19th International Computer Science and Engineering Conference (ICSEC)
Chiang Mai, Thailand, 23-26 November, 2015

A Parser Generator Using the Grammar Flow Graph

Pakawat Nakwijit and Paruj Ratanaworabhan
Faculty of Engineering, Department of Computer Engineering, Kasetsart University

email: pakawat.n@ku.th and paruj.r@ku.ac.th

***Abstract*—Recently, a new way to represent context-free grammars (CFG) has been put forward. The representation uses a directed graph called the Grammar Flow Graph (GFG). The GFG provides a framework to parse all context-free languages (CFL) based on Earley's algorithm that is easier to understand than the original Earley's presentation. In addition, the GFG connects two seemingly distant concepts for regular language parsing and parsing of CFLs. It shows that simulating all the moves through a non-deterministic finite-state automaton (NFA) can be generalized to moving along all appropriate GFG paths. This paper introduces a GFG parser generator whose functionalities are equivalent to popular parser generators like Yacc or Bison. However, it works on all CFGs - ambiguous or unambiguous. Our parser generator takes a grammar file that represents strings in CFL as an input and outputs a parser program to parse those strings. We evaluate it against two other parser generators, one based on the original Earley's representation and the other based on the Cocke–Younger–Kasami (CYK) algorithm. The result indicates that the performance of our parser generator is on a par with that based on the original Earley's scheme and superior to the CYK's parser generator. The code for our parser generator can be downloaded from the following link:**

https://bitbucket.org/kramatk/earleyparser.

Keywords—Parser generator; Grammar Flow Graph; Earley's algorithm.

I. Introduction

In this work, we are interested in building and evaluating a parser generator using a new framework called the Grammar Flow Graph (GFG). Such a parser generator can handle any context-free language (CFL) specified with either ambiguous or unambiguous context-free grammar (CFG). We compare our parser generator against those based on two existing algorithms, the original Earley's scheme and the Cocke–Younger–Kasami (CYK) algorithm. Some popular parser generators like Yacc or Bison are a special case of our parser generator as they only handle a subset of CFG, namely, SLR, LALR(1), or LR(1). These specialized grammars are already expressive enough to specify programming language syntax, hence, these parser generators are mostly used to generate parsers for computer programming languages. These parsers possess an obvious advantage in that they run in O(N) time where N is the size of the input string in a CFL. This quickens the front-end compilation process significantly. In contrast, our generated parser, although more generalized, runs much slower in O(N^3) time. While our parser generator may not be suitable for generating parsers for computer languages, it finds its niche in the field of natural language processing where the grammars are more complex and can be highly unambiguous.

A. The Grammar Flow Graph (GFG)

GFG is a direct graph that represents a CFG. Each production in the grammar is reformulated to a node which consists of a production rule and progression of its production denoted by (dot) **.** Pingali and Bilardi present and prove in [1] that any context-free grammar G can be transformed into a corresponding GFG in O(|G|) space and time. |G| denotes the size of the grammar. They also classify nodes in GFG as shown in Table 1. Intuitively, GFG is an automaton for CFL similar to a non-deterministic finite-state automaton (NFA) for regular language. So, with the GFG framework, parsing CFL can be viewed as an extension to simulating moves through an NFA to handle regular language. Some additional stack bookkeeping needs to be incorporated in to generalize this NFA scheme to handle CFL.

Table. 1. Classification of GFG nodes. A,B are non-terminal symbols; t is a terminal symbol; α,β are any strings that are constructed by the non-terminal and terminal symbols according to the rules of the grammar.

Node Type	Description
START	Node labeled .A
END	Node labeled A.
CALL	Node labeled A -> α .B β
RETURN	Node labeled A -> α B. β
SCAN	Node labeled A -> α .t β

We will now briefly explain how parsing is done with the GFG by means of an example. Consider the following grammar:

s -> np vp
np -> DET NOUN
vp -> VERB np

The three non-terminals are ***s***, ***np***, and ***vp***, while the three terminals are ***DET***, ***NOUN***, and ***VERB***. The associated GFG for this grammar is shown in Fig. 1. There are three START nodes denoted by ***.s***, ***.np***, and ***.vp*** and three END nodes denoted by ***s.***, ***np.***, and ***vp.***, accordingly. The CALL nodes have an arrow pointing out whereas the RETURN nodes have an arrow pointing in. For example, ***s -> .np vp*** is a CALL node, while ***s -> np. vp*** is a RETURN node. The rest are SCAN nodes.

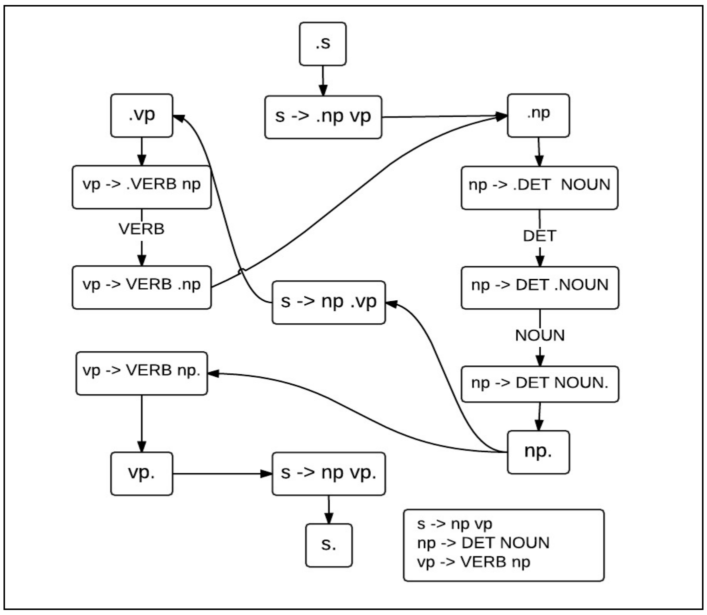

Fig. 1. An example of the Grammar Flow Graph (GFG).

Consider the following input string to be parsed:

"a cat eats a fish"

Let's assume that 'a' is a DET terminal, 'cat' and 'fish' are NOUN terminals, and 'eats' is a VERB terminal. We start at the START node ***.s*** in the GFG and move along its transitional edge to a CALL node labeled ***s -> .np vp***. Next, we enter a branch - another START node - ***.np***. Once this happens, there needs to be an additional bookkeeping to ensure that a proper RETURN edge is exercised upon exiting this branch. At this stage, we, therefore, need to have a unique ID tag and propagate this ID through the next node until the exit is reached. Once in this branch, we will move into the ***np -> .DET NOUN*** node, which is a SCAN node that checks if the current input string token is consistent with its labeled edge. If so, the parser will consume this token - in this case the DET token being consumed is 'a' - and jump to the next node. This action continues and the next token consumed is 'cat' until an unmatched token is discovered or a node with an edge to a RETURN node is reached. In the former case, the parser will reject the string input whereas, for the latter case, it will have to find a correct RETURN path by using the unique ID that has been tagged on before the branch is entered. Without this additional bookkeeping step, the GFG acts like an NFA.

In this case, the correct RETURN path leads to the ***vp -> np. vp*** node, which, in turn, will lead to the **.vp** branch and the 'eats' token will be consumed. This process continues until the **s.** node is reached. A string in CFL is recognized by the GFG if and only if there exists a path from ***.s*** to ***s.*** that is consistent with parsing conditions.

The main objective of this work is to introduce a practical parser generating tool using the GFG and evaluate it. The rest of this paper is organized as follows. The next section discusses the related work. Section 3 explains the methodology for creating the parser generator and describes how it works. Section 4 and 5 discuss the performance of the generated parser and conclude the paper, respectively.

II. RELATED WORK

The success of programming languages comes from the development of parsing algorithms. During the early period of digital computing, most compiler parsers were hand-written. This makes the parsers for large compiler projects hard to implement and maintain. In 1963, Brooker et al. [2] presented a "Compiler-Compiler" concept that leads to a parser generator that was used to eventually generate compilers from high-level description. One of the best known parser generators is YACC [3] which is widely used to create compilers such as gcc and g++. YACC handles grammars in LALR(1) form. There are a few shortfalls in YACC. For instance, the LALR parsing algorithm complicates the task of tracking syntax errors [4]. In addition, there are many restrictions when facing "shift-reduce" or "reduce-reduce" conflicts [5]. Another well-know recent parser generator is ANTLR [6]. It uses LL(*) algorithm from EBNF specification, which makes it easier to implement than YACC. However, there are only a handful of grammars that are compatible with ANTLR, and, hence, limiting its applicability.

In 1968, Jay Earley published his research that provided a general algorithm for parsing any context-free grammar [7]. Earley's algorithm was applied in a few parser generators such as Spark [8] and Accent [9]. Nevertheless, Earley's algorithm was difficult to understand and interpret because it relied on some hard mathematical concepts and models. As a consequence, many researchers try to simplify it by using different representations, for example, a graph representation proposed by Woods [10] that re-transforms a context-free grammar to Recursive Transition Networks (RTNs). There are a few variants for RTNs, a notable one is LR-RTNs [11]. The GFG can be classified as another RTN variant. The GFG represents a CFG on a single graph that concisely and clearly captures the operations in Earley's algorithm. Moreover, the GFG can be an important part to solve program analysis problems [12]. It is also a great framework for studying compilers and natural languages because it illustrates relationships among subclasses of context-free grammars as well as regular languages.

III. OUR GFG PARSER GENERATOR

We have developed a parser generator for recognizing and parsing context-free grammars. There are two main sections for the input specifications. These are a lexicon file and a grammar file. The lexicon file is a part that describes the tokenizing logic for each terminal symbol using regular expression. We decide to use the PLY library[13] which is simple and adaptable for tokenizing input strings. An example of a lexicon file is shown in Fig. 2.

```
tokens = (
      'NUMBER', 'PLUS', 'MINUS', 'LPAREN', 'RPAREN'
)

t_PLUS      =   r'\+'
t_MINUS     =   r'-'
t_LPAREN    =   r'\('
t_RPAREN    =   r'\)'
t_NUMBER    =   r'\d+'

```

Fig. 2. Example of a lexicon file.

The grammar file, on the other hand, is a part that declares the productions for grammars which comprise of a few lines beginning with a non-terminal symbol on the left side of an arrow "->" and many mixed of terminal and non-terminal symbols on the right side of "->". These symbols may be separated by a space. Every non-terminal symbol in this file must be lowercased whereas a terminal symbol must be uppercased. In addition, each terminal symbol must be declared in lexicon file. The content of the grammar file looks like the following:

[Non-terminal Symbol1] -> [Production1] | [Production2] | ...
[Non-terminal Symbol2] -> [Production1] | [Production2] | ...

When our parser generator is applied to the input specifications, it will create an output file, which is a parser in the Python language. The output file contains many functions that provide an easy way to implement additional semantic actions. An example of a generated parser is shown in Fig. 3. One who has dabbled with compiler literature will quickly realize that this generated parser is for the canonical expression grammar.

```
def sem_root_1(p):
      'root -> exp'
      print "cal >",p[0]
      return p

def sem_exp_1(p):
      'exp -> LPAREN exp PLUS exp RPAREN '
      return p[1]+p[3]

def sem_exp_2(p):
      'exp -> LPAREN exp MINUS exp RPAREN '
      return p[1]-p[3]

def sem_exp_3(p):
      'exp -> NUMBER '
      return int(p[0].value)

import sys,os
from noom.Noom import Noom
if __name__ == "__main__":
      E  = Noom(os.path.abspath(__file__) ,"expression.lex")
      print "#### THIS GRAMMAR MUST HAVE PARENs IN EVERY OPERATION ####"
      while True:
            E.run(raw_input("cal > "))
```

Fig. 3. Example of an output file.

Our parsing algorithm for the GFG is a sort of breadth-first exploration thereof. It determines reachability along a given path. The algorithm is explained using pseudocode in Alg1.

state is a tuple consisting of a current node and a "starting number". The number i means that the procedure starts from input at index i.

chart is a collection of *state* and *chart[i]* indicates the current progress with input at index i

Alg. 1. Parsing Algorithm for GFG.

```
FUNCTION GrammarFlowGraph( token , GFG):
  addState (startNode,0) to Chart[0]
  FOR i from 0 to length(words) :
    FOR each state in chart[i] :
      currentNode = state.node
      IF currentNode is END node :
        FOR each startState in chart[state.start] :
          completer(state,startState )
        ENDFOR
      ELSE :
        FOR each childNode in currentNode :
          IF there are label in edge(currentNode,childNode) :
            IF label == input[i]
                addState (childNode, state.start) to Chart[i+1]
            ENDIF
          ELSE :
            addState (childNode,i) to Chart[i]
          ENDIF
        ENDFOR
      ENDIF
    ENDFOR
  ENDFOR
  RETURN chart
```

For a more comprehensive example, let's consider Fig. 4 that shows a part of the Python grammar that we would like to parse and generate a runtime interpreter for it. Once the grammar is declared in the grammar file and the lexical tokens defined in the lexicon file, our GFG parser generator generates a parser that recognizes the code snippet shown below the grammar. Once the functionalities of a Python interpreter is put in the semantic actions, one can run the code and the correct result displayed accordingly.

We argue that implementing a parser generator with the GFG is easier than with the original Earley representation for the following reason. The GFG allows parsing to be viewed as an extension to simulating moves through an NFA and there exists well-defined coding patterns for this type of task. As a consequence, it takes less time to do the coding and also promotes better understanding of the parsing algorithm. After all, many people are more familiar with NFA than Earley's esoteric mathematics. Nevertheless, ease of implementation cannot be the main reason to justify the advent of GFG parsers. One needs to consider how they perform as well and that is the discussion topic in the next section.

```
root -> single_input
single_input -> NEWLINE | simple_stmt
single_input -> compound_stmt | compound_stmt NEWLINE
simple_stmt -> small_stmt NEWLINE
small_stmt -> flow_stmt | expr_stmt
expr_stmt -> test ASSIGN test | test
flow_stm -> BREAK | CONTINUE

compound_stmt -> if_stmt | while_stmt
if_stmt -> IF test COLON suite
if_stmt -> IF test COLON suite ELSE COLON suite
while_stmt -> WHILE test COLON suite

suite -> simple_stmt | NEWLINE INDENT stmts DEDENT
stmts -> stmts stmt | stmt
stmt -> simple_stmt
stmt -> compound_stmt

test -> expr | relexpr

 ** Python subset Interpreter **

 >>> a = 10
 >>> if a ==  10:
 ...     while a >= 0:
 ...       print a
 ...       a = a - 2
 ...
 10
 8
 6
 4
 2
 0
```

Fig. 4. Python runtime interpreter created from our parser generator.

IV. Evaluation

The evaluation of our parser generator focuses on the performance of the generated GFG parser. We measure the time it takes to parse strings of varying lengths using the GFG parser. In addition, we measure the memory footprint required to parse those strings. Two other parsers that we compare our GFG parser against are the original Earley's parser and the CYK parser. These are well-know parsers for general CFGs in the programming language as well as the natural language processing community.

For the execution time, we use the Timeit library to measure it. The measurement is taken after initializing all the variables. For each experiment, we collect the measurements for 10 rounds before calculating the average value for the execution time of each experiment. As for the memory usage, we use the Psutil library to measure this.

We will test the generated parsers on two grammar types:

1) *Simple arithmetic expression*

2) *Arithmetic expression in Chomsky Normal Form (CNF)*

These two grammars produce the same result. The first one uses fewer numbers of productions and each production may have different lengths. On the other hand, the second grammar specifies that the length of each production must be exactly two. This will, as a consequence, create more productions. Note that these two grammars are unambiguous, although we could have also introduced some degree of ambiguity to them.

For each experiment, we compare parsing of the above two grammars among these three parsing algorithms:

1) *Grammar Flow Graph (GFG)*
2) *Earley's algorithm*
3) *CYK algorithm*

These three parsing methodologies are applicable to any general CFG. They all have the same O(N^3) runtime, but have different constant factors [14, 15]. The parameter N is proportional to the size of the string being parsed.

All the experiments are conducted under Python 3.4 runtime environment. The hardware system has an Intel Core i3 2.93 GHz CPU with 4 Gbytes of RAM running 64-bit Ubuntu desktop operating system.

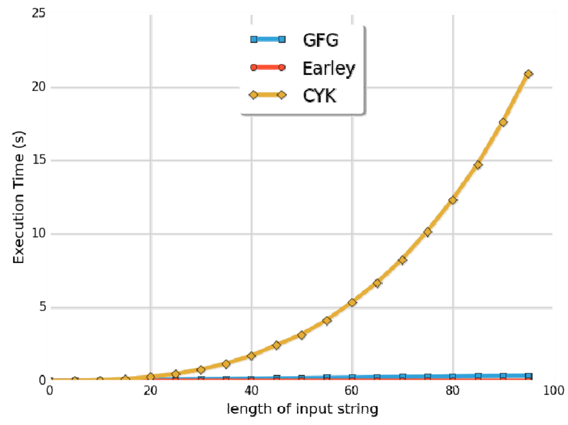

Fig. 5. Execution time for simple arithmetic expression.

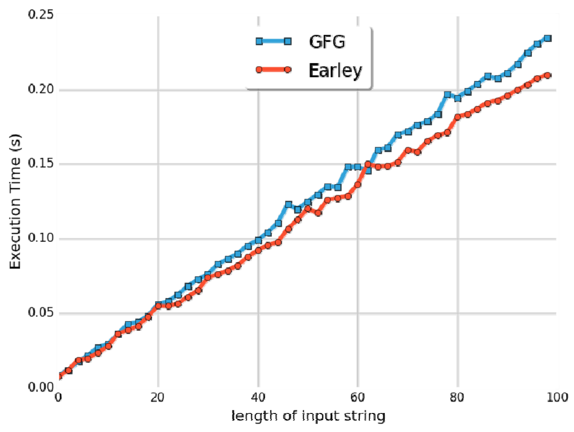

Fig. 6. Execution time for simple arithmetic expression zooming in on the Earley's and GFG's results.

A. *Execution time for simple arithmetic expression*

Fig. 5 shows the performance of the three parsing algorithms when parsing simple arithmetic expression with varying lengths of input strings. As expected, the execution time increases as the input size grows. However, the growth rate for CYK is much more significant than that of Earley and GFG. In fact, for this type of grammar, it looks as if CYK runtime is growing polynomially whereas Early's and GFG's only linearly. Fig. 6 zooms in to look more closely at the results for Earley and GFG. It indicates that the runtime growth rates for these two are indeed comparable and appear to be linear. The performance superiority of Earley parser over CYK can be attributed to the fact that Earley generates fewer "wasted" intermediate parse trees. These intermediaries have no bearing on the final parse tree [16]. As for the GFG, its performance characteristic is similar to that of Earley since the two are conceptually equivalent when it comes to parsing actions.

B. Execution time for arithmetic expression in CNF

Fig. 7 shows the performance of the three parsing algorithms when parsing arithmetic expression in CNF with different input string lengths. The results are in line with those obtained with the simple arithmetic expression. This suggests that the CNF of a grammar does not significantly affect the execution time for parsing the language it represents. The increase in the production rules seems to be counterbalanced by the decrease in the production length.

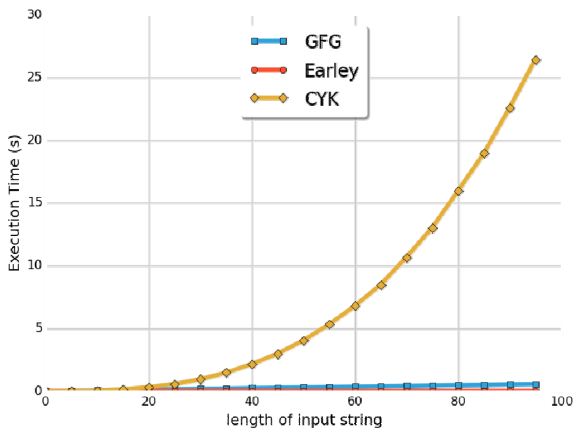

Fig. 7. Execution time for arithmetic expression in CNF.

C. Memory usage in simple arithmetic expression

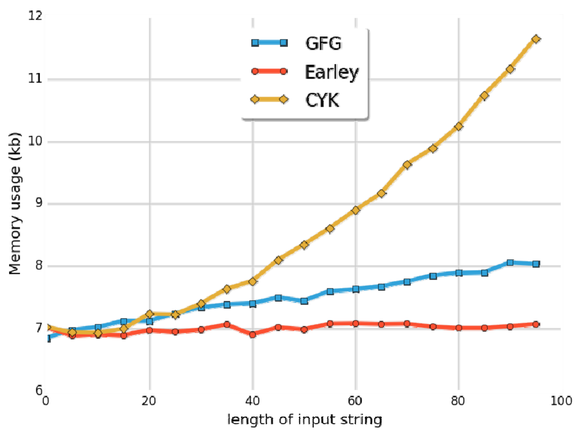

Fig. 8. Memory usage in simple arithmetic expression.

Fig. 8 shows the memory footprint of each parsing algorithm required to process input strings of varying lengths. In this case, the input strings are arithmetic expressions.

Earley has the most desirable memory usage characteristic for this type of grammar. The memory footprint seems to stay constant even with an increase in the size of the input. For GFG, the memory usage goes up relatively slowly as the input size grows. CYK memory performance is inferior to both GFG's and Earley's. The memory footprint seems to grow very quickly as the input size increases. This is because CYK operates in a bottom-up parsing style. It needs to discover patterns for producing substrings ranging from those whose lengths equal one to those whose lengths equal the entire string to be parsed. Thus, it consumes a lot of memory. In addition, CYK processing requires that a grammar be first transformed into CNF style which can lead to an undesirable bloat in required space ranging from |G| to $2^{|G|}$ [17] where |G| is the size of the CNF grammar.

As for GFG, even though its parsing operation is similar to Earley's, its memory characteristic is a bit less attractive. One notable reason is that GFG needs to do re-transformations of grammars into graph forms before further processing. This can increase the size of the grammar to n*|G| where n is the length of the longest production in a grammar of size |G|. Nevertheless, modern computers are equipped with huge amount of RAM that can make this memory issue for GFG irrelevant.

V. CONCLUSION

This paper presents a parser generator based on the GFG framework. The parser generator has been implemented in Python and released as open-source software through the following link:

https://bitbucket.org/kramatk/earleyparser

We argue that a GFG parser is easier and more intuitive to implement than the original Earley parser because it allows parsing to be viewed as a generalization of simulating moves through an NFA, a conceptual framework most compiler people are familiar with.

The evaluation indicates that, for execution time, parsers generated from our parser generator perform as well as Earley parsers and significantly better than CYK parsers. As for memory footprint, Earley and GFG are considerably better than CYK and Earley outperforms GFG by a small margin.

REFERENCES

[1] K. Pingali and G. Bilardi, A Graphical Model for Context-Free Grammar Parsing, Int. Conference on Compiler Construction, 2015

[2] Brooker, R .A.; MacCallum, I. R.; Morris, D.; Rohl, J. S. (1963), "The compiler-compiler", Annual review in automatic programming 3: 229–275

[3] S. C. Johnson, YACC — yet another compiler. UNIX Programmer’ s Manual, 7th Edition, 1978.

[4] G. Sussman and G. Steele. Scheme: An interpreter for extended lambda calculus. Technical Report AI Memo 349, AI Lab, M.I.T., 1975.

[5] J. Haberman, LL and LR in Context: Why parsing tools are Hard, http://blog.reverberate.org/2013/09/ll-and-lr-in-context-why-parsing-tools.html, retrieved on October 25, 2014.

[6] T. Parr and K. Fisher. LL(*): the foundation of the ANTLR parser generator. In Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, PLDI '11, 2011.

[7] J. Earley. An efficient context-free parsing algorithm. Commun. ACM, 13(2):94–102, 1970.

[8] J. Aycock. The design and implementation of SPARK, a toolkit for implementing domain-specific languages. In Journal for Computing and Information Technology, pages 55–66.

[9] F.W. Schröer, ACCENT, A Compiler Compiler for the Entire Class of Context-Free Languages, Technical Report, 2000: accent.compilertools.net

[10] W. A. Woods. Transition network grammars for natural language analysis. Commun. ACM, 13(10), 1970.

[11] M. Perlin. LR recursive transition networks for Earley and Tomita parsing. In Proceedings of the 29th annual meeting on Association for Computational Linguistics, ACL '91, 1991.

[12] T. Reps. Program analysis via graph reachability. Information and Software Technology, 40(11-12):701–726, 1998.

[13] PLY (Python Lex-Yacc), Retrieved October 19, 2014: http://www.dabeaz.com/ply/

[14] D. Grune and C.J.H. Jacobs, Parsing Techniques: A Practical Guide. Ellis Horwood, Chichester, 1990

[15] L. Tratt, Parsing: The Solved Problem That Isn't, Retrieved October 25, 2014: http://tratt.net/laurie/blog/entries/parsing_the_solved_problem_that_isnt

[16] J. Kegler, Is Earley parsing fast enough?, Retrieved October 25, 2014: http://blogs.perl.org/users/jeffrey_kegler/2013/04/is-earley-parsing-fast-enough.html

[17] M. Lange and H. Leiß, To CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm,Informatica Didactica 8.