A New Algorithm To Evaluate Terminal Heads Of Length K

David Pager
Department of Information and Computer Science
University of Hawaii at Manoa
pagerd@hawaii.rr.com

Xin Chen
Department of Information and Computer Science
University of Hawaii at Manoa
chenx@hawaii.edu

Abstract

This paper presents a new $FIRST_k(\alpha)$ algorithm for finding the terminal heads of length k of a given string in a context-free grammar, which is an alternative to the previous method of Aho and Ullman. Performance study shows the new algorithm in general has better performance, which can be considerable under some scenarios, such as when the input string α is long. The algorithm can be applied in situations such as LL(k) and LR(k) parser generation, and has been actually implemented in a LR(k) parser generator.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: Processors – *translator writing systems and compiler generators*. F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems – *parsing*.

General Terms Algorithms, Languages, Theory.

Keywords $FIRST_k(\alpha)$; $THEAD_k(\alpha)$; Terminal heads; LR(k)

1. Introduction

1.1 Overview

The algorithm to evaluate the terminal heads of length k of a given string in a context-free grammar, often denoted as $FIRST_k(\alpha)$, has important applications in the computation of LALR, LL and LR parser generation algorithms.

The algorithm when k = 1 is applied widely, which is the simple case and can satisfy the design needs of a large proportion of programming languages in use today. Examples are LALR(1) parser generators such as Yacc and Bison [7][8][9], as well as in LL(1) and LR(1) parser generators.

APPLC'13 February 23, 2013, Shenzhen, Guangdong, China.

The case of k > 1 is more complex and rare, however it is equally important since LALR(k), LL(k) and LR(k) are of wide interests in theoretical research, and has typical uses in practice too. One example is LL(k) parser generator ANTLR [5][6]. Other examples include language translation and natural language processing. For example, in Italian, genders are assigned to noun, verb and adjective. The English sentence "The <adjective> student is a <adjective> <person>" can derive into masculine form "*Lo studente Italiano é un uomo alto*" or feminine form "*La studentesse Italian é una donna alta*". The Italian grammar involved here is LR(k) where k > 1.

Here are two examples on the calculation of $FIRST_k(\alpha)$.

Example 1. Given grammar S → NM, N → st, M → bc. We want to find $FIRST_k(\alpha)$ for string α = NM. This is a trivial case, we can just plug N and M into NM to obtain α = stbc. Calculation of $FIRST_k(\alpha)$ is easy: $FIRST_1(\alpha)$ = {s}, $FIRST_2(\alpha)$ = {st}, $FIRST_3(\alpha)$ = {stb}, $FIRST_4(\alpha)$ = {stbc}.

Example 2. Given grammar S → NML, where N → Ns | ε, M → Mt | ε, L → bc. Here ε is the empty string. We want to find $FIRST_k(\alpha)$ for string α = NML. Then actually N = s*, and M = t*, and α = s*t*bc. $FIRST_1(\alpha)$ = {s, t, b}, $FIRST_2(\alpha)$ = {ss, st, sb, tt, tb, bc}, $FIRST_3(\alpha)$ = {sss, sst, ssb, stt, stb, sbc, ttt, ttb, tbc}, $FIRST_4(\alpha)$ = {ssss, ssst, sssb, sstt, sstb, ssbc, sttt, sttb, stbc, tttt, tttb, ttbc}.

In this paper we present a new algorithm, which we call $THEAD_k(\alpha)$, to evaluate the terminal heads of length k of a given string in a context-free grammar. It is an alternative to the $FIRST_k(\alpha)$ algorithm of Aho and Ullman [1], and takes a very different approach. In this paper, we will present the algorithm, give examples, compare to the method of Aho and Ullman, and discuss other related issues.

Since $FIRST_k(\alpha)$ is a fundamental algorithm that works as a basic building block in compiler theory and practice, its improvement should have wide impact.

1.2 Terminology

We define the following terms for the discussion:

An *alphabet* is a set of symbols, where a *symbol* is a non-divisible basic element of the alphabet.

A sequence of symbols concatenated together is called a *string*. We represent the length of a string *s* as |s|.

A *grammar* for a language *L* is defined as a 4-tuple $G = (N, \Sigma, P, S)$. Here *N* is a set of *non-terminal* symbols, Σ is a set of *terminal* symbols disjoint from the set *N*, *P* is a set of *productions*, and *S* is the *start symbol* from which the production rules originate from.

A terminal symbol appears only on the right side of productions. A non-terminal symbol can appear on either the left or right side of productions.

A *k-head* of a string *S* is a string which is made of the first k symbols of *S*, or the first k symbols of any string that can derive from *S*.

A *k-terminal head* or *k-thead* of a string S is a *k-head* of S which is made up of terminal strings only.

A string is said to *vanish* if it can derive the empty string.

We use upper case Roman letters A, B, C, … to represent non-terminals, lower case Roman letters a, b, c, … to represent terminals, and Greek letters α, β, γ, … to represent strings. An empty string is represented by ε.

Example 3. Given grammar G1:

$X \rightarrow X\ Y \mid a$

$Y \rightarrow b \mid \varepsilon$

Here a and b are *terminal* symbols because they appear only on the right side of the productions of G1. X and Y are *non-terminal* symbols because they can appear on the left side of the productions of G1.

Y *vanish*es because it can derive the empty string. The shortest string X can derive is a, therefore it does not vanish because it cannot derive the empty string.

Given string α = XY, its 1-head can be X or a, and its 2-head can be XY, XX, aY, Xb, ab or aa. Its 1-thead is a, and its 2-thead can be aa or ab.

1.3 Related Work

A survey of previous work on the calculation of $FIRST_k(\alpha)$ gives the following literature.

1.3.1 Early work

The work of DeRemer and Pennello [3] and Kristensen and Madsen [4] are examples of early discussions on the calculation of $FIRST_k(\alpha)$, which are typically vague and imprecise.

The work of Kristensen and Madsen [4] on "Methods for computing LALR(k) lookahead" discussed computing $FIRST_k$ for finding lookahead strings, which is needed by their LALR(k) algorithm. Their method is based on simulating all steps involved in parsing starting from a relevant state in a LR(0) machine. Given an example of calculating $LALR_k$ for $[A \rightarrow \bullet\ \alpha]$, their method wants to obtain the sets $\bigcup \{FIRST_k(\psi_i) \mid i = 1,2, \ldots, n\}$ for all items $[B_i \rightarrow \varphi_i \bullet A\ \psi_i]$, which "*may be computed by simulating all possible steps that the parse algorithm may take starting in the state* $GOTO_k(S, A)$ *with an empty parse stack*". They further pointed out that the set $\bigcup \{FIRST_k(\psi_i)\}$ is not enough, and proceeded to discuss how to cover edgy cases such as when the grammar is circular or contains ε-productions, and ended their discussion with cases where the simulated parsing might fail due to circularity.

1.3.2 Method of Parr

The PhD thesis of Parr [5] proposed a method to compute $FIRST_k(\alpha)$. This is used in the implementation of LL(k) parser generator ANTLR. Parr's PhD thesis introduces the GLA grammar representation in chapter 3, and explains lookahead computation and representation in chapter 4.

Basically, a data structure called GLA (Grammar Lookahead Automata) is used to represent grammars. To calculate LR(k) lookahead, do a constrained walk of a GLA, and the lookaheads are stored as a lookahead DFA (Deterministic Finite Automata). He also discussed how to solve the cycle issue with cache mechanism.

This is similar to the method of Kristensen and Madsen in that it utilizes the parsing machine to do the computation and tightly integrates the calculation of lookahead strings with parsing, and in that none of them is a standalone method to calculate $FIRST_k(\alpha)$.

1.3.3 Method of Aho and Ullman

Aho and Ullman gave a standalone algorithm to calculate $FIRST_k(\alpha)$, which is given as Algorithm 5.5 in [1, page 357]. Their method is described below.

First an operator $\oplus_k$ is defined: given an alphabet Σ and two sets $A \subseteq \Sigma^*$, $B \subseteq \Sigma^*$, $S = A \oplus_k B$ is the set of all strings formed from the ordered concatenation of string pairs (a, b), where $a \in A$, $b \in B$, and the length of strings in *S* is less than or equal to k. In addition, if A = Ø or B = Ø, then *S* = Ø.

Now given a context free grammar G = (N, Σ, P, S) and a string $\alpha = X_1X_2\ldots X_n$ in $(N \cup \Sigma)^*$, $FIRST_k(\alpha) = FIRST_k(X_1) \oplus_k FIRST_k(X_2) \oplus_k \ldots \oplus_k FIRST_k(X_n)$, so we just need to calculate $FIRST_k(X)$ for any X.

If $X \in (\{\varepsilon\} \cup \Sigma)$, then $FIRST_k(X) = X$ for $k \geq 0$.

Otherwise, $X \in N$, then $FIRST_k(X)$ can be obtained in the steps below. Define a set $F_i(X)$ for X:

1) If $X \in (\{\varepsilon\} \cup \Sigma)$, then $F_i(X) = X$ for $i \geq 0$;

2) If $X \in N$, then $F_0(X)$ is the set of all $x \in \Sigma^*$ such that a production rule $X \rightarrow x\alpha$ exists and $|x| \leq k$; If $X \rightarrow \varepsilon$, then $F_0(X) = \{\varepsilon\}$; If $X \rightarrow \alpha\beta$, $\alpha \in N^+$ and $\beta \in (N \cup \Sigma)^*$, then $F_0(X) = \emptyset$;

3) Recursively obtain $F_{i+1}(X)$ based on previous calculation: $F_{i+1}(X)$ is the set of all $x \in \Sigma^*$ such that for every production rule $X \rightarrow Y_1Y_2\ldots Y_n$, $x = \{F_i(Y_1) \oplus_k F_i(Y_2) \oplus_k \ldots \oplus_k F_i(Y_n)\} \cup F_i(X)$;

4) It is notable that step 3) will converge after a certain number of steps, such that $F_{i+1}(X) = F_i(X)$ for all $X \in N$, then $FIRST_k(X) = F_i(X)$.

In summary, the method of Aho and Ullman breaks down the task of evaluating the terminal heads of length *k* of a string α into applying the $\oplus_k$ operation on the component symbols of α. It solves the second problem by building a table from bottom up like in dynamic programming.

It should be noted that following the above method, the result set will contain strings whose length $L \leq k$, however by definition the set $FIRST_k(\alpha)$ contains strings of length k. Aho and Ullman did not discuss this in more details, since it is really just a trivial matter. To clarify this little ambiguity, we take it as that, at the end of the above calculation, we will remove those terminal strings with length less than k from the result set.

2. The New $FIRST_k(\alpha)$ Algorithm: $THEAD_k(\alpha)$

In this section we introduce the new algorithm [10], discuss its correctness and complexity, and compare to existing methods.

2.1 The $THEAD_k(\alpha)$ Algorithm

We use $THEAD_k(\alpha)$ as the name of the new algorithm, and also use it to represent the set of terminal heads of string α, where the length of each terminal head string is k, i.e., $THEAD_k(\alpha)$ is the set of k-theads of string α. $THEAD_k(\alpha)$ contains all m-theads of string α where m = k, and is the same as $FIRST_k(\alpha)$.

To illustrate the algorithm, we define these notations:

For a string $\alpha = X_1X_2\ldots X_n$, $|\alpha|$ is the length of α ($|\alpha| = n$); α[i] is the i^{th} symbol of string α; *h(α, k)* denotes the first k symbols of α, i.e., prefix string of α of length k; $h_v(\alpha, k)$ is a substring of α that consists of the prefix string of α up to the k-th symbol that does not vanish, or the entire α string if it contains less than k symbols that do not vanish; *prod(α, i)* is the set of strings obtained by applying all possible productions to the i^{th} symbol X_i of α.

We also let T stand for the set of *Terminal*s, and NT stand for the set of *Non-Terminal*s. T^k stands for the set of strings made of *Terminal*s and whose length is k. Ø stands for the empty set.

Algorithm 1, $THEAD_k(\alpha)$, is shown in Figure 1.

In Algorithm 1, ***H*** and ***S*** are sets of strings initially empty. ***L*** is an auxiliary ordered list of strings which initially consists just of $h_v(\alpha, k)$.

Lines 5 to 10 add to the end of L the result of applying all possible productions to the i^{th} symbol in the current member β of L, omitting strings that are already in L, and truncating all members added which have k or more symbols that do not vanish, by deleting the part of the string following the k-th symbol that does not vanish.

Lines 11 to 13 remove from L all strings whose i^{th} symbol is a non-terminal.

Lines 14 to 19 remove from L all strings whose prefix of length k consisting entirely of terminals, and add the prefixes of length k involved to the set H.

Lines 20 to 25 remove from L all strings of length less than k which consist entirely of terminals, and add these to the set S.

On line 26, if L is empty, the algorithm terminates. ***H*** now will contain the required set of terminal strings of length k of α, i.e., the k-head set of α; and **S** will contain the set of terminal strings of length less than k which are derived from α. Obviously, ***H*** gives the result of $THEAD_k(\alpha)$.

ALGORITHM 1. $THEAD_k(\alpha)$

INPUT: STRING $\alpha = X_1X_2\ldots X_n$; Integer *k*: length of theads.
OUTPUT: SET H – CONTAINS K-THEADS OF α, AND (OPTIONALLY) SET S – CONTAINS M-THEADS OF α, M < K.

```
1   H ← Ø
2   S ← Ø
3   L ← { h_v(α, k) }
4   for i = 1 to k do
5       foreach string β in L do
6           φ = prod(β, i)
7           foreach string γ in φ do
8               L ← L ∪ { h_v(γ, k) }
9           end foreach
10      end foreach
11      foreach string β in L do
12          if β[i] ∈ NT then L ← L – { β }
13      end foreach
14      foreach string β in L do
15          if h(β, k) ∈ T^k then
16              L ← L – { β }
17              H ← H ∪ { h(β, k) }
18          end if
19      end foreach
20      foreach string β in L do
21          if |β| < k AND β ∈ T^|β| then
22              L ← L – { β }
23              S ← S ∪ { β }
24          end if
25      end foreach
26      if L == Ø then stop
27  end for
```

Figure 1. Algorithm $THEAD_k(\alpha)$

The entire algorithm derives a closure of the initial string in L, where each derived string in the closure satisfies the requirements on the length (should be equal to k) of the strings, and on the type of symbols (should be terminal symbol) in the strings.

2.2 Correctness of the Algorithm

We show the correctness of Algorithm 1 below.

Lemma 1. In Algorithm 1, at the end of the i^{th} outer loop cycle (lines 4-27), for each string *s* in list L, where $s = X_1X_2\ldots X_n$, the first i symbols $X_1, X_2, \ldots, X_i$ of *s* (or all the symbols of *s* if $|s| < i$) are terminals.

Proof. Prove by induction. For outer loop cycle i = 1, the step of lines 11-13 removes from L all strings whose 1^{st} symbol is a non-terminal. Thus for all the strings remained in L, the 1^{st} symbol is terminal. Now assume at cycle i = n-1, for all the strings in L, the first i symbols are terminals. At cycle i = n, the inner loop (lines 5-10) only makes derivations on the n^{th} symbol, and does not introduce any non-terminal symbols to the first n-1 symbols; next, Algorithm 1 removes from L those strings whose n^{th} symbol is a non-terminal (lines 11-13), thus for all the symbols in L, now their first n symbols are terminals. The remaining steps (lines 14-26) do not alter this fact. Therefore Lemma 1 holds. □

Lemma 2. In Algorithm 1, at the end of the i^{th} outer loop cycle, all the possible combinations of i-thead derivations are generated by the inner loop (lines 5-10).

Proof. This also can be proved by induction. When i = 1, this is obvious from the inner loop. Assume this holds for i = n-1. When i = n, for each string *s* in L, the first n-1 symbols of *s* are all terminals. In the inner loop, for each string *s* in L, all the possible productions are applied to the n^{th} symbol of *s*, thus all the possible terminal and non-terminal symbols at the n^{th} position are generated by string *s* and included in L. These form new derived strings, appended to the end of L, and processed by the next cycle. Thus Lemma 2 holds. □

Lemma 3. Algorithm 1 ends in k or less outer loop cycles (lines 4-27) when L becomes empty.

Proof. From Lemma 1, for all the strings generated in the k^{th} outer loop cycle, their first k symbols are all terminals, these are then removed from L (lines 14-25). In the cycles, all members added to L that have k or more symbols that do not vanish will be truncated (lines 3, 8 and 12). Thus L will be empty at the end of at most the k^{th} loop cycle, and Algorithm 1 ends. □

Theorem 1. When Algorithm 1 ends, all the possible k-thead derivations are included in H, and all m-thead derivations are included in S, where m < k.

Proof. This follows from Lemma 1, Lemma 2 and Lemma 3. □

2.3 Complexity of the Algorithm

In Algorithm 1, the complexity of the step of lines 6-9 is $O(|P_{ij}|)$, where $|P_{ij}|$ is the number of possible productions to the i^{th} symbol in the j^{th} member of L. For the loop of lines 5-10, the complexity is $O(|P_{ij}||L|)$.

The complexity of the entire algorithm is hard to analyze directly, but it is easy to see that, since the primary output is set H, the theoretical lower boundary of the number of steps needed is equal to the number of elements in the output set: $\Omega(|H|)$. H is the set of terminal strings of length k of α, so $\Omega(|H|) = \Omega(|T|^k)$, where |T| is the number of terminals in the alphabet. This is the theoretical lower boundary of both time and space requirements. Obviously, it is exponential in nature as expected. This is demonstrated by test case 2 in section 4 "Performance Study".

2.4 Comparison with other algorithms

Aho and Ullman's method and our method are both standalone algorithms to compute $FIRST_k(\alpha)$, where the computation rely on a set of production rules of the grammar only, and the parsing machine is not needed. Thus these two methods are better than the other methods in literature research.

Aho and Ullman's method takes a bottom up approach by first calculating $FIRST_i(X)$ for each symbol X, i = 1, 2, … k, then combining these building blocks to obtain $FIRST_k(\alpha)$. This is a systematic approach, which is also demonstrated in their handling of $FIRST_1(\alpha)$, which is discussed in [2, page 189]. Once the preparation phase is done, for whatever input string, the task boils down to applying the $\oplus_k$ operation on the consisting symbols of the input string, which concatenates elements from each set. However, the systematic nature also means that the overhead must always be taken to achieve good efficiency. From a practical point of view, since input strings are unknown, the entire preparation step must be done and its result be cached for later use.

In comparison, our method takes a top down approach. No previous computation is needed. The algorithm computes $FIRST_k(\alpha)$ on the fly based on symbols included in the input string. No cache is needed. It removes unnecessary overhead strings on the way of computation.

In nature, both methods are equivalent. Our method can also be used for the preparation process of Aho and Ullman's method.

Another difference is that the $FIRST_k(\alpha)$ method of Aho and Ullman gives a set of terminal heads whose length $L \leq k$, and this set must be kept during the entire calculation process, only at the very end can we remove those L < k. In comparison, our method separates terminal heads into two sets, for one set the length of terminal heads L = k, and for the other set L < k. The second set where L < k can be ignored from the calculation process.

3. Examples

In this section we show how $THEAD_k(\alpha)$ and $FIRST_k(\alpha)$ work on the same input string.

Example 4. Given grammar G2 (ε is the empty string):

X → Y | x | ε
Y → Z | y | ε
Z → X | z | ε
U → u

Find the set of 2-theads of XYZU using Algorithm 1: $THEAD_k(\alpha)$.

Since symbols X, Y and Z can all vanish, and U does not vanish, the string XYZU contains less than 2 symbols (i.e., only 1) that do not vanish, therefore we need to include the entire string XYZU as the initial element in the list L. Thus, at the beginning, L = {XYZU}.

First round of operation for i = 1 is shown in Table 1.

Table 1. Example 4, round 1 (i = 1)

i	j	String added to L	String Sequence Number
1	1	XYZU	1
		YYZU	2
		xYZU	3
		YZU	4
	2	ZYZU	5
		yYZU	6
	3		
	4	ZZU	7
		yZU	8
		ZU	9
	5	zYZU	10
	6		
	7	XZU	11
		zZU	12
	8		
	9	XU	13
		zU	14
		U	15
	10		
	11	xZU	16
	12		
	13	YU	17
		xU	18
	14		
	15	u	19
	16		
	17	yU	20
	18		
	19		
	20		

Table 2. Example 4, round 2 (i = 2)

i	j	String added to L	String Sequence Number
2		xYZU	1
		yYZU	2
		yZU	3
		zYZU	4
		zZU	5
		zU	6
		xZU	7
		xU	8
		yU	9
	1	xZZU	10
		xy	11
	2	yZZU	12
		yy	13
	3	yXU	14
		yz	15
	4	zZZU	16
		zy	17
	5	zXU	18
		zz	19
	6	zu	20
	7	xXU	21
		xz	22
	8	xu	23
	9	yu	24
	10	xXZU	25
	11		
	12	yXZU	26
	13		
	14	yYu	27
		yx	28
	15		
	16	zXZU	29
	17		
	18	zYU	30
		zx	31
	19		
	20		
	21	xYU	32
		xx	33
	22		
	23		
	24		
	25		
	26		
	27		
	28		
	29		
	30		
	31		
	32		
	33		

At this time, the step of lines 5-10 finishes. Next we follow lines 11-25. Remove from L all strings with non-terminals in the i^{th} (first) position; remove from L all strings whose prefixes of length 2 consisting entirely of terminals, and add these prefixes to H; and remove from L all strings of length less than 2 and contains only terminal strings. At this time, we have H = {}, S = {u}, L = {xYZU, yYZU, yZU, zYZU, zZU, zU, xZU, xU, yU}.

The second round where i = 2 is shown in Table 2.

Remove all strings with non-terminals in the i^{th} (second) position, remove all strings whose prefixes of length 2 are made up of terminals, and remove all strings of length less than 2 and contains only terminal strings, we have H = {xy, yy, zy, zz, zu, xu, xz, yz, yx, yu, zx, xx}, S = {u}, L = {}.

Example 5. Given grammar G2 as in Example 4, find the set of 2-theads of XYZU, this time use the $FIRST_k(\alpha)$ algorithm of Aho and Ullman.

Following the steps in Aho and Ullman's algorithm, we need $FIRST_k(\alpha)$, where α = XYZU, and k = 2.

$F_i(p) = \{p\}$, for all $p \in \{x, y, z, u, \varepsilon\}$, and $i \geq 0$.

$F_0(X) = \{x, \varepsilon\}$
$F_0(Y) = \{y, \varepsilon\}$
$F_0(Z) = \{z, \varepsilon\}$
$F_0(U) = \{u\}$
$F_1(X) = \{x, y, \varepsilon\}$
$F_1(Y) = \{y, z, \varepsilon\}$
$F_1(Z) = \{z, x, \varepsilon\}$
$F_1(U) = \{u\}$
$F_2(X) = \{x, y, z, \varepsilon\}$
$F_2(Y) = \{x, y, z, \varepsilon\}$
$F_2(Z) = \{x, y, z, \varepsilon\}$
$F_2(U) = \{u\}$

From this point on $F_i(S) = F_2(S)$ for $i \geq 3$, S = X, Y, Z, U. It converges here. Therefore:

$FIRST_2(X) = F_2(X) = \{x, y, z, \varepsilon\}$
$FIRST_2(Y) = F_2(Y) = \{x, y, z, \varepsilon\}$
$FIRST_2(Z) = F_2(Z) = \{x, y, z, \varepsilon\}$
$FIRST_2(U) = F_2(U) = \{u\}$

Note that here $FIRST_2(X)$ contains strings of length less than 2, because we need to keep them in the intermediate steps, as discussed at the end of section 2.4.

Finally, we can calculate $FIRST_k(\alpha) = FIRST_2(XYZU)$
$= FIRST_2(X) \oplus_2 FIRST_2(Y) \oplus_2 FIRST_2(Z) \oplus_2 FIRST_2(U)$
$= \{x, y, z, \varepsilon\} \oplus_2 \{x, y, z, \varepsilon\} \oplus_2 \{x, y, z, \varepsilon\} \oplus_2 \{u\}$
$= \{xx, xy, xz, xu, yx, yy, yz, yu, zx, zy, zz, zu, u\}$

As a last step as discussed at the end of section 1.3.3, we remove strings whose length are less than 2, which is 'u' here, and obtain {xx, xy, xz, xu, yx, yy, yz, yu, zx, zy, zz, zu}. This is the same result as using our algorithm.

4. Performance Study

We implemented both the $THEAD_k(\alpha)$ algorithm and the $FIRST_k(\alpha)$ algorithm, and compared their performance. In each experiment, the start time and end time are measured multiple times, and then average start time is subtracted from average end time to obtain the running time. The study was conducted on a Sun Microsystems sun4u Netra 440 server running Solaris. CPU is 1.6GHz, memory is 12 GB. For all the experiments below, test case 2 uses the most memory (hundreds of MB), so memory is not an issue. In the figure legends, *THEAD* represents $THEAD_k(\alpha)$, and *FIRST* represents $FIRST_k(\alpha)$.

Grammar G2 is used as the testing grammar.

4.1 Test case 1: α = UUUUUUUUUU, k = 1 to 10

Result is shown in Table 3 and Figure 2. When α = UUUUUUUUUU, there is only one terminal head, which is u^k for k = 1 to 10. The speed is very fast, at the level of microsecond. The relatively long delay when k = 1 for the $FIRST_k(\alpha)$ algorithm should be caused by the initial construction of the Fi(X) table.

Table 3. Number of generated k-theads and time spent on input string UUUUUUUUUU, for k = 1 to 10

k	# of k-theads	Time (sec) By HEAD	Time (sec) By FIRST
1	1	0.000022	0.000108
2	1	0.000009	0.000014
3	1	0.000012	0.00002
4	1	0.000018	0.000017
5	1	0.00002	0.00002
6	1	0.000027	0.000026
7	1	0.000032	0.000021
8	1	0.000072	0.000022
9	1	0.000047	0.000026
10	1	0.000053	0.000055

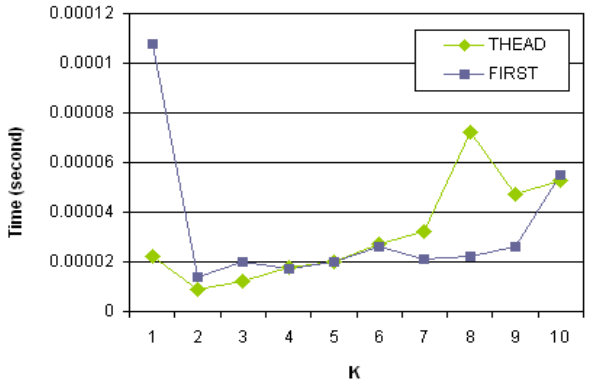

Figure 2. Time cost of $THEAD_k(\alpha)$ versus $FIRST_k(\alpha)$ for α = UUUUUUUUUU, k = 1 to 10

4.2 Test case 2: α = XXXXXXXXXX, k = 1 to 8

Result is shown in Table 4 and Figure 3. This is the worst case scenario where the theoretical bound of exponential behavior is observed. This is because each symbol of the input string is a non-terminal (X), which can derive 3 terminals x, y and z. The number of k-theads that can be generated is 3^k. When k is as small as 10, this will take hours to finish. The result is similar when α = YYYYYYYYYY or α = ZZZZZZZZZZ.

Table 4. Number of generated k-theads and time spent on input string XXXXXXXXXX, for k = 1 to 8

k	# of k-theads	Time (sec) By THEAD	Time (sec) By FIRST
1	3	0.000242	0.000221
2	9	0.001302	0.00145
3	27	0.00599	0.009041
4	81	0.032146	0.065045
5	243	0.213318	0.425997
6	729	1.463382	3.282263
7	2187	12.21782	26.23495
8	6561	135.462	297.5679

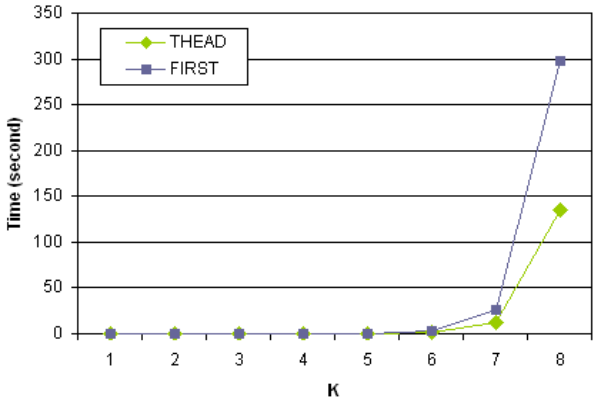

Figure 3. Time cost of $THEAD_k(\alpha)$ versus $FIRST_k(\alpha)$ for α = XXXXXXXXXX, k = 1 to 8.

4.3 Test case 3: α = XYZUXYZUYX, k = 1 to 9

Result is shown in Table 5 and Figure 4. Here α is a randomly generated string. We can see that $THEAD_k(\alpha)$ performs better than $FIRST_k(\alpha)$ for k = 1 to 9, but for k = 10, $FIRST_k(\alpha)$ runs faster. This possibly has to do with the way of implementation: in the implementation of $FIRST_k(\alpha)$, an ordered list is used to store the strings generated intermediately; for $THEAD_k(\alpha)$, the list used cannot be ordered, since new inserted strings will need to be processed and have to be attached to the end. When inserting a new generated string to the end of list L, $THEAD_k(\alpha)$ will search through the entire list to make sure it does not exist yet. To overcome this issue an auxiliary ordered list is used in the implementation. This slows it down when the list is long. Of course, better implementation using more efficient data structure can improve this scenario.

Table 5. Number of generated k-theads and time spent on input string XYZUXYZUYX, for k = 1 to 10

k	# of k-theads	Time (sec) By THEAD	Time (sec) By FIRST
1	4	0.000079	0.000315
2	16	0.00038	0.001807
3	63	0.002083	0.016498
4	162	0.011877	0.063377
5	486	0.100032	0.460147
6	1296	0.624867	2.756787
7	2916	3.3104	11.8662
8	4374	14.64284	26.12881
9	6561	62.89018	71.2255
10	6561	94.49193	81.37379

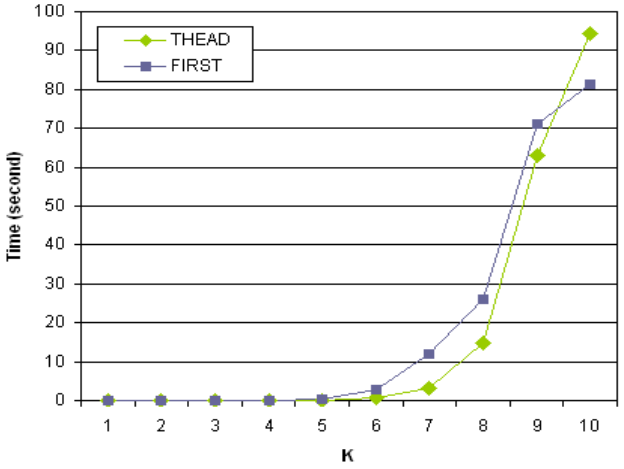

Figure 4. Time cost of $THEAD_k(\alpha)$ versus $FIRST_k(\alpha)$ for α = XYZUXYZUYX, k = 1 to 10.

4.4 Test case 4: Average on 100 strings of length 10, k = 1 to 8

100 input strings, each of length 10, are generated from the alphabet of {X, Y, Z, U} using a random number generator, and then fed to the algorithms to compare their performance. This means the input strings may be like:

```
1    YXXXYUUUUU
2    UZZUUUZXXY
3    YZZUYZZYZU
4    ZZUZUZYUZY
     …
100  UUYXXUUXUY
```

Result is shown in Table 6 and Figure 5. Table 4 shows the average number of k-theads generated and average time used by the $THEAD_k(\alpha)$ and $FIRST_k(\alpha)$ algorithms over 100 input strings of length 10, and k = 1 to 8. Figure 4 shows the graphical version of the average time used when k increases. It can be seen that the $THEAD_k(\alpha)$ algorithm uses less time.

Table 6. Average number of generated k-theads and time spent on 100 random strings of length 10, for k = 1 to 8

k	Avg # of k-theads	Time (sec) By THEAD	Time (sec) By FIRST
1	3.07	0.000068	0.000177
2	10.37	0.000381	0.000969
3	32.73	0.001999	0.006003
4	95.43	0.011761	0.03663
5	270.25	0.078635	0.246849
6	697.89	0.505454	1.496519
7	1662.39	3.484229	8.207717
8	3669.3	27.723004	55.275918

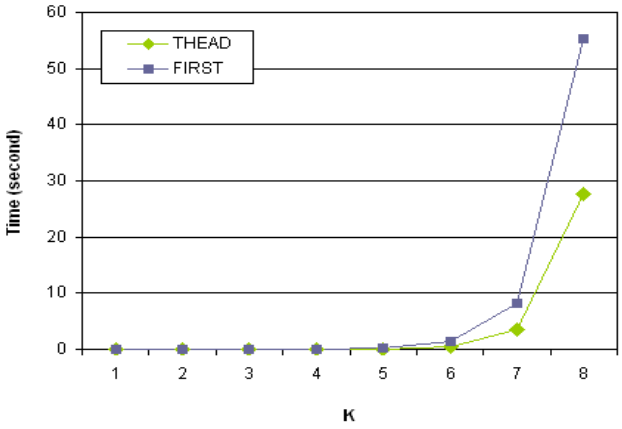

Figure 5. Time cost of $THEAD_k(\alpha)$ versus $FIRST_k(\alpha)$ when k increases. Averaged over 100 strings of length 10

4.5 Test case 5: 100 strings of length 1 to 100, k = 2

In this test case, k is fixed, while the input string is a k-prefix of the following randomly generated string, where input string length |α| = 1 to 100, i.e., the input strings may be like:

```
1    Y
2    YZ
3    YZZ
4    YZZY
     ...
100  YZZYYXZYYXYZUXYYUYXZUYYUZXUYZZYYZXX
     XXXUUUYXYZZYZYZUUXZXZYZXXUZUXYZYYYU
     YZZZZZUZXZYYYYZYYUXZZUYZUZXUY
```

Result is shown in Figure 6. The time used by $THEAD_k(\alpha)$ does not increase with k, but it does increase with $FIRST_k(\alpha)$ (and the increase is linear visually from the graph). This is easy to explain. $THEAD_k(\alpha)$ throws away the substring after the second symbol that does not vanish, so each time it starts with the prefix "YZ" of the input string. In comparison, $FIRST_k(\alpha)$ needs to do the $\oplus_k$ operation on every symbol of the input string, and n-1 $\oplus_k$ operations are applied for an input string of n symbols. To overcome this issue, $FIRST_k(\alpha)$ needs to use a pre-processing the same as line 3 of Algorithm 1.

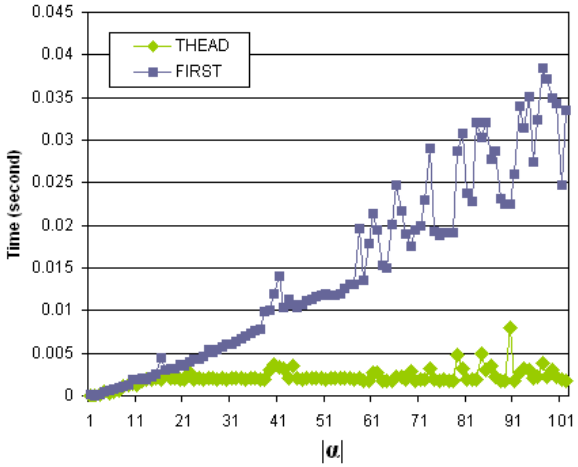

Figure 6. Time cost of $THEAD_k(\alpha)$ versus $FIRST_k(\alpha)$ when k = 2, and string length |α| increases

4.6 Summary

We can draw several conclusions from the experiments.

First of all, when the input string contains terminal symbols only, the speed is the fastest. When the input string contains non-terminal symbols only, the speed is the slowest, and may lead to the worst case scenario: exponential increase in computation time. For a grammar as simple as G2, when k = 10, it will take hours to finish using both algorithms.

In general the $THEAD_k(\alpha)$ algorithm performs better than the $FIRST_k(\alpha)$ algorithm, as shown by test case 4, which is averaged over 100 randomly generated strings of length 10 for k = 1 to 10.

However, it is also possible that $FIRST_k(\alpha)$ runs faster than $THEAD_k(\alpha)$, as shown in test case 3 when k = 10. Finally, when k is small, but input string is long, $THEAD_k(\alpha)$ will perform better than $FIRST_k(\alpha)$, as shown by test case 5. Actually, for this scenario, the time $THEAD_k(\alpha)$ takes will not increase when the size of the input string increase. However, the time used by $FIRST_k(\alpha)$ will increase linearly according to the length of the input string.

5. Implementation

We briefly discuss the implementation of the two approaches here, which is done in ANSI C from scratch.

To make the comparison of the two algorithms reasonable, it is necessary to implement them with similar data structures.

The major operations involved in both algorithms are set operations. In the current implementation, a set is implemented as a linked list. Search in the set is done by going through the list in linear order. That a linked list is chosen for the implementation is because of the nature of the $THEAD_k(\alpha)$ algorithm: a new generated string has to be appended to the end of the current set, which makes queue a natural and necessary choice. A queue of unknown size as in the current scenario is in turn naturally implemented as a linked list.

To guarantee similar search experience for both algorithms, an ordered list is used. For the method of Aho and Ullman, this is no problem. But for the $THEAD_k(\alpha)$ method, the queue (implemented as a list) to be appended to can not be ordered, so an auxiliary list is provided which stores the same strings as the queue but is in sorted order, such that when a search in the auxiliary ordered list does not return a hit, the new string is appended to the end of the queue. The maintenance of two lists in the $THEAD_k(\alpha)$ algorithm implementation obviously will slow it down to some degree.

This implementation can be improved by providing an auxiliary binary search tree or a hash table to both methods, which works much more efficient when decide if a string exists in a set. This improvement should be of more significance to the performance of the $THEAD_k(\alpha)$ algorithm implementation according to the above discussion.

Finally, a linked list suffices for all the operations of the $THEAD_k(\alpha)$ algorithm. For the algorithm of Aho and Ullman, an array is also used to store the pre-computed $FIRST_k(X_i)$ values of all the symbols X_i, such that given a random string $\alpha = X_1X_2\ldots X_n$, $FIRST_k(X_i)$ can be retrieved in constant time using index of X_i in the symbol table for the calculation of $FIRST_k(\alpha) = FIRST_k(X_1) \oplus_k FIRST_k(X_2) \oplus_k \ldots \oplus_k FIRST_k(X_n)$.

6. Application

One application of the algorithm presented here is to be used in a LR(k) parser generation algorithm.

Our study of the LR(k) algorithm shows that the calculation of LR(k) lookahead is one of the major steps involved. We designed and implemented the Edge-Pushing LR(k) algorithm, which depends on the $THEAD_k(\alpha)$ function to calculate k-lookahead. The Edge-Pushing algorithm is shown below in Figure 7, which is taken from a previous paper [11]. The $THEAD_k(\alpha)$ algorithm is used on line 13.

The Edge-Pushing algorithm is implemented in the HYACC parser generator, which is available as an open source parser generator [12][13].

Algorithm 2: Edge_Pushing(S)

INPUT: INADEQUATE STATE S
OUTPUT: S WITH CONFLICT RESOLVED, IF S IS LR(K)

```
1   Set_C ← Ø
2   Set_C2 ← Ø
3   k ← 1
4   foreach final configuration T of S do
5       T.z ← 0
6       Let C be the head configuration of T, and X be the
            context generated by C
7       Add triplet (C, X, T) to set Set_C
8   end foreach
9   while Set_C ≠ Ø do
10      k ← k + 1
11      foreach (C:A → α • B β, X, T) in Set_C do
12          k' ← k - C.z
13          calculate ψ ← theads(β, k')
14          foreach context string x in ψ do
15              if x.length == k' then
16                  Insert (S, X, last symbol of string x, C, T) to
                        Set_C2 and add to LR(k) parsing table
17              else if x.length == k' - 1 then
18                  Σ ← lane_tracing(C)
19                  foreach configuration σ in Σ do
20                      σ.z ← C.z + k'
21                      Let m be the generated context symbol in σ
22                      Insert(S, X, m, σ, T) to Set_C2 and add to
                            LR(k) parsing table
23              end if
24          end foreach
25      end foreach
26      Set_C ← Set_C2
27      Set_C2 ← Ø
28  end while
```

Figure 7. The Edge-Pushing Algorithm where $THEAD_k(\alpha)$ is used

7. Conclusion

In this paper we have presented a new algorithm to calculate the terminal heads of length k, which is called $THEAD_k(\alpha)$.

We reviewed relevant literature, of which Aho and Ullman's method is the only previously available standalone algorithm for this purpose.

We showed the new algorithm, discussed its correctness and complexity, and compared to previous work. Examples are given to evaluate terminal heads of length k of a given

string by using the $THEAD_k(\alpha)$ algorithm and the $FIRST_k(\alpha)$ method of Aho and Ullman.

An empirical study was conducted to compare the $THEAD_k(\alpha)$ algorithm and the $FIRST_k(\alpha)$ algorithm. In general, when averaged over a large number of randomly generated input strings, $THEAD_k(\alpha)$ performs faster than $FIRST_k(\alpha)$. When the input string α is long but k is small, $THEAD_k(\alpha)$ always performs better than $FIRST_k(\alpha)$.

Finally, we discussed the application of the new $THEAD_k(\alpha)$ algorithm, and pointed out that it has been used to implement the edge-pushing algorithm in the HYACC parser generator.

Due to the fact that $FIRST_k(\alpha)$ is a fundamental algorithm that works as a basic building block in compiler theory and practice, its improvement should have wide impact.

References

[1] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*, vol. 1 (page 357, Algorithm 5.5). Prentice-Hall, 1972.

[2] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. Compilers, Principles, Techniques, and Tools. Addison Wesley, 1986.

[3] Frank L. DeRemer and Thomas Pennello. Efficient computation of LALR(1) look-ahead set. TOPLAS, 4(4), October 1982.

[4] Ole L. Madsen Bent B. Kristensen. Methods for computing LALR(k) lookahead. ACM Transactions on Programming Languages and Systems, 3(1):60–82, January 1981.

[5] Terence Parr. Obtaining practical variants of LL(k) and LR(k) for k > 1 by splitting the atomic k-tuple. PhD thesis, Purdue University, August 1993.

[6] ANTLR. Available at: http://www.antlr.org

[7] Stephen C. Johnson. YACC – yet another compiler compiler. CSTR 32, Bell Laboratories, Murray Hill, NJ, 1975.

[8] Charles Donnelly, Richard Stallman. Bison, The YACC-compatible Parser generator (for Bison Version 1.23). 1993.

[9] GNU Bison. Available at: http://www.gnu.org/software/bison

[10] David Pager. Evaluating Terminal Heads Of Length K. Technical Report No. ICS2009-06-03, University of Hawaii, Information and Computer Sciences Department, November 2008. Available at: http://www.ics.hawaii.edu/research/tech-reports/terminals.pdf/view

[11] Xin Chen, David Pager. The Edge-Pushing LR(k) Algorithm. Proceedings of International Conference on Software Engineering Research and Practice, p.490-495. Las Vegas, July 18-21, 2011.

[12] Xin Chen, David Pager. Full LR(1) Parser Generator Hyacc And Study On The Performance of LR(1) Algorithms. Proceedings of The Fourth International C* Conference on Computer Science & Software Engineering, p.83-92. Montreal, Canada, May 16-18, 2011.

[13] Xin Chen. LR(1) Parser Generator Hyacc (2008). Available at: http://hyacc.sourceforget.net