## 2-8 지역 변수와 예외처리

> 언어로서 동작하기 시작한 Streem에, 이번에는 2가지 기능을 추가한다. 첫번째는 지역 변수에서, 중첩을 허용할까의 문제, 그리고 ‘클로저’를 어떻게 구현할가의 설계상 과제가 있다. 두번째 예외처리에서는 에러의 무시에 의한 계속적 처리에 대하여 검토한다.

2-7절에서 겨우 프로그래밍 언어로서 형태를 갖추기 시작한 Streem에 이번에는 ‘지역변수’ 와 ‘예외처리’ 를 추가한다.

### ■ 지역 변수

우선 지역 변수에 대해 생각해 보자. 현대의 프로그래밍 언어에서는 상식 중의 상식이라고 할 지역 변수이지만, 예전에는 그렇지도 않았었다.

#### 30년전으로 타임 슬립

예를들어 30년전으로 타임 슬립을 했다고 하자. 당시, 취미 프로그래머로서 일반적이었던 언어는 BASIC으로, 당시의 BASIC에는 지역 변수가 없었다. 지역 변수가 존재하지 않은 프로그래밍은 상상하기 어렵겠지만, 모든 변수는 어디에서든 변경이 가능했기 때문에, 값이 어디에서 변경되는지 파악하는게 정말 어려웠다.

당시 젊은 프로그래머들은, 모두 BASIC으로 프로그래밍을 하고 있어서 게임 등 상당한 규모의 프로그램을 개발하는 경우도 있었다. 지금 생각해보면 디버그가 가능했다는게 정말 기적이었다.

#### 지역 변수가 없는 세계

BASIC에서는 서브루틴을

```

gosub 4000

```

과 같이 행번호로 호출하였다. 인수와 리턴값 같은 것이 없었기 때문에, 전역 변수를 사용해 값을 전달하였다. 어떤 변수에 값을 할당하여 서브루틴을 호출하면, 계산 결과가 다른 변수에 할당되는, 그런 형태였다. 당연히 정보 은폐 등의 고급 기술들은 불가능했다. 처리과정을 함수 등으로 묶어 추상화 하는 것도 불가능했다.

더우기 함수가 없기 때문에 함수의 재귀호출도 불가능했다.

#### 지역 변수의 도입

지역 변수를 최초로 만든 프로그래밍 언어는 무엇일까, 이에 대하여 별도로 조사를 해 본적은 없지만, 최초 도입한 언어가 Algol임은 틀림없을 것 같다.

Algol은 FORTRAN처럼 그 언어 자체가 현재는 남아 있지 않지만 여기에서 도입된 아이디어가 나중에 많은 언어에 영향을 주고 있다.

지역 변수가 있으면, 일련의 처리를 은폐하여 함수로서 제공할 수 있다. 또한 재귀호출도 가능하게 된다(그림 1). 아니, 배열을 이용하여 자기 부담으로 스택을 제공하면 전역 변수를 사용하여 억지로 재귀를 수행 할 수는 있을 것이다. 사실 지역 변수가 없었던 시절의 FORTRAN에서는 이와같은 방법으로 프로그래밍했다고 들은 적이 있다. 하지만 이렇게 하는 것은 정말 말리고 싶다.

```

def fact(n)

if n == 1

1

else

n \* fact(n-1) end

end

```

<center>

(그림 1) 재귀호출

</center>

#### 지역 변수의 구현

지역 변수의 구현은 그렇게 어려운 것은 아니다. 구현기법으로서는, 예를 들어 함수의 컴파일시에 지역 변수 별로 개별 인덱스를 할당해 둔다. 함수 실행시에는 배열을 준비해서, 그 인덱스의 옵셋된 위치에 값을 할당하는 방법 등이 고려되고 있다. 이 함수 실행별로 사용되는 배열을 일반적으로 스택이라고 부른다.

주의할 점은, 함수의 실행별 사용되는 지역 변수의 갯수를 파악해 두고 스택이 넘치지 않도록 하는 것이다. 스택 오버 플로우는 보안 문제에도 연결되는 중대한 결함이다.

그러면, Streem처리 시스템에 지역 변수기능을 추가해 보자. 우선은, 구문해석의 부분에서 변수로의 대입과 참조하는 부분을 변경한다.

#### Streem에 구현한다

현재 Streem의 구문해석기는 대입 부분에서는 NODE\_LET이라고 하는 노드가, 참조 부분에는 NODE\_IDENT라고 하는 노드가 생성된다.

지금까지 NODE\_LET은 무시하고, NODE\_IDENT는 전역변수에 엑세스했지만, 이것을 좀 손을 볼것이다. 우선은 NODE\_LET에서는, 지역 변수를 초기화 한다. 앞에서 지역 변수별로 인덱스를 할당한다고 설명 했지만, 이 버전의 인터프리터는 효율을 전혀 고려치 않기로 했기 때문에, 지역 변수에도 해시테이블을 이용하도록 한다. 이후 가상머신(VM)을 도입할 때 효율에 대하여 생각하도록 하자.

NODE\_LET에 대한 작업은,(혹시 수행되지 않았다면)해시 테이블의 초기와 대입이 된다.

Streem에서는 지역 변수에서도 한번 대입된 값을 변경하는 것은 불가능하다. 이때문에, 이미 값이 대입된 지역변수에 재대입을 하려 하면 에러가 발생한다. 물론 이것도 앞으로 가상머신 도입시에는 컴파일러 에러로 처리해야 한다.

NODE\_IDENT는, 지역 변수 테이블을 엑세스하고, 변수에 값이 있으면 그 값을, 없으면 전역변수 테이블을 액세스하고 전역변수로도 정의가 되어 있지 않다면 에러를 발생시킨다.

Streem에서는, 함수실행시에는 node\_ctx라고 하는 구조체가 전달된다. 그 구조체는 실행시의 컨텍스트(문맥)을 보존한다. 지역 변수의 구현을 위한 지역변수 테이블(해시)은, 이 구조체에 멤버로서 추가된다.

node\_ctx는 어디까지나 현재의 노드를 따라 실행하기 위한 구조체이기 때문에, 장래에 가상머신화 작업시에는 구조체의 이름에 변경이 있을 거라 생각된다.

#### 중첩된 지역 변수

언어 설계상 판단해야 할 항목 중에 ‘중첩된 지역변수의 허용’ 이라는 항목이 있어, 언어에 따라 판단이 달라진다. 예를 들어, C와 Java에서는 중첩된 지역변수를 허용하며, 중괄호로 묶은 스코프(scope) 안에서 선언된 지역변수는 묶인 스코프내에서만 유효하다. 다른 스코프에서 같은 이름의 변수를 선언해도 이는 다른 변수로 인식한다(그림 2).

```

void

func()

{

int i = 10; /\* i의 적용 스코프는 func전체 \*/

while (i--) {

int j = 5; /\* j의 적용 스코프는 while문 안에서만.\*/

printf('i:%d j:%d\n', i, j);

}

/\* 중괄호 스코프 도입 \*/ {

double j = 1.5; /\* 위의 j와는 다른 변수 \*/

printf('new j:%g\n', j);

}

}

```

<center>

(그림 2) C의 중첩된 스코프

</center>

안쪽 스코프에서 바깥 스코프에서 선언한 변수와 같은 이름의 변수를 선언해도 상관없다. 이들 ‘같은 이름이지만 다른 변수’ 는 말 그대로 다른 변수이기 때문에 같은 이름이지만 타입이 달라도 상관없다. 하지만 코드의 가독성을 떨어트리기 쉽기 때문에 가능한 쓰지 않는 게 좋다.

한편, Ruby에서 스코프는 클래스 정의 및 메소드 정의에서만 도입되고 C괄호와 같은 일시적인 스코프를 도입하는 구문은 (나중에 소개하는 예외를 제외하고) 도입하지 않았다. 위에서도 설명했듯이 중첩된 스코프에 의해 도입 될 수 있는, 같은 이름이지만 다른 변수를 쓰는 경우는 Ruby에서는 있을 수 없다 (단지 다른 이름의 변수로하면 상관없다). 있으면 혼란스러울 뿐이므로 일부러 도입 할 필요는 없을 것이라는게 그 이유다. 특히 Ruby 같은 동적 타입의 언어는 형태의 불일치에 의한 오류 검출을 기대할 수 없기 때문에 불필요한 혼란은 피해야한다.

이 점에 대해서는 Streem에도 동일하게 가져가려고 생각한다.

#### 예외로서의 중첩된 스코프

Ruby에서는 ‘예외를 제외하고’ 중첩된 스코프는 없다고 했다. 이 ‘예외’ 에 대하여 이야기해 보겠다.

분명히 Ruby에는 중첩된 스코프를 회피하는 설계를 하고 있다. 하지만 예외로서 클로저는 중첩 스코프로 되어 있다. 결국, 블록과 익명함수 중에 등장하는 지역 변수의 스코프는, 클래스와 메소드의 정의 본체가 아닌, 이 블록과 익명함수의 범위내에서만 스코프가 된다(그림 3)

```

# do부터 end까지의 블록이 스코프가 된다

[1,2,3].each do |i| # i는 블록 내에서만 유효

sq = i \* i # sq도 블록 내에서만 유효

p [i, sq]

end

# 익명함수도 스코프를 도입한다

f = ->(x) { x \* x } # x는 블록 내에서만 유효

```

<center>

(그림 3) Ruby의 중첩된 스코프. p는 인수의 객체를 알기쉬운 문자열로서 표준출력으로 출력하는 메소드

</center>

생각해 보면 이는 당연한 것으로, Ruby 블록과 익명 함수는 함수이기 때문에 그 함수에 스코프를 한정한 지역 변수가 필요하다. 그렇지 않으면, 전역 변수밖에 없는 서브 루틴의 시대로의 퇴보가 되어 버린다.

혼란을 줄이기 위한 아이디어로 C 나 Java 같은 중첩된 스코프에서 외부의 범위와 같은 이름의 변수는 경고를 해 준다. 그러나 지역 변수의 유효범위를 한눈에 알 수 없게 된다는 단점도 있어 최선은 아니라는 것이 현실이다. Ruby의 변수 선언이 없는 사양이 화근이 되어버렸다(그림 4).

```

# 우연히 외부 스코프와 변수명이 겹쳤다

e = 10

[1,2,3].each do |i|

p i

end

[1,2,3].each do |e|

p e

end

# -v를 지정하면

# warning: shadowing outer local variable - e

# 라는 경고가 뜬다

# 스코프에서의 변수 반출이 귀찮다.

even = nil # 이 행을 잊어버리면 아픔을 맛본다.

[1,2,3].each do |i|

if i % 2 == 0

even = i

end

end

p even # 사전에 초기화 하지 않으면 even에 액서스가 불가능하다

# -v를 지정한다면

# warning: assigned but unused variable - even

# 라는 경고가 뜬다.

```

<center>

(그림 4) Ruby의 지역 변수의 단점

</center>

개인적으로 이 부분은 Ruby의 설계에서도 마음에 들지 않은 포인트다. 과거, 이를 개선하기 위해 여러 아이디어를 검토했다. (그림 4)의 경고 등은 이를 반영한 것이다. 실은 단순 경고가 아닌, 언어 스코프 설계에 따라 어떤 조치를 취하게 하고 싶었지만, 호환성 문제라든가, 언어 사양이 필요 이상으로 복잡해져 버리는 등의 이유로 채용할 수가 없었다.

채용하지 않은 규칙에는, 지역 변수의 전파(Propagation) 같은 것이 있다(그림 5).

```

[1,2,3].each do |i|

if i % 2 == 0

# 여기서 초기화된 변수가

even = i

end

end

# 스코프 밖에서 액세스를 하려하면

# 결국, 외부 스코프에 속하는 지역 변수로 본다.

p even

```

<center>

(그림 5) 지역 변수의 전파. 나쁘지 않은 아이디어였지만, 구현의 복잡함, 문제는 해결해도 사이드 이펙트로 혼란을 유발할 위험성 때문에 주저했다

</center>

### ■ 클로저(Closure)

블록에 의한 중첩 된 지역 변수가 있다는 것은 블록과 익명 함수에서 외부 범위의 변수를 참조 할 수 있다는 의미이다. 그리고 익명 함수는 범위 밖으로 나가 버린 후에도 살아있는 경우가 있다.

예를 들어 (그림 6)의 프로그램을 살펴 보자. 함수 incdec에서 만들어진 두 개의 익명 함수는 각각 외부의 지역 변수 acc를 참조하고 있다. incdec의 실행이 끝난 후, 보통 사라질 지역 변수지만, 익명 함수에서 참조되어 있기 때문에 사라지지 않는다. 이러한 상태를 함수 객체에 외부 범위의 변수가 ‘갇혀있기’ 때문에 함수 클로저라고 한다.

```

def incdec

acc = 0 # ‘클로저 변수‘

inc = -> {

acc += 1

acc

}

dec = ->() {

acc -= 1

acc

}

return [inc,dec]

end

inc, dec = incdec()

p inc.call # => 1 acc가 1증가

p inc.call # => 2 acc가 1증가

p dec.call # => 1 acc가 1감소

p dec.call # => 0 acc가 1감소

```

<center>

(그림 6) 클로저

</center>

#### 클로저의 구현

이러한 클로저의 구현은 꽤 복잡하다. Ruby의 처리 시스템에서는 함수 객체에 지역 변수의 중첩 관계를 ‘환경’으로서 저장한다. 범위의 외부 변수에 액세스 할 때 외부 환경을 참조하게 된다.

mruby에서 함수 객체(proc)과 환경 (env)의 정의를 (그림 7)에 나타내고 있다.

mruby 가상머신에는 스코프의 외부를 참조하는 명령, OP\_GETUPVAR과 OP\_SETUPVAR 가 있다. 양쪽 모두 몇 개의 환경에서 몇 번째의 변수를 참조할까를 지정하는 오퍼랜드를 사용한다.

환경은 독립된 Ruby의 객체에 함수 객체에서 참조되고있는 동안은 계속 살아 있다. 누구로부터도 참조되지 않는 경우 가비지 컬렉터에 회수된다.

```

struct REnv {

MRB\_OBJECT\_HEADER;

mrb\_value \*stack;

mrb\_sym mid;

ptrdiff\_t cioff;

};

struct RProc {

MRB\_OBJECT\_HEADER;

union {

mrb\_irep \*irep;

mrb\_func\_t func;

} body;

struct RClass \*target\_class;

struct REnv \*env;

};

```

<center>

(그림 7) mruby의 클로저 구현

</center>

#### Streem의 클로저

그러나 Streem에는 Ruby와 다른 점이 있어, 클로저의 구현을 보다 간단하게 할 수 있다. 다른 점이란 지역 변수도 고쳐 쓸 수 없다는 점이다. 물론 이는 (그림 6)과 같은 상태를 가진 클로저를 만들 수 없지만, 상태 및 부작용은 함수형 프로그래밍에서는 피해야 할 것이기 때문에 그다지 나쁘지 않을 것이다.

Streem의 함수 객체의 정의는 (그림 8)에 나와 있다. 앞서 언급 한 바와 같이 Streem는 지역 변수가 바뀔 걱정이 없기 때문에 클로저는 변수 값을 그대로 복사 해 오면 된다. 하지만 현재 구현에서는 편의상 외부 환경에 대한 링크를 유지하고 참조 할 때마다 링크를 찾아간다. 가상머신 버전은 성능 향상을 위해 함수 객체를 생성 할 때 변수의 값을 복사하게 된다.

```

typedef struct strm\_lambda {

STRM\_OBJ\_HEADER;

/\* 함수의 body \*/

struct node\_lambda\* body;

/\* 스코프를 유지하는 컨텍스트 \*/

struct node\_ctx\* ctx;

} strm\_lambda;

```

<center>

(그림 8) Streem의 클로저 구현

</center>

#### 컴파일시 체크

이번 구현에서는 이미 존재하는 지역 변수에 재할당과 존재하지 않는 지역 변수에 대한 참조를 실행시 에러가 발생한다. 그러나 지역 변수에 대입과 참조에서의 에러 발생 여부는 프로그램 코드만을 보고 판단이 가능하며, 원래는 컴파일 에러 처리가 되어야 한다.

런타임 오류는 ‘실행되지 않으면 감지되지 않는다’ 라는 불안이 항상 따라 다니지만 컴파일 오류에 그럴 걱정은 없다. 옛날 언어에는 문법 오류에도 실행이 되어 버리는 것이 있었지만, 현대의 언어에서는 대부분 컴파일시에 이를 체크하며 이는 프로그래밍 언어의 진화의 역사이다. 조만간 이렇게 개선이 있을 예정이다.

### ■ 예외적 사상

자, 지역 변수 다음으로 예외 처리에 대해 생각해 보자.

다양한 작업을 할 때 어떤 이유에서 의도 한대로 진행되지 않을 가능성은 항상 존재한다. 처리의 본론에 집중하고 싶은 마음은 굴뚝이지만, 여전히 이러한 예외적 인 경우를 무시할 수 없다.

예를 들어 ‘파일 열기’라는 심플한 작업을 생각해 보자. 수행 할 작업은 ‘파일 이름을 지정하여 파일을 오픈한다’ 이다. Linux 등 UNIX 계 OS에서는이 작업은 open이라고 하는 시스템 콜이 맡고 있다.

open시스템 콜의 형태는 (그림 9)에서 보여주고 있다. flags로 오픈 모드(읽기, 쓰기, 수정)을 지정하지만, 새롭게 파일을 만들게 되는 경우, 제3인수로서 파일모드, 결국 파일의 접근권한(퍼미션)을 지정한다.

```

int open(const char\* path, int flags);

int open(const char\* path, ins flags, mode\_t mode)

```

<center>

(그림 9) open시스템 콜

</center>

자, 이렇게 심플한 파일의 오픈 처리는 있지만, 그래도 예외 상황이 발생한다. open 시스템 호출이 발생할 수있는 오류를 (표 1)에 정리해 두었다. EPERM 같은 Linux 관련 것도 있지만, 실로 24 종류의 예외적 인 현상이 발생할 수있는 것을 알 수 있다

물론 대부분의 경우 파일은 무사히 오픈 하겠지만, 그렇다고 이 예외들을 무시할 수는 없다. 만일 예외적인 상황이 발생했을 경우에 프로그램이 비정상적으로 종료하는 등 곤란한 상황이 발생할 것이기 때문이다. 편집기를 사용하여 파일 이름을 잘못 타이핑하여 존재하지 않는 파일을 지정했을 뿐인데 편집기 전체가 비정상적으로 종료 해 버리면 파일이 날라는 눈물나는 상황도 발생한다.

![](table2-8-1.png)

<center>

(표 1) open에서 발생하는 에러 목록

</center>

프로그램의 실행에는 항상 예외상황이 발생하므로, 이에 대해 적절하게 처리 할 필요가 있다.

그러나 한편으로 예외적 사건은 어디 까지나 예외이기 때문에, 가능한 쓰고 싶지 않고, 읽고 싶지 않다는 욕구도 있다. 예외 처리에 묻혀 프로그램 논리의 본질을 읽기 어려워지는 것은 바람직하지 않다고 생각한다.

프로그래밍 언어에서는이 예외적 사건을 처리하는 방법이 설계상의 중요한 관심사이다.

#### 명시적 에러 체크

C와 최근에는 Go에서도 명시적인 오류 체크를 하게 되어 있다. 예를 들어, 실패 할 가능성이있는 함수를 부르면 그 반환 값을 확인하고 성공했는지 여부를 확인하는 것이다. 이 방법의 장점은 언어 사양으로 지원이 불필요하기 때문에 매우 간단하다는 점이다. 또한 나중에 소개하는 ‘예외’는 암묵적으로 실행을 중단 해 버리므로,시기에 따라 의외의 사태를 일으킬 위험이 있다. 예를 들어 실행이 도중에 중단되었기 때문에 데이터 불일치가 발생하거나 메모리의 해제가 이뤄지지 않아 메모리 누수가 발생하거나하는 사태이다. 이러한 문제가 발생하지 않는다는 것을 (역시 예외가있는 언어이다) C++에는 ‘예외 안전’ 이라고 부르고 있다. 그러나 조금 살펴보면 알 수 있듯이, C++에서도 예외 안전을 보장하는 것은 매우 어렵다. 명시적인 오류 검사는 그런 어려움과 무관하다.

한편, 명시적인 체크는 예외 사례의 처리 로직에 들어간 만큼 정상 처리가 매몰되어 버렸기 때문에 이해하기 어려워지는 단점이 있다. 또한 명시적인 체크를 잊어 버리면, 전제 조건이 성립하지 않은 채 프로그램이 충돌하여, 이상을 일으키며 중단되는 것 뿐만 아니라 보안 문제까지 야기하는 위험성도 있다.

Go에서는 함수가 복수의 값을 돌려줄 수 있는 것을 이용하여, 명시적인 체크를 잊어버릴 위험을 낮추고 있지만, 그래도 복잡하다는 것은 틀림없다.

#### ‘예외(exception)’를 발생시킨다.

다소 혼란스럽지만, 많은 프로그래밍 언어는 예외적인 상황이 발생하는 경우 ‘예외’라는 것을 생성하고 프로그램을 중단시키는 기능을 제공한다. 예외 기능을 제공하는 언어로는 C ++, Java, Ruby 등이 있다. 최근의 프로그래밍 언어에서는 오히려 명시적인 오류 검사보다 예외 기능이 일반적이라고 생각한다.

예외의 가장 큰 장점은 예외적 사건에 의해 전제 조건이 성립하지 않은 경우 (예 : 파일이 존재하지 않기 때문에 열 수없는 등) 자동으로 실행이 중단된다는 점이다. 따라서 전제 조건이 성립하지 않은 채 실행되는 위험이없고, 그만큼 안전성을 보장할 수 있다.

예외의 단점은 이미 언급 한 바와 같이, 이상 사태에 대해 자동으로 중단되므로 예외 안전성의 실현이 어려울 수 있다. 그러나 가비지 컬렉션이있는 Ruby 같은 언어에서는 수동으로 리소스 관리를하는 C ++와 같은 언어보다 예외 안전성을 유지하기 쉽다. 어려움의 정도는 언어에 따라 달라진다.

#### Swift의 Optional

미 Apple 사가 발표 한 Swift는 최근의 언어에 비해 보기드물게 예외 처리 기능이 없다. 대신 실패 할 가능성이 있는 함수는 그 타입이 Optional <T>라는 타입으로 되어 있다. Optional은 타입의 임의값 또는 nil 값을 취하는 타입으로 많은 함수는 실패하면 nil을 반환하도록 설계되어 있다. Swift는 Optional <T> 를 T? 라고 약어로 표기할 수 있다. 어느 타입이 Optional 인 경우, 해당 타입의 값은 직접 참조가 불가능하다.

```

var i: Int? = 10;

println(i + 2) // 에러

```

Optional타입으로부터 실제 값을 추출하는 경우, 그 조작을 Swift에서는 unwrap이라고 부른다. 변수명 뒤에 ‘!’ 를 붙이면 값을 추출할 수 있다. 값이 nil인 경우 실행시 에러가 발생한다.

```swift

println(i! + 2)

```

nil일때 취할 값도 지정 가능하다.

```

println((i??5) + 2)

```

나아가서는 let 과 if 을 조합하여 명시적으로 nil체크도 가능하다.

```

if let i2 = i {

println(i2 + 2)

}

```

이와같이 쓴 경우는, i2의 타입은 Optional이 아닌, Int가 되므로 더이상 unwrap을 할 필요가 없다.

마지막에 ‘.’대신에 ‘?.’ 을 사용한 메소드를 호출하는 방법을 소개한다. ‘?.’ 를 사용하는 경우, 값이 있을 대는 그 메소드를 실행하고 nil의 경우에는 실행을 하지 않고 nil을 반환한다.

```

var dog? = Dog()

dog!.bark() // nil이면 에러

dog?.bark() // nil이면 nil

```

이와 같이 Swift에서는 예외기능을 쓰지않고, Optional타입에 의해 예외상황(에러)에 대응하고 있다. 이 Optional타입은 Haskel의 Maybe타입과 OCaml의 Opion타입을 참고하여 만든 것 같다. 정적타입을 잘 이용하여 에러 처리를 하는 괜찮은 아이디어라고 생각된다.

#### 에러 무시

그러면, Streem 예외 처리는 어떻게 할 것인가에 대해 생각해 보자. 다른 언어와 달리 Streem에는 명확하게 두 실행 단계가 있다. 즉 파이프 라인을 준비하는 초기화 단계와 파이프 라인에 데이터가 흘러가는 파이프 라인 단계다. 초기화 단계에서 예외 상황이 발생하면 그 이후 실행에 어려움이 발생하기 때문에, 일반적 예외 기능을 이용한다. 한편, 파이프 라인 단계에서는 대량의 데이터가 흘러가는 것이다. 예를 들어 10GB의 데이터 파일을 읽을 때 그 중 하나의 행이 어떤 이유로 손상 때문이라는 이유로 프로세스가 실패하는 것은 반드시 바람직하지는 않다.

그래서 파이프 라인 단계에서 명시적으로 지정하지 않는 이상, 오류가 발생한 데이터에 대해서만 실행을 중단 하고, 파이프 라인은 다음의 데이터 처리를 계속한다. 이러한 오류 무시에 의한 처리의 지속은 미국 Google 사가 개발 한 클라우드 데이터 처리 언어 Sawzall에서도 채용되고 있다.

#### Streem에서의 예외처리 구현

Streem의 메소드를 구현하는 C 함수는 (그림 10)과 같은 프로토 타입을 가지고 있으며, argc와 argv가 인수를, ret이 메소드의 반환 값을 의미한다. 그리고, 함수의 반환 값이 메서드 실행의 성공 또는 실패를 의미하고, 성공하면 0을, 실패하면 그 이외를 반환한다.

```

int exec\_plus(node\_ctx\* ctx, int argc, strm\_value\* args, strm\_value\* ret);

```

<center>

(그림 10) Streem메소드를 구현하는 함수

</center>

#### 마치며

지역 변수도, 예외에 대해서도, 현대의 프로그래밍 언어로는 이제 상식이라고해도 좋은 기능들이다. 그러나 언어 설계라는 관점에서 보면, 이런 당연한 기능조차도 설계상의 절충과 과제를 통해 찾아낸 것이다.

언어 설계는 이러한 세세한 부분까지 상세히 검토해 나가는 행위이다.

<hr>

### 타임머신 칼럼

\*\*\*길것 같지만 짧은 프로그램의 역사\*\*\*

> 2015 년 7 월호 게재분이다. 이번에는 비교적 독립적인 성격의 지역 변수와 예외 처리까지 2개짜리 분량이다. 현대의 프로그래밍에서 지역 변수의 존재는 당연한 것이다. 그러나 불과 30여 년 전만해도 지역 변수 존재하지 않는 프로그래밍 언어가 현역으로 활동했다는 것은 놀라운 일이다. 실제로 그 시대를 경험 한 나조차 까맣게 잊고있어 재차 놀랐을 정도여서, 처음부터 ‘제대로 된’ 언어로 프로그래밍을 배운 사람들에게는 놀라운 사실일지도 모르겠다. 역사가 짧은 프로그래밍 세계에서는 그런 것들이 가끔 있다. 아득한 옛날 일 생각하면 불과 수십 년 전의 일이었다거나 역사적인 인물로 생각하고 있으면 생존중이거나... 예외 처리에 대해서도 짚어보자. 예외 기능 자체는 아마 Lisp과 그 주변에서 탄생했기 때문에 40 년 이상의 긴 역사를 가지고 있지 않나 싶다. 그러나 Java가 일반화되기 전까지는 그다지 널리 사용되지 않았다. 이 부분 경위는 가비지 컬렉션과 동일하다.

> 예외 기능에 의해 명시적 오류 처리가 줄어들어 프로그램의 가독성이 좋아지는 장점이 있다. 그러나, 프로그램이 예상치 못한 타이밍에 중단 될 가능성이 있기 때문에, 의외의 동작이 발생할 수 있다. Go와 같은 언어에서는 컨커런트 프로그래밍과 궁합이 나쁘다는 이유로 예외 기능을 도입하지 않고, 반드시 오류 체크를 하는 스타일을 채택하고 있다. Streem 예외 기능은 원칙적으로 ‘오류가 발생하면 그 데이터는 버린다’ 로서, 스트리밍 처리의 성질을 잘 이용하고 있다. 단 예상 외의 데이터에 의한 오류면 데이터 부분은 넘어 가겠지만, 그로 인한 프로그램의 오류가 무시되고 디버깅이 어려워지는 경우도 있으므로, 여기에는 향후 연구가 더 필요하다.