Parser combinator

In functional programming, a **parser combinator** is a higher-order function which accepts several parsers as input and returns a new parser as its output. In this context, a parser is a function accepting strings as input and returning some structure as output, typically a parse tree or a set of indices representing locations in the string where parsing stopped successfully. Parser combinators enable a recursive descent parsing strategy which facilitates modular piecewise construction and testing. This parsing technique is called **combinatory-parsing**.

Parsers built using combinators are straightforward to construct, ‘readable’, modular, well-structured and easily maintainable. They have been used extensively in the prototyping of compilers and processors for domain-specific languages such as natural language interfaces to databases, where complex and varied semantic actions are closely integrated with syntactic processing. In 1989, Richard Frost and John Launchbury demonstrated[1] use of parser combinators to construct natural language interpreters. Graham Hutton also used higher-order functions for basic parsing in 1992.[2] S.D. Swierstra also exhibited the practical aspects of parser combinators in 2001.[3] In 2008, Frost, Hafiz and Callaghan described[4] a set of parser combinators in Haskell that solve the long standing problem of accommodating left-recursion, and work as a complete top-down parsing tool in polynomial time and space.

Basic idea

In functional programming, parser combinators can be used to combine basic parsers to construct parsers for more complex rules. For example, a production rule of a context-free grammar (CFG) may have one or more ‘alternatives’ and each alternative may consist of a sequence of non-terminal(s) and/or terminal(s), or the alternative may consist of a single non-terminal or terminal or the empty string. If a simple parser is available for each of these alternatives, a parser combinator can be used to combine each of these parsers, returning a new parser which can recognise any or all of the alternatives.

A parser combinator can take the form of an infix operator, used to ‘glue’ different parsers to form a complete rule. Parser combinators thereby enable parsers to be defined in an embedded style, in code which is similar in structure to the rules of the grammar. As such, implementations can be thought of as executable specifications with all of the associated advantages.

The combinators

To keep the discussion relatively straightforward, we discuss parser combinators in terms of *recognizers* only. If the input string is of length `#input` and its members are accessed through an index `j`, a recognizer is a parser which returns, as output, a set of indices representing positions at which the parser successfully finished recognizing a sequence of tokens that began at position `j`. An empty result set indicates that the recognizer failed to recognize any sequence beginning at index `j`. A non-empty result set indicates the recognizer ends at different positions successfully.

- The `empty` recognizer recognizes the empty string. This parser always succeeds, returning a singleton set containing the current position:

$$empty(j) = \{j\}$$

- A recognizer `term 'x'` recognizes the terminal `x`. If the token at position `j` in the input string is `x`, this parser returns a singleton set containing `j + 1`; otherwise, it returns the empty set.

$$term(x, j) = \begin{cases} \{\}\,, & j \geq \#input \\ \{j+1\}\,, & j^{th} \text{ element of } input = x \\ \{\}\,, & \text{otherwise} \end{cases}$$

Note that there may be multiple distinct ways to parse a string while finishing at the same index: this indicates an ambiguous grammar. Simple recognizers do not acknowledge these ambiguities; each possible finishing index is listed only once in the result set. For a more complete set of results, a more complicated object such as a parse tree must be returned.

Following the definitions of two basic recognizers `p` and `q`, we can define two major parser combinators for alternative and sequencing:

- The ‘alternative’ parser combinator, `<+>`, applies both of the recognizers on the same input position `j` and sums up the results returned by both of the recognizers, which is eventually returned as the final result. It is used as an infix operator between `p` and `q` as follows:

  $$(p \quad <+> \quad q)(j) = p(j) \cup q(j)$$

- The sequencing of recognizers is done with the `<*>` parser combinator. Like `<+>`, it is used as an infix operator between `p` and `q`. But it applies the first recognizer `p` to the input position `j`, and if there is any successful result of this application, then the second recognizer `q` is applied to every element of the result set returned by the first recognizer. `<*>` ultimately returns the union of these applications of q.

  $$(p \quad <*> \quad q)(j) = \bigcup\{q(k) : k \in p(j)\}$$

Examples

Consider a highly ambiguous context-free grammar, `s ::= ‘x’ s s | ε`. Using the combinators defined earlier, we can modularly define executable notations of this grammar in a modern functional language (e.g. Haskell) as `s = term ‘x’ <*> s <*> s <+> empty`. When the recognizer `s` is applied on an input sequence `xxxxx` at position `1`, according to the above definitions it would return a result set `{5,4,3,2}`.

Shortcomings and solutions

Parser combinators, like all recursive descent parsers, are not limited to the context-free grammars and thus do no global search for ambiguities in the LL(k) parsing First_k and Follow_k sets. Thus, ambiguities are not known until run-time if and until the input triggers them. In such cases, the recursive descent parser defaults (perhaps unknown to the grammar designer) to one of the possible ambiguous paths, resulting in semantic confusion (aliasing) in the use of the language. This leads to bugs by users of ambiguous programming languages, which are not reported at compile-time, and which are introduced not by human error, but by ambiguous grammar. The only solution which eliminates these bugs is to remove the ambiguities and use a context-free grammar.

The simple implementations of parser combinators have some shortcomings, which are common in top-down parsing. Naïve combinatory parsing requires exponential time and space when parsing an ambiguous context free grammar. In 1996, Frost and Szydlowski[5] demonstrated how memoization can be used with parser combinators to reduce the time complexity to polynomial. Later Frost used monads[6] to construct the combinators for systematic and correct threading of memo-table throughout the computation.

Like any top-down recursive descent parsing, the conventional parser combinators (like the combinators described above) won’t terminate while processing a left-recursive grammar (e.g. `s ::= s <*> term ‘x’|empty`). A recognition algorithm[7] that accommodates ambiguous grammars with direct left-recursive rules is described by Frost and Hafiz in 2006. The algorithm curtails the otherwise ever-growing left-recursive parse by imposing depth restrictions. That algorithm was extended[8] to a complete parsing algorithm to accommodate indirect as well as direct left-recursion in polynomial time, and to generate compact polynomial-size representations of the potentially-exponential number of parse trees for highly-ambiguous grammars by Frost, Hafiz and Callaghan in 2007. This extended algorithm accommodates indirect left-recursion by comparing its ‘computed-context’ with ‘current-context’. The same authors also described[4] their implementation of a set of parser combinators written in the Haskell programming language based on the same algorithm. The X-SAIGA [9] site has more about the algorithms and implementation details.

References

[1] Frost, Richard. and Launchbury, John. "Constructing natural language interpreters in a lazy functional language." *The Computer Journal — Special edition on Lazy Functional Programming* Volume 32, Issue 2. Pages: 108 – 121, 1989

[2] Hutton, Graham. "Higher-order functions for parsing." *Journal of Functional Programming*, Volume 2 Issue 3, Pages: 323– 343, 1992.

[3] Swierstra, S. Doaitse. "Combinator parsers: From toys to tools. " *In G. Hutton, editor, Electronic Notes in Theoretical Computer Science,* volume 41. Elsevier Science Publishers, 2001.

[4] Frost, R., Hafiz, R. and Callaghan, P." Parser Combinators for Ambiguous Left-Recursive Grammars." *10th International Symposium on Practical Aspects of Declarative Languages (PADL), ACM-SIGPLAN* , Volume 4902, year 2008, Pages: 167-181, January 2008, San Francisco.

[5] Frost, Richard. and Szydlowski, Barbara. "Memoizing Purely Functional Top-Down Backtracking Language Processors. " *Sci. Comput. Program.* 1996 - 27(3): 263-288.

[6] Frost, Richard. "Monadic Memoization towards Correctness-Preserving Reduction of Search. " *Canadian Conference on AI 2003.* p 66-80.

[7] Frost, R. and Hafiz, R." A New Top-Down Parsing Algorithm to Accommodate Ambiguity and Left Recursion in Polynomial Time." *ACM SIGPLAN Notices*, Volume 41 Issue 5, Pages: 46 - 54. Year: 2006

[8] Frost, R., Hafiz, R. and Callaghan, P." Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars ." *10th International Workshop on Parsing Technologies (IWPT), ACL-SIGPARSE* , Pages: 109 - 120, June 2007, Prague.

[9] http://www.cs.uwindsor.ca/~hafiz/proHome.html

External links

- X-SAIGA (http://www.cs.uwindsor.ca/~hafiz/proHome.html) - e**X**ecutable **S**pecific**A**t**I**ons of **G**r**A**mmars

Article Sources and Contributors

Parser combinator *Source*: http://en.wikipedia.org/w/index.php?oldid=471871856 *Contributors*: AngelHaf, Brighterorange, Daniel5Ko, Edward, Justin W Smith, Kwamikagami, Liempt, R'n'B, Rjwilmsi, Ruud Koot, Shelbymoore3, Underpants, 16 anonymous edits

License