Lua 5.3 Reference Manual
루아 5.3 레퍼런스 매뉴얼

들어가기 앞서

모든 번역은 제가 개인적으로 수행했고, 저는 전문 번역가가 아닙니다. 오역 및 기술적 오류가 있을 수 있습니다.
(제보 : mochanpowder@gmail.com)
원문 링크

- 비영리목적으로 제작된 글입니다. 수정 없이 출처와 함께 퍼나르는 것까지만 허용됩니다.
- 대응되는 한글 단어가 있는 영단어들은 가급적 한글을 사용하려 했으나, 해당 한글 단어의 사용 빈도가 영단어에 비해 현격히 떨어진다고 판단되는 경우 영단어 음독을 한글로 적었습니다.
- 만약 깔끔하게 대응되는 한글 단어가 없는 경우 영단어 음독을 한글로 적었습니다.
- 원문에서 괄호가 빈번하게 사용되어 가독성이 좋지 않습니다.
- 원문에서는 괄호로 끝나는 문장의 경우, 그 괄호가 끝난 뒤 말 마침표(예: (~~~).)를 사용하였지만, 이 글에서는 한글 맞춤법 부록의 소괄호 항목의 설명에 따라 괄호의 앞에 사용하는 것을 우선적으로 고려하였습니다.(예: .(~~~))
- 괄호가 문장의 일부로 판단되는 경우는 해당 괄호 이외의 말 마침표 등의 문장부호를 가급적 사용하지 않았습니다.

1. Introduction 소개

루아는 강력하고 효율적이며 가볍고 내장 가능한(Embeddable) 스크립팅 언어입니다. 절차적 프로그래밍, 객체 지향 프로그래밍, 기능 프로그래밍, 데이터 기반 프로그래밍 및 데이터 설명을 지원합니다.

루아는 간단한 절차적 구문과 연관 배열 및 확장 가능 구문을 기반으로하는 강력한 데이터 설명 구조를 결합합니다. 루아는 동적으로 타입이 지정되고, 레지스터 기반의 가상 머신으로 바이트 코드를 해석함으로써 실행되며, 증분(incremental) 가비지 컬렉션과 함께 자동 메모리 관리 기능을 제공하므로 구성, 스크립팅 및 신속한 프로토 타이핑에 이상적입니다.

루아는 표준 C 및 C ++의 공통 하위 집합 인 C로 작성된 라이브러리로 구현되었습니다. 루아 배포판에는 루아 (lua 이하 루아) 라 불리는 호스트 프로그램이 포함되어 있으며, 이는 루아 라이브러리를 사용하여 상호작용 또는 배치 목적의 완전한 독립실행형 루아 인터프리터를 제공합니다. 루아는 강력한, 가볍고, 임베디드가 가능한 스크립팅 언어와 강력하지만 가볍고 효율적인 독립형 언어 양쪽 모두로 사용될 수 있습니다.

확장 언어로써 루아는 "메인"프로그램에 대한 개념을 가지고 있지 않습니다. 그것은 임베디드 프로그램이라고 불리는 호스트 클라이언트 또는 단순히 호스트에 내장되어 작동합니다.(때때로 이 호스트는 독립 실행 형 루아 프로그램입니다) 호스트 프로그램은 루아 코드를 실행하기위한 함수를 호출 할 수 있고, 루아 변수를 읽고 쓸 수 있으며 루아 코드에 의해 호출 될 C 함수를 등록 할 수 있습니다. C 함수를 사용하여 루아를 확장하여 다양한 도메인에 대응할 수 있으므로 구문 프레임 워크를 공유하는 사용자 정의 프로그래밍 언어를 만들 수 있습니다.

루아는 자유 소프트웨어이며 라이선스에 명시된 바와 같이 보증 없이 제공됩니다. 이 매뉴얼에 설명 된 구현은 루아의 공식 웹 사이트 인 www.lua.org에서 가능합니다.

다른 참고 설명서와 마찬가지로 이 문서는 곳곳에서 매우 지루하고 딱딱합니다. 루아의 디자인에 대한 결정을 내리는 논의는 루아의 웹 사이트에서 제공되는 기술 문서를 참조하십시오. 루아에서의 프로그래밍에 대한 자세한 소개는 Roberto의 저서 Programming in Lua를 참조하십시오.

2. Basic Concepts 기본 개념

언어의 기본 개념을 설명합니다.

2.1. Values and Types 값과 형

루아는 동적으로 타입이 지정되는 언어입니다. 즉 어떤 변수도 유형을 가지고 태어나지 않습니다. 값만을 수행합니다. 언어에는 유형 정의가 따로 없습니다. 모든 값은 고유한 유형을 가집니다. 루아의 모든 값은 최상위(First-class) 값입니다. 즉 모든 값을 변수에 저장하고 다른 함수에 인수로 전달하여 결과로 반환할 수 있습니다.

루아에는 nil, boolean, number, string, function, userdata, thread 및 table의 8가지 기본 유형이 있습니다. nil 유형은 하나의 단일 값 nil을 가지며, 주 속성은 다른 값과 다릅니다. 보통 유용한 가치가 없음을 나타냅니다. boolean 형은 false와 true 의 두 가지 값을 가질 수 있습니다. nil과 false는 조건문을 거짓으로 만듭니다. 그 외의 값들은 참으로 만듭니다. number 형은 정수와 실수(부동소수점)를 모두 가리킵니다. string 형은 변경 불가능한 바이트 시퀀스를 나타냅니다. 루아는 8비트 클린입니다 : 문자열은 임베드된 0('\0')을 포함하여 8비트값만을 가질 수 있습니다. 루아는 또 인코딩 기법과 무관하게 동작합니다 ; 루아는 문자열

에 대해 어떤 가정도 하지 않습니다. number 형은 두 가지 내부적 표현을 사용하거나 두 개의 부속 유형을 가집니다. 하나는 정수(Integer)라 하고 다른 하나는 부동소수점(float)입니다. 루아는 각각의 표현이 언제 사용되어야 하는지 명시적 규칙이 있습니다만, 자유롭게 전환도 가능합니다(3.4.3 참조). 따라서 프로그래머는 정수와 부동 소수점의 차이점을 대부분 무시하거나 각 숫자의 표현을 완벽하게 제어할 수 있습니다. 표준 루아는 64비트 정수와 배정밀도(64비트) 부동소수점을 사용하지만, 32비트 정수 나 단정밀도(32비트) 부동소수점을 사용하도록 컴파일 할 수도 있습니다. 정수 및 부동 소수점 모두에 대해 32비트 옵션은 소형 시스템 및 임베디드 시스템에 특히 유용합니다.(luaconf.h 파일의 매크로 LUA_32BITS를 참조하십시오.)

루아는 루아에서 작성된 함수와 C로 작성된 함수를 호출(및 조작)할 수 있습니다.(3.4.10 참조) 둘 다 Function 형으로 표현됩니다.

userdata 형은 임의적인 C 데이터가 루아 변수에 저장되도록 해줍니다. userdata 값은 원시 메모리 블록을 나타냅니다. userdata는 두 종류가 있습니다. 루아가 관리하는 메모리 블록이 있는 객체인 전체(full) userdata와 단순한 C 포인터 값인 라이트(light) userdata가 그것입니다. userdata는 할당과 객체판별 테스트 연산자 외에 루아에서 사전정의된 연산자가 없습니다. 프로그래머는 메타테이블을 이용해 full userdata 값에 대한 연산자를 정의할 수 있습니다.(2.4 참조) userdata 값은 루아에서 생성되거나 수정될 수 없고 C API를 통해서만 가능합니다. 이는 호스트 프로그램의 데이터 무결성을 보증합니다.

thread 형은 명령 수행의 독립적인 쓰레드를 나타내며 코루틴을 구현하는데 사용됩니다.(2.6 참조) 루아의 쓰레드는 운영체제의 쓰레드와 관련이 없습니다. 루아는 쓰레드를 자체적으로 지원하지 않는 시스템일지라도 코루틴을 지원합니다.

테이블(table, 이하 테이블) 형은 연관 배열, 즉 숫자 뿐만 아니라 모든 루아 값(nil 과 NaN 제외)으로 인덱싱 될 수 있는 배열을 구현합니다. 테이블은 여러 종류로 이루어질 수 있습니다. 즉 모든 유형의 값을 포함할 수 있습니다.(nil 제외) 값이 nil인 키는 테이블의 일부로 간주되지 않습니다. 반대로 테이블의 일부가 아닌 키는 연관된 값이 nil 입니다.

테이블은 루아에서 유일한 데이터 구조화 메커니즘입니다. 일반적인 배열이나 리스트, 심볼 테이블, 세트, 레코드, 그래프, 트리 등을 나타내는데 사용될 수 있습니다. 레코드를 표현하기 위해 루아는 필드 이름을 인덱스로 사용합니다. 언어는 a.name을 a["name"]의 신택스 슈거로 제공합니다. 루아에서 테이블을 생성하는 여러 방법이 있습니다(3.4.9 참조)

인덱스와 마찬가지로 테이블 필드의 값들은 어떤 유형이든 될 수 있습니다. 특히, 함수는 최상위 값이기 때문에 테이블의 필드에 보관될 수 있습니다. 따라서 테이블은 메서드를 보관할 수도 있습니다.

테이블의 인덱싱은 언어의 raw equality 정의를 따릅니다. a[i]와 a[j]는 i와 j가 원시적으로 동일할 경우(그리고 그럴 경우에만) 동일한 테이블 요소를 나타냅니다.(metamethod는 예외) 특히 정수로 떨어지는 값을 가지는 부동소수점형은 그에 대응하는 정수 값과 동일합니다.(예 : 1.0 == 1) 모호함을 피하기 위해 키로 사용되는 정수값을 가진 임의의 부동소수점은 각각의 정수로 변환됩니다. 예를 들어 [2.0] = true 라고 작성하면 테이블에 삽입된 실제 키는 정수 2가 됩니다. 반면, 2와 "2"는 다른 루아 값이므로 다른 테이블 항목을 나타냅니다.

테이블, function, thread, userdata 값은 객체입니다. 변수는 실제로 이러한 값을 포함하지 않으며 참조값만을 가집니다. 할당이나 매개변수의 전달, 함수의 반환값들은 언제나 각각의 값들에 대한 참조를 조작합니다. 어떤 종류의 사본도 만들어내지 않습니다.

라이브러리 함수 유형은 주어진 값의 유형을 설명하는 문자열을 반환합니다.(6.1 참조)

2.2. Environments and the Global Environment 환경과 전역 환경

뒤에 3.2와 3.3.3에서 이야기하겠지만, 자유 이름(어떤 선언에도 바인딩되지 않은 이름) var는 _ENV.var로 변환됩니다. 게다가 모든 청크(chunk 이하 청크)는 _ENV라는 외부 로컬 변수의 범위에서 컴파일되므로(3.3.2 참조) _ENV 자체는 결코 청크에서 자유 이름이 아닙니다.

이 외부 _ENV 변수의 존재와 자유 이름의 변환에도 불구하고 _ENV는 완전히 정규화된 이름입니다. 특히, 새 변수와 매개 변수를 그 이름으로 정의할 수 있습니다. 자유 이름에 대한 각각의 참조는 루아의 일반적인 가시규칙에 따라 프로그램의 해당 지점에서 볼 수 있는 _ENV를 사용합니다(3.5 참조).

_ENV의 값으로 사용되는 모든 테이블을 환경(environment)라고 합니다.

루아는 전역 환경이라 불리는 고유의 환경을 유지합니다. 이 값은 C 레지스트리의 특수 색인에 보관됩니다.(4.5 참조) 루아에서 전역 변수 _G가 동일한 값으로 초기화됩니다.(_G는 절대 내부적으로 사용되지 않습니다)

루아가 청크를 로드할 때 _ENV 상위 값의 초기값은 전역 환경입니다.(load로드 부분 참조) 그러므로 기본적으로 루아 코드의 자유 이름은 전역 환경의 항목을 참조하므로 전역 변수라고도 합니다. 또한 모든 표준 라이브러리는 전역 환경에 로드되며 일부 표준 라이브러리는 해당 환경에서 작동합니다. load 또는 loadfile을 사용하여 다른 환경의 청크를 로드할 수 있습니다.(C에서는 청크를 로드한 뒤 첫번째 상위값의 값을 변경해야 합니다.)

2.3. Error Handling 오류 핸들링

루아는 임베디드 확장 언어이기 때문에 모든 루아의 행동은 호스트 프로그램의 C 코드가 루아 라이브러리의 함수를 호출함으로써 시작됩니다.(독립실행형 루아일 경우 루아 응용프로그램이 호스트 프로그램입니다) 루아의 청크를 컴파일 할 때나, 실행할 때 오류(Error에러)가 발생하면 제어가 호스트로 리턴되어 적절한 조치를 취할 수 있습니다.(예를 들어 오류 메시지를 찍는다던지)

루아 코드는 error 함수를 호출하여 명시적으로 오류를 생성할 수 있습니다. 루아에서 에러를 캐치(catch)해야 할 때 pcall이나 xpcall을 사용하여 보호 모드에서 주어진 함수를 호출할 수 있습니다.

오류가 있을 때마다, 오류 객체(오류 메시지라고도 함)가 오류의 내용과 함께 전파됩니다. 루아 자체는 에러 오브젝트가 문자열인 에러만을 생성하지만, 프로그램은 에러 객체로서 어떤 값이든 생성할 수 있습니다. 이러한 오류 객체를 처리하는 것은 루아 프로그램이나 그 호스트에 달려있습니다.

xpcall 또는 lua_pcall 을 사용할 때 오류가 발생할 경우 메시지 처리기를 호출할 수 있습니다. 이 함수는 원래 오류 객체와 함께 호출되며 새 오류 객체를 반환합니다. 에러가 스택을 풀어주기 전에 호출되어, 스택을 검사하고 스택 트레이스백을 생성하는 등의 에러에 대한 더 많은 정보를 수집할 수 있습니다. 이 메시지 처리기는 여전히 보호된 호출에 의해 보호됩니다. 따라서 메시지 처리기 내부의 에러는 또 다른 메시지 처리기를 호출할 것입니다. 이 루프가 너무 오래 지속되면 루아는 그것을 깨뜨리고 적절한 메시지를 반환합니다. (메시지 처리기는 일반 런타임 오류에 대해서만 호출됩니다. 메모리 할당 오류나 finalizer가 실행되는 동안의 에러에는 호출되지 않습니다.)

2.4. Metatables and Metamethods 메타테이블과 메타함수

루아의 모든 값은 메타테이블을 가질 수 있습니다. 이 메타테이블은 특정 연산 하에서 원래 값의

행동을 정의하는 평범한 루아 테이블 입니다. 메타테이블에 특정 필드를 설정하여 값에 대한 연산 동작의 여러 측면을 변경할 수 있습니다. 예를 들어 숫자가 아닌 값이 덧샘의 피연산자인 경우 루아는 그 값의 메타테이블에서 "__add"필드의 함수를 확인합니다. 찾으면 루아는 그 함수를 호출하여 더하기를 수행합니다. 메타테이블에서 각각의 이벤트에 해당하는 키는 두 개의 언더스코어(_) 뒤에 붙는 이벤트 이름으로 구성된 문자열입니다. 이에 대응하는 값을 메타함수라고 합니다. 앞서의 예제에서 키는 "__add"였고 메타함수가 덧셈을 대신 수행했습니다. 어떤 값이든 getmetatable 함수를 사용하여 메타테이블을 쿼리할 수 있습니다. 루아는 원시 액세스(rawget 참조)를 사용하여 메타 테이블에서 메타함수를 쿼리합니다. 따라서 객체 o에서 이벤트 ev에 대한 메타함수를 검색하기 위해 루아는 다음 코드와 동등한 작업을 수행합니다.

```
rawget(getmetatable(o) or {}, "__ev")
```

setmetatable 함수를 사용하여 테이블의 메타테이블을 바꿀 수 있습니다. 루아 코드 이외의 타입은 C API를 사용하지 않고는 메타테이블을 바꿀 수 없습니다.(디버그 라이브러리 (§6.10) 사용 제외)

테이블과 전체 userdata에는 개별 메타테이블이 붙어있습니다.(또한 여러 테이블과 userdata는 메타테이블을 공유할 수 있습니다.) 다른 모든 유형의 값은 유형별로 하나의 메타테이블만을 공유합니다. 즉, 모든 숫자에 대해 하나의 메타테이블, 모든 문자열에 대해 하나의 메타테이블이 있는 식입니다. 기본적으로 값에는 메타테이블이 없지만 문자열 라이브러리는 문자열 유형에 대한 메타테이블을 설정합니다.(6.4 참조)

메타테이블은 오브젝트가 사칙연산, 비트 연산, 순서 비교, 연결성, 길이 연산, 호출 및 인덱싱에서 어떻게 동작하는지를 제어합니다. 또한 메타테이블은 userdata나 테이블이 가비지 컬렉션 될 때 불릴 함수를 정의할 수 있습니다.

단항연산자(음수 부호, 길이 및 비트연산 NOT)의 경우에 메타함수가 계산되어 첫 번째 것과 동일한 두 번째 더미 피연산자로 호출됩니다. 이 여분의 피연산자는 내부 루틴을 단순화(이 연산자들이 바이너리 연산처럼 동작하게 함으로써)하기 위한 것이며 이후 버전에서는 제거될 수 있습니다. 대부분의 경우 이 여분의 피연산자는 부적절합니다.

메타테이블이 컨트롤 할 수 있는 세세한 이벤트의 리스트는 아래에 기술합니다. 각각의 연산은 키로 구별됩니다.

- __add : (addition)덧셈(+) 연산입니다. 이 연산자의 피연산자가 숫자(또는 숫자로 치환할 수 있는 문자열)이 아닐 경우, 루아는 메타함수를 호출합니다. 먼저 루아는 첫 번째 피연산자(적절한 형태라고 해도)를 확인합니다. 만약 첫 번째 피연산자가 __add에 정의된 메타함수가 없다면 루아는 두 번째 피연산자를 확인합니다. 메타함수가 있을 경우 그 메타함수를 호출하여 두 개의 피연산자를 매개 변수로 넣어줍니다. 호출의 반환값은 (한 개로 조정되어) 연산의 결과값이 됩니다. 이 외의 경우에는 오류를 일으킵니다.
- __sub : (subtraction)빼기(-) 연산입니다. 덧셈과 유사하게 동작합니다.
- __mul : (multiplication)곱하기(*) 연산입니다. 덧셈과 유사하게 동작합니다.
- __div : (division)나누기(/) 연산입니다. 덧셈과 유사하게 동작합니다.
- __mod : (modulo)나머지(%) 연산입니다. 덧셈과 유사하게 동작합니다.
- __pow : (exponentiation)지수(^) 연산입니다. 덧셈과 유사하게 동작합니다.
- __unm : (unary negation)반대부호(단항 -) 연산입니다. 덧셈과 유사하게 동작합니다.
- __idiv : (integer division)나머지를 버리는 나누기(//) 연산입니다. 덧셈과 유사하게 동작합니다.
- __band : (bitwise and) AND(&) 비트연산입니다. 어떤 피연산자도 정수 또는 정수로 치환될 수 있는 값이 아닐 때 루아는 메타함수를 시도합니다. 덧셈과 유사하게 동작합니다.
- __bor : (bitwise or) OR(|) 비트연산입니다. AND 비트연산과 유사하게 동작합니다.
- __bxor : (bitwise xor)XOR(이항연산자 ~) 비트연산입니다. AND 비트연산과 유사하게

동작합니다.

- __bnot : (bitwise not)NOT(단항연산자 ~) 비트연산입니다. AND 비트연산과 유사하게 동작합니다.
- __shl : (shift left)왼쪽 쉬프트(<<) 비트 연산입니다. AND 비트연산과 유사하게 동작합니다.
- __shr : (shift right)오른쪽 쉬프트(>>) 비트 연산입니다. AND 비트연산과 유사하게 동작합니다.
- __concat : (concatenation)연속성(..)연산입니다. 어떤 피연산자도 정수 또는 정수로 치환될 수 있는 값이 아닐 때 루아는 메타함수를 시도합니다. 덧셈과 유사하게 동작합니다.AND 비트연산과 유사하게 동작합니다.
- __len : (length)길이(#)연산입니다. 객체가 문자열이 아닐 경우, 루아는 메타함수를 시도합니다. 메타함수가 있을 경우 그 메타함수를 호출하여 객체를 매개 변수로 넣어줍니다. 호출의 반환값은 (한 개로 조정되어) 연산의 결과값이 됩니다. 만약 메타메서드가 없고 객체가 테이블이라면 루아는 table length 함수를 수행합니다(3.4.7 참조). 이 외의 경우에는 오류를 일으킵니다.
- __eq : (equal)동일성(==)연산입니다. 덧셈과 유사하게 동작합니다. 단, 비교되는 두 값이 모두 테이 이거나 모두 full userdata이고, 원시적으로 동일하지 않다면 루아는 메타함수를 불러냅니다. 호출의 결과는 boolean 값으로 변환됩니다.
- __lt : (less than)미만(<) 연산입니다. 덧셈과 유사하게 동작합니다. 단, 비교하고자 하는 두 값이 모두 숫자이거나 모두 문자열인 경우 외에 메타함수를 시도합니다. 호출의 결과는 boolean 값으로 변환됩니다.
- __le : (less equal)이하(<=) 연산입니다. 다른 연산자와 다르게 이하 연산자는 두 가지 다른 이벤트를 사용할 수 있습니다. 먼저 루아는 미만(<) 연산자의 경우처럼 두 피연산자의 __le 메타함수를 찾습니다. 메타함수가 없다면 a <= b가 not(b < a)일 것으로 가정하고 __lt 메타함수를 시도합니다. 다른 비교연산자와 마찬가지로 결과는 boolean 값으로 변환됩니다.(__lt 이벤트의 사용은 앞으로의 버전에서 제거될 수 있습니다. 또한 실제 __le 메타함수보다 느리게 동작합니다.)
- __index : ***table[key]*** 에 접근하는 인덱싱입니다. 예시에서 ***table***이 테이블이 아니거나 ***key***가 테이블에 존재하지 않으면 일어납니다. 메타함수는 테이블에서 검색됩니다. 이름에도 불구하고 이 이벤트의 메타함수는 함수도, 테이블도 될 수 있습니다. 만약 함수라면 테이블과 key를 매개 변수로 호출될 것입니다. 반환값은 (한 개로 조정되어) 연산의 결과값이 됩니다. 만약 테이블이라면 최종 결과는 그 테이블을 key로 인덱싱한 결과가 됩니다.(이러한 인덱싱은 정규적으로 동작하고 원시적 처리가 아니기 때문에 또다른 메타함수를 불러낼 수 있습니다.)
- __newindex : ***table[key] = value*** 와 같은 형식으로 인덱싱을 할당하는 이벤트입니다. __index 이벤트처럼 이 이벤트도 ***table***이 테이블이 아니거나 ***key***가 테이블에 존재하지 않으면 일어납니다. 인덱싱처럼 이 이벤트에 대한 메타함수도 함수 또는 테이블이 될 수 있습니다. 테이블이라면 루아는 같은 ***key***와 ***value***을 가지고 인덱스 할당을 시도합니다.(이 할당은 정규적으로 동작하고 원시적 처리가 아니기 때문에 또 다른 메타함수를 불러낼 수 있습니다.)
- __call : 함수 호출 연산자 ***func(args)***입니다. 이 이벤트는 루아에서 함수가 아닌 값을 호출하려 할 때(즉, ***func***가 함수가 아닐 때) 발생합니다. 메타함수가 ***func***에서 조회됩니다. 메타함수가 조회되면 ***func***을 첫번째 인수로 사용하고 원래 호출의 인수를 사용하여 메타 메서드를 호출합니다. 모든 호출의 반환값은 연산의 결과값이 됩니다.(여러개의 결과를 허용하는 유일한 메타함수입니다)

어떤 객체에 메타테이블을 추가하기 전에 필요한 모든 메타함수를 테이블에 먼저 추가해두는 것이 좋은 방법입니다. 특히 __gc 메타함수는 이 순서를 지켰을 경우에만 동작합니다(2.5.1 참조)

메타테이블은 정규 테이블이기 때문에, 위에 정의한 이벤트 외에도 임의의 필드를 가질 수 있습니다. 표준 라이브러리의 일부 함수(예를들어 tostring)는 메타테이블의 다른 필드를 그 만의 목적으로

사용합니다.

2.5. Garbage Collection

루아는 자동 메모리 관리를 수행합니다. 이는 새 객체에 메모리를 할당하는 것이나 필요없는 객체를 해제할 걱정을 할 필요가 없다는 말입니다. 루아는 메모리 자동 관리를 위해 가비지 컬렉터를 실행하여 모든 죽은 객체의 메모리를 해제합니다.(죽은 객체란 루아에서 더 이상 접근할 수 없는 객체를 말합니다.) 루아가 사용하는 모든 메모리는 자동 관리의 대상이 됩니다.(문자열, 테이블, userdata, 함수, thread, 내부 구조체 등등)

루아는 증분형 마크 앤 스윕(mark-and-sweep) 컬렉터를 구현합니다. 이는 가비지 컬렉트의 주기를 제어하는데 두 개의 수를 사용합니다 : garbage-collector pause 와 garbage-collector step multiplier입니다. 둘 다 단위로 백분율 점수를 사용합니다. (예 : 값 100은 내부적으로 값 1을 의미합니다)

가비지 컬렉터의 pause는 컬렉터가 새 주기를 시작하기 전에 기다리는 시간을 제어합니다. 값이 클수록 컬렉터는 덜 공격적입니다. 값이 100보다 작으면 컬렉터는 새 주기가 시작되기를 기다리지 않습니다. 값이 200일 때는 사용 중인 총 메모리가 주기가 시작된 시점에서 두배가 되길 기다립니다.

가비지 컬렉터의 step multiplier는 메모리 할당에 상대적으로 컬렉터의 속도를 제어합니다. 높은 값이 주어질 수록 컬렉터가 더 공격적으로 움직이지만 각 증분 단계의 크기가 커집니다. 컬렉터가 너무 느려 순환을 완료하지 못할 수 있으므로 100보다 작은 값을 사용해서는 안됩니다. 기본값은 200이며 컬렉터가 메모리 할당 속도의 "두 배"로 실행됨을 의미합니다.

step multiplier를 매우 큰(프로그램에서 사용할 수 있는 최대 바이트 수의 10% 보다 큰) 수로 설정하면 컬렉터는 프로그램의 실행을 중단시키는 컬렉터(stop-the-world collector)처럼 동작합니다. 그런 다음, pause를 200으로 설정하면 컬렉터는 구버전의 루아처럼 동작하며 루아의 메모리 사용량이 두 배가 될때마다 전체 컬렉션을 수행합니다. C에서 lua_gc를 호출하거나 루아에서 collectgarbage를 호출하면 이 값들을 변경할 수도 있고, 컬렉터를 수동으로 제어할 수도 있습니다.(예 : 컬렉터를 멈췄다가 재시작)

2.5.1. Garbage-Collection Metamethods 가비지 컬렉션 메타함수

2.5.2. Weak Tables

2.6. Coroutines 코루틴

루아는 코루틴(협업 멀티 쓰레딩collaborative multithreading이라고도 불리는)을 지원합니다. 루아의 코루틴은 독립적인 실행 쓰레드를 나타냅니다. 그러나 다중 쓰레드 시스템의 쓰레드와 달리 코루틴은 명시적으로 yield 함수를 호출하여 실행을 일시 중단합니다.

coroutine.create를 호출하여 코루틴을 만듭니다. 유일한 인수는 코루틴의 주요 기능인 함수입니다. create 함수는 새로운 코루틴을 생성하고 핸들을 반환합니다(thread 유형의 객체). 이것만으로는 코루틴이 시작되지 않습니다.

coroutine.resume을 호출하여 코루틴을 실행합니다. coroutine.resume을 처음 호출하면 coroutine.create가 반환한 스레드의 첫 번째 인수로 전달되며 코루틴은 main 함수를 호출하여 실행을 시작합니다. coroutine.resume에 전달된 추가 인수는 해당 함수에 인수로 전달됩니다. 코루틴이 실행된 이후엔 완료되거나 yield 될 때까지 계속 수행됩니다.

코루틴은 두 가지 방법으로 종료시킬 수 있습니다. 일반적으로 주 함수가 리턴할 때(마지막 명령 이후에 명시적으로 또는 암시적으로), 그리고 보호되지 않은 상황에서 오류가 발생했다면 비정상적으로 종료됩니다. 일반적인 종료에서 coroutine.resume은 true를 반환하며, 거기에 덧붙여 coroutine.main 함수가 반환한 값이 반환됩니다. 오류가 발생하면 coroutine.resume 은 false와 오류 객체를 반환합니다.

코루틴은 coroutine.yield를 호출하여 제어를 넘겨줍니다. 코루틴이 yield되면 그 yield가 중첩된 함수 호출에서 불러졌더라도 해당하는 resume이 즉시 반환됩니다.(main 함수가 아니라 직접 또는 간접적으로 main 함수가 호출하는 함수) yield의 경우에 coroutine.resume 또한 true를 반환하며 yield에 전달된 모든 값을 반환합니다. 다음번에 동일한 코루틴을 다시 시작할 때는 마지막으로 yield한 지점에서 이어서 시작하며, resume에 전달 된 추가 인수를 반환하면서 해당 지점에서 실행을 계속합니다.

coroutine.create처럼 coroutine.wrap 함수 또한 코루틴을 만들지만, 이는 코루틴 자체를 반환하는 대신 호출되면 코루틴을 resume하는 함수를 반환합니다. 이 함수에 전달된 모든 매개 변수는 coroutine.resume에 추가 매개 변수로 전달됩니다. coroutine.wrap은 coroutine.resume이 반환하는 모든 값들 중 첫번째 값만을 제외한 나머지를 반환합니다.coroutine.resume과 달리 coroutine.wrap은 오류를 캐치하지 않습니다. 어떤 오류이든 호출자에게 전파될 것입니다.

코루틴이 어떻게 동작하는지 예시로 아래의 코드를 살펴보세요.

```
function foo (a)
      print("foo", a)
      return coroutine.yield(2*a)
    end

    co = coroutine.create(function (a,b)
          print("co-body", a, b)
          local r = foo(a+1)
          print("co-body", r)
          local r, s = coroutine.yield(a+b, a-b)
          print("co-body", r, s)
          return b, "end"
    end)

    print("main", coroutine.resume(co, 1, 10))
    print("main", coroutine.resume(co, "r"))
    print("main", coroutine.resume(co, "x", "y"))
    print("main", coroutine.resume(co, "x", "y"))
```

실행 결과는 다음과 같습니다.

```
co-body 1       10
    foo     2
    main    true    4
    co-body r
    main    true    11      -9
    co-body x       y
    main    true    10      end
    main    false   cannot resume dead coroutine
```

C API를 통해서 역시 코루틴을 생성하고 조작할 수 있습니다. 함수 lua_newthread, lua_resume, lua_yield 의 설명을 보세요.

3. The Language 언어

이 절에서는 어휘, 구문, 루아의 문맥에 대해 알아봅니다. 다시 말해, 어떤 토큰이 적합하고 이들이 어떻게 조합되며 그 조합의 의미가 무엇인지에 대해 설명합니다.

언어 구조는 일반적인 베커스-나우르 표기법의 확장이라고 설명할 수 있습니다. {a}는 0개 이상의 a를 의미하고 [a]는 선택적 a를 의미합니다. 비 터미널은 비 터미널처럼 표시되고 키워드는 kword처럼 표시되며 다른 터미널 기호는 = 처럼 표시됩니다. 루아의 완전한 문법은 이 매뉴얼의 끝에 있는 9절에서 확인할 수 있습니다.

3.1. Lexical Conventions 어휘 규칙

루아는 자유형 언어입니다. 이름과 키워드 사이의 구분자를 제외하고 공백(줄바꿈 포함)과 어휘 요소(토큰)사이의 주석을 무시합니다.

루아에서는 이름(식별자라고도 불리는)이 문자와 숫자, 언더스코어로 이루어 질 수 있지만, 숫자로 시작해선 안되고, 예약어(reserved word)를 사용해선안됩니다. 식별자는 변수와 테이블 필드, 레이블에 이름을 붙이는데 사용됩니다. 다음 키워드들은 예약되어있어, 이름으로 사용할 수 없습니다.

```
and       break     do        else      elseif    end
false     for       function  goto      if        in
local     nil       not       or        repeat    return
then      true      until     while
```

루아는 언제나 대소문자를 구별합니다. and는 예약어이지만 And는 사용할 수 있는 이름입니다. 규칙에 따라 프로그램은 언더스코어로 시작하여 하나 이상의 대문자가 이어지는 이름의 사용은 피해야 합니다.(예 : __VERSION)

아래 문자들은 다른 토큰을 가리킵니다.

```
+     -     *     /     %     ^     #
&     ~     |     <<    >>    //
==    ~=    <=    >=    <     >     =
(     )     {     }     [     ]     ::
;     :     ,     .     ..    ...
```

짧은 리터럴 문자열은 작은 따옴표나 큰 따옴표를 매칭하여 구분할 수 있으며 다음과 같은 C 스타일의 이스케이프 시퀀스를 포함할 수 있습니다.

```
'\a'      (벨)
'\ n'     (줄 바꿈)
'\ r'     (캐리지 리턴)
'\ t'     (가로 탭)
'\\'      (백 슬래시)
```

'\''　　(따옴표 [큰 따옴표])
'\ '　　(아포스트로피 [작은 따옴표])

백 슬래시 다음에 줄 바꿈이 있으면 문자열에 개행 문자가 생깁니다.
이스케이프 시퀀스 '\z'는 다음 공백 문자의 범위를 건너뜁니다.(줄 바꿈 포함)

짧은 리터열 문자열에 어떤 바이트라도 그 숫자값으로 담을 수 있습니다.(임베드된 0 포함) 이것은 이스케이프 시퀀스 \xXX를 사용하여 수행할 수 있습니다 여기서 XX는 정확히 두 개의 16진수 시퀀스이거나 이스케이프 시퀀스 \ddd입니다. 여기서 ddd는 최대 세 자리 10진수 시퀀스입니다. (10진수 이스케이프 문자열 뒤에 숫자가 오는 경우 정확히 3자리 숫자로 표현해야 합니다.)

유니 코드 문자의 UTF-8 인코딩은 이스케이프 문자열 \u{XXX} (괄호가 의무적으로 사용되어야 합니다.)에서, XXX는 문자 코드를 나타내는 하나 이상의 16진수 시퀀스입니다.

리터럴 문자열은 긴 괄호로 묶인 긴 형식을 사용하여 정의할 수도 있습니다. 우리는 n 레벨을 여는 긴 괄호를 대괄호와 그 뒤에 따라붙는 n개의 등호 표시로 정의합니다. 그러므로 레벨 0을 의미하는 여는 괄호 표시는 [[이 되고, 레벨 1은 [=[레벨 2는 [==[인 식입니다. 닫는 괄호도 비슷하게 동작합니다. 예를 들어 레벨 4의 닫는 괄호는]====]인 식입니다. 긴 리터럴은 여는 긴 괄호로 시작해 동일 레벨의 닫는 괄호에서 끝납니다. 이 괄호로 묶인 형식의 리터럴은 여러 행에 대해 실행될 수 있으며 모든 이스케이프 시퀀스를 해석하지 않으며 다른 모든 수준의 긴 대괄호를 무시합니다. 모든 종류의 줄 끝 시퀀스(캐리지 리턴, 개행,캐리지 리턴 다음의 개행 또는 개행 다음의 캐리지 리턴)는 간단한 개행으로 변환됩니다.

편의상 긴 괄호를 여는 즉시 개행문자가 오면 그 개행 문자는 문자열에 포함되지 않습니다. 예를 들어 ASCII를 사용하는 시스템에서 ('a'는 97로 코드화되고 개행문자는 10으로 코드화되고 '1'은 49로 크드화됨) 아래의 5개 리터럴 문자열은 동일한 문자열을 나타냅니다.

```
a = 'alo\n123"'
a = "alo\n123\""
a = '\97lo\10\04923"'
a = [[alo
    123"]]
a = [==[
    alo
    123"]==]
```

앞서 설명한 규칙에 의해 리터럴 문자열의 어떤 바이트도 명시적으로 영향받지 않습니다. 그러나 루아는 파일을 파싱할 때 텍스트 모드에서 하고, 시스템 파일 함수는 일부 제어 문자에 문제가 있을 수 있습니다. 따라서 텍스트가 아닌 문자에 대해 명시적인 이스케이프 시퀀스를 사용하여 텍스트가 아닌 데이터를 인용된 리터럴로 나타내는 것이 더 안전합니다.

숫자 상수는 선택적 분수 부분과 문자 'e' 또는 'E'로 표시되는 선택적 십진 지수로 작성할 수 있습니다. 루아는 또한 0x 또는 0X로 시작하는 16진수 상수를 받아들입니다. 16진수 상수는 선택적인 분수 부분과 문자 'p' 또는 'P' 로 표시된 선택적 2진 지수를 허용합니다. 기수점 또는 지수가 있는 숫자 상수는 부동소수점을 나타냅니다. 그 외에는 값이 정수에 들어맞으면 정수를 나타냅니다. 유효한 정수 상수의 예는 다음과 같습니다.

```
3   345   0xff   0xBEBADA
```

유효한 부동소수점의 예는 다음과 같습니다.

```
  3.0     3.1416     314.16e-2     0.31416E1     34e1
    0x0.1E  0xA23p-4   0X1.921FB54442D18P+1
```

문자열 바깥에서 두 개의 하이픈(--)으로 주석을 작성할 수 있습니다. --뒤에 곧바로 여는 대괄호만 오지 않는다면 주석은 '짧은 주석'이 되어 해당 라인에서 끝납니다. 여는 대괄호가 따라붙으면 이는 '긴 주석'이 되어 대응하는 닫는 괄호가 나타날때까지 지속됩니다. 긴 주석은 코드의 일부를 막을 때 자주 사용됩니다.

3.2. Variables 변수

변수는 값을 저장하는 장소입니다. 루아에는 세 종류의 변수가 있습니다. 전역변수, 지역변수, 테이블 필드가 그것입니다.

단일 이름은 전역 변수 또는 지역 변수를 나타냅니다.(또는 특정 종류의 지역 변수인 함수 형식 매개변수입니다)

```
var ::= Name
```

이름은 3.1에서 정의한 바와 같이 식별자를 의미합니다.

모든 변수는 지역이라는 명시적 선언이 없는 한 전역으로 간주됩니다. 지역 변수는 어휘의 범위가 한정됩니다. 지역변수의 범위 내에 있는 함수는 이에 자유롭게 접근할 수 있습니다.(3.5 참조)

변수의 최초 할당 이전에는 값이 nil입니다.

대괄호는 테이블을 인덱스하는데 사용됩니다.

```
var ::= prefixexp '[' exp ']'
```

테이블 필드에 대한 접근 수단은 메타테이블을 통해 변경할 수 있습니다. 인덱싱된 변수 t[i]에 대한 접근은 함수 gettable_event(t,i)와 동등합니다. (gettable_event 함수에 대해서는 2.4 참조. 이 함수는 루아에서 정의되거나 호출 가능하지 않습니다. 이곳에서만 설명 목적으로 사용합니다.)

구문 var.Name은 var["Name"]에 대한 신택스 슈거입니다.

```
var ::= prefixexp '.' Name
```

전역변수 x 에 대한 접근은 _ENV.x와 동등합니다. chunk가 컴파일되는 방식에 의해, _ENV는 전역 이름이 아닙니다(2.2 참조)

3.3. Statements 문장

루아는 C나 파스칼과 비슷한 거의 전통적인 문장 셋을 지원합니다.이 셋은 할당과 조정 구조체, 함수 호출, 변수 선언을 포함합니다.

3.3.1. Blocks 블록

블록은 순서대로 실행되는 문장의 목록입니다.

```
block ::= {stat}
```

루아에는 문장을 세미콜론으로 나누어, 빈 문장을 사용할 수 있습니다. 세미콜론으로 블록을 시작하거나 순서대로 두 개의 세미콜론을 씁니다.

```
stat ::= ';'
```

함수 호출과 할당이 여는 소괄호로 시작될 수 있습니다. 이는 루아 문법의 모호성을 가져올 수 있습니다. 다음 부분을 살펴보세요.

```
a = b + c
(print or io.write)('done')
```

문법상 이는 두 가지로 해석될 수 있습니다.

```
a = b + c(print or io.write)('done')
a = b + c; (print or io.write)('done')
```

현재의 파서는 여는 괄호를 호출 선언의 시작으로 보기 때문에 언제나 전자처럼 해석합니다. 이러한 모호성을 피하기 위해, 괄호로 시작하는 문장은 언제나 세미콜론을 앞에 두는 것이 좋은 습관입니다.

```
;(print or io.write)('done')
```

블록은 명시적으로 구분하여 단일 명령문을 생성할 수 있습니다.

```
stat ::= do block end
```

명시적 블록들은 변수 선언의 범위를 제어하는데 유용합니다. 또한 명시적 블록은 다른 블록의 중간에 `return` 문을 추가하는데 사용되기도 합니다.(3.3.4 참고)

3.3.2. Chunks 청크

루아 컴파일의 단위는 청크라고 불립니다. 구문론적으로 청크는 단순히 블록일 뿐입니다.

```
chunk ::= block
```

루아는 가변인수(3.4.11)가 있는 익명 함수의 본문으로 청크를 처리합니다. 따라서 청크는 지역 변수를 정의하고 인수를 받고 값을 반환할 수 있습니다. 게다가 그러한 익명 함수는 `_ENV`(2.2 참조)라고 하는 외부 로컬 변수의 범위에서 컴파일됩니다. 결과 함수는 해당 변수를 사용하지 않더라도 항상 유일한 상위값으로 `_ENV`를 가집니다.

3.3.3. Assignment 할당

루아는 다중할당을 지원합니다. '변수 목록'을 정의하는 할당 구문은 왼쪽에 위치하고 '값 목록'은 오른쪽에 위치합니다. 양쪽 리스트는 모두 각 요소를 쉼표(컴마 ,)로 구분합니다.

```
stat ::= varlist '=' explist
varlist ::= var { ',' var}
explist ::= exp { ',' exp}
```

표현식은 3.4에서 또 다룹니다.

할당 이전에 값의 목록은 변수 목록의 길이에 맞게 '조정'됩니다. 만약 값 목록의 항목 수가 변수 목록의 항목 수보다 많다면 남는 값들은 버려지고, 반대로 모자란다면 나머지 변수들은 nil로 채워집니다. 만약 값 목록이 함수 호출로 끝난다면 '조정'이 일어나기 전에 해당 함수의 모든 반환값들이 값 목록에 들어갑니다.(3.4 참조)

할당 문장은 모든 표현식을 먼저 평가한 뒤에야만 동작합니다. 코드는 이렇습니다.

```
i = 3
i, a[i] = i + 1, 20
```

a[i]에 들어있는 i는 4가 할당되기 전에 3으로 평가되기 때문에, a[4]에 영향을 미치는 일 없이 a[3]에 20을 할당합니다. 유사하게 아래 라인

```
x, y = y, x
```

은 x 와 y의 값을 서로 교체합니다. 또 아래 라인

```
x, y, z = y, z, x
```

은 원형으로 x, y, z의 값들을 순환시킵니다.

전역 변수와 테이블 필드의 할당 동작은 메타테이블을 통해 바뀔 수 있습니다. 인덱싱된 변수 t[i]에 대한 할당 t[i] = val 은 settable_event(t,i,val)과 동등합니다.(settable_event 함수에 대해서는 2.4를 참조. 이 함수는 루아에서 정의되거나 호출 가능하지 않습니다. 이곳에서만 설명 목적으로 사용합니다.)

전역 이름 x에 대한 할당 x = val 은 _ENV.x = val 과 동등합니다.(2.2 참조)

3.3.4. Control Structures 제어문

제어문 if, while, repeat는 전형적인 의미와 친숙한 구문을 가집니다.

```
stat ::= while exp do block end
stat ::= repeat block until exp
stat ::= if exp then block {elseif exp then block}[else block] end
```

루아는 for 문도 두 가지 형태로 지원합니다.(3.3.5 참조)

제어문 중 조건문에는 어떤 값이 들어가도 괜찮습니다. false와 nil은 모두 거짓으로 판단합니다. 그 이외의 모든 값들은 true로 판단합니다.0과 텅 빈 문자열까지도 그러합니다.

repeat-until 반복문에서는 내부 블럭이 until 키워드에서 끝나지 않고 조건문 이후에 종료됩니다. 그러므로 조건문은 루프 내부에서 선언된 지역 변수를 참조할 수 있습니다.

goto 문은 프로그램의 제어를 레이블(label, 이하 레이블)로 전환합니다. 구문상의 이유로 루아의 레이블은 문장으로 간주됩니다.

```
stat ::= goto Name
stat ::= label
label ::= '::' Name '::'
```

레이블의 가시범위는 정의된 블록 전체에서 똑같은 이름의 레이블이 정의된 중첩 블록과 중첩 함수를 제외한 나머지 영역입니다. goto 문은 지역 변수의 범주에 들어가지 않은 가시적인 레이블로 점프할 수 있습니다.

레이블과 빈 문장은 아무 동작도 하지 않으므로 void 문장이라고 부릅니다.

break 문은 while, repeat, for 문을 중단시키고 해당 반복문의 바로 다음 문장을 실행시킵니다.

```
stat ::= break
```

break는 가장 가까운 반복문에만 적용됩니다.

return 문은 함수나 청크(chunk, 익명함수)에서 값을 반환할 때 사용됩니다. 함수는 하나 이상의 값을 반환할 수 있기 때문에 return 구문의 문장은 아래와 같습니다.

```
stat ::= return [explist][';']
```

return 문은 블럭의 마지막 문장으로만 사용되어야 합니다. 블럭의 중간에서 return을 사용할 필요가 있다면, 명시적 내부 블록을 사용해 idion do return end 와 같이, 내부 블록의 마지막 문장으로 사용할 수 있습니다.

3.3.5. For Statement 포 문

for 문은 수치적인 것과 제너릭인 것, 두 가지 형태가 있습니다.

수치적 for문은 제어변수가 산술 연산 과정을 거치는 동안 코드 블록이 반복됩니다. 다음과 같은 구문을 가집니다.

```
stat::= for Name '=' exp1 ',' exp2 [',' exp3] do block end
```

위에서 Name이 exp1에서 시작하여 exp2를 통과하면 블럭(block)을 실행하고, 끝나면 exp3을 실행합니다. 보다 정확히 말하자면,

```
for v = exp1, exp2, exp3 do block end
```

위 for 문은 아래 코드와 동등합니다.

```
do
      local var, limit, step = tonumber(e1), tonumber(e2), tonumber(e3)
      if not (var and limit and step) then error() end
      var = var - step
      while true do
        var = var + step
        if (step >= 0 and var > limit) or (step < 0 and var < limit) then
          break
        end
        local v = var
```

```
    block
  end
end
```

아래 내용을 참고하세요.

- 세 개의 제어식 `e1`, `e2`, `e3`은 모두 반복문이 시작하기 전 단 한번만 평가됩니다. 모두 결과값으로 숫자를 내보내야 합니다.
- `var`, `limit`, `step` 은 보이지 않는 변수입니다. 여기에선 설명만을 위해 사용했습니다.
- 세 번째 표현식이 생략되어있을 땐 `1` 증가가 사용됩니다.
- `break` 나 `goto` 문으로 반복문을 빠져나갈 수 있습니다.
- 반복문 변수 `v` 는 반복문 본문에서 지역변수입니다. 반복문 바깥에서 이 값이 필요한 경우 반복문 내부에서 다른 전역변수에 할당해서 사용하세요.

제너릭 `for`문은 반복자라 불리는 함수를 사용한다. 각각의 반복에서 반복자 함수는 새 값을 생성할 때 호출되고, 새 값이 `nil`일 때 정지된다. 제너릭 `for`문은 다음과 같은 구문을 가진다.

```
stat ::= for namelist in explist do block end
namelist ::= Name{',' Name}
```

아래 `for` 문

for *var_1, …… , var_n* in *explist* do *block* end

은 다음 코드와 동등하다.

```
do
  local f, s, var = explist
  while true do
    local var_1, var_2 ···, var_n = f(s, var)
    if var_1 == nil then break end
    var = var_1
    block
  end
end
```

아래 내용을 참고하세요.

- 표현식 *explist*는 단 한번만 평가됩니다. 결과값은 반복자 함수, 문장, 첫번째 반복자 변수의 초기값입니다.
- `f`,`s`,`var`는 보이지 않는 변수입니다. 여기에선 설명만을 위해 사용했습니다.
- `break` 문을 사용해 반복문을 빠져나갈 수 있습니다.
- 반복문 변수 `var_1`,`var_2` … 등은 반복문 본문에서 지역변수입니다. 반복문 바깥에서 이 값이 필요한 경우 반복문 내부에서 다른 전역변수에 할당해서 사용하세요.

3.3.6. Function Calls as Statements 문장으로 함수 호출

부가 효과를 허용하기 위해 함수 호출은 문장으로 실행될 수 있습니다.

```
stat ::= functioncall
```

이 경우, 반환된 모든 값들은 버려집니다. 함수 호출은 3.4.10에서 다시 한번 설명합니다.

3.3.7. Local Declarations 지역 변수 선언

지역 변수는 블럭 내 어디서든 선언될 수 있습니다. 선언은 초기 할당을 포함할 수 있습니다.

```
stat ::= local namelist ['=' explist]
```

초기 할당이 존재한다면 다중 할당과 똑같은 문법을 따릅니다. (3.3.3 참조) 그 외에는 초기화된 모든 변수는 nil입니다.

청크도 또한 블럭이므로(3.3.2 참조), 지역 변수가 명시적 블럭 바깥의 청크 안에서 선언될 수 있습니다.

지역 변수의 가시성 규칙은 3.5에 설명되어있습니다.

3.4. Expressions 표현식

루아의 기본적 표현식은 아래와 같다.

```
exp ::= prefixexp
exp ::= nil | false | true
exp ::= Numeral
exp ::= LiteralString
exp ::= functiondef
exp ::= tableconstructor
exp ::= '...'
exp ::= exp binop exp
exp ::= unop exp
prefixexp ::= var | functioncall | '(' exp ')'
```

숫자와 리터럴 문자열은 3.1에서 설명하고 있습니다. 변수는 3.2, 함수 정의는 3.4.11, 함수 호출은 3.4.10, 테이블 생성자는 3.4.9에서 설명하고 있습니다. '...'으로 표현한 vararg 표현식은 vararg 함수 내부에서 직접 사용해야 합니다. 이 부분은 3.4.11에서 더 설명하고 있습니다.

이항 연산자는 산술 연산자(3.4.1 참조)와 비트 연산자(3.4.2 참조), 관계 연산자(3.4.4 참조), 논리 연산자(3.4.5), 연결 연산자(3.4.6)로 구성됩니다. 단항 연산자는 단항 마이너스(3.4.1), 비트연산자 NOT(3.4.2), 단항 길이 연산자(3.4.7 참조)가 있습니다.

3.4.1. Arithmetic Operators 산술 연산자

루아는 아래 산술 연산자들을 지원합니다.

- `+` : 더하기
- `-` : 빼기
- `*` : 곱하기
- `/` : 나누기
- `//` : 나눈 뒤 버림
- `%` : 나눈 나머지
- `^` : 지수
- `-` : 음수부호

지수와 부동소수점 나눗셈을 제외한 나머지 산술 연산자들은 다음과 같이 동작합니다. : 좌항과 우항이 모두 정수라면 연산은 정수로 이루어지고 결과도 정수가 나옵니다. 그 외에 좌항과 우항이 숫자로 치환될 수 있는 문자열(3.4.3 참조)이라면 부동 소수점으로 변환되어, 일반적인 부동 소수점 산술 규칙(대개의 경우 IEEE 754 표준)에 따라 수행되고 결과도 부동 소수점입니다.

지수와 부동 소수점 나눗셈은 언제나 피연산자를 부동 소수점으로 변환하고, 결과도 언제나 부동 소수점입니다. 지수는 ISO C 함수 pow를 사용하기 때문에 정수가 아닌 지수에서도 동작합니다.

`%` 연산은 나눗셈의 나머지(소수점 버림)로 정의됩니다. 정수 산술에서 오버플로우(overflow)가 일어날 경우, 2의 보수 연산의 일반적 규칙에 따라 연결된 모든 연산들이 랩-어라운드(wrap around)됩니다. 즉, 수학적 결과에 modulo 2^{64}와 동일한 고유 표현 가능 정수를 반환합니다.

3.4.2. Bitwise Operators 비트 연산자

루아는 아래 비트 연산자들을 지원합니다.

- `&` : 비트연산 AND
- `|` : 비트연산 OR
- `~` : 비트연산 exclusive OR(XOR)
- `>>` : 오른쪽 쉬프트(shift)
- `<<` : 왼쪽 쉬프트
- `~` : 단항 연산자 NOT

모든 비트연산자는 좌, 우 피연산자를 정수로 치환(3.4.3참조)합니다. 좌, 우 정수의 모든 비트에 대해 연산을 수행한 뒤 정수 값을 반환합니다.

왼쪽과 오른쪽 쉬프트는 공백 비트를 0으로 채웁니다. 음수로 쉬프트를 하면 반대방향으로 비트가 이동합니다. 정수 형식의 비트 수와 같거나 더 큰 값으로 쉬프트시키면 모든 비트가 이동하기 때문에 값이 0이 됩니다.

3.4.3. Coercions and Conversions 치환과 변환

루아는 일부 타입 간의 런타임 자동 변환을 지원합니다. 비트 연산자는 언제나 부동 소수점형 피연산자를 정수형으로 변환합니다. 지수와 부동 소수점 나눗셈은 언제나 정수형 피연산자를 부동 소수점형으로 변환합니다. 모든 나머지 산술 연산자는 정수형과 부동 소수점형 피연산자가 주어졌을 때 정수를 부동 소수점으로 변환하며, 이를 통상적 규칙(usual rule)이라 부릅니다. C API는 필요에 따라 양 쪽의 정수형을 부동 소수점형으로 변환하거나 그 반대쪽 변환도 합니다.