清 华 大 学

综 合 论 文 训 练

**题目：Ucore操作系统在网络环境上的建立与应用**

**系 别：计算机科学与技术系**

**专 业：计算机科学与技术**

**姓 名：王呈瑞**

**指导教师：陈渝 教授**

2016 年 6 月 6 日

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存该论文。

(涉密的学位论文在解密后应遵守此规定)

签 名： 导师签名： 日 期：

# 中文摘要

Ucore操作系统的前身，是麻省理工学院的Frans Kaashoek教授，在参考了某一个版本的UNIX的基础上，通过了MIT许可的情况下，构建了在X86机器上运行的教学用操作系统，名字叫做xv6。我们又将xv6的结构框架和设计思想拿来借鉴和完善，于是就有了功能较为完善的Ucore操作系统。

Ucore既可以在标准的x86计算机上运行，又可以运行在在基于x86的虚拟机上。现在能够在本地独立使用，将来也能搭建在网络环境上。这样，Ucore从一个独立的操作系统，就有机会能够扩展成主机与主机之间跨平台的交互系统，大大增强了应用性，适用性和便利性。

本文主要围绕着利用超文本HTML上嵌套的脚本语言JavaScript语言，建立将Ucore源代码转换成执行程序的编译器uComplier，以及测试ucore程序的运行，将中间结果清晰显示的模拟器uMonitor。这样上就可以在网络环境中实现并运行ucore操作系统了。

**关键字：**操作系统，网络，编译器，模拟器

# ABSTRACT

Before the appearance of Ucore operating system, Dr. Frans Kaashoek from MIT, refer to one version of the UNIX series, accept the MIT license, and build the OS which is base on the X86 instruction collection and named as ‘xv6’. Then we could understand the framework and designing principle, use and optimize them to our new OS ‘Ucore’.

As well as running in X86-computer, Virtual machine based on X86 could work normally. Not only the independent PC, But also the web environment. If build it on web, we have chance to extended it to the interactive system over the platform, which well improve the practicability, usability and convenience.

Surrounding the JavaScript on the HTML, this article splits into two parts: UCompiler, which translate the source code to the executable file. And UMonitor, which run and debug the Ucore OS, print the debug information on the screen. After we do it, the Ucore OS performs well on the Web.

**Keywords：**operating system, web, compiler, monitor

# 目录

中文摘要 3

ABSTRACT 4

目录 5

第1章 引言 7

1.1 研究背景 7

1.2 研究意义 8

1.3 研究现状 8

1.4 本文的主要工作 9

1.5 本文的组织架构 9

第2章 编译器 10

2.1 基础知识 10

2.2 介绍 11

2.3 c语言结构分析 11

2.4 逻辑树 13

2.4.1 逻辑树的结构 13

2.4.2 逻辑树的生成策略 16

2.4.3 表达式生成策略 18

2.5 词义分析 20

2.6 语义分析 22

2.7 程序生成 24

2.8 编译优化 28

第3章 模拟器 29

3.1 v9-cpu介绍 29

3.1.1 指令 29

3.1.2 内存 30

3.1.3 输入与输出 31

3.1.4 中断和异常 31

3.2 uMonitor模拟器 32

3.2.1 分析 32

3.2.2运行流程 32

3.2.3 辅助工具 34

3.2.4 信息处理 36

3.2.5 调试命令 39

第4章 实验测试 41

4.1 测试准备 41

4.1.1 单元测试 41

4.2 操作系统测试 43

4.2.1 测试方案 43

4.2.2 中间转译器uTranslator 43

第5章 实验过程中遇到的问题及解决方法 45

第6章 结论与展望 46

6.1 工作总结 46

6.2 目前问题和今后展望 46

插图索引 48

表格索引 49

# 第1章 引言

## 1.1 研究背景

操作系统（Operating System），是管理和控制计算机的最基本的系统软件。能够在计算机上通过文件路径打开文件或者执行程序，能够通过调度多个进程同时运行多个程序，能够给每个进程分配相应的空间，进程之间的空间资源可以互斥和共享。能够提供较为便利和美观的用户界面，为用户和开发者提供API接口等。

ucore操作系统是用于支持课程教学的内容，通过让学生了解结构和原理，来掌握课程的各个知识点。Ucore可以搭建Unix/Linux内核的X86指令集系统中，现比较常用的是Ubuntu系统，可通过qemu模拟X86计算机来运行ucore。

V9-cpu是为了教学使用，而将复杂的细支简化后，也能正常运行的cpu结构。V9-cpu虽然寄存器数量相对较少，而且指令格式也比较简单，但是指令类型更加丰富一些。因为寄存器的减少，指令格式的简化，使得同学们能把重点集中在os上，而不是一直纠结于指令使用的寄存器的编号，然后记下来每个寄存器的含义。

最初版本的V9是国外的一名本科生用C语言开发的，开发了生成和运行v9指令集程序的编译器和模拟器。但是使用的程序语言是c语言，不能够在网络前端环境上运行，所以如果能够放在前端运行的话，可以使用超文本网页内嵌的脚本语言，来构成编译模拟器的工作逻辑。HTML内的Javascript就可以担此重任。

Javascript是属于网络的脚本语言，被广泛的应用于网络相关的开发，一般情况下，Javascript通过嵌入到超文本网页HTML中，在打开网页的时候就开始运行了。和python一样，Javascript也是解释性的直译语言，且具有弱类型的特点。Javascript在其发展的初期只是用于客户端上的脚本，且主要是为了提升视觉效果，随着诸如Node.js的网络引擎，以及Javsscript自带的事件驱动，异步IO等特性，被逐渐使用于服务器端。

## 1.2 研究意义

为什么要将v9的工程移植在网络上呢？虽然v9这样小规模的工程不需要消耗太多的空间资源，但是一些大的工程不能够局限于本地的一台或者多台机器，所以需要通过远程服务器调配外部资源来使用。能在网络上搭建工程，那么并行计算，云计算等网络技术就能得到发展。

比起在一台主机上运行本地的程序，不如直接将程序移植到网页的脚本语言中，这样服务器与本地主机的交互性就会更强。打个比方，服务器向本地传输二进制流，假设流格式是受到保护的，那么像本地终端运行的程序传输这个流数据就非常不方便，如果将程序移植在Javascript这样的网页脚本中，在数据转译上就省去了很大的麻烦。

平台通常是指硬件和软件的结合，以特定的完备指令集设计的CPU硬件能够运行这套指令集的程序，而操作系统软件作为CPU的管理者，完善了多进程，内存管理，文件系统等功能，提供了与用户交互的一系列API接口。而像c,c++的跨平台只是硬件上的跨平台，当涉及到os的API时，不同os的API接口多半是不通用的，软件层面的跨平台就行不通了。而ython,Javascript这类脚本语言是解释运行的，除了解释器与平台相关以外，其他都是与平台无关的。从上述观点来看，网络环境下的脚本语言，相比较于传统的高级语言，更具备跨平台这一重要优势。

## 1.3 研究现状

目前，已经有很多能在web上较为流畅运行的软件，甚至还有完整的操作系统。用Node.js开发的操作系统NodeOS，完全建立在Linux内核之上，采用shell和NPM进行包管理。虽然在性能上不及本地软件，但是随着分布式计算和并行处理的不断发展，网络环境更有利于和外部的大量服务器资源进行信息交互和任务分配，开源程序通过web编译器生成在线的可执行程序，并通过虚拟机去运行，将会是更有前景的方式，有希望能够将传统的本地编译本地运行的方式取而代之。

## 1.4 本文的主要工作

本文主要对与vx6操作系统和ucore操作系统相关的编译器uCompiler和模拟器uMonitor的架构设计和功能实现进行详细分析，特别是支持uCompiler的核心数据结构逻辑树，uMonitor的运行流程和调试支持进行分析。此外，还将vx6和ucore移植到v9-cpu平台上，针对ucore系统开发了内存查看，函数调用路径等功能。

## 1.5 本文的组织架构

第二章主要介绍了编译器uCompiler的重要实现机制，包括逻辑树的架构和实现以及代码的生成策略和方法。

第三章主要介绍了模拟器uMonitor的重要实现机制，包括运行流程，对源码的信息处理以及调试扩展工具。

第四章展示所做的工作的实际运行效果。

第五章对本文进行总结归纳，以及对以此为基础的后续工作所提出的建议和规划。

# 第2章 编译器

## 2.1 基础知识

编译（compile），是将高级语言编写的程序代码，通过固定的逐步流程转变为机器可识别的执行程序的过程。编译器工作的主要流程为，高级语言编写的源程序，通过词法分析得到词法称谓（token），再通过语法分析得到表示代码语法的语法树，再通过语义分析得到更接近机器指令逻辑的中间代码，通过优化和链接后生成可执行的程序。

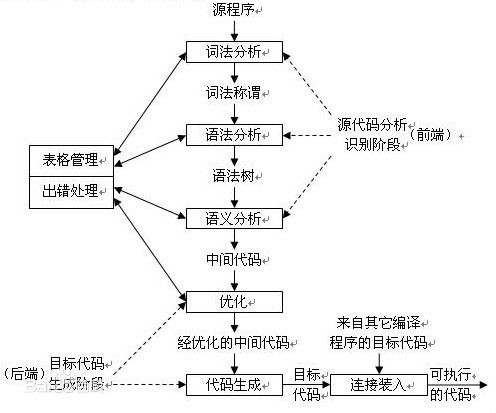


图1.1 编译器编译程序流程图

高级语言便于开发者阅读和理解，一个优秀的高级语言能够将严谨和效率兼备。但是将高级语言放在内存指令区，让计算机cpu直接去运行是完全不可能的，或者说是根本不需要的。相比较与编程语言，机器指令结构清晰，功能简单，虽然看上去非常繁琐，但是直能够接编码并让cpu去直接识别，这是机器指令的主要作用。

每个cpu的设计都需要遵循指定的功能完备的机器指令集，当今主要使用的指令集包括Intel的x86-64,SSE系列,EM64T等指令集，这里主要从x86系列开始谈起。X86一开始是专门开发并使用于intel的第一个16位的cpu——i8086中，为了能够确保运行之前开发的应用程序，intel一直沿用x86指令集至今。

## 2.2 介绍

uCompiler是通过javascript语言实现的，用于编译xv6操作系统和ucore操作系统源程序的编译器，其主要的特点是在客户端上在线运行。

uCompiler编译器能够编译c语言，其主要目的是用于编译用c语言实现的xv6和ucore的源代码。uCompiler相比较于标准的编译器，以同为c语言编译器gcc为例，gcc很多的主要功能uCompiler都忽略了。gcc有很多的编译选项，很多丰富的功能通过这些选项来打开或关闭。包括-O2,-O3等优化选项，-unused等警告声明未使用的代码元素，还有其他的编译选项。

从编译流程的角度来看，uCompiler是通过源代码->逻辑表达树->目标指令的流程来编译的，编译的过程主要围绕于表达程序逻辑的树结构来进行的。可以说uCompiler试图尝试突破原有的固定模式，引入一种新的基于树型结构的编译方法，并取得了很好的效果。

除此之外，uCompiler还将词义分析和语义分析一气呵成，在生成单词token时立刻进行逻辑检查，这样更能支持构建基于代码逻辑的树形结构。在第一步合并的工作完成后，逻辑树就已经初步完善了。到这里就已经完成大部分的工作了。

## 2.3 c语言结构分析

uCompiler要编译的高级语言，是我们经常使用的c语言，c语言的语法相对于c++和java来讲较为简单，因为没有涉及到太多面向对象的特征。因此，通过语法方程组能够清晰表示出C语言的语法。

code := unit code

unit := struct | enum | func | var

struct := ‘struct’ word ‘{‘ struct\_code ‘};’

enum := ‘enum {‘ enum\_code ‘};’

func := type name ‘(‘ func\_args ‘)’ ‘{‘ func\_para ‘}’

…

不考虑逻辑结构，c语言可以视为由**单词**和**符号**构成的序列。单词又分为关键字和非关键字两种。c语言的关键字可以分成多个类别来列举。

表1.1 c语言关键字分类

|  |  |  |
| --- | --- | --- |
| 名称 | 特征 | 举例 |
| 基本类型 | 表示c语言中的基本类型 | int, char short float… |
| 逻辑跳转 | 根据表达式的结果进行跳转 | if while for switch goto |
| 逻辑跳出 | 跳到或跳出逻辑结构的末尾 | continue break |
| 类型定义 | 定义新类型名称 | struct enum typedef |
| switch跳转点 | 根据switch的索引表达式跳转的位置 | case default |
| 可变参数 | 可以省略或任意长度的函数参数 | va\_list va\_start va\_arg |

除了大小写单词字母以外，C语言还包含了其他的符号。

表1.2 c语言特殊符号分类

|  |  |
| --- | --- |
| 符号 | 用法 |
| 二元运算符，根据符号两边的表达式，计算结果 | |
| + - \* / % | 数学计算（整数/浮点） |
| & | ^ << >> | 逻辑运算 |
| && || ^^ | 布尔逻辑运算 |
| < > <= >= == != | 比较运算 |
| = += &= 等 | 赋值，包括直接赋值和计算赋值，返回赋值后的值 |
| . -> | 成员访问，左边是对象，右边是访问的成员名称 |
| 前缀运算符，在被运算的表达式左边 | |
| ! | if (expr ==0) return 1; else return 0; |
| ~ | 逻辑非 |
| \* | 地址寻对象 |
| & | 对象寻地址 |
| - | 0 - expr |
| (type) | 类型强制转换 |
| ++ -- | expr += 1/expr -= 1，返回运算后的值 |
| 后缀运算符，在被运算的表达式右边 | |
| [offset] | \*(expr+offset) |
| (args) | 函数调用 |
| ++ -- | expr += 1/expr -= 1，返回运算前的值 |
| 其他运算符 | |
| expr1?expr2:expr3 | if (expr1) return expr2; else return expr3; |
| ( ) | 括号 |
| 其他符号 | |
| {} | 大括号（函数内容/结构内容/枚举内容等） |
| () | 圆括号（函数参数等） |
| # | 宏 |
| … | 可变参数 |

## 2.4 逻辑树

### 2.4.1 逻辑树的结构

前面提到，uCompiler的所有工作，从源码分析到程序生成，都是围绕逻辑树来进行的。这里的逻辑树指的是描述代码逻辑的一种拓扑结构。每个高级语言中的元素都有对应的树的节点，而元素间的包含关系（结构定义中的成员，函数中的语句）用边的链接来体现。

逻辑树中的节点没有标号，但是有特定的类型，而且大部分都有名称，以及其他的属性。

要注意，属性值的格式不固定，可能造成一种潜在的麻烦，那就是要实现多个子类来描述不同的节点类型，

幸运的是，javascript的对象的成员并不固定，所以这种结构在实现上并不会像其他语言一样，进行大量的类型转换操作。

表1.3 逻辑树结构列表

|  |  |  |
| --- | --- | --- |
| 名称 | 解释 | 结构 |
| 基本层/结构层/枚举层 | | |
| class | 结构定义 | text:定义内容，next:下一个结点 |
| var | 变量定义 | name:名称，s\_type:类型，value:赋值，  next:下一个结点 |
| func | 函数定义 | name:名称，s\_type:类型，args:参数，text:函数内容  next:下一个结点 |
| enum | 枚举定义 | name:名称，text:内容，next:下一个结点 |
| typedef | 类型重命名 | s\_type1:类型名称，s\_type2:新名称，next:下一个结点 |
| 函数层 | | |
| expr | 表达式 | text:表达式序列，next:下一个节点（可省略） |
| CONST | 常量（大写表示表达式内容，下同） | s\_type:类型，value:值 |
| VAR | 变量 | name:变量名 |
| FUNC | 函数调用 | name:函数名，args:参数 |
| … | 运算符 | 无 |
| CONV | 类型装换 | s\_type:装换类型 |
| para | 用{}包围的代码段落 | bracket:括号名，text:段落内容，next:下一个结点 |
| if | 条件语句 | cond:条件表达式，stmt:条件正确时执行的语句，  else:条件错误时执行的语句，next:下一个结点 |
| while | while循环 | cond:条件表达式，stmt:执行语句，next:下一个结点 |
| for | for循环 | s\_expr:循环开始时执行的表达式，c\_expr:条件表达式，  e\_expr:循环末尾执行的表达式，  stmt:循环执行的语句，next:下一个结点 |
| switch | 多路条件 | cond:条件表达式，text:结构内容，next:下一个结点 |
| case | 跳转点 | value:常量,为条件坐标，next:下一个结点 |
| default | 默认跳转点 | next:下一个节点 |
| break | 跳出 | b\_point:跳出点（一般为父结构的下一条语句），  next:下一个结点 |
| continue | 跳过 | c\_point:跳过点（一般为循环末尾），  next:下一个结点 |
| asm | 内嵌汇编 | inst:指令，imme:立即数，  next:下一个结点 |
| return | 返回 | value:返回值，  next:下一个结点 |
| 宏 | | |
| include | 头文件包含 | filename:文件名，text:语法树 |
| define | 宏定义 | text1:匹配文本，text2:替换文本 |
| 可变参数相关 | | |
| va\_arg\_list | 可变参数 | 无 |
| va\_start | 指向第一个参数的指针 | ap:要赋值的指针，v:参数列表 |
| va\_arg | 下一个参数 | ap:要赋值的指针，t:参数类型 |
| va\_end | 参数指针设为无效 | ap:要赋值的指针 |

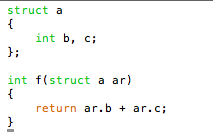


图1.2 一个给逻辑树结构举例的简单的c语言小程序

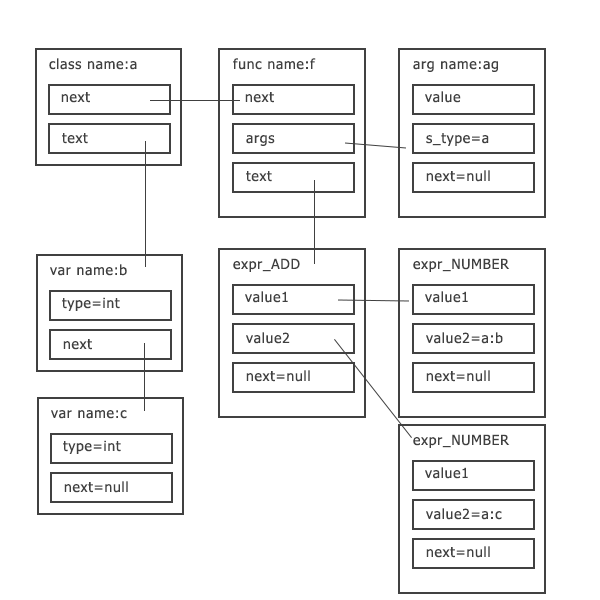


图1.3 小程序生成的逻辑树结构

这里有一个c语言的简单小程序，定义了一个类型a和函数f。uCompiler基于这个程序来构建了逻辑树结构，可以很清晰的理解这个结构。

在词义分析中生成的逻辑树，已经可以完整的表达源代码的逻辑了。然而c语言最为关键也最为复杂的运算表达式，通过第一步的词义分析，能得到由子表达式和运算符组成的序列组合。

### 2.4.2 逻辑树的生成策略

从源代码到表达结构这个转换过程中，对于源码中的任何单位元素， uCompiler都有一套通用的策略去生成建立对应的逻辑表达结构。在uCompiler程序实现的角度上看，我们读取的高级语言源码，实际上是由**单词**和**符号**排列组合构成的，而对于c和c++语言，不同的代码元素在开头的单词和符号就能体现其特征。例如类型struct name {，变量var name =/;，函数var name(等。

发现了这一特征后，uCompiler在对一个元素的分析和结构生成的开始，需要扫描(scan)得到第一个单词和符号。

i = this.spaceskip(i); //跳过空格/注释

j = this.alphaskip(i); //跳过字母/数字/下划线

word = this.text.slice(i,j); // 获得单词word

label = this.text[j]; // 获得符号label

获得了单词和符号后，是否就能初步确定这个代码元素呢？答案是不能的。举个简单例子，函数内部(no static)，代码外部，结构内部的变量定义分别为局部变量，全局变量以及成员变量。不同的状态下可能会有相同代码格式的不同代码元素，所以要考虑到状态并引入状态机来辅助词义分析。

表1.4 逻辑树生成状态机的所有状态

|  |  |  |
| --- | --- | --- |
| 状态名称 | 定义 | 可能的代码元素 |
| code | 代码的最外部 | 保留字struct（类型定义），enum（枚举定义），typedef（类型重命名），unsigned（无符号类型）  ，符号’#’（宏定义）。 |
| typedef | 类型重命名 | 一个已有的类型后面接着一个新的类型名称，注意保留字struct和unsigned |
| para | 函数内部的段落 | 分成stmt来生成，遇到}为结尾。 |
| stmt | 函数内部的语句 | 保留字包括if,while,for,switch等常用到的c语言结构，接下来判断是否为变量定义，否则跳到expr状态来检查是否为表达式。 |
| const | 通常值 | 可能是直接表示的整数，浮点数，字符，或者字符串。 |
| expr  expr类 | 表达式 | 这个比较复杂，下面会具体阐述 |
| class | 类型 | code状态中的首单词为类型时，跳转到的状态。后面根据特殊符号’{‘或者单词来判断类型定义或者变量/函数定义两种情况。 |
| enum | 枚举类的内部 | 第一个单词是枚举类中的名称，这里视为对常量的定义，类型为unsigned int。后面的符号可能为’=’,’,’或者’}’，也就是说枚举内部可能会自行赋值。 |
| type | 变量/函数定义 | 第一个单词为变量/函数名称，接下来通过判断是否为符号’(‘来辨别函数定义，’=’为变量定义的赋值，’;’为变量定义的末尾。 |
| f\_args | 函数参数 | 注意，先判断是否为可变参数…  类型名称和变量名称，后面跟着符号’,’或者’)’，前者表示后面还有参数，后者表示参数末尾。  其他符号均非法，返回syntax错误，下面遇到同样的情况也同样处理。 |
| array | 直接定义的数组 | 用{}来定义，不可修改。策略和para相同，检查’,’和’}’。 |
| fc\_args | 调用函数的参数 | 和f\_args大致相同，但是没有类型名称 |
| fp\_args | 函数指针的参数 | 同fc\_args |
| c\_code | 类型定义内部 | 定义类型内部的成员。在构造策略上是code的子集，只包括成员变量的定义。 |
| s\_code | switch内部 | 在构造策略上是para的子集，不包括变量定义。 |

在javascript里，也有switch-case这样的多重分支结构，很容易能够实现上述的状态机。这里还是要明确注意一下expr这个状态，以及表达式的结尾符。例如switch-case中的case结构，有特殊的表达式的结尾符’:’，要额外建立状态exprcase，这类子状态共同归为expr类状态。

### 2.4.3 表达式生成策略

表达式序列的构造也是通过单词-符号之间的交替识别进行的。在词义分析中，循环执行如下几步，直到遇到末尾符为止。

1. 检查是否为左括号’(‘，分为**子表达式**或者**类型转换**两种情况，检查括号内部的结构，若为(typename)或者(struct typename)这样的结构，则确定是类型转换，重复步骤1，否则以expr状态递归调用分析函数。
2. 检查是否为数字或者单/双引号。分别代表了数字常量/字符常量和字符串常量。
3. 检查是否为单词，单词是否已经定义（变量/函数），若未定义，返回uncleared，若后面跟着’(‘符号，则为函数调用，否则为变量取值。
4. 检查是否为运算符，最长的运算符优先匹配(<<=优先,<<其次，最后才是<)。

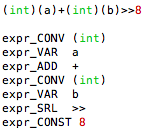


图1.4 表达式和元素序列

这样就能初步获得表达式的顺序序列，表达式的逻辑结构是接下来语义分析的任务。在语义分析里，需要将表达式的线性排列，分析生成跟逻辑树相同的树状逻辑结构。通用的方法是自顶向下，通过最优先运算符来确定表达式的结构，进而分支并递归处理子表达式，获得子表达式的逻辑树，最后结点链接。运算符分为二元运算符，前缀运算符，后缀运算符以及三目条件运算符。

1. 二元运算符：运算符的左边和右边都是子表达式，形成expr oper expr的结构。例如+,-,\*,/,&(and),|(or)，成员相关（.,->）等。
2. 前缀运算符：在子运算符的左边，形成oper expr的结构，例如地址相关（&,\*），非运算（~,!）等。
3. 后缀运算符：和前缀运算符相反，在子表达式的右边。例如数组元素[offset]，++,--等。
4. 三目条件运算符：和if-then-else的形式类似，为cond ? expr1 : expr2的结构。cond不为0返回expr1,cond为0返回expr2。

运算符可能会有多种含义，例如&的取地址和与运算，\*的取数据和乘运算，++,--的前缀后缀形式等。运算符的歧义问题迫切需要严肃对待，当发生运算符歧义时，一般是在构造运算符的逻辑树中发现的，这个时候当前构造逻辑树是非法的。这里有处理这类问题的方法。

1. 检查当前逻辑树的分支运算符是否为多义运算符，换其他的含义并从新构造新的逻辑树。
2. 可能是上层的运算符出现问题，将错误反馈给上层。

## 2.5 词义分析

传统的词义分析是将已字符顺序排列成单词+符号的高级语言转换成token流的过程。uCompiler的词义分析跟一般词义分析一样，是从左向右扫描识别单词和符号并将单词和token的对应关系放在特定的数据结构中（大多都用的是哈希表）。但是uCompiler的词义分析额外又做了语义分析的大部分工作，包括建立了大部分的程序逻辑。这样，在别的编译器的词义分析只生成了token对应和序列时，uCompiler已经开始构建逻辑树的框架了。

为了完成uCompiler中的词义分析，在javascript定义了叫做w\_ana的对象，主要功能是将源码给w\_ana，然后生成初步的逻辑树et。

表1.5 语义分析对象的主要功能

|  |  |
| --- | --- |
| 主要功能 | 描述 |
| w\_ana.spaceskip  (id) | 给定源码坐标，跳过之后的空格/换行/注释，到达终点坐标 |
| w\_ana.alphaskip  (id) | 给定源码坐标，跳过之后的字母/数字/下划线，到达终点坐标 |
| w\_ana.analysis  (id, status, prev) | 给定源码坐标，状态机状态，前缀，生成逻辑树。这是w\_ana最重要的功能。 |

注意analysis中的前缀prev，这个前缀信息是为了描述“在C语言中的结构/函数/括号的内部”的。其描述形式和c语言一致。

1. c结构类型的内部，”StructName::”
2. 函数内部，”FunctionName::”
3. 函数实现内的括号内部，每个括号都设有一个id，这样用’b’ + id来描述括号名称。例如”func1::b1::b3”，表示在func1函数内的b1括号内的b3括号内。

w\_ana将获得的源码信息储存到另一个对象rec中，这个对象是专门管理存取信息的数据结构的。

表1.6 记录对象的主要功能

|  |  |
| --- | --- |
| 主要功能 | 描述 |
| rec.setVariable(var, prev)  rec.getVariable(var) | 添加/查找变量，prev为上述的前缀信息。注意函数/枚举成员也视为变量。 |
| rec.setType(type)  rec.getType(type) | 添加/查找类型，c语言通过两种方式来获得新的类型名称：struct {}和typedef。 |
| rec.addBracket | 添加一个新的括号，返回括号id |
| rec.getLine(id)  rec.getColumn(id) | 获得对应坐标的行数/列数，这在反馈错误信息时会用到。 |

词义分析中，可能会检测到错误，像其他编译器一样，这时需要立刻将错误信息打印到屏幕上，然后决定是否继续进行词义分析。

表1.7 词义分析可能出现的编译错误

|  |  |
| --- | --- |
| 错误 | 注释 |
| expected char (after stmt) | （对于对应的语句）找不到匹配的符号 |
| syntax at char | 不合法的符号 |
| word uncleard | 未定义的单词 |
| expr is not a constant | 在需要常量的地方使用非常量 |
| unknown operator at char | 表达式内部使用未定义的运算符 |
| a label can only be part of a statement and a declaration is not a statement | 在switch内部不能声明变量 |
| include : ‘file.h’ not found | 未找到头文件 |

每个错误信息都有对应的坐标id，当发生错误后若需要进行后续处理，则后面的情况可能未知，所以需要进行括号匹配，到达下一个括号外的位置，才能继续运行。

## 2.6 语义分析

在前面的词义分析，我们得到了逻辑树，现在我们需要对逻辑树进行一些扩展工作，这些交给语义分析的对象g\_ana来完成。

表1.8 语义分析对象的主要功能

|  |  |
| --- | --- |
| 主要功能 | 描述 |
| g\_ana.analysis(et, prev) | 语义分析 |
| g\_ana.createExprTree  (sequence, left, right) | 将表达式序列sequence中从left到right的子序列，转换成逻辑树并返回。 |
| g\_ana.solve(et, prev) | 尝试去计算给定逻辑树，若为可计算得到的常数，则设为该常数值。 |

上面提到了，词义分析得到的逻辑树并不完备。语义分析要做的任务，除了表达式的结构构造以外，还有对代码语法的检查。包括表达式语法，类型相关，循环结构,switch结构中的跳出(break)和继续(continue)跳转等。

表1.9 语义分析过程中可能发生的编译错误

|  |  |
| --- | --- |
| 错误 | 解释 |
| expected expression after op token | 运算符后需要表达式 |
| expected expression before op token | 运算符前需要表达式 |
| expected ‘?’/’:’ | 条件运算符缺省符号 |
| could not convert from type1 to type2 | 类型转换不合法 |
| fixed-point types not supported for this target | 访问非结构类型的成员 |
| expected struct number after variable | 结构类型不包含该成员 |
| invalid type argument of '->' | 非指针使用->符号 |
| request for member in something not a structure or union | 指针指向的类型非结构类型 |
| invalid operands to binary op  (have type1 and type2) | 对type1和type2的运算op是非法的 |
| wrong type argument to increment | 对结构类型使用++,--运算符 |
| array subscript is not an integer | 数组运算符的索引值不是一个整数 |
| subscripted value is neither array nor pointer nor vector | 数组运算符的基地址不是一个指针 |
| used struct type value where scalar is required | 结构类型不支持数字运算 |
| wrong type argument to bit-complement | 非整数类型不支持位运算 |
| too few arguments to function | 调用参数比函数参数要少 |
| too many arguments to function | 调用参数比函数参数要多 |
| case label not within a switch statement | case语句不在switch结构里 |
| default label not within a switch statement | default语句不在switch结构里 |
| break statement not within loop or switch | break语句不在循环结构或switch结构里 |
| undifined instruction | 内嵌汇编未定义指令 |

对类型的检查分为类型运算，类型赋值，类型参数三大类。类型运算的标准很显而易见，浮点数不能位移，结构类型不能数字逻辑运算，基本类型不能访问成员，指针和浮点不能直接进行二元运算等。

表达式的结构错误在创建表达式的结构树时能够发现，一般情况都是二元运算符缺少左子式或右子式，前缀/后缀运算符缺少子式，运算符相连等结构错误。

对跳转的检查，通过在循环结构/switch结构的开始对对应prev前缀进行标记，根据prev来寻找对应的结构，若未找到则返回跳转错误。这些标记需要rec对象来维护。

表1.10 记录表支持语义分析的功能

|  |  |
| --- | --- |
| 主要功能 | 描述 |
| rec.setLoopLabel(et, prev)  rec.getLoopLabel(name) | 添加/查找循环标记，直接将相应的逻辑树对象设为标记。 |
| rec.setSwitLabel(et, prev)  rec.getSwitLabel(name) | 添加/查找switch标记 |

除了这些重点错误以外，还有参数不匹配，asm参数不合法等其他语法错误，参数匹配通过一一比对来检查，要注意的是可扩展参数va\_args，当被调用方的参数出现时立刻停止比对。

a1 = et.args, a2 = et.value.args

while (a1 != null && a2 != null) {

if (this.conv(a1.s\_type, a2.s\_type))

this.error(“invalid operands to binary op (have”+a1.s\_type+”and”+a2.s\_type);

if (a2.type == “va\_args”) break;

}

if (a1 == null && a2 != null) this.error(“too many arguments to function”);

if (a2 == null && a1 != null) this.error(“too few arguments to function”);

## 2.7 程序生成

uCompiler始终围绕着逻辑树来进行，由于我们已经得到了完备的逻辑树了，就可以直接访问逻辑树来获得目标的执行指令，而无需中转和链接等操作。这是uCompiler的优势，能够用最简洁的三步来完成要做的工作，奥卡姆剃刀般的干净利落。

最后要完成编译过程的最后一步，目标程序的生成。uCompiler通过自上而下的访问逻辑树，对于每个节点，在第一访问和访问子树完成返回进行指令生成。

表1.11 c语言结构对应的程序生成方法

|  |  |  |
| --- | --- | --- |
| 结构 | 例子 | 生成方法 |
| 常量 | 1, 0.1, ‘c’, “string” | 直接用（立即数/数据段地址）来表示 |
| 变量 | int I, j, char\* c | 根据变量地址从内存读写对应数据 |
| 运算 | i+j, ~i, expr1+expr2 | 1. 常量参与的运算，用立即数指令来完成。 2. 先计算子表达式的值，放在寄存器里，再用指令运算。 3. 二元运算若有必要，将左表达式的值暂存于栈中，计算右表达式后用栈读取运算指令。 4. 布尔与或运算的断短路问题，通过b指令跳过之后的运算。 |
| 赋值 | var = expr | 将表达式的值计算于寄存器，用s指令写入内存。 |
| 成员 | struct.number  pointer->number | 结构地址+成员偏移量  指针地址+成员偏移量 |
| 寻址 | \*pointer | 计算pointer的表达式，在根据结构读出数据 |
| 取值 | &var | 根据编译信息直接得到 |
| 调用 | func(args) | 将参数压栈，JSR跳转至所调用函数，返回出栈 |
| 条件 | if (expr) stmt  if (expr) stmt else stmt  while (expr) stmt  for (e1;e2;e3) stmt | 1. if：计算expr，betz跳转至下一条语句。 2. if-else：计算expr，betz跳转至else语句，then语句末尾跳转至下一条语句。 3. while：跳转至循环末尾，末尾计算expr，bnez跳转至循环头部。 4. for：同while，循环前执行e1,循环末尾前执行e3。 |
| switch | switch(expr) {  case 1:  case 2:  …  default:  } | 1. 将所有case的值进行排序。 2. 得到最大值，若不超过一定阈值，则可以用entrylist来实现，否者只能用通过二分法的策略来查找入口。 |
| 跳转 | break, continue, goto | 直接用跳转指令 |
| 返回 | return value | 计算value于寄存器ra，LEV 指令返回 |
| 汇编 | asm(inst,imme) | 直接用对应的指令 |

不同的结构有不同的生成策略，这些生成策略和以往的编译器大体一致，包括栈暂存，条件的否跳转，循环的jump-content-branch结构，在gcc等通用编译器里都有体现。

负责实现指令生成的对象叫做i\_gen，主要功能比较多一些。

表1.12 指令生成对象的主要功能

|  |  |
| --- | --- |
| 主要功能 | 描述 |
| i\_gen.generate(et, prev, sp\_offset) | 通过给定的逻辑树，来生成v9-cpu的机器指令，要注意栈操作造成栈地址的偏移，需要记录下来 |
| I\_gen.generate\_n  (struct, number, prev, sp\_offset) | 生成成员变量的值 |
| I\_gen.generate\_p  (et, prev, sp\_offset, offset) | 在寄存器ra中存储着类型的基地址，生成成员变量的值，注意地址可能产生额外偏移offset |
| I\_gen.save(et, prev, sp\_offset) | 保存ra的值到逻辑树对应结构中 |
| I\_gen.save\_n  (struct, number, prev, sp\_offset,offset) | 保存到成员变量中 |
| I\_gen.save\_p  (et, prev, sp\_offset, offset) | 保存到指定类型地址的成员变量中 |
| I\_gen.addr(et, prev, sp\_offset) | 取得逻辑树对应的内存地址 |
| i\_gen.generate\_bin\_ab  (et1, et2, prev, sp\_offset) | 将et1的值生成到寄存器ra中，et2的值生成到寄存器rb中 |
| I\_gen.generate\_bin\_ba  (et1, et2,prev,sp\_offset) | 与generate\_bin\_ab相反，将et1的值生成到寄存器rb中，et2的值生成到寄存器ra中 |
| I\_gen.div\_table(ets, default, l, r) | 生成通过二分查找的算法来完成switch-case的跳转的程序。ets表示每个case语句的逻辑树,default表示default语句的逻辑树 |
| i\_gen.determine\_pc | 确定每个跳转指令的地址 |
| i\_gen.find\_main(et) | 寻找main函数对应的逻辑树 |
| I\_gen.generate\_b  (et, prev, value, sp\_offset) | 条件跳转生成 |

各个功能之间根据逻辑树的类型相互调用。generate在生成二元运算符’.’和’->’时，分别调用generate\_p和generate\_n，generate\_n的struct结构为’.’和’->的运算时也要调用generate\_p和generate\_n。save同理。Generate在计算其他二元运算符时先将两个子式的值计算到寄存器ra和rb中，再用寄存器计算指令来进行主要的计算。

在switch结构的开始，先计算条件表达式，根据其值来跳转到相应的case。这里通过二分构造跳转列表来实现快速跳转。div\_table算法根据当前区间的大小进行不同的策略。

1. case的数量为1时，用BE来判断是否符合case并跳转，再用JMP直接跳转到default区域。
2. case数量为2时，分别用BE来跳转到相应的case，最后跳转到default.
3. case数量大于2时，必能得到一个中间值，并将所有的case分成左右两边。大于这个中间值跳转到右边的跳转列表，否则继续运行左边的跳转列表。

如果跳转的地址不确定，怎么办？指令会暂存要跳转到的点对应的逻辑树，最后通过determine\_pc来确定逻辑树对应的指令地址。不过if,while和for等条件结构和循环结构通过记录B指令对象的当前地址，然后与要跳转的地址相减得到。

inst = this.inst\_g(“B”, this.insts.length);

this.generate(this.stmt, prev, sp\_offset);

Inst.imme = (this.insts.length-inst.imme-1)<<2;

注意，第一次生成inst指令，将立即数imme设为当前指令数量，生成完stmt的指令后，用指令数量去减这个立即数，可以得到跳过stmt需要的指令位移，乘以4得到地址位移。

条件跳转生成generate\_b根据et的值和value进行比较，当et等于value时进行跳转，否则继续运行，并返回跳转指令。这个功能主要是为了实现进行逻辑与或运算时的断路/短路跳转。条件跳转主要处理这几种情况。

1. 逻辑与或，根据value的值的不同，进行不同的生成方法。逻辑与和value=1时，逻辑或和value=0时，进行断路跳转，即中间判断符合条件并跳转。逻辑与和value=0时，逻辑或和value=1时，进行短路跳转，即中间判断不符合条件并省略之后的指令跳到下一行。
2. 大小比较，相等判断，计算左式的值和右式的值，通过自带比较的B指令来进行跳转。
3. 其他情况，计算表达式的值，用和0比较的B指令来跳转。

## 2.8 编译优化

关于编译优化，所做的工作并不多，可简单的阐述一下。

1. 立即数优化，当表达式必定返回固定值的情况下可视为常数，在指令生成时用I类立即数指令。
2. 二元运算符计算之前对两个子式进行判断，若不用到rc寄存器，可以先将另一个子式的结果暂存到寄存器rc中，待这个子式计算完毕后从rc中取到rb，最后通过寄存器计算指令来计算。取代了关于栈的两条访存指令，效率会得到有效的提升。

# 第3章 模拟器

## 3.1 v9-cpu介绍

v9-cpu是基于常见的cpu系统上的一种简化设计，虽然没有硬件实现，但是有用c语言实现的模拟系统。读取v9-cpu自带的指令集的程序，就可以在这个cpu系统上运行程序了。v9-cpu现阶段的主要用途是教学实验和练习用。

v9-cpu的寄存器组由7 个32位寄存器和2个64位浮点寄存器构成。

表2.1 v9-cpu的寄存器

|  |  |
| --- | --- |
| 寄存器名称 | 描述 |
| ra, rb, rc | 通用寄存器 |
| rf, rg | 浮点寄存器 |
| sp | 当前栈指针 |
| pc | 程序计数器 |
| tsp | 栈顶指针 |
| flags | 状态寄存器 |

Flags中的每个位都有不同的用途，通常我们把这些位叫做标志位。

表2.2 v9-cpu状态寄存器中的标志位

|  |  |  |
| --- | --- | --- |
| 标志位 | 位长 | 描述 |
| user | 1 | 1为用户态，0为内核态 |
| iena | 1 | 中断使能 |
| trap | 1 | 异常/错误编码 |
| paging | 1 | Page mode使能 |

### 3.1.1 指令

v9-cpu指令集中的一条指令有32位，由8位类型位和24位立即数位构成。由于只有3个通用寄存器，无需编码并储存在指令信息中，类型位将对不同寄存器进行操作的指令编码为不同的类型。

不过作为取代寄存器编码的代价，指令的类型数量却大大的增多，所以本文只对指令集分为大的类别来讨论。

表2.3 v9-cpu指令分类

|  |  |  |
| --- | --- | --- |
| 类别 | 特征 | 举例 |
| 系统指令 | 系统中断，对sp和pc 的直接操作，对内存连续一段的操作 | HALT,ENT,JSR, MCPY |
| Load指令 | 从内存中读取数据，可以是任意基本类型，可以读取到ra,rb,rf,rg中。以及将立即数存到寄存器里。 | LL, LBL，LLS(short),LLB(byte),LG, LX, LI, |
| Store指令 | 将ra,rf中的值，储存到内存中。可以以任意基本类型的形式存储。 | SL, SLS, SLB, SG, SX |
| Branch指令 | 根据寄存器值来跳转 | BZ, BNZ, BLT, BLE, BGT,BLTF |
| 类型转换指令 | 32位整数与64位浮点之间的转换 | CID,CUD,CDI,CDU |
| Misc指令 | 涉及到特权级的切换，以及栈的操作 | CLI,PSHA,POPA |
| Math指令 | 浮点运算，包括指数对数，三角函数，取整等 | LOG,FLOR,SIN |

### 3.1.2内存

V9-cpu的默认内存大小为128MB，可以通过终端调用时附带参数选项”-m XX”来自行设置内存空间。

在TLB页表中设置了4个页转换表，每个大小为1MB。分别对应用户/内核态和读/写两种情况。

表2.4 v9-cpu的页转换表

|  |
| --- |
| 页转换表 |
| kernel read page translation table（内核读页转换表） |
| kernel write page translation table（内核写页转换表） |
| user read page translation table（用户读页转换表） |
| user write page translation table（用户写页转换表） |

V9-cpu的模拟程序，设置了两个指针tr，tw,来切换用户态和内核态的页表。

tr/tw[page number]=phy page number //页帧号

还有一个tpage buffer array, 保存了所有tr/tw中的虚页号，这些虚页号是tr/tw数组中的index

tpage[tpages++] = v //v是page number

### 3.1.3 输入与输出

众所周知，X86类的系统通过INT+中断号来中断实现，其中包括输入中断号和输出中断号来支持I/O功能。而v9-cpu的I/O实现通过特别设定的两条指令BIN和BOUT来支持。

表2.5 v9-cpu的IO相关指令

|  |  |  |  |
| --- | --- | --- | --- |
| 指令 | ra寄存器 | rb寄存器 | 描述 |
| BIN | 端口号 | 写入的字符 | 写入一个字符，保存到rb里 |
| BOUT | 端口号 | 写出的字符 | 写出rb的字符 |

如果在用户态执行BOUT指令，会发生FPRIV异常。如果iena为1，那么串口输入一个字符后会产生FKEYBD中断。

### 3.1.4 中断和异常

cpu系统中硬件实现的中断和异常一般都有寄存的信号向量作为支持。同样的，v9-cpu额外的设置了中断向量ivec，发生异常或中断时，通过ivec和中断向量页表，去查找并跳转到中断处理的入口。

表2.6 v9-cpu的中断向量

|  |  |
| --- | --- |
| 中断向量 | 描述 |
| FMEM | 物理地址无效 |
| FTIMER | 计时中断 |
| FKEYBD | 键盘输入中断 |
| FPRIV | 越权指令（多为用户态调用内核态指令） |
| FINST | 无效指令 |
| FSYS | 软中断 |
| FARITH | 计算中断 |
| FIPAGE | Opcode获取时发生页表错误 |
| FRPAGE | 读数据时发生页表错误 |
| FWPAGE | 写数据时发生页表错误 |
| USER | 用户态异常 |

## 3.2 uMonitor模拟器

### 3.2.1 分析

以v9-cpu的设计作为框架，主要实现并能运行的还是v9-cpu的模拟器，名称叫做uMonitor。uMonitor是一个能够仿照v9-cpu的运行机制，来运行基于v9指令集的程序，并能体现cpu运行时的所有信息的模拟软件。v9-cpu可以运行由v9指令集构成的操作系统指令程序，如xv6和ucore。并且，以uMonitor的由高级语言编写的软件特性，将c语言源码放在不同的主机上编译生成，就可以跨平台的模拟运行v9-cpu。另外，uMonitor还有媲美于通用调试软件gdk的调试能力，能够设置各类断点以及形象化的输出cpu和内存信息和程序运行信息。

uMonitor具有通常模式(normal mode)和调试模式(debug mode)两种工作模式。在通常模式下，uMonitor主要用于启动基于v9指令集的程序，所以只需指定程序路径。而在调试模式下，uMoniter需要获取源码信息来辅助调试，所以只有执行程序是不够的，还需要处理过源码，对uMoniter易于读取识别的调试辅助文档DML。

所以，对uMonitor的理解，对v9-cpu的理解和使用来讲至关重要。

### 3.2.2运行流程

uMonitor启动时，首先通过c语言的file IO库来加载执行程序os以及调试文档os.dml，然后通过分析dml文件，识别源码中的单词，建立信息表等一系列关于调试的初始化的流程，然后初始化cpu状态和内存状态，将指令写入指令区域。然后部署函数调用栈，断点监视列表，并模拟运行cpu，在运行的同时，根据当前指令来更新函数调用栈和变量地址表等调试模块。当触发断点时将dbg值设为1，并进入命令调试模式。

uMonitor具体执行流程如图所示。

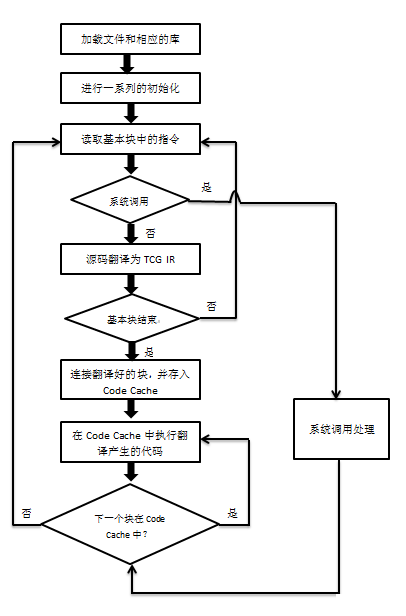


图2.1 v9-cpu的运行流程

描述如下：

1. uMonitor启动，设置读取文件的c FILE指针，若指针为NULL（未找到文件），退出。
2. 读取DML文档所提供的数据，获得所有函数，类型，枚举，常量，变量的信息，存入信息表。
3. 根据LEV指令和JSR指令，确定每个函数的入口。
4. 初始化内存，将指令写入内存。
5. 初始化断点列表，调用栈等调试工具。
6. 进入cpu运行主循环。运行一条指令。
7. 断点查询，一般断点是否符合地址，条件断点的表达式值是否为1，监视断点监视的变量是否更改，都会触发断点暂停。断点暂停时将dbg值设为1.
8. 当dbg值为1时，进入调试命令处理的循环，进行命令读取和信息的输出，当接收到S类命令（单步）和C类命令（继续）时继续运行。
9. 调整pc和sp寄存器的值，保证pc寄存器的值四对齐，sp寄存器的值八对齐。
10. 若（6），（7）步发生异常，或者（6）系统调用中断，则处理中断。若未建立中断向量表，则直接报错退出，否则跳到中断入口。
11. 识别JSR和LEV指令，更新函数调用栈(backtrace)。
12. 跳到（6）

下面详细介绍uMonitor的调试相关的辅助工具。

### 3.2.3 辅助工具

uMonitor为了能够达到gdb的效果，设计了一些调试工具，其中包括：

1. 源码的信息表，储存c语言所有源码元素信息的表项。注意枚举类中的成员视为常量来处理。

表2.7 uMonitor模拟器的源码信息表

|  |  |  |
| --- | --- | --- |
| 名称 | 描述 | 结构 |
| var\_t | 变量信息表 | name:名称，type:变量类型 |
| const\_t | 常量信息表 | name:名称，value:值 |
| func\_t | 函数信息表 | name:名称，addr:入口  line:起始行，end:末尾行  args:参数列表（类型为var\_t）,  locs:局部变量列表（类型为var\_t） |
| stu\_t | 类型信息表 | name:名称, size:所占空间大小  nums:成员变量列表（类型为var\_t） |
| stmt\_t | 源码语句信息表 | start:指令入口，end:结束地址  text:源码文本 |

（2）断点列表，用于保存通过命令添加的各类断点，分为通常断点和条件断点两种。通常断点只需用断点地址来表示，而条件断点需要分析条件表达式，并构建逻辑结构来表示。

断点表达式包括数字常量（整数，浮点数），变量名称，常量名称，以及+,-,\*,/,&,|,~运算符和&&,||布尔运算符，以及表示变量是否更改的特有后缀运算符!。

断点结构中有两个整数lvalue和rvalue，以及表示整数含义的两个标记ltype和rtype。这两个整数根据对应标记的不同，拥有不同的含义。

表2.8 条件断点的逻辑结构

|  |  |
| --- | --- |
| type | value |
| 0 | 一个有符号的32位整数。表示表达式中的数字。 |
| 1 | 指向一个断点结构，这个断点结构用来表示全局变量，lvalue为变量信息表的地址，rvalue为变量在内存中的地址。 |
| 2 | 指向一个断点结构，这个断点结构用来表示局部变量，lvalue为变量信息表的地址，rvalue为相对于当前栈帧的偏移地址(offset)。 |
| 3(rtype) | 指向一个断点结构，这个断点结构用来表示成员变量。lvalue指向成员变量所在的类型的断点结构，rvalue.lvalue为成员变量所在的信息表的地址，rvalue为成员变量相对于类型基地址的偏移。 |

计算这个断点表达式的值，通过自上而下的访问这个结构，自下而上的搜集子式的计算值，进行计算并反馈给上级。

（3）函数调用栈。维护函数调用的后向路径（backtrace），栈中内容包括前面调用的函数的信息表id,调用下一个函数所在的代码位置和调用时的栈帧顶部。

检测机器指令JSR和LEV，出现这两个指令时，说明执行了函数调用和跳出，对函数调用栈进行push和pop操作。

在函数调用（JSR）中，通过JSR指向的调用入口，在函数信息表中查找对应的函数id，将得到的信息加入函数调用栈。

在函数跳出（LEV）时，将栈顶的内容弹出。

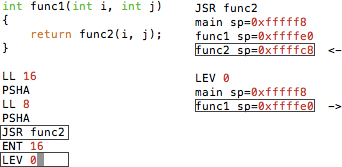


图2.2 函数调用栈的工作原理

### 3.2.4 信息处理

在该小节，我们对uMonitor如何处理给定的调试文档进行简单的介绍和分析。DML调试文档是为了辅助uMonitor的调试功能而设计的，其结构与我们所熟悉的XML相同，对每一行代码都设有一个单元。每个单元都有以下固定的格式。

<stmt start=0 end=5>

<text> int I=1,j=2;</text>

<emi>

0x00000000 0xXXXXXXXX ENT -8

0x00000004 0xXXXXXXXX LI 1

0x00000008 0xXXXXXXXX LL 4

0x0000000c 0xXXXXXXXX LI 2

0x00000010 0xXXXXXXXX LL 0

</emi>

</stmt>

其中start和end分别表示这行代码对应的指令地址的入口和出口，text表示这行代码的文本内容，这些信息都会被uMonitor读入stmt\_t信息表中。

剩下的emi单元中的内容，显示的是机器语言格式，是为了使用者便于阅读而设计的，uMonitor并不读取其中的内容，这些内容可以通过分析机器指令而得到。这里声明了两个局部变量i和j，为32位的有符号整数格式。ENT -8表示sp-=8，为局部变量腾出对应的空间。剩下的就是初始赋值操作。

一开始，我们只能从DML 中获得stmt\_t的内容。在读取了text内容后，通过分析源代码，可以得到剩下的信息。通过状态机，来分析源代码获得有用的信息表。

状态机的当前状态用一个16位的整数来描述，这个状态数的后8位表示括号层数，前8位表示当前状态。

表2.9 dml分析状态机的所有状态

|  |  |
| --- | --- |
| 状态(hex表示) | 描述 |
| 0xXXDD | DD表示当前括号匹配的层数，括号包括圆括号和大括号，视为同一种括号来匹配。 |
| 0x01XX | 表示当前在字符/字符串内部 |
| 0x02XX | 表示当前在函数参数声明的内部 |
| 0x04XX | 表示当前在函数代码的内部 |
| 0x0800 | 表示当前在类型构造的内部 |
| 0x1000 | 表示当前在枚举构造的内部 |

状态机工作时，从原代码顺序扫描，将字符作为输入。遇到’(‘,’{‘时括号层数加1，遇到’}’,’)’时括号层数减1。状态之间通过一定的规则进行转移。

1. 在0x0000状态下，输入字符为’(‘时，可判定为函数的参数声明，故转移到0x0201。
2. 在0x0201状态下，输入字符为’)’时，函数参数的声明结束，转移到0x0200，等待函数的代码实现。
3. 在0x0200状态下，输入字符为’{‘时，说明开始实现函数的代码部分了，故转移到0x0401，表示在代函数码的内部。
4. 在0x0401状态下，输入字符为’}’时，函数的代码实现已经完成，转移到0x0000。
5. 在0 x0000状态下，输入字符为’{‘时，检查前面的结构，若为”struct name {“的格式，将状态转移到0x0801，若为”enum {“的格式，将状态转移到x1001。
6. 在0x0801状态下，输入字符为’}’时，表示c类型的实现已经完成，转移到0x0000。
7. 在0x1001状态下，输入字符为’}’时，表示枚举的实现已经完成，转移到0x0000。
8. 在0x0201状态下，输入字符为’,’时，表示函数参数之间的间隔符号’,’，获得之前的函数参数的信息。
9. 在0x0000, 0x04XX,0x08XX,0x10XX状态下，输入字符为’;’时，表示一条语句的结束符号’;’。这条语句可能是变量声明语句，检查之前的格式是否为var name;或者var name = value;的格式，来获得变量信息。根据状态的不同，来确定是否为全局/局部/成员/枚举变量。

通过这类分析识别源代码的方法，来获得信息表中的数据。

表2.10 状态机使用方法

|  |  |  |
| --- | --- | --- |
| 结构 | 所在状态 | 识别方法 |
| 函数参数 | 0x0201 | 检查’,’符号，获取前面两个单词，第一个是参数类型，第二个是参数名称。  （遇到指针’\*’时，可以跳过’\*’前面的空格。） |
| 局部变量 | 0x04XX（非0） | 检查’,’,’=’,’;’符号，获取前面两个单词，若能匹配第一个单词的类型，第二个单词是合法的变量名称（数字字符下划线，第一个字符是字符或下划线），那么这就是个变量定义。  ‘,’之后只声明了变量名称，所以只需检查一个单词。 |
| 结构内容 | 0x0801 | 检查’,’,’;’符号，之后和检查局部变量相同。 |
| 全局变量 | 0x0000 | 与检查结构内容相同。 |
| 常量 | 0x0000 | 检查’const’单词，并匹配之后的两个单词（类型和名称），一个’=’符号，一个数字，字符或字符串。 |
| 枚举内容 | 0x1000 | 视为常量，和检查变量的方式相同。 |

要补充一点，c语言的函数声明可能会影响状态机对函数的识别顺序，所以需要判断参数声明后是否有’;’，将状态从0x0200还原到0x000

### 3.2.5 调试命令

uMonitor有丰富的调试命令，风格上更接近于gdb。

表2.11 uMonitor的调试命令

|  |  |
| --- | --- |
| 命令 | 功能 |
| c | 继续运行，直到到达第一个断点停止。 |
| s | 单步运行，运行一行代码 |
| si | 单步运行，运行一条指令 |
| sc | 单步执行到下一行，跳过调用 |
| b addr  b name  b name+line | 按照指令的相对地址，设置断点  按照函数名称设置断点  按照函数名称+行号，设置到所在函数的行。 |
| b if expr  b if expr! | 设置当expr值为1时暂停的条件断点  设置当expr的值改变时的监视断点 |
| bl  bd index  bdall | 查看所有断点  删除对应序号的断点  删除所有断点 |
| bil  bid index  bidall | 查看所有条件断点  删除对应序号的条件断点  删除所有条件断点 |
| bt | 查看函数调用栈 |
| ir  il  ig | 显示所有寄存器的信息  显示函数内局部变量和参数的信息  显示所有全集变量的信息 |
| x /48i 0x0080  x /32i pc  x /64x 0x0100  x /32x sp | 显示0x0080后48bit的命令  显示pc后32bit的命令  显示0x0100后64bit的数据(hex显示，下同)  显示栈顶后32bit的数据 |
| h | 显示帮助文档 |
| q | 退出uEmulator |

uMonitor对调试命令的处理，是在cpu主循环中对调试处理的循环内部进行的。每次循环先读指令的第一个字母，从而判断指令的类型。接着根据不同指令类型决定接下来的操作。

1. 读取到C类命令时，跳出调试循环，cpu继续运行。
2. 读取到S类命令时，设置step标记并继续运行cpu。下一次主循环根据step标记和pc地址进行判断是否进入调试模式。单步运行代码的场合，判断pc地址是否到达语句的指令入口，而跳过调用的单步，需要记录下一行的位置，到达下一行的指令入口才能暂停。
3. 读取到其他命令时，根据接下来读取的字母来决定要做的操作。特别的，当读取到条件断点命令时，需要对表达式进行解析构建逻辑结构。

# 第4章 实验测试

## 4.1 测试准备

至此，编译器uCompiler和模拟器uMonitor的设计与实现已经完成，若要保证整个系统能够正常运行，测试是必不可少的一环。按照以往的经验，如果直接测试一项大的工程，是个很盲目的计划，也会被搞得一头雾水。将大的问题拆分成小的问题逐步解决，是我们经常使用的策略。

### 4.1.1 单元测试

对于刚完成的编译器和模拟器，不能够确保任何模块都是准确且正常运行的，对每个模块的单元测试是必不可少的一环。我们先来回顾一下编译器编译程序的所有环节。

原代码->词法分析->(逻辑树)->语法分析->(完备的逻辑树)->代码生成->执行程序

整个流程存在着两个关键点，将原高级语言中的符号单词排列进行分析识别，以及确保指令生成的正确性。先从最简单的样例开始，逐步检查并解决发现的问题和错误。

（1）对c语言结构能否正确识别进行检查，包括类型定义，函数定义，函数参数定义等基本的代码结构能够正常识别并编译。

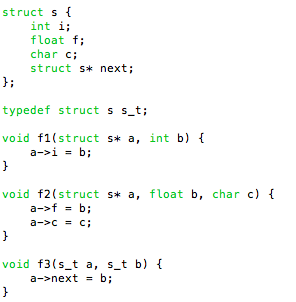


图3.1 测试样例1

（2）对c语言特有的函数指针变量的检查，包括函数指针变量的声明，赋值，调用等基本用法。

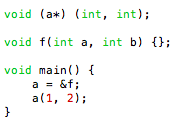


图3.2 测试样例2

（3）对if, for, while, switch-case等结构进行检查，同时对字符输出print系列进行检查。



图3.3 测试样例3

1. 在能够输出结果到屏幕的基础上，对数组访问和表达式处理进行检查。

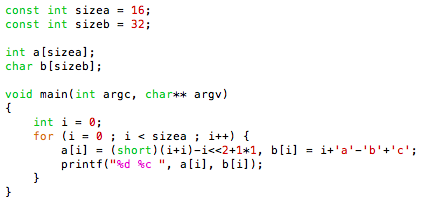


图3.4 测试样例4

确保上面的测试样例能够通过，那么编译器就非常的可靠了。

## 4.2 操作系统测试

### 4.2.1 测试方案

对xv6和ucore操作系统的测试分为以下几个步骤。

（1）通过uCompiler来编译xv6和ucore的原代码，生成机器指令的汇编语言。由于Javascript对二进制文件的读写较为麻烦，所以通过生成中间的汇编指令，然后再用转译工具翻译为二进制的机器码文件。

（2）中间汇编语言的每一条指令后面都有注释，其中包括和源代码相关联的调试信息。转移工具在翻译中间汇编语言的同时，搜集注释中的这些信息，生成dml文件。

（3）模拟器读取执行程序和dml文档，开始运行调试。

### 4.2.2 中间转译器uTranslator

众所周知，文本格式的汇编语言是不能被cpu直接识别的，就算是cpu的模拟器，也要严格按照硬件的标准来规范行为，不能内设翻译器来翻译汇编语言。这时需要设计能够将汇编语言翻译成机器指令的工具，将这个工具命名为uTranslator。

中间文件分为3个区域，分别是header，text和data。header表示可执行程序的标志头，按照c语言的结构类型的形式可以体现。

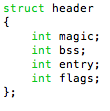


图3.5 可执行程序的标志头对应的c类型格式

text区域包含了程序的汇编指令，跟.S文件的语法一样，’#’后面的文本视为注释，data区域包含了内存data段的内容，data段在这里只用来存储字符文本，其他类型的常量通过v9-cpu立即数相关的指令来参与计算。

# 第5章 实验过程中遇到的问题及解决方法

问题一：完成Javascript版本的编译器后，无法将文件写入本地。

原因：在8.0之前版本的IE浏览器，能够支持activeX插件，其中有专门实现本地文件功能的Blob对象，能够支持本地的读写。后来新版本的IE和其他浏览器认为activeX的安全性不靠谱，就舍弃了activeX，同时也禁止了对客户端本地文件进行脚本直接的访问修改操作，以防范在这个环节出现漏洞使得用户遭到攻击。

解决方法：通过复制粘贴的形式，将汇编语言拷贝到文件中，再用中间转译工具来转换成机器指令。

问题二：编译器在firefox和chrome浏览器上运行时，多次黑屏停顿，提示忙碌，甚至出现强制退出的情况。

原因：这是jQuery在遇到循环时，若循环次数较多可能会认为是死循环，而自发的保护机制。特别是发现Javascript中某一小段代码非常频繁的被调用时，有可能会直接kill掉整个进程。

解决方法：修改代码结构，使之对运行的浏览器的角度看来，更为合理和可靠。

1. 将细节末端的函数以功能分类的形式分开。比如说词义分析过程中跳过编译无用文本的函数spaceskip，分为跳过注释，跳过空格回车，跳过字符串内内容等多个函数。
2. 简化js中的数据结构，包括最重要的逻辑树。和预测的一样，c语言表达式结构占了整个逻辑树将近一半的空间，化简表达式结构至关重要。

# 第6章 结论与展望

## 6.1 工作总结

本文主要以xv6和ucore操作系统移植到v9-cpu上，并能在web环境下运行为目标，设计并实现了生成v9-cpu指令集的在线编译器uCompiler和模拟器uMonitor，并对uCompiler和uMonitor的实现原理和工作原理进行了详细的分析。

一开始的计划是直接将原有的编译器直接移植到web环境中，但是考虑到c语言和javascript相差太大，这导致的结果就是在测试和纠错的时候，不符合现有语言的规范就无从下手。所以尝试去理解原编译器程序的实现，通过自己对原编译器的理解以javascript语言的开发规范去重新去实现c语言编译器。这样就算是借鉴原版本而重新开发编译器了。而且好多编译器相关的工具包和javascript的兼容性并不好，只能自己白手起家独立开发，做了很多代码上的工作。

模拟器是在原有的模拟器的基础上，扩展了很多用于调试执行程序的交互功能。由于时间有限，并未有实现相应的web版本，于是尝试把原模拟器的功能做的更为完善。

## 6.2 目前问题和今后展望

由于文件系统不完善，导致要做手动复制粘贴代码来实现程序的最后生成，尝试过搭建网络框架来支持文件的读取和保存，但是效果并不理想。

1. 文件的读取，通过读取表项的post方法来读取文件，但是在未找到文件时会停止响应，在ASP.NET框架下，解决了这个问题，未找到文件会返回错误号202。
2. 保存数据到本地，采取的是新建窗口->打印数据->通过window.document.write的方法来保存，结构未能成功保存文件。在ASP.NET框架下，利用aspx文件里新建一个隐藏的ASP文件控件，js脚本触发这个控件的点击事件，将数据写入文本控件里，通过文件控件来保存文件，但这个方法会让ASP.NET的后台程序产生unknown的错误。

除了解决上述的文件问题以外，能够实现web运行的v9-cpu模拟器，但是v9-cpu中用户态和内核态的读写页表切换，使用的是指针地址的赋值交换方法，还有其他的关键点都是以c的动态分配内存机制为基础的，如果直接移植到js上，需要用能够分配内存的内存池对象来代替c的动态内存机制。实现内存池对象会遇到很多的麻烦。

1. 前面提到，Javascript的对象类型是弱类型的，内存池的空间数组的类型问题比较棘手。如果设定为空间大小不定的JS Object类型的话，还需要在这个对象里维护物理地址，对这个地址的维护就非常的棘手。
2. c和c++有个很灵活的特性，不同类型指针可以自由的转换，这个特性无法再Javascript这个弱类型的语言内体现。我的想法是通过实现转换函数，来转换不同的类型。 但是诸如像c语言里16位short\*转换为32位int\*这样，需要获得额外的数据，在这点Javascript需要拿到另一半的对象，将两个对象合并来实现这个操作。这一系列问题会非常的复杂。

从以上难点可以体现出，用javascript这类解释性语言来完全代替静态高级语言，在现在和今后依旧是一个很大的挑战。

# 插图索引

图1.1 编译器编译程序流程图 10

图1.2 一个给逻辑树结构举例的简单的c语言小程序 15

图1.3 小程序生成的逻辑树结构 16

图1.4 表达式和元素序列 19

图2.1 v9-cpu的运行流程 33

图2.2 函数调用栈的工作原理 36

图3.1 测试样例1 42

图3.2 测试样例2 42

图3.3 测试样例3 43

图3.4 测试样例4 43

图3.5 可执行程序的标志头对应的c类型格式 44

# 表格索引

表1.1 c语言关键字分类 12

表1.2 c语言特殊符号分类 12

表1.3 逻辑树结构列表 14

表1.4 逻辑树生成状态机的所有状态 17

表1.5 语义分析对象的主要功能 20

表1.6 记录对象的主要功能 21

表1.7 词义分析可能出现的编译错误 21

表1.8 语义分析对象的主要功能 22

表1.9 语义分析过程中可能发生的编译错误 22

表1.10 记录表支持语义分析的功能 24

表1.11 c语言结构对应的程序生成方法 24

表1.12 指令生成对象的主要功能 26

表2.1 v9-cpu的寄存器 29

表2.2 v9-cpu状态寄存器中的标志位 29

表2.3 v9-cpu指令分类 30

表2.4 v9-cpu的页转换表 30

表2.5 v9-cpu的IO相关指令 31

表2.6 v9-cpu的中断向量 31

表2.7 uMonitor模拟器的源码信息表 34

表2.8 条件断点的逻辑结构 35

表2.9 dml分析状态机的所有状态 37

表2.10 状态机使用方法 38

表2.11 uMonitor的调试命令 39

# 致谢

在这次毕业设计上，感谢陈瑜教授对我的信息指导和宝贵建议，经常关心我的完成情况和进一步计划，并对我接下来要做的工作提出了指导性的建议，同时能为我提供适宜的实验环境。感谢参与本次毕业设计和参与操作系统课程设计的同学们，能及时指正所设计不妥的处理方法，让我少走了很多弯路。在此由衷的感谢大家的帮助，祝这次毕业设计圆满成功的结束！

# 声 明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签 名：\_\_\_\_\_\_\_\_\_\_\_\_\_ 日 期：\_\_\_\_\_\_\_\_\_\_\_\_

# 外文资料的书面翻译