Programming Language

Assignment #5

- Copy 시 소스 제공자와 카피 당사자 공히 **해당 과제 전체 0점 처리.**
- 반드시 자기 자신의 생각과 글로 작성할 것. 다른 곳에서 자료 인용 시 출처 반드시 기재 **(미기재 시 50% 감점)**
- 한글/영문 상관 없음.

7.2. State your own arguments for and against allowing mixed-mode arithmetic expressions.

찬성: 혼합형 산술식을 허용하지 않으면, 명시적 변환을 사용하지 않고는 서로 다른 두 타입 간의 연산이 불가능해진다. 이런 언어단의 제약은 프로그래밍 언어가 가지는 유연성의 손실을 초래하고, 표현력, 단순성, 작성력, 판독성 등에 좋지 않은 영향을 미칠 수 있다. 가령 C 기반 언어들에서 아래와 같은 구문을 혼합형 산술식을 허용하지 않고 작성한다고 해보자.

```
char a = 1;
short b = 2;
int c = a + b;
```

혼합형 산술식이 허용되지 않는 상황이라면, 오직 같은 타입 피연산자끼리의 산술 연산이 가능할 것이므로 위의 3번째 줄의 변수 c의 대입 구문은 다음과 같이 표현할 수 있다.

```
int c = (int) (((short) a) + b);
int c = (int) (a + (char) b); // b의 축소 변환으로 정보 손실이 일어날 수 있음
int c = ((int) a) + ((int) b);
```

위 코드에서 대입 구문의 본질은 a와 b의 덧셈 결과이다. 그런데 혼합형 산술식이 허용된 상황에서는 단순히 a + b로 표기할 수 있는 것을, 추가적인 명시적 변환 구문을 사용해서 표현해야 하므로 표현력이 떨어진다. 또한 사용자 입장에서, 범위가 서로 다른 정수형 변수끼리의 연산을 추가적인 변환 구문과 같이 처리해야 하는 것을 요구하므로, 복잡성을 증대시키고 이는 단순성을 저해한다. 그리고 a + b와 같은 단순한 연산이 아니라, 여러 개의 변수에 대한 산술 연산을 시행하

는 표현식을 작성할 때에는 위의 3가지 타입변환 예시보다 많은 명시적 변환 케이스가 존재할 것이고, 이는 프로그래밍 언어의 작성력과 판독성에 좋지 않은 영향을 끼칠 것이다.

반대: 혼합형 산술식을 허용하는 경우, 타입 검사의 이점을 감소시키고 이로 인하여 사용자의 연산 실수를 검출하지 못하게 되는 단점을 가진다. 예를 들어 다음과 같은 C 기반 언어의 코드를 보면

```
short a;
double b, c, d;
...
d = b * a;
```

"d = b * c;"로 의도되었던 것을 사용자의 실수로 "d = b * a;" 작성되었다고 가정하면, 혼합형 산술식을 허용하지 않는 경우에는 이를 타입 오류로서 검출하여 사용자에게 고지할 수 있는 반면, 혼합형 산술식을 허용하는 경우에는 컴파일러가 이를 오류로 검출하지 못하게 된다. 또한 여러 타입에 대해 창발적으로 나타날 수 있는 묵시적 변환을 모두 구현해야 하므로 컴파일러 구현 비용과 복잡성이 증가할 수 있다. 그리고 이러한 묵시적 변환 과정에서 사용자가 예측하지 못한, 축소 변환이 이루어지는 경우 프로그램이 예상된 결과와 전혀 다른 작동을 할 수 있어 위험해진다. 이는 확장 변환에서도(가령 자바에서는 int를 float으로 확장 변환할 때, int의 정수 표현 정확도(9자리 수준)와, float의 십진수 표현 정확도(7자리 수준))이 달라 2자리의 십진수 정확도 손실이 생긴다) 마찬가지로 발생할 수 있다.

7.3. Do you think the elimination of overloaded operators in your favorite language would be beneficial? Why or why not?

중복 연산자를 제거하는 것은 별로 좋지 않은 방향이라고 생각한다. 중복 연산자는 프로그래밍 언어에서 기본적으로 지원되지 않거나 설계되지 않은 기능을 만들 때 함수 표현이 아닌, 직관적이고 친숙한 연산자로 기능의 개념들을 나타낼 수 있게 해주는 중요한 수단이다. 이런 강력한 표현 수단의 남용은 언어의 판독성(예상한 연산자의 결과와 전혀 다른 결과가 나오게 할 수 있기 때문에)을 저해시킬 수 있기 때문에 사용자가 이런 기능을 분별력 있게 사용할 수 있게끔 하는 언어적인 설계가 꼭 필요하다고 생각하지만 제거가 필요하다고는 생각치 않는다.

왜냐하면 중복 연산자가 저해시키는 판독성*보다, 이를 이용하여 얻을 수 있는 프로그래밍 언어의 표현력과 편의성이 월등하게 크기 때문이다.

*. 중복 연산자가 구현된 대다수의 프로그래밍 언어들은 중요한 연산자(C++에서는 범위 지정 연산자(::)와 접근 연산자(.) 연산자를 중복 처리할 수 없다)를 중복 처리할 수 없게 구현되고, 기본 연산에 대한 중복 처리가 불가능하다. 따라서 거의 모든 중복 연산자의 사용은 피연산자가 사용자 정의 클래스의 객체인 경우에만 한정되고, 이 규칙(사용자 정의 클래스의 객체가 피연산자로 사용되는 경우에만 중복 연산자로 처리됨)을 통해 사용자는 어렵지 않게, 해당 연산이 중복 연산자로 구현되었음을 유추할 수 있기 때문에 큰 판독성 저해를 가져오지 않는다.

7.4. Would it be a good idea to eliminate all operator precedence rules and require parentheses to show the desired precedence in expressions? Why or why not?

실제로 APL에서는 일반적인 프로그래밍 언어의 연산자 우선 순위를 따르지 않고, 함수와 연산자에 대해서 범위 개념을 도입하여 구문을 처리하게 된다. 이를 나타내면 다음과 같다.

```
1 ÷ 2 ⌊ 3 × 4 – 5
⍝ 오른 쪽부터 식을 실행시키기 시작함
⍝ 4 – 5를 (뺄셈 이항 스칼라 함수) 실행시킴 (¯1)
      ¯0.3333333333
1 ÷ 2 ⌊ 3 × ¯1
⍝ 3 × ¯1을 (곱셈 이항 스칼라 함수) 실행시킴 (¯3)
      ¯0.3333333333
1 ÷ 2 ⌊ ¯3
⍝ ⌊ ¯3을 (바닥 단항 스칼라 함수) 실행시킴 (¯3)
      ¯0.3333333333
1 ÷ ¯3
⍝ 1 ÷ ¯3을 (나눗셈 이항 스칼라 함수) 실행시킴 (¯0.3333333333)
      ¯0.3333333333
```

연산자 우선 순위 규칙을 제거하는 것은 프로그램의 작성자나 판독자가 우선순위나 결합 규칙을 기억할 필요가 없게 되므로, 단순성을 증가시킬 수 있으나 별도의 처리 규칙(범위 개념, 괄호 강제 등)을 사용자에게 숙지할 것을 요구한다. 이는 사용자가 프로그래밍 언어를 배우고 사용하는 데에 있어서 추가적인 비용

이라고 볼 수 있다. 특히 괄호를 사용하여 식에서 요구된 우선순위를 표현하는 경우 복잡한 산술 연산식을 표현하려고 할 때에는 표현식의 판독성을 심각하게 저해할 것이다. 따라서 모든 연산자의 우선순위를 제거하고, 괄호를 사용하여 식에서 요구된 우선순위를 표현하는 것 보다 최대한 우리에게 가장 익숙한 수학에서의 연산자 우선 순위를 최대한 따르되, 사용자가 우선 순위를 바꾸어야 할 연산에 대해서 괄호를 적용하도록 하는 것이 더 낫다고 생각한다.

7.5. Should C's assigning operations (for example, +=) be included in other languages (that do not already have them)? Why or why not?

프로그래밍 언어에 복합 배정 연산자(+=, -=, *=, /= 등)를 포함하면 목표 연산 후 대입하는 과정을 더 짧은 구문으로 처리할 수 있다. 이는 구문 축약 뿐만 아니라 처리 속도에도 영향을 미칠 수 있다. 예를 들어 "a = a + 2"를 처리하기 위해서 컴파일러나 인터프리터는 RValue(a+2) 값을 평가하고, LValue(a)에 대입할 것이다. 이 과정에서 RValue의 구문을 분석하여 평가해야 하는 비용이 "a += 2"에서는 단항 리터럴 RValue로 나타나기 때문에 감소할 것이다. 또한 연산자 중복을 허용하는 언어에서, 배정 연산을 추가적으로 언어에서 지원하면 연산자 중복으로 사용할 수 있는 표현 방법이 늘어나게 되는 장점도 생기게 된다. 이러한 장점들을 고려했을 때, 복합 배정 연산자는 프로그래밍 언어에 포함되는 것이 좋다.

7.6. Should C's single-operand assignment forms (for example, ++count) be included in other languages (that do not already have them)? Why or why not?

C의 단일 피연산자 배정문(++, --)은 C언어의 개발자인 데니스 리치의 회고에 따르면 C 컴파일러가 코드의 의도를 분석하여, 최적화를 잘 해내지 못하던 C언어의 초창기, 직접적인 코드의 의도를 컴파일러에 전달하기를 원하던 시기에 발명되었다고 한다.

당시 C 언어의 증분/감소 연산자는 다음과 같은 평배정문으로 작성한 증가(+1) 연산에 대한 기계어 코드를

```
LOAD MEM
LOAD 1
ADD
```

```
STORE MEM
```

```
INC MEM
```

위와 같은 기계어 코드 한 줄로 축약하여 표현하는 성능 상의 이점이 있었다. 현대의 C 컴파일러는 최적화 과정에서 이를 지능적으로 분석하여 처리할 수 있게 되어, 이제는 구문 상의 편의로 남게 되었다. 실제로 단항 피연산자 배정문은 평배정문(A=A+1, A=A-1)으로 모두 표현이 가능하다. C++ 언어의 개발자인 비야네 스트롭스트룹과 "Annotated C++ Reference Manual" 집필에 참여한 마가렛 앨리스(스트롭스트룹과 공동 저자)는 실제로 C++에 대해서 언급하면서, 단항 덧셈을 역사적인 사건이라고 부르고 이 연산을 쓸모 없는 것으로 지적하기도 했다. 현대의 파이썬과 같은 프로그래밍 언어는 "실수를 유발하는 구조를 피하는"(전위, 후위 증감 연산자는 많은 실수를 유발하는 것으로 알려짐) 철학과 함께 그 구조 상(파이썬에서 변수를 변경하는 방법은 변수를 재할당하는 방법밖에 없음) 단일 피연산자 배정문을 포함하지 않았다. 이러한 점들을 고려해보았을 때, 현대의 프로그래밍 언어에서 단일 피연산자 배정문을 추가하는 것은 표현력 향상의 측면에서만 큰 의미가 있을 것이다. 따라서 나는 이것이 모든 언어에 일률적으로 적용할 수 있는 성질의 것이 아니라고 생각한다. 파이썬의 사례처럼 언어의 철학과 구조에 따라서 표현력 향상과 부작용이 언어 방향과 부합하지 않는 경우에는 단일 피연산자 배정문이 해당 프로그래밍 언어에 추가되어서는 안되며, 그렇지 않은 경우에는 추가해도 된다고 생각한다.

8.2. Python uses indentation to specify compound statements. Give an example in support of this statement.

파이썬은 동일하게 들여쓰기된 모든 문장들을 복합문에 포함한다. 다음 예제와 같이 표현된다(선택문과 반복문에서의 복합문 표현).

```
if x != y :
    x = y
    print("x is " + x)

for i in range(10)
    s = i ** 2
```

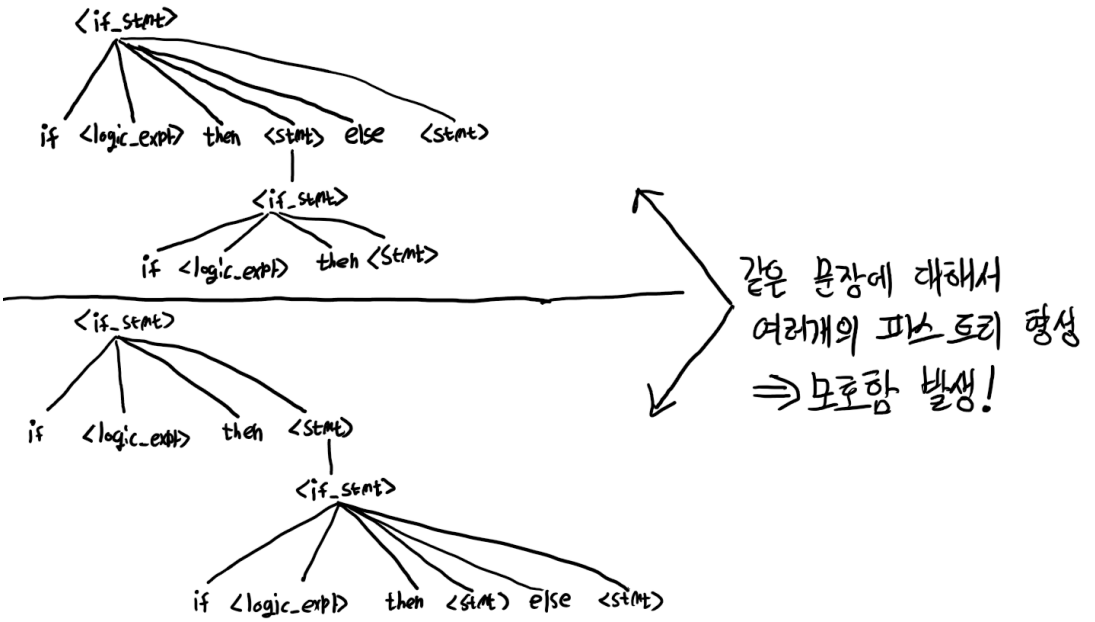
```
print(s)
```

8.3. How can a straightforward grammar for a two-way selector statement lead to the problem of syntactic ambiguity? Give an example.

Ada의 if-then-else문에 대한 BNF 규칙을 다음과 같이 간략히 기술하자

```
<if_stmt> → if <logic_expr> then <stmt>
          | if <logic_expr> then <stmt> else <stmt>
<stmt> → <if_stmt>
```

"if <logic_expr> then if <logic_expr> then <stmt> else <stmt>"와 같은 경우 then 절의 <stmt>에 <if_stmt>로서 선택문이 중첩될 때, else 절이 내부의 then절의 <stmt>에 중첩되는 <if_stmt>에 연관되는 것인지, 최외곽 if문의 <stmt>에 연관되는 것인지 불분명하여 구문적 모호성을 야기한다.



이를 파스 트리로 구현하면 한 문장에 대해서 위와 같이 두 개의 파스 트리가 생기게 된다. (한 문장에 대해서 여러개의 파스 트리가 형성될 수 있는 경우 해당 문법은 모호한 것임) 이런 모호성을 해결하기 위해서는 아래와 같이 또 다른 논터미널을 통해 추상화하여 해결할 수 있다.

```
<stmt> → <matched> | <unmatched>
<matched> → if <logic_expr> then <matched> else <matched>
| 임의의 if가 아닌 문장
```

```
<unmatched> → if <logic_expr> then <stmt>
I if <logic_expr> then <matched> else <unmatched>
```

또는 Java에서와 같이, else 절은 항상 가장 가까이 위치한 짝을 갖지 않은 이전의 then 절과 짝이 맺어지는 정적 의미적 규칙을 이용하거나 Perl과 같이 모든 then 절과 else 절이 복합문일 것을 요구하는 방법을 이용하여 해결할 수 있다.

```
// 정적 의미론 규칙을 이용하여 모호성을 해결하는 자바의 if문
if (sum == 0) {
        if (count == 0)
                result = 0;
} else
        result = 1;
// #1

if (sum == 0)
        if (count == 0)
                result = 0;
        else
                result = 1;
// #2
```

```
// 모든 then 절과 else 절이 복합문일 것을 요구하는 Perl의 if문
if (sum == 0) {
        if (count == 0) {
                result = 0;
        }
} else {
        result = 1;
}
// #3 (#1과 동일한 의미)


if (sum == 0) {
        if(count == 0) {
                result = 0;
```

```
        } else {
                result = 1;
        }
}
// #4 (#2과 동일한 의미)
```

8.6. In C language, a control statement can be implemented using a nested *if else*, as well as by using a *switch* statement. Make a list of differences in the implementation of a nested *if else* and a *switch* statement. Also suggest under what circumstances the *if else* control structure is used and in which condition the switch case is used.

switch

- 다중 선택문 구조의 대표적인 구현
- switch value에 따라 각각 일치하는 case의 세그먼트를 실행함
- 중첩된 if-else와 완전히 유사한 동일한 구조로 구현된 경우도 있음 (Ruby의 case-when-end 문)
- goto의 절제된 표현인 break문을 이용하여 현재 실행 케이스를 종료할 수 있음
- 한 개 이상의 세그먼트에 대한 묵시적 실행을 용인하는 언어(C, C++, Java)와 "goto case"문으로 이를 명시적으로 제어해야 하는 언어(C#)가 있음
- 언어에 따라 switch value로 정수, 문자, 문자열, 열거형 등의 값을 지원함

중첩된 if-else

- 2방향 선택문 구조를 중첩하여 다중 선택의 효과를 낼 수 있게한 구현
- 부울 결과가 나오는 식을 연쇄적으로 평가하며 첫번째로 참이 된 구문을 실행함
- "else if" 등으로 이루어진 이전 if문의 else와 이후 if 문의 if 식별자를 간결하게 나타내서 낮은 판독성을 완화하기 위해 "else if"를 "elsif", "elif" 등으로 확장한 구현 사례가 존재(Perl, Python)

switch문(다중 선택문)은 임의의 개수의 문장이나 문장 그룹들 중에서 한 개를 선택해서(switch value를 통해) 실행할 때 사용되고, if-else(2방향 선택문) 제어 구조는 두개 또는 그 이상의 문장(중첩되는 경우)에 대해서 한 개를 선택하여 실행할 때 사용된다. C언어에서는 switch case label이 일정한 값을 가지는 경우 switch table을 만들어서 switch value를 table에서 상수 시간만에 점프해야 할 case문의 위치로 변환한 뒤 이동하는 반면, if-else문은 순차적으로 모든 부울 식 중 참이 되는 식에 다다를 때까지 비교 연산을 계속한다. 따라서 성능과 기능 측면에서 switch 문은 주어진 변수의 여러 값에 따라서 다른 실행 경로를 제어해야 할 때 적합하고, if-else문(중첩 포함)은 선택이 부울 식에 기반해서 이루어져야 할 때 효과적이다.

8.7. Read the article by Ledgard and Macotty (1975) to find whether the features and utility of all the control structures that have been proposed is worth their inclusion in languages.

Henry F. Ledgard and Michael Marcotty, **"A genealogy of control structures"** Communication of the ACM, Volume 18, Number 11, November 1975, pages 629-639

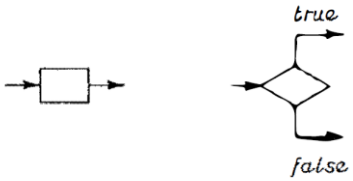
위 논문은 여러 제어 구조에 대한 수많은 이론적 결과를 검토하고, 실제적인 함의를 탐구한다. 루프와 분기를 포함한 여러 제어 구조(if-then, if-then-else, while-do, repeat-until, case statements 등)를 제시하고 이를 환원성과 동등성 개념을 도입하여 간단한 구조로 기술하기 위해 변환하고, 이러한 방법들을 통해 제어 구조의 효율성과 표현력 등에 대해 분석한다. 이 논문에서 분석 결과는 다익스트라, 밀리 등이 수행한 연구(goto를 제외하는 구조적 프로그래밍을 주장)와 일치함을 확인하고, C언어의 개발자인 커누스(goto를 포함하는 구조적 프로그래밍을 주장)가 제시한 goto 사용의 효율적인 사례들에 대한 제어 구조적인 측면에서 뒷받침할 논거를 찾지 못했다고 기술한다.

8.8. Read the article by Böhm and Jacopini (1966) and summarize the theoretical result that proves that sequence, selection, and pretest logical loops are absolutely required

to express computations.

Corrado Böhm and Giuseppe Jacopini, Flow Diagrams, **"Turing Machines And Languages With Only Two Formation Rules"**, Communications of the ACM, Volume 9, Number 5, pages. 366-371

위 논문에서는 순서도를 명시적인 방법으로 일반화한다. 순서도는 Functional, Predicative boxes로 나타낼 수 있고, Σ(α, β, γ, a, b, c)와 같이 박스 매개변수를 가지는 함수 형태로 나타낼 수 있다.



FIGS. 1–2. Functional and predicative boxes

FIGS. 5–6. Diagrams of Π

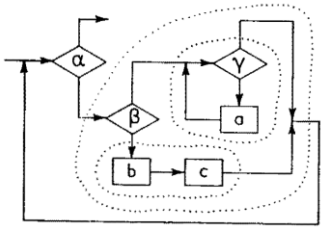
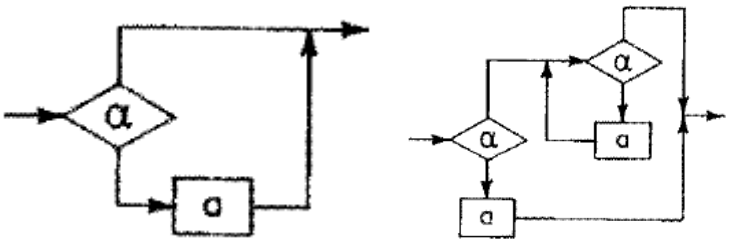


FIG. 3. Diagram of Σ

그런 다음 한 순서도가 다른 순서도의 결합으로 표현될 수 있음과, 동치 구조를 가질 수 있음을 주장한다.



<동치 구조의 두 순서도>

그리고 추가적인 Functional box인 T, B(순서쌍으로 변환), K(두번째 요소 추출)와 순서쌍에서 첫번째 요소의 값이 참인지 거짓인지 검사하는 ω Predicate box를 추가하여 논문에서 제시된 순서도에 대해서 이를 적용한 일반화와 변환을 제시한다

그리고 이 일반화 방법과, 변환 논리를 이용하여 Functional box a, b, c, ...,와 Predicates box α, β, γ를 가지는 모든 순서도가 a, b, c, ..., α, β, γ, ..., T, F, K와 괄

호 ("(", ")")로 구성된 문자열로 표현될 수 있음을 증명한다.

최종적으로 논문은 첫번째 부분에서

- X: x의 집합
- ψ: X에 정의된 단항 술어(predicates) 집합 (α, β, ...)
- O: X에서 X로 가는 사상 집합 (a, b, ...)
- $\mathcal{D}(\psi, O)$: ψ∪O에 속하는 상자를 포함하는 순서도를 통해 기술 가능한 X에서 X로의 모든 사상 집합
- Y: 귀납적으로 정의된 y 객체에 대한 집합 $(X \subset Y,\quad y \in Y \Rightarrow (t, y), (f, y) \in Y)$
- ω: Y에 의해서 정의되는 술어 논리 (ω(t, x)=t, ω(f, x)=f)

$$\mathcal{D}(\psi, O) \subset \varepsilon(\omega, O \cup \psi \cup \{T, F, K\})$$

임을 증명하고 이전 논문(Corrado Böhm. On a family of Turing machines and the related programming language. ICC bulletin, vol. 3 (1964), pp. 185–194)에서 증명한 내용을 통해 위의 수식을 튜링 기계에서 적용함으로써 어떤 프로그램(프로그램)이든 순서, 선택, 반복의 제어 구조만으로 변환될 수 있음을 증명한다.

Reference

3장 구문과 의미론, 7장 식과 배정문, 8장 문장-수준 제어 구조 - 프로그래밍 언어론 제 10판 (Robert W. Sebesta)

APL Wiki – Scalar function

APL (Wekipedia)

Why does Python not have a ++ operator? (Quora)

Why are there no ++ and -- operators in Python? (Stack Overflow)

Behaviour of increment and decrement operators in Python (Stack Overflow)

Henry F. Ledgard and Michael Marcotty, "A genealogy of control structures" Communication of the ACM, Volume 18, Number 11, November 1975, pages 629-639

Corrado Böhm and Giuseppe Jacopini, Flow Diagrams, "Turing Machines And Languages With Only Two Formation Rules", Communications of the ACM, Volume 9, Number 5, pages. 366-371

Structured programming (Wikipedia)

Böhm-Jacopini theorem (Stack Overflow)

How to prove the structured program theorem? (Stack Exchange - Computer Science)

<The End of the Assignment>