编译原理申优文档

常浩轩

19373384

Institute of Artificial Intelligence

2021 年 12 月 22 日

目录

1 开始之前

1.1 脚踏实地

在一切的一切开始之前，我们应该首先面对的是文法解读的工作，即阅读理解即将要处理的程序设计语言的文法定义和语义约束，编写合法的样例程序代码覆盖之。提交的样例程序将被建立公共测试库并向课程的所有同学公开——对于每一个接触过此类程序设计任务的人来说，都应该能够直接的理解这样工作的重大意义。一个完整的，提供正确输出的，公开的测试样例库对于程序设计中正确性的检验和错误的调试具有非凡的意义，能够极大的减少工作量。

因此，就我们的情况而言，在整个课程实验开始后的第二个任务，即完成文法解读后的第一个任务，并非是直接投身于编译器的设计或编写，而是分析，整理和组织了所公开的样例库。并且随着任务的进行，对其的改进和完善也在同步的不断进行着。毫不夸张的说，一个高度可用的测试程序库就是我们在编写程序过程中脚下坚实的土地。每一次进行一些或小或大的改

动，如果能够快捷方便的使用测试库对改动后的代码进行测试，那么当得到了通过的结果时我们就能够以相当的自信相信添加的改动是在工作的，并且没有对先前的部分造成影响。这对于快速的代码迭代是极其关键的，尤其体现在优化任务的阶段。当然，这种快速迭代与我们的优化模块设计有关，通过精心的设计使得其能够快速的进行迭代，这将在下文中进行说明。

下面将会说明为了构建这样的坚实的后盾我们所采取的行动。

1.1.1 整理测试样例

由于在编译的实验中涉及到多个阶段，主要包括词法分析，文法分析，错误检查和代码生成，在每一个阶段中为了对编译器进行测试需要的输入数据和期望的输出数据是不同的，因此我们需要首先将样例库整理成适应此需求的结构。

在我们的设计中，将其划分为了程序本身和数据分别存储。这样做的动机在于同一个程序在各个测试阶段均可以作为样例，仅有其输入和输出可能不同，因此将其分开是完全符合逻辑的。每一个程序将通过其名称对应三组输入输出，其中由于错误处理比较特殊因此在测试库中未能包含之，分别为词法分析，文法分析和代码生成的输入和期望输出。

随着实验的推进，我们还对程序进行了内容的改正，包括未定义行为的消除，求值顺序不对应的修正等等，这些改动参照了课程组给出的改正方式。同时由于 Mars 的设计问题，还修正了输入文件的格式使得每一个输入数据单独占据一行，来避免 Mars 在解析时出现错误。

同时，我们始终注意着一个细节，即不改动文件的其他格式，尤其是行尾符号等容易导致问题的部分，以期让编译器总是面对最复杂的情形，来保证其能够得到充分的测试。幸而由于将会在下文中说明的原因，相关的问题从未在我们的编译器中成为一个问题。

1.1.2 第一版测试程序

在词法分析作业开始之初，我们便着手构建了第一个版本的测试程序，使用 Rust 语言编写。此版本的测试程序支持我们的样例库组织格式，并能够根据命令行参数使用指定的编译器和编译命令进行编译，并使用多个线程快速的对编译得到的编译器进行指定阶段的测试，即其支持各个阶段的测试，不仅仅是词法分析阶段。

尽管设计还算中规中矩，并无硬伤，但由于 Rust 环境并不广泛可用，因此此测试程序在后续的过程中逐渐被弃用。

1.1.3 第二版测试程序

在逐渐弃用 Rust 评测程序的同时，我们进行了第二个版本的测试程序的开发，同时此版本也不断迭代改进并沿用至今。此版本的测试使用 shell 编写，且采用了在大多数机器上能够提供的 sh shell 的语法进行编写，以便最小化依赖，最大化兼容性。

在编写中，我们着重优化了易用度，直接使用编译好的编译器进行测试，并且在出现错误时给出尽可能详细的信息以便诊断问题。例如我们将会捕获编译器输出的日志中的错误及以上级别日志并打印，从而减轻查看文件的时间开销。

1.1.4 构造 Docker 镜像环境

尽管我们在开发中工作的环境同样为 Linux，与评测机的环境仍然存在不同，例如由于喜好问题我们采用的环境中使用了最新的 GCC11 编译器，为了尽可能的避免出现被环境差异带来的难以追踪的疑难问题困扰，我们诉诸虚拟环境以求解决方案。然而课程组提供的 Docker 镜像体积过于庞大，且在百度云上共享难以使用，因此我们自行学习了 Docker 相关的知识并进行了轻量级 Docker 镜像的搭建。通过采取各种手段，包括大幅精简了课程组提供的修改版 Mars 的体积（我们将精简后的 Mars 命名为 Sram），我们最终成功的将提供 GCC8.1.0 的完整镜像压缩到 150M 以内，并将提供 Mars 模拟功能的完整镜像压缩到 40M 以内。通过在虚拟环境中进行测试，我们可以接近评测机的视角，从而能够捕获和排除测试环境带来的干扰。当然，幸运的是，也与我们的设计有关，我们同样未遇到任何相关的问题。

1.1.5 搭建持续集成环境——集测试之大成

最后，我们使用了重量级的工具——持续集成——基本终结了测试这一问题。在此过程中，我们在本地使用 Gitea 搭建了一个私有的代码托管服务器，并相应的配置了到同样在本地运行的 Woodpecker-ci 持续集成工具的集成，利用了先前搭建的 Docker 镜像和第二版测试程序完成了一整套的持续集成的建立。

通过建立持续集成环境，我们的每一次向代码仓库的提交被推送后都将在持续集成工具中自动进行一次完整的测试，确保每一个被合并到正式分支上的代码均通过了正确性的检查。同时，我们的样例库同样建立了一个仓库进行管理，并使用持续集成工具进行完整性的校验，即确保每一个样例库的正式版本都包含完整且对应的各阶段输入输出文件。

持续集成工具在配置上开销了一定的时间，但在完成后即可几乎不需要维护的进行工作，并显著的减轻工作量，为我们提供了很大的帮助。

1.2 继往开来

一个可靠的测试支持能够让我们无畏的向前迭代，而架构的设计将会决定向前推进的代价。因此为了让开发，尤其是在本实验中这样需要不断进行增量式迭代的开发，顺利的向前推进而不会出现大量冲突和重构的需求，合理的架构设计至关重要。

可以自豪的说，尽管我们曾数次声称要重构，但是我们的编译器迭代至今并未事实上经历任何的一次重构。这与我们的设计模式是密不可分的。尽管不断由新的模块加入，甚至本身的流程设计中出现了分支——毫无疑问，这看起来像是并未删除原有代码的重构，像是自欺欺人的把戏，但是应该说明的是，从入口到出口，任何一条路径，相连的模块的组合，都是能够正确的工作的。这绝对不是一种重构。

我们最终的架构如下图所示。注意下图中的结构被有意的的复杂化了，呈现了编译器全部可能的工作流程，而在实际中通常不会采用如此复杂的工作流程，总体而言流程将会成为 2×2 的选择问题，即前端采取 CAS 实现还是预处理器加传统实现，以及后端采取 HLIR+LLIR 的实现还是 MLIR 的实现共四种流程的选择。这样的复杂化旨在说明我们设计的结构，尤其是前端部分，的宽松绑定：两侧的流程路径允许交叉，CAS 实现的词法分析器与传统的词法分析器可以相互替换（不过传统实现依赖首先经过预处理器），语法分析器类似。关于这些实现的具体不同将在下文中说明。

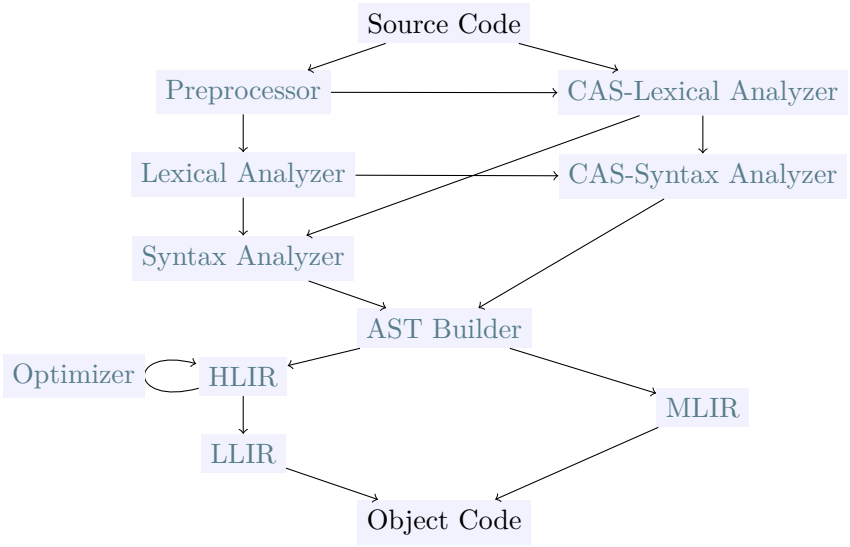


图 1: 编译器架构

为了让我们的设计能够轻松的继往开来而不是反复重构，我们需要思考的问题就是，在架构上，谁与谁应该相互联系，如何相互联系。一个根本的思路就是经常所说的“高内聚，低耦合”。我们最终设计得到的架构即是上图所呈现的流水线结构，通过保证每一个阶段使用且仅使用上一个阶段生成的结果作为输入，并独立的产生当前阶段的输出，我们使得在不相邻的流水段之间的依赖被完全的清除，从而减少了代码的逻辑复杂度。

对于无法独立工作的部分，例如错误捕获处理和符号表管理部分，我们将其封装为类，向外部提供简洁易用且稳定的接口，使得其被使用时使用方不必深入其实现内部即可实现其功能，使得其耦合度下降。

对于相邻流水段之间的数据传递，我们预先定义了完善稳定且易用的数据格式，令段内无论进行任何更改总是最终不影响向下级传递的数据的形式，从而保证了一处的修改不会影响其他阶段带来额外的开销。

关于各段的具体介绍将在下文中给出。

1.3 抉择之思

我们应该使用什么语言来完成我们的实验？对于很多人来说，这个问题可能经过了一定的思考方才得到答案，甚至我们也听闻有些同学首先选择了一种语言，经过了一段时间后转而换用了另一种的情况也在现实中存

在。就我们而言，这个问题并不成为一个问题，我们毫无犹豫的选择了使用 C++ 实现我们的编译器。其原因之一在于我们自认为我们的 Java 水平足够低，使得我们将不得不花费超过必要的时间在使用语言实现上，这对于设计的思路也将成为一种打击。

事实上，我们认为对于具有一定水平，这里的一定水平对于大多数计算机学院参与这门课程实验的同学来说应该都完全可以轻易达到，语言的选择并不是一个关键的问题。我们听到了很多关于 Java 会比 C++ 易用的评论，但我们并不以为然。诚然，Java 具有反射等 C++ 还尚未支持的高级功能，但是我们应该看到，此类的高级功能在本实验的正常需求中的需要是完全可以不存在的。我们认为 C++ 提供的标准库中的容器，智能指针等最实用的功能在课程组提供的 C++11 环境中已经基本满足我们的要求，且 C++ 具有的对指针进行直接的操作的功能使得我们可以更加轻松的实现一些仿真工作。更进一步的，在 C++11 中已经引入了相当关键的现代 C++ 的特性，充分的利用 lambda 表达式，函数或类模板等功能能够使得 C++ 同样相当易用。

总之，现代 C++ 岂会是如此不便之物!

2 流水段设计

2.1 预处理器

词法分析中几乎不存在任何值得称为困难的地方，但是每一个进行过字符串处理的人应该都能够理解在字符串处理中经常出现的各种边界情况，在细节中出现的问题等等，因此这个阶段又充满了陷阱。为了压减词法分析的工作量，我们在前端的最前方添加了在实验中未被官方要求提及的预处理器。

由于本目标语言中不存在预处理指令，因此预处理器并不具有像 C 语言中那样复杂的功能。我们对其的功能要求是要其为后续的词法分析器抽象掩盖掉平台相关的信息例如行末换行符的区别，具体的即将其替换成为统一的 `\n` 的形式，并用空格替换任何的注释内容。由此功能定义可以看到，预处理器可以被定义为一个从字符串映射到字符串的函数，且具有幂等的优秀性质。

尽管引入的预处理器是相当简单甚至有些单薄的层次，其引入明显的简化了后续词法分析器的设计和实现复杂度。

2.2 词法分析器

这里的词法分析器准确的说指的是前文中的传统词法分析器。

正如先前提到的，我们设计了统一的词法分析器的输出格式。由于在语法分析中我们将很有可能同样需要进行项目的取出和放回，且我们对一个词法成分的存储显然不能仅仅通过一个字符来完成，而 C++ 的标准库中并未提供一个自定义的流结构来处理这种问题，我们参照 C++ 的标准输入输出流自行设计了一个简单的项目流，通过模板支持任意统一的类型被放入流中，并简单的实现了类似标准输入输出流的 `get()`, `put()`, `peek()`, `unget()` 等方法，以及若干简单的实用方法，从而提供更加一致和方便的使用体验。

对于一个词法成分，我们存储其字面量，行号，列号和其类型编码，类型编码使用了强类型的枚举类型来避免意外混用，通过重载算符来提供便利的输出。这样的结构提供了完全的信息以供后续使用，因此可以确保其稳定性。

词法分析器被定义为一个从字符串到词法成分的函数，其将输入程序中的格式细节抽象并屏蔽，为后续阶段提供一个纯粹可控的词法成分流以供使用。

由于存在预处理器扫平了前方平台相关的障碍，并且注释已被删除，因此词法分析器的设计可以得到大幅的简化。我们可以安全的认为一个换行符总是一个单独的 `\n`，也不必关心任何注释带来的问题。

在具体的实现上，我们采用了比较经典的也是在课程中讲解的类似状态机的实现方式，并专门维护了变量追踪当前的行号，从而可以支持对于符号的大致定位。但由于历史遗留问题，我们并未追踪列号，因此准确的定位在此分析器的实现中是不支持的。

与课程中介绍的不同的是，我们并未采用更加谨慎的符号读取方式，而时充分利用 C++ 抽象的输入输出流中对于放回字符的支持，大胆的读入字符并在必须时直接将其退回。这样的设计不需要花费额外的注意在循环中保持各种情况下预先读取关系的正确来避免漏读或重读符号，只需要正确的处理当前一次的工作内容即可，更加直观。

2.3 词法分析器：自动生成

在实现了传统的词法分析器之后，我们还实现了更加强大也在某种意义上更加简单的自动生成的词法分析器和其自动生成工具。其函数映射定

义与传统词法分析器完全一致，保持了兼容性。

关于此工作的动机，简单概述有如下的理由。首先，我们认为传统的词法分析器在实现上太过简单，因此带来的问题是其显得太过笨重，在面临文法改动时需要付出大量的额外工作量，泛化性极差。同时由于对改动需求的存在，其大大的增加了其中出现问题的可能性。我们认为如果能够尽量的减少代码中需要改动的数量，例如对于完整实现的自动生成工具我们将不会在文法改动中需求任何的代码改动，直接将新的文法定义输入交付给生成工具便能够得到新的正确的分析器，则几乎不会出现任何错误。更进一步的，我们在实际手动实现的传统词法分析器的过程中发现其代码具有相当低的熵，即存在大量的重复部分，我们认为对于此种内容进行手工实现并非是一个合理的实践，对其进行自动化是符合逻辑的。

在实现中，我们使用了正则表达式进行词法成分的定义。我们的词法定义文件格式一定程度上参考了 lex 的源程序格式，但存在一定的改动，例如我们支持为一个词法成分定义两个名字，分别是其在程序内部的枚举类型中使用的更加易读的完整名称，另一个则是被要求的输出名称。同时，我们允许在内部名称中使用“魔法前缀”来定义具有特殊意义的词法成分，例如仅在内部使用而不会被导出的词法成分，或是代表错误的符号，例如两个连续的加号。

对于正则表达式的处理，我们采取了与课程中介绍的不同的方式。对于每一个正则表达式，我们使用递归下降法为其构建对应的语法树，并根据语法树求出各个有效字符的 `followpos`，从而直接构建出对应的 DFA。当所有的 DFA 均构建完成后，我们将会将其合并为一个 NFA，并在此基础上进行确定化得到最终的 DFA。注意到在此处我们并未实现一个最小化过程，由于经过实验我们发现对于给定的文法，我们最终生成的 DFA 仅仅包含 96 个状态，此状态数对于性能的影响微乎其微，因此我们暂时忽略了最小化。

为了尽可能的减少代码的改动量，我们设计了 DFA 的导出格式，从而允许一个固定不变的词法分析器动态的加载导出的 DFA 定义从而依此进行字符串的识别处理。这样的设计自然的允许了同一程序中存在多个不同的词法定义的分析器，同时也使得对于任何不同的文法我们不需要对分析器进行任何的改动，只需重新生成新的 DFA 定义供其加载即可，这也是讲解的分析器的实现。

在分析器中，我们的实现与课程中讲解的经典实现一致。对于冲突，我们采用了与 lex 一致的处理方式，即总是优先匹配最长的子串，在长度相同

时则根据相应的定义顺序来匹配，这也就保证了总是只会有一个词法成分被匹配。我们也会比实现的传统词法分析器更进一步的追踪列号的改变，从而可以准确的定位识别的任何一个词法成分。

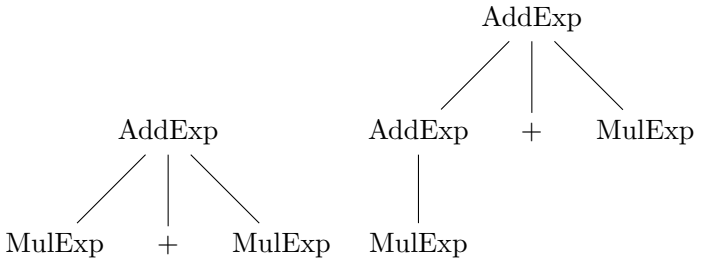
由于正则表达式的强大性能，通过合理的设计词法成分定义，我们可以自然的完成不同行末换行符的适应，注释的忽略等工作，因此并不需要预处理器的先行处理。但对于经过预处理器处理的字符串，自动生成的分析器同样能够正确的进行分析。

2.4 文法分析器

这里的文法分析器准确的说指的是前文中的传统文法分析器。

文法分析器被定义为一个从词法成分流映射到具体语法树的函数。具体语法树的形式相当简单，其为单一类型节点组织成的树，节点中包含当前的文法成分的类型标识符号，同样采用强类型的枚举类型标记，以及相关联的词法成分，此项可能为空，和一个变长数组实现的子结点列表。这样的结构具有足够的稳定性。

传统文法分析器的实现为常规的和要求的递归下降法，在实现中注意到表达式系列的文法定义存在左递归，这可能成为一个困难。在处理中，我们采用了转换为扩展的 BNF 范式形式的方法予以解决，使得同种类型的连续表达式将不再通过递归推导而是通过循环直到到达边界得到。在完成分析后，还需要将相关的树结构进行重构，使得其结构符合文法的定义。其具体操作可见下方示例。



(a) 原始结果　　(b) 调整后结果

图 2: 具体语法树调整

同时，由于在分析中可能存在需要放回词法成分的需求，我们此前实现

的项目流的放回功能可以提供充分的支持。由于因此难以避免出现预读而可能导致输出中出现未期望的项目，出于简化代码避免内部混乱的考量，我们并未在构建具体语法树的过程中进行相关的输出，而是在完整的构建之后遍历语法树进行输出。这样的设计同时使得我们可以在一个单独的位置统一的控制是否输出，避免了额外参数的传递进一步的混乱代码。

对于错误处理的支持，由于定义的错误中可能出现的在此阶段能够捕获的错误较为简单，均为右界符缺失，因此我们采用的方式是在递归下降中发现右界符缺失后在错误处理器中加以记录并自动生成一个对应的符号插入到树中，从而保证后续的工作阶段中总是得到正确的具体语法树，简化后续流程中的进一步处理如语义错误捕获。

2.5 文法分析器：自动生成

与词法分析中我们发现的问题和动机类似的，由于代码的高复杂度和高重复度，和因此带来的更大的内部问题隐患，我们决定进一步的实现一个自动生成的文法分析器和相应的自动生成工具。类似的，我们同样保证了其输入输出与原始的传统文法分析器保持一致，从而提供合理的兼容性。

我们选择实现的分析器生成器将生成 LL(k) 文法分析器，这样的选择的理由在于这样的设计较为直观，在出现问题时更好进行解决，因为文法分析器相比词法分析器其复杂程度有显著的跃升。更进一步的，为了更好的提高效率，我们生成的并不是完全的 LL(k) 分析器，而是一个自适应的 LL(k) 分析器，自适应的进行 k 值的选定。由于在实际的文法中，对于一个非 LL(1) 的 LL(k) 文法，其大多数符号的情况下仍然是 LL(1) 的，仅有少数情况下需要向前看不止一个符号来决定选择，我们生成的分析器将仅在这种情况下看更多的符号，且只要看到足够的符号就会立刻进行选择，从而尽可能的提高效率。

由于文法的复杂情况，例如我们的文法本身便不是一个 LL 文法，由于其具有左递归的性质，且存在二义性，我们需要进行对应的操作来应对这些困难。具体的，在进行 LL(k) 分析表之前，我们将会首先对文法进行自动的改写从而试图将文法改写成为一个 LL 文法。

不过应该注意到，正如在递归下降的实现中遇到的问题一样，我们被要求产生精确符合文法定义的具体语法树，但文法改写将不可避免的使得此变得不可能。为了对抗此问题，我们在变换中引入了更多的追踪信息，从而使得对于改写后的文法的每一条规则，我们都能够按照追踪信息在完成分

析后自动的将生成的结果变形回到精确符合文法定义的状态。这最终使得自动工具的应用成为可能。

对于输入，我们几乎完全支持了扩展的 BNF 范式，使得文法定义可以相当简单的编写，且自然的兼容课程组给出的文法定义。LL(k) 分析表的建立和固定不变的分析器的设计均与课上讲解的一致，因此略过不表，我们将对于树形重构的操作设计和其在文法的改写过程中的追踪方式进行阐述，这是事实上最大的困难。

我们设计了如下的操作，用于树的修正。由于时间原因，我们没有时间详细叙述此部分，目前距离提交时限还有 5 分钟。

2.5.1 Pull Up

节点将向上推送子结点，删除自身。

2.5.2 Push Up

节点将向左旋转，提升自身并改变类型。应对左递归消除。

2.5.3 Chain

节点将向下扩展并生成中间节点。应对单产生式消除。

2.5.4 PushDown

节点将向下扩展并生成更复杂的中间节点层次。应对左约减。

2.6 语义分析：抽象语法树构建

经过此前的各阶段的逐渐抽象和补足，当到达语义分析阶段时，我们将会获得一个完全正确的，精确符合文法定义的具体语法树。注意此说法成立的条件在于输入时精确符合约束的，即仅或者仅包含给出的错误类型，或者是完全正确的。则在此阶段进行语义分析时，我们可以毫无顾虑的假定接收到的具体语法树时完全符合文法的定义的，可以不加额外的检查的额进行语法制导翻译，减少了不必要的代码冗余。

在语义分析中，我们将会从具体语法树的根节点开始进行递归的遍历，在过程中将具体语法树对于原本的语言的结构的表述替换成为新声称的仅

保留语义而抽象掉了其具体的语法格式的语义表示节点，由此构造出其对应的抽象语法树。

在此部分，很难说存在任何显著的困难，但由于其位于整个编译器的最中心，具有核心的地位，其中需要完成大量的工作，其代码量显著的大，因此可以说是一个繁而不难的阶段。

在进行抽象语法树的同时，我们还将进行符号表的填写并且进行语义错误的捕获。符号标的设计将会在后文中详细介绍，其将自动的完成地址的分配等额外的工作。具体的，我们为每一种定义，包括函数定义，全局变量常量定义和局部常量变量定义，每一种表达式，包括二元运算符表达式，一元运算符表达式，短路运算符表达式，每一种语句结构，如赋值，打印等等，均定义了对应的节点并通过继承关系进行组织，根据语法制导翻译进行相应的节点的创建，递归的进行。对于变量的引用等较复杂的过程，专门生成了单独的节点以便后续进行处理。

在此处我们同样注意了进行尽可能多的编译期常量的计算和表达式的替换，对于能够进行计算的表达式，我们总是进行计算并生成为整数常量。具体的，我们将会捕获所有的字面常量，并追踪所有能够编译期求值的常数，这里的追踪通过其初值定义得到，当初值为一个能够编译期求值的表达式时，如为一个字面常量或能够转换为字面常量的表达式时，将会记录此常量的值并且直接移除此常量的定义，在后续的生成中直接使用记录后的值。对于值的记录同样将会在符号表相关的部分进行说明。为了保证正确性，我们通常不会追踪变量的编译期求值信息。但是特殊的，对于全局变量，由于文法的约束，可以保证所有的全局变量定义一定出现在函数定义之前，则就地初始化中能够使用的值仅仅包含能够编译期求值的值，且由于其为全局变量，我们能够直接在数据段中定义其初值而减少初始化带来的指令开销，因此我们将会同样追踪全局变量的编译期求值信息。但是当到达第一个函数的定义的开始时，我们将会清除所有变量的编译期常量值，来避免错误的引用。

在实现的技巧中，我们可以看到在具体语法树的树节点结构上存在较多的充分，例如对于表达式的部分，不同表达式的结构极尽相同，因此将导致大量的代码重复。为了减少这种相当有害的重复，我们在此引入了高阶函数进行简化。我们定义了一个返回函数的函数作为表达式的抽象语法树节点的构造器的构造器，通过向此高阶函数中传入目标的表达式对应的具体语法树节点类型标号，以及相关的需要的信息，在此处即为为了构建下一级

表达式需要使用的函数（例如对于 AddExp，构建下一级表达式使用的函数即为构建 MulExp 使用的函数）和当前层次表达式的计算函数（用于常量的计算和传播），则其可以自动的生成一个具有递归性质，能够处理指定的类型表达式对应的具体语法树结构的函数并返回。通过将共享的逻辑包含在高阶函数中，我们减少了代码的重复，简化的编码和后续修改的复杂度，可以说这就是现代 C++ 的使用的一个生动的例子。

2.7 HLIR

HLIR, 其名称是高级中间代码的所写, 即 High Level Intermediate Representation。在我们的设计中，HLIR 正如其名称，是一种高级的中间表示。被称为高级中间代码的原因在于其不会包含任何内存布局，寄存器分配，栈指针操作和维护等底层相关的内容，只注重功能的表述。这样的设计使得尽管并未在我们的实现中出现，但仍然保留了在不同的目标平台之间进行移植的可能性，即在不同的目标平台上将产生相同的 HLIR 序列。

HLIR 从抽象语法树进行生成，每一个抽象语法树的节点唯一固定且上下文无关的对应一条或若干条 HLIR 的指令或数据定义。这些定义可以轻松的根据抽象语法树的语义进行确定。在其中我们很可能需要一些临时的变量用于辅助我们的计算和控制，为此我们引入了临时符号。临时符号将会被按需创建，其将会需要进行内存空间的指派，但是在 HLIR 中我们并不会关心此内容。相反的，临时符号将会通过其唯一标识符，在我们的实现中为代表其的节点所存在的地址，进行区分和定位。这些临时符号将完成充当运算中的中间变量等工作。

在 HLIR 中，我们将跳转指令的跳转目标通过一个指向跳转目标指令的指针来实现，同时在创建跳转指令时将自动在目标指令的引用指令列表中添加自身，从而从一条跳转指令能够定位到对应的跳转目标，从跳转目标也可以反向找到可能跳转到其的指令。这样的设计保证了在指令列表发生变化时候仍能保持其指令间的引用关系不受影响。

特殊的，对于函数返回指令，我们并不能确定其跳转目标，因此我们无法明确的给出指针。同时对于一些指令生成的过程，由于从前到后的顺序，我们并不能总是在需要生成跳转指令时即获取其跳转目标指令的指针，例如此时可能目标指令尚未生成。对此，我们的解决方式是通过首先生成一条空操作指令作为跳转目标，后续将此空操作指令安排在正确的位置，即可完成引用。这当然会导致一些额外的空操作指令的出现，但幸而在后续的过程

中我们可以相当简单的将其优化掉。

在 HLIR 的组织中，对于全局的数据定义，通过根据数据占用的空间大小的对齐规格赋予相应的优先级，则可以通过排序操作正确的安排每一条数据定义的顺序来避免任何的对齐问题。

同时，尽管生成 HLIR 指令的过程中我们将所有指令一视同仁的列入一个线性列表中进行组织，但是在实际的 HLIR 结构的构造中，我们将会将其划分为基本块。根据上文我们的设计，可以相当容易的确定一条指令是否为跳转指令，以及一条指令是否被跳转指令所引用（通过上文提到的引用列表），因此入口点的确认相当轻松，据此可以直接切分基本块。

在 HLIR 的生成中，我们注意直接的对可能的优化项目进行优化。例如对于数组访问，对于元素大小和行大小的乘操作总是被尽可能的替换为位移操作，且进行相关的运算当且仅当其有意义，例如加 0 总是不会被生成指令。

2.8 LLIR

与 HLIR 相对应的，LLIR 表示 Low Level Intermediate Representation，即低级中间代码。在低级中间代码中，将会包含全部的具体内存器分配和维护操作，并且将会涉及全部的内存分配。在此，我们首先说明我们在此 HLIR–LLIR 工作流程中的内存分配方式。

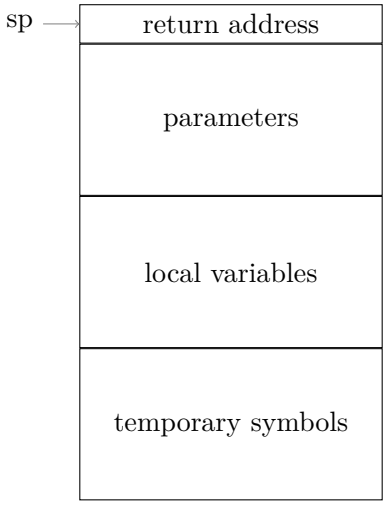


图 3: HLIR–LLIR 中的栈帧结构

栈帧中栈指针将指向当前栈帧对应的返回地址。在我们的设计中，仅有函数将具有栈帧，而局部内部的块将不具有栈帧。每当进入函数时，将会将其返回地址存入栈帧顶部。接下来沿地址递减的方向即栈增长的方向依次安排全部的形式参数，局部变量和临时符号。对于内部块中的局部变量，符号表中填入信息时将会自动的完成其内存的指派，不同块中的变量可能共用内存空间，由于超出作用域的变量将消亡，这样的设计能够减少栈空间的使用并保证正确性。临时符号的分配将是动态的，即每当遇到一个新的临时符号，将会增长当前栈帧的大小并为其指派地址。由于栈指针将总是指向栈帧顶部，此处的增长将不会对其造成任何影响，也不会影响其他变量的寻址。且由于形式参数和局部变量在符号表中已经确认占据的空间大小，临时符号将不会出现指派地址冲突从而破坏其数据。

在寄存器分配上，我们共选择了 24 个寄存器进行分配，采取了 Aging 算法进行近似 LRU 的寄存器分配方式。关于此算法的实现，在《计算机组成》和《操作系统》课程中已经进行了相当详细的说明，在此不再赘述。为了确保正确性，避免多重访问方式无法识别导致的错误，我们保证不会为任何数组成员进行寄存器的指派。对于全局，局部的标量型变量和临时符号，我们总是对其进行寄存器的指派和分配，视情况进行寄存器值的保存和加载。具体的，对于出现寄存器指派目标替换的情况，我们将会检查先前的寄存器是否被标记为修改过的，若是则将将其中的值存回指派目标的内存中，否则将不进行任何操作；对于读变量，我们将会检查当前寄存器是否被标记为无效或是发生了替换，如是则说明当前的寄存器值不正确，则将从内存中进行加载，否则不需进行任何操作；对于写变量，我们将不会进行任何的寄存器值加载，因为其值将立刻被重写，但是将会标记寄存器为修改过。每当到达基本块的结束处，我们将会将所有的修改过的寄存器进行写回，并重置分配，从而保守的确保正确性。

指令生成繁而不难，与 MLIR 有异曲同工之妙。

2.9 优化器

为了在优化中保证正确性，我们对优化器进行了相当强的限制。全部的优化器被设计为一个纯函数，此函数被定义为一个从 HLIR 到 HLIR 的映射，其仅接受一个 HLIR，不可对此 HLIR 进行任何的改变，并且生成一个新的 HLIR 进行返回，也不能访问任何非常量的全局变量。同时，我们要求任何一个能够正常工作的 HLIR 在经过任何一个优化器后仍然要是能够正

常工作的，且两者必须是等价的。这就要求任何的优化器不能对优化器的调用顺序有任何的假定，即可以因为一个优化器并未在另一个优化器之前被使用而出现优化性能下降或者无优化甚至负优化，但不可以产生错误。这样的限制相当严格，但是使得我们可以放心的对任何优化器进行开关，调整顺序等。

具体的，我们设计了如下的优化器。

2.9.1 基本块内优化

字符串输出合并 对于同一个基本块中的多条中间不被变量输出打断的字符串输出语句，我们可以将其进行合并为一条语句，每个合并将可以减少至少 4 条指令的开销。

临时符号常量传播 由于设计原因，所有的字面常量将会首先需要通过临时符号再传递给表达式或是变量。为了加快此流程，我们引入了临时符号的常量传播。通过在一个基本块中追踪临时符号的常量状态，我们可以将很多的语句进行简化，减少寄存器占用和访存次数。具体的，立即数加载将会引入常量状态，符号复制，源表达式均为常量的表达式等情形将会传递常量，而从内存加载，读入，非常量表达式等将会移除常量。通过使用常量代替进行代码生成，并删除先前不再需要的中间符号赋值，可以减少相当数量的代码。注意为了正确性，需要在基本块结束时清空常量表。

2.10 MLIR

MLIR 即中级中间代码，Middle Level Intermediate Representation，其是一种既包含高层语义又包含底层信息的中间代码。由于其直出结构，其实现简单快速，但是性能极差。

每一条的 MLIR 精确对应一条或数条目标代码。在下文中我们将会选出设计比较复杂或比较有意思的中间代码并加以说明。

函数定义 在函数定义中，实际上并不存在任何的特殊操作，仅仅对于第一个函数的定义，由于文法要求函数一定出现在全局变量定义之后，此处将出现被执行的第一条代码，我们将在此处插入开始代码：一条 jal 跳转到主函数，并跟随结束程序的指令。

函数边界 由于允许在无返回值函数中省略返回语句，我们需要确保函数能够正确的进行返回，因此对于每一个函数，我们均在最后一条语句之后添加了这条中间代码，其内容为跳转到 ra 寄存器中的地址，即返回。由于在函数调用中保证了 ra 寄存器中始终保存正确的返回地址，这总是奏效的。

字符串输出 由于 MARS 允许混杂数据段和代码段的定义，因此我们在输出字符串的中间代码生成目标代码时才将会将其字符串常量加入数据段。这样的设计也自然的避免了地址不对齐的问题。

变量访问 数组的处理是一个较为复杂的问题，为了简化起见我们进一步统一了标量类型和数组类型的存取操作。对于任何的读或写操作，我们均将会首先生成指向将要操作地址的指针，此指针可以简单的通过基地址和下标定义的偏移量进行计算，随后根据是否提供了足够的下标信息来决定是解引用此指针获取值或者直接将此指针值作为结果。当然，对于写操作，我们将要求必须提供恰好的下标维数。

短路表达式 对于短路运算符，我们将其拆分成为左右两个子表达式，计算左侧后产生一条条件跳转语句，若结果达成短路则跳转到当前表达式的计算结束位置，自然跳过了右侧表达式的计算，则完成了短路特性。

3 流水线之外

3.1 符号表

3.1.1 总体需求

符号表中需要记录在编译的分析过程中收集的，在综合过程将会被用到的符号或称标识符和与之相关的信息。比较容易直观的看到的信息有：

标识符本身 毫无疑问，需要存储标识符本身，同时出于能够产生更加可读可理解的日志和错误信息考虑，还应该包括更多的信息例如标识符在代码中出现的位置等等。

类型信息 由于编译的源语言为类 C 的语言，表现为一个静态类型的语言，因此在符号表中需要存储符号具有的类型信息，且其类型信息的组织形式

应该便于进行类型一致性的检查。注意到类型一致性的检查不仅包括类型相等的判断，还应该包括可以对于从一个类型到另一个类型是否可以进行转换，是否可以隐式转换，是否可以安全的进行隐式转换等进行方便的判断。关于这些的区别直观起见可以举出如下的例子：

- 不可以进行转换：例如从函数类型 void(*)(int, int) 到 void(*)()
- 不可以进行隐式转换：例如从 int* 到 int
- 不可以安全的进行隐式转换：例如从 int 到 char

同时存在的另一个问题是类型转换并不是一个简单的关系，上述的情形中存在非对称的关系，即例如从类型 A 能够转换到类型 B 并不一定意味着可以从类型 B 转换到类型 A，同时也不一定具有传递性。在设计中需要考虑到这些复杂情况的易用实现。

内存分配信息 应该注意到，将要处理的语言并不能够完全由静态内存分配完成其运行时内存的分配管理，但是由于每一个允许使用的数据类型均具有完全确定的大小，因此动态分配只出现在函数的运行栈帧上，可以为所有的非全局变量分配相对栈指针的地址。我们也应该在编译时完成全部的内存分配任务，来减少完全不必要的运行时开销。因此存储表示变量的标识符所被指派的地址是需要的。同时应该注意到我们将不会为全局变量分配地址，而是将其置于数据段中通过标签访问，需要能够在内存分配信息中区分全局和非全局变量。

层次信息 尽管我们需要处理的源语言在设计上极大的限制了程序块间的调用和访问，例如并不允许在函数内部进行程序块的定义并相互调用，因此极大的简化了变量访问，也无需在课上介绍的 DISPLAY 区的设计，但是出于完整性的原因我们还是计划了完整的栈式符号表，将会为每一个块分配一个独立的符号表帧。因此在一定的程度上我们仍然需要类似 DISPLAY 区的设计，尽管我们不需要显式的提供之。于时层次信息同样被需求。

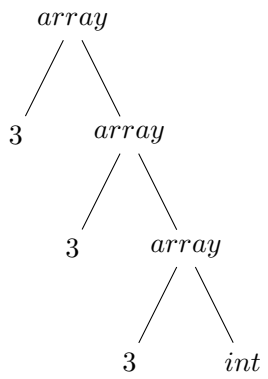
编译期求值信息 由于需要处理的语言是一个具有修改的 C 语言子集，其一个明显的改动为支持了使用常量进行数组维度的定义，因此我们需要在编译期进行尽可能多的求值工作以允许在后续遇到使用常量定义的数组维度能够进行正确的处理。因此我们需要保存其编译期值。

3.1.2 标识符

为了提供尽可能全面的标识符信息，我们采取了一个简单的策略，即在符号表中标识符对应的项中存储整个标识符的词法项目信息。因此可以直接的从中获取行号信息和其字面形式。这已经是能够提供的最全面的关于标识符本身的信息了。

3.1.3 类型信息

这一部分是整个设计中改动最大的。在先前的构思中，我们采用了相当简单的方式组织类型，即每一种类型使用一个单独的表示码标记，并添加其特有的信息，例如对于数组加入其维数。这样的设计表述能力不强，且由于未引入指针类型遇到了较大的挫折，同时在实现类型检查时复杂度骤增难以实现。后续的设计进行了较大改动，最后以如下的方式作为结果。
类型系统是一个强大的，但因此其设计实现也需要更多的付出。我们在类型的实现中采取了树式的组织方式，即将一个标识符的类型组织成一棵树，其叶子节点为基本类型（包括函数），而当存在数组或指针时，这里由于语言中允许将数组的部分维切片作为参数进行传递，因此我们向类型系统内部引入了指针类型，将会作为树的非叶子节点指向其引用或集合的基本类型或其他的数组类型。例如对于类型 int[3][3][3]，我们的设计将产生如下的类型树结构。



在这样的数据结构设计上进行相应的类型相等，可转换性等的判断就可以比较轻松的递归进行了。同时如数组模板等信息也可以在树上容易的递归计算得出。

3.1.4 内存指派

对于局部变量，由于通过相对栈指针进行内存的指派，我们只需维护一个偏移量即可。基于上方所述的内存分配方式，我们可以保证任何的变量将不会具有为 0 的偏移量，因此我们可以将全局变量的偏移量设置为 0 从而完成了两种不同内存分配方式的区分。
对于全局变量，我们直接使用其标识符名称作为其标签，由于标识符的唯一性在先前已经得到了保证，因此将不会产生两个相同的标签。

3.1.5 层次信息

层次标号相当简单，只需保存当前栈式符号表中的栈帧数即可。当然，此处得到的值具有准确的相对关系，但为了得到准确的绝对值应该在此的基础上减一。

3.1.6 编译期求值

为了支持数组的情形，由于数组中并非每一个维度上的值都将会是可编译期求值的，我们使用了一个键值对结构的数据结构保存标识符的编译期求值信息。其键为下标而值为编译期所求值。通过正确的填入和取出值，则可以使用编译期常量定义数组的维度。

3.2 错误处理

由于错误处理的任务中定义的错误类型较少且较不全面，加之在后续的大部分实践中将不涉及错误处理的内容，我们对于大部分的错误将使用更加可读的形式通过日志系统进行汇报。日志系统是我们的调试支持部分中的一个重要组件，支持若干个不同的日志级别，允许在编译时设置日志级别，同时支持根据日志级别执行或不执行特定的语句。
在日志中，我们通过错误级别的日志指出检测到的问题，并使用更低的日志级别提供更加详细的信息，例如在抽象语法树构建中将会提供出现问题的子树的结构，以期尽可能的帮助问题的诊断。