A Grammar Supporting Conditional and Parallel Parsing

Dave Bone

December 10, 2014

Abstract

This is the second paper in the series responding to the questions raised as to why deterministic context-free grammars are not making it any easier to define and compile current-day computer languages like C++. As the first paper broadly outlined the requirements supporting parallel parsing, this paper defines an extended Backus-Naur Form (EBNF) grammar to support conditional and parallel parsing in a bottom-up fashion. With this material, the reader will be able to continue with the third paper which discusses parallel parsing solutions to the questions raised by the first paper in this series.

1 Introduction

As I come from Quebec, bienvenue — welcome to the 2nd paper. To start with, the basic language theory of a context free grammar is developed along with the terminology used throughout this paper. I will draw from a marvelous book on language theory —"Formal Languages and Their Relation to Automata" by Hopcroft and Ullman. My copy is 1969 vintage. I'm impressed by its drawings, brevity, and depth on the subject even with its 35 years of age.

I will then build on this definition using as a model yacco2's grammar [1]. Below, the 3 items will be developed to extend a basic grammar into the parallel parsing domain:

- terminal classifications and definitions
- productions
- parallel and conditional parsing grammatical expressions

The principle theme of this paper is for the reader to feel comfortable with the developed ideas expressed within a grammar's context. Examples are drawn from yacco2 to illustrate the ideas being developed. The intent is not to be a user manual but to describe the "what part" needed within a grammar to support parallel parsing drawn from experience.

2 Context Free Grammar Definition

As defined by [2], a context-free grammar is composed of 4 parts with one restriction highlighted below in bold:

- V_N symbol represents the variables alphabet: synonyms are rules and nonterminals.
- V_T symbol represents the terminals alphabet.
- P symbol represents a set of productions with the following structure:

$$lhs \ \rightarrow \ rhs$$

  lhs is the left-hand-side of a production using **a rule** $\in \ V_N$, $\rightarrow$ separates the two sides, rhs is the right-hand-side of the production representing a string of symbols drawn from V_N and V_T. rhs can be the empty string.

- S start symbol drawn from V_N appearing only in the lhs of a production

V_N, V_T, and P are finite sets. V_N and V_T are disjoint: there are no common elements between them. All the components making up a grammar is called a grammar tupple — { V_N,V_T,P,S }.

Other terminology:
Below are 2 types of terminal sets described within the LR(1) domain having 1 terminal look ahead. I do not generalize their definitions to the k look-ahead scene as this does not add anything to their usage within this paper. How they are derived is out of the scope of this paper.

Lookahead: set of terminals that delineates a boundary. These are the sets that demarcate when a rhs of a production reduces to its lhs. Lookahead boundary is used inter-changeably with lookahead to emphasis the nature

of demarcation.

First set: set of terminals derived from a set of symbol strings drawn from V_N and V_T. Depending on their contextual use, the terminals are the first terminals starting (derived from) the string of symbols: consequently its name.

2.1 A Metamorphosis...

Yacco2's grammar has a basic anatomy of 4 parts: Fsm, Parallel thread, Terminals, and Productions. It has the same grammar tupple just described with Terminals and Productions being equivalent to the formal definitions of V_T and P. P is used to implicitely define S and V_N thus completing the grammar tupple. Impliciteness makes for a neater grammar and eases in its maintenance.

As the bare essence of a grammar just outlined is elegant and seductive in simplicity, reality demands taking this definition and adding source code appendages to forge it into software. There are two related problems: have the compiler / compiler digest the grammar and secondly in parallel, associate the syntax-directed code with its emitted finite state tables. These code appendages go across all the elements of a grammar tupple. Fsm and Parallel thread are explicit extensions to the basic grammar which are packaging agents. The basic framework for syntax-directed directives will be developed later relating to parallel and conditional parsing. See [1] for a complete discussion with extensions into polymorphic class design.

3 Dressing up a grammar

Within the parallel parsing paradign, there are two types of grammars:

- stand-alone
- parallel

A stand-alone grammar is per current bottom-up parsing usuage. It is monolithic in nature, housing all grammatical expressions. It is equivalent to each of the grammatical passes made on a language — lexical, syntatic, semantic, emitting phases.

A parallel thread grammar contains a fragment of a grammatical expression that gets called by the parallel operator from within either grammar type. One can think of it as a recursive descent module except that it runs

as a thread and parses in bottom-up fashion. Its productions have the same properties of a stand-alone grammar except that its lookahead boundary is tunable whereas the stand-alone is not. It allows one to apply the divide-and-conquer approach to parsing.

Yacco2's language constructs uses a design similar to subroutines found in traditional computer languages. Though the grammar parts are not callable, each has a name, formal parameters and an enclosed body of code.

3.1 Fsm Section — the outer coat of a grammar

The Fsm section of the grammar supplies all the necessary ingredients to define the container housing the grammar in emitted C++ form. Of course as you have guessed, it stands for finite state machine. It houses the shift, reduce, and lookahead tables, user syntax-directed data and code, along with basic comments and compile time relics. Below is the Fsm construct of a grammar and its formal parameters each separated by a comma.

```
fsm (
  fsm-id "eol.lex",fsm-filename eol,fsm-namespace NS_eol
  ,fsm-class Ceol
  ,fsm-version "1.0",fsm-date "17 Juin 2003",fsm-debug "false"
  ,fsm-comments "end-of-line recognizer"
)
```

Each parameter has 2 parts: a keyword prefixed by `fsm-` whose name indicates intent and its related value; Yacco2's other sections use this same pattern for their parameters. The four most important keywords of fsm are: filename, namespace, class, and debug. These data are used in creating the outputed file by the compiler / compiler, with the C++ namespace and class name for the class definition of the fsm. The `fsm-class` parameter can also take on a syntax-directed coding block. See [1] for complete details. Debug is a switch to control tracing by the pushdown automaton and is currently turned off. In a multi-threaded run environment, this comes in very handy in debugging or in regression tests.

3.2 Parallel Thread Section — birth of a sidekick

The Parallel Thread section of the grammar supplies all the necessary ingredients to define the external name of the thread, the lookahead boundary, and the code needed to run the thread. Below is a grammar snippet of the parallel thread construct. In the interior of the `parallel-parser` construct

are various keyword / *** paired components. `parallel-thread-function` at source line 2 provides the external thread name.

```
1 parallel-parser (
2   parallel-thread-function
3     TH_eol
4   ***
5   parallel-la-boundary
6     eolr
7   ***
8 )
```

At source line 5 the `parallel-la-boundary` component is the most interesting to the grammar writer. This is the environment where one can fine tune the lookahead boundary for the thread's recognizing phrase. As stated in the first paper, ambiguity comes from too many contexts being mixed together. Parallel parsing allows one to refine the lookahead and to remove the conflicts of mixed contexts. This refinement to the lookahead is a major breakthrough in extending the expressive ability of deterministic context-free grammars. The lookahead expression is similar to an arithmetical expression made from V_T, V_N and the plus, minus operators indicating to add or to remove from the lookahead set. The above example has only one V_T: `eolr`. This terminal represents all terminals including itself in V_T. The `-` operator is generally used in conjunction with the `eolr` terminal just explained.

To make it easier for the grammar writer, local productions can be defined and referenced in this expression instead of sequentially writing a long list of terminals. Use of the rule in the expression means apply its first set under the set operator in effect. It is a bulking mechanism that makes life easy for the grammar writer and gives a descriptive sense to the action being taken. For example, if the boundary is composed of hexadecimal digits, a production can be defined with these digits and the expression just references the hex rule defined. The production does not have to be used in the grammar proper. It can just be defined strictly for the lookahead expression. Here are some examples to give you a feel in the expressive ease defining the lookahead set. Let's assume that V_T has these terminals: `0..9` and `eolr` where `..` reads through to.

Example 1: `eolr - 0`
This defines a lookahead set of `eolr` and `1..9`. Zero has been removed. Remember `eolr` indicates all the terminals in V_T.

Example 2: `Roctal + 8 + 9`
To help in understanding this example, the `Roctal` definition is given below even though Yacco2's P has not formally been defined. Its correspondence to the formal definition of P should be obvious to the reader.

```
1  Roctal AD AB(){  // Roctal production having 8 sub-rules
2    -> 0           // sub-rule 1
3    -> 1
4       ...         // 2..6 have been left out for space reasons
5    -> 7           // sub-rule 8
6  }
```

`Roctal`'s first set is 0..7 terminals. Using the addition operator for terminals 8 and 9, the lookahead set now contains the decimal digits.

3.3 Terminals Section — bring on the hors d'oeuvres

Terminals in general are entities that make up the language being parsed. It is the fodder to all other parsing passes. Now that's a nice general statement but... Let's look at the contents of a file to be parsed. To be more specific, it will be a character stream file of ASCII 8 bit variety. This does not detract from other file types which could be tree based. It makes it easier for me to build my argument. Its composition is made up of raw characters. These raw characters (RC) are converted to raw terminals with positional attributes. The end-of-the-character stream condition needs expressing though it is not a character in the recognized language. Yacco2 defines these type of situations as constant terminals: LRk terminals :}. Following this, the raw terminals are digested into concatenated entities represented by other terminal types in the food chain. Three things can happen in the digestion process: the raw terminals breaks into sequences of other terminal entities, the sequence is in error, and sequence end has been reached (another LRk situation). Of course each pass over the terminal sequences can evolve other terminals or meta-terminals that are terminal carriers.

The scene just described illustrates how Yacco2 classifies its terminals into 4 basic categories according to situations. These situations are classed as: special k terminals representing situations outside of the input stream, raw characters: I use this term as it deals with raw material, error, and your evolutionary terminals (terminals) that get transformed from raw terminal sequences or evolutionary parse sequences: for example, your basic stables

like identifiers, keywords and comments, to terminals of positional declaration caused by a later pass. This classification provides a simple framework to deal with V_T. LRk and RC are unvaried in their makeup. They are fixed and ready made: done once, used forever... With this classification, Yacco2 enumerates V_T by ranking the terminals in the following order: LRk, RC, evolutionary, and errors. From this, one can see that both error and evolutionary are dynamic and grow out as V_T is being developed. This growth evolves as more passes regurgitate and continue the terminal digestion process. This ranking is published globally across all emitted grammars which allows the grammar writer to specifically identify each terminal.

Yacco2's V_T is composed of 5 sequential parts. An enumeration component followed by the 4 individual terminal types. Each language construct contains the filename and namespace for the emitted file. For the terminal types, their appropriate declarations are given. Normally these definitions are brought into the grammar by Yacco2's '@' include file operator which allows nesting.

3.3.1 V_T Enumeration — And a one and a two...

Enumeration gives a specific identity to each terminal across the 4 terminal types. It is the most efficient tag in the terminal registration process manufactured by the compiler / compiler. There is also the literal identifier used in generating the terminal class's name: great for tracing purposes but very inefficient in lineup identification. As designed by C++ advice at the time, the C++ namespace facility was used for all the given reasons. The grammar construct is minimal in that it just provides the filename and namespace for the emitted C++ terminal enumeration. The manufactured code is included across all other grammar objects for the pushdown automaton and potential use by syntax-directed code. This is the internal dictionary of terminal terms that allows to differentiate abstract terminal objects during parsing. Below is the grammar construct for enumerating terminals introduced by `T-enumeration`.

```
/*
Purpose:
Supplies emitted enumeration file name and namespace.
The lr1 compiler/compiler generates the "enum" type for
the 4 classes of terminals:
  1) lrk
  2) raw characters
  3) evolutionary terminals
  4) errors
```

```
10
11 Note:
12 The symbol ordering is as follows:
13
14    0 <= Lrk < RC < evolutionary terminals < Errors
15 */
16 T-enumeration
17 (file-name yacco2_T_enumeration,name-space NS_yacco2_T_enum)
18 {}
```

3.3.2 Just show me

LRk will be developed in detail as it defines the parallel and conditional terminals needed in this paper. It also gives to the reader the cookie cutter attitude to the other 3 categories of terminals: error, RC, and evolutionary. The same pattern of definition is used for these others. Below is a snippet defining the LRk terminals introduced by `lr1-constant-symbols`. It has the same subroutine skeleton as previously described in Fsm with the additional enclosure bounding the terminal declarations by an open / close brace pair of lines 3 and 36.

```
lr1-constant-symbols
(file-name yacco2_k_symbols,name-space NS_yacco2_k_symbols)
{
  eog              (sym-class LR1_eog
    {
      user-declaration
        LR1_eog();
      ***
      user-implementation
        LR1_eog::LR1_eog(){
        T_CTOR("eog",T_LR1_eog_,0,0,false,false,0,0)
        }
        LR1_eog LR1_eog__;
        extern CAbs_lr1_sym*
               NS_yacco2_k_symbols::PTR_LR1_eog__ = &LR1_eog__;
      ***
    }
  )

  // the following terminals take the same pattern as eof
  // the material is edited for space purposes
  eolr             (sym-class LR1_eolr)
  "|||"            (sym-class LR1_parallel_operator)
  "|r|"  AD AB     (sym-class LR1_parallel_reduce_operator)
  "|?|"            (sym-class LR1_dynamic_operator)
  "|.|"            (sym-class LR1_invisible_shift_operator)
  "|+|"            (sym-class LR1_all_shift_operator)
  "|t|"  AD AB     (sym-class LR1_fset_transience_operator)

  lrk-sufx
    extern CAbs_lr1_sym* PTR_LR1_eog__;
    extern CAbs_lr1_sym* PTR_LR1_eof__;
       ... // editorial discretion exercised
    extern CAbs_lr1_sym* PTR_LR1_eolr__;
  ***
}
```

LRk defines all the situational terminals outside of the language being recognized. These situations are end-of-grammar: `eog`, elimination of lookahead set bloat: `eolr`, and an assortment of internal grammatical operators used within the rhs of P. The eolr symbol represents all symbols in V_T including itself. Its definition is a major convenience in the detailing of a thread's lookahead expression and in dieting of its set size. Each declaration

starts with its V_T symbol followed by its C++ emitted attributes. Source line 22 for terminal `eolr` illustrates a minimalist declaration. Source Line 24 shows two attributes, AD and AB, that can be associated with the terminal definition. They are house cleaning indicators. The AD attribute signals delete the terminal when popped from the parse stack typically used when the returned terminal from a thread is a meta-terminal carrier that gets thrown away after its contents have been extracted. The AB attribute indicates delete the terminal when the parse has aborted. AD comes into its own when parallel threads have partially parsed a phase and their remnants need cleaning up.

Source lines 4 to 18 defines `eog` and shows some syntax-directed directives that adds additional C++ code to its definition. Directives use the construction model of directive keyword, code, *** format. `user-declaration` and `user-implementation` are directives placing their code within the fabricated C++ terminal class. Within a terminal declaration and across the terminal classification, directives model a before / during / after an event. Lines 30 to 35 illustrates the `lrk-sufx` directive to place its C++ code at the end of the header file for all LRk terminals. Directives try to be self defining for anyone reading the grammar.

3.3.3 A little thing called...

Okay, what are these internal grammatical operators? Lines 23 to 28 defines them with their C++ class name giving an incline of intent. Only the important ones pertinent to parallel and conditional parsing will be reviewed. The design of the parallel symbols took the basic geometrical figure of two vertical lines indicating parallelism. Building on this, common symbols on the keyboard were sandwiched between the vertical lines '||' to mimic intent. Here are the relevant operators:

- `|||` operator introduces a thread expression in the rhs of a production.
- `|+|` operator — wild shift operator allows one to generalize the shift operation.
- `|.|` operator — invisible shift operator allows one to shift out of an ambiguous situation. It is a correctional symbol in fine tuning ambiguity.

The contexts of where these parallel symbols are deployed will be developed in the P section to follow.

3.4 Production Section — waiter, did I order this?

To get quickly into the Productions part, below is a Production construct of Yacco2:

```
1   // Productions: I use rules but it really is Productions
2  rules
3  {
4    Reol AD AB()     // 1st rule which defines S: Start rule Reol
5    {
6      -> Rdelimiters // rhs using rule Rdelimiters defined later
7    }
8
9    Rdelimiters AD AB() // Rdelimiters rule having 3 sub-rules
10   {
11     ->  "x0a"         // 1st rhs - raw char terminal: line feed
12     ->  "x0d"         // 2nd rhs - carriage rtn
13     ->  "x0d" "x0a"   // 3rd rhs - carriage rtn with line feed
14   }
15 }
```

From the snippet, `rules` introduces the Productions section to the grammar with its bounding open–close braces of source lines 3 and 15. The comments in the example should orientate the reader to Yacco2's karaoke of formally defined P, and the implicit definitions of S and V_N derived from P. Where the formal definition of a grammar defines the productions having a $lhs \rightarrow rhs$ structure, for efficiency reasons, the lhs is defined once followed by its enclosure containing one or more $\rightarrow\ rhs$ expressions. Each individual sub-production is unique; I refer to them as sub-rules. As grammars are written using basic keyboard symbols that are not turned into graphical symbols, I chose to simulate the $\rightarrow$ symbol by `->`.

The attributes AD and AB are as previously defined: house cleaning indicators for the pushdown automaton applied against the rule. Following the AB attribute are parentheses illustrated by source line 4 which contains nothing. Their purpose is to declare various types of syntax-directed coding blocks for the production. This will be explored later as it handles arbitration on results returned from threads.

3.4.1 rhs — strings of things...

From the formal definition, rhs is composed of symbols, possibly empty, drawn from V_T, V_N. The example below is of no difference to conventional

grammars except for the empty string which is usually represented by epsilon — ϵ in the formal definition, Yacco2 represents it by a blank *rhs*:

```
1    ->   0 "." Rfraction
2    ->   // rhs epsilon
```

An empty *rhs* can be placed anywhere within the list of sub-rules of a Production.

To express parallelism within a grammar, one restriction was placed on the *rhs*'s string of symbols: it has a fixed format of symbols and associated syntax-directed code. To simplify the parallel expression, the normal directive declaration was removed and its contents exposed. The parallel phrase is made up of three parts which are the parallel operator `|||` introducing the grammatical phrase, the terminal returned from the called thread, and the thread to be called which is the associated syntax-directed code. Below is a sample parallel phrase:

```
1  -> ||| "bad-char" NS_bad_char_set::TH_bad_char_set
```

The first two terminals, `||| "bad-char"`, are parsed like any normal *rhs* expression while the to-be-called thread name is associated with the finite state tables. The alert reader will notice that the associated code is not executed after the returned terminal but before, a slight twist on normal syntax-directed code execution which follows at the end of a grammatical phrase; a design decision made to be consistent with all other grammatical phrases. This phrase cannot be mixed with other symbols in its sub-production.

Now let's build on a parallel thread phrase. Multiple situations can be mixed within a production. There are no exceptions to their use within a production. It is just another type of grammatical string of symbols making up a sub-rule. Typical variations of use are:

- a thread can return different terminals.
- a thread recognizes many terminals and it's too expensive in coding to acknowledge all terminals returned. So, a meta-terminal is returned where by its contents contains the specific terminal recognized.
- multiple thread phrases are present but only one terminal will be returned.
- multiple thread phrases are present and arbitration is required to select which terminal is accepted.

Here is an example illustrating these variations:

```
1  -> ||| keyword NS_yacco2_keyword::TH_yacco2_keyword {
2    op
3      CAbs_lr1_sym* key = sf->p2__->keyword();// get rtned kw
4      ADD_TOKEN_TO_PRODUCER_QUEUE(key)
5    ***
6    }
7  -> ||| comment NS_c_comments::TH_c_comments
8  -> ||| "comment-overrun"        NULL
```

The above example shows 3 sub-rules all of which are calling threads. Source line 8 uses `NULL` to indicate that one of the other threads called in the production could return its terminal `"comment-overrun"`. Source line 1 is an example of a meta-terminal where `keyword` is returned and how the syntax-directed code block extracts the real keyword found. This should keep the curious interested.

3.4.2 Arbitration — grrrr

When disputing parties cannot come to an agreement, arbitration is needed. Parallelism does not have such emotional tantrums but it does need mediation. You might be wondering 'Why any judgement is needed at all?'. Simultaneous use of parallel threads can produce these situations. So mediation is dictated by the use of parallel threads within a production; no parallel going on, no arbitration required. Now, 'What happens when there are 2 or more threads being run and are successful in their parsing?'. As occurs normally in deterministic parsing there should only be one option open for execution. If two or more options are returned, how does one continue to parse? This is quite easy, associate at that point in the parse an arbitrator who decides the outcome. Based on the decision, that specific rhs containing the winning terminal continues parsing while the other returned terminals are destroyed.

To arrive at this decision point, two requirements are needed: a holding pool for the returned terminals that can be searched, and a unique identity per terminal which is supplied by the enumeration construct of V_T. The below example shows where and how arbitration is declared within a specific production. This is one part of the subset-superset problem mentioned in the previous paper where mediation determines the outcome if the keyword is present in the pool of returned terminals:

```
1  Rtoken AD AB
2  (
3    lhs
4    ,parallel-control-monitor
5      {
6        arbitrator-code
7          using namespace NS_yacco2_T_enum;
8          i = accept_queue->find(T_ENUM::T_T_keyword_);
9          if(i != ie){
10           accept_parse_parm = (*i).second;
11           goto arbitrated_paramater;
12         }
13       ***
14     }
15 )
16   {
17     -> ||| keyword NS_yacco2_keyword::TH_yacco2_keyword
18     -> ||| identifier NS_identifier::TH_identifier
19   }
20 }
```

In this example mediation is achieved by the opening up of the specific production's syntax-directed coding block: source lines 2 to 15. Within this block are comma delimited contexts noted by their names. The arbitration context is introduced by `parallel-control-monitor`. Following this are enclosure braces, lines 5 and 14, that contains the mediation directive `arbitrator-code`. Not to become too intimate with Yacco2's implementation, the example shows where and how the grammar writer's C++ code arbitrates. The code uses predefined variables and modules within the `generated Fsm class`, and terminal enumeration to support the mediation process.

3.4.3 Conditionals — Wild thing, I think I ...

Conditionals is a slight misnomer but I feel it relays to the reader its intent. It comes about when there is a ranking of activities within the parsing configuration where failure of one activity becomes the condition to try to continue down the parsing chain of the other potential activities. These ranked activities, as stated in the previous paper, are parallel parsing, regular terminal shift, invisible shift operator, 'wild shift' operator, and finally reduce operation . The following example shows both terminal types.

```
1  -> |+|        // wild shift
2  -> "x0d" |.|  // example of invisible shift
```

There is no restriction on their use. They can be placed anywhere within the string of symbols and without any limits to their number of appearances within the *rhs*. They are treated like any other grammatical expression.

`|+|` is the last shift terminal to be tried in the potential chain of parsing activities: potential shifts first, followed by reduce. As it is a wild card facility, this allows one to be general and not be specific in the terminals recognized. It is the last-chance shift activity to succeed before trying a reduce. In this context, two things can be programmed for:

1. catchall facility for error handling or a catcher of terminals that need recognition and are just passed through the parse process.

2. wild shift facility that shrinks the size of the grammar by generalizing on the terminals accepted rather than programming for each specific terminal.

This last-chance facility gives the grammar writer the chance to handle unexpected aborts. It can be programmed throughout the grammar thus giving a form of error processing controlled by the grammar writer. Care must be exercised when specific generalities sneak into the grammar. When multiple contexts of above are needed within the specific production, the grammar must either be rewritten to eliminate the split personalities by adding specific terminals as sub-rules, or use syntax-directed code in the catchall sub-rule to distinguish the various contexts. This facility is as useful as its other personality.

`|.|` is higher up the food chain in ranked activities. Its only use is to specifically shift out of an ambiguous situation. Normally it is a reaction to the compiler / compiler stating that a specific state's configuration is in trouble. I am always amused and surprised when it happens. Using `|.|` drastically lowers the debugging time in trying to resolve ambiguity. It is simple, elegant, and efficient as a solution to ambiguity: just shift it—an aphorism to scrub it out! It is explicitly programmed into the grammar. Where ambiguity is caused by a too general lookahead of a thread, instead of fine tuning the parallel thread's lookahead expression, use `|.|` to correct the situation and be more efficient in runtime and space of the lookahead set. Ambiguous resolution becomes fun when using this operator and threads with their tuneable lookahead expressions.

4 Conclusion

Hopefully this paper has taught and entertained you regarding the parallel parsing requirements within a grammatical context. Though not a lot of time was spent describing syntax-directed code, I believe the reader should feel comfortable with the points raised from experience with other compiler / compilers. [1] gives a reasonable, though a bit out of date, appendix on its design and implementation components.

Now onto the third paper in this series. Gentle reader, you should be armed to take it on. The raised questions from paper one will be addressed with their solutions taken from real situations. Bring on the desert so that the three courses can be judged.

References

[1] Bone, D. A syntax-directed compiler/compiler emitting lr(1) object-oriented code. Master's thesis, Concordia University, Montreal, Que, Canada, 1998.

[2] Hopcroft, J. E., and Ullman, J. D. *Formal Languages and their Relation to Automata*. Addison-Wesley Publishing Company, Reading, MA, USA, 1969.