Реферат

Пояснительная записка содержит 17 страниц (из них 2 страницы приложений). Количество использованных источников – 12. Количество приложений – 1.

Ключевые слова: парсер-комбинаторы, Packrat парсеры, бестиповое лямбда-исчисление, функциональное программирование, категориальная абстрактна машина, категориальная комбинаторная логика.

Целью данной практики является исследование и реализация интеграции системы компиляции выражений лямбда-исчисления в выражения категориальной комбинаторной логики в КАМ-машину на языке Scala.

В первой главе проводится обзор и анализ парсинг комбинаторов и предствление строк в структуре лямбда-термов

Во второй главе описываются парсинг комбинаторы для представления лямбда-термов.

В третьей главе приводится описание объектной модели системы для парсинга выражений лямбда-исчисления и передачи их на модуль компиляции в выражения категориальной комбинаторной логики.

В четвертой главе приводится описание программной реализации и экспериментальной проверки интеграции системы преобразования выражений бестипового лямбда-исчисления в выражения категориальной комбинаторной логики в КАМ-машину.

В приложении А вынесены полные листинги отдельных модулей.

Содержание

Введение

Большинство языков функционального программирования основаны на одной и той же математической модели - лямбда-исчислении. При компиляции кода, написанного на функциональном языке, он преобразуется в лямбда терм, который нужно вычислить. Одним из способов вычисления лямбда-термов является метод рефокусировки, который позволяет значительно ускорить процесс вычисления. Такой метод используется в категориальной абстрактной машине, она реализует жадные вычисления и лежит в основе функционального языка Caml. На данный момент на кафедре №22 реализована рекурсивная модификация среды категориальной абстрактной машины на языке Scala. В рамках учебной исследовательской работе был реализован модуль перевода выражений лямбда-исчисления в выражения категориальной комбинаторной логики на языке Scala.

В рамках учебной практики проводиться работа по интеграции модуля компиляции термов бестипового лямбда-исчисления в термы категориальной комбинаторной логики. Для этой задачи очень актуален функциональный подход к парсингу текста на основе комбинаторов. Этот подход отличается краткостью кода и простотой понимания и написания кода для парсинга текста в нужные структуры, в данном случае в лямбда-термы. Это достигается благодаря тому, что комбинаторы легко могут образовывать из нескольких парсер один, а также существуют парсеры, которые хранят в себе информацию о вызовах, что позволяет корректно работать с рекурсивно вложенными структурами при парсинге и не приводит к переполнению стека вызова или некорректному парсингу текста.

Также немало важно интегрировать систему в веб-приложение для предоставления пользовательского интерфейса, который отсутствует в простом модуле компиляции. Для этого будет использоваться удобный фреймворк Play написанный на языке Scala.

Анализ проблематики содержит в себе исследование методов парсинга бестипового лямбда исчисления для аппликативных вычислительных систем с целью выявления получаемых преимуществ и недостатков. Далее проведен анализ структуры получаемого от компилятора КАМ-кода для интеграции в уже реализованной КАМ.

В теоретической части выполнена разработка синтаксического анализатора для выражений бестипового лямбда-исчисления для представления в структуре лямбда-термов.

В инженерном разделе будет сделано проектирование объектной модели модуля парсинга лямбда-термов из входной строки в структуры бестипового лямбда исчисления.

Далее будет практический раздел, где выполнена программная реализация спроектированной системы и выполнена проверка полученной реализации на составленных примерах.

1. Изучение парсинг комбинаторов

В этом разделе описываются модели парсинга лямбда-выражений, выбирается наиболее подходящий для данной работы способ. Выбор формальной модели аргументируется. Выводятся цели и задачи практики.

1.1 Общий анализ парсинга лямбда-исчисления

Лямбда исчисление представляет собой средство для описания математических функций и их обработки с помощью специальных правил. Лямбда-выражения преобразуются, упрощаются на основе процесса редукции, результатом которого является нормальная форма выражения, когда нельзя применить никакое правило редукции. Отсюда следует единственность нормальной формы, а также идентичность результатов вычислений при любом порядке редукций.

В системе, основанной на лямбда исчислении, для построения новых функций используется лямбда-выражение с множеством правил подстановок для переменных. Лямбда-выражение в состоянии описать все возможные вычислимые функции с любым количеством аргументов.

В этой системе всё вычисления сводятся к двум элементарным операциям - определению функции и её применению. Основное достоинство этого исчислении – это то, что его одновременно можно рассматривать как простой язык программирования, на котором можно описывать вычисления, а также как математический объект, о котором можно доказывать строгие утверждения [1].

В данной практике поставлена задача анализа произвольного текста для данных. Как правило, можно использовать регулярные выражения или кодировать исходя из предположения о формате данных о том, как вы анализируете текст (обрезать строку по определенным индексам, разделить по запятой и т.д.). Оба способа не надежные и требуют большого количества подробного кода для правильной обработки всех возможных исключений. Это может привести к написанию собственного анализатора, но это непосильная задача для большинства разработчиков. Тогда нужно научиться писать парсер или изучать генераторы парсеров, чтобы сделать даже простой синтаксический анализатор. Однако Scala имеет решение этой проблемы, и это решение представляет собой комбинаторы парсеров.[2]

Парсер - часть программы, которая строит более сложные структуры данных из линейной последовательности простых данных (символов, лексем, байтов) с учетом некоторой грамматики, неявно содержащиеся в исходной последовательности. Это может быть разбор конфи-

гурационных файлов, разбор исходного кода на каком-либо языке программирования, разбор проблемно-ориентированных языков (DSL), чтение сложных и не очень форматов данных (XML, PostScript), разбор запросов и ответов сетевых протоколов, вроде HTTP, и т. п. Здесь и далее слова «парсинг» и «разбор» считаются синонимами.

Комбинаторы — это функции высшего порядка, которые из одних функций строят другие. Возможность принимать функции в качестве аргумента и возвращать их в качестве результата — отличительная черта функциональных языков программирования, в которых комбинаторы являются обычными функциями.

Парсер-комбинаторы — это широко известная в узких кругах техника создания парсеров, использующая возможности функциональных языков программирования. [3]

Функциональные языки позволяют построить парсер динамически, используя простейшие функции и комбинаторы для синтеза по правилам грамматики сложных парсеров из простых. При этом и сама грамматика, и семантические действия (выполняющиеся при успешном разборе того или иного элемента грамматики) формулируется на одном языке, а парсер-комбинаторы выступают в качестве DSL (встроенного предметно-ориентированного языка).

1.2 Анализ программных средств для парсинга лямбда-термов

Для данной задачи был выбран язык программирования Scala. Сочетающий возможности функционального и объектно-ориентированного программирования, что как раз подходит для решаемой задачи, так как вычисляемый лямбда-терм можно представить в виде некоторого объекта ($caseclass$-а),[4] а перевод из строки с кодом в лямбда-термы можно сделать при помощи сопоставления с образцом, которое есть в функциональном программировании[5] и достаточно лаконично и удобно реализуется на языке программирования Scala. Также реализация категориальной абстрактной машины, в которой будет использован мой модуль, написан на Scala, что упростит интеграцию со сторонней системой.

Основные особенности для парсинга в языке Scala:

- В Scala разрешено давать методам замечательные названия типа "~", "~ > ", " < ~", "|", "". Комбинация парсеров p и q записывается как p q, а возможность выбрать один из них: p|q. Читается намного лучше, чем p.andThen(q) или p.or(q)
- Благодаря неявным преобразованиям (implicits) и строчка "abc" и регулярное выражение "$[0-9]+$".r при необходимости превращаются в парсеры.
- В языке мощная статическая система типов, которая позволяет ловить ошибки сразу.
- Полная поддержка всех возможностей функционального программирования.

1.3 Выводы

Исследовалась область парсинга бестипового лямбда исчисления. Установлено, что самым простым способом написания синтаксического анализатора является применение парсинг комбинаторов, которые обладают краткостью и понятностью для написания анализатора. Выведены цели и задачи курсовой работы.

- Рассмотрены модели представления лямбда-выражений и подход к построению синтаксического анализатора для преобразования текста в лямбда-термы.
- Были проанализированы варианты реализации программных способов парсинга. С учетом наличия в функциональных языках мощной концепции парсинг-комбинаторов, то для простоты и удобства разработки синтаксического анализатора выбран именном этот инструмент парсинга текста.
- Для реализации программной системы был выбран язык Scala, так как он может работать в функционально-ориентированной парадигме и с его помощью просто использовать парсинг комбинаторы для синтаксического анализатора лямбда-исчисления и более просто интегрироваться с существующей КАМ на Scala.

1.4 Постановка задачи учебно-исследовательской практики

Задачей данной практики является исследование и реализация интеграции системы компиляции выражений лямбда-исчисления в выражения категориальной комбинаторной логики в КАМ-машину на языке Scala.

- Изучить подходы к парсингу выражений бестипового лямбда-исчисления.
- Разработать формальную модель парсинга лямбда-исчисления в лямбда-термы.
- Спроектировать объектную модель системы для проведения парсинга на разработанном модуле преобразования.
- Реализовать модуль парсинга для передачи текстового кода в модуль преобразования в термы ККЛ.
- Выполнить проверку полученной реализации на составленных примерах.

2. Разработка модели синтаксического анализатора лямбда-исчисления

В данной главе представлены основные компоненты реализуемого синтаксического анализатора, его основные свойства и задачи, а также алгоритмы по которым он работает.

2.1 Общая модель лямбда-термов

Основой лямбда-исчисления служит формальное понятие лямбда-термов, которые строятся из переменных и некоторого фиксированного множества констант при помощи операций применения (аппликации) функций(бета-редукция) и лямбда абстракции. Это значит, что всевозможные лямбда-термы разбиваются на четыре класса[6]:

а) Константы: количество констант определяется конкретной лямбда-нотацией, иногда их нет вовсе. Обычной их обозначают алфавитно-цифровыми строками. В данной работе этот класс лямбда-терма не будет использоваться.

б) Переменные: обозначаются произвольными алфавитно-цифровыми строками; как правило, используют в качестве имён буквы, расположенные ближе к концу латинского алфавита, например, x, y и z.

в) Апликация: применение функции s к аргументу t, где s и t представляют собой произвольные термы. Будем обозначать комбинации как st.

г) Абстракция произвольного λ-терма s по переменной x (которая может как входить свободно в s, так и нет) имеет вид λx.s.

Подобно языкам программирования, синтаксис лямбда-термов может быть задан при помощи БНФ (форм Бэкуса-Наура):

$$Exp = Var|Const|ExpExp|\lambda Var.Exp$$

после чего мы можем трактовать термы, как это принято в теории формальных языков, не просто как цепочки символов, а как абстрактные синтаксические деревья. Это значит, что соглашения наподобие левоассоциативности операции применения функции либо интерпретации λx y.s как λx.λy.s, а также неразличимость переменных и констант по именам не являются неотъемлемой частью формализма лямбда-исчисления, а имеют смысл исключительно в момент преобразования терма в форму, подходящую для восприятия человеком, либо в обратном направлении[7].

2.2 Модель синтаксического анализатора для парсинга лямбда-термов

Исходный код для языка программирования обычно разбирается в два этапа:

1. лексический анализатор (лексер) берет символьный поток и генерирует поток токенов
2. синтаксический анализатор берет поток токенов и преобразует их в AST

У классических парсер-комбинаторов есть одна проблема — экспоненциальная сложность по отношению к грамматике. В простых случаях этого эффекта не заметно, но когда грамматика становится достаточно сложной и ветвистой (например, при разборе языка программирования), то скорость падает очень сильно. Дело в том, что схема работы парсер-комбинаторов есть рекурсивный спуск с откатами. Допустим, мы разбираем некий язык программирования с арифметическими операциями над числами, переменными и элементами массивов. Тогда, чтобы разобрать выражение «42», парсер будет перебирать варианты: исходная строка-выражение — это или а1) сумма подвыражений, или б1) разность подвыражений, или в1) просто подвыражение, где каждое подвыражение есть или а2) произведение атомов, или б2) отношение атомов, или в2) деление с остатком атомов, или г2) просто атом, где каждый атом есть или а3) число, или б3) переменная, или в3) элемент массива, где индекс — тоже выражение. Парсер сначала будет пытаться разобрать выражение как сумму, но не найдя знака ’+’, начнет пытаться разобрать выражение как разность, и не найдя знака ’-’, будет разбирать выражение как просто подвыражение. При этом подвыражение будет разобрано трижды, каждый раз по очереди пытаясь интерпретировать его как произведение, как отношение и т. д. Количество перебираемых подвариантов перемножается, а если наше исходное выражение — часть более сложного, то оно тоже будет разобрано много раз, соответственно числу вариантов ветвей грамматики, включающей выражение, т. е. умножаем еще, отсюда и экспонента.

Основная проблема тут — что одна и та же работа делается многократно, при каждом откате результат разбора теряется. Логичным решением этой проблемы является мемоизация результатов. Техника подобного разбора с мемоизацией всех результатов носит имя Packrat[8], что дословно означает «земляная крыса», но также имеет переносный смысл «человек, привыкший ничего не выбрасывать и хранить все подряд, даже ненужное». На практике нет необходимости мемоизировать абсолютно все — достаточно мемоизации нескольких самых ветвистых (с большим количеством альтернатив) правил грамматики. Для мемоизации результатов парсинга отлично подходит механизм ленивых вычислений.

2.3 Выводы

В данной главе был представлен синтаксис и абстрактные структуры для представления лямбда-термов. Рассмотрены классы лямбда термов и их синтаксис и представление выражений в БНФ.

Сделан выбор парсер-комбинаторов, с учетом недостатков классических парсер-комбинаторов были выбраны Packrat парсеры, которые при помощи ленивых вычислений и техники мемоизации позволяют ускорить парсинг и избавиться от проблем при рекурсивном анализе вложенных структур.

3. Результаты проектирования синтаксического анализатора

В данной главе предоставлена информация об архитектуре реализуемого синтаксического анализатора и кода лямбда-термов, его компонентах, связях между ними и основными функциями для дальнейшей разработки на языке Scala

3.1 Общая архитектура системы

Данный вычислитель будет работать как модуль в веб-приложении КАМ машины, которая уже реализована на кафедре на языке Scala с MVC фреймворком Play. Сам синтаксический анализатор будет состоять из одного модулей. LambdaParser.scala - парсинг текста лямбда-выражений в структуры лямбда-термов. Далее нужно будет реализовать контроллер, который будет отображать интерфейс для ввода текста и вывода результата компиляции.

3.2 Архитектурная модель синтаксического анализатора кода лямбда-термов

Для описания синтаксиса бестипового лямбда-исчисления нам необходимо использовать Packrat парсинг-комбинаторы с использованием регулярных выражений. Для этого нужно будет реализовать ленивые функции для каждого типа лямбда-термов, написать свой лексический анализатор на основе стандартного StdLexical в котором будет выделен особый символ лямбда, который для удобства мы заменил алиасом косой чертой. Такой алиас применяют в языке Haskell. Также необходимо будет реализовать методы parse и parsePhrase, которые будут из потока символов возвращать ParseResult в котором будет содержаться лямбда-терм полученный из текста.

3.3 Выводы

Была спроектирована архитектурная модель синтаксического анализатора кода лямбда-исчисления в лямбда-термы для передачи в модуль компиляции в терм ККЛ. Для этого спроектированы были:

– Спроектирован синтаксический анализатор кода лямбда-исчисления

4. Реализация и экспериментальная проверка работоспособности модуля преобразования лямбда-термов в ККЛ

В этом разделе представлены результаты разработки синтаксического анализатора структур лямбда-исчисления, описаны основные классы. Также здесь представлены результаты тестирования реализованного модуля в составе приложения.

4.1 Выбор инструментальных средств

В качестве языка реализации был выбран язык Scala. Данный язык является мультипарадигменным и нацелен в основном на написание программ в функциональном стиле и, как и большинство функциональных языков имеет возможность для так называемого сравнения с образцом[9]. Благодаря этой возможности языка, написание функций для вычисления и типизации входного вычисления является более простой задачей, которая сводится по сути к переписыванию формальный правил перехода. Язык обладает мощной системой для парсинга входных выражений в виде строки и представления их в виде дерева в памяти. Это свойство делает языка Scala ещё более привлекательным для его использования с целью реализации простого интерпретатора для языка программирования. Статическая типизация языка уменьшает количество потенциальных ошибок на этапе проверки типов структур при паттерн матчинге и анализа текста парсер-комбинаторами.

4.2 Программная реализация синтаксического анализатора и описание основных классов

Синтаксический анализатор содержит определения типов данных, при помощи которых в памяти можно представить термы, которые есть в бестиповом лямбда-исчислении. Подробнее взглянем на эти функции, которые близки к нотации БНФ.

```
lazy val lambdaExpression: PackratParser[Term] = application | otherExpressionTypes
lazy val otherExpressionTypes = variable |  parens | lambda
```

- любое лямбда-выражение можно представить в виде аппликации и любого другого лямбда-терма, а именно переменных, лямбда-абстракций либо выражение в скобках.

В библиотеке комбинаторов синтаксического анализатора Scala уже есть лексер (StdLexical), который мы можем использовать для создания токенов, таких как идентификаторы и ключе-

вые слова, разделенных пробелами (которые игнорируются в производимом потоке токенов). Чтобы использовать его сообщаем лексеру, какие символы являются разделителями.

```
type Tokens = StdLexical
val lexical = new CustomLambdaLexer
lexical.delimiters ++= Seq("\\", ".", "(", ")")
```

Для того чтобы ключевой символ лямбда-абстракции, который мы заменили символом косой черты ”

” и другие разделители лексический анализатор не путал с произвольным символьным литералом необходимо реализовать свой лексер расширяя StdLexical:

```
class CustomLambdaLexer extends StdLexical {
  override def letter = elem("letter", c => c.isLetter &&
    c != '(' && c != ')' && c != '\\' && c != '.')
}
```

Далее реализуем сами парсеры термов через PackratParser, где символ ˜ является методом, который из нескольких парсеров создает композицию нового парсера, а ^^ передают спарсенный токен для преобразования в конкретный кейс класс выражающий конкретный лямбда-терм.

```
lazy val lambda: PackratParser[Lambda] =
    positioned(("\\") ~> variable ~ "." ~ parens ^^ {
      case arg ~ "." ~ body  => {
        Lambda(arg, body)
      }
    }
    )
  lazy val application: PackratParser[App] =
    positioned(lambdaExpression ~ otherExpressionTypes ^^ {
      case left ~ right => {
        App(left, right)
      }
    }
    )
  lazy val variable: PackratParser[Var] =
    positioned("""[A-Za-z]""".r ^^ ((x: String) => {
      Var(Literal(x.charAt(0).toString))
    }))

   lazy val parens: PackratParser[Term]       = "(" ~> lambdaExpression <~ ")" | lambdaExpr
```

Наконец реализованы функции parse. Результат парсинга — либо преобразование в структуры AST, либо сообщение об ошибке, для этого используется pattern matching. Полный код модуля представлен в приложении А.

```
def parse(s: CharSequence): Term = {
    parse(new CharSequenceReader(s))
  }
```

```
def parse(input: CharSequenceReader): Term = {
  parsePhrase(input) match {
    case Success(t, _) => t
    case NoSuccess(msg, next) => throw new IllegalArgumentException(
      "Could not parse '" + input + "' near '" + next.pos.longString + ": " + msg)
  }
}

def parsePhrase(input: CharSequenceReader): ParseResult[Term] = {
  phrase(lambdaExpression)(input)
}
```

4.3 Основной сценарий работы пользователя с реализуемым модулем

В данной учебной практике удалось реализовать интерфейс для лямбда-выражений. Можно записать код выражений бестипового лямбда-исчисления используя форму ввода, данный в веб-интерфейсе, записать выражение. Далее можно применить метод преобразования в терм ККЛ. В результате пользователь получает запись лямбда выражений в синтаксисе категориальной комбинаторной логики, которые практически являются инструкциями для КАМ-машины, чтобы произвести непосредственное вычисление самого выражения.

Все эти действия можно делать через веб-интерфейс КАМ машины.

4.4 Разработка тестовых примеров

Тестовые примеры можно воспроизвести, запустив приложение через сборщик проекта sbt и в нем запустив команду test, которая запускает все тестовые примеры, которые демонстрируют работоспособность модулей. Для этого был использован модуль specs2 для написаний декларативных спецификаций, которые очень просты в понимании обычным пользователем.

4.5 Сравнение реализованного синтаксического анализатора с существующими аналогами

Данный анализатор имеет немало аналогов, но есть его менее эффективные аналоги на других языках программирования, либо с использованием классических парсер-комбинаторов, которые не эффективны на большом количестве кода. В данном анализаторе используются Packrat парсинг-комбинаторы, которые позволяют сохранить простоту парсинг комбинаторов и получить высокую эффективность в отличие от метода левой факторизации, который эффективен, но снижает простоту написания и понимания кода синтаксического анализатора.

4.6 Выводы

В результате программной реализации модуля был произведен и обоснован выбор программного средства, а именно функционального языка программирования Scala для интегра-

ции с существующей расширенной категориальной абстрактной машиной на кафедре ”Кибернетика”, которая написана на Scala и веб-фрейсворке Play.

В ходе реализации был реализован модуль синтаксического анализатора при помощи Packrat парсинг-комбинаторов и разработан контроллер в веб-приложении, который позволяет через веб-интерфейс вводить лямбда-выражения и получать их интерпретацию в виде термов ККЛ.

Далее были разработаны основные сценарии использования через веб-интерфейс КАМ-машины. Также были разработаны и описаны тестовые примеры того как работают основные модули приложения.

Далее произведено сравнение с существующими аналогами, где описано что есть немало аналогов такого синтаксического анализатора, но был выбран самый оптимальный вариант как по эффективности, так и по простоте понимания и реализации.

Заключение

В ходе учебно-исследовательской практики были получены следующие результаты:

- Была изучена проблемная область, а именно синтаксические анализаторы бестипового лямбда-исчисления, парсинг комбинаторы;
- Был разработан синтаксис и абстрактные структуры для парсинга лямбда-термов и сделан выбор типа парсер-комбинаторов, а именно эффективные для рекурсивных структур Packrat парсеры;
- Была спроектирована архитектурные части модуля парсинга текстового кода лямбда-исчисления лямбда-термов. Были описаны какие методы и классы необходимо будет реализовать и какие между ними будут взаимосвязи;
- Был программно реализован модуль синтаксического анализатора для системы компиляции выражений бестипового лямбда-исчисления в выражения категориальной комбинаторной логики и контроллер с формой для ввода и отображения термов. Был обоснован технологический стек программных средств, а именно языка программирования Scala и интеграция в структуру MVC-фреймворка Play. Были описаны основные сценарии использования и тестовые примеры, проведено сравнение с аналогичным ПО и сделаны выводы по поводу дальнейшей доработки модуля в соответствии с полученными результатами учебно-исследовательской практики;

Список литературы

1. Пирс Бенджамин. Типы в языках программирования // М.: Лямбда Пресс Добросвет. — 2012. — P. 656.
2. Odersky Martin, Spoon Lex, Venners Bill. Programming in scala. — Artima Inc, 2008.
3. Swierstra S Doaitse. Combinator parsers: From toys to tools // Electronic Notes in Theoretical Computer Science. — 2001. — Vol. 41, no. 1. — P. 38–59.
4. Леви Жан-Жак, Довек Жиль. Введение в теорию языков программирования. — Litres, 2017.
5. А. Филд, П. Харрисон. Функциональное программирование. — Мир, 1993.
6. Харрисон Джон. Введение в функциональное программирование. — 2018.
7. Danvy Olivier. Back to Direct Style // Sci. Comput. Program. — 1994. — Vol. 22. — P. 183–195.
8. Ford B. Packrat parsing: a practical linear-time algorithm with backtracking (Master's thesis) // Massachusetts Institute of Technology. — 2002.
9. Peyton Jones Simon L. The implementation of functional programming languages (prentice-hall international series in computer science). — Prentice-Hall, Inc., 1987.
10. Cousineau Guy, Curien P-L, Mauny Michel. The categorical abstract machine // Science of computer programming. — 1987. — Vol. 8, no. 2. — P. 173–202.
11. В.Э. Вольфенгаген. Категориальная абстрактная машина. Конспект лекций: введение в вычис- ления. — 2-е изд. изд. — М. : АО «Центр ЮрИнфоР», 2002.
12. В.Э. Вольфенгаген. Аппликативные Вычислительные Системы. — Москва : Труды Институт «ЮрИнфоР-МГУ», 2008. — 5.

A. Листинг модулей

```
package LambdaToCCL


import scala.util.parsing.combinator.lexical.StdLexical
import scala.util.parsing.combinator.{PackratParsers, RegexParsers}
import scala.util.parsing.input.CharSequenceReader

class LambdaParser extends RegexParsers with PackratParsers {
  type Tokens = StdLexical
  val lexical = new CustomLambdaLexer
  lexical.delimiters ++= Seq("\\", ".", "(", ")")

  lazy val lambdaExpression: PackratParser[Term] = application |
   otherExpressionTypes
  lazy val otherExpressionTypes = variable |  parens | lambda
  lazy val lambda: PackratParser[Lambda] =
    positioned(("\\") ~> variable ~ "." ~ parens ^^ {
      case arg ~ "." ~ body  => {
        Lambda(arg, body)
      }
    }
    )
  lazy val application: PackratParser[App] =
    positioned(lambdaExpression ~ otherExpressionTypes ^^ {
      case left ~ right => {
        App(left, right)
      }
    }
    )
  lazy val variable: PackratParser[Var] =
    positioned("""[A-Za-z]""".r ^^ ((x: String) => {
      Var(Literal(x.charAt(0).toString))
    }))
  lazy val parens: PackratParser[Term]       = "(" ~> lambdaExpression <~ ")" |
    lambdaExpression
}

class CustomLambdaLexer extends StdLexical {
  override def letter = elem("letter", c => c.isLetter &&
    c != '(' && c != ')' && c != '\\' && c != '.')
}

object SimpleLambdaParser extends LambdaParser {
  def parse(s: CharSequence): Term = {
    parse(new CharSequenceReader(s))
  }

  def parse(input: CharSequenceReader): Term = {
    parsePhrase(input) match {
      case Success(t, _) => t
```

```
      case NoSuccess(msg, next) => throw new IllegalArgumentException(
        "Could not parse '" + input + "' near '" + next.pos.longString + ": " + msg)
    }
  }

  def parsePhrase(input: CharSequenceReader): ParseResult[Term] = {
    phrase(lambdaExpression)(input)
  }
}
```