ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

LAMP
Laboratory for Programming Methods

Semester Project Report
Fall 2008

Packrat Parsing in Scala

"The term pack rat is [..] used in English as slang to refer to a person who collects miscellaneous items and has trouble getting rid of them (a compulsive hoarder) [...]" [1]

Author:
Manohar Jonnalagedda

Supervisor:
Prof. Martin Odersky
Assistant:
Tiark Rompf

January 17, 2009

[1] http://en.wikipedia.org/wiki/Pack_rat

Contents

1 Introduction **2**
1.1 The Scala Parser-Combinator Library 2

2 Packrat Parsing **4**
2.1 Motivation 4
2.2 Basic Idea 4
2.3 Memoizing with lazy evaluation 4
2.4 Memoizing without lazy evaluation 5

3 Left Recursion **6**
3.1 Motivation 6
3.2 Seed Parse 7
3.3 Growing the seed 7
3.4 Conclusion 9

4 Indirect Left Recursion **10**
4.1 Motivation 10
4.2 Extra Recursion 10
4.2.1 Description 10
4.2.2 Example 13
4.2.3 Issues 13
4.3 Warth's Recursion 14
4.3.1 Algorithm 15

5 Implementation Details **18**
5.1 Main Goals 18
5.2 Class Hierarchy 18
5.3 Conversion to Packrat 19

6 User Guide **20**

7 Performance **22**

8 Conclusion **26**

9 Acknowledgements **27**

Chapter 1

Introduction

Packrat Parsing is a technique for implementing backtracking, recursive-descent parsers, with the advantage that it guarantees unlimited lookahead and a linear parse time [2]. Using this technique, one can design parsers that can accept left recursive grammars [7], thereby drastically expanding the class of grammars that can be accepted by such parsers. The goal of this project was to integrate such techniques on top of the Scala [1] parser-combinator library, with the main objective being seamless integration into the library, and retaining the same ease of use as the original library.
This report is organised in the following manner:
The next section gives a brief introduction to the parser-combinator library, and how to use it. Chapter 2 introduces Packrat Parsing and a few implementation details behind it. Chapter 3 describes how one could integrate direct left recursion into Packrat Parsers. Chapter 4 looks at indirect left recursion : two different approaches to extend left recursion are shown. Chapter 5 expands on the implementation details and structuring of the code. Chapter 6 is a short tutorial on how to use the Packrat Parsing library in Scala. Chapter 7 gives some performance results. Finally, chapter 8 concludes.

1.1 The Scala Parser-Combinator Library

The Scala language provides an internal domain-specific language (DSL) for constructing parsers for context-free grammars. The building blocks of this language are known as parser-combinators : functions that help construct larger parsers in a simple way.
Parsers are represented as functions that map inputs to results :
abstract class `Parser[+T]` **extends** `(Input => ParseResult[T])`. Two of the most important combinators are :

- the `~` operator : this is used for sequencing different parsers: `A ~ B` yields a parser that succeeds if `A` succeeds, *and* if `B` succeeds on the remaining input.
- the `|` operator, used for alternative composition : `A | B` yields a parser that first tries to parse `A`. If that fails, then `B` is tried (and its result returned). Note that this means that order is important in the declaration of one's grammar.

With the above, it is very easy to write parsers in Scala (almost as simple as writing them on paper), and declare operations on them. Listing 1.1 shows a simple parser for accepting arithmetic expressions (products and sums), and how to run it [2]:

```
import scala.util.parsing.combinator.syntactical.StandardTokenParsers
class Arith extends StandardTokenParsers{
  lexical.delimiters ++= List("+","*","-","/","(",")")

  def sum: Parser[Any] = product~"+"~sum | product
  def product: Parser[Any] = primary~"*"~product | primary
  def primary: Parser[Any] = "("~expr~")" | numericLit

  def main(args : Array[String]) = {
    println(term(new lexical.Scanner("(1+2)*3")))
  }
}
```

[1] `http://www.scala-lang.org`
[2] the writing of the grammars is easy enough for us not to give bnf versions of them

Listing 1.1: An arithmetic expression parser

The parsers are recursive-descent, backtracking ones : the leftmost production is tried first, and inner productions are recursively called, till an output is produced. If at a certain stage, a production has failed, then backtracking is done till the next closest possible production [3].

[3] a detailed example is discussed in the section 2.1. A detailed tutorial on how to use this library is given in [5]

Chapter 2

Packrat Parsing

Packrat Parsing was a result of Bryan Ford's master thesis [1, 2]. He developed a new parsing technique that guarantees unlimited lookahead and linear parse time, in a lazy functional programming language. The following section presents the basic idea behind the technique.

2.1 Motivation

```
def sum: Parser[Any] = product~"+"~sum | product
def product: Parser[Any] = primary~"*"~product | primary
def primary: Parser[Any] = "("~expr~")" | floatingPointNumber
```

Listing 2.1: Another arithmetic expression parser

Consider the code in Listing 2.1, which is the same as presented in the previous chapter. Let us go through how to parse a simple input, say the number 9 :

1. We start at the `sum` production. We start on the left side, and therefore look for a `product~"+"~sum`, i.e. a `product`.
2. Similarly, we enter the `primary` production next.
3. In the `primary` production, we look for a left parenthesis, but fail on it (therefore the whole left-side fails). But the alternative `|` combinator tells us to look for a `floatingPointNumber`, where we can finally parse our 9.
4. back at the `product` level, we are in a situation where we are looking for a `*` after having parsed a `primary` (our number 9). There is nothing remaining in the input, hence we fail, and we *reparse* `primary` once again, giving us step 3 all over again.
5. at this level, we are in `sum`, and we are looking for a `+`. Not finding it, we have to reparse the `product` production again, giving us an unnecessary repetition of steps $1, 2, 3$.
6. we finally return the parsed result, i.e. the number 9.

As we can notice, simple backtracking causes the parser to reparse known productions multiple times. As most grammars are usually written this way, this is a non-negligible cost.

2.2 Basic Idea

As we can see above, a normal recursive-descent parser can cause inputs to be parsed multiple times before getting the required parse. Indeed, after step 2, we have already parsed a `primary` production, thereby making the second parse redundant.
How can we avoid doing this double work ? Intuitively, what comes to mind is : save the result *somewhere*, so when we come back, we can just fetch the result. And this is exactly the idea behind packrat parsing and memoizing parsers : once a production is parsed *at a certain position*, save the corresponding result for future reference.

2.3 Memoizing with lazy evaluation

Ford's work presents a parsing technique that makes use of a programming language evaluation technique called *lazy evaluation* : in such an environment, an argument is evaluated, or computed, only when it's result is required. Furthermore, it is computed *only once* [1] : The result is stored in some kind of cache at the programming language level, and when the argument is called a second time, this result is returned. The Haskell programming language [2], in which Ford's implemented his proof-of-concept parser, operates in such a way.
So, if we look back at step 4 in section 2.1, we do not need to reparse the `primary` production : due to lazy evaluation, it's result has been calculated and stored, and therefore, when it is called a second time, we just return 9, as has been parsed in step 3. The same argument is also valid for step 5.

2.4 Memoizing without lazy evaluation

Most languages don't have lazy evaluation, nor call-by-name strategies built into them. Scala, however, gives the choice to the programmer : in order to have call-by-name parameters for a function, one needs to add a `=>` sign in front of the argument type : **def** `method (param :=> Int) => ...`
If call-by-need emulation is needed, the following can be done[3] :

```
def method(param: => Int) {
  lazy val cachedParam = param
  // use only cachedParam from here on
...
}
```

Another option, of course, is to use an explicit dynamic lookup structure which is independent from the evaluation strategy. A hint of how this can be done is given in Listing 2.2.

```
abstract class PackratParser[+T] extends Parser[T]

//an implicit function converting a parser into a packrat parser
implicit def parser2packrat[T](p: => super.Parser[T]): PackratParser[T] = {
  memo(super.Parser{in => p(in)})
}

//the cache
val cache : HashMap[(super.Parser[_], Position), Result[_]] = HashMap.empty

//the memo function, taking care of all memoization
def memo[T](p:  super.Parser[T]): PackratParser[T] = {
  new PackratParser[T] {
      def apply(in: Input) = {
      cache.get((p, in.position)) match {
      //nothing in the cache
      case None =>
        val result = p(in)
        cache.put((p, in.position), result)
        result
      //something in the cache
      case Some(res) => res
      }
      }
    }
  }
```

Listing 2.2: Memoization of a parse result

When nothing is found in the cache for a specific parser at a specific position, we compute the result, place it in the cache, and then return the result. Otherwise, we simply return whatever we find in the cache. Though a bit less elegant, this method is not much harder to implement than

[1] this is also known as call-by-need evaluation [6]
[2] http://www.haskell.org
[3] as suggested on `http://www.nabble.com/Why-"by-name"-parameters-are-called-this-way-td20723744#a20741292.html`

the lazy one, and moreover, it has the advantage that one can dynamically change a given entry. This feature is very useful in the context of left-recursion, as we will see in the next chapter.

Chapter 3

Left Recursion

Packrat Parsers can be used to integrate left recursive grammars. Ford suggests a method to transform the involved productions into right-recursive ones behind the scenes. Frost et al. introduced a counter-based method for the above [3]. Warth et al. suggest another method, which consists in saving a seed parse, and then growing it if required [7].
We decided to implement the third method. In the following sections, we will explore the general idea, and how one could implement it for direct left-recursion. The next chapter will then describe how one can build on this basis to integrate indirect left-recursion.

3.1 Motivation

First of all, let us try to look at why left recursion is not possible with the Scala parser-combinator library, and why it would be interesting to integrate it. Consider the following grammar which parses and indefinite number of ones :

```
def lotsOfOnes : Parser[Any] = lotsOfOnes ~ "1" | "1"
```

Listing 3.1: a left-recursive grammar

If this parser is applied to any input, the `lotsOfOnes` function is going to be recalled once more, and so on and on. This results in an infinite loop, represented by a `StackOverFlowException` in Scala.
Of course, once can transform the above grammar into a non left-recursive one, as follows :
`def lotsOfOnes : Parser[Any] = "1"~lotsOfOnes | "1"`
Any left-recursive grammar can be transformed into a non-left-recursive grammar using some basic steps [1]. However, most useful grammars are left-recursive; Java's grammar, for example, contains rules that are indirectly left-recursive over 5 levels [4]. Transforming these rules by hand is a tedious job, and one can easily make mistakes.
Moreover, left-recursive grammars provide direct left-associativeness in operators when constructing an abstract syntax tree (AST). If a rule has to be transformed, then constructing the AST correctly also gets harder. As an example, consider the grammar for the simple lambda-calculus as in listing [6] :

```
t ::= x (variable)
  | λx.t (abstraction)
  | t t (application)
  | "(" t ")"
```

Listing 3.2: the simple untyped lambda-calculus

As we can notice, the abstraction rule is left-recursive; the rule therefore has to be transformed, and for correctly constructing the abstraction tree, the grammar would have to written as follows in Scala [2]:

```
def x : Parser[Term] = ident ^^ Var(_)
def term1 : Parser[Term] = (
  x
  | ("\\"~>x<~".") ~ term ^^ {case (a:Var)~b => Abs(a,b)}
```

[1] http://en.wikipedia.org/wiki/Left_recursion
[2] for more help on how to transform parse results, please have a look at the Parser-Combinator documentation [5]

```
  | "("~>term<~")"
def term : Parser[Term] = (
  (rep1(term1)) ^^ {
    case x::xs => xs.foldLeft(x){(func, arg) => App(func, arg)}
} | failure("illegal start of term") )
```

Listing 3.3: a parser for the simple lambda-calculus

As we can see, the transformation of the application rule to a non-left-recursive one has caused us to create two rules, with a special one for the application rule. Moreover, to create a left-associative AST, we have to invoke the `foldLeft` function.
Therefore, while left-recursive grammars can always be transformed into non left-recursive ones, it is quite useful to be able to directly write left-recursive grammars.

3.2 Seed Parse

How could we accept left recursion ? Let us recall the grammar from previous section :

```
val lotsOfOnes : Parser[Any] = lotsOfOnes ~ "1" | "1"
```

As described above, applying the parser on any input will cause a `StackOverFlowException`. We somehow need to *arrest* the parser from doing so. This, thanks to the cache introduced in the previous chapter, is now easy : *before* parsing an input, we first check whether something is contained in the cache. If nothing is there, then we explicitly insert a dummy failure :

```
def memo[T](p:  super.Parser[T]): PackratParser[T] = {
  new PackratParser[T] {
    def apply(in: Input) = {
    // handle caching here
      in.getCache.get((p, in.position)) match {
        case None =>
          in.cacheAndGive(p, Base(Failure("LeftRecursion", in)))
          val tempRes = p(in)
          tempRes
        case Some(res) => res
      }
    }
  }
}
```

Listing 3.4: Failing the initial parse

This way, when we try to parse `lotsOfOnes` the first time, a dummy failure is inserted in the cache (note that we wrap the result in a `Base` class : this helps us distinguish between recursive and non-recursive inputs, as we will see later). The second time around (when trying to parse the left side of the production), a failure is returned. The `|` alternating combinator therefore checks the right side, and we can safely parse the first "1" from our input.

3.3 Growing the seed

Obviously, we are unsatisfied with having only the seed input. We would like to parse as many 1s as possible. The procedure to do so is known as growing. The idea is as follows :

- We have parsed a seed which is not a failure. It is possible that some more input can also be parsed in a similar fashion : we are optimistic that the same rule can be applied for longer.
- We try to grow the parse result thus : we apply the production on the input, and get a new result. If the position in the new result is higher than the previous one, we have succeeded in parsing more. Therefore, we can try applying the production on the remaining input again. If, however, the position is lesser or equal to the previous position, it means that we haven't been able to grow our result. Therefore, we are done.

Listing 3.5 gives a possible implementation of the `memo` function which calls the grow function when required.

```
def memo[T](p:  super.Parser[T]): PackratParser[T] = {
  new PackratParser[T] {
    def apply(in: Input) = {
      in.getCache.get((p, in.position)) match {
        //nothing in the cache
        case None =>
          in.cacheAndGive(p, Base(Failure("LeftRecursion", in)))
          //future parse : this line will change the wrapper in case recursion been detected
          val tempRes = p(in)
          in.getCache.get((p,in.position)) match {
            case Some(InRecursion(_)) =>
              //recursion detected
              in.cacheAndGive(p, InRecursion(tempRes))
              grow[T](p,in)
            case Some(Base(_)) => in.cacheAndGive(p, Base(tempRes)).getResult
          }
        //something in the cache
        case Some(res) => res match {
          case Base(Failure("LeftRecursion",in2)) =>
            in.cacheAndGive[T](p, InRecursion(Failure("LeftRecursion", in2))).getResult
          case _ => res.getResult
        }
      }
    }
  }
}
```

Listing 3.5: accepting direct left recursion

Let us observe what goes on above :

- When nothing is found in the cache, a base failure is cached, as before. We then try to apply the parser to the input (storing it in `tempRes`).
- At this point, if we are looking at a left recursive function, the `memo` function will be called again. However, we will find something in the cache, precisely what we stored in step 1. Therefore the second part of the function is called.
- As we have a `Base` failure in the cache, we have now detected that there is a recursion : we wrap the result in a `InRecursion` class and cache it. (Note that `Base` and `InRecursion` are both case classes, whose function is to help distinguish between normal and left-recursive results).
- We now have a result for `tempRes` (which is simply a `ParseResult`, with no wrappers). If, in the cache, we find an `InRecursion`, we have a seed parse in a left recursion. We can therefore grow our result. Otherwise, the production in non left-recursive. We can simply cache `tempRes` as a base result.

The grow function would be as follows :

```
def grow[T](p : super.Parser[T], rest : Reader) : Result[T] = {
  val oldRes : Result[T] = rest.getCache.get(p, rest.position) match{
    case Some(InRecursion(r: Result[T])) => r
    case _ => throw new Error("impossible match")
  }

    val tempRes = p(rest); tempRes match {
    case s@Success(_,_) =>
      //success !
      if(oldRes.getPos < s.getPos){
        rest.cacheAndGive(p, InRecursion(s)); grow(p, rest)
      }else oldRes
    case _ => /* never get here*/ rest.cacheAndGive(p, InRecursion(oldRes)).getResult
  }
}
```

Listing 3.6: the growth function

3.4 Conclusion

Integrating Left Recursion involves getting a seed parse, and if possible, grow the initial seed parse. With the above, we can now parse left-recursive grammars as complex as the following :

```
val term: PackratParser[Any] = (
   term ~ "+" ~ fact
  |term ~ "-" ~ fact
  |fact
)

val fact: PackratParser[Any] = (
   fact ~ "*" ~ digit
  |fact ~ "/" ~ digit
  |digit
)
```

Listing 3.7: a left recursive parser for arithmetic expressions

However, indirect left recursion is not yet possible. The next chapter will explore two approaches on how one could integrate that.

Chapter 4

Indirect Left Recursion

This chapter discusses how one can integrate indirect left recursion into the parsers we have developed so far. To start off, we first motivate why the techniques discussed in the previous chapter are not sufficient. We then look at two approaches :

- the first one was developed by us. The idea is to continue recursion on a left-recursive production and wrap results with different labels for distinguishing them. This approach doesn't work on all types of grammars : we will justify why not.
- the second approach is adapted from Warth et al.s results [7]. After thinking about our approach, we found out that for it to work, we would have to introduce structures that would render it as complex as the one presented in the paper. We therefore decided to implement the second approach, as we found it to be more sound.

4.1 Motivation

Consider the following grammar :

```
val lotsOfOnes2: PackratParser[Any] = expr
val expr: PackratParser[Any] = lotsOfOnes2 ~ "1" | "1"
```

Listing 4.1: an indirect left-recursive parser

It should parse any amount of 1s, and there is an indirect left recursion in that `lotsOfOnes2` refers to `expr` which refers back to `lotsOfOnes2`.
Why can't the techniques used in the previous chapter be used as such ? Let the input be `"1 1"` :

- When we apply `lotsOfOnes2` to the input, we do get a seed parse of 1, In the cache, entries for both `lotsOfOnes2` and `expr` contain an `InRecursion`.
- Therefore, the grow function is called on `lotsOfOnes2`. In the latter, when we try to get the future parse, all we get is the result stored in the cache for `expr` : this result is the same as the one we started growing with.
- The growth function returns without having grown anything, because it found a result in the cache it couldn't ignore.

The cached result for `expr` is blocking us from going ahead in our parse. We need to be able to ignore this intermediary result when such a thing happens. In the following sections, we discuss how one can manage to do so.

4.2 Extra Recursion

4.2.1 Description

Let us add an extra level of indirection in the above grammar, so that we can understand the idea better :

```
val a: PackratParser[Any] = b | numericLit
val b: PackratParser[Any] = c
val c: PackratParser[Any] = a ~"+"~numericLit
```

Listing 4.2: another indirect left-recursive parser

When we apply a on an input, a parse tree is created from a to c : a → b → c → a. In order to be ignore intermediary productions, one idea is to continue the recursion (thereby expanding the tree), and label results with different wrappers, in order to use them as indicators of where in the recursion we might be. We can detect that a left-recursion is taking place once we see a for the second time (when trying to parse c). We would now like to go into the recursion till we see a for the third time : a → b → c → a → b → c → a. The reasoning is that, on the second time of travel, we cache the results we obtain with different wrappers. When we reach the end of the second recursion, we know exactly where it ends, and therefore we can grow a result without any problems.

To be more clear, we introduce the following wrappers :

```
abstract class LRWrapper[+T]
case class Base[+T](r : Result[T]) extends LRWrapper[T]
case class InRecursion[+T](r : Result[T]) extends LRWrapper[T]
case class LResult[+T](r : Result[T]) extends LRWrapper[T]
case class PlainResult[+T](r : Result[T]) extends LRWrapper[T]
```

Listing 4.3: different result wrappers

- the Base wrapper represents, as before, the base failure we insert into the cache before attempting a parse :

```
def memo[T](p:  super.Parser[T]): PackratParser[T] = {
  new PackratParser[T] {
    def apply(in: Input) = {
      in.getCache.get((p, in.position)) match {
        case None =>
          in.cacheAndGive(p, Base(Failure("LeftRecursion", in)))
          //future parse
          val tempRes = p(in)
          ...
        case Some => ...
      }
    }
  }
}
```

Listing 4.4: how the Base wrapper works

[1]

- the InRecursion wrapper is used for detecting the first level of recursion (until the second a is seen). As in the previous chapter, when a Base result is seen in the cache, we change the wrapper. Furthermore, we once again apply the parser to the input (second level of recursion) :

```
case None =>
  ...
  val tempRes = p(in)
  in.getCache.get((p,in.position)) match {
    ...
  }

case Some(res) => res match {
  //we see a base result, wrap it differently
  case Base(r:Result[T]) =>
    in.cacheAndGive[T](p,InRecursion(r)).getResult
    in.cacheAndGive[T](p, LResult(p(in))).getResult
  /*
   * when we detect InRecursion, we have reached the production
   * that started the whole thing, a second time
   */
  case InRecursion(r:Result[T]) => r
  ...
```

[1] for the next wrappers, only the code in the **case** Some and **case** None will be shown

Listing 4.5: how the `InRecursion` wrapper works

- the `LResult` wrapper is used for marking an end to the second recursion. At first, all productions are labeled as `Base`. Then, when one of these is detected, they are all labeled as `InRecursion`. As shown in listing (ABOVE), at each of the `InRecursion` labeling, the parser is once again applied to the input, and the result is labeled `LResult`.
  This label also designates all intermediate productions. Therefore, when we detect an `LResult` in the cache, we know we are inside the grow function, and therefore we can forward the parsing to the next level :

```
case None =>
  val tempRes = p(in)
  in.getCache.get((p,in.position)) match {
    ...
  }

case Some(res) => res match {
  ...
  /*
   * when we detect an LR-Result, we are inside a growing function
   * the goal is then to grow the parse. But we need a way to stop
   * this growing when we reach the recursion starter, hence caching
   * a "ParseResult"
   */
  case LResult(r : Result[T]) =>
    in.cacheAndGive[T](p,PlainResult(r))
    in.cacheAndGive[T](p, LResult(p(in))).getResult
  ...
}
```

Listing 4.6: how the `LResult` wrapper works

- Finally, the `PlainResult` wrapper is used as follows :
  in the above listings, we can observe that a future parse is kept in a `tempRes` value. At this point, the cache can only contain two types of wrappers : a normal type or an `LResult`. In the first case, no left-recursion was detected, and we simply wrap the result with a `PlainResult`, declaring we are done. In the second case, we can grow the result. Here too, we wrap a `PlainResult`, and call the grow function. In this case, the wrapper is used to recognise when in the grow function one needs to stop (because during the growth, we also traverse the tree. When we see a `PlainResult`, we know that the tree traversal has come to an end) :

```
case None =>
  val tempRes = p(in)
  in.getCache.get((p,in.position)) match {
    case Some(LResult(r : Result[T]))=>
      in.cacheAndGive(p, PlainResult(tempRes)).getResult
      in.cacheAndGive(p, PlainResult(grow[T](p,in))).getResult
    case Some(Base(_)) =>
      in.cacheAndGive(p, PlainResult(tempRes)).getResult
    case r =>
      in.cacheAndGive(p, PlainResult(tempRes)).getResult
  }

case Some(res) => res match {
  ...
  /*
   * when we detect a PlainResult, we have reached the bottom of
   * a growing tree, so we return what we see
   */
  case PlainResult(r: Result[T]) => r
  case _ => throw new Error("Unmatchable")
}
```

Listing 4.7: how the `PlainResult` wrapper works

4.2.2 Example

To understand how the algorithm works, let us try to apply the input `"1+2"` to production `a`. Figure 4.1 shows the tree that is produced before detecting the recursion. The cache on the diagram shows that all values in the cache are `Base` ones.

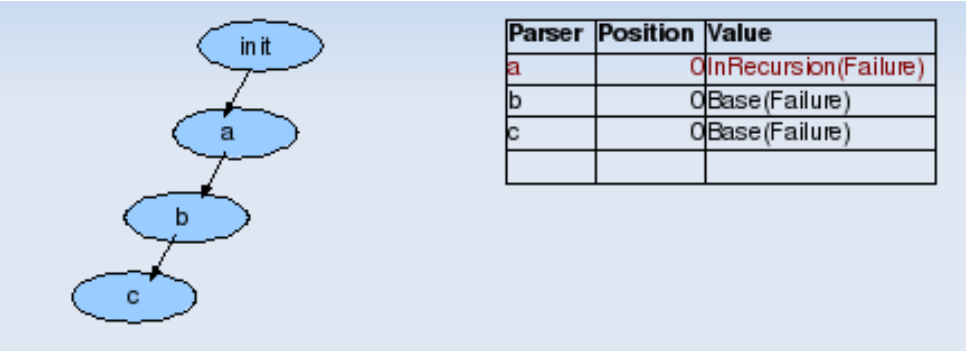

Figure 4.1: parse tree and cache before recursion is detected

Once the recursion is detected, the tree is expanded, till all productions are labeled `OneRecursion` (Fig. 4.2).

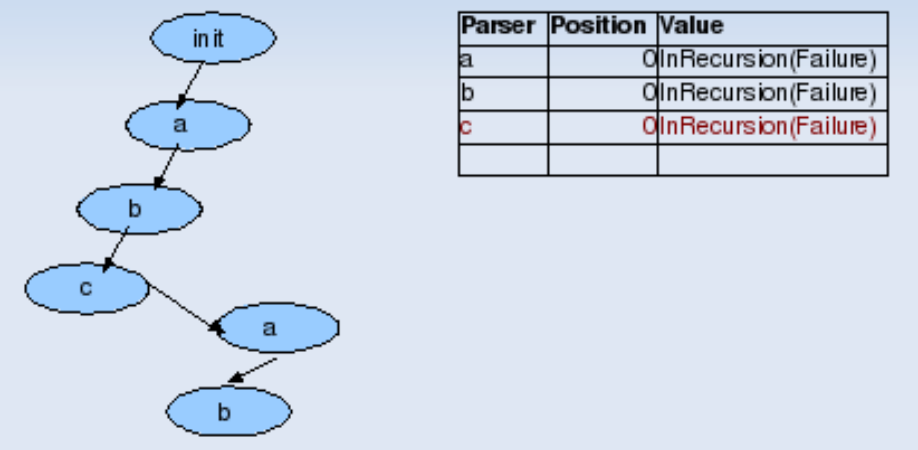

Figure 4.2: labeling all productions `oneRecursion`

Once the second level of recursion is completed, all productions are recursively labeled `LResult` (Fig. 4.3).

Once we reach the recursion initiator, the latter is labeled `PlainResult`, and growth is possible.(Fig. 4.4).

4.2.3 Issues

Adding an extra level of recursion and a few extra labels, as described above, seems to be an elegant way of introducing indirect left recursion. Unfortunately, the technique we described above doesn't work in all cases. Consider for example the following grammar :

```
val exp : PackratParser[Any] = sum | prod | digit
val sum : PackratParser[Any] = exp ~ '+' ~ exp
val prod: PackratParser[Any] = exp ~ '*' ~ (digit | exp)
```

Listing 4.8: an indirect left-recursive parser for products and sums

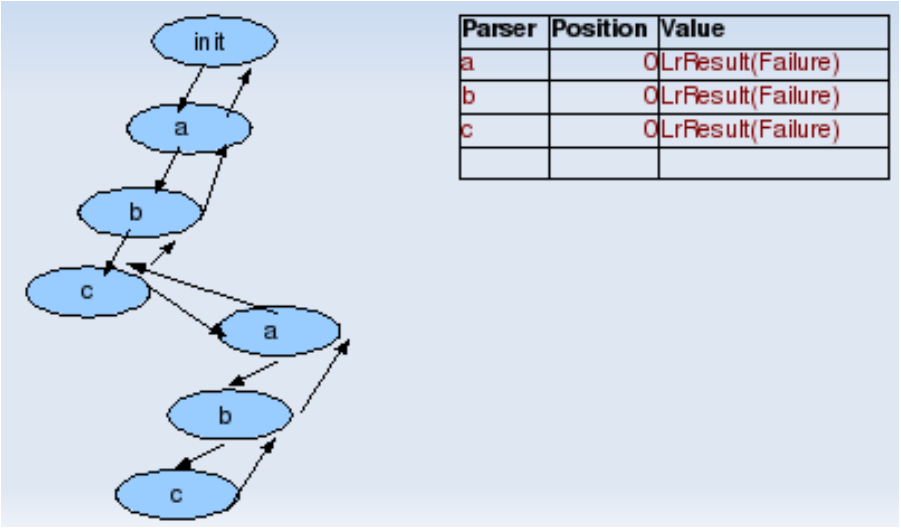

Figure 4.3: all productions are recursively labeled `LResult`

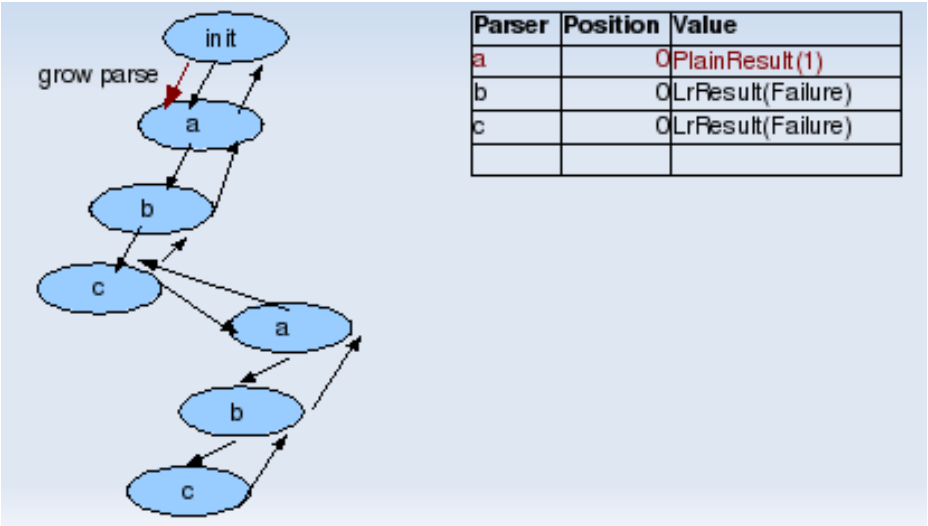

Figure 4.4: growth can commence

Let us try to parse input `"2*2"`. In this case, by the time we get a seed parse, `LResults` for both `sum` and `prod`, and a `PlainResult` for `exp`. So the growing function can take place. Unfortunately, the latter function only takes into account the `sum` production. Therefore, it will try to grow only through the `sum` production. Naturally, this won't go too far. So, a result of 2 is returned, and `"* 2"` is left as the remaining input.
The problem we note here is that some indirect ruecursion rules may be left out because the growth function only looks at the first rule causing such a recursion. In order to repair this flaw, we can think of keeping a list of members included in a recursion, and test all of these when a recursion is detected. Methods similar to these are present in the algorithm by Warth et al. In the next section, we will have a deeper look at it.

4.3 Warth's Recursion

Intuitively, for indirect left recursion, one needs to know the following things :

- We need to keep account of the recursion tree
- We need to know which parser is at the head of each left recursion
- We need to ignore the cache when growing a parse

In order to do so, we have the following data structures :

- **case class** `MemoEntry[+T](var r: Either[LR,Result[_]])` **extends** `LRWrapper[T]` : a class that helps distinguish between recursive and non-recursive results, using the `Either` variant constructor. Note the cache is a mapping between parsers, positions and `LRWrappers`.

- **case class** `LR(var seed : Result[_], var rule : Parser[_], var head: Option[Head])` : a class that represents a parse result that is involved in a left-recursive rule. Therefore, we have a reference to the rule to which it corresponds, and also to the head rule of the recursion. Notice that all these fields are `vars` : this helps to modify their values via assignment and by side-effect. Though not pleasing from a functional programming perspective, this eases the implementation of the described algorithm.

- a `Head` structure :

  ```
  case class Head(var headParser : Parser[_],
    var involvedSet : List[Parser[_]], var evalSet : List[Parser[_]])
  ```

  A class that represents the head of a recursion. Apart from keeping a reference to the head parser, we also keep a list of all parsers involved in the recursion. The `evalSet` list contains all rules that may be evaluated during a growth phase : this list can therefore change quite often.

- **val** `recursionHeads : HashMap[Position, Head]` : a global variable that maps a position to a recursion head. At any point of time, only one growth phase can take place at a given position. The above structure therefore gives rapid access to the head of the recursion. If the hash map does not contain an entry for a given position, this means that recursion growth is not underway at that position.

- **var** `lrStack : List[LR]` : another global value which acts as a stack keeping account of the tree of productions called in a parse. A recursion is detected if any production appears than once on the stack.

4.3.1 Algorithm

With the above structures, the algorithm goes as follows :

- when an input is applied to a parser, a `recall` function is called. This function verifies whether we are in the process of growing a parse or not. In the former case, it makes sure that rules involved in the recursion are evaluated. It also prevents non-involved rules from getting evaluated further :

```
private def recall(p : super.Parser[_], in : Input) : Option[LRWrapper[_]] = {
  val cached = in.getFromCache(p)
  val head = recursionHeads.get(in.position)
  head match {
    case None => cached
    case Some(h@Head(hp, involved, evalSet)) => {
      if(cached == None && !(hp::involved contains p)) {
        return Some(MemoEntry(Right(Failure("dummy ",in))))
      }
      if(evalSet contains p){
        //remove the rule from the evalSet of the Head
        h.evalSet = h.evalSet.remove(_==p)
        val tempRes = p(in)
        val tempEntry: MemoEntry[_] = cached match {case Some(x: MemoEntry[_]) => x}
        in.incrementUpdate; tempEntry.r = Right(tempRes)
      }
    cached
    }
  }
}
```

Listing 4.9: the recall function

- Once the result of the `recall` function is known, if it is `nil`, then we need to store a dummy failure into the cache (much like in the previous listings) and compute the future parse. If it is not, however, this means we have detected a recursion, and we use the `setupLR` function to update each parser involved in the recursion.

```
private def setupLR(p : Parser[_], in: Reader, recDetect : LR) : Unit = {
  if(recDetect.head == None) recDetect.head = Some(Head(p, Nil, Nil))
```

```
    lrStack.takeWhile(_.rule != p).foreach{x =>
      x.head = recDetect.head
      recDetect.head.map(h => h.involvedSet = x.rule::h.involvedSet)
    }
  }
```

Listing 4.10: the setupLR function

- when we get the result of a future parse, we need to grow it. Because of and indirect recursion, however, we call the `lrAnswer` function, which takes care of invoking the grow function only if the rule is the head of the recursion :

```
def lrAnswer[T](p : Parser[T], in : Reader, growable : LR) : Result[T] = growable match{
  case LR(seed ,rule, Some(head)) =>
    if(head.getHead != p) seed
    else {
      in.cacheAndGive(p, MemoEntry(Right[LR, Result[T]](seed)))
      seed match{
      case f@Failure(_,_) => f
      case s@Success(_,_) => grow(p, in, head)
    }
  }
    case _=> throw new Error("lrAnswer with no head !!")
}
```

Listing 4.11: the lrAnswer function

- the grow function remains the same as before ; however, before starting to grow a production, we insert it and its position in the `recursionHeads` cache. (This helps the following `recall` functions to act accordingly). Once the growth is over, we must remove what we entered in the cache.

The `memo` function is given in listing 4.12:

```
def memo[T](p:  super.Parser[T]): PackratParser[T] = {
  new PackratParser[T] {
    def apply(in: Input) = {
      val m = recall(p, in)
      m match {
        //nothing has been done due to recall
        case None =>
          val base = LR(Failure("Base Failure",in), p, None)
          lrStack = base::lrStack
          in.cacheAndGive(p,MemoEntry(Left(base)))
          val tempRes = p(in)
          lrStack = lrStack.tail
          //check whether base has changed, if yes, we will have a head
          base.head match{
            case None =>
              prettyPrint("Simple Result")
              in.cacheAndGive(p,MemoEntry(Right(tempRes)))
              prettyPrint("At the end we have : "+tempRes)
              tempRes
            case s@Some(_) =>
              base.seed = tempRes
              val res = lrAnswer(p, in, base)
              res
            }
          case Some(mEntry) => {
            mEntry match {
              case MemoEntry(Left(recDetect)) => {
              setupLR(p, in, recDetect)
              recDetect match {case LR(seed, _, _) => seed}
              }
          case MemoEntry(Right(res : Result[T])) => res
```

```
            }
          }
        }
      }
    }
  }
```

Listing 4.12: the memo function

With the above, it becomes possible to implement any left-recursive grammar ! We used this algorithm to implement the recursion techniques, getting positive results. In the next chapter, we describe how we designed the software itself, i.e. what our main goals were, which hierarchy we chose, and other details that were important.

Chapter 5

Implementation Details

5.1 Main Goals

The main goals of our implementation were :

- seamless integration of packrat parsers into the normal library. Grammar writing should be as easy as not using packrat parsers.
- choice to the user : let the user decide which productions he wants to memoize.
- integrate the cache into the `Reader` object : hence each input has its own specific cache. This allows multiple input parsing without having to reset caches between parses.

5.2 Class Hierarchy

Figure 5.1 shows the hierarchy of the `scala.util.parsing.combinator` package. The main `Parsers` trait is extended by many other more specific traits and classes, which one can directly use when constructing their grammars (like the `StandardTokenParsers` class). The former trait also defines the abstract `Parser` class, and most useful combinators. We needed to have access to all of this functionality, and at the same time, be independent from the main trait. We therefore extended the `Parsers` trait : **`trait`** `PackratParsers` **`extends`** `Parsers`.
This new trait contains all the information that is relevant for packrat parsing :

- a `PackratReader` class which extends the `Reader` trait :

  ```
  class PackratReader[+T](in: Reader[T]) extends Reader[T]
  ```

  As we wanted the cache to be coupled with the input, and as the cache needed access to information about the parsers, the `PackratParsers` trait was the right place to code it.
- all data structures and functions described in the previous section.
- an abstract `PackratParser` class that extends the `Parser` class.
- other utility functions.

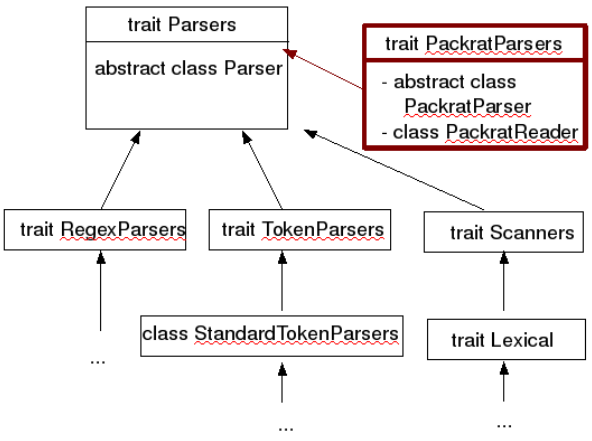

Figure 5.1: hierarchy of parser traits

5.3 Conversion to Packrat

We wanted to be able to write grammars as easily as before, i.e. the declaration of memoizing grammars to be similar to the normal ones (as seen in examples in previous chapters). To achieve this, we used an implicit conversion mechanism that turns any `Parser` into a `PackratParser` :

```
implicit def parser2packrat[T](p: => super.Parser[T]): PackratParser[T] = {
  memo(super.Parser{in => p(in)})
}
```

Listing 5.1: the implicit conversion to a packrat parser

Anytime we declare a **val** `p : PackratParser[Any] = declaration`, the implicit function is called (as we want to assign a `PackratParser` to `p`) [1]. Here, we notice that if we do not want to have a packrat parser, we simply need to change the type of `p`. This leaves the freedom to the user.

To get a better understanding of these details, we describe how to use the library in the next chapter.

[1] the `memo` function is the one that does all the memoizing and handling of recursion, as seen in the previous chapters

Chapter 6

User Guide

We are finally ready to use the packrat parsers ! We will go through an example, and highlight all important steps that we need to take in order to achieve the desired result.
Consider the example in listing REF, which constructs a simple arithmetic expression parser (and calculator) with left recursive grammars :

```
package firstPackrat

import packrat._
import scala.util.parsing.combinator.syntactical.StandardTokenParsers

object Arith extends StandardTokenParsers with PackratParsers{
  /****
   * term = term + fact | term - fact | fact
   * fact = fact * num  | fact / num  | num
   */
 lexical.delimiters ++= List("+","*","-","/","(",")")

 val term: PackratParser[Int] = (term~("+"~>fact) ^^ {case x~y => x+y}
          |term~("-"~>fact) ^^ {case x~y => x-y}
          |fact)

 val fact: PackratParser[Int] = (fact~("*"~>numericLit) ^^ {case x~y => x*y.toInt}
          |fact~("/"~>numericLit) ^^ {case x~y => x/y.toInt}
          |"("~>term<~")"
          |numericLit ^^ {_.toInt})

  def main(args : Array[String]) = {
    println(term(new PackratReader(new lexical.Scanner("(1+2)*3"))))
  }
}
```

Listing 6.1: an example of how to use packrat parsers

Compared to the first example (listing 1.1), we can notice the following differences :

- we import the `packrat` package. This is mandatory to be able to use the packrat parsing functionality
- our `Arith` trait now mixes the `PackratParsers` trait. We want to use the functionality of packrat parsers with the other classes we extend. This is a very important feature : indeed, *whenever* we need packrat parsing functionality, all we need to do is to mix `PackratParsers` in. This means that the `Arith` class could have as well been using `RegexParsers` or `JavaTokenParsers` : as long as it also used `PackratParsers`, we could have declared packrat parsers.
- each parser is declared as **val** `myParser : PackratParser` instead of **def** `myParser : PackratParser`. We are storing information regarding parsers in the cache, hence each of them *must* be a value, and not a function (which will generate different objects each time it is called).
- to run the program, we apply the `term` production to a `PackratReader` which encapsulates the `Scanner`. Each time a packrat parser needs to be run, a `PackratReader` must wrap the reader one would have used otherwise : this ensures memoization.

By following these 4 simple steps, we now have access to packrat parsing. In the next chapter, we discuss about the performance of the new library.

Chapter 7

Performance

Packrat Parsing gives us two main advantages :

- it allows to integrate left recursion. It is therefore important to know whether the additional expressivity adds extra overhead to the parse time.
- it is much faster on backtracking inputs. We therefore wanted to verify whether this holds with our library.

In order to test the first claim, we looked at the following grammars :

```
abstract class Tree
case class Leaf(name: String) extends Tree
case class Node(body: Term, name: String) extends Tree


val leftRec : PackratParser[Tree] = (
  (leftRec~("+"~>ident)) ^^ {case x~y => Node(x, y.toString)}
  | ident ^^ {case x => Leaf(x.toString)}
)

val identParser : PackratParser[Any] = ident

val leftRecAbusive : PackratParser[Tree] = (
  (leftRecAbusive~("+"~>identParser)) ^^ {case x~y => Node(x, y.toString)}
  | ident ^^ {case x => Leaf(x.toString)}
)

val rightRec : Parser[Tree] = (ident~rep("+"~>ident)) ^^{
  case x~(y::ys) => ys.foldLeft(Node(Leaf(x.toString),y))
    {case (a,b) => Node(a,b.toString)}
}
```

Listing 7.1: grammars for testing left-recursion efficiency

All these grammars parse `idents` and construct trees out of them. The first two are left-recursive. Whereas `leftRec` doesn't memoize ident, `leftRecAbusive` does so, as it parses memoizing `identParsers` (which is a `PackratParser`). Finally, `rightRec` is the right recursive alternative, which does the extra `foldLeft` operation in order to get the same tree structure at the end.

We ran these grammars on inputs ranging between 1000 and 10000 characters (in increments of 1), and measured the running times. The results are given in figure 7.1. The green line depicts the `rightRec` parser, the red line is for the `leftRec` parser, and the blue line represents the `leftRecAbusive` parser. We can notice the following :

- the time taken grows linearly with respect to the input length for all three grammars, which is quite relieving.
- the `rightRec` parser is the fastest. This is mainly due to the fact that the `rep` combinator and the `foldLeft` functions are very fast (they internally use **while** loops, which avoids extra stack space).

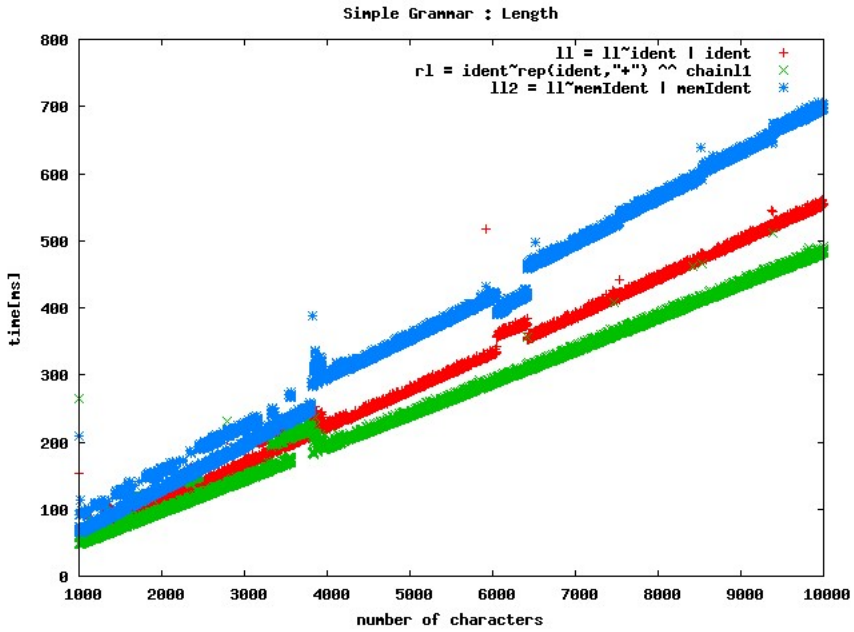

Figure 7.1: comparing left and right recursive grammars

- the `leftRec` parser is slower, but is not too much slower. We have a difference of 100ms for 10000 characters. This seems to be a decent tradeoff for the gain in expressivity we get due to left-recursive grammars.
- the `leftRecAbusive` parser, however, is much slower than the other two. It seems that memoizing each identifier on the way does add unnecessary overhead. This result vindicates our implementation decision to let the user choose when he wants to use packrat parsing : indeed, turning every parser into a packrat one causes unnecessary overkill.

In order to test whether we were faster on backtracking inputs, we considered the following grammars :

```
val identParser : PackratParser[Any] = ident
var normalGrammar : Parser[Any] = ident ||| ident ~ ident ||| ...
var packratGrammar : PackratParser[Any] = identParser ||| identParser ~ identParser ||| ...
```

Listing 7.2: grammars for testing backtracking efficiency

The `|||` combinator is similar to the `|` combinator, but it parses both sides, and returns the longest input. At each step, we added an extra identifier parser to the right side, with the input always corresponding to the length of the last alternative (if the last alternative contains 10 `idents`, the input will contain the same). As the `|||` parser both sides, backtracking will happen each time we try to parse a new option. By memoizing the identifier in `identParser`, we wanted to verify if the parse time would decrease. We ran tests for alternative sizes between 1 and 350. The results are shown in figure 7.2.

We notice the following :

- the packrat parser is indeed much faster : it is approx. twice as fast at a grammar size of 350.
- however, the parsing does not scale linearly with the size of the input here. This is because we only memoize single `identParsers`. If, for example, the grammar size is 10, once we have parsed 9 identifiers, and are starting the last alternative (`identParser ~ identParser ~...` 10 times), we go and look in the cache for all first 9 `identParsers`, instead of fetching them in one single block.

Finally, packrat parsing also increases performance with non context-free grammars such as $a^n b^n c^n$. The following grammar can be written as follows in Scala :

```
val AnBnCn : PackratParser[Any] =
  parseButDontEat(repMany1(a,b))~not(b)~>rep1(a)~repMany1(b,c)
```

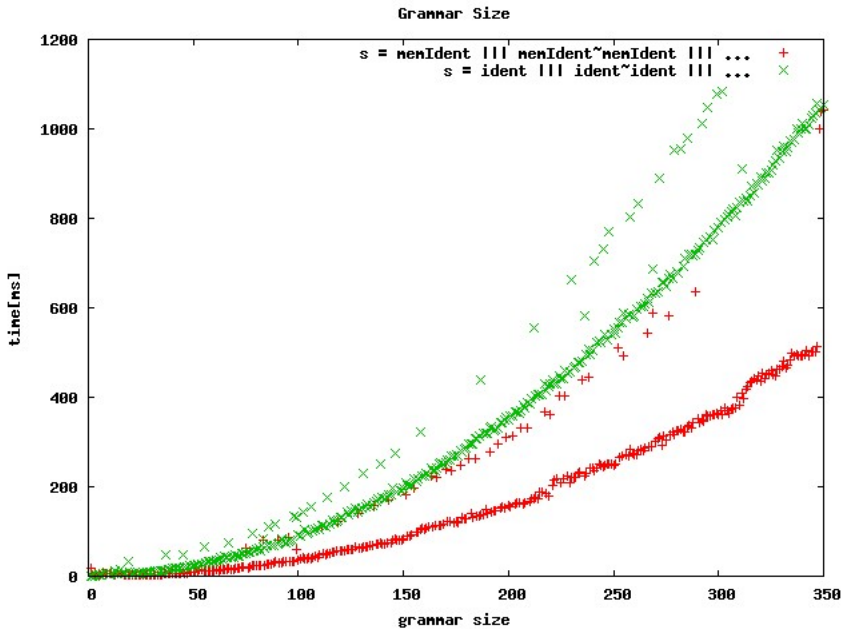

Figure 7.2: evaluating backtracking performances for packrat and non-packrat parsers

where `repMany1(a,b)~not(b)` parses $\{a^n b^n | n \geq 1\}$, `not(b)` ensuring that not `b`s follow. We can see here that both `a`s and `b`s will get parsed twice : having them memoized can decrease parse time. To make our tests more visualisable, we declared `a,b,c` as follows :

```
val a : PackratParser[Any] = numericLit^^{x => primFact(x.toInt)}
val b : PackratParser[Any] = memo("b")
val c : Parser[Any] = "c"
```

In `a`, we parse an integer and prime factorize it (a rather heavy operation) : some computations are usually done in general parsers before construction the AST, as this helps save memory space. Our inputs were randomly generated integers between 1 million and 6 million. We ran the tests for input sizes between 1 and 800 (number of elements of each type) : the results are shown in figure 7.3. We can notice the following :

- packrat parsing is indeed faster than normal parsing, in general twice as fast.
- the results are much more scattered than previous ones : this is mainly due to the random number generation (some numbers are harder to factorize than others).

All in all, we can conclude that the overhead of memoizing compared to the expressivity given by left-recursion is bearable, and that packrat parsing is much more efficient with backtracking inputs, and can be useful when performing complex operations on the parse results.

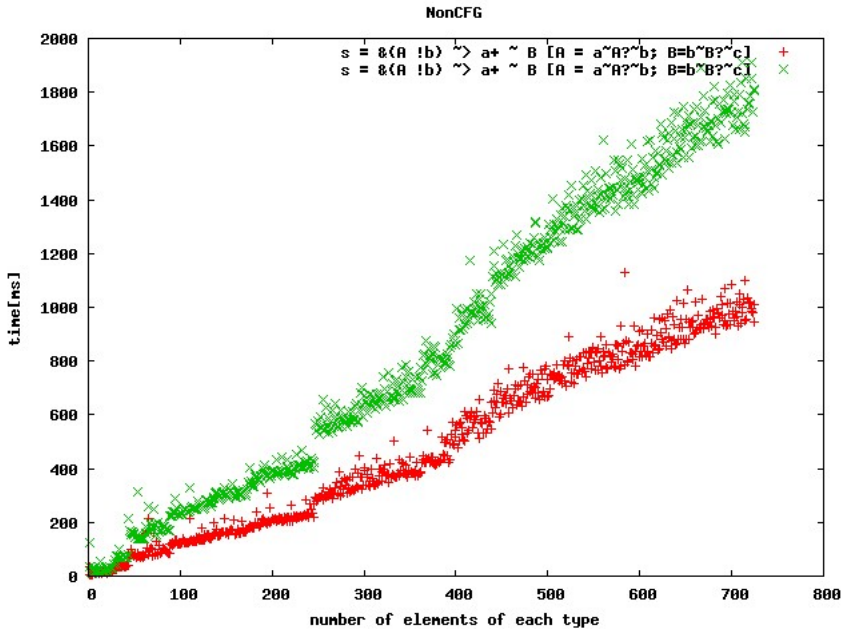

Figure 7.3: evaluating backtracking performances for packrat and non-packrat parsers

Chapter 8

Conclusion

Packrat Parsing is a parsing technique that accelerates the parsing time with the help of memoization. The original implementation idea by Ford was to use a programming language's inherent lazy evaluation (call-by-need) strategy to save parse results. We decided to use a dynamic lookup structure (cache) instead, because we needed to change an entry more than once. The reason behind this was to be able to write left-recursive grammars (direct and indirect), and to parse them. Our implementation was largely based on results from Warth et al., even though we had a few ideas ourselves on how we could integrate indirect left-recursion. The main goals behind our implementation choices were to leave things as easy as before for the programmer, and give him the choice on when to use packrat parsing or normal parsing.

On a performance level, we verified that packrat parsing is indeed faster on backtracking inputs. This is all the more critical when complex operations are performed directly on parse results (as in our non-CFG example). On the other hand, the expressivity given by left-recursion doesn't induce too much overhead on parse time ; it is however important not to make every parser memoizing, as that can cause overkill (as witnessed with the `leftRecAbusive` grammar).

Possible future work would be to implement packrat parsers for larger languages like Java, and see how the library performs with such languages. Left Recursion offers a much wider range of grammars that one can parse : research has also been done into accomodation of ambiguity in top-down parsing [3]. It would be interesting to see how such techniques can be implemented on top of the packrat parsers.

Chapter 9

Acknowledgements

I would first of all like to thank Prof. Martin Odersky, who at a very short notice suggested this exciting project. He was very helpful throughout, and especially tolerant about deadlines. I would then like to thank Tiark Rompf, who was instrumental in putting me on the right path throughout, and with whom weekly discussions were very fruitful. I would then like to thank Lukas whose test file skeletons were used for performance evaluation, and Philipp, who helped me with LaTeX typesettings for writing neat Scala code. Finally, I would like to thank all members of the LAMP for having had the patience to attend my project presentation.

Bibliography

[1] Bryan Ford. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, Sep 2002.

[2] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *Proceedings of the 2002 International Conference on Functional Programming*, Oct 2002.

[3] Richard A. Frost and Rahmatullah Hafiz. A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time. *SIGPLAN Not.*, 41(5):46–54, 2006.

[4] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *Java(TM) Language Specification (2nd Edition)*. Addison-Wesley Pub Co, June 2000.

[5] M. Odersky, L. Spoon, and B. Venners. Programming in Scala: a comprehensive step-by-step guide, 2007.

[6] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, March 2002.

[7] A. Warth, J.R. Douglass, and T. Millstein. Packrat parsers can support left recursion. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 103–110. ACM New York, NY, USA, 2008.