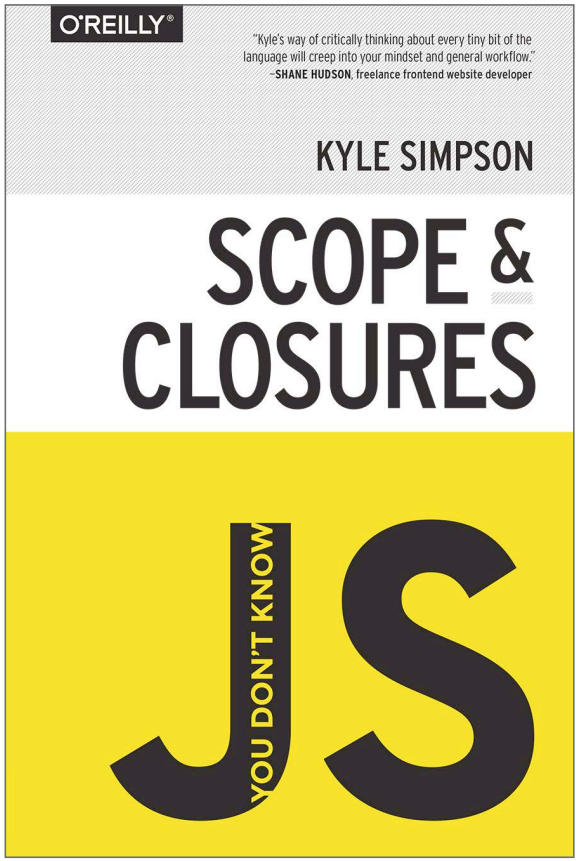

Hanbit eBook
Realtime 81
You Don't Know JS
스코프와 클로저
SCOPE & CLOSURES
카일 심슨 지음 / 최병현 옮김
O'REILLY
한빛미디어
Hanbit Media, Inc.



이 도서는 O'REILLY의
You don't know JS: Scope & Closures의
번역서입니다.

You Don't Know JS **스코프와 클로저**

초판발행 2014년 08월 29일

지은이 카일 심슨 / **옮긴이** 최병현 / **펴낸이** 김태헌
펴낸곳 한빛미디어(주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부
전화 02-325-5544 / **팩스** 02-336-7124
등록 1999년 6월 24일 제10-1779호
ISBN 978-89-6848-633-3 15000 / **정가** 12,000원

책임편집 배용석 / **기획** 김창수 / **편집** 김창수, 정지연
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 최송실
영업 김형진, 김진불, 조유미 / **마케팅** 박상용, 김옥현

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.
한빛미디어 홈페이지 www.hanbit.co.kr / **이메일** ask@hanbit.co.kr

Published by HANBIT Media, Inc. Printed in Korea Copyright © 2014 HANBIT Media, Inc.
Authorized Korean translation of the English edition of Scope and Closures, ISBN 9781449335588 © 2014 Getify Solutions, Inc.. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.
이 책의 저작권은 오라일리 사와 한빛미디어(주)에 있습니다.
저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 무단 전재를 금합니다.

지금 하지 않으면 할 수 없는 일이 있습니다.
책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanbit.co.kr)로 보내주세요.
한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

저자 소개

지은이_ **카일 심슨**Kyle Simpson

텍사스 오스틴 출신의 오픈 웹 전도사인 카일 심슨은 자바스크립트, HTML5, 실시간 P2P 통신과 웹 성능에 열정적인 관심을 갖고 있다. 열정이 없었다면, 이런 작업에 이미 진력이 났을 것이다. 그는 저술가이자 워크숍 강사와 기술 연사이며, 오픈 소스 커뮤니티에서도 열심히 활동하고 있다.

역자 소개

역자_ **최병현**

서강대학교에서 컴퓨터공학을 전공했다. 현재 인문과 IT 분야 번역 프리랜서로 일하고 있다.

저자 서문

독자 여러분도 알겠지만, 이 책 시리즈 명의 'JS'는 자바스크립트를 욕하는 단어가 아니다. 물론 자바스크립트에 별나게 짜증 나는 점이 있다는 것은 모두 동의하겠지만 말이다!

웹이 사용된 이래, 자바스크립트는 사용자들이 상호소통하며 콘텐츠를 이용할 수 있게 해주는 기반 기술이었다. 자바스크립트가 비록 마우스가 움직인 궤적을 반짝이게 하거나 짜증 나는 팝업 경고창을 띄우는 용도에서 시작했지만, 개발 후 거의 20년이 지난 지금은 비할 바 없이 성장했다. 세계에서 가장 널리 이용되는 소프트웨어 플랫폼이 웹이 된 상황에서 그 누구도 자바스크립트의 중요성을 의심하지 않는다.

그러나 하나의 언어로서 자바스크립트는 끊임없이 많은 비판의 대상이었다. 이는 부분적으로는 과거의 유산 때문이지만, 대부분은 자바스크립트의 '디자인 철학' 때문이다. 심지어 브랜든 아이츠Brendan Eich가 예전에 말했듯, 이름마저도 형인 '자바'의 '멍청한 동생' 같은 느낌을 준다. 물론 이름이야 그저 정치와 마케팅의 우연이었을 뿐이다. 이 두 언어는 여러 중요한 부분에서 엄청난 차이가 있다. '자바스크립트'와 '자바'는 '노트'와 '노트북'만큼이나 별 관련이 없다.

위풍당당한 C 스타일의 절차적 방식과 절묘하고 번하지 않은 Scheme/Lisp 스타일의 함수적 방식을 포함해 자바스크립트는 개념과 문법 표현양식을 여러 언어에서 빌려왔기 때문에 많은 방면의 개발자는 물론 심지어 프로그래밍 경험이 없는 이들도 자바스크립트를 굉장히 쉽게 이용할 수 있다. 'Hello World'를 출력해 보면 알겠지만, 자바스크립트는 매우 쉽고 금세 익숙해질 수 있다.

자바스크립트는 아마 작업하기 가장 쉬운 언어 중 하나겠지만, 얄궂은 데가 있어서

다른 언어보다 완전히 숙달한 사람은 훨씬 적다. C나 C++ 같은 언어로 완전한 프로그램을 짜려면 상당한 수준의 깊은 지식이 필요하지만, 자바스크립트는 언어에 대한 표면적인 지식만 겨우 알더라도 완전한 프로그램을 만들어낼 수 있다(사실 많은 자바스크립트 프로그램이 이렇게 작성됐다).

자바스크립트의 고급 개념은 콜백 함수를 전달하는 것과 같이 겉으로 보기에는 단순하게 보인다. 그래서 자바스크립트 개발자는 그저 보이는 대로 기능을 사용하고 기능이 수면 아래에서 어떻게 작동하는지는 별로 고민하지 않는다.

자바스크립트는 간단하고 쉽게 사용할 수 있어서 많은 이에게 매력적인 언어이지만, 세심히 탐구하지 않으면 경험 많은 자바스크립트 개발자도 제대로 이해하지 못할 정도로 복잡하고 미묘한 언어 메커니즘을 가지고 있다.

바로 이 부분이 자바스크립트의 모순이자 아킬레스건이고, 우리가 다룰 도전 과제다. 자바스크립트는 이해하지 않고도 사용할 수 있으므로 굳이 이해하려고 노력하지 않는 경우가 많다.

목적

자바스크립트를 다루면서 예상치 못한 결과가 나올 때마다 블랙리스트에 넣고 이용하는걸 꺼린다면, 결국에는 자바스크립트의 풍부한 기능 중 오직 일부만을 쓰게 될 것이다. 보통 이렇게 쓰이는 부분을 자바스크립트의 '좋은 부분'이라고 부른다. 그렇지만 필자는 감히 말하겠다. 그것은 그저 '쉬운 부분', '안전한 부분', 또는 '불완전한 부분'일 뿐이다.

『You Don't Know JS』 시리즈는 다른 종류의 도전과제를 제공한다. 자바스크립트의 모든 것을 깊게 배우고 이해해보자. '힘든 부분'까지도, 아니 특히 그 부분을 말이다.

이 책에서 필자는 자바스크립트 개발자가 간신히 일을 해나갈 수 있을 정도만 배우려는 경향, 자바스크립트의 작동 방식과 원리를 전혀 알려고 하지 않는 그 경향을 분명하게 비판할 것이다. 거친 길은 돌아가라지만, 이 책을 쓰는 데는 이런 말을 따르지 않을 것이다.

왜 그런지 알지도 못하면서 그저 작동한다고 만족할 수는 없다. 자바스크립트를 제대로 배우려면 여러분도 그래야 한다. 부탁하건대, 이 울퉁불퉁하고 "남들이 가지 않았던 길"을 걸어서 자바스크립트의 모든 것을 배워보시라. 그 어떤 테크닉, 프레임워크, 전문용어도 이 지식의 범위를 넘어서지 않는다.

『You Don't Know JS』 시리즈는 자바스크립트의 핵심 요소 중 가장 착각하기 쉽고 이해하기 어려운 부분을 깊고 철저하게 파고든다. 다 알고 있을 것이란 자신감은 버리고 책을 읽는 게 좋을 것이다. 이론적인 부분만이 아니라 실용적인 '알아야 할 필요가 있는' 부분에 대해서도 말이다.

많은 개발자가 자바스크립트를 불확실하게 알고 있는 이들에게 배운다. 그렇게 배운 자바스크립트는 진짜 자바스크립트의 껍데기에 불과하다. 『You Don't Know JS』 시리즈를 파고든다면 진정한 자바스크립트를 배우게 될 것이다. 계속 읽어보길 바란다. 자바스크립트가 여러분을 기다리고 있으니 말이다.

리뷰

자바스크립트는 멋진 언어다. 배우기 쉽지만, 완전하게 또는 충분히 배우기는 아주 어렵다. 보통 개발자가 작업하다가 헷갈리면 자신의 이해가 부족한 탓을 하기보다는 언어 탓을 한다. 이 시리즈는 바로 이 점을 바꾸려는 것이다. 이 시리즈를 읽고 나면 자바스크립트의 진면목이 이해되므로 더는 그런 혼란이 없을 테니까.

이 책의 많은 예제는 ECMA.Script version 6(ES6) 같은 최신의 자바 엔진 환경에서 실행될 것을 가정한다. 그래서 몇몇 코드는 이전 환경(ES6 이전)에서 제대로 작동하지 않을 수 있다.

추천사

어릴 적 나는 오래된 휴대전화, hi-fi 스테레오 같은 잡동사니를 잡히는 대로 분해하고 조립하면서 놀았다. 이런 기기를 사용하기에 너무 어렸지만, 망가진 기기를 볼 때마다 기기가 어떻게 작동했는지 이해하려고 노력했다.

언젠가 오래된 라디오의 회로판을 보았던 기억이 난다. 그 회로판에는 구리선이 감긴 이상하게 긴 튜브가 있었다. 튜브의 역할을 알아내지는 못했지만, 바로 연구에 들어갔다. 튜브는 무슨 역할을 하지? 튜브는 왜 라디오 안에 들어 있지? 튜브는 회로판의 다른 부분과 다르게 생겼는데, 왜 그럴까? 왜 구리선이 튜브를 감고 있는 걸까? 구리선을 벗겨내면 어떤 일이 생길까? 지금은 그 튜브가 루프 안테나고, 페라이트 막대에 구리선을 감아서 만들며 트랜지스터 라디오에 많이 사용된다는 것을 안다.

당신은 왜 그럴까라는 질문의 답을 찾기 위해 골몰해본 적이 있는가? 대다수 아이는 이런 호기심이 있다. 사실 내가 아이들을 좋아하는 가장 큰 이유도 아이들의 배우고자 하는 욕구일 것이다.

다행인지 불행인지, 나는 이제 전문가라고 불리며 무언가를 만드는 일을 한다. 어릴 적에는 분해한 기기들을 언젠가 내가 만들 수 있길 바랐다. 물론, 지금 내가 만드는 것들은 페라이트 막대가 아니라 자바스크립트로 만들지만 말이다. 뭐, 결국은 비슷한 일이긴 하다! 예전에는 무언가를 만들고 싶다고 생각했지만, 지금은 무언가를 분석하고 이해하는 것을 더 좋아한다. 그래서 문제를 해결하거나 버그를 고치는 가장 좋은 방법을 찾아내는 데 골몰했지만, 정작 내 도구에 대해서는 신경 쓰지 못했다.

바로 이점 때문에 『You Don't Know JS』 시리즈를 정말 좋아한다. 맞는 말이다. 나는 자바스크립트를 모른다. 자바스크립트를 매일 매일 수십 년간 사용했지만, 정말 자바스크립트를 이해하냐고 묻는다면 아니라고밖에 대답할 수 없다. 물론, 자바

스크립트의 많은 부분을 이해하고 많은 관련 설명서와 인터넷 문서를 읽는다. 그럼에도 모든 것을 이해하길 바랐던 어릴 적 바람만큼 완전하게 자바스크립트를 이해하지는 못한다.

스코프와 클로저는 『You Don't Know JS』 시리즈를 시작하기에 아주 좋은 주제다. 특히 나 같은 사람에게 아주 유용한 주제다(당신에게도 그렇길 빈다). 자바스크립트를 한 번도 사용하지 않은 이들에게 가르칠 만한 내용은 아니지만, 이 주제를 통해 자바스크립트 사용자들이 이 언어가 내부적으로 어떻게 작동하는지를 이해할 수 있을 것이다. 이 책은 참 적절한 시기에 나왔다. 이제 ES6는 정착되고 있고, 여러 웹 브라우저에서 잘 구현되고 있다. 아직 let이나 const 같은 새로운 기능을 익힐 시간을 내지 못했던 이들에게 이 책은 좋은 입문서가 될 것이다.

이 책이 도움되길 바라고, 무엇보다 여러분이 자바스크립트의 세세한 부분이 어떻게 작동하는지에 대한 저자 카일의 중요한 지적들을 전반적 사고와 작업 과정에 녹여 내기를 바란다. 그저 안테나를 사용할 것이 아니라, 안테나가 어떻게 작동하는지 이해하길 바란다.

셰인 허드슨Shane Hudson

www.shanehudson.net

역자 서문

이 책의 주요 독자 중에는 전문 프로그래머로서 훈련받은 이들이 많을 것이고, 아마도 C나 C++ 같은 언어를 능숙하게 사용할 줄 아는 이들도 적지 않을 것이다.

C나 C++를 처음 배웠을 때를 떠올려 보자. 다짜고짜 "Hello World" 출력을 배우는 경우도 있겠지만, 기초를 튼튼히 다지면서 배웠다면 함수의 선언, 형태, 입출력, 지역변수와 전역변수의 차이, 컴파일 과정 등도 익혔을 것이다. 이것은 언어의 규범을 익히는 과정이고, 해당 언어로 쓰인 코드 한 줄 한 줄이 어떤 역할을 하는지 이해해 나가는 과정이다. 그리고 이런 언어의 규범을 익혀야 그 언어를 한계까지 최대한 활용할 수 있다.

재귀호출은 고급 알고리즘을 구현하는 데 필수다. 그런데 재귀호출과 그 과정에서 생성되는 변수들의 스택이 어떤 식으로 작동하는지 이해하지 못하면 사용할 수 없다.

C나 C++ 컴파일러가 상당히 엄격하게 굴기는 하지만, 일반적으로 C나 C++ 사용자들은 이런 규범을 익히는 과정을 거치는 경우가 많다. 하지만 자바스크립트는 어떤가? 기본적으로 상당히 쉽게 배울 수 있는 언어라서 대학에서 배우는 경우도 드물고 배우더라도 언어 자체의 구조와 작동 방식을 배우기보다는 자바스크립트를 가지고 브라우저에서 무언가 동작시키는 실습 수업이 대부분이다. 직장에서 배우는 경우는 말할 것도 없다. 늘 촉박하게 결과를 내야 하는 기업에서 한가하게 언어의 구조를 고찰하고 있을 수는 없다(더군다나 배우기 어렵지도 않으니 말이다).

자바스크립트 엔진은 상당히 너그러워서 '적당히' 코딩해도 오류를 뿜어내는 일 없이 '적당히' 처리해준다. 그래서 자바스크립트를 사용하며 "뭘 했는지는 모르겠지만, 어쨌든 원하던 결과는 나왔어!"라고 약간은 찜찜해하며 넘어간 적이 있을 것이다. 아마도 이런 특성 때문에 자바스크립트를 학교에서도 실습에 사용하고 기업에

서도 아무나 시켜서 빨리 배우라고 할 수 있는 것이리라(그리고 많은 이들이 자바스크립트를 욕하는 이유기도 하다).

그러나 저자 카일은 『You Don't Know JS』 시리즈에서 자바스크립트를 사용할 때도 C나 C++ 같은 아니 어쩌면 더 까다로운 규범을 익혀야 한다고 말한다. 더 깔끔하고 더 안전한 코드를 작성하고 자바스크립트의 능력을 최대한 활용하기 위해 알아야 할 것이 있다는 것이다. 자바스크립트를 수년째 사용하는 프로그래머로서 또 이 책의 번역자로서, 언어의 한계를 최대한 활용하여 효율적인 코딩을 하길 좋아하는 고급 프로그래머라면 이 책을 즐겁게 읽을 것이다.

예제 파일

이 책에서 사용하는 예제 코드는 http://bit.ly/1c8HEWF에서 내려받을 수 있습니다.

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 쉽지 않습니다. 또한 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1. **eBook Only - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.**

   500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 보다 쉽게 집필할 수 있는 기회가 될 수 있으리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2. **무료로 업데이트되는, 전자책 전용 서비스입니다.**

   종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나 저자(역자)와 독자가 소통하면서 보완하여 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3. 독자의 편의를 위하여 DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT 기기에서 자유롭게 활용할 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해 독자 여러분이 언제 어디서 어떤 기기를 사용하더라도 편리하게 전자책을 볼 수 있도록 하기 위함입니다.

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

지금 보시는 전자책에 소유권한을 표시한 문구가 없거나 타인의 소유권한을 표시한 문구가 있다면 위법하게 사용하고 있을 가능성이 높습니다. 이 경우 저작권법에 의해 불이익을 받으실 수 있습니다.

다양한 기기에 사용할 수 있습니다. 또한 한빛미디어 사이트에서 구입하신 후에는 횟수에 관계없이 다운받으실 수 있습니다.

한빛미디어 전자책은 인쇄, 검색, 복사하여 붙이기가 가능합니다.

전자책은 오탈자 교정이나 내용의 수정·보완이 이뤄지면 업데이트 관련 공지를 이메일로 알려드리며, 구매하신 전자책의 수정본은 무료로 내려받으실 수 있습니다.

이런 특별한 권한은 한빛미디어 사이트에서 구입하신 독자에게만 제공되며, 다른 사람에게 양도나 이전은 허락되지 않습니다.

차례

01 **스코프란 무엇인가** 1

1.1 컴파일러 이론 1
1.2 스코프 이해하기 4
1.3 중첩 스코프 10
1.4 오류 12
1.5 복습하기 14

02 **렉시컬 스코프** 16

2.1 렉스타임 16
2.2 렉시컬 속이기 19
2.3 복습하기 26

03 **함수 vs 블록 스코프** 27

3.1 함수 기반 스코프 27
3.2 일반 스코프에 숨기 28
3.3 스코프 역할을 하는 함수 33
3.4 스코프 역할을 하는 블록 40
3.4 복습하기 50

04 호이스팅 51

4.1 닭이 먼저냐 달걀이 먼저냐 51
4.2 컴파일러는 두 번 공격한다 52
4.3 함수가 먼저다 55
4.4 복습하기 58

05 스코프 클로저 59

5.1 깨달음 59
5.2 핵심 60
5.3 이제 나는 볼 수 있다 64
5.4 반복문과 클로저 67
5.5 모듈 71
5.6 복습하기 81

부록 A 동적 스코프 82

부록 B 폴리필링 블록 스코프 85

부록 C 렉시컬 this 91

부록 D 감사의 말 95

1 | 스코프란 무엇인가

프로그래밍 언어의 기본 패러다임 중 하나는 변수에 값을 저장하고, 저장된 값을 가져다 쓰고 수정하는 기능이다. 이 기능은 프로그램에서 상태를 나타낼 수 있게 해준다. 이와 같은 개념이 없다면 프로그램은 상당히 제한적이고 지극히 심심한 작업만 할 수 있을 것이다. 그러나 변수를 프로그램에 추가하면 다음과 같은 재미있는 질문이 생긴다.

- 변수는 어디에 살아있는가? 다른 말로 하면, 변수는 어디에 저장되는가?
- 필요할 때 프로그램은 어떻게 변수를 찾는가?

이 질문을 통해 알 수 있는 것은 특정 장소에 변수를 저장하고 나중에 그 변수를 찾는 데는 잘 정의된 규칙이 필요하다는 점이다. 바로 이런 규칙을 '스코프Scope'라 한다.

그렇다면 스코프 규칙은 어디서 어떻게 정의될까?

1.1 컴파일러 이론

여러 언어를 다루어 봤다면 자명할 수도 있겠고 아니라면 놀라울 수도 있겠지만, 자바스크립트는 일반적으로 '동적' 또는 '인터프리터' 언어로 분류하지만, 사실은 '컴파일러 언어'다. 물론 자바스크립트가 전통적인 많은 컴파일러 언어처럼 코드를 미리 컴파일하거나 컴파일한 결과를 분산 시스템에서 이용할 수 있는 것은 아니다. 하지만 자바스크립트 엔진은 전통적인 컴파일러 언어에서 컴파일러가 하는 일의 상당 부분을 우리가 아는 것보다 세련된 방식으로 처리한다.

전통적인 컴파일러 언어의 처리 과정에서는 프로그램을 이루는 소스 코드가 실행되기 전에 보통 3단계를 거치는데, 이를 '컴파일레이션compilation'이라고 한다.

토크나이징Tokenizing/렉싱Lexing

문자열을 나누어 '토큰token'이라 불리는 (해당 언어에) 의미 있는 조각으로 만드는 과정이다. 예를 들어, "var a =2;"라는 프로그램을 보자. 이 프로그램은 다음의 토큰으로 나눌 수 있다.

- var,
- a
- =
- 2
- ;
- 빈칸은 하나의 토큰으로 남을 수도 있고 아닐 수도 있다. 이는 빈칸이 의미가 있느냐 없느냐에 달렸다.

> **NOTE_**
>
> 토크나이징과 렉싱은 미묘하고 학술적인 차이가 있는데, 토큰을 인식할 때 무상태 방식으로 하는지 상태유지 방식으로 하는지에 따라 구분한다. 쉽게 말해, '토크나이저tokenizer'가 상태유지 파싱 규칙을 적용해 a가 별개의 토큰인지 다른 토큰의 일부인지를 파악한다면 렉싱이다.

파싱Parsing

토큰 배열을 프로그램의 문법 구조를 반영하여 중첩 원소를 갖는 트리 형태로 바꾸는 과정이다. 파싱의 결과로 만들어진 트리를 AST(추상구문트리abstract syntax tree)라 부른다.

"var a =2;"의 트리는 먼저 변수선언VariableDeclaration이라 부르는 최상위 노드에서 시작하고, 최상위 노드는 'a'의 값을 가지는 확인자Identifier와 대입수식AssignmentExpression

이라 부르는 자식 노드를 가진다. 대입수식 노드는 '2'라는 값을 가지는 숫자리터럴 NumericLiteral을 자식 노드로 가진다.

코드 생성Code-Generation

AST를 컴퓨터에서 실행 코드로 바꾸는 과정이다. 이 부분은 언어에 따라 또는 목표하는 플랫폼에 따라 크게 달라진다. 코드 생성에 대한 세부사항을 보며 끙끙대기보다는 일단 앞서 말한 "var a = 2;"를 나타내는 AST를 기계어 집합으로 바꾸어 실제로 'a'라는 변수를 생성(메모리를 확보하는 일 등)하고 값을 저장할 방법이 있다고 치자.

> **NOTE_**
>
> 엔진이 시스템 리소스를 실제 어떻게 관리하는지에 관한 세부사항은 살펴볼 범위를 넘어서므로 엔진이 필요한 변수를 생성하고 저장할 것이라고 가정하고 넘어가겠다.

자바스크립트 엔진은 이 세 가지 단계뿐만 아니라 많은 부분에서 다른 프로그래밍 언어의 컴파일러보다 훨씬 복잡하다. 예컨대, 자바스크립트 엔진는 파싱과 코드 생성 과정에서 불필요한 요소를 삭제하는 과정을 거쳐 실행 시 성능을 최적화한다.

여기서는 개괄적인 설명만 한다. 그러나 계속 읽다 보면 여기서 왜 대략적이나마 앞의 부분을 다루었는지 알게 될 것이다.

자바스크립트 엔진이 기존 컴파일러와 다른 한 가지 이유는 우선 다른 언어와 다르게 자바스크립트 컴파일레이션이 미리 수행되지 않아서 최적화할 시간이 많지 않기 때문이다.

자바스크립트 컴파일레이션은 보통 코드가 실행되기 겨우 수백만 분의 일초 전에 수행한다. 가능한 한 가장 빠른 성능을 내기 위해 자바스크립트 엔진은 이 책에서

다룰 범위를 가볍게 넘어서는 여러 종류의 트릭(레이지 컴파일lazy compile이나 핫 리컴파일hot recompile 같은 JITs)을 사용한다.

간단히 말하자면, 어떤 자바스크립트 조각이라도 실행되려면 먼저(보통 바로 직전에!) 컴파일되어야 한다는 것이다. 즉, 자바스크립트 컴파일러는 프로그램 "var a = 2;"를 받아 컴파일하여 바로 실행할 수 있게 한다.

1.2 스코프 이해하기

스코드를 좀 더 재미있고 쉽게 설명하기 위해서 대화 형식으로 스코프를 살펴보겠다. 먼저 등장인물을 살펴보자.

1.2.1 출연진

프로그램 "var a =2;"를 처리할 주역들을 만나보자, 그래야 나중에 들을 대화를 이해할 수 있다.

- 엔진: 컴파일레이션의 시작부터 끝까지 전 과정과 자바스크립트 프로그램 실행을 책임진다.
- 컴파일러: 엔진의 친구로, 파싱과 코드 생성의 모든 잡일을 도맡아 한다.
- 스코프: 엔진의 또 다른 친구로, 선언된 모든 확인자(변수) 검색 목록을 작성하고 유지한다. 또한, 엄격한 규칙을 강제하여 현재 실행 코드에서 확인자의 적용 방식을 정한다.

만약 자바스크립트가 어떻게 작동하는지 완전히 이해한다면 엔진(그리고 그 친구들)처럼 생각해보자. 그들이 던지는 질문을 던지고 그들과 똑같이 답해보라.

1.2.2. 앞과 뒤

프로그램 "var a = 2;"를 보면 하나의 구문으로 보인다. 그러나 우리의 새로운 친구 엔진은 그렇게 보지 않는다. 사실 엔진은 두 개의 서로 다른 구문으로 본다. 하나는 컴파일러가 컴파일레이션 과정에서 처리할 구문이고, 다른 하나는 실행 과정에서 엔진이 처리할 구문이다.

그럼 이제 엔진과 친구들이 프로그램 "var a =2;"에 어떻게 접근하는지 낱낱이 살펴보자.

이 프로그램에서 컴파일러가 할 첫 번째 일은 렉싱을 통해 구문을 토큰으로 쪼개는 것이다. 그 후 토큰을 파싱해 트리 구조로 만든다. 그러나 코드 생성 과정에 들어가면 컴파일러는 몇몇 독자의 추측과는 다르게 프로그램을 처리한다.

컴파일러가 다음의 의사 코드pseudo-code로 요약될 수 있는 코드를 생성한다고 생각할 수 있다.

변수를 위해 메모리를 할당하고 할당된 메모리를 a라 명명한 후 그 변수에 값 2를 넣는다.

안타깝지만, 이는 그리 정확한 설명이 아니다. 컴파일러는 다음 일을 진행한다.

① 컴파일러가 'var a'를 만나면 스코프에 변수 a가 특정한 스코프 컬렉션 안에 있는지 묻는다. 변수 a가 이미 있다면 컴파일러는 선언을 무시하고 지나가고, 그렇지 않으면 컴파일러는 새로운 변수 a를 스코프 컬렉션 내에 선언하라고 요청한다.

② 그 후 컴파일러는 'a = 2' 대입문을 처리하기 위해 나중에 엔진이 실행할 수 있는 코드를 생성한다. 엔진이 실행하는 코드는 먼저 스코프에 a라 부르는 변수가 현재 스코프 컬렉션 내에서 접근할 수 있는지 확인한다. 가능하다면 엔진은 변수 a를 사용하고, 아니라면 엔진은 다른 곳(중첩 스코프 부분을 보라)을 살핀다.

엔진이 마침내 변수를 찾으면 변수에 값 2를 넣고, 못 찾는다면 엔진은 손을 들고 에러가 발생했다고 소리칠 것이다!

요약하면, 별개의 두 가지 동작을 취하여 변수 대입문을 처리한다. 첫째, 컴파일러가 변수를 선언한다(현재 스코프에 미리 변수가 선언되지 않은 경우). 둘째, 엔진이 스코프에서 변수를 찾고, 변수가 있다면 값을 대입한다.

1.2.3 컴파일러체[01]

더 나아가기 전에 컴파일러 관련 용어를 약간 더 살펴보자.

2단계에서 컴파일러가 생성한 코드를 실행할 때 엔진은 변수 a가 선언된 적 있는지 스코프에서 검색한다. 이때 엔진이 어떤 종류의 검색을 하느냐에 따라 검색 결과가 달라진다. 앞의 경우에서 엔진은 변수 a를 찾기 위해 LHS 검색을 수행한다. 다른 종류의 검색은 RHS라 부른다. 여기서 'L'과 'R'이 무엇을 뜻하는지 예상할 수 있을 것이다. L과 R은 각각 '왼쪽 방향Left-hand Side'과 '오른쪽 방향Right-hand Side'을 뜻한다.

방향? 어떤 방향? 바로 대입 연산의 방향을 말한다.

다른 말로 하면 LHS 검색은 변수가 대입 연산자의 왼쪽에 있을 때 수행하고, RHS 검색은 변수가 대입 연산자의 오른쪽에 있을 때 수행한다.

좀 더 엄밀하게 살펴보자. RHS 검색은 단순히 특정 변수의 값을 찾는 것과 다를 바 없다. 반면, LHS 검색은 값을 넣어야 하므로 변수 컨테이너 자체를 찾는다. 따라서 정확히 말하면 RHS는 그 자체로는 '대입문의 오른쪽'이 아니다. 좀 더 정확히 말하면 RHS는 '왼편이 아닌 쪽'에 가깝다.

---

01 컴파일러체(Compiler Speak): 컴파일러의 속어로, 사투리 정도로 이해하면 된다.

좀 더 쉽게 말하면 RHS를 "Retrieve(가져오라) his/her(그의/그녀의) source(소스)"의 약자라고 보면 RHS는 "가서 값을 가져오라"라는 뜻으로 이해할 수 있다. 이제 좀 더 깊게 파보도록 하자. 다음 구문을 보자.

```
console.log( a );
```

a에 대한 참조는 RHS 참조다. 구문에서 a에 아무 것도 대입하지 않기 때문이다. 대신 a의 값을 가져와 console.log(…)에 넘겨준다. 다른 예제를 보자.

```
a = 2;
```

a에 대한 참조는 LHS 참조다. 현재 a 값을 신경 쓸 필요 없이 '= 2' 대입 연산을 수행할 대상 변수를 찾기 때문이다.

NOTE_

LHS와 RHS가 '대입문의 왼쪽/오른쪽'을 뜻한다고 해서 반드시 문자 그대로 '대입 연산자(=)의 왼쪽과 오른쪽'을 지칭하는 것은 아니다. 대입 연산은 다른 여러 방식으로 일어날 수 있다. 따라서 개념적으로는 다음과 같이 생각하는 것이 더 낫다.

- 대입할 대상(LHS)과 대입한 값(RHS)

LHS와 RHS 참조를 모두 수행하는 다음 프로그램을 보자.

```
function foo(a) {
    console.log( a ); // 2
}
foo( 2 );
```

마지막 줄에서 foo(…) 함수를 호출하는 데 RHS 참조를 사용한다. 즉 "가서 foo의 값을 찾아 내게 가져와라"라는 뜻이다. 여기서 (…)는 실행된다는 뜻이므로 foo는 함수여야 한다.

이 부분에 미묘하지만 중요한 대입이 수행된다. 무엇을 가리키는지 알겠는가?

앞의 코드 속에 내재된 'a = 2'를 놓쳤을지도 모르겠다. 인수로 값 2를 함수 foo(…)에 넘겨줄 때 값 2를 매개변수 a에 대입하는 연산이 일어난다. 이 (내재된) 매개변수 a에 대한 대입 연산을 위해 LHS 검색이 수행된다.

변수 a에 대한 RHS 참조 역시 수행되는데, 그 결과값은 console.log(…) 함수에 넘겨진다. 또 console.log(…)가 실행되려면 참조가 필요하다. console 객체를 RHS 검색하여 log 메소드가 있는지 확인한다.

마지막으로 값 2를 RHS로 불러온 변수 a를 통해 log(…)에 넘겨주는 과정에서 LHS/RHS를 주고받는 작업에 대한 개념을 짚어 보자. 구현된 log(…)의 내부에는 매개변수가 있을 것이고, 첫 번째 매개변수(어쩌면 arg1라 부를 것)를 LHS 검색으로 찾아 2를 대입할 것이다.

NOTE_

어쩌면 함수선언문 "function foo(a) { …"를 "var foo"와 "foo = function(a) { …" 같은 일반적인 변수 선언 및 대입과 같다고 생각할지도 모르겠다. 그렇게 생각한다면 이 함수 선언이 LHS 검색을 사용할 것이라 단정할 수도 있겠다.

그러나 여기에는 변수 처리 과정과 다른 미묘하지만 중요한 차이점이 있다. 컴파일러는 앞의 선언문과 값 정의문을 코드 생성 과정에서 처리하여 엔진이 코드를 실행할 때는 foo에 함수값을 대입하는 과정이 필요 없다. 따라서 함수 선언을 앞에서 살펴본 변수의 경우와 같이 LHS 검색 및 대입 과정이라고 생각하는 것은 적절하지 않다.

1.2.4 엔진과 스코프의 대화

```
function foo(a) {
    console.log( a ); // 2
}

foo( 2 );
```

이 코드의 실행 과정을 대화라고 상상해보자. 그 대화는 이럴 것이다.

엔진 : 안녕, 스코프. foo에 대한 RHS 참조가 필요해. foo라고 들어본 적 있니?

스코프: 응, 들어봤어. 컴파일러가 좀 전에 선언하더라고. foo는 함수야. 이걸 보면 돼.

엔진 : 좋아, 고마워! 자, 이제 나는 foo를 실행해야겠어.

엔진 : 이봐, 스코프. a에 대한 LHS 참조도 구해야 하는데, 들어본 적 있어?

스코프: 물론이지. 컴파일러가 a를 foo의 매개변수로 좀 전에 선언했어. 이걸 봐.

엔진 : 항상 도와줘서 고마워, 스코프. 정말 고마워. 이제 2를 a에 대입할 시간이야.

엔진 : 스코프, 자꾸 귀찮게 해서 미안해. console에 대한 RHS 검색이 필요해. 해줄 수 있겠니?

스코프: 문제 없어, 엔진. 이게 내가 항상 하는 일이잖니. 자, console을 찾았어. 내장돼 있더라. 이거야.

엔진 : 완벽해. 이제 log(…)를 찾아볼까. 좋아 멋져, 이건 함수구나.

엔진 : 요~ 스코프! a의 RHS 참조 찾는 것 좀 도와줄 수 있을까? 나한테도 있긴 할 테지만, 확실히 해두고 싶어.

스코프: 옳은 말이야, 엔진. 변함 없이 엄밀하구나. 여기 있어.

엔진 : 멋져. 이제 a의 값을…. 값은 2구나. log(…)에 넘기자.

…

1.2.5 퀴즈

지금까지 배운 것을 확인해보자. 엔진의 역할을 맡아서 스코프와 대화해보자.

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

① 모든 LHS 검색을 찾아보라(모두 3개다!).

② 모든 RHS 검색을 찾아보라(모두 4개다!).

> **NOTE_**
>
> 퀴즈의 정답은 이번 장 복습하기에서 볼 수 있다.

1.3 중첩 스코프

스코프는 확인자 이름으로 변수를 찾기 위한 규칙의 집합이라고 앞서 말한 바 있다. 그러나 대개 고려해야 할 스코프는 여러 개다.

하나의 블록이나 함수는 다른 블록이나 함수 안에 중첩될 수 있으므로 스코프도 다른 스코프 안에 중첩될 수 있다. 따라서 대상 변수를 현재 스코프에서 발견하지 못하면 엔진은 다음 바깥의 스코프로 넘어가는 식으로 변수를 찾거나 글로벌 스코프라 부르는 가장 바깥 스코프에 도달할 때까지 계속한다.

```
function foo(a) {
    console.log( a + b );
}

var b = 2;

foo( 2 ); // 4
```

b에 대한 RHS 참조는 함수 foo 안에서 처리할 수 없고, 함수를 포함하는 스코프(이 경우에는 글로벌 스코프)에서 처리한다.

엔진과 스코프의 대화를 다시 보자.

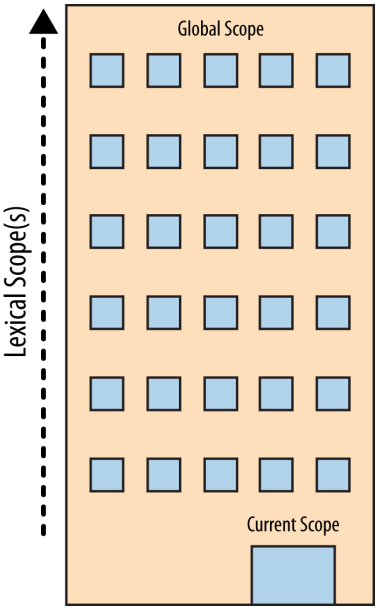
> **엔진** : 이봐 foo의 스코프, b에 대해 들어본 적 있나? b에 대한 RHS 참조가 필요한데 말야.
>
> **스코프**: 아니, 못 들어봤어. 딴 데 가봐.
>
> **엔진** : 이봐, foo의 바깥 스코프! 아, 니가 글로벌 스코프구나, 멋지군. 혹시 b에 대해 들어봤니? b에 대한 RHS 참조가 필요해.
>
> **스코프**: 응, 물론 들어봤지. 여기 있어.

중첩 스코프를 탐사할 때 사용하는 간단한 규칙은 다음과 같다.

- 엔진은 현재 스코프에서 변수를 찾기 시작하고, 찾지 못하면 한 단계씩 올라간다.
- 최상위 글로벌 스코프에 도달하면 변수를 찾았든 못 찾았든 검색을 멈춘다.

1.3.1 비유로 배워보기

중첩 스코프 검색 과정을 다음과 같은 큰 빌딩으로 상상해보자.



이 빌딩은 프로그램의 중첩 스코프 규칙 집합을 나타낸다. 어디에 있든 1층은 현재 실행 중인 스코프를 뜻한다. LHS/RHS를 참조하려면 현재 층을 둘러보고, 찾지 못하면 엘리베이터를 타고 다음 층으로 가서 찾고, 또 다음 층으로 이동하는 식이다. 최상위 층(글로벌 스코프)에 도달했을 때 찾던 것을 발견했을 수도 있고 아닐 수도 있다. 그러나 어쨌든 검색은 거기서 중단한다.

1.4 오류

LHS와 RHS를 구분하는 것이 왜 중요할까? 이 두 종류의 검색 방식은 변수가 아직 선언되지 않았을 때(검색한 모든 스코프에서 찾지 못했을 때) 서로 다르게 동작하기 때문이다.

```
function foo(a) {
    console.log(a + b);
    b = a;
}

foo( 2 );
```

b에 대한 첫 RHS 검색이 실패하면 다시는 b를 찾을 수 없다. 이렇게 스코프에서 찾지 못한 변수는 '선언되지 않은 변수'라 한다. RHS 검색이 중첩 스코프 안 어디에서도 변수를 찾지 못하면 엔진이 'ReferenceError'를 발생시킨다. 여기서 중요한 점은 발생된 오류가 ReferenceError 타입이라는 것이다.

반면에, 엔진이 LHS 검색을 수행하여 변수를 찾지 못하고 꼭대기 층(글로벌 스코프)에 도착할 때 프로그램이 'Strict Mode'[02]로 동작하고 있는 것이 아니라면, 글로벌 스코프는 엔진이 검색하는 이름을 가진 새로운 변수를 생성해서 엔진에게 넘겨준다. 즉, "없어, 없었지만 내가 널 위해 하나 만들어주지"라고 생각하면 된다.

ES5부터 지원하는 'Strict Mode'는 normal/relaxed/lazy mode와는 여러 면에서 다르게 작동한다. 예를 들어, strict mode에서는 글로벌 변수를 자동으로 또는 암시적으로 생성할 수 없다. 그래서 앞의 상황이 닥치면 글로벌 스코프는 변수를

02 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

생성하지 않아서 LHS 검색은 아무 것도 얻지 못하고, 엔진은 RHS의 경우와 비슷하게 ReferenceError를 발생시킨다.

이제, RHS 검색 결과 변수를 찾았지만 그 값을 가지고 불가능한 일을 하려고 할 경우를 보자. 예를 들어, 함수가 아닌 값을 함수처럼 실행하거나 null이나 undefined 값을 참조할 때 엔진은 TypeError를 발생시킨다.

ReferenceError는 스코프에서 대상을 찾았는지와 관계 있지만, TypeError는 스코프 검색은 성공했으나 결과값을 가지고 적법하지 않거나 불가능한 시도를 한 경우를 의미한다.

1.5 복습하기

스코프는 어디서 어떻게 변수(확인자)를 찾는가를 결정하는 규칙의 집합이다. 변수를 검색하는 이유는 변수에 값을 대입하거나(LHS 참조) 변수의 값을 얻어오기 위해서다(RHS 참조).

LHS 참조는 대입 연산 과정에서 일어난다. 스코프와 관련된 대입 연산은 '=' 연산자가 사용되거나 인자를 함수의 매개변수로 넘겨줄 때 일어난다.

자바스크립트 엔진은 코드를 실행하기 전에 먼저 컴파일하는데, 이 과정에서 엔진은 "var a = 2;"와 같은 구문을 독립된 두 단계로 나눈다.

① var a은 변수 a를 해당 스코프에 선언한다. 이 단계는 코드 실행 전에 처음부터 수행된다.
② a = 2는 변수 a를 찾아 값을 대입한다(LHS 참조).

LHS와 RHS 참조 검색은 모두 현재 실행 중인 스코프에서 시작한다. 그리고 필요하다면(대상 변수를 찾지 못했을 경우) 한 번에 한 스코프씩 중첩 스코프의 상위 스코프

로 넘어가며 확인자를 찾는다. 이 작업은 글로벌 스코프(꼭대기 층)에 이를 때까지 계속하고, 대상을 찾았든 못 찾았든 작업을 중단한다.

RHS 참조가 대상을 찾지 못하면 ReferenceError가 발생한다. LHS 참조가 대상을 찾지 못하면 자동적, 암시적으로 글로벌 스코프에 같은 이름의 새로운 변수가 생성된다(만약 'Strict Mode'일 경우 ReferenceError가 발생함).

1.5.1. 퀴즈 답안

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

① 모든 LHS 검색을 찾아보라(모두 3개다!).

c = …;, a = 2(암시적 매개변수 대입), b = …

② 모든 RHS 검색을 찾아보라(모두 4개다!).

foo(2 …, = a;, a …, … b

2 | 렉시컬 스코프

1장에서 '스코프'를 엔진이 확인자 이름으로 현재 스코프 또는 중첩 스코프 내에서 변수를 찾을 때 사용하는 '규칙의 집합'이라고 정의했다.

스코프는 두 가지 방식으로 작동한다. 첫 번째 방식은 다른 방식보다 훨씬 더 일반적이고 다수의 프로그래밍 언어가 사용하는 방식이다. 이 방식을 '렉시컬 스코프Lexical Scope'라고 부른다. 이 장에서는 렉시컬 스코프에 관해 면밀히 검토하겠다. 두 번째 방식은 Bash Scripting이나 Perl의 일부 모드와 같은 몇몇 언어에서 사용하는 방식으로 '동적 스코프Dynamic Scope'라고 부른다.

동적 스코프에 대해서는 부록 A에서 다루고, 이번 장에서는 오직 자바스크립트에서 채용한 렉시컬 스코프와 대비하기 위해 잠깐 언급한다. 동적 스코프를 알고 싶은 독자는 부록 A를 참고하길 바란다.

2.1 렉스타임

1장에서 다룬 것처럼 일반적인 언어의 컴파일러는 첫 단계를 전통적으로 토크나이징 또는 렉싱이라 불리는 작업으로 시작한다. 렉싱 처리 과정에서는 소스 코드 문자열을 분석하여 상태유지 파싱의 결과로 생성된 토큰에 의미를 부여한다. 바로 이 개념이 렉시컬 스코프가 무엇인지, 어원이 어디인지를 알 게 해주는 바탕이 된다.

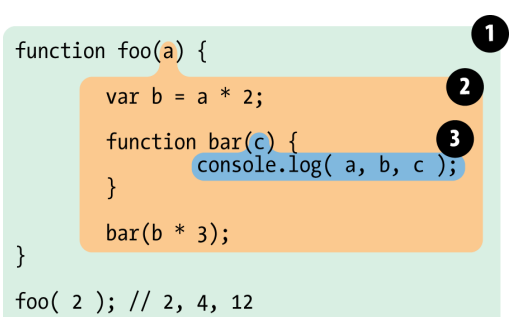
약간 순환적인 정의하면 렉시컬 스코프는 렉싱 타임lexing time에 정의되는 스코프다. 바꿔 말해, 렉시컬 스코프는 프로그래머가 코드를 짤 때 변수와 스코프 블록을 어디서 작성하는가에 기초해서 렉서lexer가 코드를 처리할 때 확정된다.

NOTE_

뒤에서 렉시컬 스코프를 속여 렉서가 지나간 이후 수정하는 방법에 대해 간단히 알아볼 것이다. 그러나 이는 권장하지 않는 방법이다. 렉시컬 스코프를 수정하는 가장 좋은 방법은 오직 구문과 관련해서 변경하는 것, 즉 사실상 코드 작성 때만 수정하는 것이다.

```
function foo(a) {
    var b = a * 2;
    function bar(c) {
        console.log( a, b, c );
    }
    bar( b * 3 );
}
foo( 2 ); // 2, 4, 12
```

이 예제 코드에는 3개의 중첩 스코프가 있다. 스코프를 다음과 같이 겹쳐진 버블이라고 가정하면 이해하기 쉽다.



버블 ❶은 글로벌 스코프를 감싸고 있고, 그 스코프 안에는 오직 하나의 확인자(foo)만 있다.
버블 ❷는 foo의 스코프를 감싸고 있고, 그 스코프는 3개의 확인자(a, bar, b)를 포함한다.
버블 ❸은 bar의 스코프를 감싸고 있고, 그 스코프는 하나의 확인자(c)만을 포함한다.

스코프 버블은 스코프 블록이 쓰이는 곳에 따라 결정되는데, 스코프 블록은 서로 중첩될 수 있다. 다음 장에서 다르게 구성되는 스코프도 다루겠지만, 지금은 각각의 함수가 새로운 스코프 버블을 생성한다고 가정하자.

bar의 버블은 foo의 버블 내부에 완전히 포함된다. 바로 foo의 내부에서 bar 함수를 정의했기 때문이다.

그림에서 중첩 버블의 경계가 엄밀하게 정해져 있는 것이 보이는가? 보고 있는 버블은 서로의 경계가 교차할 수 있는 벤다이어그램이 아니다. 다시 말해, 어떤 함수의 버블도 동시에 (일부라도) 다른 두 스코프 버블 안에 존재할 수 없다. 어떤 함수도 두 개의 부모 함수 안에 존재할 수 없는 것처럼 말이다.

2.1.1 검색

엔진은 스코프 버블의 구조와 상대적 위치를 통해 어디를 검색해야 확인자를 찾을 수 있는지 안다.

앞의 코드를 보면 엔진은 console.log(…) 구문을 실행하고 3개의 참조된 변수 a, b, c를 검색한다. 검색은 가장 안쪽의 스코프 버블인 bar(…) 함수의 스코프에서 시작한다. 여기서 a를 찾지 못하면 다음으로 가장 가까운 스코프 버블인 foo(…)의 스코프로 한 단계 올라가고, 이곳에서 a를 찾아 사용한다. 똑같은 방식이 b에도 적용된다. 단, c는 bar(…) 내부에서 찾을 수 있다.

변수 c가 bar(…)와 foo(…) 내부에 모두 존재한다고 가정하면, console.log(…) 구문은 bar(…) 내부에 있는 c를 찾아서 사용하고 foo(…)에는 c를 찾으러 가지도 않는다.

스코프가 목표와 일치하는 대상을 찾는 즉시 검색을 중단한다. 여러 중첩 스코프 층에 걸쳐 같은 확인자 이름을 정의할 수 있다. 이를 '섀도우잉shadowing'이라 한다

(더 안쪽의 확인자가 더 바깥쪽의 확인자를 가리는 것이다). 섀도우잉과 상관없이 스코프 검색은 항상 실행 시점에서 가장 안쪽 스코프에서 시작하여 최초 목표와 일치하는 대상을 찾으면 멈추고, 그 전까지는 바깥/위로 올라가면서 수행한다.

NOTE_

글로벌 변수는 자동으로 웹 브라우저의 window 같은 글로벌 객체에 속한다. 따라서 글로벌 변수를 직접 렉시컬 이름으로 참조하는 것뿐만 아니라 글로벌 객체의 속성을 참조해 간접적으로 참조할 수도 있다.

```
window.a
```

가려져 있어서 다른 방식으로는 접근할 수 없는 글로벌 변수에는 이 방법을 통해 접근할 수 있다. 그러나 글로벌이 아닌 섀도우 변수는 접근할 수 없다.

어떤 함수가 어디서 또는 어떻게 호출되는지에 상관없이 함수의 렉시컬 스코프는 함수가 선언된 위치에 따라 정의된다.

렉시컬 스코프 검색 과정은 a, b, c 같은 일차 확인자 검색에만 적용된다. 코드에서 foo.bar.baz의 참조를 찾는다고 하면 렉시컬 스코프 검색은 foo 확인자를 찾는 데 사용되지만, 일단 foo를 찾고 나서는 객체 속성 접근 규칙을 통해서 bar와 baz의 속성을 각각 가져온다.

2.2 렉시컬 속이기

렉시컬 스코프는 프로그래머가 작성할 때 함수를 어디에 선언했는지에 따라 결정된다. 그렇다면 런타임 때 어떻게 렉시컬 스코프를 수정할 (또는 속일) 수 있을까?

자바스크립트에서는 렉시컬 스코프를 속일 수 있는 두 가지 방법이 있다. 두 방법

모두 개발자 커뮤니티에서는 코드 작성할 때 권장하지 않는 방법이다. 그러나 많은 사림이 이런 방법을 비판하지만, 가장 중요한 논점을 빠트린다. 바로 렉시컬 스코프를 속이는 방법은 성능을 떨어뜨린다는 점이다.

성능 문제를 설명하기 전에 앞서 말한 두 가지 방법이 어떻게 동작하는지 살펴보자.

2.2.1 eval

자바스크립트의 eval(…) 함수는 문자열을 인자로 받아들여 실행 시점에 문자열의 내용을 코드의 일부분처럼 처리한다. 즉, 처음 작성한 코드에 프로그램에서 생성한 코드를 집어넣어 마치 처음 작성될 때부터 있던 것처럼 실행한다.

eval(…)의 성격을 생각해보면 eval(…)를 통해 어떻게 렉시컬 스코프를 수정하고 원래 작성했던 코드인양 속일 수 있는지 이해할 수 있다. eval(…)이 실행된 후 코드를 처리할 때 엔진은 지난 코드가 동적으로 해석되어 렉시컬 스코프를 변경시켰지 알 수도 없고 관심도 없다. 엔진은 그저 평소처럼 렉시컬 스코프를 검색할 뿐이다.

```
function foo(str, a) {
    eval( str ); // cheating!
    console.log( a, b );
}
var b = 2;
foo( "var b = 3;", 1 ); // 1, 3
```

문자열 “var b = 3;”은 eval(…)이 호출되는 시점에 원래 있던 코드인 것처럼 처리된다. 이 코드는 새로운 변수 b를 선언하면서 이미 존재하는 foo(…)의 렉시컬 스코프를 수정한다. 사실, 앞에서 언급한 것처럼 이 코드는 실제로 foo(…) 안에 변수 b를 생성하여 바깥(글로벌) 스코프에 선언된 변수 b를 가린다.