Happy-GLL：面向参数化非终结符的模块化、可复用且完备的自顶向下解析器

L. Thomas van Binsbergen
ltvanbinsbergen@acm.org 信息学研究所，
阿姆斯特丹大学 阿姆斯特丹，荷兰

Damian Frölich dfrolich
@acm.org 信息学研究所，阿姆斯特丹大学
阿姆斯特丹，荷兰

arXiv:2303.08044v1 [cs.CL] 14 Mar 2023

摘要

解析器生成器和解析器组合子库是生成解析器最常用的工具。解析器组合子利用宿主语言提供可复用组件，其形式为以解析器作为参数的高阶函数。很少有解析器生成器能通过抽象支持这种复用，而生成的解析器能够像其所针对的文法部分一样模块化且可复用的解析器生成器则更少。本文基于 GLL 算法的 FUN-GLL 变体，提出一种策略，可根据含有参数化非终结符的语法描述生成模块化、可复用且完备的自顶向下解析器。

本文将该策略作为 Happy 解析器生成器的一种新型后端进行讨论和演示。Happy 文法可以包含“参数化非终结符”，其中参数对文法符号进行抽象，从而提供一种抽象机制来定义可复用的文法运算符。然而，现有的 Happy 后端未能充分发挥参数化非终结符的潜力，因为参数化非终结符无法跨文法复用。此外，解析器生成过程可能无法终止，也可能导致生成的解析器规模呈指数级增长，且生成时间也呈指数级增加。

本文介绍的 GLL 后端通过生成类似解析器组合子的高阶函数，成功实现了参数化非终结符，同时继承了自顶向下解析的所有优点。该后端能够为完整的上下文无关文法类生成解析器，生成解析器的时间复杂度为线性，并且生成的解析器能够找出输入字符串的所有推导。据我们所知，本文介绍的 GLL 后端使 Happy 成为首个集齐上述所有特性的解析器生成器。

本文介绍了 GLL 后端的转换过程，并通过若干实验将其与 Happy 的 LALR 和 GLR 后端进行了比较。

1 引言

递归下降语法分析 是一种根据上下文无关文法的描述（手动或借助工具）编写自顶向下 解析器的技术。递归下降解析器有一个共同点：文法中的每个（非终结符和终结符）符号都由一段代码实现，且这些代码段可以按顺序排列，从而实现文法中的一个备选项；非终结符的备选项之间则通过分支控制流来选择。递归下降解析器生成器将文法直接转换为解析器。每个符号都单独转换（独立编译），符号对应的代码也可以单独测试（语义模块化）。因此，生成的解析器易于维护和调试；更新文法规范只需进行最少的重新编译，而且可以通过隔离解析器和输入中导致意外行为的部分，找出意外行为的原因。

递归下降解析器的功能特性因其所基于的自顶向下解析算法而异，但历史上递归下降解析器一直难以处理左递归非终结符和未进行因子提取的备选项。典型实现只接受 LL(k) 文法：对于这类上下文无关文法，在给定 $k \geqslant 0$ 个终结符向前看的情况下，不会有两个备选项同时适用。使用递归下降解析器生成器通常需要应用文法变换来消除左递归，和/或进行左因子提取以生成 LL(k) 文法。然而，应用文法变换并非总是可取，甚至并非总是可行；有些上下文无关文法不存在与之等价的 LL(k) 文法。

自底向上解析通常无需应用文法转换即可实现。自底向上解析器也非常快速，因为它们能够利用预先计算并记录在表中的信息，即所谓的“分析表”。这些特性使自底向上解析比自顶向下解析更受欢迎，成为解析器生成器的基础；LALR、SLR 和 GLR [36]等多种算法，以及 Yacc、Bison、Happy [21]、Menhir [24]、Rascal [18]、Spoofax [17] 和 SDF [40]等实现，都印证了这一点。然而，自底向上解析器生成器的用户无法受益于独立编译和语义模块化。

本文介绍了 Happy 的一种新后端，它基于广义 LL（GLL）算法生成递归下降解析器。该后端具备前述递归下降语法分析的优点，但无需进行任何文法变换。**广义解析**算法在通用意义上接受任意上下文无关文法，并且在完备意义上

能够生成给定输入字符串的所有可能推导。广义 LL（GLL）算法 [28, 29] 依赖于在可能很大的数据结构中进行复杂的记录，以同时确保解析器终止并找到所有推导。尽管增加了复杂性，GLL 解析器仍易于维护和调试，并支持独立编译和语义模块化（与其他递归下降解析器一样）。

广义解析器组合子是近年出现的技术，结合了广义解析与解析器组合子的优点 [15, 25, 38, 39]。上述方法的共同点是，它们都显式表示文法组件，例如符号和产生式规则——Earley 的广义解析算法 [10]、GLR [36] 和 GLL [28] 都需要这种表示——而传统解析器组合子并不显式表示文法组件。 [25] 和 [38] 提出的库会先生成一个实际的文法对象，再将其作为输入提供给独立的广义解析算法（从而避免独立编译）。 [20], [5], 和 [9] 也在广义解析以外的领域应用了这种所谓的文法组合子理念。

文法组合子模糊了组合子库与解析器生成器之间的界限，两者只剩下少数几个本质区别：文法组合子库在运行时生成解析器，而不是在单独的阶段生成；文法组合子库定义的是嵌入式领域特定语言（EDSL），而解析器生成器定义的是外部领域特定语言（DSL）。EDSL 通常易于扩展，且可以利用宿主语言的强大功能来定义新的运算符。这一点对解析器组合子成立，对文法组合子也在较小程度上成立 [37]

GLL 算法的**FUN-GLL** 变体会计算广义解析所需的最少文法信息 [39]。该算法可用于解析器组合子，因为文法信息是在语法分析过程中按需计算的，而不是在单独的阶段中计算的。这样，FUN-GLL 通过抽象实现了独立编译下的复用，我们基于 FUN-GLL 的 Happy 后端也是如此。

解析器生成器通常被视为用于实现 DSL，它们通常提供数量固定的语言构造，往往对应于扩展巴科斯-诺尔范式（EBNF）的某种变体。Happy 解析器生成器中的“参数化非终结符[1]”使用户能够定义作用于文法符号的自有运算符，类似于宏文法 [11, 34] 以及 Menhir [24] 和 PRECC [7] 解析器生成器中的参数化非终结符。然而，Happy 现有的（LALR 和 GLR）后端无法充分发挥参数化非终结符的潜力，因为用户定义的运算符无法在不同文法之间复用

[1]在 Happy 的用户手册中称为“参数化产生式”。

此外，这些后端依赖一种算法，该算法实际上会对所有参数化非终结符执行宏展开。这种算法可能无法终止，也可能导致生成的解析器规模呈指数级增长，并且需要指数级的生成时间。本文介绍的 Happy GLL 后端通过为文法中的非终结符生成可复用的高阶函数，克服了这些问题；这种做法类似于 FUN-GLL 解析器组合子。

The contributions of this paper are as follows. This本文：

- 提出一种策略，可从包含参数化非终结符的语法描述中生成模块化、可复用且完备的自顶向下解析器
- 为 Happy 添加一个后端，充分发挥 Happy 参数化非终结符的潜力，使 Happy 成为少数支持“通过抽象实现复用”的解析器生成器之一；它或许也是首个能够生成完备解析器、找出输入字符串所有推导的解析器生成器
- 据我们所知，GLL 后端是首个在 Haskell 解析器生成器中实现 GLL 的后端，也是 Happy 首个具备递归下降解析全部优势的后端
- 通过一系列实验，将 GLL 后端与现有的 GLR 和 LALR 后端进行比较，展示了新后端的特性

第 2 节通过说明递归下降解析和参数化非终结符的优势，阐述了采用 GLL 后端的动机。第 3 节说明 GLL 后端如何将参数化非终结符转换为可复用的 GLL 解析器，以计算输入字符串的所有解释。第 4 节讨论 GLL 后端实现中的实际问题，包括 Happy 文法形式体系的某些特定方面，例如单子语义动作。第 5 节展示了参数化非终结符的替代处理方式所带来的一些优势，并通过实证评估比较了不同 Happy 后端的运行时间。第 6 节和第 7 节讨论相关工作并作出总结。

2 递归下降与复用

本节通过介绍从 Happy 文法到“识别器[2] 函数”的转换，展示递归下降解析的优势，并通过复用示例阐述参数化非终结符的动机。假定读者熟悉语法分析以及 Happy（或类似 Yacc）的语法。Happy 的语法在在线文档中有说明， [21]和 [14] 则是学习语法分析的绝佳入门材料。

图 1 中的 Happy 文法定义了一个类元组结构的语法，其元素为字母字符。本节忽略语义动作；在 Happy 文法中，语义动作通常与产生式的备选项相关联。

[2]识别器仅指出字符串是否属于某种语言，而解析器还会以解析树的形式提供证明。

```
%name tuples AlphaTuples %tokentype {
Char } %token ',' {','} '(' {'('} ')' {')'} %token
alpha {c | c ∈ (['a' .. 'z'] ++ ['A' .. 'Z'])} %%
AlphaTuples : '(' MAlphas ')' MAlphas :|
Alphas Alphas : alpha| alpha ',' Alphas
```

图 1. 由圆括号括起的逗号分隔字母字符的 Happy 文法。

该转换会为文法中的符号生成（可能是高阶的）识别器函数（例如示例中的 *alpha* 和*AlphaTuples*）。识别器函数可以调用为其他符号生成的识别器函数，而不论这些符号是否定义在同一个文件中。因此，符号定义可以分散在多个源文件中，并且只有在其定义发生变化时才需要重新编译；至关重要的是，所用符号发生变化时则无需重新编译。本节将介绍，这种相较于 Happy 当前（LALR 和 GLR）后端的改进，是通过生成自顶向下解析器并使用高阶函数实现参数化非终结符而达成的。

2.1 识别器函数

识别器函数是一种函数，它接受一个词法单元序列*inp*（称为句子）以及该序列中的一个索引k，并判断它是否能识别*inp* 中从索引k开始的子序列的某个前缀。词法单元的类型保持抽象，在 Haskell 代码中用类型变量t表示。当文法规范中存在 **%tokentype**指令时，识别器会实例化这一类型。

```
类型 识别器 t = [t] → Int → (Int → Bool) → Bool
```

第三个参数是一个续延函数，当识别出一个前缀时会调用它，以尝试识别输入句子的其余部分。在第 3 节中，续延传递风格的应用对于扩展到 GLL 至关重要。关于用于语法分析的续延传递的说明和应用，参见 [16, 20, 31, 32]。

%词元指令将一个标识符（例如 *alpha*）与花括号内的 Haskell 模式（例如 ','）关联起来。下面的识别器实现了词元 *alpha*：

```
-- t 在以下 %tokentype 声明p_alpha之后实例化为
Char:: 识别器 Charp_alpha = matchPattern_1 匹配器
  where matcher t = case t of
          c | c ∈ (['a' .. 'z'] ++ ['A' .. 'Z']) → Just t
          _                                      → Nothing
```

%词元指令决定识别器的名称以及 **case**-表达式的第一个分支。词元识别器函数的逻辑由函数 *matchPattern_1* 实现，该函数位于单独的支持库中，如下所示：

```
matchPattern_1:: (t → Maybe t) → 识别器
t matchPattern_1 匹配器 inp k c
  | k ⩾ 0, k < length inp, Just _ ← matcher (inp !! k) = c (k + 1)
  | otherwise                                          = False
```

当词元成功匹配时，会调用续延 c（传入 k+1，因为已识别出一个词元）；否则返回*False* 。

可以将一个识别器函数传给支持函数run_1，以尝试识别一个句子。该函数会将识别器应用于句子，并传入初始索引 0 和一个续延；该续延用于检查是否已识别出整个句子。

```
run1 :: 识别器 t → [t] → 布尔值
run1 rec 输入 = rec 输入 0 (λk → k ≡ 输入长度)
```

函数 run_1 用于实现 Happy 的 **%name** 指令（该指令指定 LR 后端要使用的起始符号）。示例中的 **%name** 指令会生成以下代码。该指令决定函数名称以及所使用的识别器。

```
run_元组 :: [t] → Bool run_
元组 =run1 p_AlphaTuples
```

使用 run_1 展示了我们方法的一个优点：每个识别器都可以用于识别句子，无需重新编译。

要为非终结符的备选项生成代码，需要按顺序放置识别器，使其中一个在另一个“之前”执行。例如，*Alphas* 的最后一个备选项要求先识别一个 *alpha* ，然后识别一个逗号（‘,’），再识别更多的 *Alphas*。对于每个备选项，都会生成代码，通过定义新的续延来“串联”多个识别器函数。例如，如果符号 p 在某个非终结符的备选项中位于q 之前，那么就会应用 p的识别器，并附带一个会应用q识别器的续延。非终结符 *Alphas*会生成以下代码：

```
p_Alphas ::识别器 Charp_Alphas inp k  c0 = p_
alpha inp k  c0 ∨ p_alpha inp k  c1其中 c1 k  = p_逗
号 inp k  c2c2 k  = p_Alphas inp k  c0
```

定义*Alphas* 的非终结符的两个备选项的代码都会应用 *p_alpha*。区别在于传递给 *p_alpha*的续延。第一个备选项在 *p_alpha*之后结束，因此，给定的续延（c_0）就是 *p_Alphas*接收到的续延。在第二个备选项中，*alpha* 后面跟着

逗号 和*Alphas*，因此[3] $c_1\ k\ =\ p_$逗号 *inp k* $\ c_2$ 和$c_2\ k\ =\ p_Alphas\ inp\ k\ \ c_0$。请注意，这些定义中有针对变量 k 的遮蔽声明，以保持命名约定的一致性。

*MAlphas*的第一个备选项为空。因此，此备选项的代码会直接应用续延 c_0：

p_MAlphas ::识别器 *Char*$p_MAlphas$ 输入
$k\ \ c_0 = c_0\ k\ \vee\ p_Alphas$ 输入 $k\ \ c_0$

针对 *AlphaTuples* 的识别器[4] 完成了图 1中文法的翻译：

p_AlphaTuples ::识别器 *Char*$p_$
AlphaTuples 输入 $k\ \ c_0 = p_lparen$ 输入
$k\ c_1$**其中** $c_1\ k\ =\ p_MAlphas$ 输入 $k\ \ c_2 c_2$
$k\ =\ p_rparen$ 输入 $k\ c_0$

2.2 讨论

复用。 Happy 的参数化非终结符对符号进行抽象。例如，在 *Alphas* 的定义中，**逗号** 可以替换为一个参数，从而将逗号抽象为分隔符。抽象和应用采用熟悉的“函数式风格”书写：形式参数和实际参数放在圆括号之间，并以逗号分隔。对于建议的、更加通用的 *Alphas* 版本，其参数化非终结符写法如下：

SepAlphas (*sep*) : *alpha*
| *alpha* 分隔 *SepAlphas* (分隔)

利用 Haskell 的高阶函数，Happy 中的抽象和应用几乎可以直接转换为 Haskell：

p_SepAlphas ::*Recognizer Char* → *Recognizer Char*$p_$
SepAlphas $p_sep\ inp\ k\ \ c_0 = p_alpha\ inp\ k\ \ c_0 \vee p_alpha\ inp$
$k\ \ c_1$**其中** $c_1\ k\ =\ p_sep\ inp\ k\ \ c_2$
$c_2\ k\ =\ p_Alphas\ \ p_$分隔符 $inp\ k\ \ c_0$

图 2 展示了几个可高度复用的参数化非终结符，用于分隔符、带分隔符的重复和可选性等模式，并给出了它们的用法示例。 [35]中还可以找到更多实际应用中的复用示例。

递归下降的优势。 LALR 和 GLR 要求通过 **%name** 指令指定“入口非终结符”。这是因为 LR 算法要求文法具有起始符号。本节所述的转换不需要入口点，因为 run_1 可应用于任意识别器。实际上，这意味着可以单独调试文法中的每个非终结符，从而使整个文法更易于测试和维护。此外，为非终结符生成的代码彼此独立

元组 *(elem)*：圆括号 *(Optional (Multiple (elem,* 逗号*)))*
列表 *(elem)*：方括号 *(Optional (Multiple (elem,* 逗号 *)))*
圆括号 *(x)*：*Within ('(', ')', x)* 方括号 *(x)* | *Within ('[', ']', x) Within (l, r, x)*：*l x r Optional (x)*：| *x*
Multiple (elem, sep)：*elem*| *elem sep Multiple (elem, sep)*

图 2. 可复用参数化非终结符示例。

文法中其他符号的定义。因此，这种转换保留了非终结符固有的模块化特性。

本节展示了识别器函数可以自由组合，从而通过 *SepAlphas* 等高阶函数构成更复杂的识别器。由不同文法生成的识别器也可以用这种方式组合，从而实现跨文件复用，但需注意以下几点。类型签名会将词元的类型实例化为 **%tokentype** 指令中指定的类型。这意味着，具有不同 **%tokentype** 指令的两个文法会生成无法组合的识别器。Happy 用户可以根据应用场景，通过至少两种方式应对这一潜在问题。

一种方法是省略 **%tokentype** 指令，并依靠 Haskell 的类型推断自动赋予可能的最一般类型。如果从图 1 的文法中删除 **%tokentype** 指令，那么由于 **%token** 指令中使用了字面量，推断出的类型就是 *Char*。尽管 *MAlphas* 的定义没有引用记号符号，但词元类型 *Char* 是根据*Alphas* 的识别器类型推断出来的。如果 *Alphas* 是一个参数——如图 2 中的 *Optional*——那么推导出的类型就会是一个类型变量。为 *Optional* 生成的函数和签名如下[6]：

p_Optional ::识别器 $t\ \rightarrow$ 识别器 $t p_$
Optional $p_x\ inp\ k\ \ c_0 = c_0\ k\ \vee\ p_x\ inp\ k\ \ c_0$

在不同文法之间复用非终结符定义的另一种方法，是让相互依赖的文法使用相同的词法单元类型。如果项目只涉及一个词法分析器，或者有多个词法分析器生成相同类型的词法单元，这种做法或许很实用。与 LALR 和 GLR 后端相比，这种方法带来了切实的改进，因为大型复杂文法（可能包含许多入口点）可以

[3]第一个 **%token** 指令为模式 ',' 指定的词元名称 ',',在生成的识别器名称中会被替换为 逗号 。[4]'(' 和 ')' 的名称已分别替换为 *lparen* 和 *rparen*。不过，**%name** 指令是可选的，因为默认情况下会选择文件中的第一个非终结符作为入口点。

[6]GLL 后端的翻译略有不同，因为类型推断可能会为类型添加约束。GLL 后端不进行类型推断，而是使用 Haskell 的 PartialTypeSignatures 扩展中的“通配符”。支持此扩展的编译器会将通配符替换为（带有适当约束的）类型。

分散到多个文件中。此外，采用这种方法并不妨碍在单独的文件中定义可复用的参数化非终结符库（例如图 2 中的非终结符），且无需使用 **%词元类型**指令。请注意，图 2 中 *Within*、*Optional* 和 *Multiple* 的备选项都没有引用任何词法符号。

基本递归下降的局限性。递归下降解析器通常根据向前看来选择非终结符的一个备选项（本节的识别器函数会延迟应用所有备选项，而不考虑向前看）。向前看需要预先计算每个备选项最初能够匹配的终结符集合，并在解析过程中检查输入句子中的下一个终结符是否属于该集合。然而，一般而言，向前看无法排除每一个备选项，也无法排除除一个之外的所有备选项。LL(k) 文法类定义为满足以下条件的上下文无关文法：不存在两个备选项能够同时依据 $k \geqslant 0$个向前看符号适用。选择一个备选项后，可以采用不同形式的回溯来撤销这一决定并选择另一个备选项。然而，对于任意长度的向前看 k 以及任意回溯策略，都存在最坏情况下需要的运行时间相对于输入大小呈指数增长的文法。通过记忆化可以避免这种指数级运行时间 [13, 22]。

如果一个非终结符是左递归的，那么实现该非终结符的解析函数可能会在输入未发生变化时调用自身，导致不终止。可以使用文法变换消除左递归并对文法进行左因子化，从而将文法转换为 LL(k) 文法。然而，并非每个上下文无关文法都有等价的 LL(k) 文法；如果转换后的文法与语法的语义解释相去更远，那么转换文法并不可取。当解析函数采用续延传递风格编写时，可以使用记忆化表记录解析函数的续延，从而为左递归文法构造完备解析器 [15, 16]。Frost、Hafiz 和 Callaghan 采用一种“截断”策略来处理左递归，使递归调用的次数最多不超过句子中剩余字符的数量 [12]。

Happy 的 GLL 后端生成的解析器以类似于 [15, 16] 的方式管理续延，避免所有形式的重复工作——从而防止不终止和指数级运行时间——并找出所有可能的推导。下一节介绍的文法到 GLL 解析器的转换，也保留了本节所述递归下降解析的优点。

3 生成 GLL 解析器

本节将说明 GLL 后端如何根据本文介绍的 GLL 纯函数式（FUN-GLL）变体，将 Happy 文法转换为 GLL 解析器 [38, 39]。在 [39], 中，本文展示了如何直接将 FUN-GLL 实现为解析器组合子，而无需像 [38] 中那样生成文法对象。促成 [39] 中解析器组合子设计的洞见，也使本节介绍的转换成为可能。首先概述 FUN-GLL 算法及其在 [38] 和 [39] 中面向组合子的起源。

使用 [38] 的解析器组合子编写的组合子表达式分三个阶段求值。首先，提取组合子表达式所表示的文法。随后，将该文法交给 FUN-GLL 的独立实现，该实现会生成所有可能推导的高效表示：一组 BSR 元素，其名称源自 [31] 的二叉子树表示（BSR）。提取文法时，也会同时提取求值函数。然后，将求值函数应用于 BSR 集合，通过执行组合子表达式的语义动作，为每种推导计算一个语义值（下文将此步骤称为“语义阶段”）。Ridge [25] 首次提出了使用文法组合子开发完备解析器的类似架构。在 [39] 中，研究表明，解析器组合子可以使用 FUN-GLL 的数据结构直接计算 BSR 集合，从而无需计算中间文法对象。其关键洞见在于，FUN-GLL 使用的文法信息可以针对每个非终结符在局部计算。正是这一洞见，使得我们能够从 Happy 文法生成完备、模块化且可复用的自顶向下解析器，并直接转换 Happy 的参数化非终结符。

3.1 FUN-GLL 算法

以下段落概述了 [37–39] 中对 FUN-GLL 算法的说明。表 1 给出了该算法的一次运行示例，输出结果见图3, 展示了该算法如何处理左递归和歧义。该示例取自 [39], 其文法基于 Ridge [25] 使用的文法。

考虑一个简单的递归下降解析器，它为文法中的每个符号都提供一个解析函数。该函数接收输入句子中的一个索引 l作为参数，并返回一个索引$r \geqslant l$。如果 x 的解析函数在给定 $l$$l$时返回 r，那么 x 就匹配从 l到 $r-1$ 的子句。索引l 和 r 分别称为匹配的左范围和右边界。FUN-GLL 算法通过尝试 x 的所有备选项来推广这类递归下降解析器，从而找出所有右边界。该算法通过使用描述符标记部分工作，并确保每个描述符都不会被“处理”两次，从而避免重复工作，进而防止不终止和指数级运行时间。描述符是一个三元组 $((\mathrm{x}\ \alpha, \beta), \mathrm{l}, \mathrm{k})$，其中 $(\mathrm{x}\ \alpha, \beta)$ 是一个槽位，$k \geqslant l$ 是整数。槽位 $(\mathrm{x}\ \alpha, \beta)$ 标识 x 的备选项 $\alpha\beta$中的一个位置：其前面是符号 α，后面是符号 β。描述符 $((\mathrm{x}\ \alpha, \beta), \mathrm{l}, \mathrm{k})$ 表示正在检查备选项 $\alpha\beta$ 能否用于匹配左范围为 l的子句，并指出

#	已处理描述符	动作	*uset* 扩展。	*bsrs* 扩展。	*grel* 扩展	*PRel* 扩展。
1	hE::= •EEE, 0, 0i	下降	~~1,2,3~~		hh$E$, 0i, hE::= E•EE, 0ii	
2	hE::= •a, 0, 0i	继续	4	5		
3	hE::= •, 0, 0i	上升	5	1,2		hh$E$, 0i, 0i
4	hE::= a •, 0, 1i	上升	6	6		hh$E$, 0i, 1i
5	hE::= E•EE, 0, 0i	继续	7,8	3,7	hh$E$, 0i, hE::= EE•E, 0ii	
6	hE::= E•EE, 0, 1i	下降	9,10,11		hh$E$, 1i, hE::= EE•E, 0ii	
7	hE::= EE•E, 0, 0i	继续	12,13	4,9	hh$E$, 0i, hE::= EEE•, 0ii	
8	hE::= EE•E, 0, 1i	下降	~~9,10,11~~		hh$E$, 1i, hE::= EEE•, 0ii	
9	hE::= •EEE, 1, 1i	下降	~~9,10,11~~		hh$E$, 1i, hE::= E•EE, 1ii	
10	hE::= •a, 1, 1i	继续				
11	hE::= •, 1, 1i	上升	~~8,13,~~14	8,10,11,12		hh$E$, 1i, 1i
12	hE::= EEE•, 0, 0i	上升	~~5,7,12~~	~~2,3,4~~		~~hh$E$, 0i, 0i~~
13	hE::= EEE•, 0, 1i	上升	~~6,8,13~~	~~6,7,9~~		~~hh$E$, 0i, 1i~~
14	hE::= E•EE, 1, 1i	继续	15	13	hh$E$, 1i, hE::= EE•E, 1ii	
15	hE::= EE•E, 1, 1i	继续	16	14	hh$E$, 1i, hE::= EEE•, 1ii	
16	hE::= EEE•, 1, 1i	上升	~~8,13,14,15,16~~	~~8,10,12,13,14~~		~~hh$E$, 1i, 1i~~

表 1. 使用 $E ::= EEE \mid a \mid \epsilon$ 和字符串“a”执行 FUN-GLL 的示例。BSR 元素见图3。

$$\{\langle \mathbf{E} ::= \bullet, 0, 0, 0\rangle, \quad (1)$$
$$\langle \mathbf{E} ::= \mathbf{E}\ \bullet \mathbf{E}\ \mathbf{E}, 0, 0, 0\rangle, \quad (2)$$
$$\langle \mathbf{E} ::= \mathbf{E}\ \mathbf{E}\ \bullet \mathbf{E}, 0, 0, 0\rangle, \quad (3)$$
$$\langle \mathbf{E} ::= \mathbf{E}\ \mathbf{E}\ \mathbf{E}\ \bullet, 0, 0, 0\rangle, \quad (4)$$
$$\text{小时}\mathbf{E} ::= \mathrm{a}\ \bullet, 0, 0, 1\rangle, \quad (5)$$
$$\langle \mathbf{E} ::= \mathbf{E}\ \bullet \mathbf{E}\ \mathbf{E}, 0, 0, 1\rangle, \quad (6)$$
$$\langle \mathbf{E} ::= \mathbf{E}\ \mathbf{E}\ \bullet \mathbf{E}, 0, 0, 1\rangle, \quad (7)$$
$$\langle \mathbf{E} ::= \mathbf{E}\ \mathbf{E}\ \bullet \mathbf{E}, 0, 1, 1\rangle, \quad (8)$$
$$\langle \mathbf{E} ::= \mathbf{E}\ \mathbf{E}\ \mathbf{E}\ \bullet, 0, 0, 1\rangle, \quad (9)$$
$$\langle \mathbf{E} ::= \mathbf{E}\ \mathbf{E}\ \mathbf{E}\ \bullet, 0, 1, 1\rangle, \quad (10)$$
$$\langle \mathbf{E} ::= \bullet, 1, 1, 1\rangle, \quad (11)$$
$$\langle \mathbf{E} ::= \mathbf{E}\ \bullet \mathbf{E}\ \mathbf{E}, 1, 1, 1\rangle, \quad (12)$$
$$\langle \mathbf{E} ::= \mathbf{E}\ \mathbf{E}\ \bullet \mathbf{E}, 1, 1, 1\rangle, \quad (13)$$
$$\langle \mathbf{E} ::= \mathbf{E}\ \mathbf{E}\ \mathbf{E}\ \bullet, 1, 1, 1\rangle\} \quad (14)$$

图3. $E ::= EEE \mid a \mid \epsilon$ 与字符串 "a" 的 BSR 集合。

符号 α 已匹配，左范围为l，右边界为k。在算法执行期间的任意时刻，集合*uset* 都保存着迄今遇到的描述符。

关系 *prel* 保存了针对非终结符与左范围组成的二元组（x, l）所发现的右边界，该二元组称为起始项 （与“续延”相对应）。如果元素（（x, l），r）属于 *prel*，那么 r就是起始项（x, l）的右边界。关系 *prel* 让人联想到针对 x 的解析函数的记忆化表。

关系 *grel* 将一个起始项与零个或多个延续标识符关联起来。一个延续标识符 ((x α, β), l) 本质上是一个描述符，其中第二个索引处留有空位，供为符号序列 α发现的每个右边界填入。如果 ((y, k), ((x α, β), l)) 属于 grel，且为起始项 (y, k) 发现了右边界 r，这表示 r 是 α 的右边界，因此描述符 ((x α, β), l, r) 需要进行处理（如果它不在 uset 中）。该算法保证，若 ((y, k), ((x α, β), l)) 属于 grel，则 α的最后一个符号是 y，并且 uset 中存在一个描述符 $((x, \alpha', y\beta)\ l, k)$，其中 $\alpha = \alpha' y$。

该算法通过将 BSR 元素（（$x, \alpha' y, \beta$） l, k, r）添加到集合 bsrs 中，记录描述符（（$x, \alpha', y\beta$） l, k）和起始项（y, k）的右边界 r 会产生描述符（（$x, \alpha' y, \beta$） l, r）。索引 k 在某种意义上连接了两个连续的描述符（（$x, \alpha', y\beta$） l, k）和（（$x, \alpha' y, \beta$） l, r），因此称为枢轴。在任何时刻，bsrs 都包含足够的信息来重建截至该时刻的整个解析过程，因为 bsrs 记录了每一对连续描述符的枢轴。有关 BSR 集合如何嵌入推导的完整说明，请参阅 [31]。稍后将解释，语义阶段会重新遍历句子的所有完整匹配轨迹（可能会应用歧义消解策略），并沿途执行语义动作。根据前面介绍的集合和关系，可以通过上升、下降、处理和继续这些动作来说明 FUN-GLL 算法：

- 下降起始项 (y, k)，并使用继续标识符 ((x α, β), l)，意味着：扩展 GRel，使其包含 ((y, k), ((x α, β), l))，并求出集合 R，使得对于所有 $r \in R$，((y, k), r) 都属于 PRel。– 如果 $R \neq \emptyset$，则对每个 $r \in R$，按任意顺序继续处理描述符 ((x α, β), l, r)，并将 k 作为枢轴

– 如果 $R = \emptyset$，**处理**描述符 $((y\ \epsilon, \delta), k, k)$，对于 y 的每个备选项 δ，并按任意顺序（ϵ 表示空符号序列）

- 要以右边界r上升起始项（x,k），意味着：扩展 $prel$ 使其包含 $((x, k), r)$，并找出所有满足 $((x,k), (s,l))$ 属于 $grel$ 的续延标识符（s,l），然后以任意顺序对每个描述符（s, l,r）和枢轴 k 继续处理
- 要以枢轴k继续处理描述符 $((x\ \alpha, \beta), l, r)$，意味着：将 BSR 元素 $((x\ \alpha, \beta), l, k, r)$ 添加到 $bsrs$，然后**process** 描述符 $((x\ \alpha, \beta), l, r$
- 通过将描述符 ((x α, β), l, r) 添加到 USet 来处理它；仅当它之前不在 USet 中时，才执行以下操作：– 如果 β 是空符号序列，则以右边界 r 上升起始项 (x, l)。– 如果 $\beta = t\beta'$ 中的 t 是一个词元，则将 t 与句子中位置 r 处的词元匹配；若匹配成功，则以 $((x, \alpha t, \beta')\ l, r + 1)$ 继续处理，并将 r 作为枢轴。– 如果 $\beta = y\beta'$ 中的 y 是非终结符，则以续延标识符 $((x, \alpha y, \beta')\ l)$ 下降 (y, r)

表 1 中的示例运行从处理描述符 hE::= •EEE，0，0i、hE::= •a，0，0i 和 hE::= •，0，0i 开始。符号 • 标记由槽位标识的备选项中的位置。重复的集合元素以删除线标出（描述符 #10 对应的动作带有删除线，因为字符‘a’不是输入中的第二个字符）。描述符按先进先出的顺序处理，也就是说，第 i 个遇到的描述符也是第 i 个被处理的描述符。选择这种顺序，便可用简洁的表格形式展示算法的运行；采用任何其他顺序都会得到相同的输出。

3.2 翻译 Happy 文法

本小节通过示例说明 GLL 后端如何将 Happy 文法中的参数化非终结符翻译为实现 FUN-GLL 算法的高阶函数，并解释前述“语义阶段”是如何实现的。 -

本小节通过示例说明 GLL 后端如何将 Happy 文法中的参数化非终结符翻译为实现 FUN-GLL 算法的高阶函数，并解释前述“语义阶段”是如何实现的。示例采用图 5 中的 CSV 定义以及图 6 中的生成代码。**%词元**指令（以及其他 Happy 指令）的翻译已省略。图 4 展示了实现的支持库所提供的 FUN-GLL 算法的数据结构[7]和操作。

Happy 文法中的每条 **%词元**指令和每个非终结符定义都会生成一个返回符号的函数。如果函数是为参数化非终结符生成的，那么该函数的每个参数都对应一个类型为 符号 的参数。正如示例所示，当某个符号用于为另一个符号生成的代码中时，它

[7]一些类型定义使用 ≡ 而非 = 来表示这些数据结构等价于集合，但采用了不同的实现方式以提高效率。

```
data SID = Tn 字符串 | App 字符串 [SID]type
Slot = (SID, [SID], [SID]) type Comm = (SID,
Int) -- 起始项 type CID = (Slot, Int) -- 延续标识
符 type CMap t = Map CID (Cont t) type
USet ≡ Set Descr addDescr :: Descr → USet
 → USet hasDescr :: Descr → USet → Bool
type GRel ≡ Set (Comm, CID) addCont ::
Comm → CID → GRel → GRel conts ::
Comm → GRel → [CID]type PRel ≡ Set
(Comm, Int) addExtent :: Comm → Int →
PRel → PRel extents :: Comm → PRel → [
Int ]type BSRs ≡ Set BSR type BSR = (Slot, Int,
Int, Int) addBSR :: BSR → BSRs → BSRs
pivots :: (Slot, Int, Int) → BSRs → [Int]
```

图 4. FUN-GLL 的数据结构与操作。

```
CSV (v) : CSV (v) ',' CSV (v) {$1 + $3}
        | v                   {[$1]}
```

图 5.一个左递归参数化非终结符。

符号是作为参数提供，还是在同一命名空间中定义，都无关紧要。

```
数据符号 t a = 符号{idsm :: SID, matchsm :: 匹配器 t,
evalsm :: 求值器 t a}
```

除了标识符之外，每个符号都由一个“匹配器”和一个“求值器”组成，分别实现 FUN-GLL 和语义阶段。符号标识符（SID，见图 4）可以是词元名称（Tn），也可以是应用了零个或多个符号标识符参数的非终结符名称（App）。符号的类型参数t 和a 分别表示词法单元的类型，以及求值器生成的语义值类型。图 6 中的生成代码展示了如何根据CSV 定义中提及的符号和参数，计算该定义所需的槽位（在代码中命名为 $s_0, ..., s_5$）。

匹配器（匹配器）接收一个索引（枢轴或右端位置），以及一个与其续延标识符（类型为 CID）配对的续延。匹配器会按照前一小节所述的 **继续**、**上升** 或**下降**动作，在 FUN-GLL 数据结构（$Data$）上执行必要的操作。不可变数组用于存储输入句子，以便快速访问。

```
type Matcher t = Int → (CID, Cont t) → Data t → Data t
type Cont t    = Int → Int           → Data t → Data t
```

```
sCSV x_1 = 符号 nt 匹配器 求值器
  其中
    nt = 应用"CSV" [idsm x_1]匹配器 l (c, cf) = 下降 (nt, l) (c, cf) (c0
    l l ∘ c_4 l l) 其中c0 k r = 继续 (s_0, l, k, r) (匹配sm (sCSV x_1) r ((s
    _1, l) c1))c1 k r = 继续 (s_1, l, k, r) (匹配sm sCom r ((s_2, l) c2))c2 k
    r = 继续 (s_2, l, k, r) (匹配sm (sCSV x_1) r ((s_3, l), c_3)) c_3 k r =
    继续 (s_3, l, k, r) (上升 (nt, l) r) c_4 k r = 继续 (s_4, l, k, r) (匹配sm
    x_1 r ((s_5, l), c_5)) c_5 k r = 继续 (s_5, l, k r) (上升 (nt, l) r) s_0 =
    (nt, [ ], [idsm (sCSV x_1), idsm sCom, idsm (sCSV x_1) ]) s_1 = (nt, [i
    dsm (sCSV x_1) ], [idsm sCom, idsm (sCSV x_1) ]) s_2 = (nt, [idsm
    (sCSV x_1), idsm sCom ], [idsm (sCSV x_1) ]) s_3 = (nt, [idsm (sCSV x
    _1), idsm sCom, idsm (sCSV x_1) ], [ ]) s_4 = (nt, [ ], [idsm x_1]) s_5 =
    (nt, [idsm x_1], [ ]) 求值器 inp bsrs l r nts | nt ∈ nts = [ ] 否则 =
    res

  其中
    nts' p q | l . p ∨r . q = empty| 否则 =插入 nt nts res
    = [ sv_1 +sv_3 -- 第一个备选项的语义动作| p_3 ← 枢轴
    (s_3, l, r) bsrs , p_2 ← 枢轴 (s_2, l, p_3) bsrs , p_1 ←
    枢轴 (s_1, l, p_2) bsrs -- p_1 ≡ l , sv_3 ←求值sm (sCSV x
    _1) inp bsrs p_3 r (nts' p_3 r) , sv_2 ←求值sm sCom i
    np bsrs p_2 p_3 (nts' p_2 p_3) , sv_1 ←求值sm (sCSV x_1)
    inp bsrs l p_2 (nts' l p_2)| ⊣[ [sv_1] -- 第二个备选项的
    语义动作 , p_1 ← 枢轴 (s_5, l, r) bsrs -- p_1 ≡ l| sv_1 ←
    求值sm x_1 inp bsrs l r (nts' l r) ]
```

图 6。图 5 中产生式的 GLL 后端生成的代码。*sCom* 是词元"**,**"的符号 。

```
数据 Data t = Data {uset :: USet, bsrs :: BSRs, grel :: GRel ,
cmap :: CMap t, prel :: PRel, inp :: 数组 Int t }
```

续延标识符是一个由槽位和左范围组成的对，如上一小节所述。续延本身（*Cont*）捕获了上一节中**继续** 动作的行为。生成代码中的函数 $c_0, \ldots, c_5$ 是为*CSV*定义的续延函数，每个槽位对应一个函数。这些定义局部于*CSV* 匹配器的定义，因为调用匹配器时会传入左范围 *l*。续延所针对的槽位（例如 s_0 对应 c_0）、左范围 *l*（匹配器的参数）以及续延的参数（*k* 和*r*）共同构成执行**继续**动作所需的描述符和 BSR 元素。支持函数继续 将二者作为一个单独的（BSR）参数接收，并实现**继续**的行为。

```
继续 ::BSR → (Data t → Data t) → Data t → Data t
继续 bsr cf d =maybeProcess bsr cf (addBSR bsr d)
  maybeProcess (s, l, k, r) cf d| hasDescr (s, l, r) (uset
  d) = d| 否则 = cf (d {uset = addDescr (s, l, r) (uset
  d)})
```

BSR 元素总是由*bsrs* 中的继续添加，而*cf*（捕获下一步处理的效果）仅在描述符尚未位于*uset*中时才会应用。继续 的第二个参数是根据槽位，将匹配器应用于下一个待匹配符号后得到的结果（一个函数*Data t* →*Data t*），即槽位第三个分量中的第一个符号（例如，在 c_4 和 s_4的情况下为参数 x_1）。匹配器会应用于右范围 *r*，以及"下一个"槽位对应的续延与续延标识符对（例如，在 s_0 的情况下为 s_1，在 s_1的情况下为 s_2，等等），且左范围相同。或者，如果槽位的第三个分量为空（例如，在 s_3 和 s_5的情况下），则会应用支持函数上升，对应于**上-升**动作。换言之，为槽位（$x\ \alpha, \beta$）生成的续延函数，会向继续 提供为 β 的第一个符号生成的匹配器函数；如果没有这样的符号，则提供函数上升 。通过这种方式执行**处理**动作。

函数 上升 接收一个起始项（*nt*, *l*）和一个右边界 *r*，并执行**上升** 动作。**上升**动作包括查找*grel* 中为给定起始项存储的所有延续标识符*c* 。由*c* 标识的延续会记录在映射*cmap* 中（见下文下降 的定义）。

```
上升 ::Comm → Int → Data t → Data t ascend (nt, l) r d =
foldr (($) ∘ (λcf → cf l r)) id cs d其中 d = d {prel =
addExtent (nt, l) r (prel d)}cs = map (flip lookup (cmap d))
(conts (nt, l) (grel d))
```

应用于 *l* 和 *r*（从*Data* → *Data*到函数的映射，每个*c*对应一个）的不同延续所产生的效果，会通过函数复合累积起来。由于 FUN-GLL 可以按任意顺序处理描述符，因此复合顺序不会影响结果。

词元的匹配器（此处未展示）以类似 *matchPattern* 的支持函数定义（见第 2 节）。名称为 *nt* 的非终结符的匹配器，会对起始项（*nt,l*）以及与标识符 *c*配对的续延 *cf* 应用 *descend*（参见 *matcher* 对 *CSV*的定义）。参数 *l*和（*c, cf*）本身也是匹配器的输入，在应用续延时提供（参见 $c_0, \ldots, c_5$ 在为 *CSV*生成的代码中的定义）。函数 *descend*还会接收处理 *nt*各备选项所产生的效果。在此示例中，*CSV*各备选项所产生的效果，是通过组合应用续延 c_0 和 c_4 所得的效果生成的；这两个续延分别匹配两个备选项的首个符号。

```
descend :: Comm → (CID, Cont t) → (Data t → Data t)
                → Data t → Data t
descend (x, l) (c, cf) alts d
```

```
| 空 rs =备选项 d小 否则 = foldr (($)∘ (λr →
cf l r)) id rs d小其中 rs = 范围 (x, l) (prel
d)d小 = d {grel = addCont (x, l) c (grel
d), cmap = insert c cf (cmap d)}
```

根据给定起始项（$nt,\ l$）在 $prel$ 中是否存储了右边界（rs），下降 会将续延 cf 应用于每个右边界，或者应用备选项（$alts$）的效果。两种情况下都会扩展 $grel$。

CSV 的定义是左递归的，其匹配器会在不消耗输入句子中的词法单元的情况下调用自身（当续延 c_0 与 $r = l$一同应用时）。然而，这不会导致不终止，因为第一次应用继续 会将描述符（$s_0,\ l,\ r$）（带有 $l = r$）添加到 $uset$，而该描述符会在第二次应用时被“注意到”。

语义阶段。 下文定义的函数run[8] 会将一个符号 转换为解析器，即一个将句子映射为语义值列表的函数。

```
run:: 符号 t a → 数组 Int t → [a]
run x str = evalsm str (bsrs d_1) 0 (length str)
empty where -- 以下省略的所有字段均为空 d_0 =
Data {inp = str, uset = empty ... } -- 以下，id 是
Data t 上的恒等函数 d_1 = matchsm 0 (c0, id) d_0--
"__START" 是一个人为添加的起始符号c0 = ((App "__
START" [ ], [idsm x ], [ ]), 0)
```

解析器的结果包含一个语义值，对应于匹配器为该符号计算出的 BSR 集合中编码的句子的每一种推导。这些语义值基于文法中的语义动作，并由该符号的求值器计算得出。此处虽未展示，GLL 后端允许用户将消歧策略实现为针对 BSR 集合所计算出的语义值的过滤器。这些策略可以基于优先级、结合性规则，或语义值本身。本节余下内容假定未采用任何消歧策略。

如下文所述，如果没有消歧策略，由于存在能为句子生成指数多种推导的文法，语义阶段在最坏情况下的运行时间会呈指数级。事实上，含有同时具有左递归和右递归的非终结符的文法，可能产生无限多种推导。不过，为避免不终止，语义阶段不会考虑此类循环非终结符的所有推导。Ridge [25] 首次描述了类似的语义阶段。

[8]支持库定义了若干种 *run* 的变体，用于构造部分解析器（以及实现 **%partial** 指令），并构造返回 *Either* [字符串] [a] 类型值的解析器，其中左侧分量是一系列错误。

```
类型求值器 t a =
    Array Int t → BSRs → Int → Int → Set Symbol → [a]
```

求值器函数（符号的第三个分量）接收输入句子 inp、一组 BSR 元素$bsrs$、左范围 l、右范围 r 以及一组（非终结符）符号nts。对于由$bsrs$编码、范围从 l 到 $r-1$ 的inp 子句的每个推导，它都会返回一个语义值；但如果该求值器用于nts中的某个非终结符，则不在此列。非终结符 nt 的求值器函数，是根据nt 定义中出现的符号的求值器函数来定义的（参见图 6 中$eval_{sm}$ 的用法）。nt 的求值器可能会递归调用自身，CSV 就是这种情况。通过维护一个已遇到的非终结符集合 nts，并检查nt 的求值器是否在$nt \in nts$中时被调用，可以检测递归调用，从而确保语义阶段终止。为了只检测求值器同时存在左递归和右递归的情况，该算法会在两次求值器调用之间 l 或r 发生变化时清空集合nts （参见示例代码中nts' 的定义）。

nt 的求值器结果（图 6 中的 res）是 nt 各备选项结果的串接。如果某个备选项有 $k >= 1$ 个符号，那么求值器会找出 inp 中从 l到 $r-1$ 的子句的所有长度为 k的划分，使得划分中的每个第 i个元素都与该备选项的第 i个符号匹配。从本质上说，找出一个或多个这样的划分正是解析算法所做的事。不过在此情况下，计算这些划分所需的信息已经编码为 bsrs 中的枢轴。函数 pivots（见图 4）接收一个三元组 (s, l, r)，并返回所有满足 (s, l, k, r) $\in$bsrs 的枢轴 k。槽位 s的形式为 $(x, \alpha y, \beta)$。通过生成枢轴 k，BSR 集合告知求值器，α与左范围 l和右范围k相匹配，而 y 与左范围 k 和右范围 r 相匹配。求值器通过继续处理三元组 $((x, \alpha, y\beta)\ l, k)$ 等，找出所有划分；对所有 k 重复此过程，直到所有槽位都已检查过（形式为 (x, ϵ, β) 的槽位除外）。对于每个划分，都会调用备选项中第 i个符号的求值器，并传入该划分中第 i个元素的左、右范围。这些调用产生的语义值构成了附加到该备选项上的语义动作的输入。

如果 $k = 0$，则返回包含语义动作结果的单元素列表；如果不存在 BSR 元素（$((nt\ \epsilon,\ \epsilon)\ l,\ l,\ r) \in bsrs$，只有在$l = r$时才可能为真），则返回空列表。

词元的求值器（未显示）会检查给定的左边界和右边界是否相差 1，即检查 $l+1 = r$；如果是，则返回包含匹配的词元的单元素列表（否则返回空列表）。

4 实现

本文的补充材料包含一个实现了 GLL 后端的 Happy 版本。本文将在本节讨论这一实现的

相关方面。我们还将讨论其相对于 LALR 后端的完整性，以及若干改进和扩展的方向。

数据结构与操作。 上一节将 FUN-GLL 算法的核心数据结构描述为集合，未涉及其具体实现，而是侧重于解析器的生成方法。然而，FUN-GLL 的效率和基准情况复杂度深受数据结构及其操作的影响。直接将其实现为集合（例如使用 *Data.Set*）效率低下。因此，*USet* 被定义为嵌套的整数字典树 [23]，其嵌套层级对应描述符中的两个整数：

```
类型USet = IntMap (IntMap (Set Slot))
```

然而，这种方法并不足够：评估一节表明，在应用于 LALR 文法时，GLL 后端的速度明显慢于 LALR 和 GLR 后端。支持本文的实现工作被用于展示 GLL 后端的能力，并构建了一组可用于测试更快实现的示例。未来工作可以探索替代实现。如果不要求模块化和复用，也可以采用更快的解析器生成策略。

向前看。 GLL 后端目前尚未实现任何形式的向前看。不过，可以采用与计算槽位本身类似的方式，为每个槽位动态计算前瞻集。一种可行的方法是扩展类型 符号 ，使其包含前瞻集，并生成用于计算这些集合的表达式。如果不需要模块化和复用，也可以静态计算前瞻集。

单子动作。 图 7 展示了一个 Happy 文法，它利用 GLL 后端实现的若干额外功能，定义了一种用于整数算术的小型表达式语言。语义动作定义了该语言的解释器。**%单子** 指令会影响生成函数的类型签名，也会影响语义动作的处理方式。在这个示例中，语义动作产生的语义值是 *SeedM* 单子中的计算，这由 **%单子**指令指定。该语言的定义方式是：每当遇到“#”，就通过应用 *enroll*生成一个新的整数。

该示例展示了四种语义动作，分别由动作前面的零个、一个、两个或三个百分号符号加以区分。第一种 {...} 无论是否使用 **%Monad** 指令，都以相同方式编写语义动作（例如乘法的备选项）。由于示例中有 **%Monad** 指令，因此在将动作插入生成的解析器之前，会先对其进行转换，从而形成等价形式：

```
do v_1 ← x_1
   v_2 ← x_2
   v_3 ← x_3
   return (v_1 ∗ v_3)    -- for the action { $1 ∗ $3 }
```

```
%error {error ∘ show } %monad {SeedM Int } %token ... -- 词法单元指令已省略 %left '-' '+' -- %left 指令的顺序 %left '*' '/' -- 决定优先级 %% Expr : Expr '+' Expr {% return ( $1 + $3) } | Expr '-' Expr {%% (–) h$i $1 h∗i $3}| Expr '*' Expr { $1 ∗ $3}| Expr '/' Expr {%%% [div h$i $ 1 h∗i $3| 0 ≠ giveValue $3 ]}| '(' Expr ')' { $2}| digit { $1}| '#' {% enroll } { data SeedM Int a = SeedM {runSeedM :: (Int → (Int, a)) } instance Monad (SeedM Int) where return a = SeedM (λi → (i, a)) (SeedM p) ≫=mq = SeedM $ λseed →let (seed', pv) = p seed in runSeedM (mq pv) seed' giveValue :: SeedM Int a → a giveValue sm = snd (runSeedM sm 1) enroll :: Enum s ⇒ SeedM s s enroll = SeedM (λi → (succ i, i)) }
```

图 7. 展示由 GLL 后端实现的 Happy 其他功能的示例文法。

通俗地说，备选项中三个符号对应的三个单子解析结果会被“run”，以产生它们的值。程序员编写的语义动作就是返回的表达式，其中的 $1 和 $3 会替换为绑定第一个和第三个符号语义值的标识符。第二种形式 { %... } 的语义动作有所不同，因为不会隐式应用 *return* ；用户表达式必须具有正确的单子类型。这意味着程序员可以在语义动作中访问该单子（例如，参见针对“#”的备选项中的 *enroll* ）。在第三种形式 { %%... } 中，动作参数 $1 、 $2 、 $3 等会替换为绑定备选项中各符号对应的单子解析结果的标识符，而不会对它们进行“run”。因此，程序员可以完全控制语义动作所构造的单子计算（例如，参见减法的备选项）。此外，第四种形式[9] { %%%... } 让程序员可以访问

[9]如果没有 **%Monad** 指令，则第四种语义动作的行为会赋予第二种语义动作。

求值阶段的列表单子。这个强大功能可用于根据子表达式的语义值实现任意消歧策略。在示例中，该功能用于排除会导致除零错误的表达式。

错误处理程序。 此示例还展示了，可以使用 **%error**指令为解析错误指定处理程序。这并非易事，因为 GLL 算法会探索输入句子的所有可能解释。如果解析失败，可能会有多个失败点，而且无法明确应该将哪个报告为错误。文法的每个产生式都会生成一个包装函数，用于应用该产生式的解析器，并将错误作为 *Either* 的左侧分量返回。错误值有 n[10]，每个值对应 n 中深入输入句子最远的解析尝试之一。其数量 n 由解析器生成期间或运行时的配置选项决定。

消歧。 消歧指令 **%left**、**%right** 和 **%nonassoc** 在 GLL 后端中的行为与在 LALR 和 GLR 后端中略有不同。这是因为 GLL 后端通过筛选 BSR 集合来实现这些指令，而不是在分析表中解决冲突。其具体行为差异还有待进一步研究。

线程化词法分析器。 Happy 的 **%词法分析器** 指令使得可以基于一种“线程化词法分析器单子”进行解析，该单子会在其状态中传递输入句子。这使错误处理程序和语义动作能够与词法分析器状态交互，例如根据输入中下一个字符的行号和列号生成消息。这一特性尚未实现。要实现它，解析算法必须能够“重置”词法分析器状态，以便探索一个句子可能的多种解析方式。

属性文法。Happy 的语义动作允许程序员按照《龙书》 [3]的风格编写任意语法制导翻译。属性文法形式体系 [19] 可以解释为一种特定的语法制导翻译编写方式，其中语义方程将属性定义为彼此之间的关系。求值器会遍历程序，通过反复“访问”其组成部分来求解这些方程，直到为所有属性赋值。Haskell 的惰性求值使得我们能够以模块化的方式为属性文法中的每个产生式生成求值函数 [26,33]。然而，如果属性之间的依赖关系构成环路，求值器就无法终止。Happy 采用了这一策略，并将属性文法作为语义动作的替代方案。让 GLL 后端支持属性文法求值应该没有问题，因为 Happy 的属性文法求值策略与生成的 GLL 解析器的语义阶段天然契合。不过，当前的实现尚未展示这一点。

[10]其类型由错误处理程序推断。

5 评估

本节将评估 GLL 后端实现的特性，并与 LALR 和 GLR 后端进行比较。第 2 节讨论了 GLL 后端所独有的模块化和组合性。所有实验均在一台搭载 Intel i7-8565U（8）@ 4.600GHz 中央处理器、配备 16 GiB 随机存取存储器的笔记本电脑上，使用 Linux Mint 20.3 和 8.6.5 版格拉斯哥 Haskell 编译器（GHC）运行。重现实验结果所需的工具和文件作为补充材料提供。

超越上下文无关文法。 Happy 的参数化非终结符可用于编写非上下文无关语言的文法。例如，以下代码片段展示了语言 $a^n b^n c^n$ with$n \geqslant 1$ 的文法（此例取自 [11]）。

```
Start      : F ('a', 'b', 'c')
F (x, y, z) : F (Seq (x, 'a'), Seq (y, 'b'), Seq (z, 'c'))
            | x y z
Seq (p, q) : p q
```

LALR 和 GLR 后端依赖一种不动点算法，该算法通过类似宏展开的处理，将参数化非终结符替换为特化的（非参数化）变体。然而，此算法无法移除这样的参数化非终结符：它们递归地调用自身，且每次递归调用的参数都会改变。在此例中，算法试图计算一个文法，其中包含无限多个 F 的特化形式，每种 n取值对应一个。Happy 不会检测到这种情况，因此在此例中会一直运行下去，也不会发出警告。

如第 2 节和第 3 节所述，GLL 后端生成的解析器利用 Haskell 的抽象机制来实现参数化非终结符，从而动态执行应用。在此例中，虽然生成了一个 GLL 解析器，但它对任何输入句子都无法终止。这是因为每次调用实现 F 的解析器时，都会产生一个新的非终结符名称，因此也会产生一个对此次调用而言独一无二的描述符，它尚未存在于描述符集合中。这可以看作是左递归问题的一种高阶变体。

以下片段中的文法生成上下文有关语言 Í $_{i=0}^{\infty}(b^i a c^i)$。也就是说，该语言为 {"a"、"a(a)"、"a(a)((a))"…}；为清晰起见，用 '(' 代替 'b'，用 ')' 代替 'c'。

```
开始： List ('a')
List (e) ： e
```

| *e List* (圆括号 (*e*))圆
括号 (*e*) : '('*e*')'

List 的递归位于右侧，为该文法生成的 GLL 解析器确实能够识别该语言。这是因为，每次对 *List* 解析器的递归调用，都会接收到一个输入句子索引，且该索引更接近句子末尾。一旦整个句子被“消耗”，就不会再进行递归调用。这一观察表明，采用 [12] 的截断过程的一种变体，可以通过确保递归调用次数至多等于输入句子中剩余词法单元的数量，来克服高阶左递归问题。其他后端无法处理此示例，原因如前所述。

排列短语。 接下来的实验探讨如何解析“排列短语”[4]。 [39]中也进行过类似的实验。排列短语是由可排列元素组成的序列，其中每个元素恰好出现一次，且顺序不限 [8]。本文讨论的排列短语中，元素最多 出现一次。现实中的排列短语示例包括 Java 中与字段和方法相关联的修饰符 [35]，以及 C 的声明说明符 [8]。排列短语的语法可以通过一个非终结符来描述，为每一种可能的排列设置一个备选项。由于备选项的数量会随可排列元素数量呈指数增长，这种文法并不实用。下面的代码片段展示了如何方便地使用参数化非终结符来描述排列短语的语法。此表述基于 [35]中的 PermP3 示例。可排列元素为数字 1 到 4。假定字符 '$' 不会出现在任何输入句子中，从而确保针对*Nul*的解析器始终失败。*Choose*的每个备选项都会选择一个元素，并进行递归调用以继续选择。在递归调用中，所选元素会被替换为*Nul*，使其之后无法再被选中。

```
Permutations : Choose ('1','2','3','4')  { $1}
Choose (a, b, c, d) :                    {[]}
                    | a Choose (Nul, b, c, d)  {( $1 : $2)}
                    | b Choose (a, Nul, c, d)  {( $1 : $2)}
                    | c Choose (a, b, Nul, d)  {( $1 : $2)}
                    | d Choose (a, b, c, Nul)  {( $1 : $2)}
Nul                 : '$'                      {'$'}
```

Happy 算法成功地移除了参数化非终结符，并为这个示例生成了等价文法。然而，与可排列元素的数量相比，生成的文法包含数量呈指数增长的备选项。表 2 展示了这种指数级增长，其中列出了为包含四个、五个和六个元素的排列短语语法生成的 LALR 解析器的相关数据。表中显示了生成解析器所用的时间、生成的解析器大小（以千字节为单位）、GHC 编译该解析器所用的时间，以及解析器所用的时间来

Alg.	#	生成	Size	编译	解析
LALR	4	0.009秒	45.0千字节	1.226秒	0.001秒
	5	0.054秒	88.0千字节	3.003秒	0.001秒
	6	0.679秒	191.0千字节	11.751秒	0.001秒
GLL	4	0.004秒	34.0千字节	0.964秒	0.001秒
	5	0.004秒	35.0千字节	0.983秒	0.002秒
	6	0.004秒	37.0千字节	1.026秒	0.002秒

表 2. 排列短语的语法分析。

Alg.	194	424	934	1541	2233
LALR	0.0s	0.0s	0.01秒	0.01秒	0.01秒
GLR	0.01秒	0.02秒	0.04秒	0.06秒	0.08秒
GLL	0.02秒	0.04秒	0.11秒	0.17秒	0.29秒

表3。 LBNF 流水线的运行时间。

解析一个排列。相比之下，对于由相同文法生成且未移除参数化非终结符的 GLL 解析器，其数据呈现出小幅线性增长。

LALR 文法。 在下一项实验中，我们将三种不同后端生成的解析器的运行时间与 LALR 文法进行比较。BNF 转换器（BNFC）根据以标记 BNF（LBNF）形式编写的单个语法描述，生成词法分析器、解析器和抽象语法 [1]。实际上，该转换器会生成一个小型流水线，用于运行词法分析器和解析器、创建抽象语法树并将其美化打印。BNFC 工具能够根据 LBNF 文法生成两种 Happy 文法：一种用于 LALR 后端，也可应用 GLL 后端；另一种专用于 GLR 后端，因此它是生成本实验输入的理想工具。BNFC 项目包含大量用于真实世界语言的 LBNF 文法，例如 Java、ANSI-C 和 Prolog。不过，这些示例并未附带许多测试，而且并不总是清楚其语法描述适用于该语言的哪个版本。实验采用了 ANSI-C 和 LBNF 本身的流水线。

使用 Prolog、LBNF 本身、OCL（对象约束语言）、GF（语法框架）和 ANSI-C 的 LBNF 文法作为测试输入，对 LBNF 流水线进行了实验。表3展示了在这些输入上运行该流水线的运行时间。列标题包含流水线的词法分析器生成的词法单元数量。表格中 LALR、GLR 和 GLL 各行的唯一区别，在于用于生成该流水线解析器的后端不同。

ANSI-C 流水线实验以类似的方式进行，使用了两个小型程序和一个取自 BNFC 代码库的大型程序。最大的程序确实很大，代码超过 7500 行，

Alg.	46	271	7261	15425(x2.1)	26106(x1.7)
LALR	0.0s	0.0s	0.05秒	0.1秒(x2.0)	0.18秒(x1.8)
GLR	0.01秒	0.02秒	0.57秒	1.23秒(x2.2)	2.0秒(x1.6)
GLL	0.04秒	0.23秒	7.13秒	15.71秒（x2.2）	28.41秒（x1.8）

表4。 ANSI-C 流水线的运行时间。

大小为 187KB。两个较小但仍然很大的程序，是将最大程序的前 2500 行和前 5000 行代码分别复制到单独的文件中构造而成的。这样做是为了展示运行时间的线性增长。表 4 显示了在五个输入程序上执行 ANSI-C 流水线的运行时间。最后一列和倒数第二列给出了乘数，表示单元格中的数值相对于前一列的增长倍数。

表3和表4显示，在 LALR 文法上，LALR 解析器的速度明显快于 GLR 解析器，而 GLR 解析器又明显快于 GLL 解析器。GLL 后端的实现尚未针对性能进行优化。上一节讨论了多种提升 GLL 解析器速度的可行方法。例如，可以动态计算前瞻集（以保持模块化）。或者，也可以在生成 GLL 解析器时不考虑模块化和参数化非终结符，从而能够进行更多优化，例如静态计算前瞻集。

高度歧义文法。在下一个实验中，我们测试了 GLL 和 GLR 后端处理高度歧义文法的性能。所选文法曾由 Ridge [25] 用于与 Happy 的 GLR 后端进行比较。这些文法对完备解析器构成了严峻的压力测试。实验中未使用 LALR 后端，因为它无法生成完备解析器。每个文法都由一个非终结符组成，并生成语言 a^n，其中 $n \geqslant 0$。非终结符定义如下：

$$S_1 : \texttt{'a'}\ S_1\ S_1 \mid \qquad S_2 : S_2\ S_2\ \texttt{'a'} \mid \qquad E : E\ E\ E \mid \texttt{'a'} \mid$$

这些文法的推导数量相对于输入大小呈指数级增长。因此，不可能高效地枚举所有推导，GLR 和 GLL 解析器仅用于识别。GLL 解析器仍会计算一个包含所有推导的 BSR 集合。

表 5 展示了 GLR 和 GLL 解析器在输入包含 20 至 200 次字符‘a’重复时的运行时间。对于这些文法，GLL 解析器明显快于 GLR 解析器。事实上，针对非终结符 S_1 和 E 的 GLR 解析器似乎会出现指数级膨胀（针对 S_1 和 E 的 GLR 解析器运行

Alg.	**Nt.**	20	30	40	50	100	200
GLR	S_1	0.11秒	0.81秒	4.18秒	13.96秒	-	-
	S_2	0.004秒	0.01秒	0.02秒	0.03秒	0.32秒	3.46秒
	E	0.17秒	1.73秒	8.18秒	34.96秒	-	-
GLL	S_1	0.006秒	0.02秒	0.05秒	0.08秒	0.82秒	7.43秒
	S_2	0.007秒	0.01秒	0.03秒	0.06秒	0.45秒	3.1s
	E	0.012秒	0.03秒	0.08秒	0.12秒	1.06秒	7.72秒

表 5. 高度歧义文法的识别时间。

失败，在 100 个词法单元的输入上，超时设为 500 秒），尽管理论上 GLR 算法不会 [30, 36]。

6 相关工作

GLL 算法是语法分析领域中相对较新的发展，出现了多种变体 [27, 28]，[2, 29, 38, 39],，重新激发了人们对自顶向下解析的兴趣。该算法的实现目前尚未广泛普及，但可以在解析器生成器 [6, 29] 和组合子库 [15,38, 39]中找到。据我们所知，Happy 的 GLL 后端是首个将 GLL 解析与抽象与复用功能相结合的解析器生成器，并直接受益于该算法的自顶向下特性。PRECC 编译器生成器将 LL(∞) 解析与抽象与复用功能相结合，但不会生成完备解析器 [7]。OCaml 解析器生成器 Menhir 可根据带有参数化规则的文法生成 LR(1)语法分析器，并会针对可能导致无法终止的无界增长规则发出警告 [24]。Menhir 提供了一个可复用规则库，涵盖可选性、序列和列表。作为嵌入式领域专用语言，解析器组合子库 [32] 和文法组合子库 [9] 天然支持复用，但其可接受的语言类别以及编写解析器的便捷程度差异很大。关于这一主题的更详细讨论，请参阅 [37, 39]。

Fischer 在 1968 [11], 宏文法中引入了“类似宏的产生式”，将上下文无关文法扩展为类似 Happy 的参数化非终结符定义。宏的参数是一个符号序列，而参数化非终结符的参数则是单个符号。因此，尽管宏文法的表达能力更强，但应当可以按照第5节中非终结符 F所示的方法，为每个宏文法编写一个 Happy 文法。Thiemann 和 Neubauer 讨论了如何为受限宏文法生成 LR 解析器 [34]，并描述了一种用于检查宏文法能否转换为上下文无关文法的算法 [35]。这一算法的一个变体可以在 Happy 中实现，以防止用于 LALR 和 GLR 后端的参数化非终结符消除算法不终止。或许可以扩展这一算法，以检测第5节所称的“高阶左递归”，从而

防止 GLL 后端为无法终止的参数化非终结符生成解析器。

7 结论

本文介绍了一种策略，该策略基于 GLL 算法的 FUN-GLL 变体，可从包含参数化非终结符的语法描述中生成模块化、可复用且完备的自顶向下解析器。该策略已在 Haskell 中实现为 Happy 的一个新后端。本文中的思想应可移用于采用高阶函数的其他文法形式体系和宿主语言。生成的解析器易于测试和调试，因为每个文法符号都直接实现为可执行的解析函数。此外，GLL 后端通过生成类似于解析器组合子的高阶解析函数，直接支持参数化。因此，该后端能够为一大类文法生成实用的解析器，包括所有上下文无关文法以及某些描述上下文有关语言的文法。

新的 Happy 后端被开发为 LALR 和 GLR 后端的一种实用替代方案，同时通过继承递归下降解析的优点（例如模块化），并充分发挥“通过抽象实现复用”的潜力，扩展了 Happy 的功能和易用性。不过，生成的 GLL 解析器的运行时效率仍有提升空间。对于高度歧义的文法，GLL 解析器明显快于 GLR 解析器；但对于 LALR 文法，其速度则明显慢于 LALR 和 GLR 解析器。或许可以通过重新实现支持库提供的数据结构并加入向前看测试，来缩短生成的 GLL 解析器的运行时间。不过，一些设想中的效率提升需要预先计算基于整个文法的信息，这会导致解析器无法在不同文件和项目之间复用。未来的工作将探索这些可能的运行时优化。

参考文献

[1] Andreas Abel 和 Grégoire Détrez。2010。BNFC 的 GitHub 代码库。https://github.com/BNFC。 [在线；访问日期: 2020]年 5 月 15 日。[2] Ali Afroozeh 和 Anastasia Izmaylova。2015。更快速、实用的 GLL 解析。载于编译器构造。Springer Berlin Heidelberg，第 89–108 页。[3] Alfred V. Aho、Monica S. Lam、Ravi Sethi 和 Jeffrey D. Ullman。2006。《编译原理、技术与工具》（第 *2* 版）。Addison-Wesley Longman Publishing Co., Inc.，美国。[4] Arthur I. Baars、Andres Löh 和 S. Doaitse Swierstra。2004。《解析排列短语》。《函数式编程杂志》14 卷第 6 期（11 月 2004 635 646 10 1017 0956796804005143），–。[5] https://doi.org/。/S Arthur I. Baars 和 S. Doaitse Swierstra。 。2004。类型安全的自省代码。载于*2004* 年 *ACM SIGPLAN Haskell* 研讨会论文集（美国犹他州斯诺伯德）（*Haskell '04*）。ACM，第 69–79 页。[6] Bas Basten、Jeroen van den Bos、Mark Hills、Paul Klint、Arnold Lankamp、Bert Lisser、Atze van der Ploeg、Tijs van der Storm 和 Jurgen J. Vinju。2015。Rascal 中的模块化语言实现 - 经验报告。《计算机编程科学》114, C（2015年12月），7–19。https://doi.org/ 10 1016 2015 11 003 。/j.scico. . . [7] Peter T. Breuer 和 Jonathan P. Bowen。1995年。一种更优雅的编译器生成器：在 C 中生成高阶解析器。《Software: Practice and Experience》25, 11（1995年），1263–1297。[8] Robert D. Cameron。1993年。扩展上下文无关文法的排列短语。《ACM Letters on Programming Languages and Systems》2, 1-4（1993年3月），85–94。https://doi.org/ 10 1145 176454 176490 。/ 。Dominique Devriese 和 Frank Piessens。2011 年。显式递归文法组合子。载于第13届声明式语言实用方面国际研讨会论文集。Springer Berlin Heidelberg，84–98。[10] Jay Earley。1970年。一种高效的上下文无关解析算法。《Commun. ACM》13, 2（1970年2月），94–102。[11] Michael J. Fischer。1968年。带有类宏产生式的文法。载于第9届年度开关与自动机理论研讨会（SWAT *1968* 131 142 10 1109 1968 12 ）。– 。[12] https://doi.org/ 。/SWAT. 。Richard A. Frost、Rahmatullah Hafiz 和 Paul Callaghan。2008 年。用于二义性左递归文法的解析器组合子。载于《声明式语言实用方面》。Springer Berlin Heidelberg，167–181。[13] Richard A. Frost 和 Barbara Szydlowski。1996年。纯函数式自顶向下回溯语言处理器的记忆化。《计算机编程科学》27, 3（1996年），263–288。[14] Dick Grune 和 Ceriel Jacobs。2010年。《解析技术：实用指南》（第2版）。Springer Publishing Company, Incorporated。[15] Anastasia Izmaylova、Ali Afroozeh 和 Tijs van der Storm。2016年。实用的通用解析器组合子。载于2016年 ACM SIGPLAN 部分求值与程序操纵研讨会论文集（美国佛罗里达州圣彼得堡）（PEPM 2016）。ACM，1–12。[16] Mark Johnson。1995年。自顶向下解析中的记忆化。《计算语言学》21, 3（1995年），405–417。[17] Lennart C. L. Kats 和 Eelco Visser。2010年。Spoofax 语言工作台：语言与集成开发环境的声明式规范规则。载于国际面向对象编程系统、语言与应用会议（OOPSLA 2010）。ACM，444–463。https://doi.org/ 10 1145 1869459 1869497 。/ 。P. Klint、T. v. d. Storm 和 J. Vinju。 。2009。Rascal：一种用于源代码分析与操纵的领域特定语言。载于2009年第九届 IEEE 源代码分析与操纵国际工作会议。168–177。[19] Donald E. Knuth。1968年。上下文无关语言的语义。《Mathematical Systems Theory》2, 2（1968年），127–145。https://doi.org/ 10 1007 01692511 。/BF Peter Ljunglöf。 。2002。纯函数式解析。博士学位论文。查尔姆斯理工大学和哥德堡大学。[21] Simon Marlow 和 Andy Gill。2001年。Happy—— Haskell 解析器生成器。https://www.haskell.org/happy/。 [在线，访问于 2020]年4月28日。[22] Peter Norvig。1991年。自动记忆化技术及其在上下文无关解析中的应用。《计算语言学》17, 1（1991年），91–98。[23] Chris Okasaki 和 Andrew Gill。1998年。快速可合并整数映射。载于 ML 研讨会。77–86。[24] Francois Pottier 和 Yann Régis-Gianas。2020年。《Menhir 参考手册》。http://gallium.inria.fr/~fpottier/menhir/manual.pdf。 [在线，访问于 2020]年5月13日。[25] Tom Ridge。2014年。使用预言机为所有上下文无关文法实现简单、高效、健全且完备的组合子解析。载于《软件语言工程》。Springer International Publishing，261–281。[26] João Saraiva。1999年。属性文法的纯函数式实现：Zuiver Functionele Implementatie Van Attributengrammatica's。IPA。http://books.google.nl/books?id=yFqTAAAACAAJ

[27] Elizabeth Scott 和 Adrian Johnstone。2010年。GLL 解析。《理论计算机科学电子笔记》253, 7（2010年），177–189。第九届语言描述、工具与应用研讨会（LDTA 2009）论文集。[28] Elizabeth Scott 和 Adrian Johnstone。2013年。GLL 解析树生成。《计算机编程科学》78, 10（2013年），1828–1844。[29] Elizabeth Scott 和 Adrian Johnstone。2016年。面向性能的 GLL 解析算法结构化。《计算机编程科学》125（2016年），1–22。[30] Elizabeth Scott、Adrian Johnstone 和 Rob Economopoulos。2007年。BRNGLR：一种三次时间复杂度的 Tomita 风格 GLR 解析算法。《Acta Informatica》44, 6（2007年），427–461。[31] Elizabeth Scott、Adrian Johnstone 和 L. Thomas van Binsbergen。2019年。使用二叉子树集表示推导。《计算机编程科学》175（2019年），63–84。https://doi.org/10 1016 2019 01 008 。/j.scico. . . S. Doaitse [32] Swierstra。2009。组合子解析：简明教程。载于《语言工程与严谨软件开发》。Springer Berlin Heidelberg，252–300。[33] S. Doaitse Swierstra、Pablo R. Azero Alcocer 和 João Saraiva。1999年。组合子语言的设计与实现。载于《高级函数式编程》。Springer Berlin Heidelberg，150–206。[34] Peter Thiemann 和 Matthias Neubauer。2004年。参数化 LR 解析。《理论计算机科学电子笔记》110（2004年），115 – 132. https://doi.org/10.1016/j.entcs.2004.06.007 第四届语言描述、工具与应用研讨会论文集（LDTA 2004）。[35] Peter Thiemann 和 Matthias Neubauer。2008。上下文无关文法宏。载于第十届 *ACM SIGPLAN* 声明式编程原理与实践国际会议论文集（西班牙瓦伦西亚）（*PPDP 2008*）。ACM，120–130。https://doi.org/10 1145 1389449 1389465 。/。[36] Masaru Tomita。1985。自然语言高效解析：面向实用系统的快速算法。Kluwer Academic Publishers。[37] L. Thomas van Binsbergen。2019。使用可复用组件的可执行编程语言形式化规范。博士论文。伦敦大学皇家霍洛威学院。[38] L. Thomas van Binsbergen、Elizabeth Scott 和 Adrian Johnstone。2018。使用灵活组合子进行 GLL 解析。载于第十一届 *ACM SIGPLAN* 软件语言工程国际会议论文集（*SLE 2018*）。[39] L. Thomas van Binsbergen、Elizabeth Scott 和 Adrian Johnstone。2020。纯函数式 GLL 解析。《计算机语言学报》（2020）。[40] Mark G. J. van den Brand、J. Heering、P. Klint 和 P. A. Olivier。2002。编译语言定义：ASF+SDF 编译器。*ACM* 编程语言与系统汇刊24，4（2002），334–368。