TR#95-009

Scheme: Past, Present and Future

Technical Report No. 95-009

Guy L. Steele Jr.*
Sun Microsystems Laboratories
and
Masayuki Ida†
Computer Science Research Laboratory
Information Science Research Center

Aoyama Gakuin University
Address: 4-4-25 Shibuya, Shibuya-ku, Tokyo Japan 150

Dec. 20, 1995

*gls@East.Sun.Com
†ida@csrl.aoyama.ac.jp

Preface

This report is the transcription of the talk by Dr. Guy L. Steele Jr. at the LS-JP symposium on July 10th, 1995. The work was arranged by Masayuki Ida and the transcription was done by himself mostly. The LS-JP board asked Masayuki Ida to publish the record as a CSRL technical report, and he have made it for the LS-JP.

We like to express our gratitude to the LS-JP board members who have volunteered with us to make this event successful. Especialy, we like to note the works by Mamoru Sato, Takumi Doi, Takashi Kosaka were outstanding.

Masayuki Ida
December, 20th, 1995

— No Migration but Co-existence —

Contents

Chapter 1

Introduction

1.1 The LS-JP July meeting

The Lisp Society Japan (chairman: Masayuki Ida) have annual event series for the members. In '95 we started the series by having talks around Lisp. As the first event, the LS-JP was fortunate to have a talk by Guy L. Steele Jr while he was visiting Japan to be a guest speaker at SPARC UNIX conference at Makuhari. Since Guy L Steele is also famous as a co-designer of Scheme, the LS-JP asked him to talk about the origin, basic concepts behind Scheme design and his view on its history. He titled the talk as "Scheme: Past, Present, and Future". The talk was held at Aoyama Gakuin, Aogaku-Kaikan on July 10th, 1995. Th summer in Japan of the year was exceptionally hot, while the room was well air conditioned.

Chapter 2 is the transciption of his talk. He had used 26 slides on the subject. Opening messages by Guy Steele is included in this section.

1.2 Introduction of the speaker

Before starting the talk, the coordinator introduced the bibliography of Guy Steele Jr. using the full resume of him typed in more than 20 sheets. Here is a very short version of it.

Guy L. Steele Jr. is a Distinguished Engineer at Sun Microsystems, Inc. He received his A.B. in applied mathematics from Harvard College (1975), and his S.M. and Ph.D. in computer science and artificial intelligence from M.I.T. (1977 and 1980). He has also been an assistant professor of computer science at Carnegie-Mellon University; a member of technical staff at Tartan Laboratories in Pittsburgh, Pennsylvania; and a senior scientist at Thinking Machines Corporation. He joined Sun Microsystems in 1994.

He is author or co-author of four books: Common Lisp: The Language (Digital Press); C: A Reference Manual (Prentice-Hall); The Hacker's Dictionary (Harper&Row), which has been revised as The New Hacker's Dictionary, edited by Eric Raymond with introduction and illustrations by Guy Steele (MIT Press); and The High Performance Fortran Handbook (MIT Press).

He has published more than two dozen papers on the subject of the Lisp language and Lisp implementation, including a series with Gerald Jay Sussman that defined the Scheme dialect of Lisp. One of these, "Multiprocessing Compactifying Garbage Collection," won first place in the ACM 1975 George E. Forsythe Student Paper Competition. Other papers published in CACM are "Design of a LISP-Based Microprocessor" with Gerald Jay Sussman (November 1980) and "Data Parallel Algorithms" with W. Daniel Hillis (December 1986). He has also published papers on other subjects, including compilers, parallel processing, and constraint languages. One song he composed has been published in CACM ("The Telnet Song", April 1984).

The Association for Computing Machinery awarded him the 1988 Grace Murray Hopper Award and named him an ACM Fellow in 1994. He was elected a Fellow of the American Association for Artificial Intelligence in 1990. He led the team that received a 1990 Gordon Bell Prize honorable mention for achieving the fastest speed to that date for a production application: 14.182 Gigaflops.

He has served on accredited standards committees X3J11 (C language) and X3J3 (Fortran) and is currently chairman of X3J13 (Common Lisp). He was also a member of the IEEE committee that produced the IEEE Standard for the Scheme Programming Language, IEEE Std 1178-1990. He represents Sun Microsystems in the High Performance Fortran Forum, which produced the High Performance Fortran specification in May, 1993.

He has served on Ph.D. thesis committees for seven students. He has served as program chair for the 1984 ACM Lisp Conference and for the 15th ACM POPL conference (1988); he also served on program committees for 30 other conferences. He served a five-year term on the ACM Turing Award committee, chairing it in 1990. He served a five-year term on the ACM Grace Murray Hopper Award committee, chairing it in 1992.

He has had chess problems published in Chess Life and Review and is a Life Member of the United States Chess Federation. He has sung in the bass section of the MIT Choral Society (John Oliver, conductor) and the Masterworks Chorale (Allen Lannom, conductor) as well as in choruses with the Pittsburgh Symphony Orchestra at Great Woods (Michael Tilson Thomas, conductor) and with the Boston Concert Opera (David Stockton, conductor). He has played the role of Lun Tha in The King and I and the title role in Li'l Abner. He designed the original EMACS command set and was the first person to port TEX.

At Sun Microsystems he is responsible for research in parallel algorithms, implementation strategies, and architectural and software support.

1.3 Opening

Scheme
Past, Present, and Future

Guy L. Steele Jr.
Sun Microsystems laboratories

KONNICHIWA. Thank you for inviting me to speak here to the Lisp society of Japan.

Today, I would like to talk to you about the beginnings of the Scheme language and to discuss its history from the very beginning and before the beginning, until today. And, we will discuss little bit about the future. Perhaps some of you know Common Lisp. Common Lisp is a very big language, and a very complicated language, and Scheme is very small. I hope that during my talk today you will come to understand why Scheme is small and, I think, more elegant than Common Lisp. There are some good reasons.

こんにちは、LISP 協議会にお招きいただいてありがとうございます。今日は、Scheme 言語のはじまりについてお話しします。また、そもそもの発端のいきさつから今日にいたるまでの歴史を議論します。今後についてもいくつか述べます。おそらく、Common Lisp のことをご存知な方もいらっしゃるでしょう。Common Lisp は大変大きく複雑な言語です。一方 Scheme は大変小さいです。今日の話を聞いて、なぜ Scheme が小さいのか、なぜ Common Lisp よりエレガントだと私が考えているか、みなさんが理解していただける事を期待します。いくつかの明確な理由があるのです。

Chapter 2

Scheme: Past, Present and Future

2.1 How Did Scheme Begin ? (Slide 1)

How Did Scheme Begin ?

- Carl Hewitt and Gerald Jay Sussman had an argument...
- Artificial Intelligence languages
  - Planner (theorem proving)
  - Muddle (implementation language)
  - Microplanner (small theorem prover)
  - Conniver (hairy control structure)
  - PLASMA (actors model)

First let us discuss how the Scheme began. I wish I could tell you that it was a carefully designed language and it was the result of careful research and understanding of what a programming language should be. Perhaps that is not a true story at all.

The language began because two very brilliant persons at MIT were having a big argument. This started slightly before I came to MIT as a student. This discussion was between Prof. Carl Hewitt, and a new student to MIT, Gerry Sussman. Carl Hewitt was at the artificial intelligence labratory at MIT and was interested in designing programming language for robots. At that time, in the mid-1960s, it seemed to be important that robots be able to

prove theorems about what they ought to do. In particular, it was important that a robot could plan what action to take before actually doing the action. So Carl Hewitt designed a programming language called Planner, that was intended to help programmers or robots to do theorem proving. The idea was that the programmer could write down the rules, and the programming language Planner would automatically deduce the theorem from the rules. So Planner was perhaps one of the first rule-based programming languages.

Gerry Sussman arrived at MIT and became a part of the project to implement Planner. Carl Hewitt never designed simple languages, and Planner was a very complicated language. So they decided first to design a simple programming language called Muddle, and then they used Muddle to implement Planner. Carl Hewitt helped to design Muddle and Gerry Sussman helped to implement Muddle, then tried to implement Planner. This was still too difficult and the full Planner language was never really implemented. However, Muddle was fully implemented and used in other research projects. And some features of Common Lisp, in particular argument lists with `&optional &rest &keyword` and other keywords, came directly from Muddle.

In the meantime, it seemed important to have some programming language for robots, so another project was started for a small-version of Planner, called Microplanner. Instead of being implimented in Muddle, this is implemented in MacLisp, which was one of the predecessors of Common Lisp. This Microplanner was a reasonable success, and several artificial intelligence projects at MIT were programmed in Microplanner. Perhaps the most famous such project was SHRDLU, automatic movement of blocks in a "blocks world," which was done by Terry Winograd.

After some experiences with Microplanner, Gerry Sussman and another student, Drew McDermott, were not satisfied; it was still too hard to write programs. So they decided to invent yet another programming language. They called this programming language Conniver.

The difference between Microplanner and Conniver was that in Microplanner theorem proving was automatic and control structure was automatic. As Microplanner used the rules to prove a theorem, it would try one thing and then another with automatic backtracking. Sometimes the automatic back tracking was very inefficient. In the Conniver programming language, Sussman and McDermott wanted to let the programmer have more control over the backtracking or to do something other than backtracking.

So we see a sort of progression. This argument between Carl Hewitt and Gerry Sussman was really arguing about the best way to write programs for robots and the best way to program for artificial intelligence. But instead of fighting with each other, what they did was to design programming languages.

Each one said, "My language is better than your language!"

Conniver was not the last word. Carl Hewitt designed another language called Plasma. Plasma was better than Conniver. (I should note that there is a famous confarence paper that was written by McDermott and Sussman. The title was "Why Conniving Is Better than Planning.") Plasma was different from the earlier languages because it was based on Carl Hewitt's new idea of an actors model. So, to understand Scheme, it is important first to understand Carl Hewitt's actor model.

まず、どのようにしてSchemeがはじまったのかについてお話します。Schemeは大変注意深く設計された言語であって、プログラミング言語のあり方についての充分な研究と理解に基づいたものであったと言えれば良かったかも知れません。しかし、事実はぜんぜん違います。

この言語は、MITの二人の聡明な人間がしていた、大きな論争にはじまります。それは、私がMITに学生として来るちょっと前のことでした。この論争というのは、Carl Hewitt先生と、新しく学生として入ったGerry Sussmanの間におこったものです。Carl Hewittは、MITの人工知能研究所にいて、ロボットのための言語の設計に興味をもっていました。60年代の半ばでは、ロボットが自分でなすべきことについて定理証明できるかどうかが重要のようでした。特に、ロボットが具体的な行動をする前にその行動のプランをできるかどうかが重要でした。そこでCarl HewittはPlannerというプログラミング言語を設計しました。それは、プログラマやロボットが定理証明をするのを助けるためのものでした。アイデアというのは、プログラマがルールを書き出し、Planner言語が自動的にルールから定理を演繹するというものでした。従って、Plannerはおそらく最初のルールベースのプログラミング言語でした。

Gerry SussmanはMITに来て、Plannerを実装するプロジェクトのメンバーになりました。Carl Hewittは、簡単な言語を設計するということは決してありませんでしたので、Plannerは大変複雑な言語でした。そこで彼らはまず、Muddleと呼ぶシンプルなプログラミング言語の設計からはじめる事にしました。そして、Muddleを使って、Plannerを作成しようとしたのです。Carl HewittはMuddleの設計を受け持ち、Gerry SussmanはMuddleの実装をうけもち、そしてPlannerの実装をしようとしたのです。これもなお、困難な仕事で、完全なPlanner言語はついに実現されませんでした。しかし、Muddleは完全に作られ、他の研究プロジェクトで用いられました。Common Lispの機能のうち、引数リストの、`&optional`、`&rest`、`&keyword`その他のキーワードなどは直接Muddleからきています。

一方、ロボットのためのなんらかのプログラミング言語が必要となり、Planner の小型版、Microplannerをつくる別のプロジェクトがはじまりました。それは、Muddleで実現するかわりに、Common Lispの前身の一つであるMacLispで作られました。このMicroplannerはまずまずの成功をおさめ、MITの人工知能プロジェクトのいくつかはこれで書かれました。おそらく、その中で最も有名なプロジェクトは、Terry Winogradによる「積み木の世界」で積み木を動かすSHRDLUでしょう。

Microplannerを経験したあと、Gerry Sussmanともう一人の学生Drew McDermottはそれに満足しませんでした。プログラムするのが難しかったからです。彼らはさらに別のプログラミング言語を発明することを決心しました。それは、Conniverと呼ばれました。

MicroplannerとConniverの間の違いは、Microplannerでは定理証明も制御構造もオートマティックであったことです。Microplannerは定理証明をするのにルールを使っていましたので、自動バックトラックで次々とテストをしていきました。ときどき、この自動バックトラックは非効率的です。Conniver言語では、SussmanとMcDermottは、プログラマがこのバックトラックをもっと制御できるように、言い替えれば、バックトラック以外の何かもできるようにしようとしました。

そこで、ある種の進歩があったのです。Carl HewittとGerry Sussmanの間の議論は、ロボットを

プログラムする最も良い方法、そして、人工知能をプログラムする最も良い方法に関してであったのです。しかし、互いに争うのではなく、やったことは競って別の言語の設計をしたのです。「私の言語の方が君のよりいいよ。」と互いに言い合ったのです。

Conniver で終りではありませんでした。Carl Hewitt は、Plasma という名の言語を設計しました。Plasma は、Conniver より良かったのです。（ここで、McDermott と Sussman によって書かれた論文があることを指摘しておきます。そのタイトルは、「なぜ Conniver は Planner より良いのか」です。）Plasma は、アクタモデルという Carl Hewitt の新しいアイデアに基づいていたので、以前の言語とは異なっているものでした。だから、Scheme を理解するのには、まず Carl Hewitt のアクタモデルを理解する必要があります。

2.2 What Is an Actor ? (Slide 2)

What Is an Actor ?

- Inspired by Simula-67 and Smalltalk
- An actor is an object
- An actor can send and receive messages
- An actor knows certain other actors
- Try to explain all computation this way

What is an actor? The idea was inspired by the programming languages Simula 67 and smalltalk. Actors are very much like objects. I will explain it to you, but not quite in the same words that Carl Hewitt used.

An Actor is an object. An actor can send and receive messages. Also, an actor knows certain other actors—we would say it has pointers to other actors.

Carl Hewitt tried to explain all kinds of computation in terms of actors. Today we would say that Plasma was a strictly object-oriented language. But at that time the term "object-oriented language" was not common. People were still struggling to understand the language. Carl Hewitt's particular contribution was that he thought the English word "object" means something that sits there and does nothing. "Actor" means something that does something—it is an agent. In fact, I was strongly surprised that the word "agent" was not used instead of "actor"—in the Latin language, "actor" and "agent" are from the same verb.

アクタとは何か？そのアイデアは、SIMULA67 と Smalltalk にヒントを得ています。アクタは、オブジェクトのようなものです。これらの説明をしますが、私の言い方は Carl Hewitt が使った言葉とは全く同じではありません。

アクタはオブジェクトです。アクタはメッセージを送受できます。アクタは他のアクタを知っています。もう少し言えば他のアクタへのポインタを持っています。

Carl Hewitt は、すべての計算をアクタを使って説明しようとしました。今日、Plasma はまったくオブジェクト指向言語であったということができます。しかし、あの時点では、オブジェクト指向言語という言葉はあまり使われていませんでした。それで、この言語を理解するのにもがいていました。Carl Hewitt の大きな貢献は、彼は「オブジェクト」という言葉がそれ自身何もしないが、そこにあるものとし、「アクタ」は何かをする何かとして見たことです。それは、エージェントです。事実、私はアクタではなくエージェントという言葉が使われなかったことに大変驚きました。ラテン語では、アクタとエージェントは同一の動詞から来ています。

2.3 Actor Example (Slide 3)

Actor Example

- "1 is an actor. If you send it the message '+ 2 c' then eventually the actor 'c' is sent the message '3'."

So, let's look at the examples of Actor computation. To Carl Hewitt, a number, such as 1 or 2, is an actor.

In an odrinary programming language, you might say we give 1 and 2 to the addition function; the addition function returns the number 3. But Hewitt suggested an different explanation as well. The number 1 is an actor. If you send a message to number 1, it takes a continuation c. In particular, if you send, to the number 1, a message `(plus 2 c)`, then eventually the actor c gets 3 as a message. At that time, about in 1974, it was a very strange kind of computation style. Gerry Sussman and I—I had arrived at MIT as a student at about that time—found it very difficult to understand.

アクタ計算の例をみてみましょう。Carl Hewitt にとって、1とか2とかいった数というのはアクタです。

普通のプログラミング言語では、1と2を加算に送ると考えるでしょう。そして、加算関数が3を返します。しかし Hewitt は違った考え方をしました。1は一つのアクタだとします。もし、1にメッセージを送ると、コンティニュエーション c を得ます。特に、1にメッセージ `(+ 2 c)` を送ると、最終的にアクタ c はメッセージとして3を受け取ります。74年頃の時点で、これは非常に奇妙な考え方でした。Gerry Sussman と、そのころ MIT に学生として入った私には、理解に困難なものでした。

2.4 Conses as Actors (Slide 4)

Conses as Actors

- A cons cell is an actor that knows about two other actors *a* and *d*.
- If you send it the message 'car *c*' then actor *c* receives the message '*a*'.
- If you send it the message 'cdr *c*' then actor *c* receives the message '*d*'.
- If you send it the message 'atom? *c*' then actor *c* receives the message '*false*'.

Let me show you another example of an actor. Suppose we think of a Lisp cons cell as an actor. I will draw a picture. A cons cell is an actor that knows two other actors.

Usually we call the other two actors `A` and `D`, or the `car` and the `cdr`. If we send the message `(car C)` to this cons cell, then eventually actor `C` receives `A` as a message. So, we can ask a cons actor "what is your car?" and it will hand it back to you as a message. Actually it doesn't really hand the actor `A` back to *us*; it hands it to the actor `C`. Nowadays we called the actor `C` a "continuation" but at that time the term was not used. Another message you can send to a cons cell is `(cdr C)`; then eventually actor `C` receives message `D`. Another kind of message you might send to this cons cell is `(atom? C)` in which case `C` receives the message false, that is, "I am not an atom."

To Lisp programmers in 1974, this was an extremely strange concept. Everyone knew that a cons cell was just 36 bits of memory: 18 bits for the `car` and 18 bits for the `cdr`. Anyone could look at the `car` and the `cdr`, and anyone could replace them by using `rplaca` and `rplacd`.

Carl Hewitt said, "No, a cons cell is not an object. It is an actor. You must say, 'please replace your car with the object I give to you. Thank you.' "

アクタの別の例をお見せしましょう。Lisp のコンスセルがアクタだとします。図で示しましょう(図を書く)。コンスセルは、2つの別のアクタを知っているアクタです。

普通、この2つのアクタを `A` と `D` あるいは car と cdr と呼びます。もし、'car c' のメッセージを

このコンスセルに送ると、アクタ c はメッセージとして A を受け取ります。したがって、コンスアクタに、「あなたの car は何か」と尋ねることができるのです。そしてそれをメッセージとして返します。実際にはアクタ a を私たちに返すのではありません。それをアクタ c に渡すのです。今日、私たちはアクタ C を「コンティニュエーション」と呼びます。しかし、そのころはそうした用語はありませんでした。コンスセルに送れるもう一つのメッセージは、'cdr c' です。そうすると、アクタ C は、メッセージ D を受け取ります。このコンスセルに送れるもう一つのメッセージに'atom? c' があります。この場合、C は、偽というメッセージ、「私はアトムではない」ということを受け取ります。

１９７４年の Lisp プログラマにとってこれは非常に奇妙な概念です。すべての人はコンスセルはただ３６ビットのメモリだということを知っていました。１８ビットが car で、１８ビットが cdr です。誰でも、その car や cdr をのぞいてみることができ、rplaca や rplacd で置き換えることができます。

Carl Hewitt はいいます。「そうじゃない。コンスセルはオブジェクトではない。それはアクタだ。『お渡しするオブジェクトであなたの car を置き換えて下さい。お願いします。』といわなければいけない。」

2.5 PLASMA: An Actor Language (Slide 5)

PLASMA: An Actor Language

- Supported actors
- Very complicated language
- Sussman and Steele could not understand
- They decided to write an interpreter for a small actor language
  - Not many features
  - No complicated syntax – write it in Lisp

So, Hewitt designed Plasma, which was an actor language. Everything in this language was an actor. It was a very complicated language and it had a very complicated syntax. Sussman and I could not understood how it worked.

There was an implementation of Plasma in MacLisp. We could try it, and we tried to write Plasma programs. It worked and we could not understand why!

So, Sussman and I educated ourselves to try to understand how it worked. Sussman and I tried to write a small interpreter. The easiest way to understand the language is to write an interpreter. And the best language for writing interpreters is Lisp.

So, we decided to write very small version of Plasma. We were not trying to design next AI language, but wanted to understand Plasma without any unnecessary features. it was small language. Because we wrote the interpreter in Lisp, the toy language naturally had Lisp syntax. We did not have the complicated syntax of Plasma. just a simple Lisp syntax.

Hewitt は、アクタ言語 Plasma を設計しました。この言語の中のすべてのものはアクタです。これは、大変複雑な言語で、大変複雑な文法を持っています。Sussman と私は、それがどう動くのか理解できませんでした。

MacLisp で書かれた Plasma の処理系がありました。それを試すことができました。また、Plasma でプログラムを書いて見ました。それは動作しましたが、なぜ、動くのか理解できませんでした！

Sussman と私はそれがどう動くのか理解しようとしました。Sussman と私は、小さなインタプリタを書きました。言語を理解する最も簡単な方法はそのインタプリタを書くことです。インタプリタを書く最も良い言語は Lisp です。

そこで、我々は Plasma の大変小さなバージョンを書くことにしました。次のＡＩ言語を設計しようという気はありませんでした。そうではなく、不要な機能を省いた Plasma を理解しようとしたのです。それは小さな言語でした。そのインタプリタは Lisp で書いたので、そのおもちゃの言語は lisp 構文を持っていました。Plasma の複雑な構文を持つのではなく、単純な Lisp 構文を持たせたのです。

2.6 The Toy Actor Language (Slide 6)

The Toy Actor Language

- Start with a small Lisp interpreter
- Use lexical scoping
  - Seemed to be necessary for actors
  - Sussman had been studying ALGOL 60
- Add a way to create actors (`alpha`)
- Add a way to ssend messages to actors
  - At first we had a `send` special form
  - Then we realized application could distinguish actors from functions and do the right thing

Here is how we wrote the toy actor language. We started by writing a very small Lisp interpreter, which was about two pages long.

We made one unusual change from the Lisp 1.5 manual. The unusual change was to use lexical scoping instead of the dynamic scoping that was usual at that time. We use lexical scoping for two reasons. First, it looked as if actors would need lexical scoping—that was from our study of the Plasma language. Second, the other reason was that Sussman was teaching Algol 60 at that time and he was interested in lexical scoping. He thought it would be fun to make a language with lexical scoping like algol. This was the direct influence of the design of Algol on Scheme.

We started to write a lexically scoped Lisp. There were two more things to add. One was a way to create actors; the other was message sending. To create actors, we added a kind of expression, the alpha expression, which was just like a lambda expression. An alpha expression just started with the word `alpha`, which is a Greek letter; and then variable names, that represented the elements of messages sent to the actor; And then there was a

body, which was code to be executed when a message was received.

As nearly as we could understand from what Carl Hewitt said, the difference between an actor and a function was that an actor does not return a value. Instead, the body must somehow send the value to other actors. Usually the actor you send something to is in the original message as a continuation.

We chose the letter alpha, because that is the first letter of the Greek word for "actor." So evaluating an expression like this produces, as its Lisp value, an actor. This Alpha expression produces an actor; the Lisp value of the expression is an actor. So that is the way we created actors. To send messages—I asked Sussman about this. I cannot remember and he cannot remember it. But I think we had a special form **send**. But then we realized that the function-application syntax could also be the send-message syntax because if the first thing is a function, it must be a function call, and if the first thing is an actor, it must be a message send. Anyway, in the example I will show you, I will use **send** because it is easier to see what is happening.

So, the way to send a message to an actor is to write **send**, the actor, and then the arguments.

So, it was very small simple language which is just enough to create actors and send messages. And we could experiment with using all functions, or all actors, or combinations of them. Here is sample code in a very early implementation of Scheme.

どうやって、そのおもちゃのアクタ言語を書いたのかお話しましょう。まず、2ページくらいの長さの大変小さい Lisp インタプリタを書くことから始めました。

1つだけ Lisp1.5 マニュアルから特殊な変更をしました。それは、そのころ普通に使われていたダイナミックスコーピングの代わりにレキシカルスコーピングを用いることでした。2つの理由でレキシカルスコープをしました。まず、Plasma 言語を調べた結果、アクタはレキシカルスコーピングがいるように思ったこと。もう1つは、Sussman はその頃、Algol60 を教えていて、レキシカルスコーピングに興味を持っていたことからです。彼は Algol のようなレキシカルスコーピングを持つ言語を作ることはおもしろそうだと感じたのです。これが、Scheme に Algol の設計が及ぼした直接の影響です。

私達はさらに2つのことを加えて、レキシカルスコーピングの Lisp の開発を始めました。一つは、アクタの生成手段、もう一つはメッセージ送信です。アクタの生成のためには、alpha 式を用意しました。これはラムダ式に似ています。アルファ式は、最初にギリシャ文字に由来して **alpha** をおき、次にそのアクタに送られるメッセージの要素を表す変数名を並べます。そのあとに、メッセージを受け取られた時に実行されるコードであるボディを書きます。

Carl Hewitt がいっていることから私達が理解できたのは、アクタと関数の違いは、アクタは値を返さないということでした。そのかわりにボディは何らかの方法で値を他のアクタに送ります。普通、何かを送っている宛先のアクタは、元のメッセージの中に、コンティニュエーションとして存在します。

私達は、alpha という言葉を選びました。といのは、それがアクタというギリシャ語の最初の文字だからです。このような式の評価は、Lisp の値としてアクタを生成します。このアルファ式はアク

タを生成するのです。つまり、その式の Lisp としての値はアクタです。それが私達がアクタを生成する仕方でありました。メッセージを送る仕方について Sussman に尋ねました。私は覚えていません。そして彼も覚えていません。しかし、私達は send という特殊形式をもっていたように思います。しかし、関数適用の構文は、また、メッセージ送信の構文となりうると、その後分かりました。というのは、もし、最初に関数があるのなら関数呼び出しになるし、最初にアクタがあるのならメッセージ送信になるだろうということです。

アクタにメッセージを送るには、send、そのアクタ、そしてその引数という順に書きます。

それはアクタを生成し、メッセージを送るのには充分な、大変小さいシンプルな言語でした。それでもあらゆる関数、アクタ、そしてそれらの組み合わせを試してみることができました。Scheme のまさに初期の処理系でのサンプルコードを示しましょう。

2.7 Functions and Actors (Slide 7)

Functions and Actors

- Factroial function:

```
(define factorial
  (lambda (n)
    (if (= n 0)
        1
        (* n (factorial (- n 1))))))
```

- Factorial actor:

```
(define actorial
  (alpha (n c)
    (if (= n 0)
        (send c 1)
        (send actorial (- n 1)
              (alpha (z) (send c (* n z)))))))
```

Here is the `factorial` function, which is seen in many Lisp tutorials.

Factorial is defined as a lambda expression, which creates a function.

This function takes one argument `n` and if it is zero, it then returns `1` as the value. Otherwise, it makes a recursive call with the value `n-1`, multiplies the result by `n` and returns that. Compare this definition and the actor version; I call it—this is a joke in English—`actorial`.

Actorial is defined as an `alpha` expression that creates an actor. It is not supposed to return a value. Instead, the value `c` is a continuation to which to send the value.

Let us look at `actorial`. Actorial receives two arguments, `n` and `c`. If `n` is zero, we send the value `1` to `c`. Otherwise we send to `actorial` the value `n-1` and a new continuation.

The new continuation is itself an actor, which receives `z`, and it will multiply `n` and `z` and

send the result to `c`. So, eventually, `c` is sent the correct value.

Actually this definition of `actorial` is not a pure actor. This is a kind of mixed notation, because `actorial` is an actor but the equals sign `=` and minus `-` and multiplication `*` are functions.

Suppose that equals sign, minus, and multiplication were actors; then the code would be pure actors. Then on the next slide, `actorial` would look like this.

これは、Lisp の教科書にも良く出てくる階乗 (factorial) のプログラムです。factorial は、関数としてラムダ式で定義されています。この関数は、一引数、n を取り、もしそれがゼロなら 1 を値として返し、そうでなければ、値 n-1 で再帰呼び出しをし、その値と n をかけ、結果を返します。この定義とアクタ版を比べて下さい。アクタ版は、英語のジョークで actorial という名にしています。

actorial はアクタを生成するアルファ式で定義されています。値を返すようには考えられていません。そのかわりに、値 c は、値をそこに送るコンティニュエーションです。

actorial を見てみましょう。actorial は、2引数、n と c を受け取ります。もし、n がゼロなら値 1 を c におくります。そうでなければ、2つの値、n-1 と新しいコンティニュエーションを actorial に送ります。

この新しいコンティニュエーションはそれ自身アクタです。それは、z を受取り、n と z をかけ、c に結果を送ります。したがって、最終的に c は正しい値にセットされます。

実際には、actorial のこの定義は純粋なアクタではありません。これは入り交じった記法です。というのは、actorial はアクタだけれど、等号とマイナスと乗算は関数だからです。

等号とマイナスと乗算をアクタとすると、全体が純粋なアクタとなります。次のスライドでそれをお見せしましょう。

2.8 Detailed Factorial Actor (Slide 8)

Detailed Factorial Actor

```
(define actorial
  (alpha (n c)
    (send = n 0
      (alpha (p)
        (if p
            (send c 1)
            (send - n 1
              (alpha (m)
                (send actorial m
                  (alpha (z)
                    (send * n z c)))))))))))
```

This is the most complicated slide in this talk. If you understand this, the rest is easy.

Okay, so let's try this. It's a little complicated: `actorial` receives the message with two things, `x` and `c`. It then sends three things to to the equals-sign `=` actor: the value `n`, the value `0`, and the continuation. Eventually, the continuation receives the value `p`. If the value is true, then it sends the value `1` to `c`. If `p` is false, then sends three values to minus-sign `-`. The values are `n` and `1` and another continuation.

Eventually, this minus sign sends the value to the continuation. This new value should be equal to `n-1`. This is called `m`. This new value `m` is sent to `actorial` with a third continuation. The value that `actorial` computes is called `z`. This continuation finally sends to the multiplication actor `*` three values. The three values are `n`, `z`, and the continuation, which is the original continuation `c`. Eventually multiplication sends the product to `c`.

That is exactly what we want. If you look at it carefully, it is the same computation. Now you can understand why we found it difficult to understand!

A Plasma program might look like this, but would have more complicated syntax.

The Plasma program is twice as long as the Lisp program. And 10 times as difficult to understand.

Gerry Sussman and I were very pleased to have a small interpreter of a simple language with simple syntax. We wrote many programs in the small language. We took many examples from Carl Hewitt's papers and ran them on our interpreter.

このスライドが私の話の中で最も複雑なものです。あとは簡単になります。

OK, さぁはじめましょう。これはちょっと複雑です。Actorial は、メッセージとして２つのものｘとｃを受け取ります。次に３つのものを等号アクタに送ります。値ｎ、値ゼロ、それとコンティニュエーションです。そして、そのコンティニュエーションは、値ｐを受け取ります。もし、その値が真であれば、値１をｃに送ります。もし、ｐが偽ならば、３つの値をマイナスに送ります。値は、ｎと１ともう一つのコンティニュエーションです。

このマイナスは、値をコンティニュエーションに送ります。この新しい値は、n-1 と等しいものです。これはｍと呼ばれます。この新しい値ｍは、actorial に第三のコンティニュエーションと共に送られます。actorial が計算する値はｚと呼ばれます。このコンティニュエーションは、その後、乗算アクタに３つの値、ｎとｚと元々のコンティニュエーションｃ，を送ります。次に乗算はその積をｃに送ります。

それが、我々がほしいものです。もし、この計算過程を注意深く見てくれれば、これは同一の計算なのがわかるでしょう。なぜ、我々が理解が困難だと感じたかを理解してもらえると思います。

Plasma プログラムは、これに似ていますが、もっと複雑な構文を持っていました。Plasma プログラムは、Lisp に比べて２倍は長く、１０倍は理解するのが困難でした。

Gerry Sussman と私はシンプルな構文のシンプルな言語の、小さなインタプリタが大好きでした。楽しいので、この小さな言語でたくさんのプログラムを書きました。Carl Hewitt の論文からたくさんの例を引いてきて、それを私達のインタプリタで走らせました。

2.9 The Next AI Language ? (Slide 9)

The Next AI Language ?

- Comes after Planner, Conniver, ...
- Let's call it "Schemer"!
- But 1960's-design operating system allowed names only up to 6 characters
- So the file name was "Scheme" – and that became the popular name

We began to be very excited, because we thought it might be a good language for AI. This would not be the first time that Carl designed a complicated language and then Gerry implemented a version we could understasnd and use! Carl Hewitt designed Planner, which was too complicated a language to use, and then Gerry Sussman designed MicroPlanner and people could use it. We thought this was the same story. Carl designed the complicated Plasma language and Gerry Sussman (and I) designed Scheme. We thought this was a simple Lisp version to use. We were not trying to make a new AI language. We thought we were making a simple language, trying to understand it. It was an accident.

But if this language could be the next AI language after Conniver and Plasma, it would need a good AI language name. Planner, in English, means something that plans. A conniver is a sneaky planner. So, what we called it was a very sneaky planner. That is a schemer. We called it "Schemer."

Unfortunately, we had used an operating system designed in the 1960s. Every file name had to be 6 character or less. So, the file name `SCHEMER` was truncated to the first 6 characters and the file name `SCHEME` became the popular name. The whole language was designed by accident and the name was an accident. Many people think we did a wonderful job. I don't understand why!

これは人工知能に良い言語かもしれないと分かってきて、私達はだんだん興奮してきました。
Carl Hewitt が複雑な言語を設計し、Gerry が私達が理解し使えるものを実現してくれるというの

はこれが初めてではありませんでした。Carl Hewitt は、使うのには複雑すぎる言語である Planner を設計し、一方、Gerry Sussman は MicroPlanner を設計し、人々はそれを使いました。これはそれと同一のストーリーだと思いました。

Carl Hewitt は、複雑な Plasma 言語を設計し、Gerry Sussman（と私）は、Scheme を設計しました。これは使えるシンプルな Lisp だと思いました。私達は、新しい AI 言語を作ろうとしているのでは有りませんでした。私達は自分達が理解できるようなシンプルな言語を偶然作ったのです。

もし、これが Conniver や Plasma のあとの新しい AI 言語となり得るなら、よい AI 言語の名前を持つべきです。英語で Planner はプランする何かです。Conniver はスニーキー（こそこそする、卑劣）なプランナーです。だから、もっとスニーキーなプランナーを何と呼ぼうか。そりゃ、スキーマー（陰謀家、策略家）だ。それで、Schemer とつけたのです。

残念ながら、我々は６０年代に設計された OS を使っていたので、すべてのファイル名は６文字以下でなければなりませんでした。それで、ファイル名 `SCHEMER` は最初の６文字だけに切り捨てられました。その結果、`SCHEME` というファイル名がポピュラーな名前となりました。言語全体は偶然に設計され、その名も偶然につけられました。多くの人は私達がすばらしい仕事をしたと考えています。私にはどうしてだか分かりません。

2.10 The Interesting Accident (Slide 10)

The Interesting Accident

- We inspected the code in the interpreter
- The implementation of `lambda` closures and of `alpha` actors was the *same*
- The implementation of function calls and of message `send` was the *same*
- We decided that actors and function closures must be the same thing!

But there were not just two accidents. There was another accidents. The third accident was the interesting accident. We made a Lisp interpreter, which supported functions and actors. And we could call functions with arguments and send messages to actors. Then we looked at the way actors were implemented in this interpreter and we were very astonished. We noticed that the implementation of lambda closures and the implementation of alpha actors were *the same code.* We noticed that function calls and message sending were implemented by *the same code.* SUGOIDESUNE. We decided actors and function closures must be *the same thing.* This had very profound consequences, because we realized this meant that actors could be expressed in lambda calculus. This really wasn't an accident, but it felt like an accident. Sussman and I realized that lexically scoped Lisp is the same as lambda calculus.

This was in 1975. Immediately, we started to read papers about lambda calculus! In particular, we went back to the paper by Alonzo Church, which was written in 1941. And we noticed, in his paper,a simulation of pairs very much like cons cells as actors. I don't think Church understood it to be message passing but it was still interesting to find something like cons cells in a paper from 1941.

Four years after that, in 1979, I got married. My wife and I were both students at MIT. In fact, we were both in Sussman's laboratory. About 6 months after we were married, my wife received a letter from her mother. The letter contained much news about relatives. She

mentioned my wife's grandfather Herbert Taylor and uncle Sam Church. The letter also said, "I just received some mail from cousin Alonzo. He used to be at Princeton." My wife gave to me to read. I said "Barbara, how many persons named Alonzo Church used to be in Princeton but are now retired ?" She said, "There could be only one!... Oh, my goodness!" We two hackers realized that *the* Alonzo Church was her cousin. It is a very small world.

しかし、設計とネーミングの２つの偶然だけでなく、もう一つの偶然もありました。それはおもしろいものでした。私達は、関数とアクタをサポートするLisp インタプリタを作りました。関数を引数をつけて呼びだしたり、メッセージをアクタへ送ることができます。そしてそのインタプリタにアクタが実装された仕方を見て、私達は、大変驚きました。ラムダクロージャの実装とアルファアクタの実装は、同一のコードだったのです。

「すごいですねぇ。」

我々はアクタと関数クロージャは同一のものだと決断しました。これは大変有益な結論です。というのは、このことから、アクタはラムダカリキュラスで説明できるということを意味すると気がついたのです。これは、実際は偶然ではありませんでした。しかし、偶然のように感じました。Lisp の中のレキシカルスコーピングはそれと同一のものです。Sussman と私はレキシカルスコープのLisp は、ラムダカリキュラスと同一であるということを発見しました。

それは、１９７５年のことです。すぐにラムダカリキュラスの論文を読みだしました。特に、私達はAlonzo Church の１９４１年の論文に戻りました。彼の論文の中では、アクタとしてのコンスセルのようなペアのシミュレーションが論じられていることに気がつきます。彼がそれをメッセージ送信になるべきだと理解していたとは思いませんが、１９４１年の論文でコンスセルのような何かがでているのを見つけるのはたいへん興味深いことです。

それから４年後、１９７９年に私は結婚しました。妻と私は２人ともＭＩＴの学生でした。事実、私達はSussman の研究室にいました。結婚して６カ月後に妻は、彼女の母から、親戚のことがたくさん書かれている手紙を受取りました。彼女は祖父のHerbert Taylor と叔父のSam Church について話してくれました。また、「いとこのAlonzo からの手紙をもらった。彼はプリンストンにいました。」と書かれていました。それを彼女は私に見せてくれました。

私はいいました。「バーバラ、プリンストンにいて、引退したAlonzo Church という人は何人いると思う？」彼女は、「たった一人よ。」「なんてこった！」Alonzo Church は、彼女のいとこだったのです。全く世界は狭いですね。

2.11 How Can This Be ? (Slide 11)

How Can This Be ?

- Functions are intended to return values
- Actors are intended to send messages

- So what ?

Sussman and I decided that functions and actors are the same. How can this be? Function are supposed to return values. Actors are not. Instead, they send messages. The correct answer to the question is "So what?" "NAN-DEMO"

You cannot tell whether or not something is an actor or a function, because all it does is receive arguments and do something.

Sussman と私は関数とアクタは同一だと認めました。どんなことになるのでしょう？関数は値を返すものと考えられています。アクタはそうではありません。かわりにアクタはメッセージを送ります。この質問に対する正しい答は「それがどうした。」「なんでも、いいじゃないか」です。

引数を受取り、何かをするからということで、何かがアクタであるか関数であるかを、識別することはできないのです。

2.12 It depends on the Primitives (Slide 12)

It depends on the Primitives

- What can an actor or closure do ?
  - Invoke another actor or closure
  - Maybe do a conditional test
  - Invoke a primitive – this is the basis case
- If all the primitives return values, then every actor/closure will return a value
- If all the primitives send messages, then every actor/closure will send a message

Let us ask what kind of things an actor or closure can do. It can do several things. It can invoke another actor or closure; this is a recursive operation. Maybe it can do a conditional test; this is also a recursive case: the "then" part or the "else" part does something else. Or it can invoke some kind of primitive in the language. This is the basis case for induction. But what Sussman and I discovered by accident is if the primitives in the language return values, then every actor or closure would return a value. If all the primitives send messages, then every actor or closure will send messages.

So whether something behaves as an actor or a closure depends on the primitives that it uses. Whether a language is pure functions or pure actors depends on its primitives.

アクタ、あるいはクロージャがどんなことをできるか考えてみましょう。いくつかのことがあります。まず、他のアクタやクロージャを呼び出せます。再帰的に行なってもかまいません。たぶん条件テストもできるでしょう。これも再帰的な場合もあります。then 部あるいは else 部が何か別のことをしてもいいです。あるいは、言語に用意されたプリミティブを呼び出すこともできます。帰納の基礎となるケースです。しかし Sussman と私は偶然、もし言語中のプリミティブが値を返すなら、すべてのアクタあるいは labels クロージャも値を返すことを発見しました。もしすべてのプリミティブがメッセージを送るなら、すべてのアクタやクロージャもメッセージを送るのです。

だから、アクタやクロージャとしてふるまうものというのは、用いているプリミティブに依存するのです。したがって言語が純粋に関数なのか純粋にアクタなのかはそのプリミティブに依存します。

2.13 Initial Report on Scheme (Slide 13)

Initial Report on Scheme

- A very simple language
  - Function constructor `lambda`
  - Fixpoint operator `labels`
  - Conditional `if`
  - Side effect `aset`
  - Continuation accessor `catch`
  - Function application
  - Variables
  - Some primitive Lisp data types (lists, numbers)

Shortly thereafter, Sussman and I wrote an initial report on Scheme. This was in December 1975. There was a function constructor named `lambda`; as we have seen, it is also an actor constructor. We added fixed point operator, `labels`; this was like the `label` in Lisp 1.5. In modern programming languages, it is usually called `letrec`. We had one conditional operator, `if`. We could have put `cond` instead of `if`, but we thought `if` was simpler.

We had one side effect called `aset`, sort of like `setq`. We put in a way to get the implicit function continuation, called `catch`. Catch was a special form; in modern versions of Scheme it is usually called `callcc`. Function application gives arguments to functions. And of cource you can refer to variables. There were some primitive Lisp data types: lists, numbers, and others. And that's about all there was in the first Scheme. We tried very hard to include only one kind of each thing, because we wanted to keep the language very small, for educational purpose rather than for development.

そのあと、Sussman と私は Scheme に関する最初のレポートを書きました。１９７５年１２月のことです。`lambda` という関数コンストラクタをおきました。見てきたように、それはアクタコンス

トラクタでもあります。`labels` という不動点オペレータも加えました。これは Lisp1.5 の label のようなものです。モダンなプログラミング言語では letrec と呼ばれています。条件オペレータとして if を加えました。if のかわりに cond を置けたかも知れませんが、その方がシンプルだと思ったのです。

setq のような aset と呼ぶ副作用オペレータを置きました。それらに加えて、catch と呼ぶ暗黙の関数コンティニュエーションを得る方法を加えました。catch は特殊形式で、最近の Scheme では callcc と普通呼んでいるものです。関数の適用は、関数に引数を与えます。そして、変数を参照することももちろんできます。リスト、数、などの基本的なデータタイプがあります。それが最初の Scheme でした。いろいろなことにはできるだけ 1 つのことをいれるだけで済ませようとしました。開発用というのではなく、教育の為に、この言語を大変小さいものに保とうとしてきたからです。

2.14 Lambda:The Ultimate Imperative (Slide 14)

Lambda:
The Ultimate Imperative

- Paper on modelling control structures
- Translated many existing ideas into Scheme to show how simple it was
- Drw ideas fromPeter Landin and John Reynolds, among others

Sussuman and I then tried to use this small language to explain control structures all over again. Carl Hewitt had written a paper about actors called *Viewing Control Structures as Patterns Passing Messages.* It was 1974 or 1975. Hewitt's paper explained iterations, recursions, loops, and all kinds of complicated control structures in terms of actors and messages. Sussman and I were, with Scheme, trying to explain Carl Hewitt's work in a simpler way.

So we wrote a paper called *Lambda: The Ultimate Imperative* in which we tried to explain control structures in terms of lambda calculus. This was not a new idea. Peter Landin and John Reynolds tried to explain Algol in terms of lambda calculus. The whole theoretical area of denotation semantics was emerging at that time. In effect, Sussman and I were repeating these ideas in our own framework.

Our new contribution was that the models of control structure in lambda caliculus could be executed. We could take all the theoretical lambda calculus models and translate them into Lisp syntax and run them. So, other people at MIT found it very useful.

Sussman と私は、次にこの小さな言語を使って制御構造全体というものを、もう一度説明しようとしました。Carl Hewitt は、アクタに関する論文を書きました。「Viewing Control Structures as Patterns of Passing Messages」です。７４年か７５年のことです。Hewitt の論文は、繰り返し、再帰、ループ、そしてあらゆる種類の複雑な制御構造をアクタとメッセージによって説明しました。Sussman と私は、Scheme を使って、Carl Hewitt のやっていることをシンプルな方法で説明しようとしました。

それで、「Lambda: the Ultimate Imperative」という論文を書きました。そこでは、制御構造をラムダカリキュラスを使って説明しようとしたのです。それは新しいアイデアではありませんでした。Peter Landin と John Reynolds は、Algol をラムダカリキュラスで説明しようとしました。denotational semantics という理論領域がその頃現れてきました。事実、Sussman と私はこれらのアイデアを自分達のフレームワークで繰り返しました。

私達の新しい貢献だというのは、ラムダカリキュラスでの制御構造のモデルは実行できるということでした。ラムダカリキュラスの理論モデルをすべて Lisp 構文則に翻訳し、それを走らせました。それで、MIT の他の人たちは大変便利だと言ってくれました。

2.15 Lambda: The Ultimate Declarative (Slide 15)

Lambda:
The Ultimate Declarative

- Lambda as a renaming construct
- Function call is GOTO (with arguments)
- More on similarity of actors and closures
- Object-oriented programming in Lisp
- Suggestions for writing good compilers

In our next paper, *Lambda: The Ultimate Declarative*, we looked at the declarative side of lambda.

In this paper, we emphasized two points. First, the main purpose of lambda is to give names to the arguments that arrive in a message. The other thing was that function call doesn't necessarily return a value. If a function call does not have to return a value, it doesn't have to return at all. We can understand this because a function is the same as an actor, and a call is the same as sending a message.

This suggested that function call can be thought of as goto, but it also passes arguments. This idea tells us that we can compile the function call as gotos. In particular, function call doesn't have to push a return address on the stack; you push a return address only if you want to come back. Or, to think differently about this, the return address is really a continuation. If you don't want to make a new continuation, you don't need to push a return address.

We realized, if you design a compiler around the principle, tail recursion is automatic. And is the compiler guarantees proper tail recursion, then you could do object-oriented programming in Lisp.

次の論文「Lambda: The Ultimate Declarative」では、ラムダの宣言的サイドを見ていました。

この論文では、2点を強調しました。まず、ラムダの主な目的は、メッセージに到着する引数に名を与えることです。もう一つは、関数の呼び出しは必ずしも値を返すわけではないことです。もし、関数呼び出しが値を返さなくても良いとすると、帰る必要もないのです。このことを、関数はアクタと同じであり、呼び出しはメッセージと同じなので、理解できます。

これは、関数呼び出しを、引数を渡せるGoToのようなものとしてコンパイルできることを教えてくれます。特に、関数呼び出しは、戻り番地をスタックに積む必要がなくなるのです。戻りたいときにだけ積めばよい。あるいはこのことを別の考え方でみると、戻り番地はまさにコンティニュエーションです。もし、新しいコンティニュエーションを作りたく無ければ、戻り番地をプッシュする必要はないのです。

この原則に基づいてコンパイラを設計するなら、末尾再帰は自動的に処理できるようになります。そしてそのコンパイラは末尾再帰の正しい処理を保証します。そして、Lispでのオブジェクト指向プログラミングができるようになるのです。

2.16 RABBIT: A Compiler for Scheme (Slide 16)

RABBIT:
A Compiler for Scheme

- Steele's master's thesis
- How lambda calculus supports code transformations for optimizing compilers
- Written in Scheme
- Very slow compiling itself!

Following these ideas, I actually built a Scheme compiler as part of my master's thesis. In this, I studied the idea of compiling code as we have discussed. I also studied how you can use lambda calculus to describe compiler optimization. At that time, the 1970s, program transformation at the source level was still a new idea. I realized program transformations described as denotational semantics could be used directly in a compiler.

At that time, I strongly was influenced by the saying, "A programming language is no good if its compiler cannot be written in the language." So, I started to write compiler in Scheme. I found another problem in bootstrapping. To get the compiler to compile itself, I had to run interpreter. Worse yet, as I made improvements, it needed even longer compile time to compile itself. So I got a computer terminal and took it home with me. Every night, I started the compiler. When I woke up, usually it was done. I had to compile about one hundred times in three months. Once I was able to get the compiler working, then the compiled compiler could compile the compiler and it ran much faster. But several times I changed the compiler and introduced a bug. Then I had to I re-bootstrap using the interpreter. This was very tedious work!

Let me show you a few examples of compiler optimization based on the lambda calculus.

これらのアイデアに従って、私は自分の修士論文の一部として、Scheme コンパイラを実際に作りました。その中で、私は今まで話してきたようなコードコンパイルのアイデアについてまとめました。

また、ラムダカリキュラスを使って最適化コンパイルをする方法をまとめました。その頃、７０年代では、ソースレベルのプログラム変換はなおも新しいアイデアでした。私は denotational semantics として説明されるプログラム変換は、コンパイラ中で直接使えることを認識しました。

その時、私は「その言語でコンパイラが書けなければ、いい言語ではない」という言葉に強く影響を受けていました。だから、私は Scheme でコンパイラを書きました。ブートストラッピングでの別の問題を発見しました。自分自身コンパイルするようにするには、インタプリタを走らせなければなりませんでした。更に悪いことに、改良をどんどんすると自分自身をコンパイルするコンパイル時間はどんどん長くなります。そこで、家に端末を持って帰って毎晩コンパイルしました。朝起きるといつもできていました。３カ月で１００回くらいコンパイルしたことになります。コンパイラをうまく作れると、それを使ってそのコンパイラをコンパイルして、もっと速く実行できるようにします。けれども、何回かはコンパイラにバグをいれてしまったこともありました。その場合、インタプリタを使ってブートストラップし直しました。これはとてもあきあきするような作業でした。

ラムダカリキュラスに基づくコンパイラ最適化の２、３の例を示しましょう。

2.17 Example: OR (Slide 17)

Example: OR

```
(or x y)

      (if x x y)

                ((lambda (p)
                   (if p p y))
                 x)
                             ((lambda (p q)
                                (if p p (q)))
                              x
                              (lambda () y))
```

The first example is an `or` expression, `(or x y)`. The meaning is as in Lisp: if `x` is true, then return true and don't evaluate `y`; but if `x` is false, then evaluate `y` and return that value. So here is a possible translation, `(if x x y)`. But you don't want to do this as a macro, because `x` may be a complicated expression. Maybe there is a side effect; you don't want to do it twice.

So what you want to do is evaluate `x` once, look at the value, and, if it is true, return that. Don't execute it again. Many compilers had this idea already; the C compiler had this idea already. But until that time, no compiler could handle it as a source to source transformation. In Rabbit, we did try this extension: we evaluate `x` and bind the value to the variable `p` then test that. Here is the source code: we expressly evaluate `x` only once. But there is another problem. What if the variable `p` appears in `y`? Then the source expression `y` will refer to the wrong variable `p`. The usual trick at that time was to use a generated variable name, a "gensym." But Sussman and I did not find it satisfactory with this hack. Such magic should be outside the language. We wanted to do the source transformation completely within the language using lambda variable.

We discovered we can do this by using more lambda expressions. In this lambda expres-

sion, we bind the variable x to the variable p, and we also bind the variable q to another lambda expression, which then calls and execute y.

In Algol literature, this lambda expression is called a "thunk." By the way, "thunk" sounds very silly in English.

The meaning of this is that this lambda expression (the outer one) simply names two things. It names the variable x and it names this thunk. Now, using the names p and q, we say, if p is true then p, otherwise call q. We can think that this function call is actually changed to a goto to the thunk, which is in the correct scope.

最初の例は or です。(or x y)。この意味は Lisp の場合と同じで、もし x が真なら真を返し、y の評価はしない。しかし、もし x が偽ならば y を評価し、その値を返す。そこで、次のような変形もありえます；(if x x y)。しかし、これは、マクロとしてはやりたくありません。特に x が複雑な式の場合に。またそれは副作用があるかもしれません。2 回実行はしたくありません。

したがって、やりたいことは x を 1 回だけ評価し、その値を見て、もしそれが真ならそれを返し、再び、実行しないようにさせます。多くのコンパイラはこのアイデアをすでに持っています。C コンパイラもすでに持っています。しかしその頃までは、どんなコンパイラもソースからソースへの変換で扱ってはいませんでした。Rabbit では、この拡張を試みました。x を評価し、その値を変数 p に束縛し、それをテストします。ここのソースコードを見て下さい。x の評価は 1 回だけおこないます。もう一つ別の問題があります。変数 p が y の中で使われていたとしたらどうしますか？ y の中では誤った p を参照してしまいます。その頃の普通のトリックは gensym で変数名を作ることです。しかし、Sussman と私はこの仕掛では満足できませんでした。そうしたマジックは言語の外にあるべきです。私達はラムダ変数を使って、完全にその言語の範囲内でソース変換をしたかったのです。

私達は、もっとラムダ式を使うことでこれが可能なことを発見しました。このラムダ式では、変数 x を変数 p に束縛し、変数 q を別のラムダ式に束縛しそしてそれが y を呼びだし、実行します。

Algol の分野では、このラムダ式は thunk と呼ばれます。さて、thunk は英語では変な（silly）名前です。

この意味は、このラムダ式（外側のもの）は単に 2 つのものを名付けます。変数 x とその thunk を名付けます。ここで名前 p と q を用います。もし p が真なら p，そうでないなら q を呼びます。この関数呼び出しを、正しいスコープにある thunk への goto に実際に変更することを考えることができます。

（なぜ thunk という名前が silly だと感じたかについての補足：
二つの理由があった。まず、第一に、音のひびき。thunk というのは何か大変堅いものにある程度堅いものがぶつかったときの音。たとえば、The bat hit the baseball, thunk!など。第二に、子供たちはしばしば、think の過去形は thunk だと間違える。sink-sunk、stink-stunk などがあるから。Here is a new game I thunk up などといったりする。本当は thought up なのに。だから、その言葉は子供の言葉のように思える。それで、大人の英語の話し手には、thunk を名詞として使うのは silly だと言える。）

2.18 Example: IF (Slide 18)

Example: IF

```
(if (if a b c) d e)

        (if a (if b d e)
              (if c d e))

                ((lambda (x y)
                   (if a (if b (x) (y))
                         (if c (x) (y))))
                 (lambda () d)
                 (lambda () e))
```

As the second example, consider this if expression. Suppose we have a nested if expression. It was recently discovered that a good source to source transformation is to change it to something like that.

I first found this transformation in a catalog at UC Irvine. The Irvine Program Transformation Catalog was the first attempt to systematically list transformations as source-to-source level transformations.

One aspect of this transformation is not satisfactory, Because the code for `d` and `e` is duplicated. We can use this idea: you need only one copy of `d` and `e`. And after going down this decision tree, you can go to the correct piece of code. Using Scheme, the Rabbit compiler expressed it in this way. `x` and `y` name two thunks for `d` and `e`. So, the code for `d` appears only once, and code for `e` appears only once. Then, actually, this executable code is the decision tree on `a`, `b`, and `c`. After making this decision, then a direct `goto` is made to the thunk for `d` or the thunk for `e`. And the Rabbit compiler compiled such code exactly that way. These function calls turned into branch instructions. So the interesting thing is, these lambda variables (`x` and `y`) do not stand for data; `x` and `y` are more like statement labels. So, Scheme is able to explain names of variables and names of statement labels in one mecachims.

2つ目の例として、このifを考えてみましょう。入れ子になったifがあるとします。ここで示すようなものへのソース変換がよいことが最近発見されています。

私はこの変換をカリフォルニア大学アーバイン校のカタログの中で発見しました。アーバインのプログラム変換カタログは、まずソースレベルでの変換として変換をシステマティックに並べようとした最初のものでした。

この変換は必ずしも充分なものではありません。というのは、dとeのためのコードが重複しているからです。この考え方によるとdとeに対して、ただ一つのコピーだけを持てばいいのです。そしてこの意志決定木を下がって行って正しいコードの断片に到達します。Schemeを使ってRabbitコンパイラは次のように表現します。xとyがdとeに対する2つのthunkを名付けます。それでdのコードは一回だけあらわれます。eのコードも一回だけ現れます。その後、実際の実行できるコードは、a,b,c上の意志決定木となります。この意志決定した後、dのthunk、あるいはeのthunkのどちらかへの直接のgotoがおきます。Rabbitコンパイラはまさしくこのようにコンパイルします。これらの関数呼びだしは、分岐命令になるのです。それで、興味深いことは、これらのラムダ変数(xとy)は、データ変数を現すのではないことです。xとyは、文のラベルのようなものです。それで、Schemeは変数の名と文ラベルの名を1つのメカニズムで説明できるのです。

(UC Irvineのカタログについての補足：
これは、次のものである。

Standish, T.A. et. al. "The Irvine Program Transformation Catalogue", University of California (Irvine, CA, Jan. 1976))

2.19 Revised Report on Scheme (Slide 19)

Revised Report on Scheme

- Updated definition of Scheme
  - Dynamic binding
  - Built-in macros: `let, cond, do, ...`
  - User-defined macros
- Tribute to Revised Report on Algol 60

Sussman and I wrote several more papers about Scheme. The next was the "Revised Report on Scheme." This was a slightly updated definition of Scheme. The main new feature was to add dynamic binding, so each variable could be lexical or dynamic—the programmer could choose.

Also, by this time, other users were using Scheme, basically for writing code. So, we added a few more things to make Scheme more convenient. We added let, cond, and do as macros, also user-defined macros. We called this paper the *Revised Report on Scheme* because we want to pay honor to Algol 60. We thought the *Revised Report on Algol 60* was a very clearly written paper. We tried very hard to write clearly at that level.

Sussman と私は Scheme に関して更にいくつかの論文を書きました。次の論文は「Revised Report on Scheme」です。これは Scheme の定義を多少改訂したものです。主な新しい機能としては、ダイナミックバインディングを追加したことがあります。それで各変数はレキシカルかダイナミックのどちらであってもよく、プログラマが選択できるようになりました。

この時までに、他のユーザは基本的にコードを書くために Scheme を使っていました。それで、Scheme をもっと便利にするためにいくつかの機能を加えました。let, cond, do をマクロとして加えました。また、ユーザがマクロを定義できるようにしました。この論文を我々は、「Revised Report on Scheme」と呼びました。というのは、私達は Algol60 に敬意を払いたく思っていたからです。「Revised Report on Algol60」は大変良く書かれたレポートだと思っていました。私達はそれと同じ

くらいにきれいに書きたいと努力しました。

2.20 The Art of the Interpreter (Slide 20)

The Art of the Interpreter

- Monograph on small Lisp interpreters
- Illustrates how small variations affect the behavior of the interpreted language
- Exploration of side effects and state

Next, we wrote another paper, called *The Art of Interpreter.* We wanted to explain to other people how you can experiment with programming language design.

While Sussman and I were designing Scheme and writing other papers like *Lambda: The Ultimate Declarative*, we wrote many small interpreters. Sometimes we wrote ten different interpreters in a week. It's very easy when you are modifying an old interpreter and making a new one, and the language is small. We wrote interpreters to illustrate how small changes in an interpreter can make a big difference in the programming language.

We also explained side effects and states. The best definition of side effect came from Dan Weinreb. We were trying to understand the meaning of "two objects are the same." Sussman and I read many books by great philosophers and tried to understand this idea. But, Dan Weinreb said, "Two coins are the same if one is on the railroad track and the other gets squashed." It meant that if you have two objects and change one and it always affects the other one, then they are the same.

次に私達は、別の論文を書きました。「The Art of Interpreter」というものです。プログラミング言語の設計をどうやって実験できるのか説明しようとしたものです。

Sussman と私がSchemeを設計している間に、「Lambda: The Ultimate Declarative」など、いくつかの論文を書き、また、たくさんの小さなインタプリタを書きました。ある時などは、一週間に１０あまりの異なるインタプリタを書きました。古いインタプリタを修正し、新しいものを作るのは、そして、言語が小さいのであれば大変簡単な作業でした。インタプリタを書いて、インタプリタの中のさまざまな小さな違いが言語に大きな違いを与えることを示しました。

私達は、また、副作用とステートについても説明しました。副作用のもっともよい定義は Dan

Weinreb によるものです。私達は「２つのオブジェクトが同一である」ことの意味を理解しようとしたことがあります。Sussman と私は、偉大な学者の本を調べてそれを理解しようとしました。しかし、Dan Weinreb は、言いました。「２つのコインがあって、一方がレールの上にあって、もう一つがつぶれたらそれは同一のものだ。」つまり、２つのオブジェクトがあって、一方での変化が常に他方に影響するのであればそれらは同一だ、という意味です。

2.21 Scheme Spreads Elsewhere (Slide 21)

Scheme Spreads Elsewhere

- Indiana University
- Yale
  - T compiler
  - ORBIT compiler
- Texas Instruments (PC Scheme)
- Oregon (MacScheme)

About this time, Scheme became much more popular. Sussman and I exchanged papers with workers at Indiana university in the 1970s, Daniel Friedman and David Wise. Indiana and Yale did work on Scheme. Yale produced two very good Scheme compilers. In the early 1980s, commercial versions of Scheme appeared. Texas Instruments produced TI Scheme, with a very low cost, under $100. Workers in Oregon including Will Clinger produced MacScheme. At about this time, Common Lisp efforts started.

このころ、Scheme はもっとポピュラーになってきました。Sussman と私は１９７０年代にインディアナ大の研究者、Daniel Friedman と David Wise たちと、論文を交換しました。インディアナ大学、イエール大学などが Scheme の研究をしていました。イエールは２つの大変良い Scheme コンパイラを作りました。８０年代初期には、商用の Scheme が出現してきました。Texas Instruments は 100 ドル以下という大変安い価格で TI Scheme を出しました。Will Clinger のいたオレゴン大学の人たちは MacScheme を作りました。そのころ Common Lisp の開発がはじまったのです。

2.22 Scheme Affects Common Lisp (Slide 22)

Scheme Affects Common Lisp

- NIL dialect planned for VAX, S-1
- Had lexical scoping
- NIL was one basis for Common Lisp

Scheme affected the design of Common Lisp. There was a strange dialect called NIL—I was involved with starting the project to some extent. It was designed to build a Lisp for the Digital VAX and the Stanford S-1. NIL was very much like MacLisp, but it had lexical scoping. NIL then become an important dialect and joined Common Lisp. That's why Common Lisp has lexical scoping.

Scheme は Common Lisp の設計に影響を与えました。NIL というちょっと奇妙な名前の Lisp があります。このプロジェクトの開始には多少関わっていました。これは Digital VAX とスタンフォード S1 に対する Lisp を作ろうとして設計されたものです。NIL は Mac Lisp に似ていますが、レキシカルスコーピングをします。NIL はその後、Common Lisp の重要な前身となりました。それが Common Lisp がレキシカルスコーピングをする理由です。

2.23 More Reports on Scheme (Slide 23)

More Reports on Scheme (by a committee)

- Revised Revised Report
- Revised Revised Revised Report
- Revised Revised Revised Revised Report
  - A version of this became IEEE Scheme standard
- The *Revised*[5] Report is still in progress

Because so many people were involved with Scheme, a standardization committee was formed. The committee started trying to improve Scheme and wrote series of reports. They made the Revised Revised Report, the Revised Revised Revised Report, the Revised Revised Revised Revised Report, and now they are working on the Revised Revised Revised Revised Revised Report.

They revised the report four times. It became the IEEE Scheme standard.

The IEEE Scheme standard is five years old now. There was a question os whether we should affirm or revise. We affirmed. It looks like the same language that the Scheme community provided for five years is still good.

たくさんの人々が Scheme に関係してきたので、標準化委員会が作られました。委員会がはじまり、Scheme を改良しようとしていくつものレポートが書かれました。委員会では、Revised Revised Report, Revised Revised Revised Report、Revised Revised Revised Revised Report、そして今は、Revised Revised Revised Revised Revised Report が作られています。

Scheme のレポートは 4 回改訂され、それが IEEE Scheme 標準となりました。

IEEE Scheme 標準は 5 年前に出ています。それを改訂するか否かが問題になったことがあります。我々はそのままで良しとしました。Scheme コミュニティは、同一の言語仕様を 5 年間提供してきたのです。

2.24 Scheme Community Is Very Friendly (Slide 24)

Scheme Community Is Very Friendly

- Operates by consensus
- If someone says "no" to a feature, then it does not go into the language
- Compare this to Common Lisp committee

The Scheme community is very friendly. They operate by consensus agreement mainly. The general rule of the committee when working on the revised report was, "If a change is proposed, and if someone says no, it is not included." Put in another way: everyone must agree on a change. This tends to keep language small.

Maybe you can compare this with Common Lisp community. Maybe the Common Lisp community is not friendly. It does not operate by consensus. I have chaired the committee for five years. I have moderated arguments. This was the most difficult job of my life! Some people think that in Common Lisp, the rule was, "If a change is proposed, and someone says yes, the feature goes into the language!" So Common Lisp becomes big.

Maybe this comparion is not fair. I think the purpose of Common Lisp is very different. Common Lisp was intended to be an industrial programming language. Keeping Common Lisp small for educational purposes was not a goal. So, Common Lisp became a very, very big subroutine package. I think it is easier to use, and harder to teach.

Scheme コミュニティは、とてもフレンドリです。ほとんど彼らはコンセンサスで合意してきました。Revised report の作業をしていたときの委員会の大原則は、「もし変更が提案された場合、誰かが no といったらそれは入れられない」ということでした。別の言い方をすると、誰もがそれに同意しないと直さない。これは言語を小さいものに保ちます。

これと Common Lisp コミュニティと比べてもいいかもしれません。Common Lisp はフレンドリではないとも言えるでしょう。コンセンサスで動くのではなかった。5年間私は委員長をしていました。論点の調停をしていました。それは私の一生でもっとも困難な仕事でした！Common Lisp での原則は、「変更が提案される時、誰かが yes と言うとその機能は言語に入れられて行きます！」だから、Common Lisp は大きくなりました。

この比較はフェアでないかもしれません。Common Lisp の目的は大変異なっているものだと思います。Common Lisp は産業用のプログラミング言語となるように意図されていました。教育のために Common Lisp を小さいままにすることはゴールではありませんでした。だから、Common Lisp はとってもとっても大きなサブルーチンパッケージとなったのです。それを使うのは簡単ですが、教えるのは困難です。

2.25 Contributions of Scheme (Slide 25)

Contributions of Scheme

- A small language
- Almost no new ideas
- Showing that a few old ideas explain a lot
- Easy to implement
- Bridge between theory (lambda calculus, denotational semantics) and practice (practical programming language)

I would like to modestly suggest these are the contributions of Scheme. It is very small but still a useful language. Actually, there are almost no new ideas. Instead, by accident, it showed that a few old ideas can explain a lot, and especially that a programming language based on lambda calculus is useful. Because Scheme is small, it is easy to implement, and it provides the bridge between theory and practice. The way of theoretical lambda calculus and denotational semantics can be used for writing practical compilers.

Scheme の貢献にはどんなものがあったのか、控えめにまとめてみたいと思います。まず、Scheme は大変小さな、しかし、便利な言語です。実際にそこには新しいアイデアは何もありません。そのかわりに、偶然、いくつかの古いアイデアがたくさんのことを説明できることを示しています。ことに、ラムダカリキュラスに基づく言語が有用なことを示しました。Scheme は小さいので、実現は容易です。そして、理論と実際の間の架け橋を与えています。理論的なラムダカリキュラスと denotational semantics のやり方は実際的なコンパイラを書くのにも便利なものです。

2.26 The Future of Scheme (Slide 26)

The Future of Scheme

- IEEE standard reaffirmed (5 years) with no changes
- Maybe add macros and modules ?
- Used a sextension language for CAD
- Actually pretty good as it is
- Transfer of technology to ML, Haskell ?

Here is my guess about future of Scheme. As I said, the IEEE standard is firm and has not changed for five years. The extra changes the users suggested were macros and some kind of module systems. Unfortunately, after several years, there is not good agreement for these features.

There is pretty general agreement that macros are a very good thing. There are several design proposals for macros. The same thing is true for modules.

My guess is the Scheme will probably stay pretty much as it is now. That may be okay; it is widely used in education, in both high schools and universities. It is also found in industrial applications, for example CAD, for extension languages; just as the Emacs editor has Emacs lisp as an extension language, some CAD programs use Scheme. So, overall, Scheme may be pretty good and seems no needs to change.

In the last few years, I have seen a transfer of technology from Scheme to other functional languages. The programming languages ML and Haskell are like Scheme in many ways, but with Algol-like syntax. In some ways, the kind of programming language innovation that used to be done in the Scheme community, I now see done by ML and Haskell.

I am now not sure whether Scheme is frozen or whether innovation from other areas will come back to Scheme. For example, maybe Scheme will eventually have a module system based on the ML design.

As always, the future is a little uncertain, but I think Scheme for now is very useful version of Lisp.

I think this is the end of my presentation.

DOUMO ARIGATOU GOZAIMASITA.

Scheme の将来について私の推理を述べましょう。もう言ったようにIEEE 標準はしっかりとしたもので5年間変化がありませんでした。ユーザが変更してほしいと示唆したものは、マクロと、ある種のモジュールシステムでした。不運なことに、数年を経てもこれらの機能に良い合意はありませんでした。

マクロが大変良いものだという点には、かなり共通した合意があります。マクロにはいくつかの設計プロポーザルがありました。モジュールについても同じことが言えます。

私の推理では、Scheme はおそらく今と同じようなままでいるだろうと思っています。それでOKかもしれません。高校や大学や教育機関で広く使われているのです。また、CAD の機能拡張言語用、ちょうどEmacs には Emacs Lisp があるように、などのインダストリアルな応用も見られます。CAD のいくつかは Scheme を使っています。Scheme は全体として大変よく出来ていて、変更の必要はないようにも見えます。

最近の数年間、私は Scheme から他の関数型言語への技術移転を認識しています。プログラミング言語 ML や Haskell は Algol に似た構文を持っていますが、多くの点で、Scheme に似ています。かって Scheme コミュニティにあったたぐいのプログラミング言語のイノベーションが、ML や Haskell でなされているのを知っています。

Scheme は凍結されているのか、Scheme にイノベーションが戻ってくるのかはよくわかりません。たとえば、Scheme は、徐々に ML のデザインに基づくモジュラーシステムになるのでしょう。

いつものように将来は不確定です。でも Scheme は今の所、もっとも便利な Lisp だと思っています。

これで、私の話は終わりです。

どうもありがとうございました。

Chapter 3

Readings and References

This section contains the lists of various publications both which have quoted in the talk, and which have key milestone relation with the background of the Scheme talk, gathered by Masayuki Ida.

(The lists are sorted by publication year.)

3.1 Scheme Itself

Revised^5 Report is reportedly going on.

Sussman, Gerald Jay, and Steele, Guy Lewis Jr. "SCHEME: An Interpreter for Extended Lambda Calculus", AI Memo 349 MIT AI Lab (Cambridge, December 1975).

Clinger, William, "The Revised Revised Report on Scheme: or, An Uncommon Lisp", AI Memo 848, MIT AI Lab (Cambridge MA, Aug. 1985).

Rees, Jonathan and William Clinger. (eds.) "*Revised*^{3} Report on the Algorithmic Language Scheme" ACM Sigplan Notices, vol.21 no.12, December 1986.

IEEE CS, "IEEE Standard for the Scheme Programming Language" IEEE STD 1178-1990, 1991.

3.2 Guy Steele's Related Publications

This list does not always cover his ALL publications during the period. Please ask him directly for more detail if needed.

Steele, Guy Lewis Jr., and Sussman, Gerald Jay. "LAMBDA: The Ultimate Imperative" AI Memo 353 MIT AI Lab (Cambridge, March 1976).

Steele, Guy Lewis Jr. "LAMBDA: The Ultimate Declarative" AI Memo 379. MIT AI Lab (Cambridge, November 1976).

Steele, Guy Lewis Jr. "Compiler Optimization Based on Viewing LAMBDA as Rename plus Goto" S.M. thesis. MIT (Cambridge, May 1977). Published as "RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization)." AI TR 474. MIT AI Lab (Cambridge, May 1978).

Steele, Guy Lewis Jr. "Macaroni is Better than Spaghetti" Proceedings of the Symposium on Artifical Intelligence and Programming Languages (Rochester, New York, August 1977). SIGPLAN Notices 12, 8, SIGART Newsletter 64 (August 1977), 60-66.

Steele, Guy Lewis Jr., and Sussman, Gerald Jay "The Revised Report on SCHEME: A Dialect of LISP" MIT AI Memo 452 (Cambridge, January 1978).

Steele, Guy Lewis Jr., and Sussman, Gerald Jay "The Art of the Interpreter; or, The Modularity Complex (Parts Zero, One, and Two)" MIT AI Memo 453 (Cambridge, May 1978).

Steele, Guy Lewis Jr., and Sussman, Gerald Jay "Constraints" AI Memo 502. MIT AI Lab (Cambridge, November 1978). Invited paper, Proc. APL79 conference. ACM SIGPLAN STAPL APL Quote Quad 9, 4 (June 1979), 208-225.

Steele, Guy Lewis Jr., and Sussman, Gerald Jay "Design of LISP-Based Processors; or, SCHEME: A Dielectric LISP; or, Finite Memories Considered Harmful; or, LAMBDA: The Ultimate Opcode" AI memo 514. MIT AI Lab (Cambridge, March 1979).

Sussman, Gerald Jay, and Steele, Guy Lewis Jr "Constraints – A Language for Expressing Almost-Hierarchical Descriptions" Artificial Intelligence Journal 14 (1980), 1-39.

3.3 Others

The list tries to include the 'first version' of the story. (For example, there is the revised versions for MacLisp manuals.)

Church, Alonzo, "The Calculi of Lambda Conversion" Anunals of Mathematics Studies 6. Princeton University Press (Princeton, NJ 1941)

Naur, P., et. al. "Revised Report on the Algorithmic Language ALGOL 60", CACM vol.6 no.1 pp 1-17, Jan. 1963.

McCarthy, J., et al., "Lisp 1.5 Programmer's Manual," The MIT Press, (Cambridge, MA 1963).

Landin, Peter, "A correspondence between Algol 60 and Church's lambda notation: Part I." CACM vol.8 no. 2 pp89-101, 1965.

Dahl, O-J, et. al. "The Simula 67 Common Base Language", Norwegian Computing Centre, (Oslo, 1968).

Hewitt, Carl E., "PLANNER: A Language for Proving Theorems in Robots," Proc. of the First Int'l Joint Conference on AI, pp 295-301, (Washington DC 1969).

Sussman, Gerald Jay, Terry Wonograd, and Eugene Charniak, "Microplanner Reference Manual," Report AIM-203A, AI Lab, MIT (Cambridge, 1971)

Sussman, G.J., and McDermott, Drew V., "From PLANNER to CONNIVER – A Genetic Approach", Proc. the 1972 FJCC, pp 1171-1179, (Montvale, NJ, Aug. 1972)

Sussman, G.J., and McDermott, Drew V., "Why Conniving is Better than Planning" AI Memo 255A, MIT AI Lab (Cambridge MA April 1972)

Reynolds, John, "Definitional Interpreters for Higher Order Programming Languages", in Proc. ACM National conference pp 717-740 ACM, 1972.

McDermott, Drew V., and Sussman Gerald Jay, "The CONNIVER Reference Manual", AI Memo 295A, MIT AI Lab (Cambridge MA Jan. 1974).

Smith, Brian C., and Hewitt, Carl, "A PLASMA Primer", Working Paper 92, MIT AI Lab (Cambridge MA October 1975)

Galley, S.W. and Pfister, Greg., "The MDL Language", Programming Technology Division Document SYS.11.01. Project MAC, MIT (Cambridge, MA Nov. 1975) NOTE: This is Muddle!

Wand, Mitchell, and Friedman, Daniel P., "Compiling Lambda Expressions Using Continuations and Factorization" Technical Report 55, Indiana University, July 1977.

Hewitt, Carl E., "Viewing Control Structures as Patterns of Passing Messages," Artificial Intelligence, vol. 8, no.3, pp 323-364 1977.

Stoy, Joseph E. "Denotational Semantics: The Scott-Strachey Approach to Programming Theory" MIT Press, (Cambridge, Mass. 1977).

Wand, Mitchell, "Continuation-based program transformation strategies", J.ACM vol.27 no.1 pp 164-180, 1978.

Moon, David, "MacLisp Refence Manual version 0", Technical Report, MIT LCS, 1978.

Sussman, Gerald Jay; Holloway, Jack; Steele, Guy Lewis Jr.; and Bell, Alan. "Scheme-79 – LISP on a Chip." IEEE Computer 14, 7 (July 1981), 10-21.

Pitman, Kent. "The revised MacLisp Manual (Saturday evening edition)", Technical report 295, MIT LCS, 1983.

Milner, Robin, "A proposal for Standard ML," ACM sympo on Lisp and Functional Programming, pp 184-197, 1984.

Hudak, Paul, et.al. "Report on the Programming Langauge Haskell", Technical Report Yale University and Glasgow University (New Haven and Glasgow, Aug. 1991)

Kay, Alan C., "The Early History of Smalltalk", History of Programming Languages Conference Preprints, Sigplan Notices, vol.28 no.3 pp69-96, March 1993.

Steele, G.L., and Gabriel, Richard, "The Evolution of Lisp", History of Programming Languages Conference Preprints, Sigplan Notices, vol.28 no.3 pp231-270, March 1993.

Dahl, O-J, et. al. "The Simula 67 Common Base Language", Norwegian Computing Centre, (Oslo, 1968).

Hewitt, Carl E., "PLANNER: A Language for Proving Theorems in Robots," Proc. of the First Int'l Joint Conference on AI, pp 295-301, (Washington DC 1969).

Sussman, Gerald Jay, Terry Wonograd, and Eugene Charniak, "Microplanner Reference Manual," Report AIM-203A, AI Lab, MIT (Cambridge, 1971)

Sussman, G.J., and McDermott, Drew V., "From PLANNER to CONNIVER - A Genetic Approach", Proc. the 1972 FJCC, pp 1171-1179, (Montvale, NJ, Aug. 1972)

Sussman, G.J., and McDermott, Drew V., "Why Conniving is Better than Planning" AI Memo 255A, MIT AI Lab (Cambridge MA April 1972)

Reynolds, John, "Definitional Interpreters for Higher Order Programming Languages", in Proc. ACM National conference pp 717-740 ACM, 1972.

McDermott, Drew V., and Sussman Gerald Jay, "The CONNIVER Reference Manual", AI Memo 295A, MIT AI Lab (Cambridge MA Jan. 1974).

Smith, Brian C., and Hewitt, Carl, "A PLASMA Primer", Working Paper 92, MIT AI Lab (Cambridge MA October 1975)

Galley, S.W. and Pfister, Greg., "The MDL Language", Programming Technology Division Document SYS.11.01. Project MAC, MIT (Cambridge, MA Nov. 1975) NOTE: This is Muddle!

Wand, Mitchell, and Friedman, Daniel P., "Compiling Lambda Expressions Using Continuations and Factorization" Technical Report 55, Indiana University, July 1977.

Hewitt, Carl E., "Viewing Control Structures as Patterns of Passing Messages," Artificial Intelligence, vol. 8, no.3, pp 323-364 1977.

Stoy, Joseph E. "Denotational Semantics: The Scott-Strachey Approach to Programming Theory" MIT Press, (Cambridge, Mass. 1977).

Wand, Mitchell, "Continuation-based program transformation strategies", J.ACM vol.27 no.1 pp 164-180, 1978.

Moon, David, "MacLisp Refence Manual version 0", Technical Report, MIT LCS, 1978.

Sussman, Gerald Jay; Holloway, Jack; Steele, Guy Lewis Jr.; and Bell, Alan. "Scheme-79 - LISP on a Chip." IEEE Computer 14, 7 (July 1981), 10-21.

Pitman, Kent. "The revised MacLisp Manual (Saturday evening edition)", Technical report 295, MIT LCS, 1983.

Milner, Robin, "A proposal for Standard ML," ACM sympo on Lisp and Functional Programming, pp 184-197, 1984.

Hudak, Paul, et.al. "Report on the Programming Langauge Haskell", Technical Report Yale University and Glasgow University (New Haven and Glasgow, Aug. 1991)

Kay, Alan C., "The Early History of Smalltalk", History of Programming Languages Conference Preprints, Sigplan Notices, vol.28 no.3 pp69-96, March 1993.

Steele, G.L., and Gabriel, Richard, "The Evolution of Lisp", History of Programming Languages Conference Preprints, Sigplan Notices, vol.28 no.3 pp231-270, March 1993.