Kscript5 기술문서

목차

1-1. Kscript 개괄

```
test.print("Hello, world!")
```

kscript는 유연한 프로그램 사용과 매크로 제어를 위해 파이썬과 유사한 문법을 가지도록 만들어진 스크립트 언어다. 사용자는 원시 소스코드를 컴파일 한 뒤, 다른 고급 언어로 제공되는 가상머신에서 실행시킬 수 있다. 구현체에 따라 미세한 차이가 발생할 수 있으나, OS와 호스트 언어 상관없이 동일한 실행파일을 사용해서 동일한 결과를 볼 수 있다.

언어 문법은 python을 기반으로 하여, golang의 for문과 javascript의 함수 호이스팅과 관련하여 영향을 받았다. 제어문 형식, 절차지향, 함수 기반 구조는 c와 유사하다. kscript는 다음 특징을 가진다.

- 절차지향 : 객체 개념이 없고 변수와 함수는 명확하게 구분되며 실행의 주체는 함수다.
- 동적 타이핑 : 변수의 기본값은 None이고, 타입은 고정되어 있지 않다.
- 변수 선언의 부재 : 암묵적으로 전역/지역 변수가 선언된다.
- 중괄호 사용 : 들여쓰기 대신 중괄호를 사용해 블록을 표시한다.
- KDB 형식 리터럴 : 리터럴 값 표기 시 KDB 형식을 사용한다.

1-2. 변수와 타입

`a = None`
`b = True`
`c = -1557`
`d = 3.1415`
`e = "Hello, world!"`
`f = '8a91'`
	kscript의 변수 선언은 파이썬과 같이
	(변수명) = (값 또는 식) 으로 한다.
	변수명은 예약어를 사용할 수 없으며 공백이 들어가서는 안 된다.
	kscript의 타입 시스템은 총 6개의 타입을 가진다.

- none : 널 값은 가장 기본적인 값으로, 리터럴 표현은 **None** 이다.
- bool : 참 또는 거짓을 나타내며, 조건문은 최종적으로 bool 값만을 받을 수 있다. 리터럴 표현은 **True** 혹은 **False** 이다.
- int : 정수 값으로, 구현체에 따라 범위가 다를 수 있다. 리터럴 표현은 **(부호)[0-9...]** 이다. 현재 go runtime은 64비트 정수를, py runtime은 파이썬 기본 정수를 쓴다.
- float : 실수 값으로, 구현체에 따라 범위가 다를 수 있다. 리터럴 표현은 소수점이 하나 들어간 정수로, **(부호)[0-9...].[0-9...]** 이다. 현재 go/py runtime 모두 64비트 실수를 쓴다.
- str : 유니코드 문자열 값으로, 선언에 쌍따옴표가 필요하며, 이스케이프 글자로 #을 쓴다. 문자열 리터럴 표현은 이스케이프 글자를 사용하여 한 줄 안에 끝내는 것이 좋으나, 여러 줄을 사용하는 것도 가능하다. 리터럴 표현 예시 : **“” “#n” “#s” “#”” “##”**
- bytes : 이진 데이터 값으로, 선언에 따옴표가 필요하며, 0~f 헥스값으로 표현된다. 바이트 리터럴 표현은 한 줄 안에 끝내는 것이 좋으나, 사이에 공백이나 줄바꿈을 넣는 것도 가능하다. 리터럴 표현 예시 : **‘’ ‘001020’ ‘FFFa’**

1-3. 연산자

모든 연산자는 이항 연산자이다. 동작에 전후로 2개의 값 또는 식이 필요하다. 또한 연산자 결함 순서가 모두 좌->우 임에 주의하라. 거듭제곱 연산자의 경우 이 부분이 파이썬과 반대이다. 사칙연산은 bool 타입에 대한 연산 외엔 파이썬과 동일하게 동작한다. (n : none, b : bool, i : int, f : float, s : str, c : bytes)

기호	의미	피연산자와 결과
+	덧셈 | 논리합	bb->b, ii->i, if,fi,ff->f, ss->s, cc->c
-	뺄셈	ii->i, if,fi,ff->f
*	곱셈 | 논리곱	bb->b, ii->i ; if,fi,ff->f, si,is->s, ci,ic->c
/	나눗셈	ii,if,fi,ff->f
//	몫	ii->i, if,fi,ff->f
%	나머지	ii->i, if,fi,ff->f
**	거듭제곱	ii->i|f, if,fi,ff->f

기호	의미	피연산자와 결과	비고
==	동등	nn->b, bb->b, ii,if,fi,ff->b, ss->b, cc->b	이외 타입은 모두 False
!=	동등하지 않음	nn->b, bb->b, ii,if,fi,ff->b, ss->b, cc->b	이외 타입은 모두 True
<	작음	ii,if,fi,ff->b, ss->b, cc->b	s,c는 바이트 순서로 비교
>	큼	ii,if,fi,ff->b, ss->b, cc->b	s,c는 바이트 순서로 비교
<=	작거나 같음	ii,if,fi,ff->b, ss->b, cc->b	s,c는 바이트 순서로 비교
>=	크거나 같음	ii,if,fi,ff->b, ss->b, cc->b	s,c는 바이트 순서로 비교

연산자는 우선순위에 따라 실행된다. 우선순위가 같은 연산자는 앞에서부터 실행된다.
(==, !=, <, >, <=, >=) < (+, -) < (*, /, //, %) < (**) < (괄호)

`a = 5 - 1 == 8 + -1 * 32 // 2 ** 3` `test.print(a) # True`	연산자 우선순위대로, 왼쪽에서 오른쪽으로 계산이 진행된다.
`a = 2 ** 3 ** 2` `test.print(a) # 64`	파이썬은 512가 나온다. 거듭제곱 연산자의 결합 순서가 반대임에 주의하라.
`a = 1+1 # 컴파일 에러!!` `test.print(a)`	연산자는 항상 앞뒤에 공백이 와야 제대로 인식된다.
`a = ((12 / 3) + 2)` `test.print(a) # 6.0`	반면 괄호는 공백 없이도 인식된다.

note. 리터럴, 변수, 함수 호출, 괄호로 묶인 식은 모두 피연산자로 취급된다. 호출 인자, 변수 할당, 이항 연산은 모두 피연산자를 대상으로만 할 수 있다는 점에 주의하라.

1-4. 제어문

조건문과 반복문은 프로그램의 흐름을 제어하는 중요한 부분이다. c와 유사하게 소괄호가 조건, 중괄호가 제어 범위를 나타낸다. kscript 파서는 다양한 중괄호 스타일을 지원한다.

- if 문 : 특정 조건일때만 코드를 실행시킬 때 사용한다.

`if (cond) {`
`    ...`
`}`
	cond가 True일때만 ...을 실행한다. if의 조건절은 항상 bool 타입의 값이 와야 한다.

- if-else 문 : 조건이 참/거짓일 때 다른 코드를 실행시킨다.

`if (cond) {`
`    ...`
`} else {`
`    ...`
`}`
	cond가 True라면 위쪽의 ...을, False라면 아래쪽의 ...을 실행시킨다. if와 마찬가지로 bool 타입의 조건만 받으며, if 문 바로 뒤에 else가 오는 형식이다.

- while 문 : 조건이 참인 동안 계속 코드를 실행한다.

`while (cond) {`
`    ...`
`}`
	cond가 True일 동안 계속 ...을 실행시킨다. 처음부터 cond가 False라면 ...은 실행되지 않고 지나간다.

- for 문 : int, str, bytes에 대해 반복자와 비슷한 효과를 낸다.

`for (인덱스, 반복자 <- 변수|리터럴) {`
`    ...`
`}`
	for변수는 for문에 진입하며 딱 1회 평가된다. ... 코드에서 인덱스, 반복자, for변수의 값을 바꿔도 전체 반복엔 영향이 없다.

인덱스는 현재 반복한 횟수이며 0 ~ n-1의 값을 가진다. 반복자는 for변수의 인덱스 번째 원소이며, for변수와 같은 타입을 가진다. for변수의 종류에 따른 행동을 알아보자.

int (+n)	index = 0 ~ n-1	iter = 0 ~ n-1
int (-n)	index = 0 ~ n-1	iter = 0 ~ -n+1
int (0)	실행되지 않음	실행되지 않음 (바로 다음 코드로 넘어감)
str (s)	index = 0 ~ len(s)-1	iter = s[index] (유니코드 길이 기반)
bytes (c)	index = 0 ~ len(c)-1	iter = c[index] (iter 또한 bytes 타입이다)

`for (i, r <- "Hello") {`
`    i = i + 1`
`    test.print(i)`
`    test.print(r)`
`} # 1H2e3l4l5o`
	go의 for와 비슷하다. 특정 타입의 값들에 대해 동작하며, ... 코드 실행 전 인덱스와 반복자 변수가 설정된다.
	for의 for변수 위치에는 리터럴이나 변수명만 올 수 있다.

지원되는 중괄호 스타일 중 대표적인 3가지를 보여준다.

K&R	BSD	GNU

for (i, _ <- 5) {
  test.print(i)
}
	for (i, _ <- 5)
	{
	  test.print(i)
	}
		for (i, _ <- 5)
		  {
		  test.print(i)
		  }

note. 간단한 구구단과 소인수분해 프로그램을 예시로 추가한다.

```
for (i, _ <- 8) {
    for (j, _ <- 9) {
        test.print(i + 2)
        test.print(" * ")
        test.print(j + 1)
        test.print(" = ")
        test.print((i + 2) * (j + 1))
        test.print("#n")
    }
    test.print("#n")
}
```

```
i = 9699690
d = 2
test.print(i)
test.print(" = ")
while (d < i) {
    if (i % d == 0) {
        i = i // d
        test.print(d)
        test.print(" * ")
    } else {
        d = d + 1
    }
}
test.print(i)
```

1-5. 함수와 지역변수

지금까지는 모두 mainflow에서 전역변수만을 이용해 프로그래밍했다. 하지만 이 방식은 변수명 겹침 문제나 복잡한 프로그램 작성이 어렵다는 문제가 있다. 이제 구조적 프로그래밍의 꽃인 함수에 대해 알아보자.

- 함수 구문

```
def 함수명(인자명...) {
}

def add(a, b) {
    return a + b
}
```

함수는 실행 흐름을 잠시 가져와 내부 코드를 실행하고 다시 되돌린다. 모든 함수는 1개의 반환값을 가진다.
return ...으로 명시적으로 반환하거나, 함수가 끝나면서 자동으로 **None**이 반환된다.

```
함수명(인자명...)
add(1, 1.5) # 2.5
```

함수 정의엔 def 키워드가 필요하지만, 함수 호출은 함수명과 인자들로만 할 수 있다. 각 인자들은 쉼표로 구분한다.

- 전역변수와 지역변수

전역변수는 모든 스코프에서 접근할 수 있고, 지역변수는 해당 함수 내부에서만 접근할 수 있다. 변수의 선언과 할당은 똑같이 **(변수명)** = ...으로 하지만, 전역/지역 여부는 스코프와 환경에 따라 달라진다.

def f(b) {
  a = 1 # global
  b = b + 1 # local
  c = a + 2 # local
}
 	
a = 0 # global
b = 1 # global
f(b)
test.print(a) # 1
test.print(b) # 1
	mainflow에서 변수 선언시 항상 전역변수가 된다. (a, b) 전역변수는 모든 함수에서 접근하고 값을 바꿀 수 있다.
	
	함수 내부에서 변수를 선언/할당하는 구문은 로컬-전역-로컬 순서로 스코프가 바뀐다. a는 local a가 없으니 전역으로 가고, global a가 존재해 할당문이 되었다.
	b는 local b가 존재해 (인자는 지역변수로 취급) 할당문이 되었다. c는 local c가 없고 global c도 없어 새 지역변수 선언문이 되었다.

basic = 100
car()
bus()
def car() {
  fee = basic + 50
  test.print(fee)
  test.print("원 (승용차)#n")
} # 150원 (승용차)
def bus() {
  fee = basic + 100
  test.print(fee)
  test.print("원 (버스)#n")
} # 200원 (버스)
	전역변수는 모두가 같은 하나의 이름을 공유하지만, 지역변수는 함수별로 독립적이다. 따라서 다른 함수와 이름이 같은 지역변수를 선언해도 독립적으로 사용된다.
	
	note. 함수 car와 bus가 정의 코드가 나오기 전에 호출되고 있다는 것이 보이는가? 함수 호이스팅을 통해 모든 함수의 이름을 먼저 인식하기에 코드 어느 부분에서나 함수를 호출할 수 있다.

note. 간단한 피보나치 수, 팩토리얼 계산 프로그램을 예시로 추가한다.

def fib(n) {
  if (n < 3) {
  return 1
  } else {
  return fib(n - 2) + fib(n - 1)
  }
}
test.print(fib(10)) # 55
	def fact(fact) {
	  if (fact <= 1) {
	  return 1
	  } else {
	  return fact(fact - 1) * fact
	  }
	} # 변수명과 함수명은 겹칠 수 있다
	# 그러나 이름 겹침을 권장하진 않는다.
	test.print(fact(6)) # 720

1-6. 외부함수

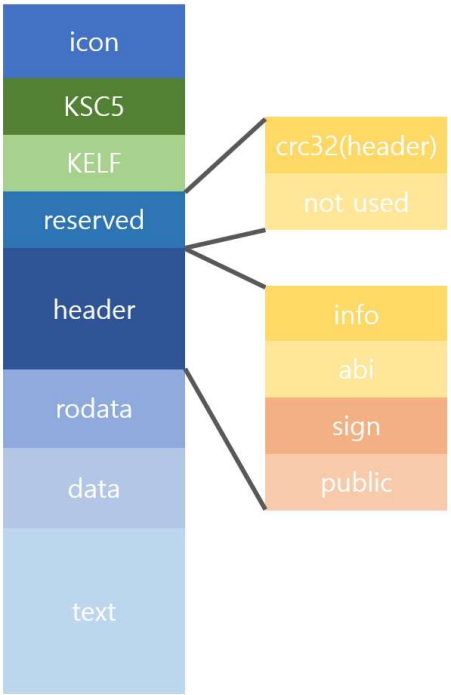
앞서 살펴본 kscript 문법은 host language, OS 불문 동일하게 작동한다. 그러나 필요한 기능이 복잡하거나 실행 환경에 따라 다르게 돌아가야 할 수 있다. 외부함수는 런타임에게 인자와 인터럽트 코드를 넘겨서 kscript에 없는 기능을 활용할 수 있게 해준다.

위에서 사용한 test.print 함수도 외부함수이다. kscript 단독으로는 입출력을 할 수 없기에 (환경이나 호스트 프로그램에 따라 요구사항이 달라짐) 외부함수를 사용해서 입출력 기능을 부여한 것이다. 런타임에서 기본으로 지원되는 test.*에는 현재 6개의 함수가 디버그용으로 존재하며, 인터럽트 코드 16~21을 가지고 있다. [intr 16~32, abi 1]

`test.input(any q) -> str # 질문 q를 출력하고 사용자 입력을 반환.`
`test.print(any v) # v의 문자열 형태를 출력. (줄바꿈 없음)`
`test.read(str path, int size) -> bytes # path 경로의 파일을 처음부터 size 만큼 읽어옴. (음수라면 전체읽기)`
`test.write(str path, str|bytes data) # path 경로의 파일을 만들고 data 기록.`
`test.time() -> float # 현재 유닉스 시간을 실수 형태로 반환.`
`test.sleep(int|float t) # t초만큼 실행을 멈춤.`

외부함수는 인터럽트 코드에 따라 호스트 프로그램이 해당 기능을 처리해주어야 하기 때문에 실행 환경마다 지원 여부나 동작 방식이 다를 수 있다. 후술할 ABI 정보가 어떤 외부함수를 사용하는지를 나타낸다. 외부함수명은 예약어 취급받기 때문에 .을 사용하여 도메인을 나누거나 필요한 함수군만 제한하여 사용해야 한다.

2-1. 실행파일 형식



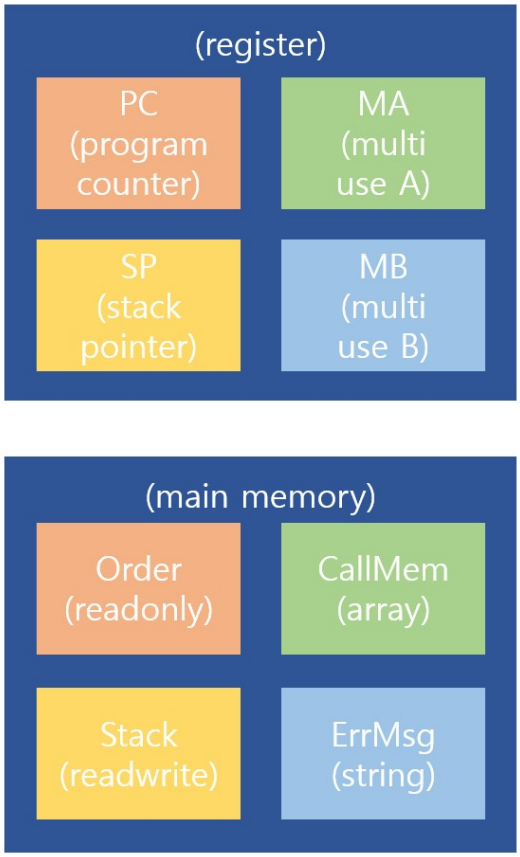
kscript 실행파일은 KELF 형식을 쓴다. KSC 청크를 사용하며, KSC 헤더에는 프로그램 아이콘 사진과 첫 데이터 청크의 CRC32 체크섬이 들어간다. 이후로 4개의 데이터 청크(header, rodata, data, text)가 위치한다.

header 청크는 프로그램 정보를 나타낸다. 프로그램 설명 (info), 사용하는 외부함수 종류 (abi), [rodata + data + text]의 해시값 서명 (sign), 서명 공개키(public)가 위치한다.

rodata는 상수 데이터를 적재하는 바이트코드가 들어간다. 코드에서 리터럴이 상수 데이터로 들어간다. data는 같은 형식의 전역변수 데이터 바이트코드가 들어간다.

text는 실행할 바이트코드가 들어간다. 이 코드를 0번 위치부터 실행하게 된다. 실행 바이트코드는 항상 8의 배수 크기이다.

2-2. 스택과 레지스터



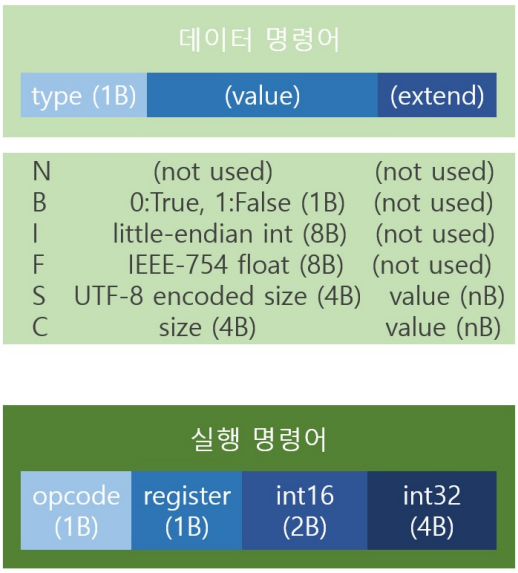
KVM은 속도 향상을 위해 레지스터를 추가한 스택 기반 가상머신이다. PC와 SP는 다음 명령어 주소와 현재 스택 프레임의 주소를 가지는 특수 레지스터이고, MA와 MB는 다양한 목적으로 사용할 수 있는 범용 레지스터이다.

메인 메모리의 Order에는 실행할 명령어가 로드 시 적재되며 읽기만 가능하다. Stack엔 상수와 변수, 임시 값들이 들어가며 임의 위치에 접근 & 값 변경, 말단에 push & pop이 가능하다.

CallMem은 외부함수 호출 시 인자들이 저장되며, ErrMsg는 발생한 오류가 기록된다. 외부호출 이후 다시 실행 사이클로 돌아오려면 반환값을 레지스터 MA에 넣어야 한다.

note. 스택은 push & pop을 빠르게 할 수 있는 자료구조지만, 레지스터를 사용하는 것보다 연산과 제어 명령어를 추가로 사용해야 하기에 더 느리다.

2-3. 실행파일 로드



KVM이 실행파일을 로드하면, 옵션에 따라 파일의 서명을 검증하고 명령어들을 Order와 Stack에 올린다. KVM은 Load에서는 데이터 명령어를, Run에서는 실행 명령어를 읽어 동작한다.

rodata의 상수와 data의 전역변수들은 순서대로 스택에 값이 올라오는데, 이 데이터를 지시하는 데이터 명령어는 1B의 타입과 인코딩된 값들로 되어 있다. str과 bytes처럼 크기가 정해지지 않은 값들은 먼저 길이가 나오고 뒤이어 실제 값이 나온다.

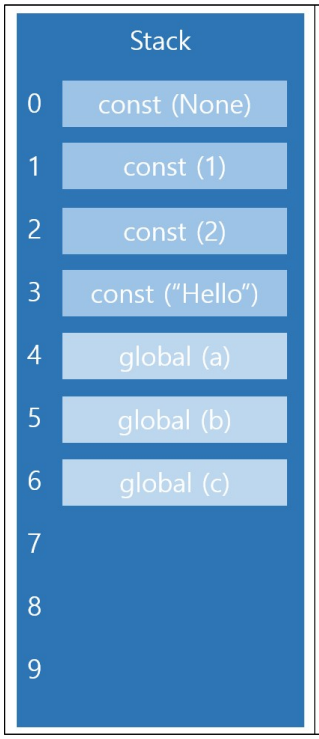
실행할 내용을 지시하는 실행 명령어는 항상 8B로 되어 있으며, 명령어 종류와 피연산자 3개로 구성된다. 명령어 종류에 따라 피연산자 3개를 모두 사용할 수도, 아무것도 쓰지 않을 수도 있다.

2-4. 실행 사이클

KVM의 Run은 fetch - decode - execute - interupt 사이클을 따른다.

- fetch : PC 위치의 명령어를 가져오고 명령어와 피연산자를 분리한다. PC는 1 증가한다.
- decode : 어떤 명령어인지 해석한다. 현재 KVM은 0 ~ 127 범위의 opcode 중 36가지를 받아들인다. 대략적인 명령어 구분은 상위 4비트로 할 수 있다. 0x0?, 0x1? : 흐름 제어. 0x2? : 메모리 제어. 0x3?, 0x4? : 산술 연산. 0x5? : 비교 연산. 0x6?, 0x7? : 단축 명령.
- execute : 명령어 종류와 피연산자에 따라 데이터를 조작하거나 인터럽트를 일으킨다.
- interupt : 예외 발생 여부, 옵션 등에 따라 인터럽트를 일으키거나 바로 다음 사이클로 넘어간다. (다시 fetch부터 반복)

3-1. 스택 프레임

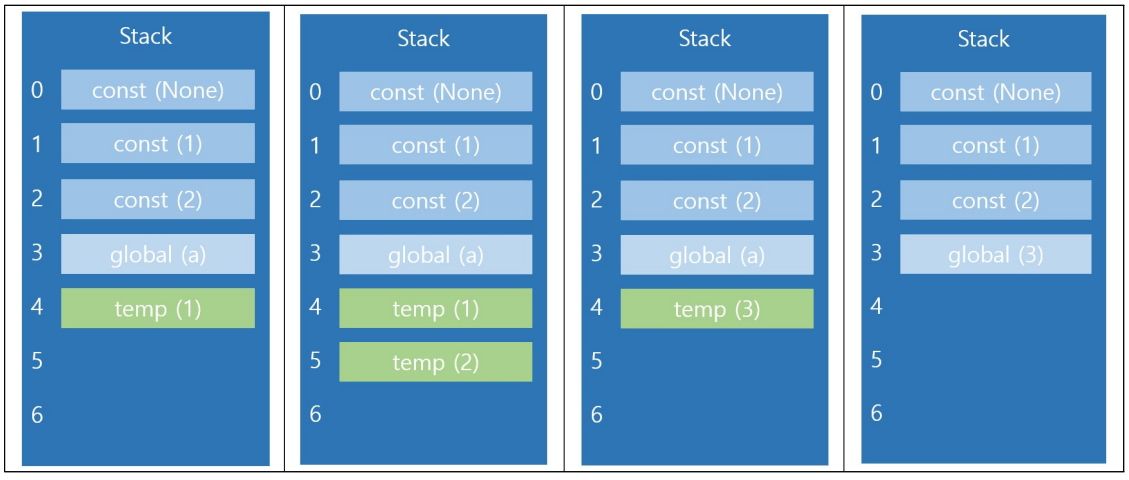


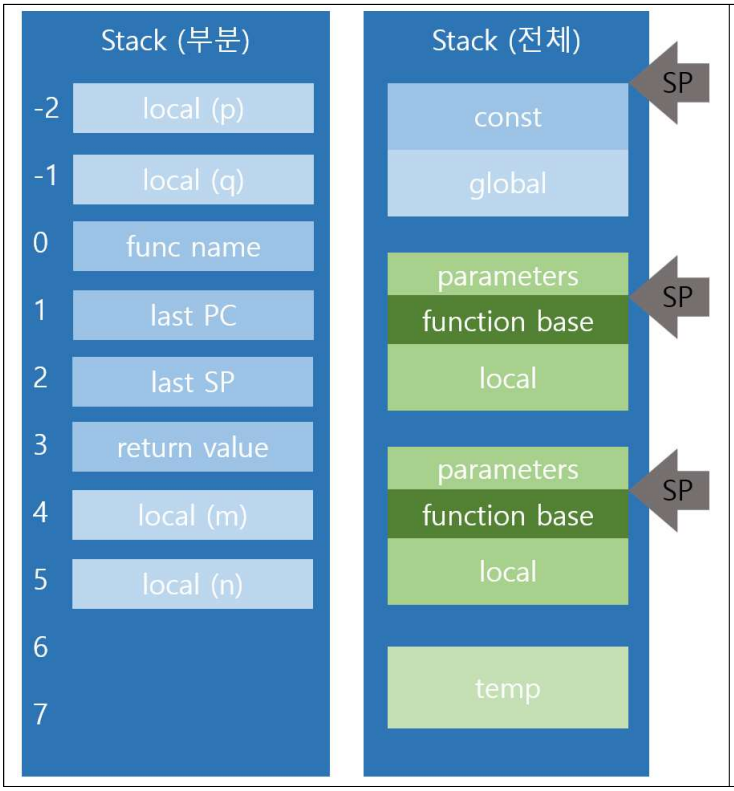
스택은 KVM에서 변수가 저장되고 임시 계산값이 위치하는 메인 메모리 공간이다. 스택은 상수 - 전역변수 - 함수 프레임 구조이다. 상수와 전역변수 영역은 크기가 고정되어 있고, 함수 프레임은 함수가 호출될 때마다 생겨나고 종료되면 사라진다.

SP는 현재 스택 프레임이 시작되는 주소이다. mainflow 실행 시 SP는 0이지만, 함수가 호출되면 원래 SP의 값은 스택에 저장되고 해당 함수의 스택 프레임의 주소로 바뀐다. 상수와 전역변수 접근은 절대주소 기반이지만, 지역변수 접근은 변수 위치에 SP를 더해야 접근할 수 있다.

note. 아래 그림은 `a = 1 + 2`가 4 cycle의 저수준 코드로 변환되어 실행되는 모습이다.

- `PUSHSET CONST &1` : 위치 1의 상수를 스택에 push
- `PUSHSET CONST &2` : 위치 2의 상수를 스택에 push
- `ADD` : 스택에서 2개를 pop 한 뒤, 더해서 push
- `POPSET GLOBAL &3` : 스택에서 pop 한 값을 위치 3에 세팅





함수 프레임은 함수 호출과 복귀에 대한 정보를 가진다. 함수 프레임 위쪽에는 함수 인자(지역변수 취급)가 온다.

SP +0 ~ +3 위치는 함수, 기존 PC/SP, 반환값 (기본값 None)이 있다. 아래에 함수의 지역변수가 위치한다.

함수가 반환/종료되면 자동으로 SP+3 위치의 반환값이 MA에 세팅된다.

note. 전체적인 스택의 구조는 상수 - 전역변수 - 함수 프레임 - 임시 계산값 순서로 되어 있다.

3-2. 인터럽트

인터럽트는 현재 KVM의 실행 상태를 알려주거나 외부함수를 호출하는 과정이다. 인터럽트는 옵션에 따라 매 사이클 이후 발생하거나 실행에 문제가 생겨 나올 수 있다. 실행 명령어로 직접 인터럽트를 호출할 수도 있다. 인터럽트 코드는 32비트 정수 값을 가질 수 있다. 예약된 범위가 아닌 인터럽트 코드(32 이상)는 컴파일 환경이나 ABI 정보에 따라 달라질 수 있다.

기본 인터럽트 (예약됨)

코드	의미	비고
-1	심각한 오류 (c_err)	잘못된 메모리 접근 등 프로그램 비정상종료
0	정상 (normal)	정상적으로 사이클이 진행됨, 이후 계속 실행
1	정지 (halt)	정지 명령어를 만나 프로그램 정상종료
2	일반 오류 (err)	타입 오류 등 계속 진행은 가능한 정도의 오류

테스트 인터럽트 (예약됨)

코드	의미	비고
16	콘솔 입력	test.input(any q) -> str
17	콘솔 출력	test.print(any v)
18	파일 읽기	test.read(str path, int size) -> bytes
19	파일 쓰기	test.write(str path, str|bytes data)
20	현재 시각 가져오기	test.time() -> float
21	프로그램 일시정지	test.sleep(int|float t)

3-3. 데이터 명령어

데이터 명령어는 rodata와 data에서 상수와 전역변수의 값을 지시한다.

kasm	binary	설명	예시
DATA NONE	0x4e	none 타입. 크기 1B.	DATA NONE
DATA BOOL [V]	0x4200, 0x4201	bool 타입. 크기 2B.	DATA BOOL T
DATA INT [V]	0x49[8B]	int 타입. 크기 9B.	DATA INT 1557
DATA FLOAT [V]	0x46[8B]	float 타입. 크기 9B.	DATA FLOAT -2.718
DATA STRING [V]	0x53[4B][nB]	str 타입. 크기 n+5B.	DATA STRING EAB080
DATA BYTES [V]	0x43[4B][nB]	bytes 타입. 크기 n+5B.	DATA BYTES A73B

note. 모든 kasm 코드는 ASCII 문자로 되어 있으며 대소문자를 구분하지 않는다. .으로 섹션을 표시하며 ; 로 주석을 작성할 수 있다. kasm 코드와 실제 바이트코드는 거의 1:1 대응이 된다. 다만 후술할 LABEL 명령어는 NOP로 바뀌고 점프 위치가 심볼에서 라벨의 라인 위치로 바뀐다는 차이가 있다.	.RODATA DATA INT 3 ;CONST_0 .DATA DATA INT 0 ;GLOBAL_1 .TEXT LOAD MB CONST &0 STORE MB GLOBAL &1 ;ASSIGN HLT ;PROGRAM_END

3-4. 제어 명령어

note. 실행 명령어의 피연산자는 r, i16, i32가 있다. r은 레지스터 지정으로 MA 또는 MB를 쓸 수 있다. i16, i32는 정수인데 위치 심볼은 @, 메모리 주소는 &, 정수 값은 $가 앞에 붙는다. 예외적으로 변수를 지정할 때 i16은 const, global, local을 사용한다.
CALL MA $3 @7 ;0x1161030015060000, FORCOND MB LOCAL &-1 ;0x15626c00ffffffff

프로그램 제어 명령어는 전체 프로그램을 제어하거나 라벨 심볼이 된다.

kasm	binary	설명
HLT	0x00	인터럽트 코드 1로 프로그램 정지.
NOP	0x01	지나가기. (no operation)
LABEL @i32	0x01	심볼이 해당 라인의 위치가 된다. (점프시 심볼 **@정수**를 사용)

흐름 제어 명령어는 분기나 점프 등 실행 흐름을 제어한다.

kasm	binary	설명
INTR $i16 $i32	0x10	스택에서 i16번 pop하여 callmem으로 설정하고 코드 i32로 인터럽트. (외부호출 반환값은 MA에 세팅 필요)
CALL r $i16 @i32	0x11	함수 호출, 스택 프레임 생성. 함수명 r, 로컬변수 개수 i16, i32(함수 시작 위치)로 PC 설정.
RET $i16	0x12	반환값 MA에 세팅하고 PC, SP 복구. 스택 프레임 정리하고 인자 정리 (i16번 pop).
JMP @i32	0x13	PC를 i32(심볼 위치)로 설정.
JMPIFF @i32	0x14	스택에서 1개 값 pop하고 만약 False라면 PC를 i32(심볼 위치)로 설정.
FORCOND r i16 &i32	0x15	반복수 r, for변수 i16:i32로 루프 여부 구해서 push.
FORSET r i16 &i32	0x16	반복수 r, for변수 i16:i32로 반복자 구하고 MA에 세팅.

메모리 제어 명령어는 레지스터와 스택의 데이터를 제어한다.

kasm	binary	설명
LOAD r i16 &i32	0x20	레지스터 r에 스택 변수 i16:i32의 값을 세팅.
STORE r i16 &i32	0x21	스택 변수 i16:i32에 레지스터 r의 값을 세팅.
PUSH r	0x22	스택에 r을 push.
POP r	0x23	스택에서 pop한 값을 r에 세팅.
PUSHSET i16 &i32	0x24	스택에 스택 변수 i16:i32의 값을 push.
POPSET i16 &i32	0x25	스택에서 pop한 값을 스택 변수 i16:i32에 세팅.

3-5. 연산 명령어

산술 연산 명령어는 스택에서 값 2개를 pop하고 설명의 연산자에 해당하는 연산을 하여 스택에 push 한다. 즉 스택의 마지막 값 2개를 이항연산하여 다시 넣는다.

kasm	binary	설명
ADD	0x30	+ (더하기, 논리합)
SUB	0x31	- (빼기)
MUL	0x32	* (곱하기, 논리곱)
DIV	0x33	/ (나누기)
DIVS	0x40	// (나눗셈 몫)
DIVR	0x41	% (나눗셈 나머지)
POW	0x42	** (거듭제곱)

비교 연산 명령어는 산술 연산과 똑같이 스택의 값 2개를 이항연산하여 다시 넣지만, 비교작업을 하기 때문에 결과가 bool 타입이다.

kasm	binary	설명
EQL	0x50	== (동등)
EQLN	0x51	!= (동등하지 않음)
SML	0x52	< (작음)
GRT	0x53	> (큼)
SMLE	0x54	<= (작거나 같음)
GRTE	0x55	>= (크거나 같음)

3-6. 단축 명령어

단축 명령어는 기존 명령어 여러 개를 묶어 처리해야 했던 작업을 한 사이클로 처리하여 효율을 높인 명령어이다. 단축 명령어 없이도 작업은 가능하지만 속도가 조금 느려지며, 사용 여부는 컴파일 옵션으로 설정할 수 있다.

kasm	binary	설명
INC i16 &i32	0x60	스택 변수 i16:i32를 정수 1을 더한 값으로 세팅. (v++)
DEC i16 &i32	0x61	스택 변수 i16:i32를 정수 1을 뺀 값으로 세팅. (v--)
SHM i16 &i32	0x62	스택 변수 i16:i32를 정수 2를 곱한 값으로 세팅.
SHD i16 &i32	0x63	스택 변수 i16:i32를 정수 2로 나눈 값으로 세팅.
ADDI $i32	0x70	스택에서 값 하나를 pop하고 정수 i32를 더한 후 push.
MULI $i32	0x71	스택에서 값 하나를 pop하고 정수 i32를 곱한 후 push.
ADDR r $i32	0x72	레지스터 r에 정수 i32를 더한 후 push.
JMPI $i16 @i32	0x73	i16 값에 따라 MA, MB를 비교연산하고 False면 PC를 i32 (심볼 위치)로 설정. (1: ==, 2: !=, 3: <, 4: >, 5: <=, 6: >=)

4-1. 성능 테스트

테스트는 3회 시행 최고치를 기준으로 하며 표의 측정 단위는 M/s (4천만 회 계산), s (38th 피보나치 수 계산)이다. 컴퓨팅 환경은 다음과 같다.

운영체제	CPU	RAM	SSD	기온
windows11	i5-13600kf	32GiB	SK P31	22`C

언어별 속도 테스트는 실사용 시 체감할 속도 차이에 대한 테스트이다.

측정(상대)	golang	python	kscript -o (go -o)	kscript -o (py)
int	8635.6 (12088)	75.33 (105.4)	71.44 (**100.0**)	0.59 (0.826)
float	5185.4 (7258)	95.12 (133.1)	61.43 (85.99)	0.51 (0.714)
fib	0.0705 (4648)	1.979 (165.6)	3.277 (**100.0**)	331.6 (0.988)

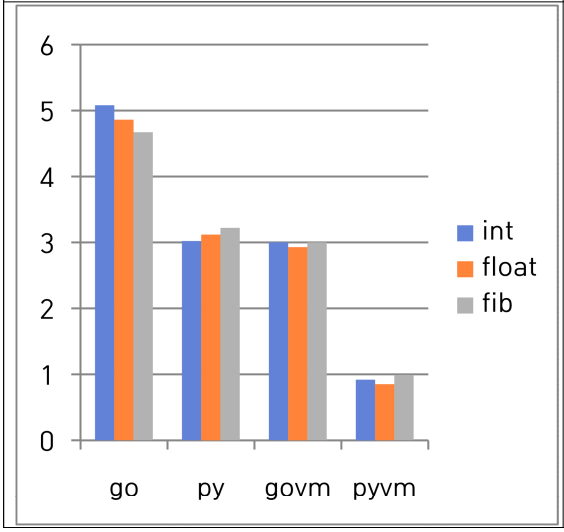
VM별 속도 테스트는 루프문과 런타임 구현에 따른 속도 차이를 테스트한다.

측정(상대)	kscript (go -o)	kscript -o (go -o)	kscript (py)	kscript -o (py)
int while	51.72 (72.40)	71.44 (**100.0**)	0.41 (0.574)	0.59 (0.826)
int for	40.62 (56.86)	49.14 (68.78)	0.39 (0.546)	0.53 (0.742)
float while	50.86 (71.19)	61.43 (85.99)	0.42 (0.588)	0.51 (0.714)
float for	40.45 (56.62)	44.65 (62.50)	0.40 (0.560)	0.46 (0.644)
fib	4.158 (78.81)	3.277 (**100.0**)	404.2 (0.811)	331.6 (0.988)

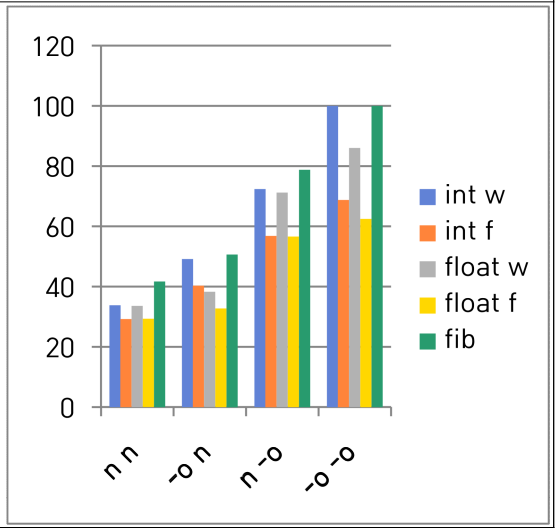
옵션별 속도 테스트는 컴파일과 런타임 옵션에 따른 속도 차이를 테스트한다.

측정(상대)	kscript (go)	kscript -o (go)	kscript (go -o)	kscript -o (go -o)
int while	24.17 (33.83)	35.13 (49.17)	51.72 (72.40)	71.44 (**100.0**)
int for	20.89 (29.24)	28.81 (40.33)	40.62 (56.86)	49.14 (68.78)
float while	24.01 (33.61)	27.32 (38.24)	50.86 (71.19)	61.43 (85.99)
float for	20.94 (29.31)	23.39 (32.74)	40.45 (56.62)	44.65 (62.50)
fib	7.853 (41.73)	6.463 (50.70)	4.158 (78.81)	3.277 (**100.0**)

언어별 속도 차이. (kscript OptAsm=True)



옵션별 속도 차이. (OptAsm, runtime opt)



4-2. 기본 외부함수

외부 상호작용 없이 바로 사용하는 기능들이다. [intr 32~100, abi 2]

`type(any) -> str`	6-type 중 하나의 값을 소문자로 반환.
`int(int|float|str|bytes) -> int`	정수로 내림, 문자열 해석, 리틀엔디안 디코딩.
`float(int|float|str) -> float`	실수화, 문자열 해석.
`str(any) -> str`	문자열 형식화, UTF-8 디코딩.
`bytes(int|str|bytes) -> bytes`	리틀엔디안, UTF-8 인코딩.
`hex(int|bytes) -> str`	16진법으로 나타낸 문자열 반환.
`chr(int) -> str`	해당 번호를 가지는 유니코드 문자 반환.
`ord(str) -> int`	유니코드 문자의 번호 반환.
`len(str|bytes) -> int`	문자열/바이트 길이 반환.
`slice(str|bytes, none|int, none|int) -> str|bytes`	문자열/바이트 슬라이싱.

4-3. 표준 외부함수

외부 입출력 기능이다. [intr 100~150, abi 4] - 공용 파일 핸들을 구현해야 한다.

`io.input(any) -> str`	stdout으로 출력 후 stdin으로 한 줄을 읽음.
`io.print(any, str)`	stdout으로 출력, 끝문자 수동지정.
`io.println(any)`	stdout으로 출력, 끝문자는 줄바꿈.
`io.error(any, str)`	stderr으로 출력, 끝문자 수동지정.
`io.open(str, str) -> str`	경로와 모드 설정해 파일 오픈.
`io.close(str)`	오픈했던 파일 닫음.
`io.seek(str, int, int) -> int`	기준점과 위치 설정해 읽기 위치 변경하고 반환.
`io.readline(str, int) -> str`	n줄 문자열 읽기.
`io.read(str, int) -> bytes`	n바이트 또는 전체 읽기.
`io.write(str, str|bytes)`	UTF-8 문자열 혹은 바이트 쓰기.

동적 할당을 구현한다. [intr 150~200, abi 8] - 공용 메모리 풀을 구현해야 한다.

`m.malloc(str, int)`	지정 이름에 변수 n개 크기의 공간 할당.
`m.free(str)`	지정 이름에 묶인 공간 해제.
`m.realloc(str, int)`	지정 이름에 공간 재할당. (원소 보존)
`m.len(str) -> int`	지정 이름에 묶인 공간의 크기 반환.
`m.set(str, int, any)`	n번째 원소 세팅.
`m.get(str, int) -> any`	n번째 원소 반환.
`m.split(str, str|bytes, str|bytes)`	지정 이름에 문자열/바이트를 나눈 결과로 채움.
`m.join(str, str|bytes) -> str|bytes`	지정 이름에 들어있는 문자열/바이트를 결합 내용을 사이에 두고 결합해 반환.

문자열 처리 기능이다. [intr 200~250, abi 16]

`str.change(str, bool) -> str`	대문자/소문자로 문자열 변환.
`str.find(str, str, bool) -> int`	문자열에서 하위 문자열 처음 나오는 위치 찾기.
`str.count(str, str) -> int`	문자열에서 하위 문자열 등장 횟수 반환.
`str.replace(str, str, str, int)`	문자열의 하위 문자열을 다른 문자열로 교체.

시간 관련 기능이다. [intr 250~300, abi 32]

t.stamp() -> str	현재 로컬 시간 기준 타임스탬프 반환.
t.time() -> float	현재 유닉스 시간 반환.
t.timef(str, int|float) -> str	유닉스 시간을 로컬 기준 포맷에 맞춰 반환.
t.sleep(int|float)	프로그램 실행 일시정지. (단위 : 초)

수학 함수들을 구현한다. [intr 300~350, abi 64]

math.const(str) -> float	수학 상수들의 값을 반환.
math.abs(int|float) -> int|float	절댓값을 반환.
math.ceil(int|float) -> int	올림한 정수를 반환. (크거나 같은 최소의 정수)
math.round(int|float) -> int	반올림한 정수를 반환. (기준 0.5)
math.log(int, int|float) -> float	n을 밑으로 하는 로그함수.
math.sin(int|float) -> float	호도법 기준 삼각함수. (sin)
math.cos(int|float) -> float	호도법 기준 삼각함수. (cos)
math.tan(int|float) -> float	호도법 기준 삼각함수. (tan)
math.random() -> float	0 이상 1 미만의 난수.
math.randrange(int, int) -> int	[n, m)인 정수 난수.

운영체제와 파일시스템 관련 기능들이다. [intr 350~400, abi 128]

os.name() -> str	운영체제 종류를 반환.
os.chdir(str)	현재 작업 경로 변경.
os.getcwd() -> str	현재 작업 경로 반환.
os.exists(str) -> bool	파일/폴더가 존재하는지 확인.
os.abspath(str) -> str	절대경로로 변환.
os.is(str, str) -> bool	경로가 파일/폴더인지 확인.
os.fsize(str) -> int	파일/폴더의 크기 반환.
os.fdate(str) -> int	파일/폴더의 수정 시각 반환.
os.mkdir(str)	폴더 생성.
os.rename(str, str)	파일/폴더 이름 바꾸기.
os.move(str, str)	파일/폴더 이동.
os.remove(str)	파일/폴더 삭제.

4-4. 기타 외부함수

화면/마우스 매크로 기능이다. [intr 1000~1100, abi 16384]

mouse.delay(int|float)	초 단위 작업 딜레이를 설정. (기본 0.1초)
mouse.screenshot(str, none|int, none|int, none|int, none|int)	스크린샷 파일 저장. none은 화면의 끝을 의미. (x_start, y_start, x_end, y_end)
mouse.size(bool) -> int	화면 크기 반환. x 또는 y.
mouse.pos(bool) -> int	마우스 좌표 반환. x 또는 y.
mouse.move(int|float, int|float, bool, none|int|float)	마우스를 상대위치 (x, y)로 이동. 드래그 여부, 이동시간 설정 가능.
mouse.moveto(int|float, int|float, bool, none|int|float)	마우스를 절대위치 (x, y)로 이동. 드래그 여부, 이동시간 설정 가능.
mouse.click(int, int|float, bool)	마우스 n번 클릭. 클릭 간격, 좌클릭 옵션.
mouse.scroll(int|float)	마우스 스크롤.

키보드 매크로 기능이다. [intr 1100~1200, abi 32768]

key.write(str, none|int|float)	키보드 문자열 입력, 입력 간격 설정 가능.
key.press(str, none|int|float)	키보드 글쇠 누름. 누르는 시간 설정 가능.
key.set(str, bool)	키보드 글쇠 누름/해제 세팅.
key.status(str) -> bool	특정 글쇠가 눌렸는지 감지.
key.event() -> str	입력된 키보드 이벤트 반환.
key.wait(str)	특정 키가 눌릴 때까지 대기.

note. 이 문서에 수록된 외부함수는 미구현 상태이며, 향후 세부사항이 변경될 수 있다.

quiz. 다음 프로그램이 하는 동작은 무엇인가?

line 1 ~ 25 <data>	line 26 ~ 55 <main>	line 56 ~ 92 <function>
		LABEL @1 ;FUNCTION_START
		PUSHSET LOCAL &-1
		INTR $1 $17 ;OUTERCALL
	.TEXT	LOAD MA LOCAL &-1
	NOP ;PROGRAM_START	LOAD MB CONST &4
.RODATA	LOAD MB CONST &8	JMPI $2 @4 ;CONDITION
DATA NONE ;CONST_0	STORE MB GLOBAL &14 ;ASSIGN	PUSHSET CONST &12
DATA BOOL T ;CONST_1	LOAD MB CONST &9	INTR $1 $17 ;OUTERCALL
DATA BOOL F ;CONST_2	STORE MB GLOBAL &15 ;ASSIGN	PUSHSET LOCAL &-1
DATA INT 0 ;CONST_3	LOAD MB CONST &4	PUSHSET GLOBAL &14
DATA INT 1 ;CONST_4	STORE MB GLOBAL &16 ;ASSIGN	DIVR
DATA FLOAT 0.000000 ;CONST_5	LOAD MB CONST &3 ;FOR_LOAD	PUSHSET CONST &3
DATA STRING ;CONST_6	LOAD MA CONST &10	EQL
DATA BYTES ;CONST_7	STORE MB GLOBAL &19	JMPIFF @5 ;CONDITION
DATA INT 2 ;CONST_8	STORE MA GLOBAL &20	PUSHSET LOCAL &-1
DATA INT 3 ;CONST_9	LABEL @3	PUSHSET GLOBAL &14
DATA INT 50 ;CONST_10	FORCOND MB GLOBAL &20	DIVS
DATA STRING 0A0A ;CONST_11	;FOR_CHECK	LOAD MA CONST &13
DATA STRING 202D3E20	JMPIFF @2	CALL MA $0 @1 ;INNERCALL
;CONST_12	FORSET MB GLOBAL &20	PUSH MA
DATA STRING 66 ;CONST_13	;FOR_ASSIGN	POPSET LOCAL &3
	STORE MB GLOBAL &17	RET $1 ;RETURN
.DATA	STORE MA GLOBAL &18	JMP @6
DATA INT 0 ;GLOBAL_14	PUSHSET GLOBAL &17	LABEL @5
DATA INT 0 ;GLOBAL_15	ADDI $1	PUSHSET GLOBAL &15
DATA INT 0 ;GLOBAL_16	LOAD MA CONST &13	PUSHSET LOCAL &-1
DATA INT 0 ;GLOBAL_17	CALL MA $0 @1 ;INNERCALL	MUL
DATA INT 0 ;GLOBAL_18	PUSHSET CONST &11	PUSHSET GLOBAL &16
DATA INT 0 ;GLOBAL_19	INTR $1 $17 ;OUTERCALL	ADD
DATA INT 0 ;GLOBAL_20	INC GLOBAL &19 ;FOR_ADD	LOAD MA CONST &13
	LOAD MB GLOBAL &19	CALL MA $0 @1 ;INNERCALL
	JMP @3	PUSH MA
	LABEL @2	POPSET LOCAL &3
	HLT ;PROGRAM_END	RET $1 ;RETURN
		LABEL @6
		LABEL @4
		RET $1 ;FUNCTION_END