A Cross-Language Analysis of Parsing Using Automatic Memoization

Benjamin Cassell
University of Waterloo
becassel@uwaterloo.ca

Abstract

This paper demonstrates the key abilities of varying languages to support functional recursive descent parsing through automatic memoization. It begins by summarizing the most pertinent works that have dealt with using automatic memoization for the purposes of top-down parsing. Next, this paper provides an in-depth examination of a variety of popular modern day languages to determine their support for the easy implementation of an automatic memoization mechanism that is sufficiently powerful to aid recursive descent parsing. In particular, two primary types of solutions are examined throughout the various languages: Macro-based solutions, and first-class citizen function-based solutions in combination with mutation.

1 Introduction

Memoization has been an important technique in various computationally-intensive environments for decades. Michie, in one of the earliest published works to examine the role of memoization in computing, described it as a method akin to the way that humans learn, and extolled the virtues of using memoization to improve performance in computationally-intensive algorithms [19].

Due to the ability of memoization to prevent unnecessary work on input that has been previously examined, it is a critical component in the successful reduction of computational complexity for a variety of parsing techniques. This review chooses to focus on the benefits of memoization for functional top-down backtracking parsers, which can be greatly improved in a simple, natural way with the intelligent definition of memoized functions [20].

Interestingly, recursive descent parsers gain more than just improved performance from memoization. In fact, recursive descent parsers, when combined with the proper techniques and sufficient memoization abilities, are proven to be able to parse left-recursive languages [18]. This surprising increase in ability provided by memoization is an incredibly potent argument in favour of the technique.

It is important to consider, however, that manual implementation of memoization in a complex parser can consume a considerable amount of time and effort. Regardless of the potential benefits, memoization that is tedious for the programmer and results in unreadable and messy code is far less useful. This alone often serves as a barrier to the use of memoization in fields that require rapid development or prototyping [13].

This is where automatic memoization becomes useful: It allows for all the benefits of memoization with minimal coding effort and loss of maintainability. For parsing, this is particularly important as the grammar of the parser becomes highly complicated or convoluted (for example, the notoriously complicated grammar for the `C++` programming language). But this raises the question: Is automatic memoization available to coders in most situations? More importantly, is the automatic memoization powerful enough to support context-free parsing (in other words, does it properly memoize recursive functions)?

The contributions of this paper are as follows:

- A brief summary of the automatic memoization techniques introduced by Norvig [20] and Johnson [18] in their respective works, and how they benefit parsing.
- An examination of multiple popular modern programming languages to determine the possibility (and simplicity) of implementing general-purpose automatic memoization in them, which in turn allows for the usage of the parsing techniques demonstrated by Norvig and Johnson.
- A sample implementation of a subset of `Scheme` in `C#`, and a working implementation of Norvig and

Johnson's memoized parsing algorithms built on top of it.

- A discussion of the features between the languages that make automatic memoization possible or difficult.

The remainder of this work is structured as follows: Section 2 gives a brief overview of the Johnson and Norvig articles that serve as the basis for this review. Section 3 explores implementations of automatic memoization for use in parsing across a variety of popular modern languages, and section 4 discusses the findings of this paper about the availability of automatic memoization for context-free parsing. Section 5 discusses possible future work, and section 6 concludes the paper.

2 Background and Related Work

As mentioned in section 1, the concept of memoization in is not a new. The idea of memoizing the answers to well-known problems is ancient, and the application of memoization in computer science has been studied since at least 1968, when Michie published an article in Nature [19] examining memoization's usefulness within the context of machine learning algorithms. Since then, a great body of work has focused on using memoization to improve everything from thread scheduling [10] and proof systems [5] to multimedia applications [4].

More recently however, the concept of automatic memoization has become increasingly important. Automatic memoization can be thought of as the implementation of a higher level memoization paradigm, which allows a programmer to provide memoization benefits to any given function with minimal effort (usually by producing memoized versions of functions using a generic memoizer function, or by by marking a functions as memoizable using preprocessor statements or other language properties).

Automatic memoization is highly valuable to parsing as it allows for rapid prototyping without losing high performance. It allows programmers to focus their efforts on optimizing important special cases [20], and it also prevents code bases from becoming bloated and unmaintainable [13, 14]. For recursive descent parsing in particular, automatic memoization techniques allow functions to continue resembling grammar production rules, even when memoized. This prevents grammar rules from becoming cluttered with a blurred relationship between the recursive function and its associated grammar rule.

2.1 Parsing with Automatic Memoization

Although automatic memoization predates Peter Norvig's 1991 work (for example, with a sample produced by Abelson and Sussman six years earlier [3]), it was the first major work to examine automatic memoization within the scope of parsing context-free grammars [20]. Norvig demonstrates the complexities associated with providing automatic memoization for functional recursive descent parsing: Because grammar rules can be recursive, simply producing a lookup-function generator can result in improperly memoized recursive functions.

Norvig provides an implementation of automatically memoized parsing in `Lisp`, using the `setf` function to circumvent issues with recursive function calls. Norvig adds a generic memoizer call on top of functions representing production rules in a sample grammar. Calls to these rules are re-bound to the memoized versions, even in recursive (and mutually recursive) cases. Norvig additionally demonstrates the utility of making the automatic memoization function flexible enough to allow different types of hashing and equality testing on arguments, with a simple change to the hashing and equality algorithms changing the time complexity of a sample backtracking parser from quartic to cubic.

Norvig also shows that a similar implementation for automatic memoization would be effective for parsing in `Scheme`, either through the use of the `set!` function or through macros. Finally, Norvig demonstrates that other languages, for instance `Pascal`, could implement automatic memoization through constructs such as a `memo` keyword (given that the language's parser can be changed). This review contends that this method is too heavy-handed to be useful. As such, we concentrate on demonstrating which languages support automatic memoization through already-provided features and not through explicitly modifying the language core.

2.2 Extending Memoized Parsing

Most typical implementations of recursive descent parsing suffer from similar problems, and foremost among these is the inability of minimally-implemented top-down parsers to recognize left-recursive grammars. Given a left-recursive grammar, naive recursive descent parsers fail to terminate and, without proper supplementation, automatic memoization does nothing to change this. In his 1995 work, Johnson shows how to solve this problem using automatic memoization in combination with continuation-passing style (CPS) [18].

CPS is employed by Johnson to reverse the flow of computation in a top-down parser, resulting in values that travel downwards through the computation instead of being pulled upwards from the bottom. Additionally, Johnson memoizes on arguments to the parser and associates them with lists of continuations. This creates a system in which no grammar function is evaluated on

the same arguments more than once (subsequent evaluations are passed forwards to the continuations memoized in the lookup table). In the case of left recursion, this implies that the recursive cycle is broken after at most one recursive call, preventing non-termination.

Johnson provides a full implementation of both the automatic memoizer and CPS-based parser in `Scheme`, affirming Norvig's previous conjectures about its feasibility [20]. Johnson requires some manipulation to be able to define mutually-recursive production rules (due to what he refers to as a vacuous deficiency in the `Scheme` specification), and this highlights an important limitation on languages that do not allow properly mutual-recursive definitions: Such languages need to support some combination of function predeclarations or first-class citizen functions with lambda expressions to directly allow for easy implementation of recursive descent parsers. Other more inconvenient methods may be available to languages without these features, for example defining a set of values up front and operating over interpretations of said values, but this defies some of the spirit of functional recursive descent parsing, which is to have functions that resemble the production rules of the grammar.

3 Exploration of Automatic Memoization

This section examines implementations of automatic memoization in various popular modern programming languages. It highlights the techniques used to achieve automatic memoization, whether or not it is sufficient for use in functional context-free parsing, and the benefits and difficulties of the implementation or language. For every language listed here, unless otherwise noted, a functioning code sample is included with the source code that accompanies this paper.

3.1 `C#`

The most comprehensive examination of automatic memoization performed by this paper was conducted using the `C#` programming language. This, and all other examinations of `.NET` languages (`C++.NET` and `VB.NET`), were conducted using the `.NET` 4.0 language SDK and run-time environment.

Automatic memoization in `C#` has been attempted by others before with success, for example by Wes Dyer in 2007 [11]. Like these previous implementations, the `C#` version of automatic memoization employed by this work uses the `.NET` `Func` templated type, which is a first-class representation of a function (albeit one with specifically typed arguments and a specifically typed return value). The `Memoize` function takes in a similarly templated `Func` object, as well as an object implementing the `IDictionary` interface to use as a memo (allowing for different kinds of caching algorithms, including custom ones). The function returns a lambda expression representing the memoized version of the input function. The implementation of this function can be seen in listing 1.

Of note is that the function must check whether or not the input key is `null`, as `IDictionary` types in `.NET` languages do not operate on `null` keys. Furthermore, this function only operates on functions that take a single input for ease of implementation. There are, however, several methods that can be used to extend the number of parameters accepted by this function. These methods are are discussed in section 4.3. The `this` keyword qualifying the function `fn` is syntactic sugar, and defines `Memoize` as an extension method of the type `Func`. With this definition, all objects of type `Func` accepting one value and returning one value can invoke `Memoize` as if it were a method belonging to their class definition.

Having also defined a templated `ConsCell` data type representing two values joined by a `cons` function (as in `Lisp` or `Scheme`, based on an implementation by Dustin Campbell [9]), as well as the higher order functions implemented by Norvig and Johnson in their works, a `C#` parsing function representing the production rule (S $\rightarrow$ NP VP) in Johnson's example grammar would resemble the sample in listing 2. The implementation of the higher order functions `alt` and `seq` function can be seen in listings 3 and 4. Note that the `vacuous` macro from the Johnson work is explicitly in-lined, as `C#` does not support macro expansion. Furthermore, as with `Memoize`, the `seq` function is defined as an extension method.

The full implementation of the Norvig parser can be seen in the `C#` source code that accompanies this work and is available upon request. It also includes a fully functioning example of Johnson's CPS-based parser using automatic memoization written in `C#`. It is not included in this paper due to space constraints.

3.2 `Lisp`, `Scheme` and `Racket`

As described in section 2, automatic memoization that supports context-free parsing is readily available in both `Lisp` and `Scheme`, as shown by Norvig [20] and Johnson [18]. Sample code is provided in `Racket` using Johnson's `Scheme` method (which is textually identical as `Racket` is just a super-set of the older `PLT Scheme`). Interestingly, `Lisp`, `Scheme` and `Racket` all support macro expansion, meaning an alternative (but likely inferior) automatic memoization solution that uses macros and would be able to support context-free parsing is viable.

3.3 `Visual Basic .NET`

Because `VB.NET` is a completely Microsoft-defined language (unlike `C++.NET`), it is able to work natively with

```
static Func<T1, T2> Memoize<T1, T2>
    (this Func<T1, T2> fn, IDictionary<T1, T2> memo) {
  return key => {
    if (key == null)
      return fn(key);
    T2 value;
    if (memo.TryGetValue(key, out value))
      return value;
    value = fn(key);
    memo.Add(key, value);
    return value;
  };
}
```

Listing 1: Automatic Memoization in C#

```
static Func<ConsCell<object>, ConsCell<object>> S
  = Memoize<ConsCell<object>, ConsCell<object>>((input => NP.Seq(VP)(input)));
```

Listing 2: Sample Grammar Rule (S → NP VP)

```
static Func<ConsCell<object>, ConsCell<object>> Alt(this Func<ConsCell<object
    >, ConsCell<object>> a, Func<ConsCell<object>, ConsCell<object>> b)
{
  return input =>
  {
    return a(input).Union(b(input));
  };
}
```

Listing 3: Sample Higher Order `alt` Function

```
static Func<ConsCell<object>, ConsCell<object>> Seq(this Func<ConsCell<object
    >, ConsCell<object>> a, Func<ConsCell<object>, ConsCell<object>> b)
{
  return input =>
  {
    return Reduce<ConsCell<object>, ConsCell<object>>(Union, null, b.Map(a(
        input)));
  };
}
```

Listing 4: Sample Higher Order `seq` Function

most of the complex .NET types that C# has access to. This makes automatic memoization supporting context-free parsing very straight-forward to implement, by directly translating the C# version into VB.NET line by line. This translation can be seen in listing 5, as well as a sample application on the Fibonacci sequence. It has all the same associated benefits and drawbacks as the C# version, and can also be turned into an extension method if desired with a small addition.

3.4 C++.NET and C++11

Although it is possible to create a version of automatic memoization that can support context-free parsing in C++.NET using .NET constructs, it would be much more useful if there was a generic version that could also be used without any changes inside C++11. This code would be much more portable and thus applicable across a much greater variety of systems. Happily, such a compromise exists. The sample in listing 6 shows a C++ memoize function that properly handles the automatic memoization of recursive functions in both environments.

Like the C# implementation, the sample given here is templated and accepts a single-argument function, but could be easily modified using the methods in section 4.3 to accept functions with more generic amounts of arguments. It could also be modified to accept a collection-style object like the .NET automatic memoization implementations, instead of simply using the stl map object. Because C++ supports macro expansion through the C preprocessor, a more limited alternative solution could also be constructed using macros, in a manner similar to the solution demonstrated in section 3.8.

3.5 Python

In Python, automatic memoization that handles recursive functions can be provided in a variety of ways. One particularly clean way, as suggested in various online resources [17], is to create a memoized class which, when initialized, attaches an empty associative list and an input function to itself. When the object is invoked, it performs the appropriate memoized lookup. A sample of this implementation style, which is able to handle arbitrary parameters and types, is given in 7.

3.6 JavaScript

JavaScript is often an unfairly-maligned language among programmers. Implementations of automatic memoization that can handle context-free parsing in JavaScript are surprisingly simple, elegant and easy to understand. The implementation of memoize given in listing 8 is an example provided by Colin Ihrig [15]. It is functionally equivalent to the C# example, but as with the Python example can support functions with arbitrary numbers and types of arguments.

3.7 Java

Whereas most other languages examined in this work succeeded in some way or another, Java falls short, unfortunately. Although a solution for direct automatic memoization exists, as shown by Tom White's 2003 implementation [24], it does not properly memoize recursive functions. The technique employed by White is to create an memoization class, which has an invocation handler for other interfaces. The programmer subsequently creates an object which adheres to a desired interface, and memoizes it by passing it to the memoization class. Any function calls to interface-bound members of the returned object are then routed through the memoization class first.

Because only the first call to the object passes through the memoized layer, recursive calls and calls to other functions are not properly memoized. To make matters worse, White's solution is likely the best automatic memoization available for Java in its current release. Java does not treat functions as first class citizens, does not offer lambda expressions, and does not natively support macro expansion. Because of these limitations, there is likely no easily accessible Java solution for true automatic memoization that is appropriate for use in context-free parsing. Of course, a manual solution is always possible, and probably not burdensome enough to warrant avoiding Java as a language if other circumstances support its usage.

3.8 C

Automatic memoization in C is tricky. While C does provide function pointers, they are little more than the memory addresses that functions live at post-compilation and provide little of use in terms of automatic memoization. What C does provide, on the other hand, is the powerful C preprocessor, which can be used to provide automatic memoization macros as demonstrated in listing 9. This is a very basic but functional example, and something more user-friendly and generic could almost certainly be created given more time and thought. For the sake of brevity, the hash table implementation is excluded from this document.

The memoize macro takes in several values, the first of which is a hash table initialization function. This function in turn accepts a pointer to the hash table (allowing it to be initialized on the stack or allocating new memory with malloc if the pointer is null), as well as the

```
Public Function Memoize(Of T1, T2)(fn As Func(Of T1, T2), memo As IDictionary(
    Of T1, T2)) As Func(Of T1, T2)
  Return (Function(key As T1) As T2
    If (IsNothing(key)) Then
      Return fn(key)
    End If
    Dim value As T2
    If (memo.TryGetValue(key, value)) Then
      Return value
    End If
    value = fn(key)
    memo.Add(key, value)
    Return value
  End Function)
End Function

Dim Fib As Func(Of UInteger, ULong) = Memoize(
  Function(n As UInteger) As ULong
    If (n < 2) Then
      Return n
    End If
    Return Fib(n - 2) + Fib(n - 1)
  End Function)
```

Listing 5: Automatic Memoization in VB.NET

```
template <typename ReturnType, typename ArgType>
function<ReturnType (ArgType)> memoize(function<ReturnType (ArgType)> fn)
{
  map<ArgType, ReturnType>* memo = new map<ArgType, ReturnType>();
  return ([=] (ArgType key) -> ReturnType {
    if (memo->find(key) == memo->end())
      (*memo)[key] = fn(key);
    return (*memo)[key];
  });
}

function<unsigned long (unsigned)> fib = memoize<unsigned long, unsigned>([=]
    (unsigned n) { return ((n < 2) ? n : fib(n - 2) + fib(n - 1)); });
```

Listing 6: Automatic Memoization in C++

```python
class Memoize:
  def __init__(self, fn):
    self.fn = fn
    self.memo = {}
  def __call__(self, *args):
    if not args in self.memo:
      self.memo[args] = self.fn(*args)
    return self.memo[args]

def Fib(n):
  if n < 2: return n
  return Fib(n - 2) + Fib(n - 1)

Fib = Memoize(Fib)
```

Listing 7: Automatic Memoization in Python

```javascript
function memoize(func)
{
  var memo = {};
  var slice = Array.prototype.slice;
  return function()
  {
    var args = slice.call(arguments);
    if (args in memo)
      return memo[args];
    else
      return (memo[args] = func.apply(this, args));
  };
}

function fib(n)
{
  if (n < 2)
    return n;
  return fib(n - 2) + fib(n - 1);
}
fib = memoize(fib);
```

Listing 8: Automatic Memoization in JavaScript

byte size and number of keys that will be stored in the hash table. The next arguments accepted by `memoize` are hash table functions to determine if a key exists in the hash table and to return a value based on its key. It further accepts the name of the key, the type of value being returned by the function, and the number of values the hash table should retain. When successful, the `memoize` macro statically allocates a hash table pointer to heap-initialized memory for the function being memoized. Calls to the function are then checked for hash table residence before the remainder of the function is executed.

Because the `C` implementation relies on macros, it also requires the use of an additional `memo_return` macro, which replaces any `return` statements that should have their values memoized. The `memo_return` macro accepts the function used to set values in the hash table, the name of the key in the function, and the expression whose value should be returned. This macro expands to the appropriate statements that hash and return the value of the expression. Although the `memoize` and `memo_return` macros sound complicated, in practice they are very easy to use and provide the full benefits of automatic memoization.

3.9 `Haskell`

Due to time constraints, this work has not performed an in-depth examination of the `Haskell` language for its compatibility with automatic memoization, and thus no accompanying source code is provided. That said, there is no reason to believe that `Haskell` should not be able to support automatic memoization which is appropriate for use in context-free parsing, as the `Template Haskell` language extension provides even more powerful and expressive macro support than the `C` preprocessor.

Other sources have claimed moderate success using `Template Haskell` to create automatically memoized functions in `Haskell` that support recursion [8], and although these results have not been independently confirmed by this work, the scope of the claims being made are entirely reasonable. In the cited example, Mike Burrell reports that a first pass at an automatic memoization macro that supports recursive functions works as intended, but consumes far too much memory. As such, further work would be needed to ensure that memory use does not become a concern when automatically memoizing functions for the purposes of context-free parsing in `Haskell`.

4 Discussion

Almost all of the languages examined in this work demonstrate similar abilities which allow for the types of declarations necessary to implement automatic memoization and recursive descent parsers with functions that directly parallel the production rules of a grammar. These language features, roughly summarized are:

- Function predilection's or lambda expressions, allowing for the definition of mutually-recursive functions representing grammar rules.
- First class functions and the ability to re-bind a label to a newly generated function or value (mutation).
- If mutation or higher-order functions are lacking, sufficient preprocessor power to be able to add memoization boilerplate code wrapping the contents of a function.

Despite most most of the discussed languages sharing some or all of these features, there are a few language-specific properties that render several of the explored languages less attractive for usage in recursive descent parsing.

4.1 Tail Call Optimization

One particularly pressing problem is the issue of tail and sibling call optimization. A successful parse using a recursive descent parser can require a significant amount of recursive calls to complete (CPS style, for example, can potentially recurse without unwinding any stack frames until final completion), and if an environment is incapable of properly providing tail call optimization this can drastically limit the usability of recursive descent parsers implemented in that environment. That said, compilers may also harm a memoized application if they eliminate tail recursion altogether: If a tail-recursive call is improperly optimized out (for example, turned into a modified loop), this could prevent an application from taking advantage of memoized results.

Most of the languages examined in this work have either poor or no support for tail call optimization. While languages like `Scheme` and `Haskell` provide explicit, well-defined and predictable tail call optimization, many languages like `Java` do not. For directorial, historical or other reasons, some languages do not support tail call optimization and never will without a significant change in development focus and principals. `Python`'s creator, as an example, is known to be particularly resistant to the idea of adding tail call optimization to `Python`'s language specification [22, 23].

Potentially even worse than knowing that tail call optimization will not happen in a memoization-friendly way is being unsure of whether or not it will. Although many implementations of `C` allow for some form of tail or sibling call optimization, in `gcc` (for example) this decision

```c
#define MAX_FIB 16384 // The maximum size to use for the hash table

#define memoize(init_fn, contains_fn, get_fn, key_name, ret_type,
    num_memo_vals) \
  static hashtable* __memo__ = NULL; \
  ret_type __memo_ret__; \
  if (__memo__ == NULL) \
    __memo__ = init_fn(NULL, sizeof(ret_type), num_memo_vals); \
  if (contains_fn(__memo__, key_name)) \
    return get_fn(__memo__, key_name); \

#define memo_return(set_fn, key_name, ret_expr) \
  __memo_ret__ = ret_expr; \
  set_fn(__memo__, key_name, __memo_ret__); \
  return __memo_ret__;

unsigned long fib(unsigned int n)
{
  memoize(hashtable_init, hashtable_contains, hashtable_get,
    n, unsigned long, MAX_FIB);
  if (n < 2)
    return n;
  memo_return(hashtable_set, n, fib(n - 2) + fib(n - 1));
}
```

Listing 9: Automatic Memoization in C

is left to the compiler (at levels of optimization O2 and higher) [1, 2] which can result in highly variable optimization decisions.

```c
Func<int, long> Fib = null;
Fib = (n => n > 1 ?
  Fib(n - 1) + Fib(n - 2) : n);
```

Listing 10: Stack Overflow in C#

```c
Func<int, long> Fib = null;
Fib = (n => n > 1 ?
  Fib(n - 2) + Fib(n - 1) : n);
```

Listing 11: Insignificant Change, Significant Gains

Managed environments suffer as well. All .NET languages compile into an intermediate language called MSIL which is just-in-time (JIT) compiled by the .NET run-time environment. None of these MSIL compilations include the .tail instruction that induces tail call optimization [7]. Worse still, the JIT compiler for .NET programs may or may not dynamically add .tail instructions at program run-time, ensuring that programmers do not know whether or not their programs will overflow the stack. The code sample in listing 10 is a very simple C# lambda expression that can cause stack overflows even when memoized. Here, assuming that the JIT compiler will perform the appropriate optimization leads to failure cases.

The sample code in listing 10 was run for about 10,655 recurses before it crashed. This number was not static, and slightly different values were be obtained with varying stability based on the performance of the JIT compiler. This sample further highlights an important aspect of programming recursive functions in languages that do not provide adequate tail call optimization: Simply reversing the order that the recursive calls to Fib are made in (shown in listing 11) reduces the use of the call stack by half for equivalent inputs. It can be generalized from this that recursively programmed functions should be intelligently catered towards early and shallow termination when possible, depending on environmental limitations.

Despite the grim prognosis for tail call optimization in a large subset of languages, some other languages are improving. JavaScript, which is becoming substantially more popular thanks to the adoption of HTML5, traditionally did not support tail call optimization. With the introduction of the newest JavaScript standard however, this is reported to be changing [6]. Even with a complete lack of tail call optimization, it is still possible to provide the illusion of optimized tail recursion through the use of trampolining in languages that support higher-order functions [16]. Other techniques can also allow

for the illusion of trampolining in languages that do not cleanly support first-class citizen functions (for example, with function pointers, `setjmp` and `longjmp` in `C`).

4.2 Macro Support

Lack of macro support can limit development of simple automatic memoization, or at least make it more difficult. Currently, `VB.NET` does not support macro expansion, and neither does `C#` by design choice [12]. Regardless, both languages support automatic memoization through their implementations of first class functions and lambda expressions. Other languages, like `C` and `Haskell`, have a much easier time supporting automatic memoization through the use of macro expansion than through any other means.

As mentioned in section 3.7, `Java` currently lacks support for functions as first class citizens. Because it also lacks macro expansion capabilities, `Java` is incapable of easily providing true automatic memoization support, and is perhaps, as a result, an inappropriate language choice for recursive descent parsing. This is due to change with the release of `Java 8`, however, which is expected to introduce both functions as first-class citizens and lambda expressions [21].

One potential solution for macro-less languages is to write macros using a text-based macro implementation from another language, passing any necessary source code through the external macro handler before compiling or interpreting. This, however, is clunky and inconvenient - a poor substitute for language-native support of paradigms enabling automatic memoization.

4.3 Variadic Argument Support

Although variadic argument support - the ability for functions in a language to accept and work on variable amounts and types of parameters - is not required for enabling context-free parsing through automatic memoization, it is highly convenient and prevents the programmer from having to define multiple memoization functions for varying numbers and types of arguments. `VB.NET` `C++.NET`, `C++11` and `C#`, the languages whose examples in this paper did not use variadic arguments, did not do so due to lack of capability. All of these languages support some form of variadic arguments, as does `Java`. `C#` and `VB.NET` both have the ability to operate over parameter lists (`params` and `ParamArray` respectively), and accepting a parameter list of type `object` would allow a user to pass in arbitrary numbers of arguments of any type. `C++.NET` supports similar functionality through templated array lists in function definitions. Lastly, `C++11` supports a generic `Args...` type, which is nearly identical to the `.NET` parameter lists in practice.

An alternative strategy for a language that does not support variadic arguments can come in different forms: In an object-oriented language, a wrapper class containing variable amounts of elements can be used as the sole parameter to functions (for example, the `tuple` type available in `C++` and all `.NET` languages). In a language where all types inherit from a common `object` type, collections containing the `object` type can be used to emulate this as well. Templating the memoization function is also an effective strategy for accepting generic argument types. Finally, function overloading can also be used to create multiple memoization functions accepting different amounts of parameters up to a sensible finite limit. These overloaded functions can be used to simulate true variadic functions for most useful purposes.

4.4 Limitations of Automatic Memoization

It should be noted that there are several important limitations to automatic memoization. First and foremost, automatic memoization is only useful if the results of a function are deterministic based on input, or failing this, if results that are simply ”good enough” are sufficient. Automatic memoization on procedures that can return varying values (for example, based on user input) are most likely not useful. See section 5 for a discussion of how automatic memoization can be taken advantage of in a system where values are acceptable if they are simply ”good enough”.

Automatic memoization can also incur penalties in terms of both computation time and space if used ineffectively. A poor caching algorithm can result in inefficient lookups and slow performance. Mistakes in how items are cached can result in degenerate behaviour (in some languages, for example, mistakenly indexing on the addresses of objects instead of a stringified representation of an arbitrary object will cause all lookups to result in cache misses). Furthermore, automatic memoization can actually slow down fast-running functions and should not be applied without careful consideration.

Finally, unless there is a way to flush an automatically memoized function’s cache, it can continue to grow and consume space indefinitely. This is easy enough to circumvent, by forcing the programmer to include an externally referenced cache object when memoizing a function, but also implies that the burden of knowing how and when to flush the cache is on the programmer. A programmer must fully examine whether or not it is useful to use memory by memoizing results: Ignoring the closed-form of the algorithm, a `Fib` function that will only ever be used to list terms in successive order should not be memoized, as it can be implemented using constant space. On the other hand, if the `Fib` function is to be accessed arbitrarily, memoization will provide signif-

icant wins.

5 Future Work

This research has opened up several interesting possibilities for me in future work. The first does not stem directly from automatic memoization for context-free parsing but rather from the mini-recreation of `Scheme` in `C#`. The recreation used for the purposes of this paper was very ad hoc and limited. It was also not as well-engineered as it could have been. Having learned some important lessons during the implementation of this project, I would like to re-examine creating an elegant and intelligent solution that better represents `Scheme` inside `C#` and that simplifies ad-hoc development and prototyping.

A second possibility for future work, which is directly related to context-free parsing, is adding automatic parallelization to the Norvig and Johnson parsers. With the provided `C#` implementation, it should be fairly straightforward to parallelize certain aspects of parsing. In particular, the `alt` function could spawn two threads which simultaneously handle both parsing paths. A thread that fails or succeeds could use futures or raise events to inform other thread groups of acquired information. In the case of constructing unions (as is required for both `seq` and `alt`), the unions could be built in parallel by adding results to thread-safe collections.

The `.NET` environment makes integrating parallelization into automatically memoized context-free parsers especially easy: The implementation of automatic memoization provided by this work in `C#` already allows for a generic object adhering to the `IDictionary` interface to be used for memoization. Because `.NET` provides a thread-safe dictionary type (`ConcurrentDictionary`) that implements this interface, making the memoization thread-safe and thus parallelization-compatible is as easy as passing an instance of `ConcurrentDictionary` in when memoizing functions. Automatically parallelizing the parsing techniques in this implementation would be a large step towards making it as powerful and useful as similar parallelized parsing libraries available in languages like `Racket`.

One final possibility for future work is examining the applications of automatic memoization in cloud and internet-based computing. As previously mentioned in section 4, when results that are simply "good enough" are considered acceptable, programmers gain lenience on the functions and procedures that they may apply memoization to. One could imagine a cloud-based ad serving system in which a programmer could memoize ad information based on consumer preferences. In such a system, the advertisement data could be stored along with datestamps, and the automatic memoization function could be modified to take the staleness of the returned information into account when serving it. If the cached information is considered sufficiently fresh or otherwise "acceptable", it would be returned. Otherwise, the cached copy would be updated to a value fetched from a remote location.

6 Conclusion

Recursive descent parsers are straight-forward, effective, and have the added benefit of being defined such that their functions closely resemble the grammar rules that they represent. Unfortunately, typical recursive descent parsers can be very slow, and in naive implementations they do not terminate on left-recursive grammars. Automatic memoization can fix both of these issues, drastically improving performance and providing termination on left-recursive grammars using Johnson's 1995 CPS techniques.

It has been shown in this work that a significant amount of modern languages support adequate facilities to provide automatic memoization for context-free parsing (primarily, the ability to memoize recursive functions properly). These languages include `C`, `C#`, `JavaScript`, `VB.NET`, `Racket`, `Python`, and all modern versions of `C++`. Although `Haskell` was not directly examined, there is sufficient evidence to reasonably expect it to behave similarly. These languages achieve this automatic memoization in two primary ways, either through lambda expressions and first-class citizen functions combined with mutation, or through the use of macro expansion.

Unfortunately, some languages, even though they may support some form of automatic memoization, cannot currently support automatic memoization that handles recursive descent parsing. `Java` is one such language, although it should be updated in the near future to have full support for recursive automatic memoization. Furthermore, there are scenarios where automatic memoization can be harmful if used carelessly, and as such it should be approached with careful consideration.

Automatic memoization is a fascinating topic, with a wide array of applications in and beyond context-free parsing. It is heartening to see that the kinds of techniques it requires are becoming a regular focus in most programming languages. With potential applications from this work in terms of cloud-based programming as well as the automatic parallelization of context-free parsing, one hopes that automatic memoization continues to be increasingly considered as an appropriate and effective method for improving the performance and power of algorithms and applications.

7 Acknowledgements

I would like to thank Professor Prabhakar Ragde in addition to my supervisor Tim Brecht, as well as my other colleagues and friends at the University of Waterloo. I would additionally like to thank all the authors whose code samples and insights acted as the base for this work. Without their contributions this would not have been possible.

References

[1] GCC: Options that control optimization. http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[2] Tail recursion. http://c2.com/cgi/wiki?TailRecursionhttp://c2.com/cgi/wiki?TailRecursion, January 2012.

[3] ABELSON, H., AND SUSSMAN, G. J. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, 1985.

[4] ALVAREZ, C., CORBAL, J., AND VALERO, M. Fuzzy memoization for floating-point multimedia applications. *IEEE Transactions on Computers 54*, 7 (2005), 922–927.

[5] BEAME, P., IMPAGLIAZZO, R., PITASSI, T., AND SEGERLIND, N. Memoization and DPLL: Formula caching proof systems. In *Proceedings of 18th IEEE Annual Conference on Computational Complexity* (July 2003).

[6] BENVIE, B. Javascript (ES6) has proper tail calls. http://bbenvie.com/articles/2013-01-06/JavaScript-ES6-Has-Tail-Call-Optimization, January 2013.

[7] BROMAN, D. Enter, leave, tailcall hooks part 2: Tall tales of tail calls. http://blogs.msdn.com/b/davbr/archive/2007/06/20/enter-leave-tailcall-hooks-part-2-tall-tales-of-tail-calls.aspx, June 2007.

[8] BURRELL, M. Memoizing haskell. http://www.onjava.com/pub/a/onjava/2003/08/20/memoization.html, April 2007.

[9] CAMPBELL, D. Building data out of thin air. http://diditwith.net/2008/01/01/BuildingDataOutOfThinAir.aspx, January 2008.

[10] CUI, H., WU, J., CHE TSAI, C., AND YANG, J. Stable deterministic multithreading through schedule memoization. In *Proceedings of OSDI 2010* (October 2010).

[11] DYER, W. Function memoization. http://blogs.msdn.com/b/wesdyer/archive/2007/01/26/function-memoization.aspx, January 2007.

[12] GUNNERSON, E. Why doesn't C# support #define macros? http://blogs.msdn.com/b/csharpfaq/archive/2004/03/09/86979.aspx, March 2004.

[13] HALL, M., AND MAYFIELD, J. Improving the performance of AI software: Payoffs and pitfalls in using automatic memoization. In *Proceedings of Sixth International Symposium on Artificial Intelligence* (Monterrey, Mexico, September 1993).

[14] HALL, M., AND MCNAMEE, J. P. Improving software performance with automatic memoization. *Johns Hopkins APL Technical Digest 18*, 2 (1997), 254–260.

[15] IHRIG, C. Implementing memoization in javascript. http://www.sitepoint.com/implementing-memoization-in-javascript/, August 2012.

[16] JACK, S. Bouncing on your tail. http://blog.functionalfun.net/2008/04/bouncing-on-your-tail.html, April 2008.

[17] JASON. Re: What is memoization and how can i use it in python? http://stackoverflow.com/a/1988826, July 2010.

[18] JOHNSON, M. Memoization in top-down parsing. *Computational Linguistics 21*, 3 (1995), 405–417.

[19] MICHIE, D. Memo functions and machine learning. *Nature 218* (1968), 19–22.

[20] NORVIG, P. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics 17*, 1 (1991), 91–98.

[21] ORACLE. Lambda expressions. http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html, October 2013.

[22] VAN ROSSUM, G. Final words on tail calls. http://neopythonic.blogspot.com.au/2009/04/final-words-on-tail-calls.html, April 2009.

[23] VAN ROSSUM, G. Tail recursion elimination. http://neopythonic.blogspot.com.au/2009/04/tail-recursion-elimination.html, April 2009.

[24] WHITE, T. Memoization in java using dynamic proxy classes. http://www.onjava.com/pub/a/onjava/2003/08/20/memoization.html, August 2003.