编译器申优文章

19375449 罗宇轩

2021 年 12 月 20 日

目录

1 词法分析

1.1 代码架构的初步确定

初次接触编译器编程，还不了解编译器的架构应该如何设计。按照老师和助教的提示，我首先认真阅读和理解了本学期的**文法规则**，再阅读了**课本 17 章和 18 章 PL/0 和 PASCAL 语言的简单的编译器**，并下载了代码进行简单的参考。参考了 **PASCAL 的词法分析部分**，我开始编写我的编译器的第一阶段——词法分析的代码。

1.2 单行注释与多行注释的处理

1.2.1 注释问题的难点

词法分析分的难点之一在于**注释的处理**，而注释又分为单行注释和多行注释，除此之外，**注释符号的'/' 和'*' 分别与除号和乘号相冲突**，因此词法分析程序需要往前多看几步。以下是我对注释问题的解决方法：

1.2.2 注释问题的解决

利用两个标志 **isComment 和 multiComment**，标识是出于单行注释还是多行注释。在解析时根据是否处于注释状态分开讨论：

1. 在非注释状态下，遇到//和/* 进入相应的注释状态
2. 若处于单行注释状态，则忽略本行中后续的所有字符。

3. 若处于多行注释状态，则逐个字符读入，寻找 */以结束多行注释状态。

2 语法分析

2.1 递归下降的非递归实现

2.1.1 为什么会选择非递归实现？

对编译器的语法分析采用递归下降，而对其采用非递归实现主要是担心递归实现会出现递归深度过大的问题（现在看来似乎不需要担心这个问题）。非递归的实现虽然让我的代码量变大，但是却加深了我对递归下降运行过程的理解。

2.1.2 总体的语法分析过程

1. 在 Parser 中设置一个栈，初始将 CompUnit 压入栈中。在 CompUnit 类中进行解析，解析出的子语法成分返回 Parser 并压栈。当**遇到终结符**，则将其弹栈，挂载到栈顶元素下形成树结构；对于**非终结符，当其解析完成时**也需要弹栈，并挂在到栈顶元素下。
2. 当栈为空时，语法分析结束。

2.1.3 每个语法成分的状态记录

在每个语法成分类里面，需要根据文法规则各自识别，但由于是非递归实现，需要一些额外的属性记录识别到的状态。对于每个语法成分类，内部属性和方法大致相同：

1. chooseCandidate()：在对象构造之前，如果该语法成分有多个候选，需要对后面的单词进行“偷看”，确定按照哪种候选来进行解析。
2. wordMatching()：这是语法分析的主要方法。
3. grammerRules：内部属性，是语法成分的列表，表示改语法成分的文法。
4. ruleIndex：内部属性，用于记录该语法成分的语法分析进行到哪个阶段。

2.2 左递归文法的处理

对于左递归文法，将其**改写成右递归文法**，将整个表达式解析完后，再进行特殊处理，**对表达式的子树结构进行重构**，使其变成与左递归解析实现的语法分析树无异。

3 符号表管理/错误处理

3.1 各变量使用域的识别

本模块比较重要的一个设计是 **Field** 类的设计与使用。这里定义的 Field 可以理解为一个 **“使用域”**，相当于程序中的一个 Block，但是有细微的差别，下面对 Field 的设计进行叙述：

1. 与 Block 的区别：体现在函数形参。函数定义中的形参不包含在函数体的 Block 中，但是在 Field 的设计中，要将函数形参加入函数体所在的 Field 中，因为函数形参可以看作是函数体内的一种“定义”。

2. **Field 之间相互链接形成“使用域”链**：这是为了处理重定义或未定义错误而设置的，这一步的目标是为了明确某个**标识符的“可使用域”**。每个 Field 内部有一个属性为 **prevField**, 用于记录它的**前驱 Field**，所谓的“前驱 Field”，指的是本 Field 的**直接外层**，这保证了不会指向本层的其他 Field，沿着这条链一直寻找，直至寻找至最外层。——这就是每个 Field 的 **“使用域”链**。

3. **标识符的“可使用域”链依赖于 Field 的“使用域”链**：简单而言，如果没有切换新的 Field，符号表项中的 prevIdentifier 记录的就是上一个符号表项；如果进入新的 Field 后，再回来本层 Field 时的第一个符号表项的 prevIdentifier 指向**上次进入新 Field 之前的最后一个符号表项**。

4. **沿着标识符的“可使用域”链一直往前回溯，可以查看是否有重名或未定义**

“可使用域”链如下图所示

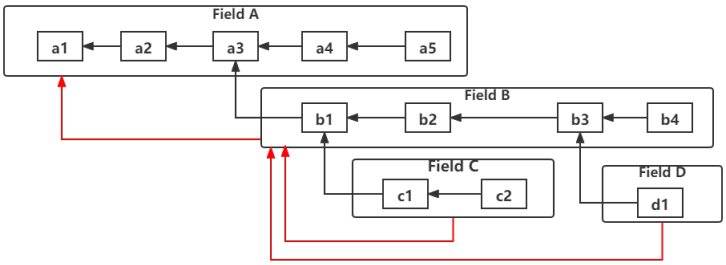

图 1: ” 可使用域”

红线表示 Field 的 “使用域” 链，黑线表示符号表项的 “可使用域” 链以 d1 为例，通过这条链可以搜索到：**a1<-a2<-a3<-b1<-b2<-b3<-d1**，可以准确判断是否存在重定义和未命名的错误。

此外，对于语法错误，仅在语法分析阶段进行相应的处理即可，这部分的设计较为平常。

4 中间代码生成

4.1 布尔变量和普通变量的区分问题

由于第二阶段的编译器引入了布尔变量，**用 0 表示 False，用 1 表示 True**，且也会从 Cond 推导到 AddExp、MulExp、...、PrimaryExp、...。因此递归下降生成中间代码的时候，需要**区分是普通变量还是布尔变量**，从而生成正确的中间代码。

根据**语法制导翻译文法中的继承属性**，为表达式相关的语法元素设置一个标志位 isBool，默认值为 False，表示普通变量。自上而下，如果是从 Cond 推导，则 AddExp、MulExp、...、PrimaryExp 等语法元素相应地将 isBool 置成 Ture(逐层传递)。当然继承属性的获取遵循一定的规则，**不同的文法推导将可能获得不同的继承属性** (isBool)。

当表达式元素的 isBool 被置成 True 时,将生成 NOT 和 NUM_AS_BOOL 中间代码将普通变量转换为布尔变量，保证后续运算的正确性。

4.2 短路计算的处理

为了解决条件短路的问题，在代码中引入 labelManager 类，**专门管理跳转语句与标签的关系**，是中间代码能按照条件正确跳转。

具体做法为：

1. 在解析每个 LAndExp 后，将返回值挂载一个 JUMP 中间代码，将 JUMP 加入 labelManager 的**缓冲队列**中。当所属的 LOrExp 解析完后，在结尾生成一个标签 label，传入 labelManager，**处理缓冲队列**中的 JUMP 语句——**JUMP 语句设置跳转标签为 label，跳转条件 (mode) 为 false**，清空缓冲队列。

2. 在解析每个 LOrExp 后，将返回值（最后一个 LAndExp）挂载一个 JUMP 中间代码，将 JUMP 加入 labelManager 的**缓冲队列**中。当 Cond 解析完成后，在结尾生成一个标签 label，传入 labelManager，**处理缓冲队列**中的 JUMP 语句——**JUMP 语句设置跳转标签为 label，跳转条件 (mode) 为 true**，清空缓冲队列。

4.3 函数调用中间代码歧义问题及解决方案

如果在某一个函数传参的过程中，需要先调用另一个函数并计算出返回值，则会出现参数传递错误的问题。例如：

PUSH a
PUSH b
CALL func1 c
PUSH c
CALL func2 d

源程序中 func2(a, func1(b)) 和 func2(func1(a,b))，均可以生成这段中间代码序列，因此会产生歧义。为解决这个问题，在生成中间代码的时候进行如下处理：

1. 在遇到 CALL 中间代码时，**查符号表，得到函数调用的参数个数**。

2. 从中间代码序列往前回溯，找到对应个数的 PUSH 指令，将它的标志位置位（表示已找到对应的 CALL 指令），**将其按顺序移动到 CALL 指令之前**。若遇到已经置位的 PUSH，则跳过继续往前寻找。

经过处理后的中间代码序列不存在歧义。func2(a, func1(b)) 的中间代码序列变为：

PUSH b
CALL func1 c
PUSH a
PUSH c
CALL func2 d

func2(func1(a,b)) 的中间代码维持不变，因此**解决了函数调用传参出现歧义的问题**。

5 目标代码生成

5.1 mips 代码生成架构的确定——模拟 mars 运行环境

代码生成架构如下图所示

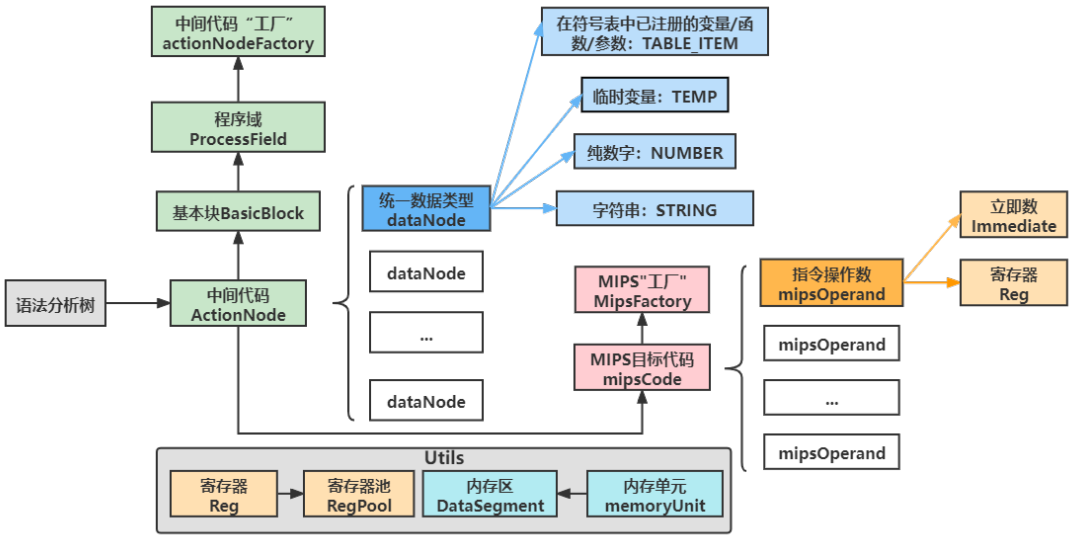

图 2: 代码生成总体架构设计

以下对目标代码生成部分的架构进行一定的解释：

1. **MIPS 目标代码 mipsCode**：mipsCode 类按照 mips 指令的规范来设计，每个 mipsCode 均有一个 **type**，表示的是指令的类型。此外，设计了 **“指令操作数”** 类，寄存器 Reg 和立即数 Immediate（在 MIPS 编码中均视为操作数）均继承于 mipsOperand 父类。

2. **寄存器池 RegPool 和内存区 DataSegment**：在架构图中的 Utils 部分，主要是为了模拟 Mars 中的各个组件：寄存器池和内存区，作用分别是为了便于在生成 MIPS 代码时**管理寄存器和管理堆栈**。

3. MIPS 代码的管理：MIPS 代码由 MIPS 工厂类（mipsFactory）统一管理。

6 代码优化

6.1 寄存器分配

6.1.1 冲突图的构建与 s 寄存器的分配

构建冲突图时，遵循的冲突原则是：如果一个变量在另一个变量的定义处仍活跃，则这两个变量冲突（不能使用同一个 s 寄存器）。**假设基本块 B 中变量 a 的某个定义点 D，另有变量 b，具体判断变量 a 和 b 冲突的逻辑**如下：

1. 若 $b \in in[B]$ 且 $b \in out[B]$。则说明在基本块内变量 b 均活跃，因此 a 和 b 冲突。

2. 若 $b \notin in[B]$ 且 $b \in out[B]$，则如果 b 在第一次定义时活跃。若 **b 的第一次定义早于 a 的定义点 D**，则符合冲突条件，可判断 a、b 变量冲突，否则暂不能判断冲突。

3. 若 $b \in in[B]$ 且 $b \notin out[B]$，则 b 在最后一次使用时停止活跃。若 b 在基本块内没有使用点（理论上不可能），则 a 与 b 暂不冲突；若有使用点且 **b 的最后一次使用晚于 a 的定义点 D**，则可以判断 a 与 b 冲突，否则暂不能判断 a 和 b 冲突。

4. 若 $b \notin in[B]$ 且 $b \notin out[B]$，则 b 在第一次定义时开始活跃，在最后一次使用后停止活跃。只需判断 **a 的定义点 D 是否在 b 的这段生命周期内**即可。若在，则可以判断 a 和 b 冲突。

冲突图构建之前将**仅使用一次的临时变量**剔除，不参与冲突图的构建；与此同时，还将**全局变量与排名前 4 的形参**排除在外，对于前四个形参则直接分配 $a0-$a3 寄存器。

在剩下的所有变量中，需要对每个变量的**每个定义点与所有其他变量**作以上判断，才能准确判断出变量之间是否存在冲突关系，以便后续寄存器的分配。

冲突图构建成功后，按照**韦尔奇·鲍威尔着色算法**对寄存器进行分配，具体做法是：

1. 将冲突图中所有变量按照度的大小**降序排列**。
2. 从队列中的第一个变量开始（度最大），为其分配一个寄存器 $s0；而后沿着队列线性扫描，当遇到一个变量，**它不与已分配的寄存器冲突**时，为其分配相同的寄存器 $s0；如此进行下去，直至扫描至队列末尾。
3. 将已分配寄存器的变量从队列中移除，再次按照度的大小降序排列，按照 2 的做法进行扫描，分配第二个寄存器 $s1...
4. 当所有的寄存器都被分配完或所有变量都被分配到寄存器后，分配结束。**若仍有变量未分配到寄存器，则它们将使用临时寄存器**。

6.1.2 临时寄存器池的使用机制

对于临时寄存器，采取临时寄存器池的方式，管理临时变量对临时寄存器的占用，具体的使用机制如下：

1. 对于临时寄存器而言，仅有**空闲和被占用**两种状态，由临时寄存器池来对寄存器的状态和占用情况进行统一管理。
2. 对于变量来说，它的寄存器使用生命周期为：
   (a) 从解析中间指令到 mips 代码生成这段时间，变量对寄存器的占用处于“**强制占用状态**”，即不可被其他变量抢占写回。
   (b) 当变量需要 use 或者 define 时，均需要从寄存器池中申请一个临时寄存器。**若本身已占用着一个临时寄存器**，则不必寻找新的临时寄存器。**否则**，寄存器池将为变量寻找一个未被占用的寄存器返回；**若寄存器池中的所有寄存器均被占用**，则选择一个不处于

"强制占用" 状态的寄存器，将其对应的变量写回到内存中，为申请寄存器的变量分配该寄存器。

(c) 当 mips 代码生成后，对于仅使用一次的临时变量，**解除强制占用状态，主动释放临时寄存器，不写回**；如果是全局变量，则应该**解除强制占用状态，释放临时寄存器，写回**；如果是其他变量，则**解除强制占用状态，暂不主动释放临时寄存器，不写回**，若有其他变量需要使用该寄存器时再被动写回。

6.1.3 参数传递时的寄存器冲突问题及解决方案

在参数传递时,若本函数的参数为 a, b,而此时需要调用函数 $func1(b, a)$,若直接用参数寄存器进行传参操作:

```
move $a0, $a1
move $a1, $a0
```

则会出现参数传递的冲突：b 传递到了第一个参数的位置，而此时 $a0 寄存器已经不是 a 的值了，因此无法把 a 传递到第二个参数的位置。为此，需要解决参数传递的冲突，不能直接通过参数寄存器进行参数传递。

解决方法：在参数传递时，将 $a0-$a3 中需要保存的寄存器保存到堆栈上，解除形参对寄存器的占用状态（如果传参过程中需要使用，**必须从堆栈中取出**而不是从 $a0-$a3 中直接获取）。返回后恢复现场时，要将前四个参数恢复到寄存器上，开启参数对相应寄存器的占用。

6.2 除法指令的优化

以下讨论的是**除数为常量的除法指令优化**。

6.2.1 最初的优化构想

在编码前，我在网上搜索关于除法优化的博客。基本的套路是计算魔数 (magic number)，相乘后再右移相应的位数，如果结果是负数需要加一。这样做其中有一个问题是，魔数的计算涉及到**误差**的问题，需要根据误差的大小来判断魔数计算是需要向上取整还是向下取整，且还存在除数是正或负的区别，情况非常复杂。经过在 mips 上试验，无法得到一个完全正确的方式，使得对于任何合法的除数，总能得到正确的运算结果（**每种做法均能找**

到运算错误的反例）。主要问题在于没有论文的支撑，按照我目前的数学水平还不足以对某种优化算法的正确性加以证明。

6.2.2 通过查阅相关文献和书籍解决问题

因此我在网上搜集更靠谱的资料。当我搜集到一篇关于 c++ 如何进行除法优化的介绍文章时，它引用了 Henry S. Warren 的著作 **Hacker's Delight**，说的是里面专门介绍了除法优化的算法及有正确性证明。因此我在网上搜索这本著作的电子版，**对其中除法优化算法的介绍和证明进行研读和理解，并将其运用到我的编译器上**。

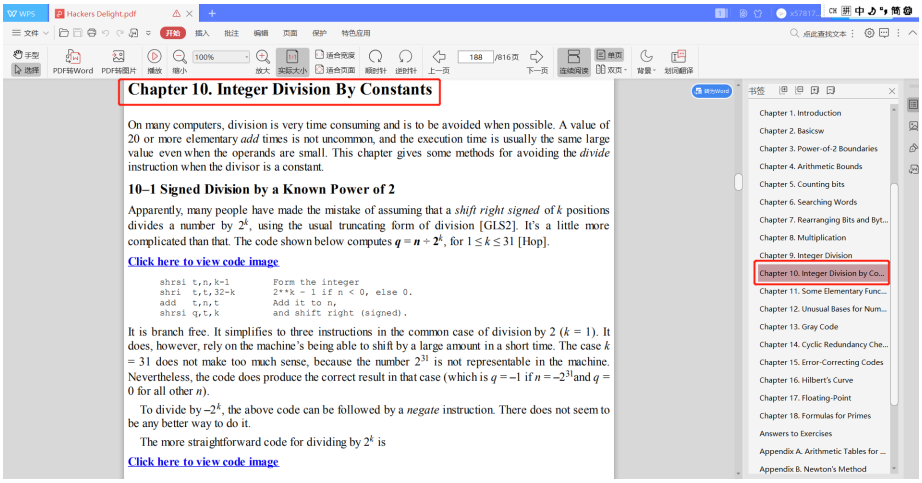

图 3: Hacker's Delight 中关于除法优化的章节

其主要思想还是计算相应的魔数和右移的位数。对于不同的除数，按照常规的算法有可能使魔数超出了 $-2^{31} - 2^{31} - 1$ 的的范围，因此需要进行特别的处理。假设计算出的魔数为 m，右移位数为 s，则以下对除数 d 和 m 的不同情况进行讨论：

1. 若魔数 m 与除数 d 符号相同，说明**魔数计算过程中没有溢出**。计算结果为：

$$\frac{a}{d} = (a \times m) >> (32 + s)$$

如果结果是负数，还需要在此基础上加一。除数为 10 时即为这种情况，按照算法计算出 $m = 1717986919$，$s = 2$，mips 指令序列如下所示（$s1 为被除数，$t1 是除法结果）：

li $t2, 1717986919

mult $s1, $t2

mfhi $t1

sra $t1, $t1, 2

slt $t2, $s1, $zero

addu $t1, $t1, $t2

值得一提的是，序列中最后两条指令 **slt 和 addu** 是为了判断结果是否是负数，**如果结果是负数需要给结果加一**。

2. 若魔数 m 与 d 符号相反，说明**魔数计算过程中存在溢出**，计算过程如下。

   (a) 若 $d > 0$，则计算结果为

   $$\frac{a}{d} = (((a \times m) >> 32) + a) >> s$$

   。

   (b) 若 $d < 0$，则计算结果为

   $$\frac{a}{d} = (((a \times m) >> 32) - a) >> s$$

   。

   如果结果是负数，还需要在此基础上加一。除数为 7 时即为这种情况，按照算法计算出 $m = -1840700269$，$s = 2$，mips 指令序列如下所示（$s1 为被除数，$t1 是除法结果）：

li $t2, -1840700269

mult $s1, $t2

mfhi $t1

addu $t1, $t1, $s1

sra $t1, $t1, 2

slt $t2, $s1, $zero

addu $t1, $t1, $t2

以上是求除法的商的优化，对取模运算进行优化，只需要在求出商的基础上，根据求余的公式，利用**减法和乘法指令**计算出余数即可，此处不再赘述。

7 总结

一学期的编译器编码和设计中，从一开始的迷茫，到逐步熟练；从一开始遇到困难时的懊恼，到自己学会一步一步解决问题：通过查阅文献、修改架构等等解决编译器设计上遇到的问题——可以说我在一学期的编译课设中收获颇丰。经历了苦思冥想的设计和若干次重构后，我逐渐找到了正确的架构，找到了适合的设计方式，完成了编译器的设计。