

学 号： 2018211991

密 级： 公开

合肥工业大学

Hefei University of Technology

# 本科毕业设计（论文）

UNDERGRADUATE THESIS



类 型：	设计
题 目：	基于 x86 平台的 Lisp 语言编译器的设计与实现
专业名称：	计算机科学与技术
入学年份：	2018 级
学生姓名：	余梓俊
指导教师：	安宁 教授
学院名称：	计算机与信息学院
完成时间：	2022 年 5 月



合 肥 工 业 大 学

本科毕业设计（论文）

基于 x86 平台的 Lisp 语言编译器的设计与实现

学生姓名：余梓俊

学生学号：2018211991

指导教师：安宁 教授

专业名称：计算机科学与技术

学院名称：计算机与信息学院

2022 年 5 月



A Dissertation Submitted for the Degree of Bachelor

**Design and Implementation of a Compiler  
For a Lisp Dialect Targeting x86**

By

Yu Zijun

Hefei University of Technology

Hefei, Anhui, P.R.China

May, 2022



## 毕业设计（论文）独创性声明

本人郑重声明：所呈交的毕业设计（论文）是本人在指导教师指导下进行独立研究工作所取得的成果。据我所知，除了文中特别加以标注和致谢的内容外，设计（论文）中不包含其他人已经发表或撰写过的研究成果，也不包含为获得 合肥工业大学 或其他教育机构的学位或证书而使用过的材料。对本文成果做出贡献的个人和集体，本人已在设计（论文）中作了明确的说明，并表示谢意。

毕业设计（论文）中表达的观点纯属作者本人观点，与合肥工业大学无关。

毕业设计（论文）作者签名：余梓俊 签字日期：2022年 5月 31日

## 毕业设计（论文）版权使用授权书

本学位论文作者完全了解 合肥工业大学 有关保留、使用毕业设计（论文）的规定，即：除保密期内的涉密设计（论文）外，学校有权保留并向国家有关部门或机构送交设计（论文）的复印件和电子光盘，允许设计（论文）被查阅或借阅。本人授权 合肥工业大学 可以将本毕业设计（论文）的全部或部分内容编入有关数据库，允许采用影印、缩印或扫描等复制手段保存、汇编毕业设计（论文）。

（保密的毕业设计（论文）在解密后适用本授权书）

学位论文作者签名：余梓俊 指导教师签名：姜宁

签名日期：2022年 5月 31日 签名日期：2022年 5月 31日





## 摘 要

函数式编程语言作为高度抽象的编程语言，与计算机的机器指令之间存在着巨大的差别。作为计算机科学领域一门重要的基础学科，学习构建编译器能让我们更深刻地理解软件与硬件之间的关联，指导我们编写正确高效的代码。

传统的编译器构造通常希望在一趟处理中完成尽可能多的工作，从而尽可能地缩短编译程序需要的时间。但这种方案缺乏灵活性，难以展示或修改编译过程中的特定步骤，不利于编译原理的研究学习。本文基于 Nanopass 思想，使用 Racket 语言，充分利用 Pattern Matching 和 Structural Recursion，实现了一个简单的静态类型 Lisp 方言的编译器，并详细描述了从源程序到语法树再到汇编代码所涉及的各种算法。

本系统能够从文件中读取源程序代码，解析生成语法树，历经十余趟变换后生成 x86-64 汇编程序，然后与一个使用 C 语言实现的运行时一同链接生成可执行文件。生成的可执行文件可以在类 Unix 系统下运行。本文还设计了一个展示每一趟编译过程的中间结果的图形界面程序，以此来可视化整个编译流程。

**关键词：**函数式编程语言，Lisp，编译器，Nanopass

# ABSTRACT

There is a huge gap between high-level programming languages such as functional programming languages and machine instructions. Compilers being a fundamental subject of the broad Computer Science, learning how to build one will let us have a deeper understanding of the relation between software and hardware, guiding us to write correct and efficient code in the future.

Traditional compiler constructions typically want to do as much work as possible in a single pass to minimize the time it takes to compile a program. However, this approach lacks flexibility and is difficult to display or modify a specific step in the compilation process, which is not conducive to the study of compilers. Based on the idea of Nanopass, this thesis implements a simple statically typed Lisp dialect compiler using the Racket programming language, making intensive use of Pattern Matching and Structural Recursion. Algorithms involved in all the way down to generate x86-64 assembly code from syntax trees are presented in detail.

This system can parse the source code in a file to an abstract syntax tree, and generate x86-64 assembly code after more than a dozen of passes of transformation. The assembly code is then linked together with a runtime system implemented in C, generating an executable that can run in Unix-like systems. This thesis also designed a graphical user interface that shows the intermediate results of each pass to visualize the entire compilation process.

**KEYWORDS:** Functional Programming Languages; Lisp; Compilers; Nanopass

# 目 录

<b>1 绪论</b>	<b>1</b>
1.1 课题背景	1
1.2 国内外研究现状	2
1.3 本文工作	2
1.4 本章小结	2
<b>2 目标平台介绍</b>	<b>4</b>
2.1 x86-64 程序示例	4
2.2 x86 中的函数调用	6
2.3 高效的尾调用	7
2.4 本章小结	8
<b>3 编译器前端设计实现</b>	<b>9</b>
3.1 抽象语法树和文法	9
3.2 构造和操纵语法树	10
3.3 源语言语法	11
3.4 本章小节	13
<b>4 编译器后端设计实现</b>	<b>14</b>
4.1 去糖化	14
4.2 标识符唯一化	15
4.3 赋值转换	15
4.4 处理函数	17
4.4.1 区分函数和局部变量	17
4.4.2 限制参数数量	17
4.4.3 生成闭包	17
4.4.4 赋值对闭包的影响	20
4.5 内存管理	20
4.5.1 垃圾回收算法	21
4.5.2 拷贝回收	22
4.5.3 向量在内存中的表示	24
4.5.4 展开向量定义	25

4.6	原子化操作数	26
4.7	显示化控制流	27
4.8	生成汇编	28
4.9	寄存器分配	31
4.9.1	活变量分析	31
4.9.2	使用图着色算法进行分配	33
4.10	整理指令	33
4.11	本章小结	34
<b>5</b>	<b>编译过程展示系统</b>	<b>35</b>
5.1	展示系统设计与实现	35
5.2	展示系统运行效果	35
5.3	本章小结	37
<b>6</b>	<b>总结和展望</b>	<b>38</b>
	<b>参考文献</b>	<b>39</b>
	<b>致谢</b>	<b>41</b>

# 插图清单

图 2.1	x86-64 程序示例 . . . . .	4
图 2.2	栈帧的内存布局 . . . . .	5
图 2.3	尾调用示例 . . . . .	7
图 3.1	具体语法示例 . . . . .	10
图 3.2	抽象语法示例 . . . . .	10
图 3.3	语法树示例 . . . . .	10
图 3.4	源语言的完整语法 . . . . .	12
图 3.5	源程序示例 . . . . .	12
图 4.1	闭包示意图 . . . . .	19
图 4.2	闭包转换示例 . . . . .	19
图 4.3	双空间拷贝回收示意图 . . . . .	22
图 4.4	拷贝回收的详细过程 . . . . .	23
图 4.5	向量在内存中的表示 . . . . .	25
图 4.6	展开向量定义 . . . . .	25
图 4.7	显示化控制流的实现 . . . . .	28
图 4.8	显示化控制流示例 . . . . .	29
图 5.1	展示系统输入源程序 . . . . .	35
图 5.2	并列展示编译过程的中间结果 . . . . .	36
图 5.3	调用 graphviz 绘制干涉图 . . . . .	36
图 5.4	示例程序的运行结果 . . . . .	37



# 1 绪论

## 1.1 课题背景

当今世界计算机技术飞速发展，各种软件让人眼花缭乱，这得意于程序设计语言的发展。常用的程序设计语言如 C 语言、C++ 语言、Java 语言等，用这些高级语言编写程序极大提高了软件开发人员的效率。但要使这些代码能真正被计算机识别并执行，需要将它们转化成计算机指令，完成这一工作的便是编译器。编译器将由高级语言编写的程序翻译成二进制代码或其他目标语言，并在特定平台上运行，因此程序语言的发展，又很大程度上依赖编译技术的发展。

自上个世纪 50 年代以来，编译器的相关研究一直是计算机科学领域的一个活跃主题。从目标机器的角度来说，目标机器处理器从单核到多核，其体系结构在不断的发生变化；从程序语言的角度来说，高级程序语言在不断的丰富及扩展。

1954 年，John Backus 在 IBM 发明了 FORTRAN。它是第一个广泛使用的具有功能实现的高级通用编程语言，而不仅仅是纸上的设计。1957 年，他领导的研究小组完成了 FORTRAN 编译器，该编译器被普遍认为是引入了第一个明确完整的编译器。1958 年，John McCarthy 发明了 Lisp<sup>[1]</sup>，这是世界上第一门动态类型的函数式编程语言。六十年代的 Simula 是第一门支持面向对象编程的语言，其后继 Smalltalk，一门纯面向对象的语言，在七十年代被发明。1969 年至 1973 年，C 被发明，用作 Unix 操作系统的系统编程语言，流行至今。Prolog，第一门逻辑编程语言，在 1972 年被发明。1978 年，ML 在 Lisp 之上建立了一个多态类型系统，开创了静态类型的函数式编程语言。他们都产生了大量的后代，绝大多数现代编程语言都可以追溯到这些语言。

作为最早的编程语言之一，Lisp 开创了计算机科学的许多理念，比如树形数据结构，自动内存管理，动态类型，高阶函数，递归，自举，REPL 等等。但函数式编程语言作为高度抽象的高级编程语言，与现代计算机的指令之间存在巨大的差别。而传统的编译器课程只重点讲授教编译器的某几个阶段，如解析、语义分析和寄存器分配等。这种方法的问题是很难理解整个编译器是如何结合在一起的，为什么每个阶段是那样设计的，以及那些更具表达力的语言成分究竟是如何实现的。本文实现了一个简单的 Lisp 方言，将其编译到 x86 汇编语言，以此来学习并探索编译技术。

## 1.2 国内外研究现状

Scheme 是一门极简主义 Lisp 方言，采用了词法作用域，实现了尾递归优化，支持 first-class continuations 等。1978 年，Guy Steele<sup>[2]</sup> 实现了第一个 Scheme 语言编译器，该编译器也是第一个使用 CPS 作为中间表示的编译器。1984 年，Kent Dybvig 创造了第一个商用 Scheme 编译器 Chez Scheme<sup>[3]</sup>，它的编译速度很快，生成的目标代码也非常高效。1992 年，Andrew Appel<sup>[4]</sup> 在其著作中描述了 Standard ML of New Jersey 的实现。1994 年，Christian Queinnec<sup>[5]</sup> 在他的书中全面地介绍了 Lisp 语言家族的语义和实现，给出了 11 个解释器和 2 个目标语言分别为字节码和 C 语言的编译器。2004 年，Kent Dybvig 等人<sup>[6]</sup> 在印地安纳大学数年的编译器教学中演化提出了 nanopass 的思想。2012 年，Andrew Keep<sup>[7]</sup> 在其论文中描述了 nanopass 框架对商用编译器 Chez Scheme 的改造，论证了该框架的效率并不显著低于传统编译器结构。2021 年，Jeremy Siek<sup>[8]</sup> 详细总结讲解了编译一个简单的函数式语言到机器语言的完整过程。

## 1.3 本文工作

本文的工作主要设计并实现了一个简单的 Lisp 方言，将其编译为 x86-64 汇编语言，二者具体来说分别是 Typed Racket 和 x86-64 的严格子集。语言支持的功能主要包括可变变量与循环，向量（元组）和垃圾回收，一等函数和闭包。

编译器前端利用了 Lisp 类语言方便的读取符号的特性和 Racket 的结构体来生成语法树，这一步在“编译器前端设计实现”一章中说明。编译器后端的主要流程包括去糖化，变量唯一化，赋值转换，函数转换为闭包，内存管理和垃圾回收，原子化操作数，显示化控制流，生成汇编，寄存器分配，以及最后的指令整理。具体算法流程在“编译器后端设计实现”一章中给出。

本文还设计实现了一个简单的图形界面，允许我们在输入框中输入一段代码，并排展示出每一趟的编译结果，输出使用图着色算法时生成的变量相关图以及最终的可执行文件。这一部分在“编译过程展示系统”一章中详细描述。

## 1.4 本章小结

作为最早的编程语言之一，Lisp 对后续的语言设计和发展产生了极大的影响，整个 Lisp 语言家族也在不断完善和发展。在“课题背景”一节中，本章首先简要介绍了早期几个重要的编程语言的发展历程，Lisp 语言的地位和特性，说明了实现



一个 Lisp 语言的现实意义与背景。“国内外研究现状”一节简要介绍了 Lisp 语言及其方言 Scheme 的发展历程，列出了关于其编译器构造的几篇重要文献。然后指出了本文的编译器实现思想，即 Nanopass 相关的文献。最后，“本文工作”一节中简要叙述了编译器从源语言到目标语言的各个环节，并给出了对应的章节安排。

## 2 目标平台介绍

### 2.1 x86-64 程序示例

本文使用 GNU 汇编器要求的 AT&T x86-64 汇编语法。程序以 `main` 标签开始，后面跟着一系列指令。`globl` 指令表明 `main` 过程是外部可见的，这是操作系统可以调用它的必要条件。x86 程序存储在计算机的内存中。就本文的目的而言，计算机的内存是 64 位地址到 64 位值的映射。计算机在 `rip` 寄存器中存储一个程序计数器 (PC)，它指向下一条要执行的指令的地址。对于大多数指令，程序计数器在指令执行后递增，因此它指向内存中的下一条指令。大多数 x86 指令有两个操作数，每个操作数要么是一个整型常量 (称为立即数)，要么是一个寄存器，要么是一个内存位置。

寄存器是一种特殊的变量，我们能使用的是通用寄存器，共有 16 个，每一个都有一个 64 位的值。我们使用 `%` 符号后跟寄存器名，比如 `%rax`，来指代某一个寄存器。

立即数使用语法 `$n` 来指明，其中 `n` 是整数。访问内存使用语法 `n(%r)` 来指明，它的意思是拿到存储在寄存器 `r` 中的地址，然后给地址加 `n` 个字节。

图2.1是一个简单的 x86-64 程序。该程序计算式子  $(+ 52 - 10)$  的值，并将结果作为整个程序的返回值。

```
start:
    movq    $10, -8(%rbp)
    negq    -8(%rbp)
    movq    -8(%rbp), %rax
    addq    $52, %rax
    jmp     conclusion
    .globl main
main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    jmp     start
conclusion:
    addq    $16, %rsp
    popq    %rbp
    retq
```

图 2.1 x86-64 程序示例

这个程序使用一个称为过程调用栈 (简称栈) 的内存区域。堆栈由每个过程调用的独立帧组成。单个帧的内存布局如图2.2所示。寄存器 `rsp` 被称为堆栈指针，它

位置	内容
(8%rbp)	return address
(0%rbp)	old rbp
(-8%rbp)	variable 1
(-16%rbp)	variable 2
...	...
(0%rsp)	variable <i>n</i>

图 2.2 栈帧的内存布局

指向堆栈顶部的项。堆栈在内存中向下增长，通过减去堆栈指针来增加堆栈的大小。在过程调用的上下文中，返回地址是调用方调用指令之后的指令。函数调用指令 `callq` 在跳转到过程之前将返回地址压入堆栈。寄存器 `rbp` 是基指针，用于访问存储在当前过程调用帧中的变量。调用者的基指针在返回地址之后被压入堆栈，然后基指针被设置为旧基指针的位置。变量 1 存储在地址 `-8(%rbp)`，变量 2 存储在地址 `-16(%rbp)`，以此类推。

回到这个程序，首先，操作系统发出一个 `callq main` 指令，将其返回地址推送到堆栈上，然后跳转到 `main`。在 x86-64 中，堆栈指针 `rsp` 在执行任何 `callq` 指令之前必须整除 16 个字节。因此当控制到达 `main` 时，`rsp` 偏离对齐 8 个字节（因为 `callq` 压入返回地址）。

前三条指令是一个程序典型的准备工作。指令 `pushq %rbp` 将调用者的基指针保存到堆栈上，并从堆栈指针中减去 8。第二个指令 `movq %rsp, %rbp` 读取 `rsp` 中的内容，赋值给 `rbp`。这条指令改变了基指针，使其指向旧基指针的位置。指令 `subq $16, %rsp` 读取 `rsp` 中的内容，减去 16，再重新赋值给 `rsp`。这条指令将堆栈指针向下移动，以便为存储变量留出足够的空间。

虽然这个程序只需要在栈上存储一个变量（8 个字节），但是我们还是需要减去 16，这样 `rsp` 是 16 字节对齐的，就可以调用其他函数。准备工作的最后一条指令是 `jmp start`，它使程序跳转到由 Racket 表达式 `(+ 52 (- 10))` 生成的指令，也就是 `start` 标签下的指令。

`start` 标签下的第一条指令是 `movq $10, -8(%rbp)`，它将 10 存储在图 2.2 中 `variable 1` 的位置。指令 `negq -8(%rbp)` 将值更改为 -10。下一条指令将 `%rax` 中值更改为 -10。最后，`addq $52, %rax` 将 52 加到 `rax` 上，将其值更新为 42。我们总是约定将程序的返回值放在寄存器 `rax` 中。

`conclusion` 标签下的三个指令是一个程序典型的结束工作。前两条指令将 `rsp` 和 `rbp` 寄存器恢复到过程开始时的状态。指令 `addq $16, %rsp` 将栈指针移回

指向旧的基指针。然后 `popq %rbp` 把旧基指针返回给 `rbp`，并在堆栈指针上加 8。最后一条指令 `retq` 返回到调用它的过程，并在堆栈指针上加 8。

## 2.2 x86 中的函数调用

x86 提供标签，以便可以引用指令的位置，这是跳转指令所需要的。标签也可以用来标记函数指令的开头。我们还可以通过使用 `leaq` 指令和 PC 相对寻址来获得一个标签的地址。例如，下面这条指令把 `add` 函数的地址放入 `rbx` 寄存器中：

```
leaq add1(%rip), %rbx
```

指令指针寄存器 `rip`，也就是程序计数器始终指向要执行的下一条指令。当与一个标签结合时，如 `add1(%rip)`，链接器计算 `add1` 标签的地址与 `rip` 此刻的位置之间的距离 `d`，然后将 `add1(%rip)` 更改为 `d(%rip)`，通过这种方式在运行时计算出 `add1` 标签的地址。

由于支持一等函数，也就是将函数当作普通的变量，函数可能会被赋值给其他变量。也就是说，某个变量中存放的可能是一个函数地址，调用这个变量就相当于调用函数，亦即间接函数调用。对应的 x86 指令是接受一个寄存器名的 `callq*` 指令：

```
callq *%rbx
```

`callq` 指令为实现函数提供了部分支持：它将返回地址压入堆栈，然后跳转到目标函数。但是 `callq` 不处理：(1) 参数传递；(2) 把帧压入过程调用栈以及取出或访问它们；(3) 确定不同函数如何共享寄存器。

函数调用需要两段代码之间的协调，这两段代码可能由不同的程序员编写或由不同的编译器生成。这里我们遵循 Linux 和 MacOS 上的 GNU C 编译器使用的 System V 调用约定<sup>[9]</sup>。调用约定包括关于函数如何共享寄存器使用的规则。特别是，调用方负责在函数调用之前释放一些寄存器，供被调用方使用。这些被称为“调用者保存寄存器”：

```
rax rcx rdx rsi rdi r8 r9 r10 r11
```

另一方面，被调用方负责保留“被调用者保存寄存器”，即：

```
rsp rbp rbx r12 r13 r14 r15
```

我们可以从调用方视角和被调用方视角两个角度来考虑这个调用约定：

- 调用方应该假定所有调用者寄存器都可以被被调用方用任意值重写。另一方面，调用者可以安全地假设所有调用者保存的寄存器在调用后包含与调用前

相同的值。

- 被调用方可以自由地使用调用者保存寄存器。如果被调用方希望使用被调用者保存寄存器，被调用方必须在返回到调用方之前将原始值放回寄存器中。这可以通过在准备工作中将值保存到调用栈中，并在函数的结束工作中恢复来实现。

在 x86 中，寄存器还用于向函数传递参数和保存返回值。函数的前六个参数按以下顺序，在以下六个寄存器中传递。

```
rdi rsi rdx rcx r8 r9
```

多余的参数则保存在调用方的栈帧上。寄存器 `rax` 则用于函数的返回值。

## 2.3 高效的尾调用

通常，程序所使用的栈空间的大小是由最长的嵌套函数调用链决定的。也就是说，如果函数  $f_1$  调用  $f_2$ ， $f_2$  调用  $f_3$ ， $\dots$ ， $f_{n-1}$  调用  $f_n$ ，那么堆栈空间的大小就是  $O(n)$ 。在递归或相互递归函数的情况下，深度  $n$  可以变得很大。但是，如果函数调用表达式位于尾位置（尾位置的定义见4.7节），我们把它称为尾调用，它就不会导致调用栈的深度增加。例如，下面的求和函数中的递归调用就是尾调用。

```
(define (sum n result)
  (if (eq? n 0)
      result
      (sum (- n 1) (+ result n))))
(sum 5 0)
```

图 2.3 尾调用示例

尾调用时不再需要调用方的帧，因此可以在进行尾调用之前弹出调用方的帧。通过这种方法，只进行尾调用的递归函数将只使用  $O(1)$  的栈空间。函数式语言严重依赖递归函数，所以要尾调用必须被优化。

弹出帧的指令也就是前面说的函数的结束工作。因此我们还需要在每个尾调用之前插入这些指令。在将来尾调用完成时，它就不再需要也不能返回当前函数，而是直接返回到调用当前函数的那个函数。因此尾调用不再使用 `callq` 指令，而是直接使用 `jmp` 指令进行跳转。与间接函数调用类似，x86 使用前面带一个星号的寄存器跳转指令来表示间接跳转：

```
jmp *%rax
```

这里我们统一使用 `rax`，因为跳转指令前面插入的那些结束工作指令会把一些数据恢复回到寄存器里，这些数据将来还要被使用，不能覆盖掉它们。而 `rax` 此时

总是空闲的。

另外还要注意的一点是，前文说过函数调用的前六个参数分别使用约定的六个寄存器，多余的参数则使用过程调用栈来传递。加入尾递归优化之后，用栈来传参就很困难了。因此在后面的编译过程中，我们会把第六个及其后面的参数放入一个向量中，把这个向量作为第六个参数，以此来解决这个问题。

## 2.4 本章小结

本章简要介绍了 x86-64 汇编语言的相关知识。首先通过一个简单的示例程序来讲解 x86-64 汇编语言的语法和过程调用栈；然后讲解了 x86 对函数调用的支持，并介绍了本文选择的 System V 调用约定和寄存器分配策略；最后一节介绍了尾调用的概念，为什么尾调用可以被优化以及如何在汇编层面实现尾调用的优化。

### 3 编译器前端设计实现

程序通常由程序员以文本形式输入，即字符序列。程序作为文本的表示称为具体语法。我们用具体语法简明地写下和讨论程序的语法。在编译器中，我们使用抽象语法树来表示程序，因为编译器可以很方便高效地操作抽象语法树。翻译具体语法到抽象语法的过程称作解析。本文不过多谈及解析的理论和实现，一方面是因为现阶段解析已经有了相对固定化的套路和工具，研究也已经趋于完善。另一方面由于本文选择了 Racket 作为源语言和实现语言，我们可以很方便地从输入的本文序列中构造出结构体，而 Racket 的结构体也是本文选择的用以表示抽象语法树的数据结构。

在本章中，我们通过一个简单的例子来说明如何表示语法，如何使用 Racket 语言从文本中构造并操纵语法树，然后给出本文实现的语言的完整语法。

#### 3.1 抽象语法树和文法

在计算机科学中，抽象语法树，或简称语法树，是源代码语法结构的一种抽象表示。它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构。之所以说语法是“抽象”的，是因为这里的语法并不会表示出真实语法中出现的每个细节。比如，嵌套括号被隐含在树的结构中，并没有以节点的形式呈现；而类似于 if-condition-then 这样的条件跳转语句，可以使用带有三个分支的节点来表示。

编程语言可以被看作是合法程序的集合，这一集合通常是无限大的，因为我们总是可以写出更加庞大复杂的程序。因此，我们无法简单地通过把所有的合法程序列出来来描述一门语言。通常，我们使用文法——一系列规则，来构造程序。文法通常用来描述具体语法，但它也可以被用来描述抽象语法。本文使用巴科斯-诺尔范式<sup>[10-11]</sup>的变体来描述我们的语言。图3.1中的几条规则给出了本文实现的语言的一个子集的描述，该语言仅支持包含加减法的整数运算，但它允许任意复杂的嵌套表达式。其中 read 函数读取用户在键盘上输入的一串数字，返回一个整数。

下面这行代码是这个示例语言的一个合法的程序：

```
(+ (read) (- 8))
```

该程序对应的语法树如图3.3。

```

type ::= Integer
exp  ::= int | (read) | (- exp) | (+ exp exp)
L    ::= exp

```

图 3.1 具体语法示例

```

type ::= Integer
exp  ::= (Int int) | (Prim 'read ()) | (Prim '- (exp))
      | (Prim '+ (exp exp))
LInt ::= (Program '() exp)

```

图 3.2 抽象语法示例

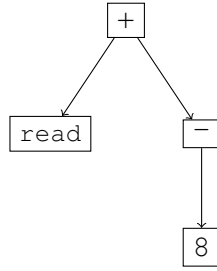


图 3.3 语法树示例

### 3.2 构造和操纵语法树

我们使用 Racket 的结构体来表示语法树，用抽象语法来描述各个结构体的定义，用 Racket 的模式匹配来构造和操纵语法树。

图3.1对应的抽象语法的描述如图3.2所示。其中 Int, Prim 等即为对应的 Racket 结构体的构造函数。这些结构体可以非常简单地被定义出来：

```

(struct Int (value))
(struct Prim (op args))

```

构造结构体也非常简单，下面这行代码对应的即为具体语法为 (+ (read) (- 8)) 的程序：

```

(Prim '+
  (list (Prim 'read ())
        (Prim '- (list (Int 8)))))

```

而对结构体使用模式匹配，我们可以很方便地判断结构体的类型以及访问结构体的成员，也就是语法树的子树。例如，下面这段代码将对变量 ast 进行模式匹配，如果变量是一个包含两个（可能复杂的）操作数的语法树，就打印操作符。同时，这段代码还会将两个操作数，也就是两个子树，分别绑定到变量 child1 和 child2 上去。例如，如果 ast 的值是 (+ (read) (- 8))，则 child1 和 child2 的值将分别被赋值为 (read) 和 (- 8)，并且代码将打印出 +。

```

(match ast
  [(Prim op (list child1 child2))
   (print op)])

```



一个模式匹配当然也可能包含多个子句，这些子句将会从上到下依次被判断是否匹配。例如，下面这个 `leaf?` 函数接受一个名为 `ast` 的参数，判断它是否是叶子节点：如果是一个数字或者 `read` 函数，则为叶子节点；如果是加运算或者取负运算，则不是叶子节点。

```
(define (leaf? ast)
  (match ast
    [(Int n) true]
    [(Prim 'read '()) true]
    [(Prim '- (list e1)) false]
    [(Prim '+ (list e1 e2)) false]))
```

以下三行代码的返回值将分别为 `true`, `false`, `true`:

```
(leaf (Prim 'read '()))
(leaf (Prim '- (list (Int 8))))
(leaf (Int 8))
```

接下来我们看一下如何使用模式匹配，配合 **Racket** 的 `read` 函数来构造语法树。事实上，**Racket** 提供的各种 `reader` 本身就已经是一个递归下降解析器。`read` 函数接收一个输入端口，返回一个包含数值和符号的 `list`。例如，假设一个源代码文件中存放了上面的例子中的代码：`(+ (read) (- 8))`。对这个文件调用 `read` 函数会返回这样的一个 `list`：`('+ ('read) ('- 8))`。单引号表示这是一个符号。模式匹配可以对 `list` 和符号进行匹配。

```
(define (parse e)
  (match e
    [(? symbol?) (Var e)]
    [(? fixnum?) (Int e)]
    [ `(,op ,es ...)
      (Prim op (map parse es))]
    ...
  ))
```

上面的 `parse` 函数就可以从 `list` 中构造出我们想要的语法树。如果它接收的参数是符号，则返回一个变量节点；如果是整数，则返回一个整数节点；如果是一个运算，则对操作数列表递归调用 `parse` 函数，将返回的节点作为运算结构体的成员，也就是作为该节点的各个子树。如此我们就利用 **Racket** 的 `reader` 和模式匹配实现了编译器的前端部分。

### 3.3 源语言语法

在语言特性的选择上，我们主要考虑让它们能尽可能地体现不同的编译原理的知识面。对于基本数据类型，本文实现了 64 位整数及其加、减、乘、整除、模运算，布尔类型及其与、或运算；本文没有实现浮点数与浮点运算，因为它们只是

```

type ::= Integer | Boolean | Void | (Vector type ...)
      | (type ... -> type)
bool ::= #t | #f
op    ::= + | - | * | quotient | remainder | and | or | not
cmp   ::= eq? | < | <= | > | >=
exp   ::= int | bool | var | (read) | (void)
      | (- exp) | (op exp exp) | (cmp exp exp)
      | (let ([var exp]) exp) | (if exp exp exp)
      | (set! var exp) | (begin exp ...) | (while exp exp)
      | (vector exp ...) | (vector-length exp)
      | (vector-ref exp int) | (vector-set! exp int exp)
      | (procedure-arity exp)
      | (lambda: ([var:type] ...) : type exp)
def   ::= (define (var [var:type] ...) : type exp)
L     ::= def... exp...

```

图 3.4 源语言的完整语法

```

(define (even? [x : Integer]) : Boolean
  (if (eq? x 0)
      #t
      (odd? (- x 1))))

(define (odd? [x : Integer]) : Boolean
  (if (eq? x 0)
      #f
      (even? (- x 1))))

(if (even? (read)) 1 0)

```

图 3.5 源程序示例

单纯地覆盖了更多的 x86-64 指令，并不会增加语言的表现力；本文没有采用纯函数式的方式，而是支持了可变变量和循环，因为这会迫使我们在进行数据流分析时考虑循环数据流；对于复合数据结构，本文仅实现了向量（元组），向量是一个存放在堆上的、定长的、允许存放不同类型元素的结构，这足以让我们考虑堆上的内存数据和垃圾回收；最后是函数，本文实现了一等函数，利用向量实现了闭包，并且在翻译过程中正确地处理了尾递归。

在类型系统方面，本文没有实现类型推导，但只有函数的形式参数以及返回值需要显式声明类型，其余变量（即局部变量）定义时并不需要声明类型。图3.5是一个合法的程序。该程序从键盘读取一个（大于0的）整数，如果是偶数则返回1，奇数则返回0。

图3.4展示了源语言完整的具体语法。

### 3.4 本章小节

本章介绍了编译器前端的设计与实现。首先用一个示例语言说明了本文描述具体语法的方法，即巴克斯-诺尔范式，以及如何用该范式描述抽象语法及其与语法树的对应关系。然后讲解了如何使用 **Racket** 的结构体建立抽象语法对应的语法树，也就是编译器的前端部分，以及如何使用模式匹配来操纵它。最后用巴克斯-诺尔范式列出了源语言的完整语法，展示了源语言的一个示例程序。

## 4 编译器后端设计实现

### 4.1 去糖化

语法糖是由英国计算机科学家彼得·兰丁发明的一个术语，指计算机语言中添加的某种语法，这种语法对语言的功能没有影响，但是更方便程序员使用。语法糖让程序更加简洁，有更高的可读性。

一个实用的通用编程语言会提供非常多的功能，但从语言实现的角度考虑，在编译之前将语言的某些部分转换成使用其他更核心的部分来实现，能让语言维持一个比较小的核心，以便更容易维护同时也更具灵活性。我们把该步骤称为去糖化，对应的编译过程（pass）称为 shrink。

本文实现的语言中，布尔与、或运算并不属于核心语言，它们会被编译成 if 语句。

$$\begin{array}{ccc} (\text{and } e_1 \ e_2) & \Rightarrow & (\text{if } e_1 \ e_2 \ \#f) \\ (\text{or } e_1 \ e_2) & & (\text{if } e_1 \ \#t \ e_2) \end{array}$$

这个转换维持了与、或运算的短路特性。该转换的代码实现如下：

```
(define (shrink-exp e)
  (match e
    [(or (Int _) (Var _) (Bool _) (Void))
     e]
    [(Let x rhs body)
     (Let x (shrink-exp rhs) (shrink-exp body))]
    [(Prim 'and (list e1 e2))
     (If (shrink-exp e1)
         (shrink-exp e2)
         (Bool #f))]
    [(Prim 'or (list e1 e2))
     (If (shrink-exp e1)
         (Bool #t)
         (shrink-exp e2))]
    [(Prim op es) (Prim op (map shrink-exp es))]
    ...)
```

这里处理程序，或者说操作语法树的方法是使用自然递归，也叫结构化递归，即在结构的子结构上进行递归。例如，在转换与、或运算时，先对两个运算数进行递归转换，然后把转换的结果放入 if 表达式中。这样我们就可以处理任意复杂的与、或运算。后面的绝大部分编译过程也是如此。

## 4.2 标识符唯一化

由于 x86-64 以及后面使用的线性的中间代码不具备作用域，也为了方便后续对程序进行分析，我们需要确保所有的标识符的名字是唯一的，包括变量名和参数名，就像下面这样。

以 let 语句为例，标识符唯一化将进行如下的转换：

```
(let ([x (let ([x 4])
              (+ x 1))])
      (+ x 2))    ⇒    (let ([x2 (let ([x1 4])
                                   (+ x1 1))])
                          (+ x2 2))
```

这个编译过程的实现如下：

```
(define ((uniquify-exp env) e)
  (define recur (uniquify-exp env))
  (match e
    [(or (Int _) (Bool _) (Void)) e]
    [(Var x) (Var (dict-ref env x))]
    [(Let x e body)
     (let ([new-x (gensym x)])
       (Let new-x
            (recur e)
            ((uniquify-exp
              (dict-set env x new-x))
             body)))]
    [(If e1 e2 e3)
     (If (recur e1) (recur e2) (recur e3))]
    [(SetBang x e)
     (SetBang (dict-ref env x) (recur e))]
    ...)
```

这时处理语法树的递归函数上除了当前结点这一参数外，还多了一个 `env` 参数，这个参数是一个字典，每当绑定一个变量时，我们使用 `gensym` 函数生成一个全局唯一的符号作为新变量名，并在字典中加入新的键值对，键为变量名，值为新变量名。使用变量时，也就是访问变量或对变量赋值时，我们查找这个字典，对变量名进行替换。函数的参数也需要进行相同的处理。在遇到函数定义时把原参数名和新生成的唯一参数名作为键值对加入字典，在函数体内使用该字典进行标识符替换。

其中，`gensym` 可以接受一个字符串或符号参数，生成的符号将以这个参数作为前缀。例如 `(gensym 'print)` 将会生成类似 `print46352` 的符号。这个功能可以帮助我们生成更易读、易调试的中间代码。

## 4.3 赋值转换

考虑这样的例子：

```
(let ([x 2])
  (+ x
    (begin (set! x 40) x)))
```

其中 `begin` 语句接受若干个表达式，依次执行，然后将最后一个表达式的结果返回。由于在我们的语言中，操作数先依次从左往右被求值，然后执行运算，上面的代码应该得到 42。

因为汇编语言的运算符的操作数不支持复杂表达式，而仅仅支持立即数或者从一个寄存器或内存地址中获取操作数。在后面有一趟被称作原子化操作数的编译过程，会把所有复杂的操作数提取出来，求值后赋值给一个临时变量，用这个临时变量替换原来的复杂操作数。像下面这样：

```
(+ (+ x 1) 2)      ⇒      (let ([tmp (+ x 1)])
                           (+ tmp 2))
```

但是对于上面包含赋值的例子，如果我们只是进行简单的提取替换，会得到下面的结果：

```
(let ([x 2])
  (let ([tmp (begin (set! x 40) x)])
    (+ x tmp)))
```

由于对 `x` 重新赋值的语句原来处于操作数的位置，它会被提取到加法运算的前面。显然，转换后的程序会得到错误的结果：80。

解决的方法是对于所有涉及到被重新赋值的变量，我们把对它们的读取包裹在一个复杂操作 `get` 的后面。当然，在最终翻译成汇编代码时，通过 `get` 语句访问变量和直接访问变量并没有区别。上面的程序将被转换成这样：

```
(let ([x 2])
  (+ (get! x)
    (begin (set! x 40)
           (get! x))))
```

这样一来，对这些变量的读取也会被提取到运算之前。只要原子化操作数这一过程按正确的顺序提取复杂操作数，我们就能得到正确的程序：

```
(let ([x 2])
  (let ([tmp1 (get! x)])
    (let ([tmp2 (begin (set! x 40)
                      (get! x))])
      (+ tmp1 tmp2))))
```

赋值还会导致在编译闭包时出现另一个问题，我们将在后文描述闭包的实现时指出。

## 4.4 处理函数

### 4.4.1 区分函数和局部变量

编译器前端生成的语法树，在没有进行语义分析之前，没有也无法区分函数名和变量名。但是对于访问函数，无论是直接调用还是将其赋值给其他变量，我们需要将其翻译成 `leaq` 指令，这和访问变量是不同的。因此我们需要区分开来函数名和变量名。

由于我们已经进行了标识符唯一化，这个工作就变得非常简单。就像处理可变变量那样，我们引入 `Funref` 结构体，把所有出现的使用全局定义的函数名的地方，全部换成 `Funref` 语句即可。后面翻译为 x86 时再将其翻译为 `leaq` 指令。例如：

<pre>(define (f) : Integer   ...) (let ([g f])   g)</pre>	$\Rightarrow$	<pre>(define (f) : Integer   ...) (let ([g (Funref f)])   g)</pre>
---	---------------	--

### 4.4.2 限制参数数量

2.3节中提到，为了优化尾部调用，我们需要利用向量来把参数的数量限制在不超过六个。这一编译过程非常简单。对于每个函数定义，如果它超过六个参数我们用一个新的参数名来替换第六和后面的参数，新参数的类型是一个向量，里面的元素类型分别对应那些多余的参数类型。然后在函数体的开头加上一系列 `let` 绑定，把参数值从这个向量中拿出来，分别绑定到原来的参数名上。而对于函数调用，就只需要把多余的实参放进一个向量中作为第六个参数即可。

### 4.4.3 生成闭包

如果一个变量出现在 `e` 中，但在 `e` 中没有定义，那么就称该变量在表达式 `e` 中是“自由变量”。

例如，在下面的表达式中，`x` 和 `y` 为自由变量。

```
(lambda: ([z : Integer]) : Integer
  (+ x y z))
```

而闭包则是引用了自由变量的函数，被引用的自由变量和函数一同存在，即使已经离开了自由变量的环境也不会被释放或者删除，在闭包中可以继续使用这个自由变量。

例如：

```

(define (f [x : Integer]) : (Integer -> Integer)
  (let ([y 4])
    (lambda: ([z : Integer]) : Integer
      (+ x y z))))

(let ([g (f 5)])
  (let ([h (f 3)])
    (+ (g 11) (h 15))))

```

f 接受一个参数 x，在其内部创建一个值为 4 的局部变量 y，然后返回一个匿名函数。这个匿名函数接受一个参数 z，返回 x+y+z 的值。对于这个 lambda 表达式，x 和 y 是自由变量。

接下来是两次对 f 的调用，参数 x 分别被传入了 3 和 5。从 f 返回的匿名函数被绑定到变量 g 和 h。尽管这两个函数是由同一个 lambda 创建的，但它们实际上是不同的函数，因为 g 和 h 中 x 的值显然不同。然后，调用 (g 11) 会执行 (+ 5 4 11)，调用 (h 15) 会执行 (+ 3 4 15)，二者相加最终得到 42。显然，在调用 g 和 h 时，我们需要某种方法来读取到正确的 x 和 y 的值。

我们的解决方法是在每次调用函数时，把自由变量的值与函数指针放到一个向量中，并把这个向量作为额外的参数传入进去。这个技术称作扁平闭包<sup>[12]</sup>，通常直接就称为闭包。我们看一下在上面的例子中，闭包是如何工作的。

首先，匿名函数也是函数，编译成汇编代码后它们同样是从一个标签开始的一段指令。因此，所有匿名函数都会被提出来变成全局函数，并赋予它们新的函数名。

程序首先调用函数 f，它为 lambda 创建一个闭包。闭包其实就是一个向量（元组），它的第一个元素是一个指针，指向我们将来为 lambda 生成的顶层函数，第二个元素是 x 的值，即 5，第三个元素是 4，即 y 的值。闭包中不包含 z，因为 z 不是 lambda 的自由变量。创建结束是第一步。闭包从 f 返回并绑定到 g，如图 4.1 所示。对 f 的第二次调用创建另一个闭包，这一次闭包的第二个元素是 3。这个闭包也从 f 返回，但绑定到 h，如图 4.1 所示。

接下来考虑 (g 11)。要调用闭包，我们拿到闭包第一个元素中的函数指针并调用它，传入闭包本身，然后传入常规参数，也就是 11。

最后，我们还要为 lambda 生成顶层函数。这个顶层函数会多出一个参数，用来接受闭包。然后对于每个自由变量，我们要在函数的开头从闭包中取出这些自由变量的值，绑定到这些自由变量上。这一步称为闭包转换。

图 4.2 展示了这个例子的转换结果。注意，虽然 f 函数本身就是全局定义的函数，它的函数体里没有任何自由变量，我们还是给它添加了一个额外的参数用以



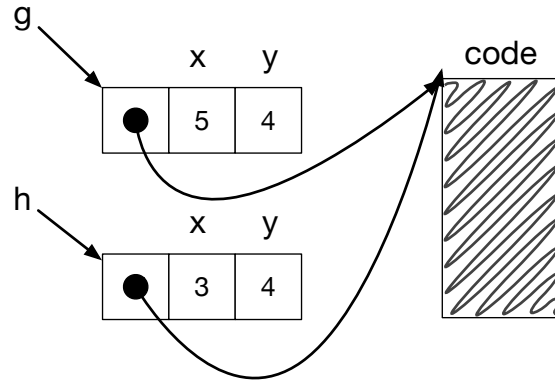


图 4.1 闭包示意图

```

(define (f [x : Int]) : (Int -> Int)
  (let ([y 4])
    (lambda: ([z : Int]) : Int
      (+ x (+ y z))))))

(define (main) : Int
  (let ([g ((fun-ref f) 5))]
    (let ([h ((fun-ref f) 3)]]
      (+ (g 11) (h 15)))))

⇒

(define (f [clos2 : _] [x : Int]) : (Vector ((Vector _) Int -> Int))
  (let ([y 4])
    (vector (list (fun-ref lambda2) x y)))))

(define (lambda2 [clos3 : (Vector _ Int Int)] [z : Int]) : Int
  (let ([x (vector-ref clos3 1)])
    (let ([y (vector-ref clos3 2)])
      (+ x (+ y z)))))

(define (main) : Int
  (let ([g (let ([clos5 (vector (list (fun-ref f6)))]
                ((vector-ref clos5 0) clos5 5)))]
    (let ([h (let ([clos6 (vector (list (fun-ref f6)))]
                ((vector-ref clos6 0) clos6 3)))]
      (+ ((vector-ref g 0) g 11) ((vector-ref h 0) h 15)))))

```

图 4.2 闭包转换示例

接受闭包。这样一来，我们就可以统一对待从匿名函数提取出来的全局函数和原本的全局函数，不需要再在后续的翻译工作中区分它们。

我们使用了一条下划线来表示闭包的类型，下划线表明我们不在意这个变量的类型，也不需要对其进行类型检查。之所以可以这么做是因为闭包是由编译器生成的，我们的转换代码可以保证我们总是能正确的使用这些闭包变量。而之所以要这么做是因为本文现有的类型系统无法正确地表示闭包的类型：闭包中的元素间接指向了闭包自己——第一个元素是函数，而这个函数又接受这个闭包作为第一个参数。（想要正确描述闭包的类型需要使用存在类型<sup>[13]</sup>。）

#### 4.4.4 赋值对闭包的影响

考虑下面这个例子：

```
(define (f) : (-> Integer)
  (let ([x 0])
    (let ([g (lambda: () : Integer x)])
      (begin
        (set! x 42)
        g))))
((f))
```

函数  $f$  中定义了局部变量  $x$ ，初始值为 0。然后定义了一个无参匿名函数，函数体只有一个表达式  $x$ 。也就是它什么都不做，直接返回  $x$ 。接着我们把  $x$  的值改为 42，然后返回刚才的匿名函数。我们期望  $((f))$  的返回值是 42。

但是上一小节描述的闭包转换会把正在编译闭包时  $x$  的值，也就是 0，放入闭包中。这样编译出来的程序就会返回错误的结果 0。

并且，虽然  $x$  是函数  $f$  的局部变量，它的生命周期却超过了  $f$  的生命周期。这说明了自由变量的生命周期是不确定的。在这个例子中，函数  $f$  执行结束后就退出了，但局部变量  $x$  仍然需要被匿名函数访问。只有当这个匿名函数也无法被访问到时，我们才可以放心地清理掉变量  $x$ 。因此，自由变量的值需要存放在堆上。我们通常使用“box”来表示这种在堆上分配单个值的行为，“unbox”则指代对 box 的解引用。这里我们并不需要引入 box 结构，单元素的向量就是 box。

虽然理论上来说自由变量的生命周期是不确定的，但并非所有的自由变量我们都要放在堆上。对于那些在代码中只有定义，没有重新被赋值的变量，我们还是可以简单地把它们的值放在闭包中。由于 box 会引入额外的开销，因此在本文的实现中，只有那些可能被重新赋值的（也就是在代码中出现在 `set!` 语句中的）自由变量才会被 box。上面例子中的  $f$  函数，在转换后将得到以下结果。

```
(define (f) : (-> Integer)
  (let ([x (vector 0)])
    (let ([g (lambda: () : Integer
                (vector-ref x 0))])
      (begin
        (vector-set! x 0 42)
        g))))
```

#### 4.5 内存管理

这一节讨论堆上的内存数据（在本文中只有向量这一种数据类型）以及如何管理堆内存。

堆内存与过程调用栈相互区分开，栈上的数据在函数退出后就无法再被访问了。而堆允许我们通过引用来在不同的代码之间访问运行时创建的数据。也正因此，堆上的数据的生命周期是不确定的。

```
(let ([v (vector (vector 44))])
  (let ([x (let ([w (vector 42)])
              (let ([_ (vector-set! v 0 w)])
                0))])
    (+ x (vector-ref (vector-ref v 0) 0))))
```

例如在上面这个例子中，变量 *w* 的生命周期在绑定完 *x* 后就结束了，但它指向的向量在这之后仍然可以被访问到，因此上面这段代码的结果是 0 和 42 相加，也就是 42。我们在前一节讨论闭包时也使用了单元向量来延长自由变量的生命周期。

从程序员可观察行为的角度来看，向量永远存在。当然，如果它们真的永远存在，堆将越来越大，最终耗尽内存。我们的语言必须实现自动地清理那些肯定不会再用到的数据，也就是垃圾回收。

#### 4.5.1 垃圾回收算法

本文使用的是一个较为简单的垃圾回收算法：双空间拷贝回收<sup>[14]</sup>。图4.3给出了在垃圾回收之前(上面)和之后(下面)的内存情况。在双空间收集器中，堆分为两个部分，分别命名为 *FromSpace* 和 *ToSpace*。最初，所有向量都分配到 *FromSpace*，当某次分配请求发现剩余的空间不够时，回收器开始工作。

由于程序的行为是无法预知的，我们不可能准确地知道哪些向量在以后会被使用，哪些是垃圾。但是我们可以保留下从当前所有变量出发，能访问到的那些向量，它们可能会被使用，也可能不会，但剩余的不可达的向量一定是垃圾。

首先，地址存放在寄存器和栈上的向量自然是可达的，我们把它们称为根集。从根集出发，所有能遍历到的向量均是活向量。然后，我们把所有的活向量拷贝到 *ToSpace* 中，并把 *ToSpace* 当成新一轮的 *FromSpace*，在里面分配接下来的数据，原来的 *FromSpace* 则当成下一次的 *ToSpace*。

在图4.3的例子中，根集中有三个指针，一个在寄存器中，两个在堆栈中。所有活向量都以保留指针间的关系的方式复制到 *ToSpace*。例如，寄存器中的指针仍然指向一个二元向量，它的第一个元素是一个三元向量，第二个元素是一个二元向量。有 4 个向量不能从根集到达，因此不需要复制到 *ToSpace* 中。

像这样的拷贝回收器的优点是分配数据很快（只需要看一下 *FromSpace* 的剩

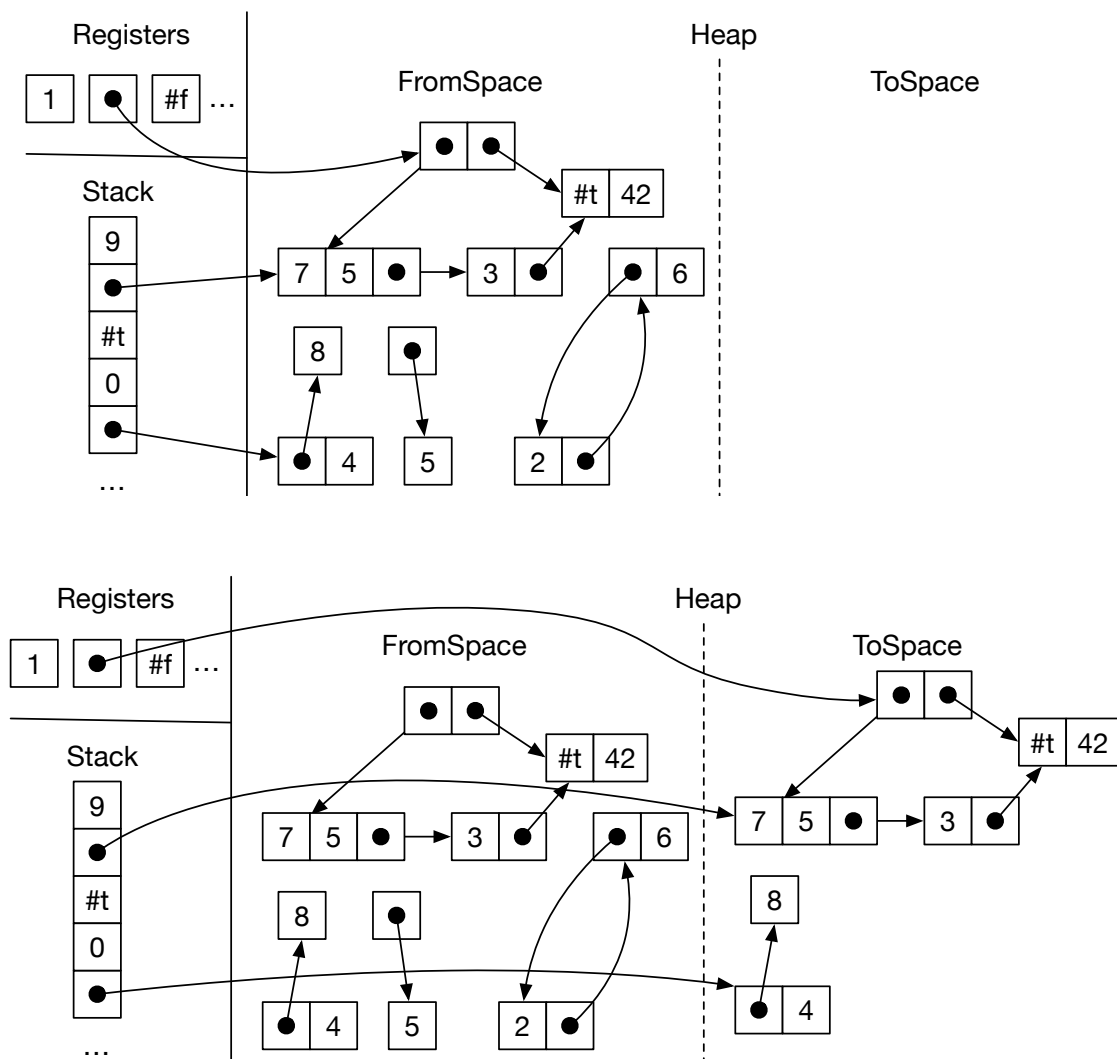


图 4.3 双空间拷贝回收示意图

余空间是不是够), 没有碎片, 可以回收包含循环引用的垃圾, 回收的时间复杂度取决于有用的数据大小, 而不是垃圾量。主要缺点是它浪费了一半的内存, 并且执行回收需要花长时间进行图遍历, 后者可以使用分代回收算法加以优化。

#### 4.5.2 拷贝回收

我们仔细看一下活对象拷贝的过程。堆上的对象和指针可以被视为一个图, 我们需要复制那些从根集可以到达的结点。图遍历算法如深度优先搜索或宽度优先搜索通过标记哪些顶点已经被避免重复遍历, 从而确保算法的终止。这些遍历算法还需要使用栈或队列等数据结构。本文使用宽度优先搜索和 Cheney<sup>[15]</sup> 的链表紧凑算法中的一个技巧, 在拷贝数据到 ToSpace 的同时使用 ToSpace 来表示队列。

图4.4展示了在拷贝过程中 ToSpace 的几个瞬间。队列由 ToSpace 开头的一块连续内存表示, 使用两个指针跟踪队列的头和尾。该算法首先将所有可以立即从

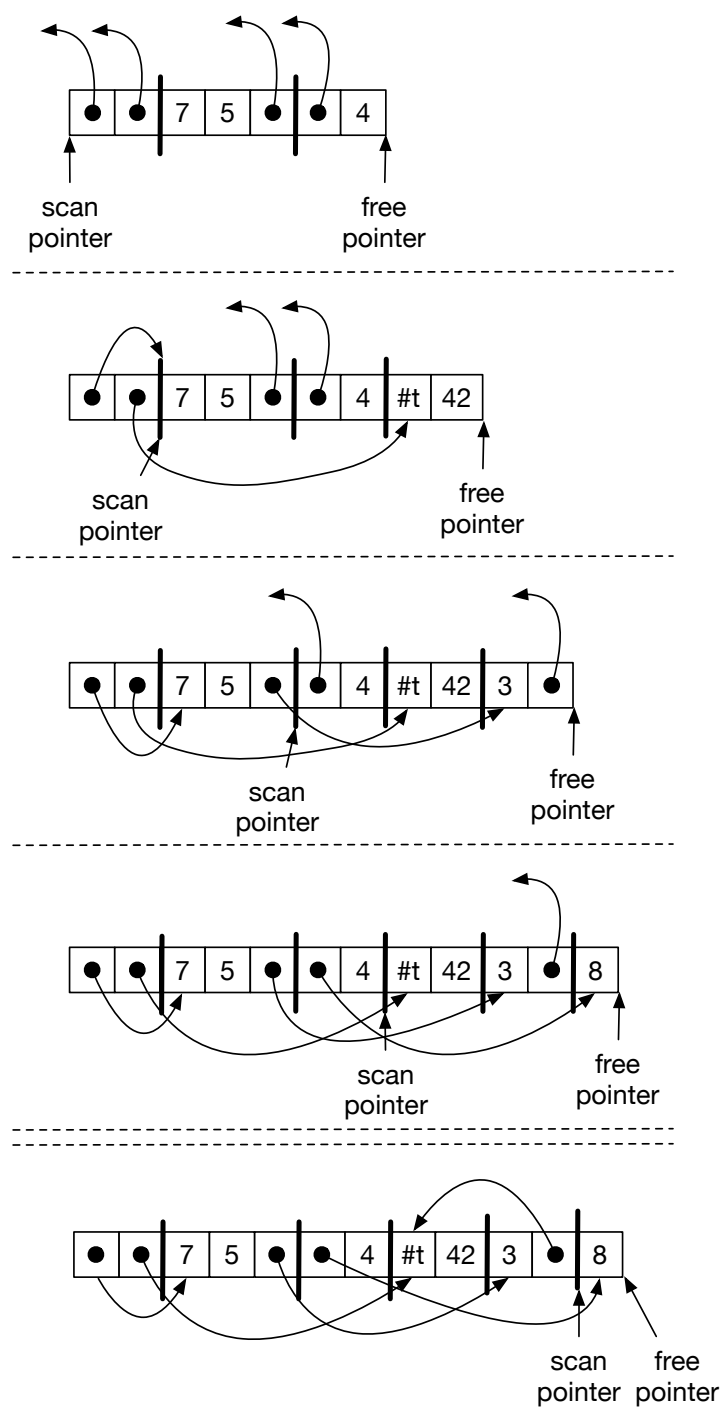


图 4.4 拷贝回收的详细过程

根集到达的向量复制到 ToSpace 中，以形成初始队列。当复制一个向量时，将旧向量标记为已被访问。下一小节将会讨论如何完成这个标记。注意，队列中复制的向量内的任何指针仍然指向 FromSpace。创建好初始队列后，算法将进入一个循环，在循环中，算法会考察队头的向量，将从该向量所有可以直接到达的向量复制到 ToSpace 并放到队尾（后移队尾指针），然后更新队头向量中指针，使它们指向新复制的向量，之后队头向量出队（后移队头指针），重复循环，直至队列为空，也就是队头指针赶上队尾指针。

在图4.4中，开始时队列中有三个向量。队头向量的第一个元素指向的 FromSpace 中的一个被标记为已访问的向量，这个已访问过的向量会保存它的克隆体在 ToSpace 中的地址（这一点在下一小节详细说明），在这个例子中正是 ToSpace 中的第二个向量，因此我们把这个指针改为指向第二个向量。

队头向量的第二个元素是指向 FromSpace 中一个内容为 #t 和 42 的向量，并且它没有被标记为已访问。我们把它复制过来，放在队尾，并修改队头向量中的指针。同时，我们还要标记 FromSpace 中的旧向量为已访问，并在其中记录它的克隆体的地址（同样，见下一小节）。此时队头向量处理完毕，我们后移队头指针，然后重复这个过程，直至队列为空。

### 4.5.3 向量在内存中的表示

首先，垃圾回收器需要区分指针和其他类型的数据。本文的实现策略如下：对于指向根集的那些指针，我们不放在寄存器或者普通的过程调用栈上，而是放在一个单独的栈上，这个栈当然也会随着函数的调用和返回相应地出栈入栈一些数据，我们这个栈称作根栈，也叫影子栈。

而对于向量中的指针和非指针数据，我们通过在每个向量的开头增加一个额外的 64 位标签来实现。图4.5展示了两个示例，注意这个图是以大端方式绘制的，从右到左，位置 0(最低位)在最右边，它对应着 x86 移动指令 salq (左移) 和 sarq (右移) 的方向。“指针掩码”部分用于标记向量中哪些元素是指针，1 代表为指针，0 则为其他类型的数据。指针掩码从第 7 位开始，总共占 50 位，因此该语言最多只允许长度为 50 的向量。元组的长度(元素的数量)存储在位 1 到 6 的位置。位置 0 则是我们的访问标记，如果它的值是 1，则表示没有被访问，也就是还没有被复制到 ToSpace。如果值是 0，则说明这个向量已经被访问过了，并且此时整个标签实际上是一个地址，表示的是复制后的向量在 ToSpace 中的位置。（因为我们的所有数据都是 64 位的倍数，所以的向量 8 字节对齐的，它们的地址的低 3 位总是 0。）

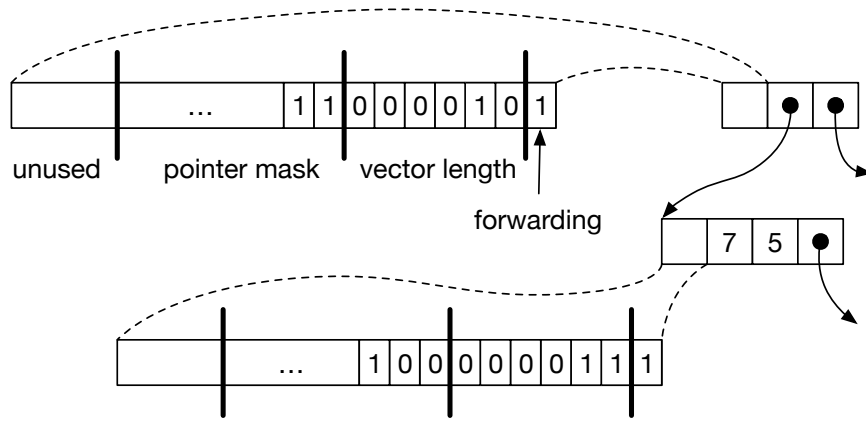


图 4.5 向量在内存中的表示

```

(vector  $e_0 \dots e_{n-1}$ )
  ↓
(let ([ $x_0$   $e_0$ ]
      [ $x_{n-1}$   $e_{n-1}$ ]
      ...))
  (begin
    (if (< (+ (global-value free_ptr) bytes)
            (global-value fromspace_end))
        (void)
        (collect bytes))
    (let ([ $v$  (allocate len type)])
      (begin
        ([_ (vector-set!  $v$  0  $x_0$ ))]
        ([_ (vector-set!  $v$   $n-1$   $x_{n-1}$ ))]
         $v$ ))))

```

图 4.6 展开向量定义

#### 4.5.4 展开向量定义

我们用 C 语言来实现垃圾回收器，编译生成一个目标文件，和我们的编译器翻译出的汇编代码生成的目标文件一起链接成最终的可执行文件。垃圾回收器提供了几个接口。其中 `initialize` 函数初始化 `FromSpace`、`ToSpace` 和根栈，在主函数的准备工作中被调用。`initialize` 函数将 `FromSpace` 开头的地址放入全局变量 `free_ptr` 中。全局变量 `fromspace_end` 指向的地址是 `FromSpace` 的最后一个元素的后面一个位置。`rootstack_begin` 变量指向根栈的第一个元素。

只要 `FromSpace` 中还有剩余空间，生成的代码就可以通过向前移动 `free_ptr` 来分配向量。`FromSpace` 中剩余的空间是 `fromspace_end` 和 `free_ptr` 之间的差值。当 `FromSpace` 中没有足够的空间留给下一次分配时，就调用 `collect` 函数。`collect` 函数接受一个指向根栈当前顶部的指针（栈顶元素的后面一个位置）和需要分配的字节数。`collect` 函数执行前文描述的复制回收过程。

编译器需要把定义向量的表达式翻译成一系列表达式：1) 把向量的各个元素的表达式绑定到一系列临时变量上；2) 判断 `FromSpace` 是否有足够的空间，如果不够则调用 `collect` 函数；3) 调用 `allocate` 函数；4) 初始化向量。如图4.6所示。

其中 `allocate` 接受参数 `len` 和向量的类型。`len` 指的是向量的长度，而 `bytes` 是需要为向量分配的总字节数，即 `len` 加上 1（一个 64 位的标签）再乘以 8。`allocate` 指令后续将被翻译为若干条汇编指令，包括根据向量的类型来计算标签的值。

## 4.6 原子化操作数

前面已经提到过，汇编语言指令的操作数不允许放置复杂表达式，只能是一个立即数或寄存器或内存位置。我们通过为每一个复杂的操作数引入一个新的 `let` 绑定，将复杂操作数绑定到新变量，然后使用新变量来代替复杂操作数，就像下面这样。

$$\begin{array}{ccc} (\text{let } ([x \ (+ \ 42 \ (- \ 10))]) & \Rightarrow & (\text{let } ([x \ (\text{let } ([\text{tmp1} \ (- \ 10)]) \\ (+ \ x \ 10)) & & (+ \ 42 \ \text{tmp1}))) \\ & & (+ \ x \ 10)) \end{array}$$

本文使用两个相互递归的函数：`rco-atom` 和 `rco-exp` 来实现这个过程。其思想是将 `rco-atom` 应用于需要成为原子的子表达式，将 `rco-exp` 应用于不需要原子化的子表达式。

`rco-exp` 函数根据需要在子表达式上分别调用 `rco-atom` 和 `rco-exp`。`rco-atom` 同样要对其接受的表达式分类讨论，继续在复合子表达式上相应的调用 `(+ 42 (- 10))` 或 `rco-exp`。`rco-atom` 函数会返回两个东西：一个原子化的表达式（一个变量或者一个数字之类的）和一个将临时变量映射到原来的复杂操作数的列表。`rco-exp` 根据这个列表创造出一系列的 `let` 绑定，包裹在原表达式外面，并把原表达式的那些复杂操作数用原子化后的表达式进行替换。

在上面的例子中，对表达式 `(+ 42 (- 10))` 应用 `rco-exp` 函数，`rco-exp` 会对子表达式 42 和 `(- 10)` 分别应用 `rco-atom`。前者返回 42 和空列表，后者返回变量 `tmp1` 和把 `tmp1` 映射到表达式 `(- 10)` 的列表。

$$\begin{array}{ccc} (- \ 10) & \Rightarrow & \text{tmp1} \\ & & ((\text{tmp1} \ . \ (- \ 10))) \end{array}$$

低效的实现可能会导致冗余变量的产生。我们要确保 `rco-exp` 只对需要的原子化的子表达式调用 `rco-atom`，`rco-atom` 则要确保不要为本身就已经是原子的表达式再创建临时变量。



## 4.7 显示化控制流

这一趟编译过程将把我们的语言翻译成类似四元式的序列，复杂的控制结构将会变成一些包含跳转语句的简单的语句序列。考虑下面的程序：

```
(let ([y (let ([x 20])
              (+ x (let ([x 22]) x)))]])
  y)
```

前面描述的编译过程会把程序翻译成左边的结果，显示化控制流则进一步得到右边的过程：三

<pre>(let ([y       (let ([x1 20])         (let ([x2 22])           (+ x1 x2)))]])   y)</pre>	⇒	<pre>start:   x1 = 20;   x2 = 22;   y = (+ x1 x2);   return y;</pre>
---	---	--

我们先来考虑最简单的情况：假设程序不包含分支指令，也不包含可变变量带来的结构，即只关心最后一个表达式的返回值的 `Begin` 语句和循环语句。那么程序就只剩下变量定义和运算（当然还有函数，但是这趟编译过程正是针对每个函数单独处理的，而函数调用则和运算按同样的方式处理）。可以看到，就像上面的例子一样，最终的结果就是一系列的赋值语句和一个返回语句。返回语句是由处于尾位置的表达式编译得到的为。尾位置的定义如下：

- (1) `(FunctionDef params body)` 中，`body` 处于尾位置。
- (2) 如果 `(let x e body)` 处于尾位置，则 `body` 处于尾位置。

我们使用两个相互递归的函数 `explicate-tail` 和 `explicate-assign` 来实现，如图4.7所示。

`explicate-tail` 函数应用于位于尾位置的表达式，而 `explicate-assign` 函数应用于出现在 `let` 右侧的表达式。二者都返回一串四元式。

如果 `explicate-tail` 收到的表达式不是 `let` 表达式，那么就简单的返回一个四元式 `return`。如果是 `let`，那么 `let` 中的 `body` 部分就属于尾位置，对其递归调用 `explicate-tail`，得到一串四元式 `instructions`，把变量名，`let` 语句的右侧，以及这串四元式作为参数传递给 `explicate-tail`。`explicate-tail` 对 `let` 语句的右侧表达式进行判断，如果不是 `let`，那么生成一个赋值四元式，把这个右侧表达式赋给变量名，然后添加到那串四元式的前面即可。这里用参数名 `cont` 来接收四元式序列，意思正是这串四元式将会变成赋值语句的 `continue`。而如果 `let` 的右侧又是一个 `let`，我们则应当先调用 `explicate-tail` 把这个内部 `let` 的 `body` 赋给外面的变量，得到一个新的 `cont`，把它和内部 `let` 的变量名和右侧表达

```

(define (explicate-tail e)
  (match e
    [(or (Int n))
     (Return (Int n))])
    [(Let x rhs body)
     (let ([instructions (explicate-tail body)])
       (explicate-assign rhs x instructions))]
    ...
  ))

(define (explicate-assign e x cont)
  (match e
    [(Int n)
     (append (Assign (Var x) (Int n))
              cont)]
    [(Let y rhs body)
     (let ([new-cont (explicate-assign body x cont)])
       (explicate-assign rhs y new-cont))]
    ...))

```

图 4.7 显示化控制流的实现

式再一次传给 `explicate-tail`。就像上面的例子，我们先把 `(+ x1 x2)`；赋给 `y`，`return y`；作为其 `cont`，然后再把这两条语句作为为 `x1`，`x2` 赋值的语句的 `cont`。可以说，这一趟编译过程生成四元式序列的过程是倒过来的。这种实现方式被叫做累积传递风格。

接下来我们考虑上分支语句和赋值，扩展尾位置的定义。

1. 如果 `(if pred then else)` 处于尾位置，则表达式 `then` 和 `else` 均处于尾位置。
2. 如果 `(begin ... final-e)` 处于尾位置，则 `final-e` 处于尾位置。

与此类似，我们再定义谓词位置和副作用位置，并相应地定义 `explicate-pred` 和 `explicate-effect` 函数。`explicate-effect` 接收 `cont`，`explicate-pred` 则接收 `then-cont` 和 `else-cont`，它们也都返回一串四元式序列。加入分支后，`explicate-pred` 函数中要生成一些跳转语句，分别指向 `then-cont` 和 `else-cont`。最终，这趟编译过程将能够完成如图4.8的转换。

## 4.8 生成汇编

这一步我们会将上一节中生成的四元式序列转换为 x86 汇编指令序列。把四元式转换成汇编指令的过程非常直白，因为四元式只有运算、赋值、函数调用、跳转等少数几种可能的类别，并且运算的操作数和函数调用的参数都是原子表达式。唯一的难点在于四元式包含变量，变量名当然是无穷多的，但 x86 没有变量，他

```

                                start:
                                x = (read);
                                y = (read);
                                if (< x 1) goto block40;
                                else goto block41;
                                block40:
                                if (eq? x 0) goto block38;
                                else goto block39;
                                block41:
                                if (eq? x 2) goto block38;
                                else goto block39;
                                block38:
                                return (+ y 2);
                                block39:
                                return (+ y 10);

(let ([x (read)])
  (let ([y (read)])
    (if (if (< x 1)
            (eq? x 0)    ⇒
            (eq? x 2))
        (+ y 2)
        (+ y 10))))

```

图 4.8 显示化控制流示例

只有寄存器和过程调用栈。在生成汇编这趟过程中，我们保留这些变量名，在下一趟编译过程再把这些变量放到合适的寄存器或栈上去。

接下来我们举一些典型的四元式来说明生成汇编的过程。首先是整数运算操作，以加法为例：

```

var = (+ atm1 atm2);    ⇒    movq arg1, var
                                addq arg2, var

```

翻译运算操作时要注意可以优化的情况。比如，如果加法的一个参数与赋值的左边的变量相同，那么就不需要额外的 `move` 指令。整个赋值语句可以被翻译成一条 `addq` 指令：

```

var = (+ atm1 var);    ⇒    addq arg1, var

```

`read` 函数在 x86 汇编中没有直接对应的功能，所以我们在 `runtime.c` 中用 C 语言实现一个 `read_int` 函数来提供这个功能。前文提到，`runtime.c` 中还包括垃圾回收器以及一些全局变量，这些功能合在一起被称为“运行时系统”，或简称为“运行时”。

```

var = (read);    ⇒    callq read_int
                        movq %rax, var

```

`Return` 语句的翻译方法和上面两个例子是一样的，只要把赋值左边的 `var` 改为 `%rax` 即可。对于布尔值真和假，我们把它们分别编译成立即数 1 和 0。然后是布尔运算 `not`：

```

var = (not atm);    ⇒    movq arg, var
                        xorq $1, var

```

和整数运算一样，这里也需要考察 `atm` 来判断能否优化进而减少一条指令。接下来我们看一下 x86 对布尔运算和条件分支提供的支持。

`cmpq` 指令比较它的两个参数来确定一个参数是小于、等于还是大于另一个参数。`cmpq` 指令在参数的顺序和结果放置的位置上不太一样。参数顺序是颠倒的：如果我们想知道  $x$  是不是小于  $y$ ，就需要写 `cmpq y, x`。`cmpq` 的结果被放置在特殊的 `EFLAGS` 寄存器中。这个寄存器不能直接访问，但可以通过许多指令进行查询。指令 `setcc d` 根据条件代码 `cc`（`e` 表示等于，`l` 表示小于，`le` 表示不等于，`g` 表示大于，`ge` 表示大于等于），将立即数 1 或 0 放入目标 `d` 中。还需要注意的是 `d` 必须是单字节寄存器，例如 `al` 或 `ah`，它们是 `rax` 寄存器的一部分。`movzbq` 指令可以将单个字节寄存器中的值移到 64 位寄存器里。综上所述，`eq?` 运算可以被这样翻译：

$$var = (eq? \text{atm}_1 \text{atm}_2); \quad \Rightarrow \quad \begin{array}{l} \text{cmpq } arg_2, arg_1 \\ \text{sete } \%al \\ \text{movzbq } \%al, var \end{array}$$

再来看 `if` 表达式的编译。指令 `jcc label` 更新程序计数器指向 `label` 之后的指令，这取决于 `EFLAGS` 寄存器中的结果是否与条件代码 `cc` 匹配，否则条件跳转指令还是继续执行下一条指令。因为条件跳转指令依赖于 `EFLAGS` 寄存器，所以通常它前面会立即有一个 `cmpq` 指令来设置 `EFLAGS` 寄存器。

$$\begin{array}{ll} \text{if } (eq? \text{atm}_1 \text{atm}_2) \text{ goto } \ell_1; & \Rightarrow \\ \text{else goto } \ell_2; & \end{array} \quad \begin{array}{l} \text{cmpq } arg_2, arg_1 \\ \text{je } \ell_1 \\ \text{jmp } \ell_2 \end{array}$$

接下来考虑和向量有关的指令。我们保留 `r11` 寄存器，专门用它来访问向量也就是堆上的数据。

$$\begin{array}{ll} var = (\text{vector-ref } vec \ n); & \Rightarrow \quad \begin{array}{l} \text{movq } vec, \%r11 \\ \text{movq } 8(n+1)(\%r11), var \end{array} \\ \\ (\text{vector-set! } vec \ n \ arg); & \Rightarrow \quad \begin{array}{l} \text{movq } vec, \%r11 \\ \text{movq } arg, 8(n+1)(\%r11) \end{array} \\ \\ vec = (\text{allocate } n \ type); & \Rightarrow \quad \begin{array}{l} \text{movq } free\_ptr(\%rip), \%r11 \\ \text{addq } 8(n+1), \\ \quad free\_ptr(\%rip) \\ \text{movq } \$tag, 0(\%r11) \\ \text{movq } \%r11, var \end{array} \\ \\ (\text{collect } bytes); & \Rightarrow \quad \begin{array}{l} \text{movq } \%r15, \%rdi \\ \text{movq } \$bytes, \%rsi \\ \text{callq } collect \end{array} \end{array}$$

`free_ptr` 中的地址是 `FromSpace` 中的下一个空闲地址，把它赋给 `r11`，然后把它向前移动  $8(n+1)$  个字节以留出足够的空间来放置向量的内容（每个元素是 8 个字节（64 位），以及一个 64 位的标签）。在编译时我们通过 `type` 信息计算出标签，放到向量的第 0 号位置。标签的组成参见图 4.5 处的描述。对垃圾回收函

数 `collect` 的调用被编译成对运行时里的 `collect` 函数的调用，参数分别是根栈的顶部地址和需要分配的字节数。我们还保留了 `r15` 寄存器，专门用来保存指向根栈顶部的指针，它会在主程序的准备工作中被初始化。

最后是函数。对于每个函数定义，我们都需要把参数从参数寄存器里移到对应的变量名中：

<pre>start:   instr<sub>1</sub>   ⋮   instr<sub>n</sub></pre>	$\Rightarrow$	<pre>start:   movq %rdi, x<sub>1</sub>   movq %rsi, x<sub>2</sub>   ⋮   instr<sub>1</sub>   ⋮   instr<sub>n</sub></pre>
---	---------------	---

函数赋值给变量成为 `load-effective-address` 指令：

<code>var = (fun-ref f);</code>	$\Rightarrow$	<code>leaq (fun-ref f), var</code>
---------------------------------	---------------	------------------------------------

然后是函数调用，正如第2.2节中所述，它们被编译成 `x86` 的间接函数调用：

<pre>var = (call fun arg<sub>1</sub> arg<sub>2</sub> ...);</pre>	$\Rightarrow$	<pre>movq arg<sub>1</sub>, %rdi movq arg<sub>2</sub>, %rsi ⋮ callq *fun movq %rax, var</pre>
--	---------------	--

如果函数调用是尾巴调用，如第2.3节所述，我们要把上面的 `callq` 换成 `jmp`，并在其前面加上弹出当前栈帧的一系列指令，但由于此时我们还没有进行寄存器分配，也没有生成函数的结束工作，我们不知道这些指令具体都有哪些，因此我们在这里引入一个临时的 `TailJump` 指令来替换 `callq`，后续分配完寄存器之后再把它替换成完整的弹出栈帧和跳转指令。

## 4.9 寄存器分配

这一步将会把前一节生成的包含变量的 `x86` 指令变成真正的 `x86` 指令，包含三个步骤：活变量分析、建立变量之间的干涉图、根据干涉图分配寄存器或内存。

### 4.9.1 活变量分析

在程序中的某一个点，如果一个变量或寄存器的当前包含的内容在后面会被使用，我们就说这个变量或寄存器在当前这个点是活的。

我们需要计算出每条指令后所有活变量的集合，对于不包含分支指令的简单情况，我们从后往前计算按下面的方法计算。

设  $I_1, \dots, I_n$  为指令序列。记指令  $I_k$  之后的活变量集为  $L_{\text{after}}(k)$ ， $I_k$  之前为

活变量集为  $L_{\text{before}}(k)$ 。一条指令之后的活变量集与下一条指令之前的活变量集相同。

$$L_{\text{after}}(k) = L_{\text{before}}(k + 1) \quad (4.1)$$

开始时，在最后一指令之后没有活变量，所以

$$L_{\text{after}}(n) = \emptyset \quad (4.2)$$

然后我们重复应用以下规则，将指令序列从后向前遍历。

$$L_{\text{before}}(k) = (L_{\text{after}}(k) - W(k)) \cup R(k) \quad (4.3)$$

其中  $W(k)$  是指令  $I_k$  中写入数据的变量或寄存器， $R(k)$  是指令  $I_k$  中读取数据的变量或寄存器。

注意 callq 指令应该在其写集  $W$  中包含所有调用者保存寄存器，因为这些寄存器可能在函数调用期间被被调用者使用。同样，callq 指令应当根据调用的函数的参数个数，在其读集  $R$  中包含适当的参数传递寄存器。

从显示化控制流这一编译过程的输出开始，编译器的中间形式就从原来的树变成了图。每个标签下的一系列四元式或者一系列汇编指令，一直到下一个标签或者到结束，被称作一个基本块。跳转指令可以从一个基本块跳转到另一个基本块。二者分别作为有向图的顶点和边，构成了控制流图 (CFG)<sup>[16]</sup>。

加入了分支和循环之后，我们要考虑两个问题。第一个问题是跳转指令的活变量分析。假设第  $k$  条指令是跳转指令，它指向某一个基本块，记该基本块的第一条指令前面的活变量集为  $L_{\text{before}'}(1)$ 。如果是无条件跳转指令，则

$$L_{\text{before}}(k) = L_{\text{before}'}(1) \quad (4.4)$$

如果是条件跳转指令，则

$$L_{\text{before}}(k) = L_{\text{before}'}(1) \cup L_{\text{before}}(k + 1) \quad (4.5)$$

第二个问题是以什么样的顺序、重复多少次才能计算出各个基本块中各指令的  $L_{\text{after}}$ 。解决方法如下：对控制流图进行拓扑排序，选择最末尾的一个基本块开始分析，计算块内指令的 live-after 以及块的 live-before，也就是第 1 条指令的 live-before，如果块的 live-before 中的活变量在这趟分析后增加了，那么对于所有能跳转这个基本块的块都再进行一次分析，重复进行下去，直到所有块的 live-before 集合都不再增大。这种迭代分析控制流图的方法叫作数据流分析<sup>[17]</sup>，它也适用于许多其他静态分析问题。

## 4.9.2 使用图着色算法进行分配

知道了任意时刻的活变量集合，我们就能知道某两个变量能不能被分配到同一个寄存器：如果它们在某一刻同时是活变量，那我们显然不能把它们分配到同一个寄存器。把这种情况称之为两个变量互相干涉。我们创建一个无向图，称之为干涉图，顶点是所有变量，然后在所有干涉的变量之间添加边。

接下来就是为图上的变量分配寄存器，用从 0 到  $k-1$  的整数对应于可以用来分配变量的  $k$  个寄存器。整数  $k$  及以上对应的是堆栈位置。这样一来寄存器分配就变成了图着色问题，这些整数就是可以使用的颜色，有边的顶点之间不能分配相同的颜色。

本文使用的图着色算法是 DSATUR 算法<sup>[18]</sup>。其中 SATUR 是饱和度的缩写，它指的是一个顶点上的约束。我们为某一个顶点涂上一个颜色后，所有与它相邻的顶点就不能再选用这个颜色，也就多了一个约束。DSATUR 算法每次都选择一个约束最多的顶点，给他涂上编号尽可能小的颜色，然后给相邻的为上色顶点加上约束，再在剩下的未上色顶点中重复这个过程。

图着色完成后，我们就为所有变量都分配了一个寄存器或者一个栈上的空间。x86 代码已经大致生成完毕，接下来只需要再整理一下一些不合法的 x86 指令，再添加上函数的准备工作和结束工作指令即可。

## 4.10 整理指令

我们生成的 x86 指令中还存在一些不符合规范的地方。比如，x86 中一条指令最多访存一次，而前面的编译过程可能会生成包含两次内存引用的代码，这时我们可以利用保留的 `rax` 寄存器做一次中转：

```
movq -8(%rbp), -16(%rbp)    ⇒    movq -8(%rbp), %rax
                                movq %rax, -16(%rbp)
```

如果间接函数调用指令的操作数没有被分配到寄存器上，而是使用了栈空间，我们同样要做一次中转：

```
callq *-8(%rbp)             ⇒    movq -8(%rbp), %rax
                                callq *%rax
```

前面的编译过程会引入很多临时变量，而图着色算法会把一些变量分配到一个寄存器或栈位置，这就可能出现一些 `movq` 指令，它的两个操作数是一样的，我们可以直接把这些指令删除掉。

最后就是为每个函数生成第2.2节中描述的准备工作和结束工作。在准备工作中，我们要把所有用到的被调用方保留寄存器存入栈中，因为这里面可能存放着

调用这个函数的调用方的一些数据。结束工作中再把这些内容逆序弹出回这些寄存器里。准备工作中还要根据程序中向量的数量移动根栈指针 `r15`。相应地，在结束工作中也需要把它移动回去。

生成了结束工作后，我们还要把4.8一节中临时创造的 `TailJump` 指令换成真正的间接跳转指令 `jmp *%rax`，并在它的前面插入结束工作中除了 `retq` 之外的指令。最后，主程序的准备工作者还需要初始化垃圾回收器，即调用垃圾回收器的 `initialize` 函数和把根栈顶部 `rootstack_begin` 送入根栈指针寄存器 `r15`。

至此，所有的编译工作均已完成，编译器已经能够将一段源程序编译成正确、合法的 x86 汇编程序，并且能和 `runtime.o` 一起被 GCC 编译套件链接生成一个能在 x86-64 平台的 Linux 系统上运行的可执行文件。

#### 4.11 本章小结

本章详细描述了从源程序中解析出抽象语法树后到生成可执行文件的所有编译过程，其中每一节对应了对中间表示，也就是语法树或四元式的一趟或若干趟转换，包括其作用和目的以及具体的算法或转换示例。本章也是本文工作的重点。



## 5 编译过程展示系统

### 5.1 展示系统设计与实现

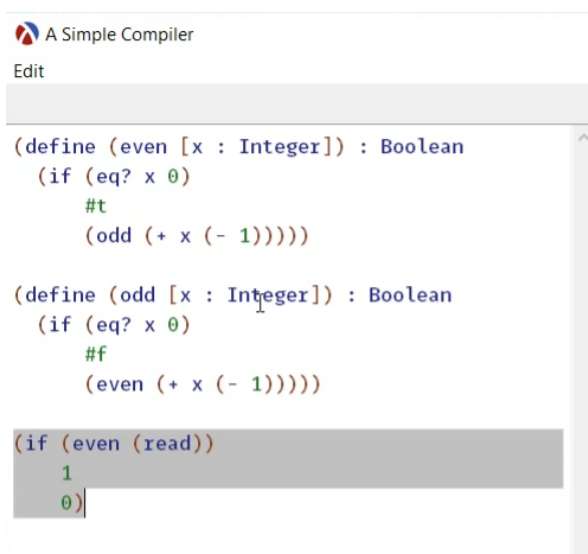
既然采用了 Nanopass 策略来实现编译器，在编译的过程中输出每一趟编译过程的结果是很自然的想法。我们可以通过观察每一个编译过程的输入和输出来确保我们的想法是可行的，我们的实现是正确的。但是，在终端中从上到下打印出的编译结果不方便进行对比，也缺乏交互性。

因此，本文实现了一个图形界面程序，允许我们在一个能自动格式化 Lisp 源码的文本框中输入源程序，然后在一系列文本框中展示编译过程的每一趟的转换结果，并在指定目录中生成最终的汇编代码和可执行文件。

本文采用了 Racket 的 framework 库图形用户接口来实现展示系统，该库是对 Racket 的原生图形用户接口 racket/gui 库的二次封装，提供了许多更易用的类和函数。本文使用 racket:text% 类的实例作为源程序的输入框，该输入框能自动缩进 Lisp 类语言的源代码，高亮所有 Racket 关键字。展示中间结果的文本框同样使用该类，但对它们调用 lock 函数进行锁定，这样这些文本框就会停止高亮并且无法被编辑。由于中间过程数量众多，这样做可以减少程序运行时的开销，同时也防止由于误操作导致中间结果被修改。

### 5.2 展示系统运行效果

图5.1展示了在该编译过程展示系统中输入一段源程序的效果。

The screenshot shows a window titled "A Simple Compiler" with a sub-header "Edit". Below it is a text area containing Lisp code. The code is color-coded: keywords like 'define', 'if', 'even', 'odd', 'read' are in blue; literals like '#t', '#f', '1', '0' are in green; and symbols like 'x', 'Integer', 'Boolean' are in red. The code defines two functions, 'even' and 'odd', and then calls 'even' on the result of 'read'.

```
(define (even [x : Integer]) : Boolean
  (if (eq? x 0)
      #t
      (odd (+ x (- 1)))))

(define (odd [x : Integer]) : Boolean
  (if (eq? x 0)
      #f
      (even (+ x (- 1)))))

(if (even (read))
    1
    0)
```

图 5.1 展示系统输入源程序

点击编译按钮后，所有的编译过程的中间结果以及最后的 x86-64 代码都将并排展示在一系列横向排列的文本框中，方便我们对各个中间结果进行横向对比，指定的目录下也会生成最终的可执行文件。展示系统还利用 graphviz 软件绘制了所有函数的干涉图，这些图片也会以 svg 格式的文件存放在同一个目录下。图5.2展示了点击编译按钮后的展示系统。图5.3展示了图5.1中的 even 函数的变量和寄存器干涉图。

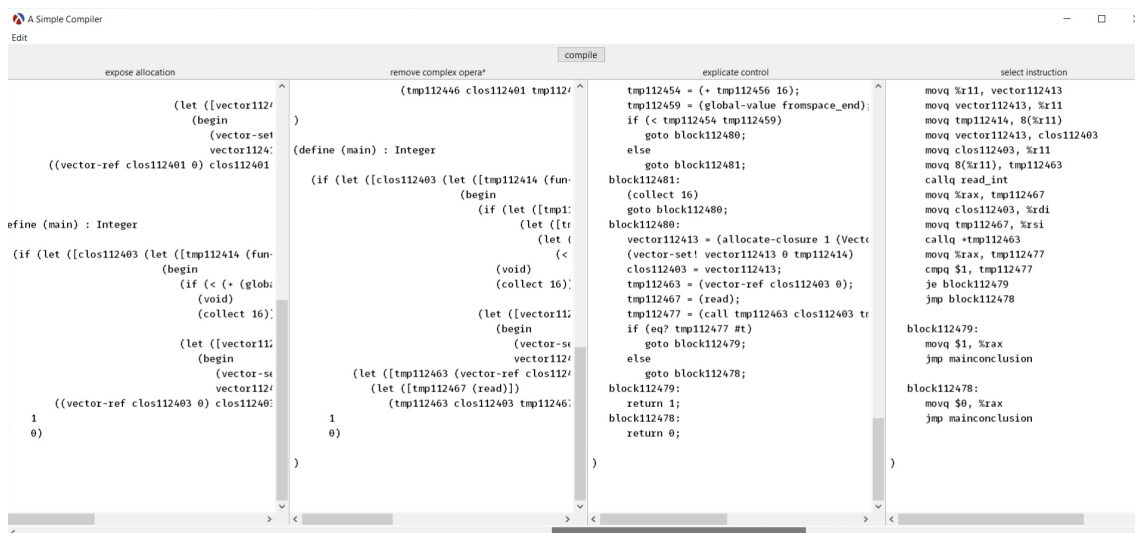


图 5.2 并列展示编译过程的中间结果

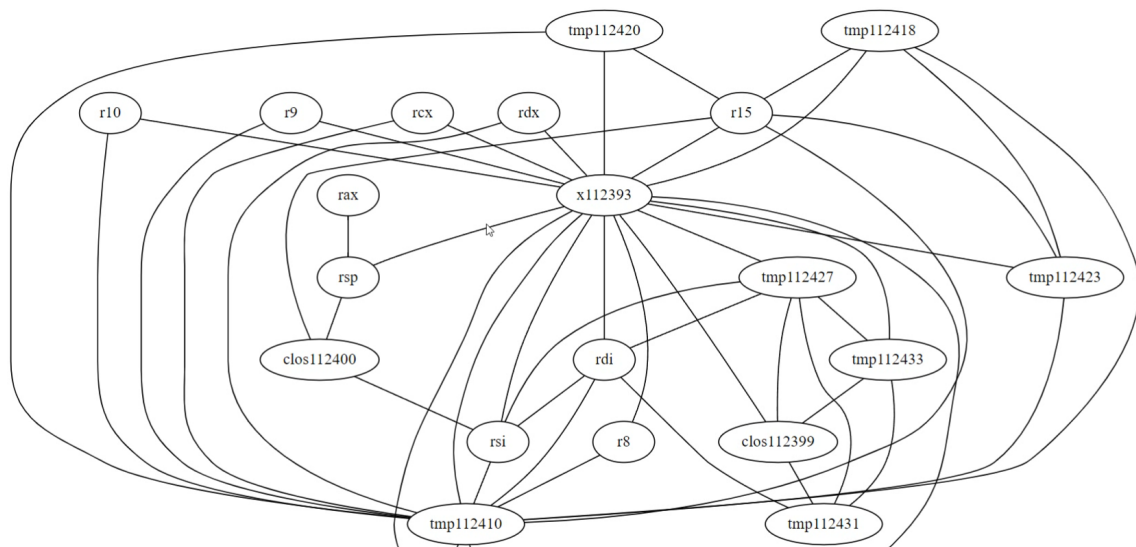


图 5.3 调用 graphviz 绘制干涉图

最后，图5.4展示了图5.1中示例程序的运行结果。

```
zijun in /mnt/c/Users/zijun/dev/eoc on master ● ● λ gcc -o gui-test gui-test.s runtime.o
zijun in /mnt/c/Users/zijun/dev/eoc on master ● ● λ ./gui-test
123456
zijun in /mnt/c/Users/zijun/dev/eoc on master ● ● λ echo $?
1
zijun in /mnt/c/Users/zijun/dev/eoc on master ● ● λ ./gui-test
1234567
zijun in /mnt/c/Users/zijun/dev/eoc on master ● ● λ echo $?
0
zijun in /mnt/c/Users/zijun/dev/eoc on master ● ● λ
```

图 5.4 示例程序的运行结果

### 5.3 本章小结

本章首先说明了对于 Nanopass 策略的编译器而言，设计一个交互式的编译过程展示系统的意义和必要性，然后简要介绍了本文的实现方法，最后展示了系统的具体效果。

## 6 总结和展望

编译技术作为计算机科学领域中一个必不可少的组成部分，涉及的专业知识繁多，深入研究其实现原理及技术，能促进我们更好地理解计算机软硬件系统，提升我们的开发能力。本文简要阐述了一个编译系统的实现过程，主要工作和成果包括：

（1）编译系统前端的设计与实现，能从文件中读取源代码生成使用结构体表示的抽象语法树；

（2）基于 Nanopass 思想，设计实现了编译系统后端，历经十余趟转换后生成 x86-64 汇编代码；

（3）将生成的汇编代码与 C 语言实现的运行时目标文件链接生成可执行文件，运行时实现了垃圾回收等功能；

（4）最后针对本文实现的编译系统进行了详细的测试，保证了编译系统的可靠性。

实现的语言支持可变变量和整数运算，if 语句和 while 语句，头等函数和闭包，自动内存管理等。虽然所有功能都得到了正确的实现，但仍然存在许多不足和需要改进的地方：

（1）增加语言的功能，例如结构体和模式匹配等，目前的语言非常简单，远远无法满足日常使用的需求；

（2）采用更先进的算法，加入更多优化，例如对闭包和垃圾回收算法的优化，目前系统各个部分的算法采用的大多是比较简单的算法，且本文未与现有的编译系统进行对比，没有对编译速度以及编译得到的程序大小与运行速度进行测试。

## 参考文献

- [1] McCarthy J. Recursive functions of symbolic expressions and their computation by machine, part i[J]. Communications of the ACM, 1960, 3(4): 184–195.
- [2] Steele G.L. Rabbit: A compiler for scheme[D]. Massachusetts Institute of Technology, 1978.
- [3] Dybvig R.K. The development of chez scheme[A]. ICFP ' 06: Proceedings of the eleventh ACM SIGPLAN international conference on functional programming. New York, NY, USA: Association for Computing Machinery, 2006: 1–12.
- [4] Appel A.W. Compiling with continuations[M]. Repr ed. Cambridge: Cambridge Univ. Press, 1992.
- [5] Queinnec C. Lisp in small pieces[M]. Cambridge: Cambridge University Press, 1996.
- [6] Sarkar D., Waddell O., Dybvig R.K. A nanopass infrastructure for compiler education[A]. ICFP ' 04: Proceedings of the ninth ACM SIGPLAN international conference on functional programming. New York, NY, USA: Association for Computing Machinery, 2004: 201–212.
- [7] Keep A.W., Dybvig R.K. A nanopass framework for commercial compiler development[A]. ICFP ' 13: Proceedings of the 18th ACM SIGPLAN international conference on functional programming. New York, NY, USA: Association for Computing Machinery, 2013: 343–350.
- [8] Siek J.G. Essentials of compilation[M]. Massachusetts Institute of Technology Press, 2022.
- [9] Matz M., Hubicka J., Jaeger A., Mitchell M. System v application binary interface, amd64 architecture processor supplement[M]. 2013.
- [10] Knuth D.E. Backus normal form vs. backus naur form[J]. Communications of The Acm, 1964, 7(12): 735–736.
- [11] Backus J.W., Bauer F.L., Green J., Katz C., McCarthy J., Perlis A.J., et al. Report on the algorithmic language algol 60[J]. Communications of The Acm, 1960, 3(5): 299–314.
- [12] Cardelli L. The functional abstract machine[M]. AT&T Bell Laboratories, 1983.
- [13] Minamide Y., Morrisett G., Harper R. Typed closure conversion[A]. POPL ' 96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on principles of programming languages. New York, NY, USA: Association for Computing Machinery, 1996: 271–283.
- [14] Wilson P.R. Uniprocessor garbage collection techniques[A]. Bekkers Y., Cohen J. Memory management[C]. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992: 1–42.
- [15] Cheney C.J. A nonrecursive list compacting algorithm[J]. Communications of The Acm, 1970, 13(11): 677–678.
- [16] Allen F.E. Control flow analysis[A]. Proceedings of a symposium on compiler optimization[C]. New York, NY, USA: Association for Computing Machinery, 1970: 1–19.
- [17] Kildall G.A. A unified approach to global program optimization[A]. POPL ' 73: Proceedings of

the 1st annual ACM SIGACT-SIGPLAN symposium on principles of programming languages.  
New York, NY, USA: Association for Computing Machinery, 1973: 194–206.

- [18] Brélaz D. New methods to color the vertices of a graph[J]. Communications of The Acm, 1979, 22(4): 251–256.

## 致谢

感谢我的指导老师安宁和杨矫云，不论是在我的本科学习期间还是在此次毕业设计中，两位老师都给了我许多宝贵的指导和建议。

感谢我的父母，他们在我生活和学业的方方面面给予了无条件的支持和鼓励。

感谢麻省理工学院的 Gerald Jay Sussman 和 Hal Abelson，他们充满热情、幽默风趣的课程带我进入了编程的大门，激发了我对编程语言的兴趣。

感谢印第安纳大学的 Jeremy Siek 及其前辈 Kent Dybvig 和 Daniel Friedman，他们提出的思想方法为我的毕业设计扫除了障碍，让过程充满乐趣。

感谢我的母校合肥工业大学，给我提供了一个优美的学习和生活环境。每当焦虑迷茫时，俩人湖畔的晚风总能让我感到平静。

感谢这四年中遇到的所有朋友们，你们的陪伴是我人生中最宝贵的财富。

作者：余梓俊

2022 年 5 月 18 日