GHC 源码剖析：语法分析

陈逸凡，吴昊泽

2017 年 6 月 1 日

1 概要

GHC 在前端使用 Alex[4] 和 Happy[5] 这两个在 Haskell 开发中常用的工具来生成词法分析器和语法分析器。在这一模块的源文件 [6] 中，除了词法分析器（Lexer.x）和语法分析器（Parser.y）的生成文件之外，还包括了辅助的语法定义文件（RdrHsSyn.hs），以及处理 C-FFI（C Foreign function interface）和 Haddock 文档生成所需的辅助函数等。接下来主要介绍 alex 与 happy 这两个工具和 GHC 在语法解析过程中一些比较重要的额外处理。

2 词法分析器生成工具 Alex

2.1 功能和结构

Alex[4] 是一个主要用 Haskell 开发的词法解析器生成器。功能与 C 语言开发常用的 Lex 或 Flex 类似，将用户定义的描述文件转换为包含生成的扫描器函数的 Haskell 源码。一个描述文件主要包括前缀代码，wrapper 声明，宏定义，规则和后缀代码这些部分。前缀代码和后缀代码为 Haskell 代码，主要功能有声明导入模块，定义导出模块，定义全局类型等。需要注意的是在 Alex 中，用户需要自己声明和定义 token 的类型和值，一般放置在后缀代码这一板块中。

2.2 模式

Alex 中的正则表达式语法与 lex 风格的正则表达式语法基本一致，除了可以用”()” 来匹配空串以外。

2.3 动作

由于 Haskell 是静态类型的纯函数式语言，所以相比 Lex 中的规则，Alex 描述文件的规则中模式所对应的代码有更严格的要求，每条规则对应的动作代码，应当是符合约定类型的函数。

对于要求动作函数满足约定类型这一较强的限制，Alex 提供了 wrapper 声明的语法，wrapper 声明的作用在于，约定了每个动作的类型，从而使得用户可以使用预定义的高阶 API，免去自定义的麻烦。不同的 wrapper 对应不同样式的函数类型，如"basic" wrapper 约定每条规则的输入类型为 String 输出类型为自定义的 token 类型，"posn" wrapper 则在输入中提供了匹配串的位置信息，使用这两种 wrapper 时都可以；"monad" wrapper 和"monadUserState" wrapper 则提供了传递全局状态的 monad 结构，其中后者允许用户自定义全局信息，举个例子，如果需要统计被匹配到的所有 identifier 数目，就需要定义 AlexUserState 包含一个累计 identifier 数目的条目。

而不声明 wrapper 时，用户只能使用最底层的 API。有趣的一点是，使用底层 API 时，用户需要声明更底层的函数输入类型和获取单个字符的函数。Alex 开放输入类型约定的主要原因在于，Haskell 标准库的 String 实现是非常低效的，在强调性能的场合，往往使用 Text 或 ByteString 这样的高性能字符串实现。在定义好最基本的函数之后，每个模式对应的 action 就应该有统一的类型 $AlexInput \to Int \to AlexReturnaction$，或 $user \to AlexInput \to Int \to Maybe\ (AlexInput, Int, action)$，后者相比前者，增加了对于向前看/向后看谓词的支持。

在 GHC Lexer 中，使用的就是最后一种形式的 action 函数，虽然复杂但也是最灵活的。其中有趣的一点是 token 的定位信息与 token 是部分分离的，比如注释段等。这样做，可以使语法分析树只关注有效代码，而错误定位时，可以将注释等通过 ApiAnnotations 这一 map 来定位。

2.4 工作方式

Alex 的原理与 Lex 相同，仍然是依据输入文件的模式构造确定性有限自动机。

3 语法分析器生成工具 Happy

3.1 功能

Happy[5] 是一个（也是）用 Haskell 实现的语法分析器生成器，功能与 Yacc/Bison 类似，将用 BNF 表示的语法转换为一个包含（一个或多个）语法分析器的 Haskell 模块源码。

3.2 语法和文件结构

每个 Happy 输入文件也是由前缀代码、声明、模式和后缀代码组成。前缀代码主要定义导出模块，声明导入模块，放置宏定义等，其他用户代码一般放在后缀代码中。

声明主要包括声明被生成的语法分析器的名字、作为输入的 token 的类型、解析错误时的处理函数等。此外还有若干可选的声明条目，选取一些解释如下：

- %name 条目指导 happy 对哪些符号生成指定名字的解析函数。
- Precedence 优先级声明，对一些符号，可以声明其左结合或右结合性质，而且声明顺序代表其优先级先后。这样可以指导其移入规约冲突的解决方法，有助于保持语法的简洁。这与 Yacc 中的处理是一致的。
- %lexer 条目可以声明一个与 parser 串联的 lexer，类型为 $(Token \to P\ a) \to P\ a$，是一个 Continuation 形式的函数。使用这个声明之后，lexer 每生成一个 token 将会被直接被 parser 作为参数接收进行移入/规约，这样可以避免一次产生大量 token 占用空间。
- %monad 条目声明了一个封装 parse 结果的 monad 结构，可以用于传递全局状态。
- %token 条目声明终结符号及对应的值。

3.3 规则

Happy 的输入规则也是将产生式的每个产生式体与对应的动作函数一一匹配，每个输入符号对应的值依次用 $1、$2 等表示。每个表达式都应当具

有一个类型，可以是值类型也可以是函数类型，每个产生式体对应的语义动作表达式应当具有与之相同的类型。

3.4 生成方式与使用技巧

Happy 与 Yacc 类似，都是按 LALR(1) 方法生成语法分析器，这一方法是 SLR(1) 方法和 full LR(1) 方法的折中，产生状态与 SLR(1) 一样少但解析能力弱于 full LR(1)。

由于这一生成方式，所以在手写 Happy 的输入文件时，尽可能用左递归语法能极大减小分析栈的增长规模，如在解析一个数组时，右递归语法导致线性的栈空间增长，而左递归是常数规模的栈空间占用；这样也能减轻垃圾回收的负担。而更为简洁的语法也能使生成的语法分析器更高效。

3.5 对 Alex 和 Happy 的一些使用体验

Alex 和 Happy 相比 Lex & Yacc，最大的不同应当体现由 Haskell 语言本身带来的在类型上的限制非常强。这一方面的确限制了程序的书写，但另一方面类型也有助于使用者考虑清楚到底需要哪些全局状态、是否需要某些上下文相关性等。而尤其在语法分析器构造中，标明每个产生式的类型都有助于写出正确的语法。

4 GHC Parser 的一些实现细节

本节主要讲与解析出的 Haskell 抽象表示有关的一些事实。

4.1 抽象语法树

经过 GHC 的多种扩展，GHC Haskell 的语法已经变得相当复杂，所以在这里不展开解释，完整的 Hasekll 语法定义详见 ghc/compiler/hsSyn 文件夹下内容，而语法分析得到的语法树由 ghc/compiler/parser/RdrHsSyn.hs 文件所定义，Rdr 是 Reader 缩写。与 HsSyn 中的定义所不同之处在于，RdrName 大部分名字没有与定义位置所绑定，需要在接下来 renamer 这一步才能将其与其实际所指向的代码产生关联。

下面就是 RdrHsSyn 所产生的语法分析树中所使用的对标识符的定义，OccName 仅根据上下文对该名字是 type 还是 term 做了区分。

```
data RdrName
  = Unqual OccName
    -- Used for ordinary, unqualified occurrences

  | Qual ModuleName OccName
    -- A qualified name written by the user in
    --  *source* code.  The module isn't necessarily
    -- the module where the thing is defined;
    -- just the one from which it is imported

  | Orig Module OccName
    -- An original name; the module is the *defining* module.
    -- This is used when GHC generates code that will be fed
    -- into the renamer (e.g. from deriving clauses), but where
    -- we want to say "Use Prelude.map dammit".

  | Exact Name
    -- We know exactly the Name. This is used
    --  (a) when the parser parses built-in syntax like "[]"
    --  and "(,)", but wants a RdrName from it
    --  (b) by Template Haskell, when TH has generated a unique name
```

另外关于抽象语法树还有一点细节在于上文提到的部分定位信息与语法树分离。对于比如 let <let-bindinds> in <expression> 这样的结构，token “in” 只在做语法分析时产生作用，在构建完成的抽象语法树中应当不占据位置；再比如注释，在语法树中也不起作用。但在错误定位以及编辑器语义高亮等场景下，需要将解析出的语法树还原成与原来一摸一样的代码，就需要这些代码的位置信息。如果直接将这些结构插入到语法树中，会导致极其复杂的语法树构造，而且对之后做语义分析带来许多麻烦。GHC Parser 的处理办法是，将这些不起语义作用的 token 记录在最接近的语义单元的定位信息中，再将这些 token 的定位信息存放在一个全局的从 token 种类到

定位信息（和内容）的 map 中。这样既不会增加语法分析的复杂度，也能保持一个简洁的抽象语法树。

4.2 左递归语法

由于一方面 LALR(1) 方式生成的分析器在处理左递归时更节省空间，另一方面大部分中缀符号在这一阶段无法确认结合方向（Haskell 允许自定义中缀运算符，并可定义结合方向和优先级），所以在做语法分析时，认为所有符号是左结合的，直到 renamer 时确认符号结合方向和优先级时才重构语法树。

4.3 缩进

Haskell 是缩进敏感的语言，通过缩进来表达代码块层次结构。在 GHC Parser 中，缩进在 Lexer 中作为一个状态来处理，并且会产生一个名为 vocurly (virtual open curly) 的 token，而当缩进结束时，产生一个 vccurly (virtural close curly) token，这样就能完成包裹子代码块的功能。

5 在主程序中的代码

Parser 模块仅仅是提供了将输入字符串处理成解析树的功能，在完整的 GHC 编译流水线中，还需要在这之前预处理输入文件（如加入定位信息）、根据编译参数设置环境变量，并在输入后补充信息等。这样才能将 Parse 的结果作为参数提供给接下来的 rename 和 typecheck 函数。具体代码在 ghc/compiler/main/HscMain.hs 中。其中包含将 Parser 模块所产生的函数再次封装成输入输出符合流水线约定的类型的过程（主要看 hscParse 函数）。

6 一点偏题：Monadic Combinatorial Parser

解析器组合子 [2] 是一种常用于快速搭建解析器的技术。由于其将语法结构与解析过程完全结合在一起，所以（在有设计精妙的组合子库 [3] 前提下）用代码“组装”解析器非常便捷直接。

从设计方法上说，组合子充分利用语言的**高阶函数**特性，通过分解语法的过程的逆过程来将简单的语法分析函数修饰、串联或并联成复杂的语法分析器。

从原理上说，这是下推自动机的一种实现（确定性和非确定性都可实现，取决于是否支持将两个组合子合并的操作）。

从性能上来看，解析器组合子可以处理任意向前看文法，但性能在向前看字符较多或处理左递归时都会变差 [1]。

具体的讲解和一个简单实现，详见另一篇 Intro to Monadic Combinatorial Parser 及代码包 monparse 。

参考文献

[1] Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. Parser combinators for ambiguous left-recursive grammars. In In ????, pages, 2007.

[2] Graham Hutton and Erik Meijer. Monadic parser combinators, 1996.

[3] Daan Leijen. Parsec, a fast combinator parser, 2001.

[4] Simon Marlow. Alex: A lexical analyser generator for haskell.

[5] Simon Marlow. Happy: The parser generator for haskell.

[6] The GHC team. ghc/compiler/parser/.