

Implementation of

SEMO C()MPILER Kernel

Reference Manual 1.0.1

semo c()mpiler[®]

Techniques of Knowledge

<http://Tok.cc>

jelo.wang@gmail.com

TOK
techniques of
knowledge
an open source group
since 2008

Part A Architecture and Implementation

Chapter A1 Introduction to the SEMO Architecture

[A1.1 About the SEMO architecture](#)

[A1.2 Front-Para](#)

[A1.3 Middle-Para](#)

[A1.4 Back-Para](#)

[A1.5 About the source directory](#)

[A1.6 About the SEMO ABM](#)

Chapter A2 Common-Basics Library Implementation

[A2.1 String data processing module](#)

[A2.2 General data-structure module](#)

[A2.3 Compiling-Render](#)

[A2.4 Hardware abstractive layer](#)

Chapter A3 C Language Front-Para Implementation

[A3.1 Preprocessor](#)

[A3.2 Lexical value definations](#)

[A3.3 Lexical Analyzer](#)

[A3.4 Syntax analyzer](#)

[A3.5 An universal semantic algorithm](#)

[A3.6 Aynalyzing name scope](#)

Chapter A4 Middle-Para Implementation

[A4.1 Abstractive symbol azonal](#)

[A4.2 Abstractive syntax Lgnosia AST](#)

[A4.3 Lgnosia CODE IR form](#)

[A4.4 SSA based on Lgnosia CODE](#)

[A4.5 Register Allocator based on Lgnosia CODE](#)

[A4.6 Parameter passing specification](#)

Chapter A5 Jvm-back-para Implementation

A4.1 Byte-code generating based on Lgnosia CODE

A4.2 Assembler

Appendix A

[A1.1 Hello world from Semo Compiler ☺](#)

[A1.2 About the SEMO system](#)

[A1.3 About the Techniques of Knowledge Group](#)

Chapter A1

Introduction to the SEMO Architecture

This chapter introduction to the SEMO architecture contains the following sections :

- ◆ A1.1 About the SEMO architecture
- ◆ A1.2 Front-Para
- ◆ A1.3 Middle-Para
- ◆ A1.4 Back-Para
- ◆ A1.5 About the SEMO ABM

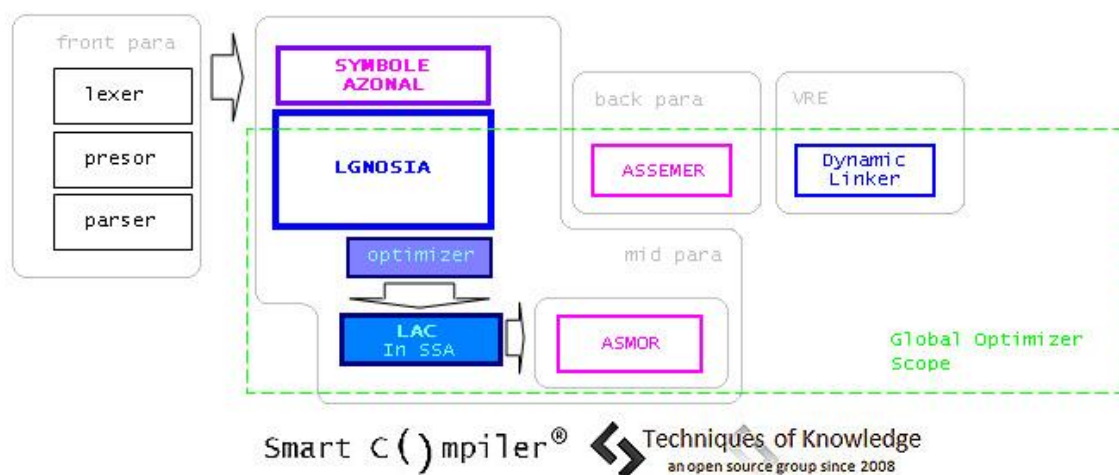
A1.1 About The *SEMO* Architecture

在正式进入析码的开发之前，你需要对它的架构、流程、以及一些基础库的用法有一个整体的了解，这章主要为你介绍析码的架构，其余部分可以通过其它章节了解。

析码的架构被划分为“前”、“中”、“后”三个段落(front-para、middle-para、back-para)。在代码上，段与段之间没有任何关联，基于这个特性，我们可以在最大可重用前提之下，使析码支持 n 中语言， n 种机器。语言相关的部分被划分到了“前段”，而与机器相关的部分被划分到了“后段”，中段是跟具体某个语言或某型号机器无关的部分，它被完全重用了。

首先是“前段”，任意语言之间前段非常地接近，这种接近是指有对应的单位可以相互转换，例如“函数”、“循环”、“条件语句”等等，忽略语义层的差异，不同类型的语言之间在词法跟语法上也有对应的单位可以等价地转换。这些语言相同的部分，或说它的相关性被隔离在了“前段”，让其某组分析器作为组件存在很大程度上提高了析码的语言兼容性。

其次是关于“后段”，与前段相同，后段的硬件体系部分也是大同小异，如果把范围缩小到最小，那些差异仅仅局限于“指令集”之上，指令集抽象出来作为后段的一部分，使司马在代码生成初期不去关心具体体系，后期只需要简单地将中间形式其映射到某体系即可，如此，在更换目标机器时，只需要替换一下映射函数，提高了机器兼容性。再有中段，它完全独立于任意语言，任意机器，如图 G1.1.1。



G1.1.1

① “front-para” 的作用是分析源代码结构，最终输出抽象语法 (Lgnosia AST)，它由“预处理器”、“词法分析器”、“语法语义分析器”组成。

② “mid-para” 为 “front-para” 以及 “back-para” 提供了一些共享接口，如 AST 与符号解析等。它是语言无关的，并且独立与任意种物理机器，总而言之 “mid-para” 是一个抽象模块，主要由 “抽象语法 (Lgnosia AST)”、“中间代码 (Lgnosia CODE)” 组成。

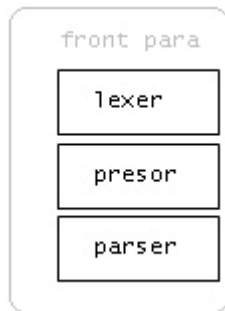
③ “back-para” 是跟某一种物理机器相关的部分，它由 “汇编器”、“连接器” 组成。

需要注意的是，“front-back” 跟 “back-para” 是一个统称，表示任意语言跟任意机器，它不涉及实现，而它的具体实现会是跟某一个语言以及跟某一个物理机器有关的，例如：“c-front-para” 这表示它是一个 C 语言的前段，而 “arm-back-para” 表示它是一个 ARM 机的后段。类似的还有 “java-front-para”，“X86-back-para” 等等。至于他们的组合以及种类，在架构没有任何限制，你可以将一个 JAVA 的前段跟一个 ARM 机的后段组合在一起，这样你就有了一个可以将 JAVA 在 ARM 物理层运行的编译器。换句话说，当你需要实现一个语言时，只需要实现一个 “front-para” 就可以了，同样地，当你需要为某一种机器实现一个编译器时，只需要实现一个 “back-para”，这是析码多目标编译的由来。

Introduction to the SEMO architecture

A1.2 Front-Para

具体来说，“front-para”由“预处理器”、“词法分析器”、“语义语义分析器”三部分组成。任意一种语言，包括自然语言，都是由“词法”、“语法”、“语义”这三部分组成的。因此前面我们说“front-para”跟语言相关，它负责解析源代码结构，输出抽象语法。如图 G1.2.1

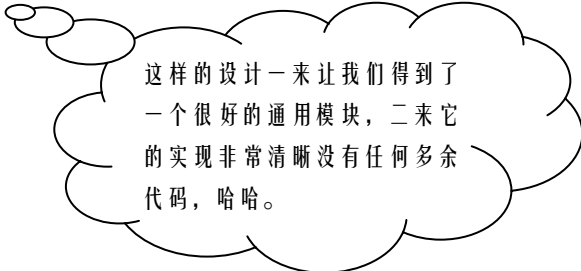


G1.2.1

A1.3 Middle-Para

中段非常重要，也是析码编译器最复杂的模块，它的存在这使得析码整体架构彻底地把前后段相互分离，使其实现上的自由度最大化。前后段的分离，不仅仅局限在语言层面，在体系层面也使得该编译器与某具体机器无关。

具体来说中段主要是负责符号管理、抽象语法、抽象代码生成、SSA、寄存器分配等。如图 G1.3.1，我们以此解析各模块，首先是符号管理，它的作用是把语法分析过程中遇到的各类符号收集起来，以便随后使用，例如变量赋值“int a=10”。在析码编译器中出于可重用性的考虑，我们把编程语言中的各类符号进行了抽象，不管它是变量还是数组或是函数，乃至结构体、类等等统统用一个称为“AZONAL”的抽象符号代替，它足以描述任意一种前段的具体数据对象，一个 ANL 的具体含义取决于语义，根据语义的不同，它可以是一个变量，或是数组，或者是一个函数等等。

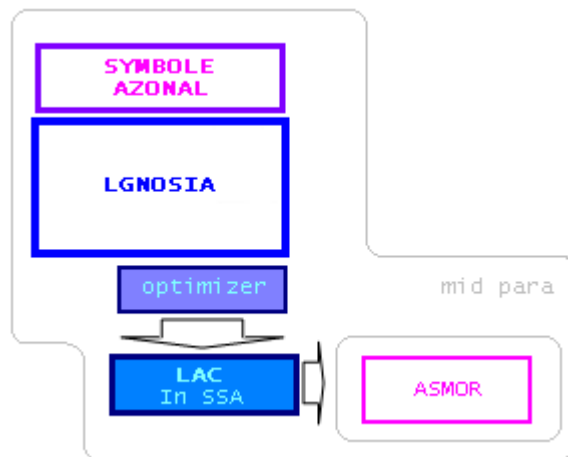


这样的设计一来让我们得到了一个很好的通用模块，二来它的实现非常清晰没有任何多余代码，哈哈。

Introduction to the SEMO architecture

介绍完了符号管理，接下来看一下抽象语法吧，所谓的抽象语法就是某个具体语言语法的精简形式，它去除了一些与语法本身无关的细节，如词法层面的若干细节问题。所以一个设计合理并且足够抽象的抽象语法应该具备描述大多数编程语言结构的能力，抽象语法是一种语法形式，它在析码编译器中实实在在的实现了“抽象语法树”，注意这里有点绕，不妨这样理解：抽象语法是思想，抽象语法树为实现。

在析码编译器中抽象语法树称为“Lgnosia AST”其为任意编程语言语法的抽象形式，它的抽象程度要比我们刚才提到的 AZONAL 还高级一些，与后者相比 Lgnosia AST 跟某语言特性完全分离了。



G1.3.1

“Lgnosia AST” + “AZONAL” 就是析码编译器 “front-para” 的输出数据，中段的中间代码生成器给予该组合产生 IR 代码。Lgnosia CODE 是析码中的一种 IR 形式，在后段运行初期，优化器会对 Lgnosia AST 做修剪，例如删除冗余代码，优化控制结构等等，这种结构性的优化只需要简单地修剪 Lgnosia AST 即可完成，保证优化后的 Lgnosia CODE 已经足够精简。

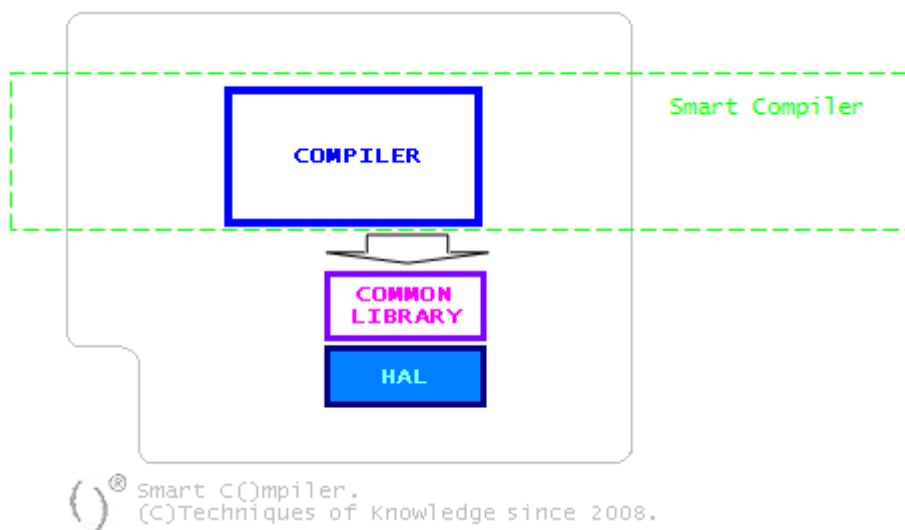
Introduction to the SEMO architecture

A1.4 Back-Para

后段由汇编代码生成、汇编器、跟链接器组成。到了这里，前段语言的特性已经完全不可见，后段看到的只是 Lgnosia CODE，汇编代码生成将其转换成某个体系结构的汇编代码，就是说它的输入是 Lgnosia CODE，输出是汇编代码。之后汇编器将汇编代码进行二进制编码，最终链接器链接目标文件。

A1.5 About The Source Directory

光有一个清晰的架构设计方案，如果在实现环节不遵循它，那就等同于废品。我们在析码的实现过程中对代码规范有极高追求，首先因为它是一个开源项目需要跨地域维护，其次这也是对编码者自己的要求——绝不放过打磨任何一个细节。



G1.5.1

析码的代码同样有三部分组成，它们分别是最上层的“Compiler”，三段模块的代码属于“Compiler”，第二层的“COMMON LIBRARY”是一些公共库函数，比如字符串操作、常用的数据结构如“栈”、“链表”、“图”等。最底层的“HAL”是设备抽象模块，由一些系统相关的模块组成，例如文件操作、内存分配等。如图 G1.5.1。



G1.5.2

目录的命名与划分完全遵循架构中的定义，如图 G1.5.2。除了“front-para”、“mid-para”、“back-para”之外，“COMMON LIBRARY”以及“HAL”在“common”目录下。

Introduction to the SEMO architecture

A1.6 About SEMO ABM

刚才我们有提到“Lgnosia CODE”，它是析码编译器的 IR。这一节我们需要换一种方式去重新认识它，这是系统设计的常用手法，通常对于复杂问题我们需要严谨地、系统地理解它就需要求助于一个假设的“模型”。假设我们有一块叫做“ABM”的 CPU，Lgonisa CODE 是这块 CPU 的汇编指令集，那么这种 IR 就应该具备所有“ABM”CPU 的主要特性，如“加减乘除”、“内存系统”等。当你站在这个高度去观察 IR 时，一切细节都会变得非常清晰易懂了。在此给出定义：Lgnosia CODE 是符合“ABM”体系结构的 IR 形式。

Chapter A2

Common-Basics Library Implementaion

This chapter common-basics library implementation contains the following sections :

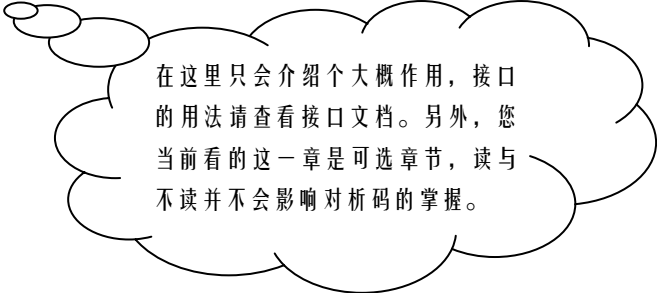
- ◆ A2.1 String data processing module
- ◆ A2.2 General data-structure module
- ◆ A2.3 Compiling-Render
- ◆ A2.4 Hardware abstractive layer

NOTE

本章只会框架性的介绍该模块的功能跟主要接口，所有接口的具体定义及用法请阅读《Semo Compiler Programming Reference Manual》。这些都是析码得以顺利开发的基石，参与项目必须要掌握的技术。Common-Basics 顾名思义，它为上层提供了给类基础接口，如字符串处理，常用数据结构等等。

A2.1 String Data Processing Module

相比数值，字符串是最难处理的数据形式，析码编译器重实现了一整套常用的字符串处理接口，如字符串切割、两串拼接、批量替换子串等，基于这些接口让字符串处理变得像高级语言一样高雅简单，最大的优点是其内存管理特性，调用者不需要自己分配内存，只管丢字符串过来析码会根据情况自行处理。



在这里只会介绍个大概作用，接口的用法请查看接口文档。另外，您当前看的这一章是可选章节，读与不读并不会影响对析码的掌握。

Common-basics library implementaion

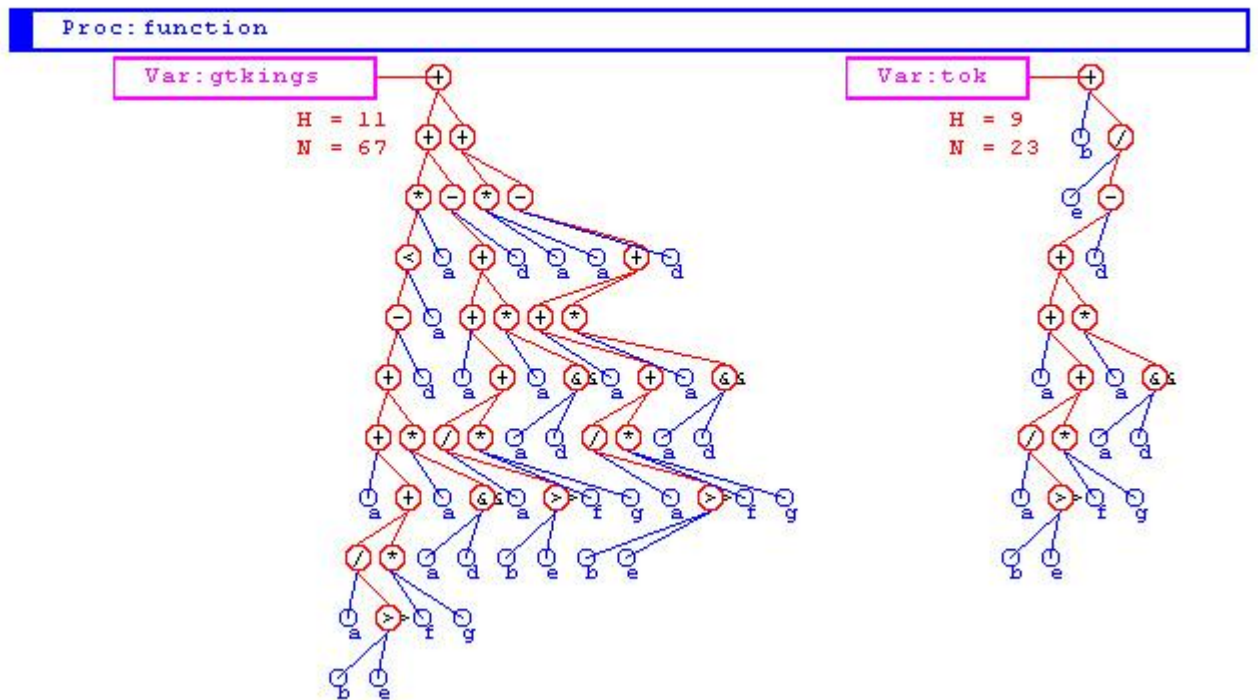
A2.2 General Data-Structure Module

该模块为上层提供了一组抽象的，与存储数据类型无关的“数据结构”，如：链表、栈、队列、树、图。

A2.3 Compiling-Render

“Compiling-Render” 是析码的一个调试工具，主要用来图形化编译器内部的数据模型，例如“抽象语法树”、“控制流”、“数据流”、“生命域”等等。此前，像是表达式翻译我们需要花费大量时间手工绘制相关图形然后人工验证翻译结果，而现在使用“Compiling-Render”可以自动将想要的模型图形化，使得我们可以将主要精力集中在中后段的优化上，方便开发。后续版本会加入“寄存器分配热度分布图”、“时间开销分布图”、“内存开销分布图”等等。图 G2.3.1 是一棵 Lgnosia AST。

Smart Compiler Compiling-Render
(C) Techniques of Knowledge



G2.3.1

A2.4 Hardware abstractive layer

析码编译器 99%的代码都是操作系统无关的，相关的只有“文件系统”、“内存系统”若干接口，这些接口统一封装与“schal.c”中，上层模块对系统接口的访问都通过设配抽象层转接，确保了析码的易移植性，就是说对于析码的移植，只需要重新实现“schal.h”定义的接口。

Chapter A3

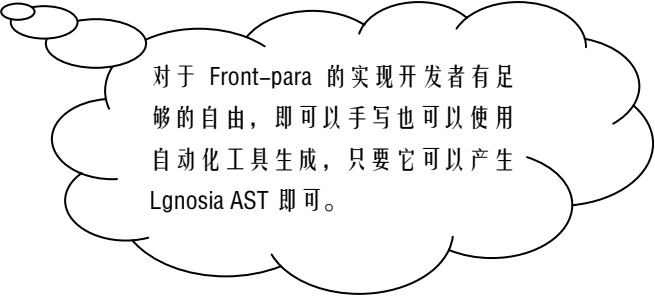
C Language Front-Para Implementation

This chapter C language front-para implementation contains the following sections :

- ◆ A3.1 Preprocessor
- ◆ A3.2 Lexical value definations
- ◆ A3.3 Lexical analyzer
- ◆ A3.4 Syntax & Semantic analyzer
- ◆ A3.5 An universal semantic algorithm
- ◆ A3.6 Aynalyzing name scope

NOTE

析码的架构支持多种语言前段同时存在，为了简单在此我们以 C 语言前段为例介绍 front-para 的实现，既 c-front-para。



对于 Front-para 的实现开发者有足够的自由，即可以手写也可以使用自动化工具生成，只要它可以产生 Lgnosia AST 即可。

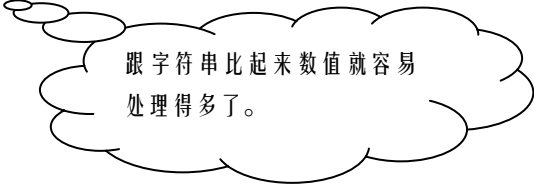
C language front-para implementaion

A3.1 Preprocessor

预处理器没什么特别的，就是 “Macro” 的处理，实现：c-presor.h、c-presor.c。

A3.2 Lexical Value Definations

“int”、“hello”、“func()” 这些字符串对语法分析器来说过于复杂，为了解决这个问题我们需要将它封装一下。结合我们刚才讲的，语法分析只关系词的词性，基于这个特性我们把这些字符串封装为一个整体：词法值，每一次词法值分别表示不同的词性，详细定义请见：c-grammar.h。



跟字符串比起来数值就容易处理得多了。

A3.3 *Leixcal Analyzer*

在编译周期中有很多个阶段，传统的观点中，前段部分一般有三个阶段：“词法分析”、“语法分析”、“语义分析”。词法分析是这样的，在自然语言中经常需要分析一个句子的结构，比如这么一句话，“DREAM WILL BE COMING TRUE”，如果不去管含义是什么，只搞清楚这句话的结构的话，那么只要确定单词的词性就可以了。

在编译中，这个确定单词词性的过程叫做“词法分析”。一个单词包括两个特性，第一个特性是单词的词性，比如人类语言中的动词、名词等等，相应的在编程语言中就是变量、操作数，运算符等等。第二个特性是单词的内容，一个变量的名字也是它的内容，当然“内容”还可以扩展，在析码中一个“单词”的内容还包括：行号等等。在编译理论中对词法分析的定义是，词法分析的输出是一组词法值，词法值由“词性”以及“词素”组成，词素就是我们说的变量的名字等等。

词法分析的主要目的就是确定单词的类型，那么当分析器碰到一串字符串的时候如何确定它的类型呢？通常的做法是，对单词进行分类，然后定义这些分类，分类类型的定义很简单，可以用一个简单的数值与之对应，但必须要唯一。例如“1001”代表变量，“1002”代表函数等等（详见 c-grammar.h）。

在析码中的实现：c-lexer.h、c-lexer.c，这个词法分析器是根据 C 词法规则实现的，因为 C 语言设计年代比较久远，随后的很多语言都是在它基础上衍生出来的后辈，因此这些语言在词法层面都非常接近，这给 c-lexer 通用化接口化创造了条件。（预处理器跟语法分析器都会调用它去分析字符流）

```
void livescope ( int x ) {  
  
    int a ;  
  
    call(x) ;  
  
    a = a + x ;  
  
}
```

词法分析器依次返回“词法值”：

- 词性：C_FUNC_DEF;词素：“livescope”；返回值类型：C_VOID_BIT
- 词性：C_XKL;词素：“(”；
- 词性：C_VAR_DEF;词素：“x”；数据类型：C_INT_BIT
- 词性：C_XKR;词素：“)”；
- 词性：C_XKL;词素：“{”；

- 词性: *C_VAR_DEF*; 词素: “a”; 数据类型: *C_INT_BIT*
- 词性: *C_FEN*; 词素: “,”;
- 词性: *C_FUNC_REF*; 词素: “call”;
- 词性: *C_XKL*; 词素: “(”;
- 词性: *C_VAR_REF*; 词素: “a”;
- 词性: *C_XKR*; 词素: “)”;
- 词性: *C_FEN*; 词素: “,”;
- 词性: *C_VAR_REF*; 词素: “a”;
- 词性: *C_EQU*; 词素: “=”
- 词性: *C_VAR_REF*; 词素: “a”;
- 词性: *C_ADD*; 词素: “+”;
- 词性: *C_VAR_REF*; 词素: “x”;
- 词性: *C_FEN*; 词素: “,”;

```
lexerc_ready ();  
lexerc_setmode ( LEXERC_FLITER_MODE | LEXERC_HEADBIT_MODE );  
for ( lexerc_genv () ; !lexc->stop ; lexerc_genv () ) {  
    if ( C_VAR_DEF == lexc->v ) {  
    } else if ( C_ARRAY_DEF == lexc->v ) {  
    } else if ( C_FUNC_DEF == lexc->v ) {  
    }  
}
```

请仔细体会并理解以上代码。

A3.4 Syntax & Semantic Analyzer

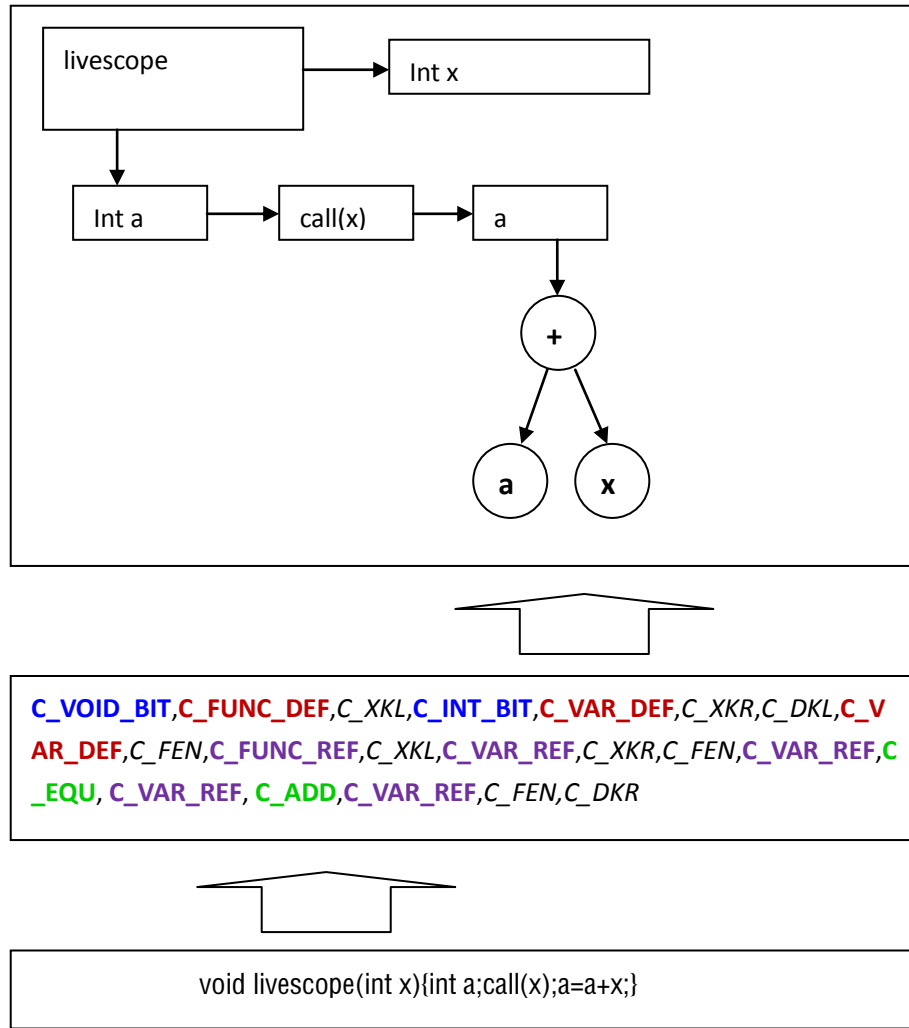
语法分析的前提是要确定“单词”的词性：“形容词”、“名词”或是“动词”等等，然后才能根据语法去预测下一个单词应该是什么词——这就是语法分析过程。

从搜索语法树的方向来看，语法分析有“自顶向下（先序）”跟“自底向上（后序）”两个类型，自顶向下算法的基本流程是，它首先从语法树根节点 Root 开始，生成第一个子节点 Root1.0，然后生成 Root1.1.0，最终生成 Root1.1.1.1.1.1...0 直到生成到 Root 第一个分支的叶子节点为止，然后进入了“漫长”的回溯过程，回溯的过程中同样重复着之前的步骤，除了“回溯”LL 算法并没有很明显的缺点，而且它的可读性更好。自底向上算法跟自顶向下算法正好相反，它从语法树的叶子节点 Root1.1.1.1.1.1...0 开始生成整个 Root 的分支，如果代码的语法正确，最终会生成一个完整的以 Root 为头结点的语法树，LR 算法虽然没有回溯，但是为了要确定某一个叶子节点的父节点它需要搜索大量“产生式”，这些产生式存储在一个分析表中，纯手工整理出一个分析表非常繁琐，这是 LR 算法的缺点。析码的“c-front-para”实现基于“LL(n)”算法，这个算法足够简单而且方便手工实现，以下我们将以一个实例讲解该算法的实现。

```
void livescope ( int x ) {  
  
    int a ;  
  
    call(x) ;  
  
    a = a + x ;  
  
}
```

还是以上面代码为例，分析过程如下图 G3.4.1。

C language front-para implementaion



G3.4.1

词法分析器将字符流量化并返回给语法分析器，语法分析器遍历产生式并生成 Lgnosia AST。有了 AST 语法分析器的任务就达成了，错误检测只是语法分析器的附带功能而已，那么对于语义分析来说道理也是一样的，我们可以在语法分析过程中检测语义，缺点是每个语法分析器都需要实现语义检测。

能否让语义分析也根据语言无关呢？试想一下大部分语言的差异仅仅实在语法层面，他们的词法跟语义规则都是极其接近的，接下来我们讲介绍一种通用算法，不过该算法并没有在新版本的析码中应用，因为它需要在语法分析结束后再进行一遍代码扫描，再次加以介绍只为探索性研究。

A3.5 An universal semantic algorithm

在“0.1.0”中，我们实现了一个抽象的语义分析器，既语义分析不依赖于语法分析过程，它们是完全独立的两个模块，之间没有任何关系，一般编译器中语义分析是在语法分析的时做的，在该实现中把语义分析从语法分析中分离出来的目的是因为我们搞了一个叫“抽象语义”的东西。

我们先看一下语义大概的分析流程，什么是语义分析，以及如何实现等问题。说到底语义分析只有两个问题，一是类型一致问题，二是作用域，首先先看类型一致性问题。句子“`int a = c + b;`”的语义分析需要回答下面五个问题：

- (1) “a”是什么类型，已经定义了吗。
- (2) “c”是什么类型，已经定义了吗？
- (3) “b”是什么类型，已经定义了吗？
- (4) “+”可以运算的类型是什么？
- (5) 三个变量都定义了吗？

回答上面五个问题，需要下面两个操作：

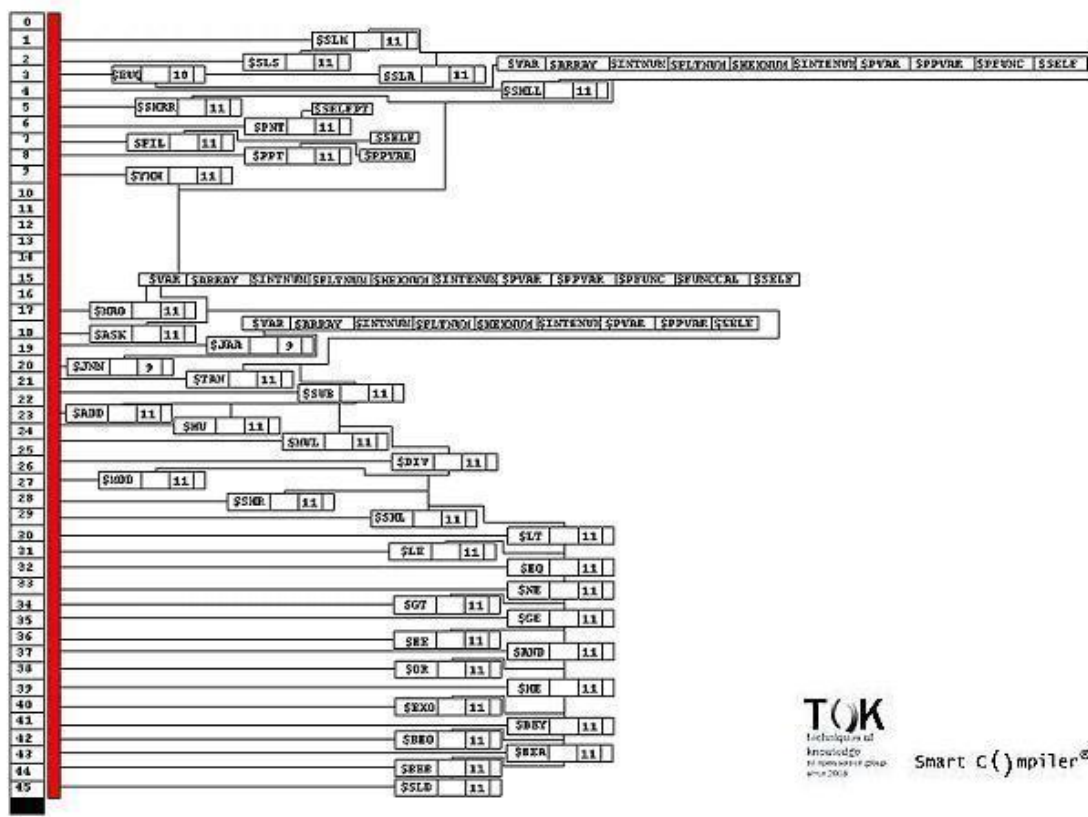
- (1) `t = return type(a)` 返回 a 的类型 t。
- (2) `match_functer(‘+’,t)` 算符“+”可以作用在 t 上面吗？

其中 `return type(a)` 是“SYMBOLE”接口，如果变量 a 没有定义，返回“假”。其中 `match_functer(‘+’,t)` 运行在“运算符语义表”上面。

遇到错误后的错误提示：

- (1) “a”的右值类型错误。
- (2) “+”操作数类型错误。
- (3) 变量没有定义。

C language front-para implementation



G3.5.1

match_functer 在一个叫“运算符语义表”的散列表上查找，上图所示，每一个运算符都存储在了唯一的单元中，除第三个单元外发生了一次冲突外，其余各单元的地址都是唯一的，该表对应的源代码模块在“GLOBAL.h”中。

当分析器遇到数组的时候，它需要被告知下列三个问题：

- (1) 数组的类型对吗？
- (2) 数组的赋值对吗？
- (3) 数组的维数对吗？

前两个问题处理方法跟前面提到的一样，对于问题三，也是一个查表过程。

作用域分析需要用到“SYMBOLE”：

每一个作用域都应该有一个唯一的编号，到遇到一个作用域的时候，会进入一个循环，所有在这个循环中遇到的变量，数组的作用域必须与作用域编号一致。

可能出现编号不一致的情况有两种：

- (1) 全局变量。
- (2) 不同作用域的局部变量。

C language front-para implementation

对应第一种情况好解决，全局变量在“符号表”中的作用域编号为“0”。

第二种情况稍微复杂一点，因为可能会出现下面这种嵌套的情况：

```
int gtkings() {  
  int a;  
  if(1) {  
    int b;  
  }  
}
```

我们这样给作用域编号，gtkings 编号 10，那么变量 a 的作用域就是 10，换句话说，在语义上表示它只属于作用域 10。当“循环”在 gtkings 中的时候，a 的作用域覆盖整个模块。对于变量“b”来说，它只属于“if”模块，为了简化语义分析算法的设计，再给“b”和 if 编号的时候，最好直接能够表示出这个嵌套关系来。也就是说，当语义分析器读到“b”的时候，它很容易就知道“b”是一个只属于“if”作用域内的，在“gtkings”模块内的局部变量。

编号怎么编能直接反映出“b”的作用域属性？用一个二进制模式表示，可以这样把二进制拼起来，我们这样约定，高位代表最外层模块，用 1 表示，连续两个 1，比如 11，表示外层模块内还有一个模块，但是没有变量。变量用 0 表示，如果有 110，这表示，在第二个模块内有一个变量。如果 1010，该编码是 gtkings 模块的作用域编码。

上面这个方法解决了嵌套作用域的语义分析问题，但还是没有解决变量“b”的编码。因为“作用域”这个信息，只有在语义分析跟代码翻译两个阶段才会用到，可不可以在语义分析阶段再收集这个信息呢？如果在语法分析阶段收集这个信息可能会让代码很乱，而且最好不要去分散语法分析器的“注意力”，这个阶段我们只关心语法上的问题。那好“作用域”信息就让语义分析器收集吧。下面我们看语义分析器如何给变量“b”编号。

为了更好地理解这个过程，我们用一个“分析实例”来描述：

```
int gtkings() {  
  int a;  
  if(1) {  
    int b;  
  }  
  b++;  
  a++;  
}
```

C language front-para implementation

分析器读到了一个模块“gtkings”，注意，这里先不要去关心模块的编号，暂时只关注作用域信息就行。模块作用域的信息为 1，分析器继续往下读取，他遇到了一个变量“a”，这时候的作用域编号为 10，因为之后的分析工作这个信息仍然有用，

分析器把这个作用域信息存到了 a 在符号表中的属性中。等到分析器读到 if 模块的时候，作用域信息变成了 101，它继续往下读取，遇到了另一个变量“b”，作用域信息变为 1010，同样地，它把这个作用域信息存放到了“b”在符号表中的属性中。

“if”模块结束，作用域变成了 101，语义分析器这时又遇到了变量“b”，它在符号表中找到“b”的作用域信息，为 1010，但是语义分析器发现，现在的作用域是 101，这两个编码不相等，它发现了一个错误，于是打印了一条错误信息“变量 b 越界使用。”之后，语义分析器继续往下工作，它读取到了一个变量“a”，它的作用域跟当前的作用域匹配，模块 gtkings 的语义分析结束。

可能你已经发现了，这个所谓的二进制编码有栈的特性，因此在实现的时候，可以把它做成一个栈，这样分析器就更容易实现了。但是它不能作为变量的属性，变量的属性还需要用二进制。

对于 C 语言中的函数返回值问题，很容易解决。在进入作用域分析的那个“循环”的时候，把函数的返回值类型保存起来，等遇到返回句的时候，匹配一下类型即可。

到此对于作用域分析得出了下面三个结论：

- (1) 栈。
- (2) 栈中内容转换为二进制。
- (3) (2) 的逆过程，二进制转换为栈中内容。

思考一下：

- (1) int a = 1.1; 有什么问题？

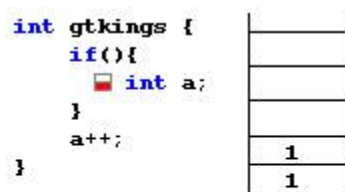
问题是：a 是一个整型变量，它不能被赋值浮点。

对于这种句子如何实现语义分析呢？首先对于描述“a 是一个整型变量”，很明显需要有一个地方可以得到 a 的类型信息。“它不能被赋值浮点”，这个可以作为“a”的属性，可以把所有它可以赋值的类型，放到一个表中，用 C 中的数组就可以。

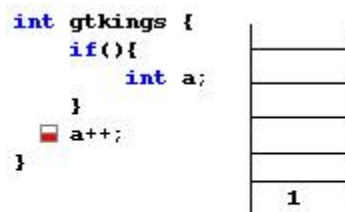
C language front-para implementation

C language front-para implementation

关于作用域信息的表示可以用字符，也可以用数值，总之它表示深度信息。如下图 G3.5.2 所示：



栈不变，变量a的作用域属性为 11.



此时，栈中字符 1，a的作用域属性为 11. 这时候需要考虑到，如果a不是全局变量，那么a就属于越界使用，打印错误信息。

G3.5.2

作用域属性生成规则，如果是定义变量，或数组，则把栈中信息放入符号表对应表项。如果是函数，或者是任何其他结构语句，把字符 1 压入栈。以上结构语句结束，出栈一次。栈空时作用域分析结束。

```

int gtkings () {
  if (1)
  int a,b,c,d,e ;
}

```

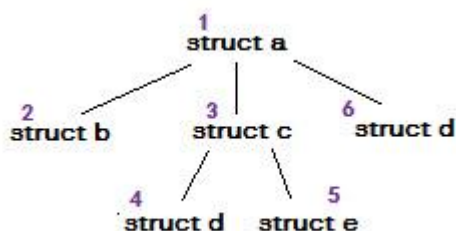
对于这种 if 语句的写法，作用域规则如,覆盖 if 语句后面第一个语句，整个逗号表达式算是一个语句。

C language front-para implementaion

A3.6 Analyzing nake scope

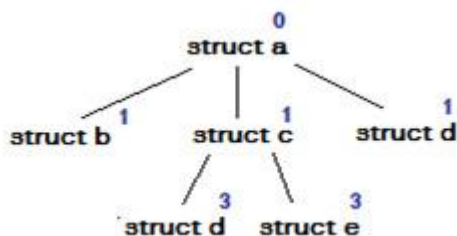
以下算法可以解决命名空间分析问题，看一下具体实现：析码中实现了一种叫作“线锁”的结构，线锁把结构体的定义头链在一起，逻辑上可以把它看作是树形结构。这种结构可以把诸如结构体变量属性的引用简化为一个线性遍历过程。当分析某个结构体变量语义时可以在线锁中搜索，上例中的结构体“定义头”a是线锁父节点，其余b、c、d、e、d都它在线锁上，该规则可以推广到任意满足这个特性的结构体定义头上，例如上例的定义头c是a的递归，它的基本性质跟a相同。

C语言规定，凡是出现在分析树同一层次上的结构体定义头必须是唯一的。这个语义可以用线锁号来实现，如图G3.6.1。



G3.6.1

为了生成线锁号，我们必须先为每个节点编号，C语言的语法分析器是预测分析算法，从遍历分析树的方向来看是先根遍历，节点编号号可以在遍历分析树的时候生成。有了节点编号之后分析器就可以生成节点的线锁号了，某个节点的线锁号是它的父节点的编号，如下图G3.6.2所示：



G3.6.2

对于结构体的定义头命名空间分析只需要比较线锁号即可，当一个定义头名称相同并且线锁号也相同时，说明出现了语义错误。

Chapter A4

Middle-Para Implementaion

This chapter Middle-Para contains the following sections :

- ◆ A4.1 Abstractive symbol azonal
- ◆ A4.2 Abstractive syntax Lgnosia AST
- ◆ A4.3 Lgnosia CODE IR form
- ◆ A4.4 SSA based on Lgnosia CODE
- ◆ A4.5 Register Allocator based on Lgnosia CODE
- ◆ A4.6 Parameter passing specification

Middle-Para implementaion

A4.1 Abstractive symbol azonal

在 [A1.3 Middle-Para](#) 我们有介绍，Azonal 是一个抽象的符号我们可以用它来表示任意符号，如变量、数组、函数、结构体、类。此处你将会看到它是如何实现的，关键在于 ANL 的数据结构，如下表：

T4.1.1 数据结构

结构名称	成员名称	成员类型	说明
AZONAL	name	char*	名称
	azonaltype	int	ISA_VARIABLE , ISA_ARRAY , ISA_FUNCTION , ISA_EXPR , ISA_CONTROLFLOW , ISA_NUMERAL
	datatype	int	一个符号的数据类型
	belong	int	表示这个 ANL 符号属于哪个 ANL 符号，例如函数跟变量的所属关系
	scope	int	作用域，数值越大作用域越深
	lgabelong	int	所属的 Lgnosia AST 节点
	used	int	是否被引用过
	line	int	位于的行号
	isparam	int	是否是参数
	number	int	编号
	layer	int	如果是 ISA_ARRAY，表示数组的维度
	size	int	数据长度
	scale	int	NULL
	tack	SCCList	如果是 ISA_FUNCTION，则参数存于此处，参数同样是一个 ANL
	DRC	struct	用于记录一个符号的引用情况，用于生成 SSA
	head	ANODE*	上一个 ANL 节点
	next	ANODE*	下一个 ANL 节点

Middle-Para implementaion

A4.1.1 AZONAL 语法分析中的应用

```
azonal = SymboleAddVarAzonal (
    lexc->token ,
    lexc->headbit ,
    ISA_VARIABLE ,
    scope ,
    ( parserc->scope == PARSERC_SCOPE_PARAM ) ,
    lexc->line ,
    SymboleGetCurrentFunc () ,
    (int) LgnosiaStackTop ()
);

LEXERC_HEADBIT_RESET();

// create a LGNOSIA_SYM_IDENT
lgnosia = LgnosiaNew ( (int)azonal , LGNOSIA_SYM_IDENT );
```



掌握 AZONAL 的用法至关重要。

把变量封装为一个 ANL，然后再将 ANL 封装为一个 Lgnosia AST 节点。到此，变量的解析，跟 AST 节点的创建已完成。

A4.1.2 AZONAL 在 Lgnosia AST 中的应用

```
lgnosia = (LGNOSIA*) lacgentor.lgnosia ;
azonal = (AZONAL*) lgnosia->azonal ;

LACAdd ( azonal->name , LAC_PROC , lacgentor.identor.deep );
LACAdd ( "(" , -1 , -1 );
```

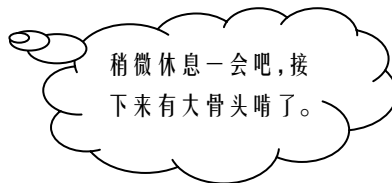
我们在 [A1.3 Middle-Para](#) 节介绍过，Lgnosia 只用来描述源代码的结构是什么样子的，在生成 IR 时需要根据结构生成不同的 basics-block，出结构信息外还需要知道更具体的细节，比如一个 if 语句的条件表达式，这就得借助于 ANL 了。比如上面这段代码，在获取到 Lgnosia AST 节点后，从节点内部把 ANL 取出来，然后进行 IR 代码的翻译，此时你想把它翻译成什么形式都可以，管它是 JAVA 还是 C#或者是 ASM。

A4.2 Abstractive syntax Lgnosia AST

本节我们将正式地介绍析码编译器核心——Lgnosia AST，这是一个非常给力的“工具”。它具有很好的数学特性，这为后端简化了问题的复杂度，试想一下如果后端的面对的是前段那些凌乱的字符串会是什么情形？

“Lgnosia”是析码编译器的抽象语法，既 AST。说起来“Lgnosia”这个单词的命名我们构思了很久，大概得将近一个月吧，最终把两个单词各取一部分组成了现在的“Lgnosia”，含义是：“语言无关”。以下简称“LGA”，前段解析源代码，将代码的结构用 LGA 表示，前段运行结束后，最终输出“Lgnosia AST”。从数据结构观点来看，前段输出的“Lgnosia AST”是一个森林，它精确的描述了目标语言的代码结构。

前段运行结束后，析码编译器中所有跟目标语言相关的操作都已经结束。到了中段运行阶段，其看到的只有抽象化的“Lgnosia AST”。理论上，“Lgnosia AST”可以用来描述所有语言的语法结构，因此基于析码实现一门语言，只需要保证可以正确产生“Lgnosia AST”即可。



Lgnosia 中定义了六类抽象语法：

- LGNOSIA_SYM_IDENT：基块，前段最基本的语言单位被归为该类，例如变量，数组等符号。
- LGNOSIA_CP_IDENT：控制点，语言中控制流被归为为该类，例如条件语句，循环等。
- LGNOSIA_EXP_IDENT：算术表达式、逻辑表达式。
- LGNOSIA_POC_IDENT：控制流条件为真时的分支。
- LGNOSIA_POC_IDENT：控制流条件为假时的分支。
- LGNOSIA_TOP_IDENT：一个 LGA 树最顶端的节点，如果目标语言是 C 语言，那么 C 语言函数就是最顶端节点。

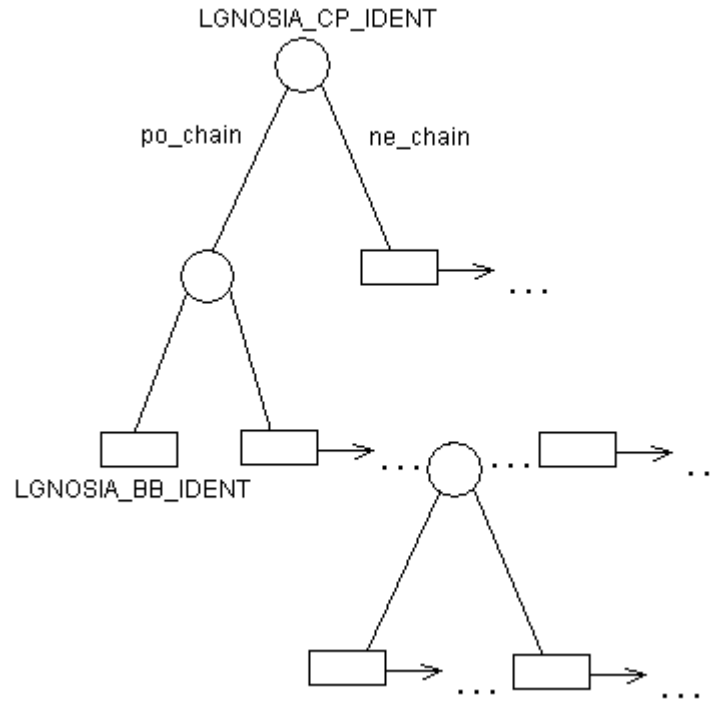
Middle-Para implementaion

T4.2.1 数据结构

结构名称	成员名称	成员类型	说明
LGNOSIA	type	LGNOSIA_IDENT	抽象语法类型
	azonal	int	每个 Lgnosia AST 节点都包含一个 ANL。
	deep	int	Compiling-Render 专用
	father	int	父节点
	allinyer	int	Compiling-Render 专用
	x	int	Compiling-Render 专用
	y	int	Compiling-Render 专用
	parameter	SCCList	参数列表，如果是一个 LGNOSIA_CP_IDENT，参数表达式存在此处
	context	SCCList	上下文，如果是一个 LGNOSIA_TOP_IDENT，则作用于内部的 Lgnosia AST 树都在此处
	po_chain	LGNOSIAN*	如果是一个 LGNOSIA_CP_IDENT，该域表示真值为“真”的控制流
	ne_chain	LGNOSIAN*	如果是一个 LGNOSIA_CP_IDENT，该域表示真值为“假”的控制流

LGNOSIA 结构内部各个域的含义，取决于该 LGA 节点的具体类型，对于一个“Basics-Block”来说，只有“context”是有意义的，context 是一个链表，其中保存了 LGAATOM，具体信息在该结构中。当一个 LGA 节点类型是“控制点”时，“parameter”保存了其参数，“context”保存了该控制点所管辖的所有基块与控制点，“po_chain”保存了当条件为真时的控制链，“ne_chain”保存了为假时的控制链。“type”是 LGNOSIA_IDENT，“azonal”指向一个 ANL 节点。

Middle-Para implementaion



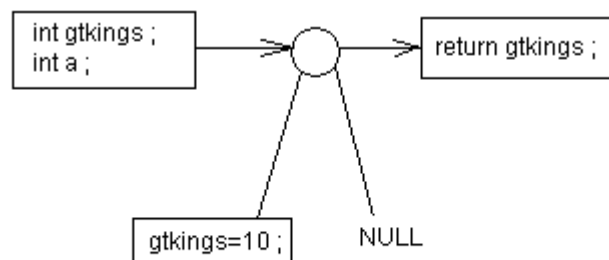
G4.2.1

图 G4.2.1 圆形节点为控制点，“po_chain”与“ne_chain”指向的矩形为基块，所有基块与控制点的集合构成一个更大的基块。从结构观点来说，控制点是一个非线性结构，它满足所有二叉树的基本性质。而基块则是一个线性结构，除此之外，基块中可以包含其它非线性结构。

```
int gtkings ;
int a ;

if (1) {
    gtkings = 10 ;
}

return gtkings ;
```



G4.2.2

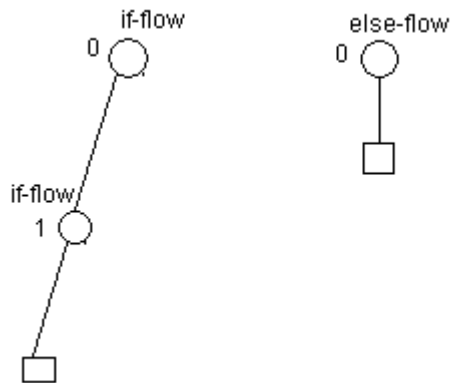
Middle-Para implementation

图 G4.2.2 是一个 C 语言前段的 LGNOSIA 实例，变量 gtkings 跟 a 被存入一个基块，该基块指向一个控制点，控制点 po_chain 指向 gtkings=10;ne_chain 为空。

A4.2.1 生成 LGNOSIA_CP_IDENT

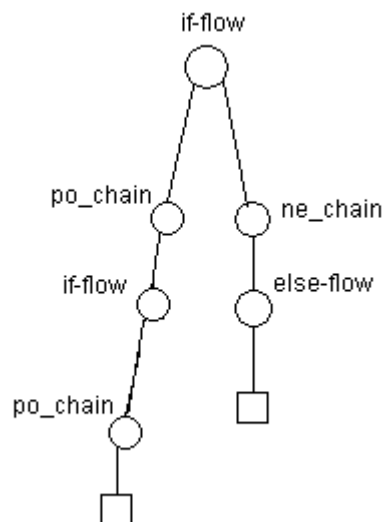
以 if-else flow 为例，它的基本性质：

- (1) 分析树中一个 if 节点，至多有两个分支，其左子分支或右分子支可以是 if 节点，至少有一个分支。
- (2) Else 节点与 if 节点所属同一层，else 只有一个子分支，分支可以是 if 节点。注：在 if-else 流之外，所属于另一个更大的控制流。



G4.2.3

图 G4.2.3 是一个 if-else 分析树，圆形节点是控制点，叶子矩形节点为具体符号。可以理解为由控制点所控制的语言符号。控制点外侧的数字，表示深度。



G4.2.4

Middle-Para implementation

图 G4.2.4 是 if-else 分析树对应的 LGNOSIA, 与分析树不同, LGA 符合 if-else 逻辑定义, 控制点为真的路径在其 “po_chain” 上, 为假的路径在 “ne_chain” 之上。

◇ 算法:

if-flow (lv)

当前节点、深度 压栈

深度 + 1

if 下一个节点为 if-flow

flow(下一个节点)

end

深度 - 1

end

else-flow (lv)

取得栈顶元素 A

如果 A 的深度比当前深度大

while(1)

出栈

until A.深度 == 当前深度

if-flow = 出栈

将该 else 与 if-flow 连接

深度 + 1

if 下一个节点为 if-flow

flow(下一个节点)

end

深度 - 1

End

标号生成是析码编译器的难点之一, 也是核心, 没有标号就没有流程。

每次遍历将 if-flow 入栈, 这可能会产生垃圾信息, 需要在出栈的时候过滤这些冗余信息, 冗余的产生取决于源代码的书写, 例如没有 else 分支的 if 语句。

遍历到 else-flow 时, 需要将该节点与之前的 if-flow 对应起来, 并将其连接到 if-flow 的 ne_chain 之上。应该连接到哪一个 if-flow 上面, 可以从分析栈中获取。由于可能会有递归的情况, 栈中不只保存了一个 if-flow。

这情况比较复杂, 我们忽略掉其它, 这里只需要关心深度值, 只要当前深度比栈中元素深度小, 就需要出栈, 直到搜索到相等为止。

搜到后, 将节点弹出栈, 并与 else-flow 连接。

Middle-Para implementaion

A4.3 Lgnosia CODE IR Form

“Lgnosia CODE”，它是前段语言的 IR 形式，该形式独立于具体的机器，析码编译器主要用它来做一些与前后段无关的优化工作。此外，LAC 有利于析码编译器提高自身的兼容性。

LAC 这种形式，降低了析码编译器核心模块与某个具体实现的相关度。除了前面这种提高维护性的作用之外，在代码优化方面抽象形式为析码编译器提供了基本优化模型，因为模型是抽象的，它与具体体系无关，这让优化变得非常纯粹。该形式，根据它与前后段的“相似”程度，析码编译期中定义两层 LAC。这里的相似度是指“语义”方面，层次越高越接近高级语言，层次越底越接近机器。

A4.3.1 About The 0-Level Lgnosia CODE

第 0 层是最接近目标机器的，它与汇编语言类似，但这种类似只局限于形式上，这一点非常重要，比如析码编译器的寄存器分配器就在第 0 层之上产生优化策略，如此以来即便改变了目标机器，原先的寄存器分配器仍然可用。

总之，第 0 层的定义有两个目的：

- (1) 因为形式非常接近汇编语言，可以简化汇编代码生成器的实现。
- (2) 让原本与机器相关的优化都与之无关，达到了通用化，如寄存器分配等。

Middle-Para implementaion

A4.3.1.1 Keywords

\$\$CA
\$\$IF
\$\$ELSE IF
\$\$ELSE
\$\$WHILE
\$\$STACK
\$\$MEMORY

A4.3.1.2 Identifiers

Alpha : one of

_ a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

Digit : one of

0 1 2 3 4 5 6 7 8 9

A4.3.1.3 Constant

integer-constant

floating-constant

character-constant

integer-constant:

decimal-constant integer-suffixopt

octal-constant integer-suffixopt

hexadecimal-constant integer-suffixopt

decimal-constant:

nonzero-digit

decimal-constant digit

octal-constant:

0

octal-constant octal-digit

hexadecimal-constant:

hexadecimal-prefix hexadecimal-digit

hexadecimal-constant hexadecimal-digit

hexadecimal-prefix: one of

0x 0X

nonzero-digit: one of

Middle-Para implementaion

1 2 3 4 5 6 7 8 9

octal-digit: one of

0 1 2 3 4 5 6 7

hexadecimal-digit: one of

0 1 2 3 4 5 6 7 8 9

a b c d e f

A B C D E F

integer-suffix:

unsigned-suffix long-suffixopt

unsigned-suffix long-long-suffix

long-suffix unsigned-suffixopt

long-long-suffix unsigned-suffixopt

unsigned-suffix: one of

u U

long-suffix: one of

l L

long-long-suffix: one of

ll LL

floating-constant:

decimal-floating-constant

hexadecimal-floating-constant

decimal-floating-constant:

fractional-constant exponent-partopt floating-suffixopt

digit-sequence exponent-part floating-suffixopt

hexadecimal-floating-constant:

hexadecimal-prefix hexadecimal-fractional-constant

binary-exponent-part floating-suffixopt

hexadecimal-prefix hexadecimal-digit-sequence

binary-exponent-part floating-suffixopt

fractional-constant:

digit-sequenceopt . digit-sequence

digit-sequence .

exponent-part:

e *signopt digit-sequence*

E *signopt digit-sequence*

sign: one of

+ -

digit-sequence:

digit

Middle-Para implementaion

digit-sequence digit
hexadecimal-fractional-constant:
hexadecimal-digit-sequenceopt .
hexadecimal-digit-sequence
hexadecimal-digit-sequence .
binary-exponent-part:
p *signopt digit-sequence*
P *signopt digit-sequence*
hexadecimal-digit-sequence:
hexadecimal-digit
hexadecimal-digit-sequence hexadecimal-digit
floating-suffix: one of
f l F L

A4.3.1.4 Character Constant

enumeration-constant:
identifier

character-constant:
'*c-char-sequence* '
L' *c-char-sequence* '
c-char-sequence:
c-char
c-char-sequence c-char
c-char:
any member of the source character set except
the single-quote ' , backslash \ , or new-line character
escape-sequence
escape-sequence:
simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence
universal-character-name
simple-escape-sequence: one of
'|' '\" \? \\
\a \b \f \n \r \t \v
octal-escape-sequence:
octal-digit
octal-digit octal-digit
octal-digit octal-digit octal-digit
hexadecimal-escape-sequence:

Middle-Para implementaion

\x hexadecimal-digit
hexadecimal-escape-sequence hexadecimal-digit

A4.3.1. 5 String literals

string-literal:

" s-char-sequenceopt "

L" s-char-sequenceopt "

s-char-sequence:

s-char

s-char-sequence s-char

s-char:

any member of the source character set except
the double-quote *"*, backslash **, or new-line character

escape-sequence

A4.3.1. 5 Punctuators

punctuator: one of

[] () { } . ->

*++ -- & * + - ~ !*

/ % << >> < > <= >= == != ^ | && ||

*? : ; = *= /= %= += -= <<= >>= &= ^= |= ,*

A4.3.1. 6 Primary Expressions

primary-expression:

identifier

constant

string-literal

(expression)

A4.3.1. 7 Postfix Expressions

postfix-expression:

primary-expression

postfix-expression (argument-expression-list_{opt})

postfix-expression ++

postfix-expression --

(type-name) { initializer-list }

(type-name) { initializer-list , }

argument-expression-list:

assignment-expression

argument-expression-list , assignment-expression

Middle-Para implementaion

Procedure-Definations

Procedure-Name (*Parameter-List*)
{
 Code-Block
}

Procedure-Call

%%\$CA Procedure-Name

Control-Flow

%%\$IF(*Parameter-List*) false goto *Lable_{num}*
 Code-Block

%%\$ELSE IF (*Parameter-List*) false goto *Lable_{num}*
 Code-Block

%%\$ELSE
 Code-Block

%%\$WHILE(*Parameter-List*) false goto *Lable_{num}*
 Code-Block

Object-Operation

%%\$STACK.TOP INIT 10
%%\$STACK.TOP-4 IN 3
%%\$STACK.TOP-8 IN 2
%%\$STACK.UNIT 10
%%\$MEMORY.Address digit

其它如算式表达式与 C 语言类似，在此暂不做详细说明。

Middle-Para implementaion

A4.4 SSA Based on Lgnosia CODE

“SSA——Static Single Assignment 静态单赋值”一种对 IR 的优化思想。具有“SSA”性质的代码具备 2 个特点：（1）不存在同名的符号、（2）对同一个符号的每次赋值都会产生别名。“SSA”的作用：（1）优化别名分析、（2）压缩符号的生命域、（4）降低优化开销。在此我们向发现“SSA”的研究者们致敬，有了你们的研究成果，如同站在巨人肩上，让我们走得更远。

在析码编译器中已经实现了“SSA”，下面主要介绍析码中如何将原始的“LAC”变换为“SSA”，但不会涉及“SSA”的基本概念。

我们来看几个代码例子，以下是没有控制流的代码：

```
a = 0 ;  
b = 0 ;  
a = b + a ;  
b = a + b ;  
a = b ;  
b = a ;
```

对应的 LAC，已经具有“SSA”特性。

```
a0 = 0 ;  
b0 = 0 ;  
a1 = b0 + a0 ;  
b1 = a1 + b0 ;  
a1 = b1 ;  
b2 = a1 ;
```


Middle-Para implementation

上面这种没有控制流的代码，其“SSA”变换非常简单，只需要根据 2 个原则处理：（1）对于引用，取上一次别名、（2）对于赋值，生成新的别名。跟之前这种代码相比，有控制流的代码处理起来要复杂很多，我们看一下代码例子：

```
a = 0 ;
b = 0 ;
if ( ... ) {

    a = b + a ;
    b = a + b ;

    if ( ... ) {
        a = a + a ;
    }

    a = a ;
}
if ( ... ) {
    a = a + a ;
    b = b + b ;
}
a = b ;
b = a ;
```

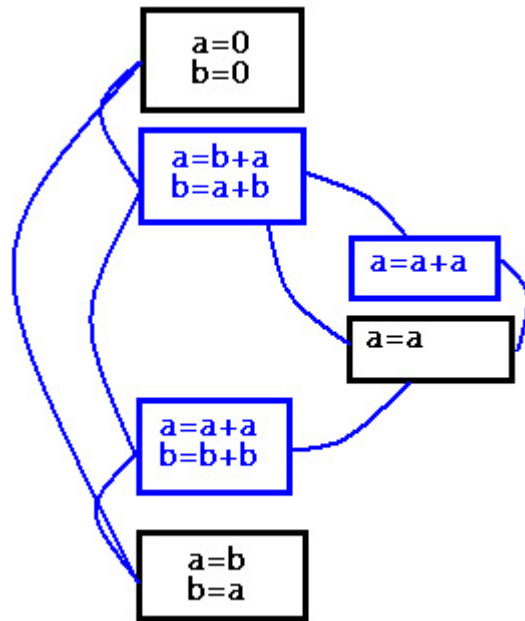
对应的 LAC，已经具有“SSA”特性。

```
a0 = 0 ;
b0 = 0 ;
if ( ... ) {
    a1 = b0 + a0 ;
    b1 = a1 + b0 ;
    if ( ... ) {
        a2 = a1 ;
    }
    a3 = SSA_MEG(a2,a1) ;
}
if ( ... ) {
    a4 = SSA_MEG(a3,a0) ;
```

```
b2 = SSA_MEG(b1,b0) ;  
}  
a5 = SSA_MEG(a4,a3,a0) ;  
Middle-Para implementation
```

```
b3 = SSA_MEG(b2,b1,b0) ;
```

以上代码中可以看出，插入“SSA_MEG”的位置都是在“基块”控制路径末端，
如图 G4.3.1



G4.3.1

我们将这种情况想象为两个矩形间的碰撞，当“基块”与“控制点”碰撞时，或“控制点”与“控制点”碰撞时我们将“SSA_MEG”插入到发生碰撞的位置。每次由“基块”进入“控制点”时，或由“控制点”退出到“基块”时，碰撞都会发生。这些信息记录在“ANLDR”中，碰撞发生后有两种情况：

1. 检查触发的区域，如果是在一个“控制点”中触发了碰撞，以该触发点为始点到第一个别名为终点，将所有有效别名加入到集合 SSA_MEG 中。
2. 如果碰撞区域不属于“控制点”，将之前的“控制点”变换为“CFF”节点，将 CFF 内所有有效别名加入到集合 SSA_MEG 中。

针对“ANLDR”共有三个操作：

1. DRG add
2. DRG get
3. CFF generate

符号赋值对应 “DRC add”，它有以下 2 种操作：

Middle-Para implementation

1. 当前作用域与当前记录作用域不相等时，生成 CFF。
2. 当前作用域与当前记录作用域相等时，生成 alias。
3. 无论何时，每次调用都会生成一个别名。

符号引用对应 “DRC get” 有以下 5 种操作

1. 当前作用域与当前记录作用域不相等时，生成 CFF。
2. 当前记录是 CFF 时，将其 CFFSET 转换为 SSA FORM，返回。
3. 当前作用域与当前记录作用域相等时，搜索 CFF，如果没有搜到 CFF，并且与当前记录同属一个 LGA，直接取别名。
4. 当前作用域与当前记录作用域相等时，搜索 CFF，如果没有搜到 CFF，并且与当前记录不属一个 LGA，循环直到作用域不相等，取的同一层各个不同 LGA 的最后一个记录的别名，当循环直到作用域不相等时，取得记录别名。生成 SSA FORM 返回。
5. 如果搜到了 CFF，将 CFF HEAD 为始点 S，到当前记录为终点 P，取区间[S,P]中所有“有效别名”，生成 SSA FORM 返回。

CFF generate 将同一层的干扰符生成 CFF，可以确定，当前作用域与当前记录作用域不相等时，必定遇到了干扰符（碰撞）。

◇ 名词解释

1. 干扰符：控制点，CFF 的生成根据干扰符号进行。
2. DRC：全称是 “Define-Reference Chain”，记录了某个符号的定义与引用的情况，多数语言的符号只对应一个定义节点，多个引用节点。每一个符号都有一个 DRC，保存在 ANL 中。其目的是为了准确地生成满足 SSA 规则的别名。
3. 有效别名：同一作用域，同一 LGA，最后一条记录的别名为有效别名
4. 无效别名：同一作用域，同一 LGA，非最后一条记录的别名为无效别名。
5. CFF:一个作用域内所有有效别名集合。

Middle-Para implementation

```

void gtkings () {
    int a ;
    if ( a + a ) {
        if(a+a){
            a = a +a ;
            a = a +a ;
        }
        if(a+a){
            a = a +a ;
            a = a +a ;
        }
    }
    a = a + a ;
}

```

基于以上代码，生成 SSA 过程，析码内部数据状态，图 G4.3.2。

0, 0, func, 0, 0, 0	0, 0, func, 0, 0, 0
1, 1, if, 1, 0, 0	1, 1, if, 1, 0, 0
2, 1, if, 1, 0, 0	2, 1, if, 1, 0, 0
3, 1, if, 2, 0, 0	3, 1, if, 2, 0, 0
4, 1, if, 2, 0, 0	4, 1, if, 2, 0, 0
5, 0, func, 0, 0, 0	5, 0, func, 0, 0, 0

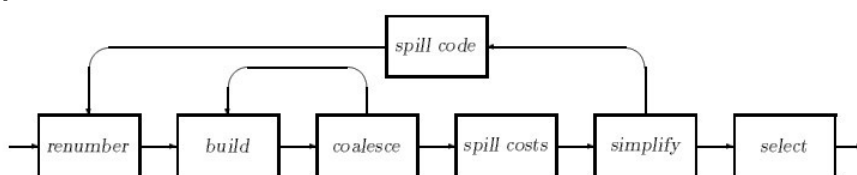
CFF

G4.3.2

Middle-Para implementaion

A4.5 Register Allocator based on Lgnosia CODE

编译历史上第一个基于图颜色算法实现的分配器是由“Chaitin”等人在 IBM 研究中心完成的，当时他们为 PL.8 编译器实现了一个代号为“Yorktown”的寄存器分配器。该分配器一次运行有六个步骤，析码的寄存器分配模块基于“York-Town”模型实现：



G4.5.1

本文并不会针对“York-Town”本身展开详细讨论，其可以在http://www.tok.cc/wiki/doku.php?id=projects:compiler:regoc_%E5%AF%84%E5%AD%A8%E5%99%A8%E5%88%86%E9%85%8D获得更多细节。在一个例程中，一个变量“i”可能会在很多位置被使用很多次，一般地，分配器为“i”分配一个寄存器，并将一直占用，生命域会影响到整个例程。这只是一个变量的情形，如果将这种情况推广到任意数量，我们会发现情况要变得复杂得多，因为会有大量生命域“相交”。就是说，在最坏情况下，干涉图 G 中所有节点的度都会大于等于 K，最终结果将导致，目标代码中充斥着大量“S/L”指令。生命域压缩是指，将生命域相交的情况减到最少，减少干涉图 G 中节点的度，最终减少“S/L”指令开销。在析码中“SSA”对寄存器分配起到了关键作用，其提供了一个有效的手段，它规定所有变量只赋值一次，第二次赋值将会产生一个新的实例。显然，一个生命域，只对应一个 SSA，因此我们可以在这一个非常直观的规则上搜索生命域。在一个基于 SSA 实现的中间代码中，变量的赋值次数越多将会产生越多短小的生命域，提高了寄存器利用率。

析码的寄存器分配模块有以下几个步骤：

1. 以“LAC”函数为单位，对生命域进行编号，也就是识别生命域过程。对于遇到的每一个“LAC_L_DELT”或“LAC_R_DELT”节点都保存在“LiveScopeMonitor”之中，对于 LAC_L_DELT 每次都会产生新编号。对于 LAC_R_DELT 需要取得该生命域的编号。同时“LAC_L_DELT”与“LAC_R_DELT”具有 LAC 引用链关系，保存在 LAC 的 refchain 之中。
2. 函数内部的生命域全部识别之后，建立干涉图，一个生命域为一个节点。如果两个生命域相交，则在它们之间建立一条边。
3. 干涉图建立之后，对图进行染色，颜色的个数是某体系可编程寄存器的个数。在对于干涉图染色时，如果遇到无法染色的情况，需要将某一个生命域分裂，需

要分裂的生命域保存在"LiveScopeMonitor"之中，其中可以找到指向“LAC”节点的指针。生命域分裂是对“LAC”的操作，遍历其引用链，将其类型由原来的“LAC_L_DELT”、“LAC_R_DELT”、改为“LAC_L_MEM”或“LAC_R_MEM”。

Middle-Para implementaion

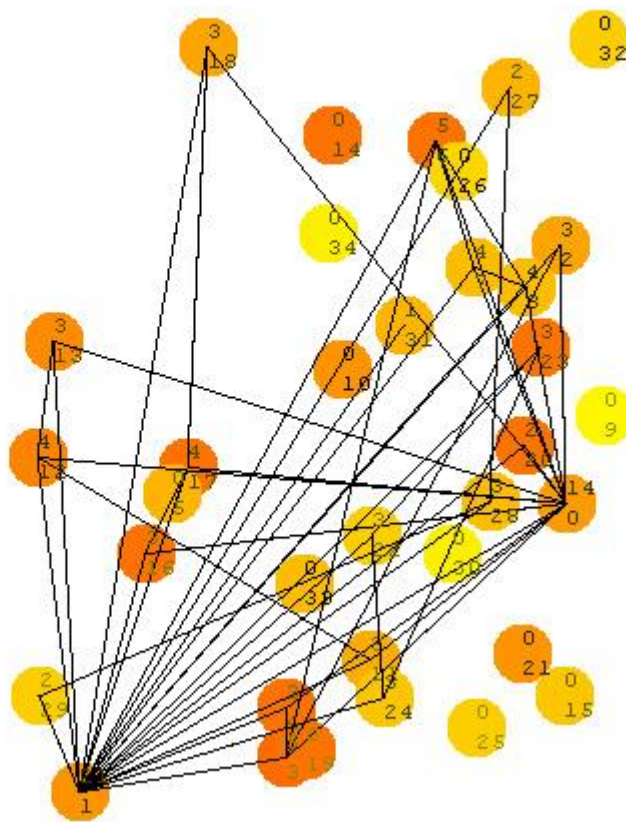
A4.5.1 析码编译器寄存器分配器实验

析码编译器的寄存器分配模块核心基于“York-Town”模型，其将复杂的生命域特征建模为直观的无向图染色问题，“York-Town”的核心是通过一系列过程分析中间代码，从中将其生命域之间的关系抽象为一个无向图图，然后对该图染色，从而确定寄存器的分配策略。以下是简单的实验，先来看一个列子，下图由析码编译器在解析中间代码过程中实时生成。

我们来看一下“YorkTown”分配器的工作情况，图中节点表示生命域，节点之间的边表示两个生命域相交。节点中间有两个数字，从上到下依次是：邻接点个数、生命域编号。当染色总数为3时干涉图的染色情况如图 G4.5.1。

```
Semo C()mpiler Compiling-Render  
(C)Techniques of Knowledge  
an open source group since 2008
```

```
interference-graph  
live scopes : 35
```



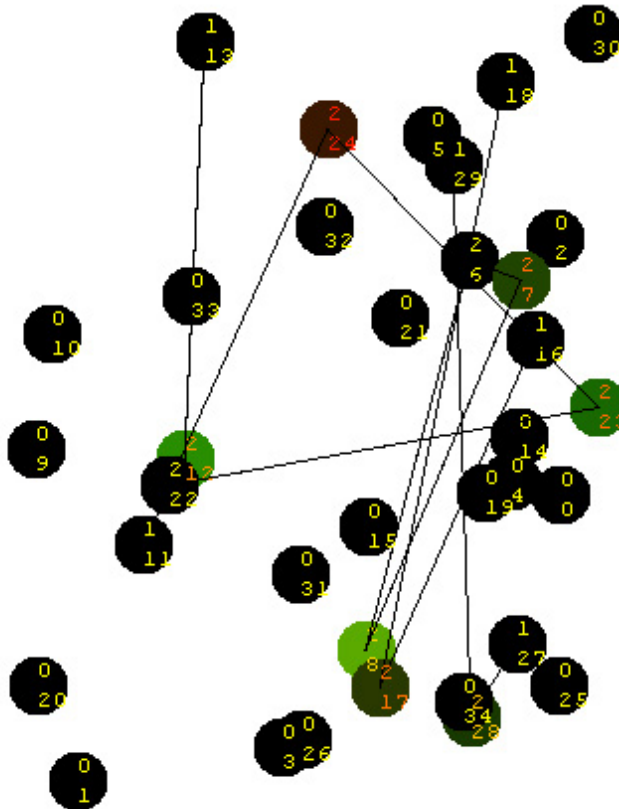
G4.5.1

Middle-Para implementaion

黄色表示染色失败了，“YorkTown”中有一种情况，如果一个节点有N条边，染色总数有N-1个，则无法染色。分配器这时估计不能有一种策略可以满足所有干涉关系。这时的做法是选择一个生命域将其“分裂”到其它存储器之中，然后重建干涉图，直到完成染色为止。把所有度大于等于颜色总数的节点分裂之后，干涉图可以顺利染色，如图 G4.5.2:

```
Semo C()mpiler Compiling-Render  
(C)Techniques of Knowledge  
an open source group since 2008
```

```
interference-graph  
live scopes :35
```



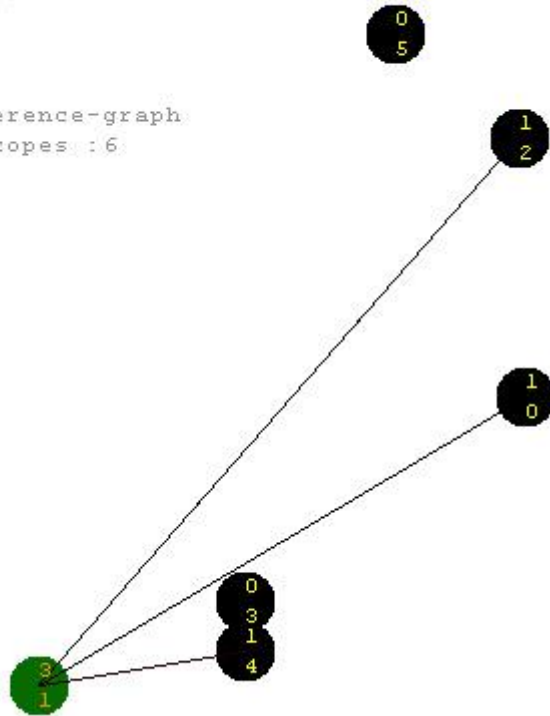
G4.5.2

上图是经过分裂之后的干涉图，每个节点都被染色，并且与其邻接点颜色唯一，这个图用于寄存器分配已经没有障碍了。由于生命域分裂的缘故，上图中有边节点数量比之前没有分裂之间少了很多。可以发现，分配器可以保证在寄存器资源很极端的情况下尽可能多的增加寄存器的分配次数，这一点同时还得利于“SSA”，它增加了生命域的密度，换句话说，通过减少生命域的重叠，增加了寄存器利用率。

Middle-Para implementation

```
Semo C()mpiler Compiling-Render  
(C)Techniques of Knowledge  
http://www.tok.cc
```

```
interference-graph  
live scopes : 6
```



G4.5.3

“如果一个节点有 N 条边，颜色总数有 $N-1$ 个”这个说法在多数情况下是正确的，只是有一种特例，当节点有 N ($N > 2$) 个邻接点，颜色总数有 2 个，如果 N 的邻接点之间没有边存在，这时该图仍然可以被染色。很明显 “ $2 \leq N-1$ ”，与定义矛盾。让 “York-Town” 去识别这个特例并没有多大意义，因为它产生的条件非常极端。不过从优化角度来说，“York-Town” 毕竟是 30 年前的研究成果，其自身还有很大的优化潜力。编译理论，经过几十年的发展，针对优化寄存器的分配策略出现了很多理论，不过万变不离七宗，多数都是在 “York-Town” 基础上的扩展。

Middle-Para implementaion

A4.6 Parameter passing specification

这里的参数传递是指在汇编层面两段代码之间相互调用时，数据如何传递的问题。关于这个问题，不同的编译器都有自己一套机制。例如有些编译器只用寄存器传递参数，而有些编译器有时候使用栈有些时候使用寄存器传递参数。目前国际上有三种常用的标准：__stdcall、__cdecl、__fastcall，只有基于同一套标准的二进制才可以正确地调用并执行。

析码编译器支持以上提到的三种标准，出此之外还至支持：__armcall，这个名称只是在析码编译器中的定义，它的规则与 ARMCC/TCC 类似。因为用于析码编译器试验的软件环境由 ARM 工具链搭建而成，为了能与其兼容，析码的参数传递机制支持 ARMCC/TCC 标准。

■ ***__stdcall***

参数自右到左压栈传递，函数返回前清空堆栈。

```
__$STACK.TOP INIT 2
__$STACK.TOP-2 IN 1
__$STACK.TOP IN 2
__$CA ...
```

■ ***__cdecl***

参数自右到左压栈传递，函数调用者清空堆栈。

```
__$STACK.TOP INIT 2
__$STACK.TOP-2 IN 1
__$STACK.TOP IN 2
__$CA ...
__$STACK.TOP UNIT 2
```

■ ***__fastcall***

使用寄存器 ECX、EDX 传递参数，其余参数按照自右到左压栈传递，函数返回前清空堆栈。

```
__$STACK.TOP INIT 2
__$STACK.TOP-2 IN 1
__$STACK.TOP IN 2
__$PA R0
__$PA R1
__$CA ...
__$STACK.TOP UNIT 2
```

Middle-Para implementaion

■ *__armcall*

参数个数小于等于 4，依次使用寄存器 R0、R1、R2、R3 传递参数。

参数个数大于 4，第 5 个参数之前的参数用寄存器传递，第 4 个之后的参数使用栈传递参数（减堆栈），函数调用者清空堆栈。

\$\$STACK.TOP INIT 2

\$\$STACK.TOP-2 IN 1

\$\$STACK.TOP IN 2

\$\$PA R0

\$\$PA R1

\$\$PA R2

\$\$PA R3

\$\$CA ...

\$\$STACK.TOP UNIT 2

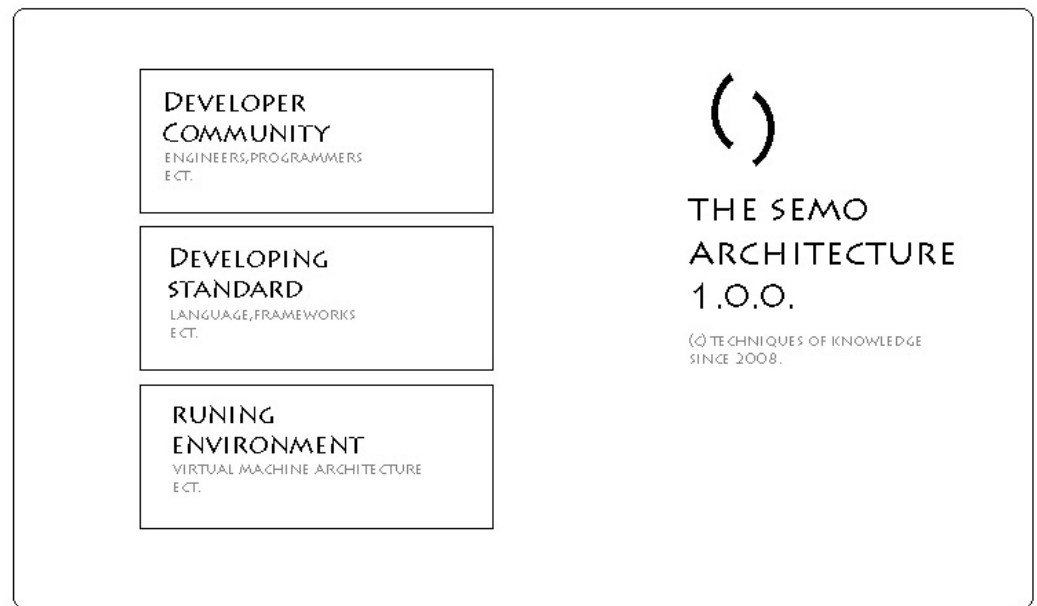
Appendix A

A1.1 Hello world from Semo Compiler

7F 45 4C 46 01 01 01 00	00 00 00 00 00 00 00 00	ELF
01 00 28 00 01 00 00 00	00 00 00 00 00 00 00 00	(
80 00 00 00 00 00 00 02	34 00 20 00 00 00 28 00	4 (
05 00 01 00 53 45 4D 4F	20 43 28 29 4D 50 49 4C	SEMO C()MPIL
45 52 20 30 2E 33 2E 30	20 42 75 69 6C 64 20 2C	ER 0.3.0 Build ,
20 28 43 29 54 65 63 68	6E 69 71 75 65 73 20 6F	(C)Techniques o
66 20 4B 6E 6F 77 6C 65	64 67 65 20 7C 20 68 74	f Knowledge ht
74 70 3A 2F 2F 77 77 77	2E 74 6F 6B 2E 63 63 00	tp://www.tok.cc
00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	
00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	
00 00 00 00 00 00 00 00	01 00 00 00 03 00 00 00	
00 00 00 00 00 00 00 00	48 01 00 00 21 00 00 00	H !
00 00 00 00 00 00 00 00	00 00 00 00 28 00 00 00	(
0B 00 00 00 01 00 00 00	06 00 00 00 00 00 00 00	
48 01 00 00 00 00 00 00	00 00 00 00 00 00 00 00	H
00 00 00 00 28 00 00 00	11 00 00 00 02 00 00 00	(
00 00 00 00 00 00 00 00	C6 01 00 00 40 00 00 00	Æ @
00 00 00 00 00 00 00 00	00 00 00 00 10 00 00 00	
19 00 00 00 03 00 00 00	00 00 00 00 00 00 00 00	
69 01 00 00 5D 00 00 00	00 00 00 00 00 00 00 00	i]
00 00 00 00 00 00 00 00	00 2E 73 68 73 74 72 74	.shstrt
61 62 00 2E 74 65 78 74	00 2E 73 79 6D 74 61 62	ab .text .symtab
00 2E 73 74 72 74 61 62	00 00 2E 74 65 78 74 00	.strtab .text
24 74 00 24 64 00 42 75	69 6C 64 41 74 74 72 69	\$t \$d BuildAttri
62 75 74 65 73 24 24 54	48 55 4D 42 5F 49 53 41	butes\$\$THUMB_ISA
76 31 24 4D 24 50 45 24	41 3A 4C 32 32 24 58 3A	v1\$M\$PE\$A:L22\$X:
4C 31 31 24 53 32 32 24	7E 49 57 24 7E 53 54 4B	L11\$S22\$~IW\$~STK
43 4B 44 24 7E 53 48 4C	24 4F 53 50 41 43 45 24	CKD\$~SHL\$OSPACE\$
50 52 45 53 38 00 01 00	00 00 00 00 00 00 00 00	PRES8

Appendix A

A1.2 About the SEMO system



一切尽在不言中。

Appendix A

A1.3 About the Techniques of Knowledge Group



“壳”——条条框框，既有模式，“突”——股肱之力，破壳而出。

蕴含着生命力的
“突壳”，象征着
创新与进取的力
量。