 새로운 연산자 기호로 펑터(functor)를 정의할 수는 있다. 다만, 펑터는 데이터 구조를 생성하는 용도로만 사용할 수 있으므로 실제 연산을 수행하는 용도로는 사용할 수 없다. 그러나 Prolog는 패턴 매칭(pattern matching)을 활용한 반증(refutation)을 주요 계산 모델로 삼고 있으므로, 데이터 구조라고 하더라도 적극적으로 연산에 사용될 수 있다. Prolog의 이러한 특징을 객체지향 언어(object-oriented language) 관점에서 해석하면, 생성자에 대해서만 제한적으로 새로운 연산자 기호를 사용할 수 있는 것으로 간주할 수 있다.

최신 함수형 언어(functional language)인 Haskell[10]은 함수 형태로 새로운 연산자를 정의할 수 있다. 다음은 두 문자열 사이에 공백을 넣어 접합하는 연산자 +++를 Haskell에서 정의한 것이다.

```
1 (+++) :: String -> String -> String
2 s +++ t = s ++ " " ++ t
```

Haskell에서는 이항 연산자를 함수로 정의할 수 있으므로, 이처럼 새로운 연산자를 정의하는 것이 자연스러우며 보편적으로 사용되고 있다. 앞서 언급한 것처럼, Haskell에서는 모나드 연산자(monadic operators)나 적용 펑터(applicative functors)에서 새로운 연산자를 적극적으로 활용하고 있다.

Heinlein은 사용자 정의 연산자가 가능하도록 C++를 확장한 C+++를 제안하였다[7]. C+++에서는 새로운 연산자 기호를 `new` 키워드를 통해 정의할 수 있도록 하고 있으며, 결합

방향과 우선순위는 연산자 기호 정의 시 명시하도록 하고 있다. C+++는 언어를 확장해야만 한다는 제한이 있지만, 본 연구의 방법은 언어 확장 필요 없이 프로그래밍 방법을 통해 새로운 연산자 기호를 정의할 수 있다.

2.2 연산자 오버로딩

C나 C++, Java에서는 새로운 연산자를 정의할 수 없지만, C++에서는 연산자의 의미를 추가로 정의할 수 있도록 하고 있다. 다시 말해서 이미 언어에서 제공하고 있는 연산자의 의미를 확장하는 것이 가능하며, 이를 연산자 오버로딩이라고 부른다. 구문 분석 관점에서 보면 연산자 오버로딩이란 특별한 함수 호출 기능을 지원하는 것으로 생각할 수 있다.

그러나 C++처럼 연산자 오버로딩만을 허용하는 언어에서는 새로운 연산자를 정의할 수 없다. C++에서는 기존에 존재하는 연산자만 오버로딩할 수 있으며, 연산자 중에서도 멤버 연산자(dot operator)나 멤버 참조 연산자(member access operator), 범위 지정 연산자(scope resolution operator) 등은 오버로딩할 수 없다. 원칙적으로는 불가능함에도 불구하고, 이 논문에서는 연산자 오버로딩 및 사용자 정의 클래스를 이용하여 새로운 연산자 기호를 정의하고 사용하는 방법에 설명한다.

Java는 C++보다 늦게 설계된 언어임에도 불구하고 연산자 오버로딩 기능을 지원하지 않는다. 대신 메소드 오버로딩만을 지원하고 있는데, 이는 시그너처 매칭(signature matching)을 통한 메소드의 동적 호출(dynamic invocation)을 허용하는 언어에서 더 자연스러운 선택이라고 볼 수 있다. C++도 메소드의 동적 호출을 허용하긴 하지만 가상 함수(virtual function)로 선언된 메소드에 대해서만 동적 호출을 허용한다.

2011년에 발표된 표준 C++에서는 사용자 정의 리터럴(user-defined literal)을 정의할 수 있도록 하고 있는데[15] 여기에도 연산자 오버로딩과 유사한 방식이 사용된다. 사용자 정의 리터럴 기능을 활용하면 `11_km`, `1200_krw` 등을 정의할 수 있다. 사용자 정의 리터럴을 위해서 `operator ""`가 사용되는데 `11_km`를 `double` 타입의 상수로 간주하고 싶다면 `double operator "" _km(unsinged long long);` 함수를 정의해야 한다. 실제로 `""`라는 연산자는 없기 때문에 이 방식은 엄밀히 말해서 연산자 오버로딩이라고 할 수는 없다. 다만 연산자 오버로딩을 활용한 기법이라고 볼 수 있을 것이다.

제3장 새로운 연산자의 필요성

먼저, 구체적인 예를 통해 새로운 연산자가 필요한 상황을 생각해 보자. 앞서 설명한 것처럼 두 문자열을 공백으로 연결하는 연산자가 존재한다면 문자열 사용 시 편리할 수 있다. 또한, Python과 같이 거듭제곱 연산자가 존재한다면 라이브러리 함수를 사용하는 대신 간편하게 연산자를 활용할 수 있다. 공백을 활용한 문자열 접합 연산자 사용 예를 보면 다음과 같다.

```
int main()
{
    String hello("Hello"), world("World");
    cout << hello +++ world << endl;
}
```

이 프로그램에서 `String`은 새로운 연산자가 정의된 클래스이다. 연산자에 새로운 의미를 부여하기 위해 새로운 클래스를 정의해야 하는 것은 연산자 오버로딩의 경우에도 마찬가지이다.

또, 거듭제곱 연산자를 새로 정의한 경우 다음과 같이 사용할 수 있다.

```
int main()
{
    Long a(2), b(10);
    cout << a ** b << endl;
}
```

이 프로그램에서 `Long`은 역시 새로운 연산자를 정의한 클래스이다. 새로운 연산자를 정의한 `Long` 클래스 덕분에 마치 Python에서처럼 C++에서도 거듭제곱 을 간단하게 계산할 수 있다.

앞서 제시한 두 프로그램을 보면 모두 작성력(writability)[13]이 증가된 것을 확인할 수 있다. 특히 후자의 경우, Python에 익숙한 프로그래머의 경우에는 어떠한 거부감도 없이 사용할 수 있을 것이다. 특히 Python의 거듭제곱 연산자에 익숙한 프로그래머가 C++의 비슷한 기능을 묻는 글이 지속적으로 게시되는 것을 보면[16, 5], 이러한 새로운 연산자에 대한 사용자의 요구가 얼마나 높은지 알 수 있다.

문자열 접합 연산자의 경우에는, 해당 연산자가 제공되지 않는다면 다음과 같이 복잡하게 코드를 작성해야 한다.

```
    cout << hello + " " + world << endl;
```

이 경우에도 연산자 +++를 사용하여 코드를 더 간단히 작성할 수 있으며, 결과적으로 프로그램 작성력을 높일 수 있음을 알 수 있다.

연산자 오버로딩과 가독성(readability)에 관해서는 논란의 여지가 있지만, Parrish는 연산자 오버로딩이 클라이언트 코드의 가독성을 높이는 데 기여한다는 것을 논한 바 있다[12]. 이 논문에서 제시하는 방법은 연산자 오버로딩을 확장함으로써 클라이언트 코드의 작성력을 더 높일 수 있으며, 일부 프로그램에 대해서는 읽기 쉬워졌음을 알 수 있다. 다만 정량적인 수치로 분석한 것이 아니므로 절대적으로 그렇다고 보기는 어렵다. 하지만 새로운 연산자를 정의하는 방법이 있다면 프로그래머에게는 이를 활용할 수 있는 여지가 추가로 생기는 것이므로, 결국 프로그래머의 코드 작성 자유도를 높일 수 있을 것이리라 생각된다.

그렇다면 C++에서 어떻게 새로운 연산자를 정의하고 사용할 수 있을까? 과연 C++에서 Python의 거듭제곱 연산자와 같은 연산자를 사용할 수 있을까? 다음 절에서 연산자를 새로 정의하는 아이디어에 대해 상세히 살펴본다.

제4장 C++에서 새로운 연산자 프로그래밍 방법

4.1 크게 베어 물기 정책

새로운 연산자 정의 없이 정수 타입 long에 대하여 다음과 같은 코드를 작성해 보자.

```
int main()
{
    long a(2), b(10);
    cout << a ** b << endl;
}
```

이 코드는 을 출력하기 위해 Python 프로그래머가 작성한 C++ 코드이다. 그러나 연산자 ** 가 정의되어 있지 않으므로 이를 GNU C 컴파일러(g++ 9.3.0 기준)로 컴파일하면 다음과 같은 오류 메시지가 출력된다.

```
error: invalid type argument of unary '*' (have 'long int')
```

Microsoft Visual C++ 등 다른 컴파일러를 이용해도 이와 유사한 오류 메시지를 얻을 수 있다.

이와 같은 오류 메시지의 원인은 컴파일러 어휘 분석기(lexical analyzer)의 텍스트 처리 방식 때문이다. 어휘 분석기는 a ** b를 a * (* b)로 해석한다. 그 원인은 두 가지인데, 하나는 단항 연산자의 우선순위가 높기 때문이며, 다른 하나는 "크게 베어 물기(maximal munch)" 정책[6] 때문이다.

크게 베어 물기 정책이란, 각 상태에서 파악할 수 있는 접두사 중 최대 길이로 인식할 수 있는 만큼 읽어 들이는 어휘 분석기의 정책으로서 최장 부문자열 원칙[9], 최장 접두사[1] 원칙이라고 부르기도 한다. 다시 말해서 입력 텍스트의 시작 부분이 어휘 분석기의 여러 패턴에 일치할 경우, 이 중에서 가장 길게 일치하는 시작 부분을 선택한다는 정책이다. 위 예의 a ** b에서 a를 읽은 상태, 즉 ** b만 남은 상태에서 C++ 언어의 정의상 인식할 수 있는 최장 접두사 *뿐이며 따라서 a *를 읽고 * b는 다음 입력으로 남겨 둔다. 그러므로 "단항 연산자 '*'에 대해 잘못된 인수 오류"가 발생하는 것이다. 여기서 잘못된 인수란 long int 타입의 b를 의미한다.

바로 이 오류가 새로운 연산자를 만들 수 있는 아이디어의 바탕이 된다. 만약 `* b`가 `long int` 타입이 아닌 새로운 자료형의 객체를 반환하도록 연산자를 오버로딩한다면, 또 필요에 따라서 이항 연산자 *의 의미도 새로 정의한다면 새로운 연산자를 정의할 수도 있지 않을까? 구체적으로, 이항 연산자 *의 왼쪽에 나타날 수 있는 객체가 `Long` 타입이라고 하고 오른쪽에 나타날 수 있는 객체가 `Expo` 타입이라고 하면, 다음과 같은 프로토타입으로 이항 연산자 *를 오버로딩할 수 있으며 이를 통해 위와 같은 오류를 해결함과 동시에 새로운 연산을 정의할 수 있다.

```
Long operator *(Long left, Expo right);
```

이 과정에서 비록 새로운 타입을 정의해야 하므로 구현 코드가 늘어나긴 하지만 결과적으로 클라이언트 코드는 매우 간단해진다. 클라이언트 코드와 구현 코드의 비중 조정(trade-offs)은 연산자 오버로딩을 사용할 때에도 흔히 발생하는 일이다.

크게 베어 물기 정책이 항상 오류를 발생시키는 것은 아니다. 크게 베어 물기 정책이 오류를 발생시키지 않아서 놀랍게 여겨지는 경우도 있는데, 대표적인 예로 스택 오버플로(Stack Overflow)에 게시된 질문[17]의 코드 일부를 인용하면 다음과 같다.

```
    while (x --> 0) // x goes to 0
    {
        printf("%d ", x);
    }
```

위 코드에서 `x --> 0` 부분은 마치 "변수 x가 0이 될 때까지"라고 해석되는 듯하지만, 사실 변수 x에 후치 감소 연산자를 적용한 결과가 0보다 큰지 검사하는 조건식이다. 3장에서 논의한 +++ 형태에 대해서도 유사한 질문[16]이 게시된 바 있다.

4.2 새로운 연산자 정의 방법

앞서 언급한 것처럼 이항 연산자와 단항 연산자가 겹쳐져 사용되는 경우에 새로운 연산 기호처럼 사용할 수 있다. 이러한 조합의 유형으로는 다음과 같은 두 가지 유형이 있다.

1. 후치 단항 연산자 및 이항 연산자 조합
2. 이항 연산자 및 전치 단항 연산자 조합

두 가지 유형을 바탕으로 연산자를 프로그래밍하는 알고리즘 대신 유클리드 알고리즘을 기술하면 알고리즘 1과 같다.

알고리즘 1: 유클리드 호제법

입력: 두 양의 정수 a, b

출력: a와 b의 최대공약수

$r \leftarrow a \bmod b$;

while $r \neq 0$ **do**

    // r이 0이면 다른 수가 최대공약수

    $a \leftarrow b$;

    $b \leftarrow r$;

    $r \leftarrow a \bmod b$;

// 최대공약수는 b

return b;

다시 원래 문제로 돌아가서 이 중에서 조금 더 쉽게 구현할 수 있는 첫 번째 조합의 예를 살펴보자. 첫 번째 유형의 조합 예로 연산자 +++를 구현해 보려고 한다. "크게 베어 물기" 정책에 따라 이 연산자는 후치 단항 연산자 ++와 이항 연산자 +의 조합으로 구현할 수 있다. 공백을 사이에 두고 결합하는 문자열을 구현하기 위해, 표준 문자열 클래스 `string`을 확장하여 `String`으로 구현하고자 한다. 이 구현 코드를 기술하면 코드 4.1과 같다.

코드 4.1: 연산자 +++의 구현 코드

```
1  #include <iostream>
2  using namespace std;
3
4  class String: public string
5  {
6  public:
7      String(string &&s): string(s) {}
8      String(const char s[]): string(s) {}
9      String operator ++(int) {
10         return *this + " ";
11     }
12 };
13
14 int main()
15 {
16     String hello("Hello"), world("World");
17     cout << hello +++ world << endl;
18 }
```

코드 4.1의 9에서 11행을 보면 클래스 `String`에 후치 증가 연산자 ++가 정의되어 있는 것을 볼 수 있다. 매개변수 `int`는 후치임을 나타내기 위한 것으로 실제로 사용되지 않는다. 따라서 매개변수 이름도 생략하였다. `String` 객체에 대해 정의된 후치 증가 연산자는 문자열의 맨 끝에 공백 문자를 추가하는 기능을 수행한다. 따라서 이항 덧셈 연산자 +와 더불어 공백

문자를 이용한 접합 연산자 +++를 구성할 수 있는 것이다. 연산자 +++의 사용 예는 17행에 예시되어 있다.

두 번째 유형의 조합 예로 연산자 **을 구현해 보자. "크게 베어 물기" 정책에 따라 이 연산자는 이항 연산자 *와 전치 단항 연산자 *의 조합으로 구현할 수 있다. 연산자 **을 거듭제곱 연산자로 구현하기 위해, 클래스 Long을 정의하고자 한다. 앞서 제시한 코드 4.1과 달리 이 경우에 크게 달라지는 것은 Long 앞에 놓인 전치 단항 연산자를 구현하는 작업인데, 전치 단항 연산자를 통해 별도의 Expo 타입 객체로 변환해야 하기 때문이다. 이 구현 코드를 기술하면 코드 4.2와 같다.

코드 4.2: 연산자 **의 구현 코드

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  class Long {
6  private:
7      long _val;
8  public:
9      class Expo {
10     private:
11         long _val;
12         friend class Long;
13         Expo(long n): _val(n) {}
14     };
15     Long(long n): _val(n) {}
16     Long operator *(const Long &b) {
17         return this->_val * b._val;
18     }
19     Long operator *(Long::Expo b) {
20         return Long((long)pow(this->_val, b._val));
21     }
22     Long::Expo operator *() {
23         return Long::Expo(_val);
24     }
25     long val() {
26         return _val;
27     }
28 };
29
30 ostream &operator <<(ostream &out, Long a) {
31     return out << a.val();
32 }
33
34 int main()
35 {
36     Long a(2), b(10);
37     cout << a ** b << endl;
38 }
```

먼저, 코드 4.2의 37행을 보면 2의 10승을 연산자 **을 통해 계산하여 수행하는 것을 볼 수

있다. 30행에서 32행은 단순히 Long 형 객체를 ostream에 출력하기 위해 정의한 것으로, 사용자 정의 클래스를 표준 출력 스트림에 출력하기 위한 일상적인 방법이다.

연산자 **을 구현하는 코드는 19행에서 24행 사이에 정의되어 있는데, 19행에서 21행에서는 특별한 이항 연산자 *를 오버로딩하고 있으며 22행에서 24행은 단항 연산자 *를 오버로딩하고 있다. 22행에서 확인할 수 있듯이 단항 연산자는 Long::Expo 형 객체를 반환하고, 19행에서 볼 수 있는 것처럼 이항 연산자 *는 우측 피연산자가 Long::Expo 형인 경우에 거듭제곱 연산을 수행한다. 물론 우측 피연산자가 Long 형인 경우에는 일반적인 곱셈 연산자를 수행한다.

제5장 정리 및 토의

이 절에서는 앞서 구현한 방법의 몇 가지 주의사항과 한계에 대해 논한다. 먼저 코드 4.2의 9행에서 14행에 정의된 `Expo` 클래스는 `Long`의 외부에 정의해도 된다. 그러나 `Long`의 내부에 정의하는 편이 캡슐화 관점에서 더 이득이다. `Expo` 클래스는 항상 `Long` 클래스와 함께 동작하게 되어 있기 때문이다. 또한, `Expo` 클래스를 `Long` 클래스 내부에 정의함으로써 클래스 `Long`의 외부에 `Expo` 클래스가 드러나지 않도록 할 수 있는데, 즉 전용 클래스로 사용할 수 있도록 하고 있는데, 결과적으로 이는 이름 충돌(name conflict)로부터 프로그래머를 자유롭게 하는 긍정적 효과를 가져오게 된다.

연산자 오버로딩을 이용하여 새로운 연산자를 프로그래밍함에 따라 치러야 하는 불편함도 물론 존재한다. `Long` 클래스의 경우 `long` 데이터를 `Long` 객체로 만드는 것은 `Long` 생성자로 간단히 처리할 수 있지만, `Long`을 `long`으로 변환할 때에는 어쩔 수 없는 불편함이 존재한다. 이를 위해서는 형 변환 연산자를 `Long` 클래스에 추가로 정의해야 한다. 이렇게 정의하더라도 `long` 데이터를 꺼내기 위해 형 변환 연산자를 별도로 호출해야 하는 불편함은 여전히 존재한다.

이상에서 설명한 새로운 연산자는 후치 단항 연산자 및 이항 연산자 조합이나, 이항 연산자 및 전치 단항 연산자의 조합으로만 가능하다. 따라서 만들 수 있는 연산자의 최대 개수는 표 5.1과 같이 정리할 수 있다. 표 5.1에서 볼 수 있는 것처럼 연산자 조합으로 만들 수 있는 연산자 개수는 이론상 512개에 달하지만 실제로는 연산자의 대칭성, 방향성 등을 고려할 때 유용한 연산자는 이보다 훨씬 적을 것이라고 예상된다. 512개는 128개와 384개의 합으로 산출된 값인데, 128개는 후치 단항 연산자(4개)와 이항 연산자(32개)의 조합 개수이며 384개는 이항 연산자(32개)와 전치 단항 연산자(12개)의 조합 개수이다.

그러나 `new`나 `delete` 등의 연산자는 실제로 연산자 프로그래밍에 사용될 것으로 예상되지 않는다. 따라서 연산자의 대칭성, 방향성을 고려하면 실제 유용한 연산자는 앞서 예시한 `+++`와 `**`, `-->`를 비롯하여 `---`, `+=+`, `-=-`, `!=!`, `<--`, `--->` 등으로 한정될 것으로 생각되며, 따라서 512개에 훨씬 못 미칠 것으로 예상된다.

표 5.1: 연산자 조합으로 새로 만들 수 있는 연산자 최대 개수

Types	Left Operator	Right Operator	Number
postfix unary + binary	`++ -- () []`	`-> ->* * / % + - << >> < <= >= == != & ^ | && || = += -= *= /= %= <<= >>= &= ^= != ,`	128
binary + prefix unary	`-> ->* * / % + - << >> < <= >= == != & ^ | && || = += -= *= /= %= <<= >>= &= ^= != ,`	`++ -- + - ! ~ * & new new[] delete delete[]`	384
Total			512

제6장 결론

프로그래밍 언어를 사용하는 중요한 이유 중 하나로 가독성과 더불어 작성력을 꼽을 수 있다. 기계어나 어셈블리어로 프로그램을 한 번이라도 작성해 보았다면 작성력의 중요성은 재삼 논할 필요가 없다. 기계어나 어셈블리어에서는 간단한 연산을 매우 단순한 기계어로 모두 풀어서 써 주어야 한다. 이 보고서에서 소개한 새로운 연산자 프로그래밍 방법은 작성력을 높이는 데 기여할 수 있다. 또한, 연산자를 새로 정의하는 방법을 이용해서 언어를 확장하다 보면 프로그래밍 자체에 대한 흥미도 높일 수 있을 것으로 판단된다.

이 보고서에서 소개한 방법도 언어의 의미에 합당하도록 사용하는 것이 좋다. 이는 연산자 오버로딩의 경우에도 마찬가지인데, 연산자의 대칭성이나 방향성을 고려하지 않고 연산자를 정의하면 오히려 가독성을 저하시킬 우려가 있기 때문이다. 의미를 유추할 수 있도록 사용하면 새로운 연산자 정의 방법의 장점이 더욱 두드러질 것이다. 예컨대 +++는 더하기나 접합 연산으로, **는 거듭제곱 연산으로 사용하는 것이 바람직하다. 새로 정의한 연산자 기호의 의미를 유추하기 힘든 경우에는 반드시 이를 명시적으로 문서화해야 할 것이다.

참고문헌

[1] Alfred V. Aho, Monica S. Lam, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2007.

[2] Zachary Battles and Lloyd N. Trefethen. An extension of MATLAB to continuous functions and operators. *SIAM Journal on Scientific Computing*, 25(5):1743–1770, 2004.

[3] Piraye Bayman and Richard E. Mayer. A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Communications of the ACM*, 26(9):677–679, 1983.

[4] Ivan Bratko. *Prolog: programming for artificial intelligence*. Pearson Education, 4th edition, 2011.

[5] J. Burgess. Raising to powers in C++ vs. Python, 2015. Online document. Available at: `https://stackoverflow.com/questions/34518653/raising-to-powers-in-c-vs-python/34518772`. Checked on 2021-12-30.

[6] Roderic Geoffrey Galton Cattell. *Formalization and automatic derivation of code generators*. PhD thesis, Carnegie-Mellon University., 1978.

[7] Christian Heinlein. C+++: User-defined operator symbols in C++. In *Informatik 2004, Informatik verbindet, Band 2, Beiträge der 34. Jahrestagung der Gesellschaft für Informatik eV (GI)*, pages 459–468. Gesellschaft für Informatik eV, 2004.

[8] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2nd edition, 2016.

[9] Kenneth C. Louden. *Compiler construction: principles and practice*. PWS Publishing Co., 1997.

[10] Simon Marlow. Haskell 2010 language report, 2010. Online Document. Available at: https://www.haskell.org/onlinereport/haskell2010. Checked on 2021-12-30.

[11] Falco Nogatz, Dietmar Seipel, and Salvador Abreu. Definite clause grammars with parse trees: Extension for Prolog. In *Proceedings of the 8th Symposium on Languages, Applications and Technologies (SLATE 2019)*, pages 7:1–7:14. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019.

[12] A. Parrish, David Cordes, R. Borie, and S. Edara. Illustrating client and implementation readability tradeoffs in Ada and C++. *Software: Practice and Experience*, 26(7):799–814, 1996.

[13] Robert W. Sebesta. *Concepts of programming languages*. Pearson Education, 2016.

[14] Michael Sperber, R Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. Revised^6 report on the algorithmic language Scheme. *Journal of Functional Programming*, 19(S1):1–301, 2009.

[15] Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 4th edition, 2013.

[16] user2015064. Why doesn't C++ have a power operator?, 2013. Online Document. Available at: https://stackoverflow.com/questions/14626960/why-doesnt-c-have-a-power-operator. Checked on 2021-12-30.

[17] user595985. What does the operation c=a+++b mean?, 2011. Online Document. Available at: https://stackoverflow.com/questions/7485088/what-does-the-operation-c-ab-mean. Checked on 2021-12-30.