Final Project

CS 421, Summer 2022

Sean Enright

August 5, 2022

Overview

I have implemented a packrat parser for a simple imperative programming language in Haskell, based on Bryan Ford's 2002 paper, "Packrat parsing: simple, powerful, lazy, linear time"[1].

Packrat parsing is a method of implementing parsers that run in linear time with backtracking and unlimited lookahead for LL(n) or LR(n) grammars, among others. Its backtracking and lookahead allow a few difficult parsing problems to be handled with relative ease, including rules for the longest match, followed-by and not-followed by. It also allows for simple integration of lexical analysis, so packrat parsers typically do not have separate lexing and parsing stages, but rather interleave them.

These benefits come at the expense of memory consumption, which is significantly higher in packrat parsers than typical parsers for LL(n) and LR(n) grammars. This limitation prevented this style of parsing, first described in the 1970s, from gaining prominence until the early 2000s, when hardware and optimizing compilers made these downsides an acceptable trade-off for certain applications.

Ford describes a packrat parser implementation through a grammar for a trivial language describing basic arithmetic expressions. I have extended this grammar to express a variant of the IMP language[2], following Ford's outline. This language adds Boolean expressions and command statements to demonstrate the application of packrat parsing to different data types and control structures.

My implementation uses monadic combinators for clarity and to reduce the amount of code.

A REPL is provided to allow a user interface to the parser.

Implementation

Major Tasks and Capabilities

Capabilities of the Parser

The key concept behind packrat parsing is the use of a specialized data structure, consisting of a matrix of possible results for each of the parse functions in the grammar for each of the input characters. The results of parsing each character of input are memoized, potentially allowing for greater efficiency in accessing repeated patterns. For an input string of length n, and p parsing functions, this matrix will have $(n + 1) * p$ cells, with the $n + 1$ term accounting for each character of the string as well as an empty string. This is potentially a very large table, so in order to efficiently make use of it, this parsing style effectively requires the use of a lazily

evaluated language. For this reason, and for the potential to implement it via a monadic approach, Haskell was chosen for this project.

Parsing functions are also included for lexical analysis, including matching whitespace of varying length, matching sets of operators and keywords, and of matching numerical characters and forming them into possibly signed integers.

Grammar

To demonstrate a minimal implementation of a packrat parser, Ford introduces a trivial language described by the following grammar. This grammar uses non-terminals of different levels of the parsing chain to enforce associativity. The non-terminal on the right side of each operator is higher in the chain and therefore creates right-associativity.

```
Additive  <- Multitive ‘+’ Additive | Multitive
Multitive <- Primary ‘*’ Multitive | Primary
Primary   <- ‘(’ Additive ‘)’ | Decimal
Decimal   <- ‘0’ | ... | ‘9’
```

This language was extended to closely approximate the IMP language[2], with a few minor modifications made to reduce ambiguity and reduce the amount of symbols for the sake of demonstration.

In Backus-Nauer form, IMP grammar is described as:

```
Arithmetic Expressions
a ::= "n" | "X" | a0 "+" a1 | a0 "-" a1 | a0 "*" a1

Boolean Expressions
b ::= "true" | "false" | a0 "=" a1 | a0 "<=" a1 | "!"b | b0 "and" b1 | b0 "or" b1

Commands
c ::= "skip" | X ":=" a | c0 ";" c1 | "if" b "then" c0 "else" c1 | "do" c "while"
b
```

This grammar was slightly modified to include the right associativity in arithmetic expressions, modify non-terminals to reduce ambiguity, reduce similar symbols, and to remove assignment.

```
Arithmetic Expressions
a ::= m "+" a | m "-" a | m
m := p "*" m | p "/" m | p "%" m | p
p := "(" a ")" | i
i ::= "-" d | d
d ::= "0" | ... | "9"

Boolean Expressions
b ::= "true" | "false" | a0 "<" a1 | a0 ">" a1
```

```
Commands
c ::= "if" b "then" c0 "else" c1 "fi" | "do" c "while" b "od" | "seq" c0 ";" c1
"qes" | "skip"
```

Most notably, the grammar was extended to encompass Boolean expressions and commands, increasing the expressive power of the language beyond that of a simple calculator. Commands were also given unique initial and final terminal symbols to reduce parsing errors in nested expressions. The introduction of new syntactic sets (i.e., Boolean expressions and commands) required wrapper data types to allow the parser to handle combinations of different expression types or commands.

The addition of commands to the grammar required some modifications to the lexing functions to be able to capture commands evaluated repeatedly in do-while loops and in sequence. For this purpose functions were introduced to capture a string of input until a particular character or keyword is encountered, and to then parse these strings according to the loop or sequence status.

Components of the Code

Core Data Structures

The key data structures of the parser are found in Core.hs

A column of the specialized parsing matrix described above is represented by the algebraic data type Derivs, which uses record syntax to create accessor functions that can be passed into parsing functions. Each field is of type Result, representing the result of an elementary parse.

```
data Derivs = Derivs {
               -- Top level
               dvInputString    :: Result Types,
               -- Commands
               dvIfThenElse     :: Result (),
               dvWhileDo        :: Result (),
               dvSequence       :: Result (),
               dvSkip           :: Result (),
               -- Boolean
               dvBoolVal        :: Result Bool,
               dvRelExpr        :: Result Bool,
               -- Arithmetic
               dvAdditive       :: Result Int,
               dvMultitive      :: Result Int,
               dvPrimary        :: Result Int,
               dvInteger        :: Result Int,
               dvMultipleDigits :: Result (Int, Int),
               dvSingleDigit    :: Result Int,
               -- Lexical analysis
               dvKeyword        :: Result String,
               dvMultipleChars  :: Result [Char],
               dvSymbol         :: Result Char,
               dvWhitespace     :: Result (),
               -- Raw input
```

```
                dvChar :: Result Char
                }
```

The Result data type represents the result of a parsing function, which either is "NoParse", indicating the end of a line or a parsing failure, or "Parsed", which also includes the value returned by the parsing function and an instance of a Derivs data type, which provides a link to the next column of the parsing matrix.

```
data Result v = Parsed v Derivs
              | NoParse
```

These two data structures, Result and Derivs, are mutually recursive, thereby providing a link between columns of the parsing matrix. A parsing function that succeeds yields a Result, which itself contains a Derivs instance linking to the next column. This continues until either a parsing failure occurs or the input string is completely consumed.

The `Parser` data type is constructed with a parsing function that returns a Result and a link to the next Derivs.

```
newtype Parser v = Parser (Derivs -> Result v)
```

Its implementation as a monad follows the approach in Ford's listing, allowing for automatic handling of parsing function failures and convenient "do" notation. The default definitions of the pure, ap and fmap functions are used. The choice operator (<|>) is also provided to allow alternate parsing decisions and for grammars to be expressed succinctly. This results in parsing functions that closely resemble the grammar as written.

The parser is made an instance of the MonadFail class to forcibly terminate parsing functions. This follows the implementation in Ford's listing.

Parsers

The parsing functions are found in Parse.hs

The function `parse` is the main parsing function and the only entry into the key data structure of the packrat parser by way of an input string. It is the most important parsing function of the parser, since it constructs the parsing result matrix and ties together the many parsing functions of the grammar.

It takes an input string and returns a single column of a parsing result matrix, a Derivs instance, and also recursively iterates over the string one character at a time, creating and linking to Derivs instances, until the end of the string is reached and the matrix is terminated.

The Derivs constructor is used to specify the many parsing functions that will be used in the language. Each function has one parameter, which references the encompassing Derivs instance, producing the results that will be passed to subsequent parsing functions. Since Haskell is lazily evaluated, many parsing functions will not actually be referenced in a given column.

This function also specifies the parsing functions for each non-terminal in the grammar, each of which begin with the letter "p".

```
parse :: String -> Derivs
parse s = d where
   d      = Derivs instr
                   ite wd seq skip
                   boolv relex
                   add mult prim int
                   muldig sindig kwd mulchr sym spc chr
   -- Top level
   instr = pInputString d
   -- Commands
   ite    = pIfThenElse d
   wd     = pWhileDo d
   seq    = pSequence d
   skip   = pSkip d
   -- Boolean
   boolv  = pBoolVal d
   relex  = pRelExpr d
   -- Arithmetic
   add    = pAdditive d
   mult   = pMultitive d
   prim   = pPrimary d
   int    = pInteger d
   -- Lexical
   muldig = pMultipleDigits d
   sindig = pSingleDigit d
   kwd    = pKeyword d
   mulchr = pMultipleChars d
   sym    = pSymbol d
   spc    = pWhitespace d
   chr    = case s of
               (c:s') -> Parsed c (parse s')
               []     -> NoParse
```

Below is an example of a typical parsing function. In this case, for comparing two arithmetic expressions and returning a Boolean. Each line of the "do" block corresponds to the parsing of one or many characters, which can correspond to a non-terminal, whitespace, a symbol (an allowed character, such as the operator "+"), or a keyword string.

Note the use of the alternative function for expressing two non-terminals of similar form.

```
pRelExpr :: Derivs -> Result Bool
Parser pRelExpr =
   do vl <- arithExp
      symbol '>'
      vr <- arithExp
      return (vl > vr)
   <|>
```

```
    do vl <- arithExp
       symbol '<'
       vr <- arithExp
       return (vl < vr)
```

Limitations

My implementation of a packrat parser serves to demonstrate and extend Ford's concept, but the implemented language has shortcomings that limit its practicality.

Most importantly, it is a stateless language. This significantly limits its use for general purpose programming. For example, the do-while loop of the language I implemented is not of much use without the conditional Boolean expression being dependent on a variable. Ford describes[1] this limitation and offers[3] a solution, with words of caution, since the introduction of state would require the re-calculation of the entire parsing results matrix at every state change, which potentially eliminates the performance benefits of this method of parser writing.

Another limitation is error handling. An unrecognized string raises and error which causes the REPL and program to terminate.

There are also some limitations in the grammar chosen. It is susceptible to left recursion. A simple solution is to rewrite the grammar, adding suffix non-terminals, as covered in this course. This was omitted to keep the grammar more concise and readable for debugging. The grammar also does not support nested expressions or commands, e.g. `do do skip while false od while false od`.

Status of the Project

Original Proposal

I will implement a parser for a toy language using packrat parsing, as described in the paper listed at the bottom of this message. I will be using Haskell as my language of choice, so I will employ a monadic approach to the parser's implementation.

I will provide a suite of tests that list common use cases for packrat parsing, as well as cases where it fails.

Additionally, to demonstrate the advantages and shortcomings of packrat parsing, I will benchmark its performance against another type of parser, either hand-written, or included in a library, depending on how time consuming it is.

My planned implementation schedule will divide the month of July as follows:

- Week 1: Organize project and begin implementation of parsers.
- Week 2: Finalize implementation of parsers.
- Week 3: Introduce test suite and performance comparison.
- Week 4: Summarize results, package source code and report.

Successfully Implemented Features

A packrat parser was successfully implemented using monadic combinators, and was extended to handle grammar for a simple programming language with control structures and multiple data types, including

arithmetic expressions, Boolean expressions and commands. The lookahead and backtracking capabilities were extended to skip ahead until desired keywords are found, or to collect strings in between keywords.

A testing suite was designed to evaluate strings representing all possible input types, and report of the outcome.

Challenges

The initial challenge in this project was in fully comprehending the level of recursion required to implement the basic data structures and parsing functions. In addition to the main memoization matrix, which is implemented through a pair mutually recursive data structures, the main parsing function itself is doubly recursive, assigning variables that refer to itself, and to later iterations of the parsing results. These are described in detail below.

Another challenge was updating the example code to work with contemporary GHC Haskell. Although only some twenty years have passed since the publication of Ford's paper and thesis[1,3], significant alterations to the source code were necessary to make the examples functional. I encountered GHC standard changes such as the 2014 Functor-Applicative-Monad Proposal and the 2016 MonadFail Proposal, among others.

The last hurdle was adapting the parser beyond the limitation of arithmetic expressions. While extending it to Boolean expressions was without issue, adding commands required several new accessory parsing functions and some creative solutions to introduce control structures for if-then-else commands and do-while loops, without leaving the parsing results matrix. In these cases, my solution was to use the unlimited lookahead capability of the parser to scan for the strings of interest between terminal symbols and handle them accordingly.

For example, in the if-then-else command, if the "if" expression evaluates to true, the "else" command is executed and the parser scans the following string until the terminal "fi" is encountered to exit the parser successfully. If the "if" expression is false, then all characters until "else" are scanned and ignored, and the following command is parsed and evaluated.

```
pIfThenElse :: Derivs -> Result ()
Parser pIfThenElse =
   do keyword "if"
      b <- boolExp
      keyword "then"
      -- Case 1: boolean is true
      when b $ do cif <- command
                  keyword "else"
                  skipUntil "fi"
                  return cif
      -- Case 2: boolean is false
      unless b $ do skipUntil "else"
                    cel <- command
                    keyword "fi"
                    return cel
   <|>
   do Parser pWhileDo
```

In the case of "do-while" loops, the strings corresponding to the "do" and "while" keywords are scanned and passed into a function that parses the strings and evaluates them accordingly until the "while" string returns a false value. This is likely not the most efficient method, since it will repeatedly parsing two strings, which will each create a parsing results matrix.

```
pWhileDo :: Derivs -> Result ()
Parser pWhileDo =
   do keyword "do"
      s_do <- collectUntil "while"
      b_while <- collectUntil "od"
      evalWhile s_do b_while
      return ()
   <|>
   do Parser pSequence
```

Features Not Implemented

In my proposal, I planned to also implement a recursive-descent parser and compare its performance against my packrat parser. I chose to omit the recursive-descent parser to focus more time on extending and documenting the packrat parser. After having implemented a packrat parser, a recursive descent parser could be made by keeping the parsing functions of the packrat parser, but modifying the underlying data structures. `Derivs` would be eliminated along with the mutual recursion between it and `Result`. A more simple `parse` function would be implemented as well, as the lazily-evaluated and recursive structure of the function would no longer be necessary. The overall effect would be to backtrack when failing a parse, rather than lazily traverse through the parsing results matrix.

In the absence of a recursive descent parser with which to compare, I also chose to omit the performance analysis. I identified the GHC `-prof` option and the hp2any libraries as capable tools for memory profiling.

Using the Packrat Parser

Building

This project uses stack for build management.

To build: `stack build`

To run the packrat REPL: `stack run`

To execute the test suite: `stack test`

More details on testing are provided in the Test section below.

REPL

`stack run` will build and start the REPL. The result of the entered expression will be printed on the next line.

Enter `exit` or `quit` to leave the REPL.

```
Staring Packrat REPL...

Packrat> 1 + 2
3
Packrat> 2*(3+4)
14
Packrat> 34<-12
False
Packrat> do skip while 3<1 od
Packrat> exit

...Leaving Packrat REPL
```

Tests

The test suite provided through the `stack test` command provides input strings to the parser and compares the output of the parser with the expected value. Through this method, the tests require the use of the packrat parsing algorithm to process input.

In each test, a string containing a input expression or string is passed into a parser and evaluated, and its output is compared with the desired result. Specialized parsers are used to enter the parsing matrix at different points corresponding to their datatypes, i.e. Int, Bool and () for arithmetic expressions, Boolean expressions and commands, respectively. A top-level parser is also used to parse strings of any type. In the case of commands, if successfully parsed and evaluated, the return value is (), so the parser is effectively testing whether the command was executed without issue.

The tests include:

- Parsing arithmetic expressions with an arithmetic expression parser
- Parsing Boolean expressions with a Boolean expression parser
- Parsing commands with a command parser
- Parsing all three types with varying amount of whitespace
- Parsing all three types with the top-level parser

References

1. Bryan Ford. 2002. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. *SIGPLAN Not.* 37, 9 (September 2002), 36–47. https://doi.org/10.1145/583852.581483

   Available locally: Packrat parsing: simple, powerful, lazy, linear time, functional pearl

2. Glynn Winskel. 2001. *The Formal Semantics of Programming Languages: An Introduction*. 12-24. MIT Press.

3. Bryan Ford. 2002. Packet parsing: a practical linear-time algorithm with backtracking. Master's Thesis. MIT, Cambridge, MA. Retrieved August 2, 2022 from https://dspace.mit.edu/handle/1721.1/87310

Available locally: Packet parsing: a practical linear-time algorithm with backtracking