Generating LR(1) and GLR grammars.

William Woody
Glenview Software
woody@alumni.caltech.edu

Abstract

This paper gives an outline of the techniques used to generate various LR style parsers, such as LALR and LR(1) grammars, and outlines the techniques behind GLR parsers. Originally intended to document how OCYacc works, this document has expanded into an exploration of the concepts behind LR parsers, and gives algorithms for generating LR(0), SLR, LALR and LR(1) state machines. We even touch on a method for constructing LR(k) algorithms and a potential method for using parallel processing for LR(k) algorithms.

Table of Contents

Introduction

This paper gives a survey of LR parsing techniques, for constructing the state machines and state tables used by a parser such as OCYacc.[1]

Originally this was to simply document the algorithms used by OCYacc. However, I've found that a number of papers on the topic of building a parser generator obtuse, to say the least. For example, in The Dragon Book[2], in the discussion of LALR parsers, the concept of a "kernel" is introduced quickly in a discussion that is difficult to follow, providing examples that sometimes are not very clear. It seems in order to follow the text to learn how to construct parser generators, you need to already know the subject!

My purpose with this paper is to describe the process in a way that is relatively clear and easy enough to follow, so that any reasonably competent developer can build their own parser generator tool. (Of course, "relatively clear" and "reasonably competent" are subjective, and it could be that after all this work, I will fail in my goal.)

As I wrote this paper, the size sort of got out of hand as the purpose of this paper shifted from documenting a single parser to providing some understanding of generating LR(0) parsers, Knuth's LR(1) parsing algorithm, SLR parsers and LALR parsers. And even now, as I rewrite this introduction with a better understanding of these ideas, I find the discussion given in various papers (such as the Lane Table Method outlined in Pager 2012[3]) opaque.

This paper also covers some simple error handling techniques, and GLR parsing,[4] which extends the LR(1) parser engine by allowing multiple actions to be taken at ambiguous points in a constructed LR parser table. This technique can help further reduce ambiguity for grammars that cannot be translated into an unambiguous LALR or LR(1) parser.

The paper concludes by putting it all together, with references to earlier locations in the document for the various algorithms that are used by OCYacc, including the LR(1) parser (with error handling) that is generated, and some documentation describing the generated Objective-C parser file.

Hopefully you'll find this paper useful.

---

[1] https://github.com/w3woody/OCTools

[2] Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D., "Compilers: Principles, Techniques and Tools", Addison Wesley Publishing Company, 1988.

[3] Pager, David; Chen, Xin: "The Lane table Method of Constructing LR(1) Parsers", APPLC'12, 2012.

[4] Tomita, Masaru, "Efficient Parsing for Natural Language", Kluwer Academic Publishers, 1986.

Implementation Notes

As I wrote this paper I tested the concepts with code that is now checked into the OCTools source kit on GitHub. The source code is contained in the files OCYaccTestXXX.cpp/h, stored in the directory ocyacctest.[5] You can examine the source code here as well as run the code if you wish. The final production code used by OCYacc is contained in the OCYaccLR1 class, which is similar in structure to the OCYaccTestLR1 class in our test directory, and uses the algorithm outlined in "Knuth's LR(1) Algorithm."

[5] https://github.com/w3woody/OCTools/tree/master/OCYacc/ocyacc/ocyacctest

Preliminary Concepts

Before diving into automatic parser generation, it is worth a quick review of some basic ideas from Computer Science. If you are already familiar with these concepts, feel free to skip this section.

How To Read Algorithms In This Paper.

Consider the following program fragment written in C:

```
for (int i = 0; i < 10; ++i) {
    printf("%d\n",i);
}
printf("Done.\n");
```

When we describe the flow of this program we would write the following:

```
Print numbers from 0 to 9.

1.     Set i = 0
2.     While i is less than 10
2a.    Print i.
2b.    Increment i.
3.     Print "Done."
```

The notation we use uses numbers and subsection numbers to indicate grouping of statements that are to be executed together. So, for example, the items "2a" and "2b" are part of step 2. In this case, they are repeated so long as the conditional in step 2 is true. If step 2 is false, we skip "2a" and "2b" and move onto 3.

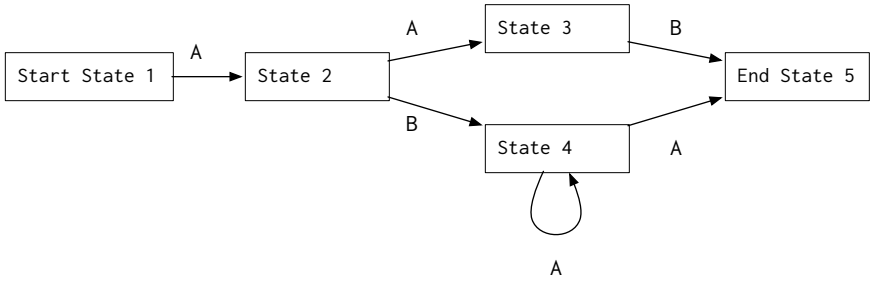
Sometimes we may add a final statement like "2c. Goto 2." between "2b" and "3" for clarity.

State Machines

A finite state machine[6] is a collection of states S combined with events E which transition from one state to the next. A finite state machine generally has a single start state S1, and usually has one or more end states E1 through En.

We can draw a finite state machine to help us understand the concept. For example, the following represents a state machine with 5 states, and two events A and B which drive the transitions from one state to the next.

[6] Finite-state machine, https://en.wikipedia.org/wiki/Finite-state_machine



In the state machine above, if we see the events A, A and B, we would start at state 1, then move to state 2, then to state 3 and finally to state 5. Any event that does not correspond to a transition would be an error; for example, if our list of events start with B, we have nowhere to go from state 1.

Deterministic verses Non-Deterministic State Machines

Look at our example above, at state 4. This state is non-deterministic, because if we are at state 4, and we see event A, then what do we do? Do we go to state 5? Or do we loop back to state 4?

This gives us the following definitions:

Definition: *A deterministic finite automaton (DFA) is a state machine where every state has at most one and only one events associated with it.*

In our state machine above, if we were to remove the looping transition in state 4, we would have a DFA: for every state, for every event, there is at most one action that can be taken.

Definition: *A non-deterministic finite automaton (NFA) is a state machine that contains at least one state with multiple transitions associated with the same event.*

In our state machine above, state 4 is ambiguous: when we see event A we cannot determine if we should loop back to state 4 or move to state 5.

Powerset Construction Algorithm

We can convert a non-deterministic finite automaton (NFA) to a deterministic finite automaton (DFA) by using the powerset construction algorithm.[7]

The idea here is simple: we construct a new set of states for our DFA where every state can represent one or more states in our NFA. Can't decide if we loop back to state 4 or move to state 5? Construct a new state that represents both states 4 and 5! And this is guaranteed to generate a state machine because the maximum number of possible DFA states is 2^n states, where n is the number of NFA states; thus, the name "powerset construction."

---

[7] Powerset Construction, https://en.wikipedia.org/wiki/Powerset_construction

This idea becomes important as we construct state machines from our grammars; each state in our grammar is a rule "R -> A + B" for some production R and grammar symbols A, + and B, combined with the current position in the rule that we're reading. So, if we're reading the input tokens "1 + 2", and we've already read the plus sign, then our state in the grammar would be "R -> A + . B", where the period marks where we are in our grammar as we're reading our input stream.

*More importantly, this idea helps when parsing multiple rules at the same time. For example, consider the two rules n + n and n * n for some number n. If you are parsing the expression 1 + 2, and reading tokens one at a time, and have only read the 1--when reading the next character, you could read a '+' or a '*' symbol. The Powerset construction idea combines the two possible rules together into a single state, one which represents reading either a '+' for n + n, or a '*' for n * n.*

The idea of using a Powerset-like construction algorithm to build a parsing state machine is, in fact, the subject of the rest of this paper.

Our Powerset Construction Algorithm is below. Note that we assume our NFA has no empty transitions; see the Wikipedia article for more information.

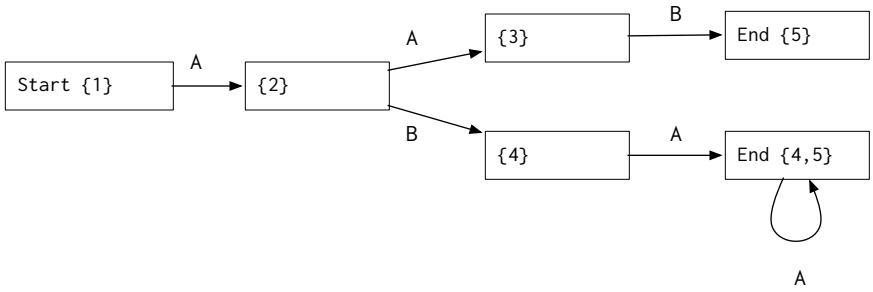
```
Powerset Construction

Given an NFA N with start rule N0
Return a DFA D

1.     Set S an empty set of DFA states. Set T an empty map of transitions on
       event Ei in our events E to a new state Si in S.
2.     Create a queue Q of items to process, and create a new set V of visited
       items.
3.     Add a new set S0 = { N0 } into S. Add S0 to Q and V.
4.     If Q is not empty
4a.    Remove Si from Q.
4b.    For every state Ni in our state Si, find all events E that transition out
       of our states Ni.
4c.    Crate a new empty set Sj. Sj will be our new DFA state.
4d.    For every event Ei in E
4d1.   For every state Ni in Si, find all states Nj that transitions from Ni
       through Ei, and add to Sj.
```

For example, in state 4 of our example NFA above, for event A there are two possible destination states: 4 and 5. We then add both 4 and 5 to our new set Sj.

After we execute the loop 4d, we now have an event Ei and a new set Sj; Sj represents the destination state from Si through Ei.

```
4e.    If Sj is not in V
4e1.   Add Sj to V. Add Sj to Q for later processing.
4e2.   Add Sj to S, our set of DFA sets.
4e3.   Add the transition (Si, Ei) -> Sj to our map of transitions T.
4f.    Go to 4.

5.     Return our new DFA as the set S of constructed sets, and our transitions T.
```

When this algorithm is applied to our original example, we get the new state machine:



We create a new state which is the combination of states 4 and 5. Because it contains the end state 5, this is a potential end state as well.

This idea, that we can combine non-deterministic states into deterministic states in order to create a deterministic finite automaton which a computer can then follow like a recipe to perform an operation, will come back as we construct our LR(0) and LR(1) state machines.

Standard Iteration Algorithm

It is worth a quick note that most of the algorithms below follow a very similar pattern:

```
1.    Create a queue Q of items to process. Add an initial state S0 to Q.
2.    Create a set V of items that have been visited, and add S0 to V.
3.    If Q is not empty
3a.   Remove Si from Q.
3b.   Do some process P on Si which returns 0 or more items Sj...Sk. This process
      may perform side effects for our return value.
3c.   For each Sn in Sj...Sk, if Sn is not in V, add Sn in V and Q.
3d.   Goto 3
4.    Return our result.
```

This pattern is suitable for any process which attempts to discover objects within a finite space S that can be traversed using a process P from one state Si to another set of states Sj...Sk.

The idea here is simple. Q contains a list of items that we haven't processed yet, and V guards that list by preventing us from inserting items that have either been processed, or are about to be processed. If the space S is finite, this process is guaranteed to halt, if only because eventually, V = S, and no more items will be inserted into Q for processing.

We can use this algorithm with few modifications, for example, to traverse a map of streets attempting to travel to another location (where P includes the side effect of early exit), or to flood fill a bitmap. For us, we use this essential pattern for discovering nodes in a state machine that represents our parsing algorithm.

Each algorithm for constructing a state machine, from LR(0) to LR(1) listed below, follow this pattern, sometimes with modification. And we used this pattern with our powerset construction algorithm, above.

Introduction to LR Parsers

LR parsers[8] are a type of bottom-up parser which can be used to handle parsing an input language in linear time. The term "LR" means that this category of parsers read files from Left to Right, producing a Rightmost derivation in reverse, parsing the input file from bottom up.

In other words:

Left to Right We read the input file from left to right, from the first symbol to the last. (Yes, there are parsers which may read input files in right to left order, or in arbitrary order. We don't consider them here.)

Rightmost We find rules bottom-up; that is, we find the most specific statements of our input language first (such as "1+2"), increasingly parsing larger and larger components of our language (such as "a = 1+2;" and "if (test) { a = 1+2; }") until we finish by putting together the top of our parsing tree. It's considered "rightmost" because for most grammars, the right-most tokens are generally towards the bottom of the parse tree.

(LL parsers, such as Antlr[9], produce Leftmost derivations, and parse top-down: considering the full "if" statement first, then considering the assignment expression, then the addition operation.)

The category of parsers we are generally discussing in this paper are LR(k) parsers. LR(k) parsers read symbols form left to right one at a time, and when considering if we have seen a completed grammar rule (such as "1+2" or "a = 1+2;"), will only consider k symbols at the current position in the file to determine what to do next.

This paper considers several classes of LR parsers: LR(0) parsers (which provide the template for all the rest of our parsers, as well as introduces the concept of parsing), LR(1) parsers, SLR parsers and LALR parsers.

The LR(0) Parser

The LR(0) parser is one of a category of LR parsing techniques for parsing a file by reading it from left to right, constructing a right-most derivation of the grammar, using a "shift" (that is, read the next token) and "reduce" technique (that is, replace a list of tokens with the matched production) which can be used to execute chunks of code during the reduction step.

(That is, it allows us to write YACC-style files where each rule corresponds to code; as we find each rule we "reduce" the rule--and execute the corresponding code as we do a reduction.)

---

[8] https://en.wikipedia.org/wiki/LR_parser

[9] http://www.antlr.org

An LR(0) state machine is so-called because it does not look ahead in the list of tokens in order to make decisions about reducing rules. Of course, LR(0) state machines are very weak; from a practical perspective few (if any) reasonably large grammars can be unambiguously parsed using an LR(0) state machine. To see why, consider the following grammar for parsing simple addition and multiplication statements:

```
expression :  addexpr
           ;

addexpr    :  mulexpr              { $$ = $1; }
           |  addexpr '+' mulexpr  { $$ = $1 + $3; }
           ;

mulexpr    :  NUMBER
           |  mulexpr '*' NUMBER   { $$ = $1 * $3; }
           ;
```

The structure of our grammar groups multiplication operations over addition; thus, the statement 1+2*3 is grouped 1+(2*3), and 1*2+3 is grouped (1*2)+3.

(Notice we also include the reduction code to illustrate what we mean by "reduction"; as we parse, for example, the multiplication statement we also calculate the result of multiplying two values.)

Intuitively as we parse from left to right the following expression:

```
1 + 2 * 3
```

If we have already seen the subexpression "1 + 2", and we are not permitted to look ahead to the next '*' symbol, how can we determine that we need to read the next symbol rather than collapse the current subexpression? We need to know the next symbol '*' follows our subexpression--but LR(0) state machines do not read ahead.

Which is what motivates our later state machine algorithms, including LR(1), SLR, LALR and the like.

Augmenting The Grammar

The first step in building our LR(0) state machine is to augment the grammar. In essence, we create an initial implicit rule which handles the end of file token. (Notice in our grammar above, the end of file is implicit rather than explicit in the definitions. It also helps because in the process we create a single end state, unlike our powerset example above, where there were multiple potential end states.)

This technique is used in Bison[10], and is described in The Dragon Book, Section 4.7.[11]

---

[10] Popuri, Satya Kiran, "Understanding C parsers generated by GNU Bison", https://www.cs.uic.edu/~spopuri/cparser.html

[11] Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D.; "Compilers: Principles, Techniques and Tools"

To our original grammar we add the following start rule:

```
$accept    :  expression $end
           ;

expression :  addexpr
           ;

addexpr    :  mulexpr              { $$ = $1; }
           |  addexpr '+' mulexpr  { $$ = $1 + $3; }
           ;

mulexpr    :  NUMBER
           |  mulexpr '*' NUMBER   { $$ = $1 * $3; }
           ;
```

This statement is the implicit start statement of our grammar. The '$end' symbol represents the end of file; typically the token value -1 when using OCLex or other lexical parsers.

For the rest of the discussion we also rewrite our grammar into a list of rules, and abbreviate each of the items above, so it's easier to draw the diagrams.

```
0. S -> E $
1. E -> A
2. A -> M
3. A -> A + M
4. M -> n
5. M -> M * n
```

This is the same as our grammar above; we replace '$accept' with 'S', '$end' with '$', 'expression' with 'E' and so forth.

Terminology: *We use the term "rule" to refer to a single rule, such as "A -> A + M".*

The rule index is the number of the rule in our list of rules, and for our examples, we assume that index starts at 0.

We use the term "production" to refer to a single non-terminal token that is the target of each rule. For our above grammar, our productions are S, E, A and M.

Some of the discussion below may conflate a rule P and a production P; by this we mean the production the rule defines. So, for example, "A -> B + C" would be rule which defines production A.

We use the term "terminal" to refer to a terminal token; that is, a symbol that we get from our lex tool. For our grammar above, our terminals are n (a number), '+', '' and $ (end-of-file).*

Terminals and productions are sometimes collectively referred to as grammar symbols.

Some Useful Definitions

Before we proceed with our algorithm for constructing an LR(0) state machine, it is useful to define a few things.

Rule Items

A rule item, LR(0) item (as used) or just "item" of a grammar is a rule in our grammar and a location within our grammar.

Intuitively, this represents our state while parsing a rule. For example, with the rule:

```
3. A -> A + M
```

It would be useful to know where we are in the parsing process, for example, if we are about to look for the '+' sign while parsing.

Typically in textbooks and papers, items are represented with a period at the point just before the next symbol we are seeking to parse. So, in our above example, we would write our rule combined with the fact that we're looking for a '+' sign as:

```
3. A -> A . + M
```

When we start parsing, we start with our 0th rule at the very start of our rule:

```
0. S -> . E $
```

Once we're completely done parsing our grammar, we should be at the end of our start rule:

```
0. S -> E $ .
```

> *Implementation Note: We represent each item with a pair of integers. The first is the rule number. The second is an index in the list of grammar symbols of our rule, from 0 to len, with len being the total length of the contents. So, in our example above, our start would be the pair (rule = 0, position = 0), and the item representing our rule 3 just before the + sign would be (rule = 3, position = 1).*

Item Sets

As we parse our grammar, there are times when the rule we're parsing could be ambiguous. It's therefore useful to represent the set of all the rules we may be parsing at a given state. (This idea is related to the powerset construction algorithm above.)

We do this by using item sets.

An item set is basically a set of items, stored as a single object. For example, it can be the set of items:

```
1. E : A .
3. A : A . + M
```

Intuitively, this represents the point in parsing an expression where we've seen an addition statement, and up next could either be another plus sign, or the end of the addition expression.

> *Observation: It is this process: combining a number of non-deterministic rules in a deterministic state machine, which represents the power of these algorithms to generate an efficient parser.*

Item Set Closure

Consider, for example, the start rule at the start of parsing:

```
0. S -> . E $
```

In order to parse this statement we will need to consider all of the other rules that we may need to examine as we parse this statement.

That is, in our above statement, we are about to parse the statement E. That means we are about to parse the item:

```
1. E -> . A
```

And the item above means we're about to parse the items:

```
2. A -> . M
3. A -> . A + M
```

Rule 2 also means we're about to parse the items:

```
4. M -> . n
5. M -> . M * n
```

All this means if we're about to parse the expression E, we may be about to parse "1+2" (rule 3), or "5" (rule 4), or "5*3" (rule 5). We need to roll all this up into a single item set so we can track all of these possibilities.

That process of rolling up all the possibilities is called "closure."

Our closure algorithm is as follows:

```
Find Closure.

Given IS an initial set of items I
Return the closure of (IS)

1.     Add all items IS to our closure C.
```

```
2.    Create a queue Q with the list of all productions that follow the current
      position in each rule.
```

For example, if we have an item "A -> A + . M" in our set, the production referred to here would be "M". If the next grammar symbol in our item is a token, the token is ignored.

```
3.    Create set V of productions in Q. (The set V helps us track productions
      we've already processed.)[12]
4.    If Q is empty, go to 5.
4a.   Remove production P from Q.
4b.   For all rules for production P, construct items which represent the start
      of parsing rule R for production P, and insert into closure C.
```

*For example, with production M, we construct items "M -> . n" and "M -> . M * n".*

```
4c.   For all the newly constructed items in 3b above, if any start with a new
      production rule P'
4c1.  If P' is not in V, the set of processed productions, insert P' into our
      queue Q and into V.
4d.   Go to 4.

5.    Return closure C.
```

When we apply our closure algorithm on our initial set:

```
0. S -> . E $
```

We get the item set:

```
0. S -> . E $
1. E -> . A
2. A -> . M
3. A -> . A + M
4. M -> . n
5. M -> . M * n
```

This set is represents a state in our resulting state machine. That is, it is assigned a state number, and treated as a single object for the rest of our LR(0) algorithm.

Our LR(0) Construction Algorithm

Our LR(0) construction algorithm boils down to finding all of the item sets in our grammar combined with all the transitions to new item sets. This represents the behavior of our parser as we read characters: if we see a '+', we're in the middle of parsing addition, and the next set of rules (the item set) is then constrained by the fact that we're reading a '+' sign.

In brief, our algorithm acts similarly to the powerset construction algorithm above:

[12] Standard Iteration Algorithm

```
1.     Construct the initial state from our initial rule.
2.     For every token and production in our grammar
2a.    For every state in our set of states
2a1.   Find the set of items we transition to with the grammar symbol we're
       looking at.
2a2.   Find the closure of the new set of items.
2a3.   Note the transition from the original item set to the new item set through
       the token or production.
2a4.   Insert our new item set into our list of sets.

Repeat until done.
```

When we're done, we should have a list of states and a list of transitions which represent the state machine used while parsing our grammar.

So, given our grammar:

```
0. S -> E $
1. E -> A
2. A -> M
3. A -> A + M
4. M -> n
5. M -> M * n
```

We first construct our initial state using the closure algorithm above for our initial item "S -> . E $" (which means "we're starting parsing our file"):

```
0. S -> . E $
1. E -> . A
2. A -> . M
3. A -> . A + M
4. M -> . n
5. M -> . M * n
```

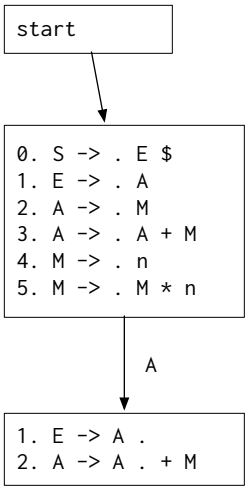
Now we loop through all our tokens ("n", "$", "+" and "*") and our production rules (S, E, A, M), and our current set of item sets. We then construct the items a particular token transitions to for a given item set. So, with our item set above, suppose we're examining production A.

There are only two items in our item set which can advance when we see production A:

```
1. E -> A .
3. A -> A . + M
```

Next, we perform a closure on this list of items. (In this case, it happens the closure is equal to the original item set.)

And this gives us the following subset of our overall grammar:



Of course there are some optimizations we can perform.

First, as we do with the closure algorithm, we can maintain a queue Q of items we've yet to process, and a set V of items which we've already processed. This allows us to track the things we have yet to deal with, and track the things we've already completed. A general form of this idea is given in our introduction.[13]

Second, we can examine the item set to find the list of tokens and productions which are possible from that item set. In other words, if we are looking at the item set:

```
1. E -> A .
3. A -> A . + M
```

By inspection we can see that the only possible token that will do anything is the '+' token.

We combine these two observations for our final algorithm.

```
Find LR(0) state machine.

Given G the augmented grammar, with rule 0 the inserted grammar S -> R $, with R the
starting production named in the YACC file.

Return the LR(0) parsing table, consisting of a set GS of item sets, and a list of
transitions T between the item sets in GS.

1.     Construct the initial item set IS = closure(I), with I the item S -> . R $
2.     Construct Q = [ IS ] a queue of item sets to process
3.     Construct V = Q, the set of item sets already processed.
4.     Construct T as an empty list of transitions t(IS,symbol) -> IS'
5.     If Q is empty, go to 6
5a.    Remove IS from Q
```

[13] Standard Iteration Algorithm

```
5b.    For every item I in IS, find PS = the set of tokens and productions
       referred to by the current position in each item.
```

For example, if the item being examined is 'A -> A . + M', the token for this item is '+'. If we have a item 'E -> A .', there is no token associated with this item, and the item is skipped.

```
5c.    For each token or production P in PS,
5c1.   For each item I in IS, add to a new item set IS' each item I' which
       represents the transition I through P.
```

For example, if I is 'A -> A . + M', and P is '+', then I' is 'A -> A + . M'.

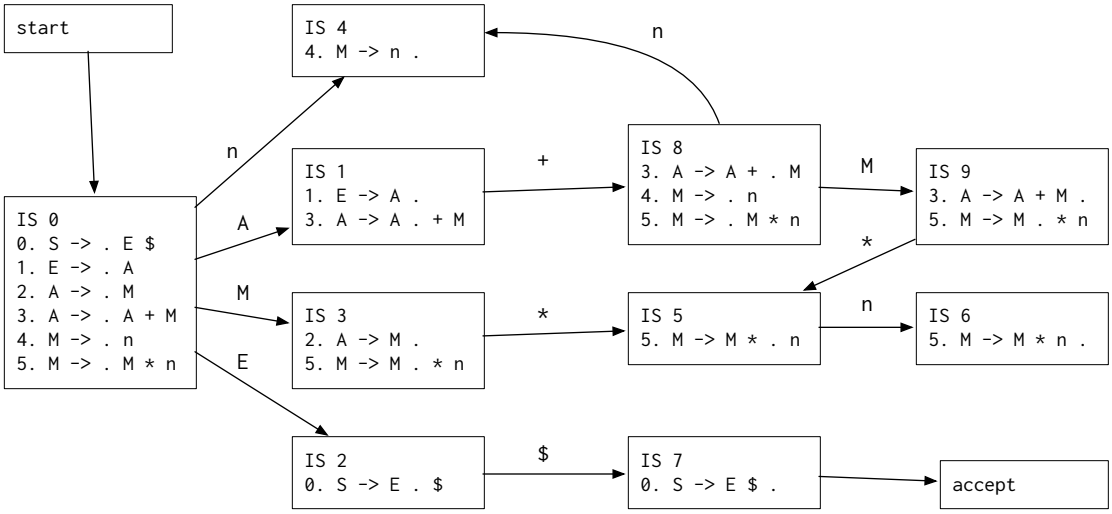
Implementation note: in our code base, we collapse steps 5b and 5c into a single step by constructing a mapping M which maps a token or production rule P to the item set IS' being constructed. This allows us, in one shot, to figure out the P we're transitioning through for each I, and accumulate I' into a new IS' as we construct them.

```
5c2.   Find C = the closure of IS'.
5c3.   If C is not in V, add C to V and to Q.
5c4.   Add transition t(IS,P)->IS' to our list of transitions T.
5d.    Goto 5.

6.     Return LR(0) as I, the set of item sets found, and T, the list of
       transitions between item sets.
```

Incidentally, this pattern: iterating a queue of states to process, finding the next state by transitioning the items within an item set through a given token, is used for all of our parser generators. The differences between the different state machine generation algorithms all have to do with the way we construct the states and the information we store in the states as we generate them.

LR(0) Shortcomings and The Motivation To Build A Better State Machine.

When we apply this algorithm to our original grammar, we get the following state machine. This state machine represents the total set of item sets and transitions for our grammar, found using our algorithm's implementation.



The shift operations are obvious; they are the state events that transition from one item set to another through a specified token or production rule.

The reductions—that is, in our parsing process, the points where we recognize a rule and reduce the rule to the production that rule represents—can be found with any item set that has reached the end of a rule.

For example, the item set 6 in the state machine above:

```
5. M -> M * n .
```

This shows that we've parsed a complete multiplication expression, and we can reduce our parse state from something like "2 * 3" down to "M". (As a side effect of this reduction we would execute the segment of code "$$ = $1 * $3" in our original YACC-style declaration above.)

To understand what we mean by "shift" and "reduce", it is worth a side journey into how we will use this state machine to build our parser.

The LR(1) Parsing Algorithm

For more details see the discussion below. But for now it is worth a quick note as to how we intend to use our state machine by outlining the process for processing a stream of tokens.

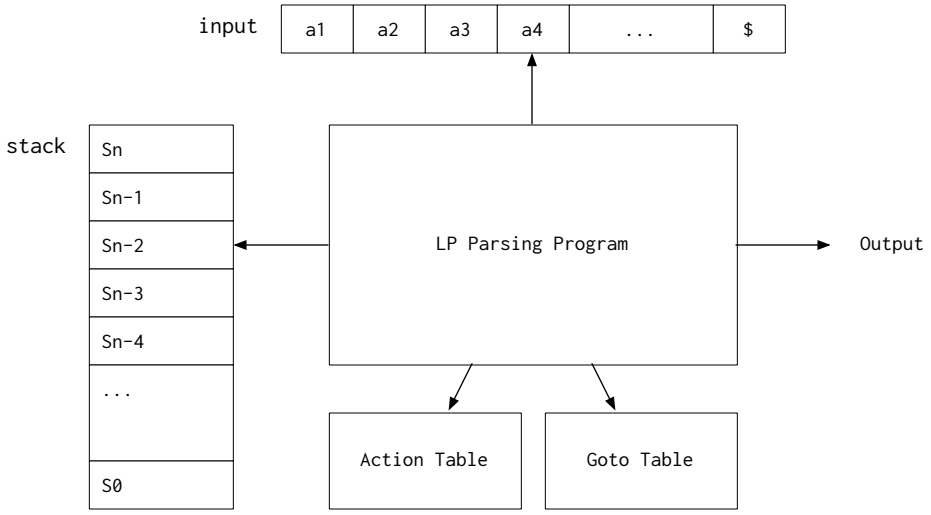
Our LR(1) parsing algorithm consists of a stack, an input list of tokens (i.e., from our OCLex reader), and a state machine consisting of an array of action and goto tables. In the traditional Dragon Book discussion[14], items on the stack consist of the state index (the index we assign to each Item Set as we construct them), and an integer representing the token or production rule being parsed. For the OCYacc implementation, we we

[14] Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D.; "Compilers: Principles, Techniques and Tools", Section 4.7: The LR Parsing Algorithm

store more than the token and state on our stack. We store the file location (including the line and column) where each token was read, the token string read, and we also track the value associated with each token or production rule.

This corresponds to the $$ or $nn (where nn is an integer from 1) symbols in the code associated with each rule that gets executed during a reduction.

The algorithm manages a push-down stack of items that include the current state (the index of each Item Set), and reads from a stream of input tokens, following a state machine defined by an action table and a goto table to either read the next token ("shift"), or "reduce" a rule (and execute the code associated with the rule).



The parsing program is given in the following algorithm. Note that we do not show how we handle error states; that is later in this document.

```
LR Parsing Algorithm

Given A an action table of items A(state,token) -> action, where action is either "shift
state" or "reduce rule", G a goto table of items G(state,production), and an input A
consisting of symbols a1...an $, with $ representing the end of the file,

Our algorithm will parse the file. (As a side effect, we will execute the reduction rules
associated with our grammar.)

1.     Build a stack S and insert the initial state (an integer representing Index
       Set, and by construction assumed to be 0) onto the stack. Let ip point to
       the first symbol in input A.
2.     Let s be the state on top of the stack, let a be the symbol pointed to by
       ip.
3.     If A(s,a) = shift s' then
3a.    Push s' (and associated context) onto stack S.
3b.    Advance ip to next symbol.
3c.    Goto 2.

4.     If A(s,a) = reduce rule, with rule of form P -> t1 t2 ... tl of length l.
4a.    Execute code associated with rule.
```

Note that as we execute the rule, any value stored in S is accessed as follows: suppose the rule is of length l. Then $1 is stack item l-1 from the top of the stack, $2 is stack item l-2 from the top, and $l is the top of the stack. The value $$ is a special value that will be pushed later onto the stack.

```
4b.    Pop l symbols from the top of the stack.
4c.    Let s' = the new top state of the stack.
4d.    Push state G(s',P) (and associated context including new value of $$) onto
       stack.
4d.    Goto 2.

5.     If A(s,a) = accept
5a.    Finish.

6.     If none of conditions at 3, 4 or 5 hold true, then perform error
       processing. (See later in document for discussion.)
```

Notice that our process assumes from our LR grammar we generate a goto table which expresses how we handle the new state after parsing a reduce, and an action table which indicates if we shift or reduce based on the current top token on the stack.

Extending to LR(k) by separating the action table into two tables.

As a side note, notice that we use a single action table to handle both the transition from a new state during a shift operation, and how we handle a reduce statement.

We can combine these two actions into a single table by virtue of the fact that we are only looking at most one symbol ahead in the list of input symbols.

Intuitively it seems if we were to use a look ahead greater than 1, we would want to separate the action table into two separate tables, one for shift and one for reduce.

For shifting we would want to maintain a shift table that gives the appropriate new state during a shift operation. As shift operations represent the process of matching a token with a grammar symbol in one or more rules, intuitively it would seem that our table SA of shift actions should represent the transition from one state to another by processing a single token. Our LR(1) parser algorithm's behavior at step 3 and 6 would remain the same.

Our determination of acceptance can also change. Note that in our example LR(0) state machine above, once we reach Item Set 7 (or state 7), we've reached the end of our grammar. That is, we've read the terminal symbol '$' in our shift from state 2 to state 7. So we can redefine step 5 as testing if our state is at the end state.

```
5'.    If s = es, the end state in our grammar
```

(This is an optimization taken by Bison.)

Reductions are handled differently. As we are looking ahead k symbols for k > 1, we would need to construct a table RA of reduce actions that give a reduction rule given a state s and a list of tokens [a1, a2, …, ak] representing the top k tokens pointed to by ip.

So, generalizing our LR(1) parser algorithm, we would substitute rule 4 with:

```
4'.    If RA(s,[a1,...,ak]) = reduce rule, ...
```

Of course we would need to handle the case where we are at the end of the file by defining any symbol past the end of the list of input tokens as an end of token symbol $.

This idea, of separating the action table in the traditional LR(1) parser into two separate tables, one for shift operations and one for reduce operations, can also apply for an LR(0) parser, with our RA actions determined only by the current state s'. However, as our interest in this paper is to construct an LR(1) style parser, we will maintain the custom of a single combined action table.

Constructing the Action and Goto Tables from our LR(0) state machine

We can construct our action and goto tables for the machine above from our LR(0) state machine through the following process:

1. Build an empty action table A = a(IS,P) -> action, where IS is an item set (or the index number of an item set), P is a token in our grammar, and action is one of "shift state", "reduce rule", "error" or "accept." By default fill this 2-dimensional table with error.
2. Build an empty goto table G = g(IS,P) -> state, where IS is an item set (or index), P is a production rule in our grammar, and state is the new item set (or index) we transition to. By default fill this 2-dimensional table with empty states.
3. Populate G with g(IS,P) with the transitions in our state machine that pass through a production rule P.

In our state machine above, the state transitions that pass through the production rules E, A and M are transitions like the one from IS 0 to IS 1 above. This results in the table G equal to:

	E	A	M
0	2	1	3
1			
2			
3			
4			
5			
6			
7			
8			9
9			

4. Populate A with a(IS,P) with the transitions in our state machine that pass through terminals in our grammar.

We do this in the same way as we do 3 above. This results in the action table A equal to:

	$	*	+	n
0				S4
1			S8	
2	S7			
3		S5		
4				
5				S6
6				
7				
8				S4
9		S5		

Note that in the table above, we use the notation "A" as "accept", "Sn" to mean "shift to item set n", and "Rn" to mean "reduce by rule n".

5. Now add the reduction rules. For our LR(0) state machine, if an item set contains an item where our parser position is at the end of the rule, then we reduce by that rule.

Reduction means we do two things. First, we replace on our stack the top set of symbols we've parsed with the collapsed production represented by that rule. Then, we transition to a new state depending on the symbol we found at the top of the stack. That transition is given in our goto table G, already constructed above.

Intuitively, this two-step process can be seen as we've found our rule, so we can replace our rule with the production representing the rule (and execute the code associated with the rule), and now the next symbol we're handling is the production we just collapsed into.

*So, for example, if our parser just saw the symbols "1 * 2", we first reduce this to the production "M", and then we look at the previous state in the stack (say, state 8), and we carry the transition of state 8 through production "M" to state 9, as if we just read "M" off the list of input symbols.*

If we detect the reduce of our original start rule (which, by convention, is rule 0), instead of reducing we mark the input stream as accepted. "Acceptance" means we're done and our state machine can terminate.

For our example grammar, our reduce rules look like:

	$	*	+	n
0				S4
1	R1	R1	S8/R1	R1
2	S7			
3	R2	S5/R2	R2	R2
4	R4	R4	R4	R4
5				S6
6	R5	R5	R5	R5
7	A	A	A	A
8				S4
9	R3	S5/R3	R3	R3

Because we do not take into account the token we're examining, the reduce rule spans the entire state row of the table, and would be equivalent to the LR(k) reduce table discussed above, with k = 0.

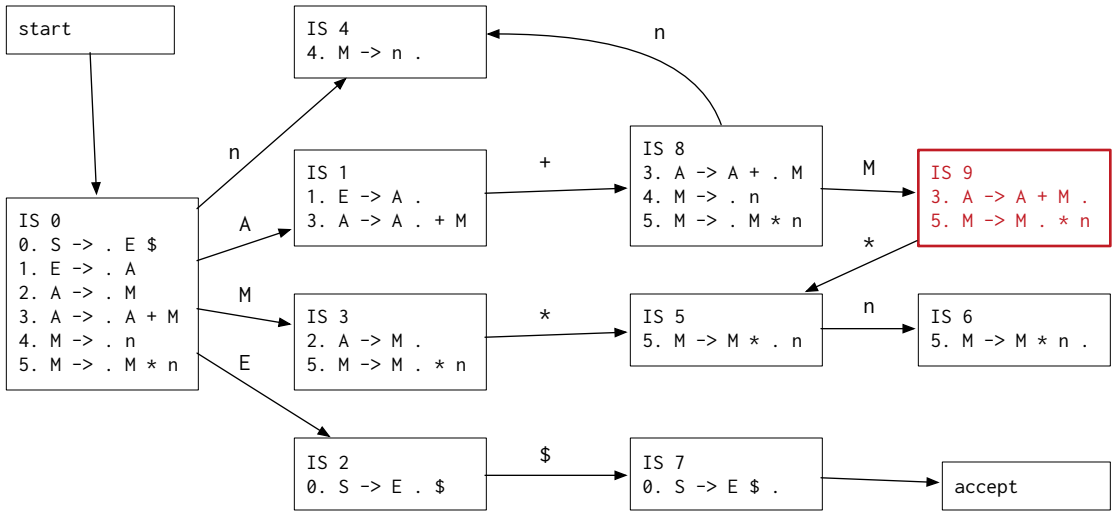
Notice the states where we have both a shift and a reduce.

6. `Return our final tables A and G.`

Revisiting the LR(0) problem.

At this point you may or may not have noticed things have gone haywire with our grammar.

For example, notice state 9, highlighted in red.



We have a problem here, called a "shift/reduce" error.

The problem is that the state is non-deterministic: if we see a '*', do we shift by '*' to state 5? Or do we reduce the rule "A -> A + M"?

This is a concrete illustration of our intuitive problem presented earlier:

> *If we have already seen the subexpression "1 + 2" while parsing "1 + 2 * 3", and we are not permitted to look ahead to the next '*' symbol, how can we determine that we need to read the next symbol rather than collapse the current subexpression?*

Shift/Reduce and Reduce/Reduce Errors.

The above is an example of a shift/reduce error. It represents any state in our constructed state machine where there is uncertainty: where a state could either shift to a new state, or be reduced to a different production.

There is a second category of errors, reduce/reduce errors, and as you would guess, that is any item set where it is unclear which rule should be reduced. For example, from Pager 1973[15], given the grammar

```
0. S -> E $
1. E -> a A d
2. E -> b A c
3. E -> a B c
4. E -> b B d
5. A -> e A
6. A -> e
7. B -> e B
8. B -> e
```

We eventually construct state 8, which illustrates both a shift/reduce and a reduce/reduce error. (See figure 3 in the above referenced paper.)

```
A -> . e A
A -> e . A
A -> . e
A -> e .
B -> . e B
B -> e . B
B -> . e
B -> e .
```

The reduce/reduce error can be seen by noting the two items in our set: "A -> e ." and "B -> e ."; do we reduce by rule 6? Or rule 8?

(And even more perverse, if we see an 'e' in the list of symbols, we also have a shift/reduce conflict: do we shift by 'e' to state 8 again? Or do we reduce by rules 6 or 8?)

> *Note: We will never have shift/shift errors. A shift/shift error would imply the same token or production rule could transition from one Item Set into one of two different potential Item Sets. But during construction of new Item Sets after a transition, any potential ambiguous Item Sets are combined into a single set prior to Item Set closure.*

Resolving shift/reduce and reduce/reduce errors via an ad-hoc mechanism.

YACC and other tools provide tools for resolving shift/reduce and reduce/reduce errors via an ad-hoc mechanism. For example, YACC provides the "%left" and "%right" declarations which declare a token as either left-associative or right-associative, to decide on a shift/reduce error if we shift, or if we reduce.

But notice that this mechanism doesn't really help us here. From or original example as we parse the expression "1+2*3", because we do not look ahead by a token to notice the '*' token after parsing '1+2", we have no way to examine the rule to determine if it is appropriate to shift or reduce. The best we can do is either

---

[15] Prager, David: "The Lane Tracing Algorithm for Constructing LR(k) Parsers", Proceedings of the fifth annual ACM symposium on Theory of Computing, 1973., Figure 3.

always reduce—giving us the wrong grouping "(1+2)*3", or always shift—which gives the wrong grouping for the statement "1*2+3".

We need a different algorithm, one which can look ahead a symbol, and which can help us resolve the grammar shortcomings of an LR(0) state machine.

Once we are able to look ahead a symbol, we can revisit resolving shift/reduce and reduce/reduce errors.

Knuth's LR(1) Algorithm

Donald Knuth's algorithm for generating an LR(1) state machine attempts to solve the problem of shift/reduce and reduce/reduce errors by tracking the tokens that are expected to follow each rule item as we construct our item sets. This allows us to know which rule we should reduce by given the current token in the input stream; thus, our state machine is considered LR(1) in that it looks ahead by 1 token.

In the following discussion we use the same grammar we used when constructing our LR(0) state machine. Our discussion is to help understand how we resolve conflicts in our LR(0) state machine to create a state machine for our LR(1) parser without all the excess states generated by Knuth's algorithm.

As a reminder, the augmented grammar is below, and specifies a simple parser that parses addition and multiplication:

```
0. S -> E $
1. E -> A
2. A -> M
3. A -> A + M
4. M -> n
5. M -> M * n
```

Redefining Our Definitions

Our strategy for building the LR(1) state machine remains the same as when we built the LR(0) state machine: we track Item Sets which represent an Rule Item that describes where we are as we parse individual rules in our grammar. However, we include an additional piece of information: the next token we expect to read after we finish reading a rule. This token will help us sort out shift/reduce and reduce/reduce conflicts by understanding which token is expected next in order to successfully reduce by a given rule.

That is, if in our LR(0) state machine example above, for Item Set 9:

```
IS 9
3. A -> A + M .
5. M -> M . * n
```

If we somehow knew the next token for Rule Item 3 was either + or $:

```
IS 9
3. A -> A + M .          next = +
3. A -> A + M .          next = $
5. M -> M . * n
```

We could then know that if we see a + or $ symbol we reduce in state 9, and if we see a * we shift. We can also know if we see a number n, we have a syntax error, since we'd be seeing something like "1 + 2 3", with 2 and 3 are separate tokens.

LR(1) Rule Items

The first step is to extend each rule item with an additional piece of information, the token which follows the expression, or the $ symbol. So our new Rule Item contains three bits of information: the rule being processed, the position in the rule, and the token we expect to read after we read this rule item.

When we start parsing, our new start item is the 0th rule, combined with the terminal symbol:

```
0. S -> .E$, $
```

Our LR(1) Item Sets are sets of LR(1) Rule Items.

The First Algorithm

We now can extend our closure definition to handle LR(1) Item Sets. But before we do this, we need to define a new function First.

Intuitively the First function on any set of grammar symbols is the set of first terminal symbol we can encounter when parsing those symbols.

For example, given the rule in our grammar

```
A -> A+M
```

The list of grammar symbols is "A+M". And First(A+M) would be a set containing one symbol { n }. Intuitively this means if we are parsing the rule "A -> A+M" on a sequence of tokens "1 + 2 * 3", we would expect to see the token "1" first. The sequence of tokens "+ 2 * 3" does not match our rule, and would be a syntax error.

Formally we can calculate our First function as:

```
Find First

Given a sequence of tokens and productions T = t1..tn, and our grammar G

Return a list of tokens which are first.

1.     If the list T is empty, return $ the end of file marker
2.     If the first item t1 of T is a token, return the set { t1 }
3.     Initialize a set S of tokens as the empty set. Initialize Q a queue of
       production rules and add t1 to Q. Initialize a set of visited production
       rules V and add t1 to V.
4.     If Q is empty, goto 5.
4a.    Remove production P from Q.
4b.    Find all rules R for production P. For each rule r in R
4b1.   Find the first grammar symbol t in rule r
4b2.   If t is a production rule
4b2a.  If t is not in V, add t to V and add t to Q.
4b3.   If t is a token, add t to our set S of tokens.
4c.    Goto 4.
5.     Return set S of tokens.
```

Note: as above, we do not handle the case of an empty production rule. If we were, we would have to extend our definition to look for productions that may be empty, and examine the rest of the list T.

LR(1) Item Set Closure

With the definition of First(), we can now extend our definition of Closure.

Note that we needed First() to figure out, for productions in a rule that are followed by a token, what follows our found production rule. For example, suppose we were handling the closure of the following LR(1) Item:

```
A -> .A+M, *
```

Our closure process includes finding all the rules that could expand into A, the token that immediately follows the '.' which represents our position in the rule. In our grammar's case, that is our rules:

```
A -> A+M
A -> M
```

But we also need to know what follows the symbol A in our expanded closure. From observation of our original rule set above, that's the '+' symbol.

Formally, as we run our closure process we must take a rule like A -> .A+M, *, and find First(+M*), the list of tokens which we know follow the production A--which we can easily observe is the token '+'. And of course, by definition, that includes the token associated with our rule set, which is the token we know follows our rule. We'll need this when handling an item like A -> .M, $: we know since our item ends with $, any attempt to parse M for this item should also end with $.

```
Find LR(1) Closure.

Given IS an initial set of items I
Return the closure of (IS)

1.     Add all items IS to our closure C.
2.     Create a queue Q with the list of all item IS. Create a visited set V and
       set to Q.
```

Note that we need to process each item in our item set individually, unlike our LR(0) closure algorithm.

```
3.     If Q is empty, goto 5.
4.     Remove I from Q.
5.     If the next grammar symbol in I is a production P (and the format of the
       item I is ( X -> a.Pb, c ), where a and b are (potentially zero-length)
       lists of grammar symbols)
5a.    Find the set of tokens F = FIRST(bc), with b the (potentially zero-length)
       list of grammar symbols in our item I above, and c is the token associated
       with our rule item.
5b.    For each rule R that expands production P, for each token t in F
```

```
5b1.   Create item I' = the rule item with rule R, with the position at the start,
       and the token t.
```

For example, if our production is A, and our rule is A -> A+M and our token t is '+', construct the item I' = (A->.A+M, +).

```
5b2.   If I' is not in C, add I' to C.
5b3.   If I' is not in V, add I' to Q and to V.
5c.    Goto 3.

6.     Return closure C.
```

Our closure process allows us to track not only where we are in the grammar as we parse our input file, but we also track what token follows each rule set. And we discover what follows each rule set during the construction of our closure.

When we apply our closure algorithm on our initial set:

```
S -> .E$, $
```

We get the (more complicated) item set:

```
S -> .E$, $
A -> .M, $
A -> .M, +
A -> .A+M, $
A -> .A+M, +
E -> .A, $
M -> .n, $
M -> .n, *
M -> .n, +
M -> .M*n, $
M -> .M*n, *
M -> .M*n, +
```

Look at the last set of LR(1) items. Intuitively it makes a certain degree of sense; what our grammar is telling us that when we read an addition rule A, it should be followed by a '+' sign, Multiplication rules M can be followed either by a '+' or a '*' sign. At some level we can see the idea of precedence being carried down our rule set.

The LR(1) Construction Algorithm

Our LR(1) construction algorithm basically operates like our LR(0) algorithm; we construct all the LR(1) item sets by starting with our initial item set, and constructing all the sets we can reach by transitioning through all our grammar items.

Like our LR(0) algorithm, intuitively we do the following:

```
1.     Construct the initial state from our initial rule.
2.     For every token and production in our grammar
```

```
2a.    For every state in our set of states
2a1.   Find the set of items we transition to with the token or production we're
       looking at.
2a2.   Find the closure of the new set of items.
2a3.   Note the transition from the original item set to the new item set through
       the token or production.
2a4.   Insert our new item set into our list of sets.

Repeat until done.
```

And in fact, so long as we recognize that our LR(1) Item Set contains the final token and a new item I' for item I which represents a transition through a grammar item (step 5c1 in the LR(0) state machine construction algorithm above) uses the same final token, our LR(1) algorithm is nearly identical to the LR(0) algorithm, except we use our LR(1) closure algorithm.

```
Find LR(1) state machine.

Given G the augmented grammar, with rule 0 the inserted grammar S -> R $, with R the
starting production named in the YACC file.

Return the LR(1) parsing table, consisting of a set GS of item sets, and a list of
transitions T between the item sets in GS.

1.     Construct the initial item set IS = closure(I), with I the item (S -> .R$,
       $)
2.     Construct Q = [ IS ] a queue of item sets to process
3.     Construct V = Q, the set of item sets already processed.
4.     Construct T as an empty list of transitions t(IS,symbol) -> IS'
5.     If Q is empty, go to 6
5a.    Remove IS from Q
5b.    For every item I in IS, find PS = the set of tokens and productions
       referred to by the current position in each item.
```

For example, if the item being examined is 'A -> A.+M,', the token for this item is '+'. If we have a item 'E -> A .', there is no token associated with this item, and the item is skipped.*

```
5c.    For each token or production P in PS,
5c1.   For each item I in IS, add to a new item set IS' each item I' which
       represents the transition I through P, if one exists.
```

For example, if I is 'A -> A.+M,', and P is '+', then I' is 'A -> A+.M,*'.*

Note that some items I cannot transition through P, so they are skipped.

Implementation note: in our code base, we collapse steps 5b and 5c into a single step by constructing a mapping M which maps a token or production rule P to the item set IS' being constructed. This allows us, in one shot, to figure out the P we're transitioning through for each I, and accumulate I' into a new IS' as we construct them.

```
5c2.   Find C = the closure of IS'.
5c3.   If C is not in I, add C to V and to Q.
5c4.   Add transition t(IS,P)->IS' to our list of transitions T.
5d.    Goto 5.
```

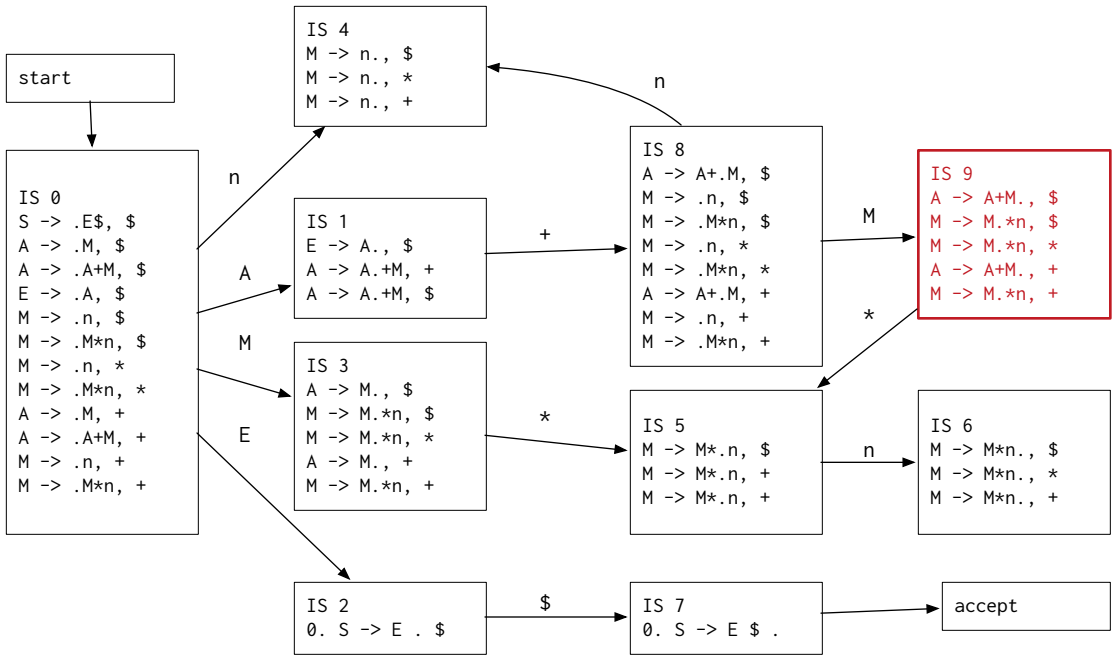
```
6.    Return LR(1) as I, the set of item sets found, and T, the list of
      transitions between item sets.
```

Our algorithm is exactly the same as the LR(0) algorithm, except for the construction of new items as we shift through a grammar symbol (in step 5c1), and the closure algorithm.

> *Implementation Note: The code that implements Knuth's LR(1) algorithm is contained in the class OCCYaccLR1, which is not actually used in production. It is used to validate generated LR(1) grammars for testing purposes during development.*

Our Example Grammar and the LR(1) State Machine.

For our limited example, the state machine has not grown in size, and looks very similar to the LR(0) state machine above.



Note our highlighted state 9, which was a problem in our LR(0) state machine. When previously we saw only one reduce rule, now we see two--but both reduce by the same rule A->A+M. Previously our ambiguity (do we reduce or shift?) is now resolved. We reduce if we see a '$' token or a '+' token. We shift if we see a '*' token. Anything else is an error.

> *Note: This does not eliminate shift/reduce or reduce/reduce errors. However, for a grammar parsed using this algorithm to have a shift/reduce or reduce/reduce error means it is not an LR(1) compatible grammar. There is a theoretical result that any LR(k) compatible grammar can be rewritten as an LR(1) compatible grammar, which implies that if we were to see a shift/reduce or a reduce/reduce error, an end-user of our system could fix the bug by hand with some effort.*

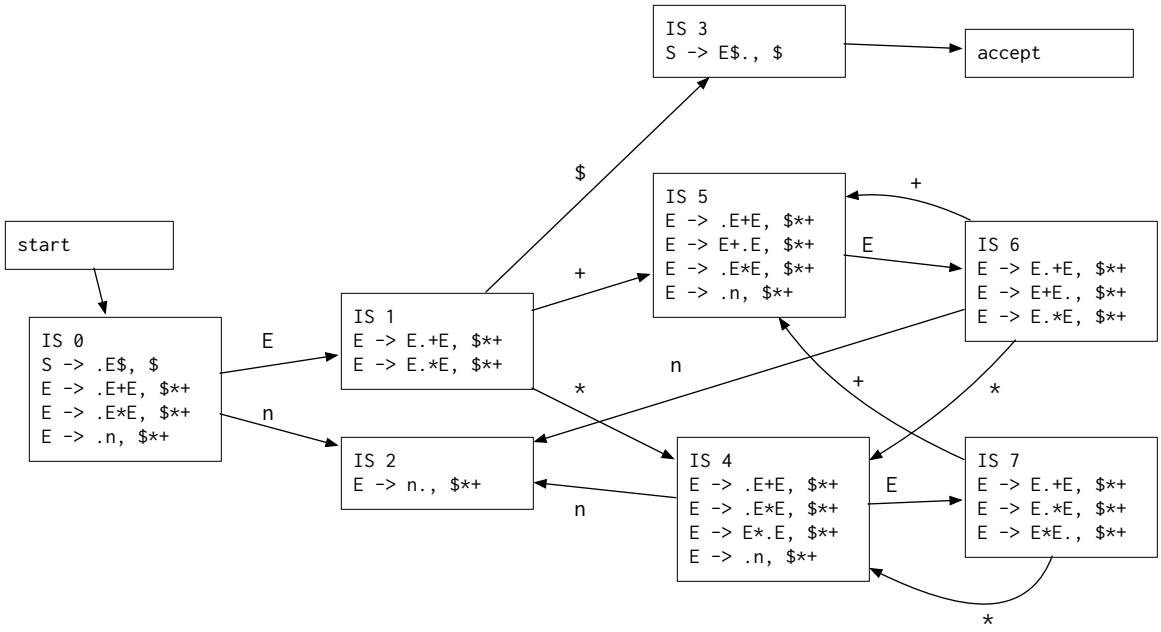
This is different from state machine construction techniques weaker than LR(1) but stronger than LR(0), such as LALR or SLR, which may not be able to express all grammars expressible with LR(1).

Resolving shift/reduce errors using precedence in an ambiguous LR(1) state machine.

Suppose we have the following augmented grammar, but with the implicit understanding that '*' has a higher precedence than '+', specified (for example) by using a %left or %right declaration.

```
S -> E $
E -> E + E
E -> E * E
E -> n
```

The LR(1) state machine for this grammar constructed using our algorithm above (collapsing rule items that share the same rule and position but have different final tokens):



Note: In the above diagram, we use the expression "E -> E.+E, $+" to represent three separate rule items: "E -> E.+E, $", "E -> E.+E, *" and "E -> E.+E, +". The expression is simply for the sake of compact notation.*

Notice the shift/reduce conflict in states IS 6 and 7. We can resolve these by considering the precedence of the + token relative to the * token: if the precedence of the token in the reduction item is higher than the shift operation, we reduce instead of shift. This would imply if we see the expression "1 * 2 + 3", on seeing the "+" symbol in state 7, electing to reduce means we reduce the "1 * 2" first--giving us the grouping "(1 * 2) + 3".

If the precedence is equal, we rely on if the token is left associative, right associative or non-associative. For state IS 7, if the token is left-associative, we reduce; this means the leftmost expression is handled before we

process the rest. If the token is right-associative, we shift. And if the symbol is non-associative, we consider the state with the associated token an error.

Note that precedence only makes sense if we have a single token associated with our grammar rule; that is, so long as we have a grammar rule that contains a single operator that has precedence associated with it. Thus, during the preprocessing step we may wish to associate precedence with each of our grammar rules.

Resolving reduce/reduce errors using ad-hoc rules.

Reduce/reduce errors can be handled in an ad-hoc way.

Bison[16] handles reduce/reduce conflicts by choosing the rule that appears first in the grammar. Because (as observed in the GNU documentation) this sort of resolution is fairly risky, it is worth noting to the user that a reduce/reduce error exists, and which rules are in conflict for a given token, so the user can attempt to resolve the error in another way.

Both shift/reduce and reduce/reduce errors can be resolved as we generate the action table for our LR(1) parser.

Generating the Action/Goto Tables for our LR(1) parser.

We can also generate our action/goto tables using the same steps we used for our LR(0) state machine, but with some minor changes which deal with the final tokens associated with each rule item to generate our reduce states.

The discussion below operates on the original augmented grammar introduced on page 5.

Note that much of the discussion below is similar to our discussion with building our LR(0) grammar.

1. Build an empty action table A = a(IS,P) -> action, where IS is an item set (or the index number of an item set), P is a token in our grammar, and action is one of "shift state", "reduce rule", "error" or "accept." By default fill this 2-dimensional table with error.
2. Build an empty goto table G = g(IS,P) -> state, where IS is an item set (or index), P is a production rule in our grammar, and state is the new item set (or index) we transition to. By default fill this 2-dimensional table with empty states.
3. Populate G with g(IS,P) with the transitions in our state machine that pass through a production rule P.

In our original grammar example, the state transitions that pass through the production rules E, A and M are transitions like the one from IS 0 to IS 1 above. This results in the table G equal to:

	E	A	M
0	2	1	3

[16] https://www.gnu.org/software/bison/manual/html_node/Reduce_002fReduce.html

```
1
2
3
4
5
6
7
8                       9
9
```

4. Populate A with a(IS,P) with the transitions in our state machine that pass through terminals in our grammar.

We do this in the same way as we do 3 above. This results in the action table A equal to:

```
          $      *      +      n
0
1                       S8     S4
2         S7
3                S5
4
5                              S6
6
7
8                              S4
9                S5
```

Note that in our action table, we use the notation "A" as "accept", "Sn" to mean "shift to item set n", and "Rn" to mean "reduce by rule n".

Note also that, as both our LR(0) and LR(1) state machines are identical, the resulting tables are also identical to our LR(0) tables generated above.

```
5.    Now add the reduction rules. For every item set IS in our LR(1) state
      machine
5a.   For every item I in IS
5a1.  If the item I is at the end position
5a1a. Get the final token t associated with I.
5a1b. If the table a(IS,t) contains a shift operation, if the precedence of the
      rule that triggered the shift is lower than the rule to reduce, then
      replace the item at a(IS,t) with the reduction.
5a1c. If the table a(IS,t) contains a reduce operation, if the precedence of the
      new rule is higher than the existing reduce rule, replace the item.
5a1d. If the reduce rule contains the same precedence, then if the associativity
      of the rule is %left, we reduce; if %right we shift, and if %nonassoc, we
      mark the state as an error.
5a1e. If a(IS,t) is empty, we note the reduction.
```

Reduction means we do two things. First, we replace on our stack the top set of symbols we've parsed with the collapsed production represented by that rule. Then, we transition to a new state depending on the symbol we found at the top of the stack. That transition is given in our goto table G, already constructed above.

Intuitively, this two-step process can be seen as we've collapsed our rule, and now the next symbol we're handling is the production we just collapsed into.

For our example grammar, our reduce rules look like:

	$	*	+	n
0				S4
1	R1		S8	
2	S7			
3	R2	S5	R2	
4	R4	R4	R4	
5				S6
6	R5	R5	R5	
7	A	A	A	A
8				S4
9	R3	S5	R3	

Unlike our LR(0) state machine, we do examine the last token of each item in the item set representing our state when determining the reduction rule. Thus, for our grammar above we have no conflicts.

6. Return our final tables A and G.

So Why Don't We Use This Algorithm?

Well, it boils down to the overall size of the resulting state machine.

For example, in Pager 2012[17], the Knuth LR(1) algorithm is shown to produce a much larger grammar than produced using the LALR grammar techniques used by Bison. For a grammar of the C language, the Knuth LR(1) table has 4.5 times more states, while for an Ada grammar, the Knuth LR(1) table has 14.8 times more states. While in today's environments, generating a significantly larger table does not render the algorithms as completely useless, it is worth some effort to attempt to generate the LR(1) tables in a more efficient and compact fashion.

There are two general strategies that are employed to create more compact LR(1) tables.

The first is by generating the full LR(1) state machine generated by Knuth's algorithm, and then take a post-processing step to reduce the size of the table set. The second is to generate the LR(0) state machine, then use various techniques to eliminate conflicts so that all LR(1) grammars can be represented.

Do we really need to avoid LR(1) state machines?

OCYacc generates a full LR(1) state machine using Knuth's algorithm, though this paper continues with a discussion of LALR state machines and GLR parsing techniques to avoid the occasional indecipherable reduce/reduce conflicts that may arise.

---

[17] Pager, David; Chen, Xin: "The Lane table Method of Constructing LR(1) Parsers", Table 3: "Parsing table size comparison"

The reason for this is that, in todays' computing environment, it may not necessarily be desirable to reduce the number of states in order to save space in the parsing tables.

Error handling, for example, is far easier with an LR(1) state machine. If an erroneous state and token is received, LR(1) state machines for an LR(1) grammar halt immediately. Further, the erroneous state can provide considerable context as to where the error happened and, in some limited cases, automatically provide good diagnostics and even some limited recovery.

Consider, for example, the types of computers available when Knuth's original algorithm was introduced in 1965.[18]

In 1965, the Digital Equipment Corporation produced the DEC PDP-8, the first commercially successful minicomputer, costing an inflation-adjusted $140,000. The original PDP-8 was a 12-bit processor which could address up to 4k-words of memory. (Around 6k bytes.) A later version had an optional expansion system which could increase the amount of memory to 32k.

Also available at that time was the IBM 709x series of computers; in 1964 an IBM 7094 II was installed at places such as Carnegie-Mellon University. Similar IBM 7094 systems ran up to $25 million (in inflation-adjusted dollars), and came standard with 32k-words of 36-bit per word memory. (That's around 144k-bytes of memory.) The IBM 7094 system ran instructions at 500 thousand instructions per second.

Needless to say, using Knuth's LR(1) techniques to parse a language such as C, which can generate around 2,000 states, was simply impractical, and it motivated the drive to finding more compact state machine representations.

Today, an Arduino embedded micro-controller can run circles around the original PDP-8, running 16 million instructions per second, running with just as much storage memory. A cell phone such as the iPhone 5 contains 7,200 times more memory than the IBM 7094 II, and runs instructions a minimum of 5,000 times faster. As of the writing of this paper, Apple is phasing out support for the iPhone 5 because it is "under-powered."

Even when YACC was first introduced in 1975 on Version 3 Unix, the fastest computer in the world (introduced the following year) was the Cray-1 supercomputer. Costing around $40 million (inflation adjusted), the Cray-1 supercomputer weighed around 5.5 tons and contained 4 megabytes of memory, and ran at 80 MHz, 12.5 times slower than an iPhone 5. (The iPhone 5 has greater parallelism, which means the iPhone 5 can execute more instructions in parallel on every clock cycle.) YACC, however, was typically not used on computers that fast.

Generally YACC would be run on a computer with similar power to the Digital VAX computer, introduced in 1977. The Vax-11/780 was introduced with 2 megabytes of memory, and ran approximately 1 million instructions per second--making it about a thousand times slower than an iPhone 5.

---

[18] Knuth, Donald E., "On the Translation of Languages from Left to Right", Information and Control, 1965

In light of this, do we need to avoid an algorithm which generates state machines between 4.5 and 12 times larger, when our computers generally have thousands of times more memory and run thousands of times faster?

It is possible that there are other reasons for wanting to keep the number of states relatively small. If the number of states are below certain thresholds you can conceivably use smaller integer values to store the states. (If you have fewer than 128 states and 128 rules, for example, you can store the action and goto tables using 8-bit bytes, with the MSB indicating if an action is a shift or reduce.) It is also possible that you may wish your parser to target a much smaller footprint device. The rise of embedded processors that rival the power of mainframe computers from the 1980's comes the opportunity to do some very interesting things.

So we will continue with the description of SLR and LALR parser techniques.

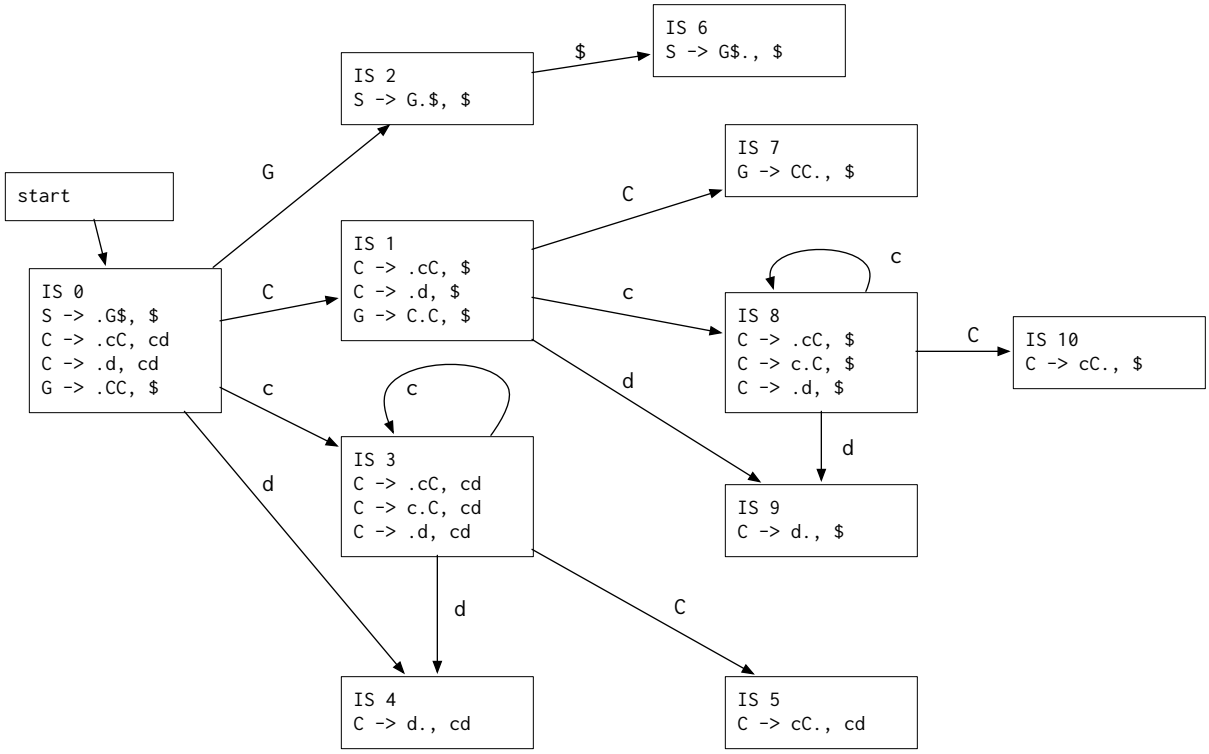
Collapsing LR(1) States.

Consider the following grammar, borrowed from The Dragon Book:[19]

```
S -> G $
G -> C C
C -> c C
C -> d
```

The constructed state machine using the algorithm above is:

[19] Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D.; "Compilers: Principles, Techniques and Tools", Example 4.42



Notice that several of the states appear similar. For example, look at states 4 and 9; they differ only by the token that we need to see to perform a reduction.

It would make logical sense, then, that we may want to detect states like this and collapse them into a single state. In the case of states 4 and 9, we could collapse them into a new state, state 4', which performs a reduction regardless of the next token. The Dragon Book states[20] that in fact, even with an erroneous reduction that would be triggered if, for example, we are in state 9 but the next token is a 'c', that the error will be caught before the next shift operation takes place. And conceptually this makes sense: state 9 can only happen when the next token is an end token $. After performing the reduction, at some point we will transition to a state which expects an end token (by construction), at which point we will trigger an error.

This motivates a technique for generating an LALR state machine and parsing table from an LR(1) grammar, outlined in the Dragon Book.[21]

[20] Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D.; "Compilers: Principles, Techniques and Tools", Section 4.7: Constructing LALR Parsing Tables.

[21] Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D.; "Compilers: Principles, Techniques and Tools", Algorithm 4.11

Fixing up deficient LR(0) grammars.

Another technique which can be used to construct our LR(1) grammar is to examine those states in an LR(0) grammar where a shift/reduce or reduce/reduce error exists, and applying techniques to either split the states or to mark the rule items within the item set defining a state so as to correctly determine if a state should be reduced or shifted.

Various papers have described various techniques to do exactly this. Pager 1972[22] and Pager 2012[23] describe one technique; Spector 1981[24] and Spector 1988[25] all describe various methods for resolving conflicts that result from an LR(0) grammar. Further, the original Dragon Book algorithms for finding an SLR[26] and LALR[27] state machines.

A far more complete survey of techniques for fixing up deficient LR(0) grammars can be found in Grune 2008.[28]

Extending to LR(k) Grammars For k > 1

It seems to me that we should be able to extend this algorithm to an LR(k) parser where k is greater than 1 by making two observations.

First, consider the definition of a Rule Item. If we define a LR(k) Rule Item as being the rule, position and a string of tokens of length k, then each LR(k) Item Set would represent the list of rules and potential k-token lists that follow our Item Set for dealing with reduce/reduce conflicts.

We could start our state machine computation algorithm by finding the LR(k) Closure of the start item [S->.E$, $$...], where $$... is a k-length list of terminal symbols. (We could also introduce the symbol % to represent 'undefined', and use the follow string $%%%..., with k-1 undefined tokens '%', since in theory we will never actually read the % token in a shift production.)

---

[22] Prager, David: "The Lane Tracing Algorithm for Constructing LR(k) Parsers"

[23] Pager, David; Chen, Xin: "The Lane Table Method Of Constructing LR(1) Parsers"

[24] Spector, David: "Full LR(1) parser generation", ACM SIGPLAN Notices, 1981

[25] Spector, David: "Efficient full LR(1) parser generation", ACM SIGPLAN Notices, 1988.

[26] Aho, Alfred V., Sethi, Ravi, Ullman, Jeffrey D., "Compilers, Principles, Techniques and Tools", Algorithm 4.8: Constructing an SLR parsing table.

[27] Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D.; "Compilers: Principles, Techniques and Tools", Algorithm 4.12: Determining lookaheads, and Algorithm 4.13: Efficient computation of the kernels of the LALR(1) collection of sets of items.

[28] Grune, Dick; Jacobs, Ceriel J.H., "Parsing Techniques, A Practical Guide", Second Edition, Springer Publishing, 2008

We now need to define a LR(k) First algorithm. Intuitively this would be the list of tokens that could start a sequence of grammar symbols, and this function could be defined recursively. (Note we assume we have no production rules A -> *empty*.

1. $First_0$ of any list of grammar symbols is the empty string. (I include this rule for completeness. Note if we use this definition for First, then our LR(k) grammar will be equal to our LR(0) grammar.)

2. If the list of grammar symbols $G = A_1A_2...A_k$ starts with a token $A_1 = t$, then $First_k(G)$ = the set of tokens $First_{k-1}(G')$, where $G' = A_2...A_k$, with the t token prepended to each found item.

3. If the list of grammar symbols G starts with a production P, then accumulate all the rules that P would expand to. (That includes rules of the form $P \rightarrow B_aB_b...B_n$ as well as any rules that expand to any production if B_a, if B_a is a production.) If, in walking through the list of rules we uncover a rule which starts with token $t = B_a$, then add to our return set the set of strings returned by $Firstk(B_2...B_nA_2...A_k)$ with the token t prepended to each string.

The idea, in other words, is to define our LR(k) First algorithm recursively on k.

With our definition of LR(k) First algorithm, we can define our LR(k) Closure algorithm method as exactly the same as our LR(1) Closure, substituting our new First algorithm. Keep in mind that because our LR(k) Item always has a follow value of a string of k tokens, we will never have a case where our LR(k) First algorithm is called with fewer than k tokens.

Our overall LR(k) state machine algorithm then operates in exactly the same way as our LR(1) state machine algorithm.

Of course the resulting action table would need to be split into two separate tables; a shift table and a reduce table, as described in our description of the LR(1) Parser algorithm discussion above.

Because of the theoretical result given in Knuth 1965,[29] we know LR(k) algorithms can be rewritten as LR(1) algorithms, so the observation how we can construct an LR(k) algorithm is left as an interesting side note.

Constructing LR(1) State Machines in Parallel

We can also take advantage of some potential parallelism in the above algorithm by observing the process of computing our LR(1) First and LR(1) Closure algorithms do not alter the underlying grammar as they operate. In fact, the only point where the results of a closure is used in a way which potentially requires thread-safe access to a common state is at steps 5c3 and 5c4 of our LR(1) Construction Algorithm.

We can potentially launch a separate thread or process for each call into our (computationally expensive) LR(1) Closure algorithm.

---

[29] Knuth, Donald E., "On the Translation of Languages from Left to Right"

We can also apply this observation to our LR(k) process, making constructing LR(k) grammars for small values of k practical on modern desktop computers.

The way to do this is by considering our standard iteration algorithm, outlined above:

```
1.     Create a queue Q of items to process. Add an initial state S0 to Q.
2.     Create a set V of items that have been visited, and add S0 to V.
3.     If Q is not empty
3a.    Remove Si from Q.
3b.    Do some process P on Si which returns 0 or more items Sj...Sk.
3c.    For each Sn in Sj...Sk, if Sn is not in V, add Sn in V and Q. Perform side
       effect P' on Sj..Sk as appropriate
3d.    Goto 3
4.     Return our result.
```

This is the form our state machine construction algorithm takes.

If we assume the process P is an operation that can be run in parallel (as our Closure algorithm can as it has no side effects on the grammar or state machine being constructed), then we can alter our basic algorithm as:

```
1.     Create a queue Q of items to process. Add an initial state S0 to Q.
2.     Create a set V of items that have been visited, and add S0 to V.
3.     Create integer count c = 0. This is used to track how many items are in
       process in our thread.
4.     Create a mutex and signal.
5.     Wait on mutex and signal
5a.    If c == 0 and Q is empty, goto X
5b.    If Q is not empty, remove Si from Q
6.     Increment c.
7.     Create thread to operate on Si:
7a.    Do some process P on Si which returns 0 or more items Sj...Sk. This process
       may perform side effects for our return value.
7b.    Lock the mutex
7c.    For each Sn in Sj...Sk, if Sn is not in V, add Sn in V and Q. Perform side
       effect P' on Sj..Sk as appropriate
7d.    Decrement c.
7e.    Signal our signal. (This causes 5a/5b to look for more items in Q, or quit
       if no more threads are running.)

8.     Return our result.
```

Computing SLR State Machines.

Before we dive into the Dragon Book's presentation of the LALR algorithms, it is worth a side venture into constructing SLR parsing tables as a very simple example of fixing up a broken LR(0) state machine.

SLR parsers are not as powerful as LALR parsers in that there are many expressions SLR parsers cannot handle that LALR parsers can. And LALR parsers are less powerful than full LR(1) parsers. However, SLR parsers can handle some constructed grammars that LR(0) parsers cannot.

Some Definitions

In order to discuss our SLR parsing algorithm, we must define a new algorithm, Follow. Follow is similar in idea to First, defined in our discussion about the Knuth LR(1) Closure algorithm above, except of finding all possible tokens that can precede a string of grammar symbols, Follow finds the list of all tokens that can possibly follow a production, by looking at all of the rules that contain that production.

The Follow Algorithm

The Follow algorithm will produce a mapping which maps productions P in our augmented grammar to a list of tokens which can possibly follow our production rule P. This mapping can then be used to break some reduce/reduce conflicts.

Note that our Follow algorithm uses our First algorithm above.

```
Follow

Given an augmented grammar G with start rule S -> S'$.
Return a mapping F which maps productions P -> token set T of tokens t1..tn.

1.     Create empty map F
2.     Add S -> { $ } to map F, where S is the start rule and $ is the end of file
       token.
```

Now find all follow tokens for productions P that appear in a rule R -> aPb, where a is a (potentially empty) list of grammar symbols, and b is a non-empty list of grammar symbols.

Note that, as above, we assume we have no empty rules; that is, we have no rules with an empty list of grammar symbols.

```
3.     For every production P in our grammar G
3a.    For every rule R in our grammar G that contains production P in the grammar
       symbols of the rule,
3a1.   For every position i in our list of grammar symbols in R that points to P,
       where i is not at the end of the list of grammar symbols
3a2.   Find the list L of grammar symbols in rule R from i+1 to the end of the
       grammar.
3a3.   Add P -> First(L) to our map F.
```

Now we find all follow tokens for productions P that appear at the end of rule R -> aP, where a is a (potentially empty) list of grammar symbols.

Note that our algorithm restarts whenever we add new tokens to an existing map F; this is to handle the case where a production A is at the end of rule B, and production B is at the end of rule C. This is guaranteed to terminate by virtue of the fact that the number of tokens in our grammar G is finite. (Eventually we run out of stuff to add.)

```
4.    For every production P in our grammar G
4a.   For every rule for production R in our grammar G where P appears at the end
      of the list of grammar symbols of our rule, and P != R
```

We skip rules of the form P -> aP, because it's redundant.

At this point everything in F(R) should be in F(P).

```
4a1.  If F(R) contains tokens not in F(P)
4a1a. Add tokens F(R) to F(P).
```

Now we restart our loop from the top. This is why we have the test in 4a1; otherwise, we'd loop forever.

```
4a1b. Goto 4.

5.    Return follow map F.
```

When we apply this algorithm to our original augmented grammar:

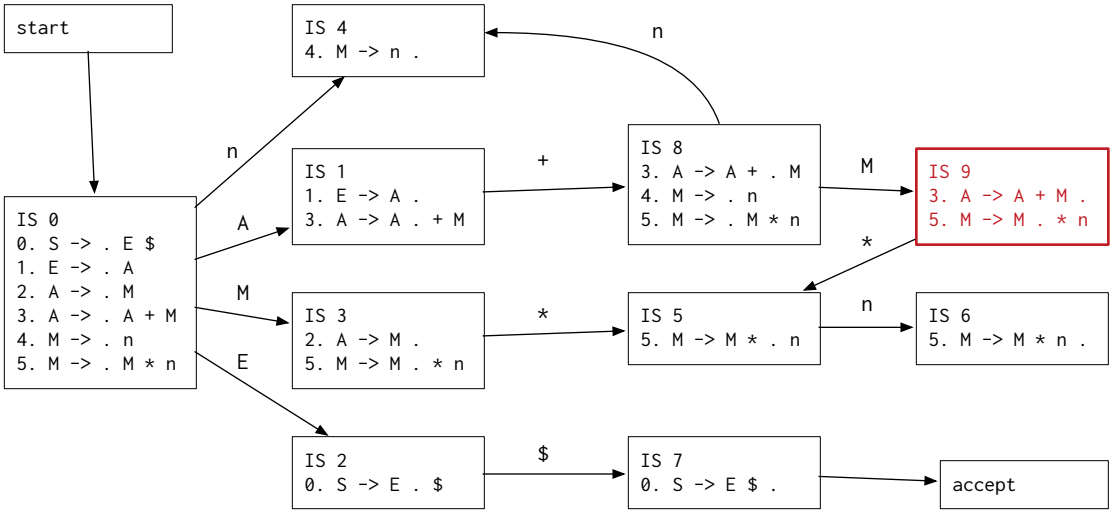
```
0. S -> E $
1. E -> A
2. A -> M
3. A -> A + M
4. M -> n
5. M -> M * n
```

We get the follow list:

```
S: $
A: $ +
E: $
M: $ + *
```

Intuitively this makes a certain degree of sense: as we parse addition statements we expect to see an addition statement followed by another addition statement. Multiplication statements should be reduced prior to the next addition statement. Meaning if we see the expression "1 + 2 * 3 + 4", since addition statements "A + M" are followed by other addition statements, the middle "2 * 3" should be reduced first--and from our "A -> A + M" rule, we see "1 + M + 4", where "M" is "2 * 3". This finally gives us our proper grouping "1 + (2 * 3) + 4".

This helps us when fixing up shift/reduce conflicts in our original LR(0) state machine. Take a look at state IS 9, of our LR(0) Item Set:



The conflict at state 9 can be resolved by looking at Follow(A), the production of item in item set 9 that is to be reduced. As Follow(A) = { $, + }, we can modify how we populate the action table to only populate those columns in state 9 which are in Follow(A). The same can be done in the other conflicting states, such as state 3 and state 1.

Our final action table (constructed in this way rather than using the method outlined above in our LR(0) construction algorithm) yields:

```
          $      *      +      n
0                               S4
1         R1            S8            (Follow(E) = { $ })
2         S7
3         R2     S5     R2            (Follow(A) = { $, + })
4         R4     R4     R4            (Follow(M) = { $, +, * })
5                               S6
6         R5     R5     R5            (Follow(M) = { $, +, * })
7         A      A      A      A
8                               S4
9         R3     S5     R3            (Follow(A) = { $, + })
```

The SLR Algorithm.

We can now give our final SLR Algorithm.

```
SLR Algorithm

Given an augmented grammar
Return the action table and goto table for our SLR state machine

1.    Build our LR(0) State Machine for our grammar as above.

2.    Build our Follow map F for our grammar as above.

3.    Build our action table A and goto table G:
```

```
3a.    Build an empty action table A = a(IS,P) -> action, where IS is an item set
       (or the index number of an item set), P is a token in our grammar, and
       action is one of "shift state", "reduce rule", "error" or "accept." By
       default fill this 2-dimensional table with error.
3b.    Build an empty goto table G = g(IS,P) -> state, where IS is an item set (or
       index), P is a production rule in our grammar, and state is the new item
       set (or index) we transition to. By default fill this 2-dimensional table
       with empty states.
3c.    Populate G with g(IS,P) with the transitions in our state machine that pass
       through a production rule P.
3d.    Populate A with a(IS,P) with the transitions in our state machine that pass
       through terminals in our grammar.
3e.    Now add the reduction rules. For our LR(0) state machine with our Follow
       set, if an item set IS contains an item where our parser position is at the
       end of the rule:
3e1.   Find the follow set of tokens T = F(P) for the production P of our rule in
       our item set IS.
3e2.   Add a reduction rule in our action table to reduce by our rule only for
       those tokens.

4.     Return our final tables A and G.
```

The steps for constructing our SLR action table A is exactly the same as for the action table in our LR(0) algorithm above, except we only populate the token columns that are in our follow set.

Shortcomings of the SLR State Machine.

The problem with using the Follow algorithm to determine the context of an LR(0) state for resolving shift/reduce and reduce/reduce conflicts is that the results of our Follow algorithm is too generic: we assume every production is followed by a set of tokens regardless of how or where our rule is used.

While our SLR algorithm does resolve the original problem in our example grammar, there are other grammars which can confuse our algorithm.

For example, the following grammar, from the Dragon Book:[30]

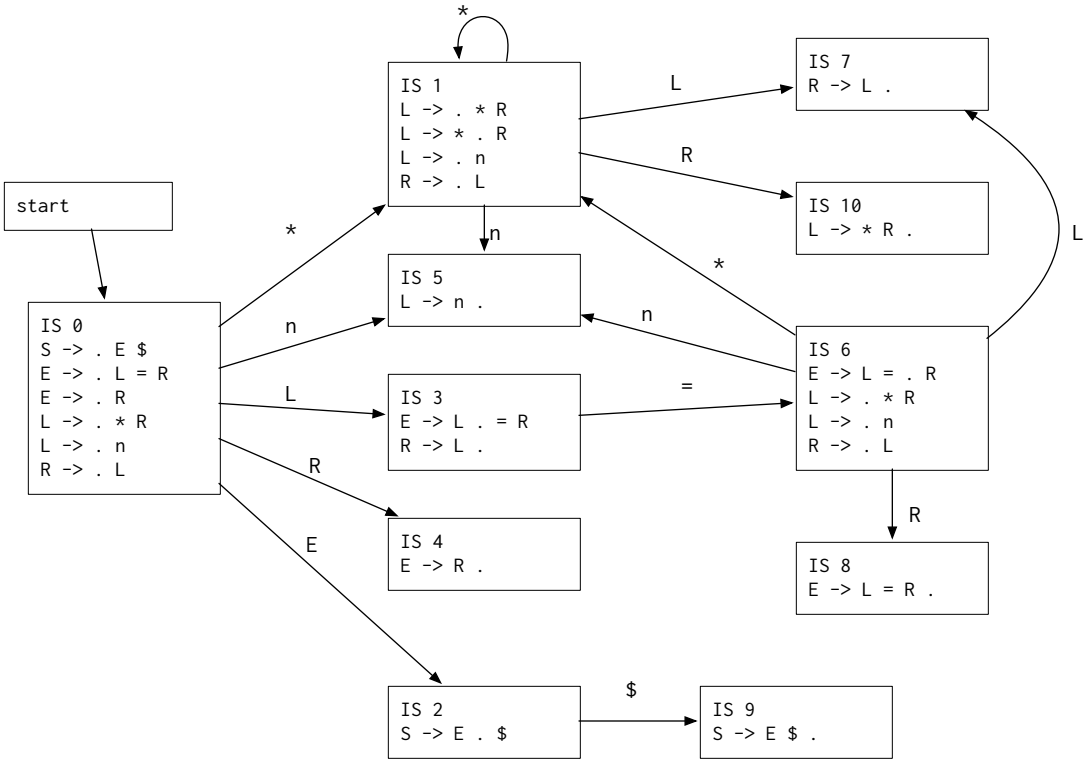
```
S -> E $
E -> L = R
L -> * R
L -> n
R -> L
```

Our SLR algorithm will produce the follow sets:

```
S: $
E: $
L: $ =
R: $ =
```

[30] Aho, Alfred V., Sethi, Ravi, Ullman, Jeffrey D., "Compilers, Principles, Techniques and Tools", Example 4.39.

And the state machine:



The problem with our SLR state machine can be seen in state 3. That state has a shift/reduce ambiguity, since for the reduction "R -> L .", our follow algorithm indicates we can reduce on either '$' or '='. But state 3 transitions to state 6 on a shift '=' symbol.

This motivates the desire to create a better algorithm which has more information about what follows each item in the various item sets, but does not require us to create as many states as LR(1) above.

Computing LALR State Machines.

The LALR algorithm given below is similar to the SLR machine in that it starts by finding the LR(0) state machine, then makes a second pass resolving shift/reduce conflicts and reduce/reduce conflicts without creating any additional states.

LALR state machine are a little more powerful than an SLR state machine in that, by doing some extra work to find the token which follows each item within an item set that is to be reduced, we can resolve the same shift/reduce and some reduce/reduce conflicts resolved by an LR(1) state machine, but keeping the number of states the same as an LR(0) state machine.

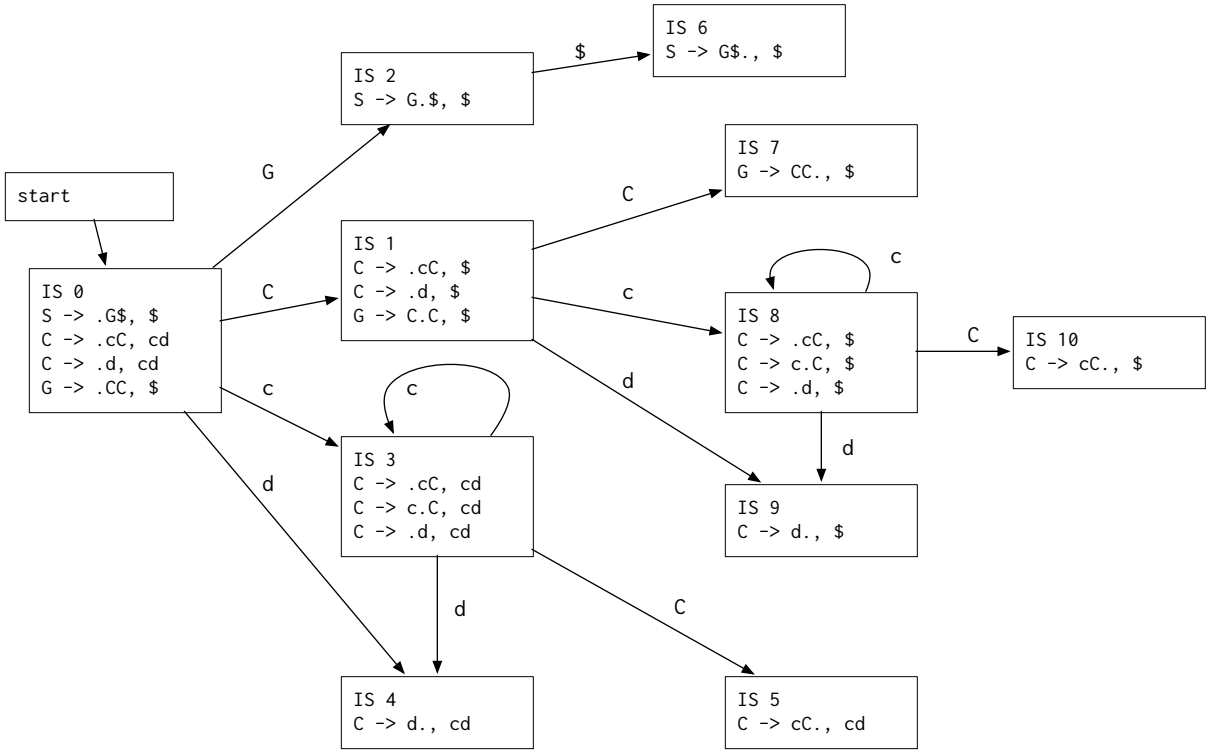
The additional power comes from the fact that our follow lists are not calculated globally for each production P in our grammar (as done by SLR), but is calculated locally for each state in our LR(0) state machine.

The LALR state machine comes from the observation that certain states in an LR(1) state machine can be merged together if they represent the same LR(0) item set.

Let us revisit the following grammar:

```
S -> G $
G -> C C
C -> c C
C -> d
```

The constructed state machine using the algorithm above is:



Examining states IS 8 and IS 3:

```
IS 3
C -> .cC, cd
C -> c.C, cd
C -> .d, cd

IS 8
C -> .cC, $
C -> c.C, $
C -> .d, $
```

Each state represents essentially the same parsing state; the same LR(0) Item Set, but with different follow tokens.

In fact, if we were to merge all states which are similar in this way, we would be left with a state machine which looks like the LR(0) state machine.

We can make this assertion by recalling how we constructed the LR(0) state machine. Each transition in the LR(0) state machine represents what happens to the rule and position in the rule when we advance to the next position. The token which follows the rule/position pair that comprises the LR(0) item does not matter at all to the next state. (As an example, examine the transition from IS 3 through production C to IS 5, and from IS 8 through production C to IS 10.)

Additional Definitions

Our strategy for constructing our LALR state machine is the same as when we built our LR(0) or LR(1) state machines: we track Item Sets which represent a Rule Item that describes where we are as we parse individual rules in our grammar. Like the LR(1) algorithm we track with each rule item the next token expected after the Rule Item. However, unlike the LR(1) algorithm, we track these next tokens differently.

LALR Rule Items

Our LALR rule items are essentially the same as LR(0) rule items, but with some minor changes.

First, to each rule item, we attach a set of tokens which follow. This is similar to the LR(1) rule items which were uniquely identified by the rule, position and token which follows this item.

Second, two rule items are considered the same if the rule and positions are the same; we do not compare the token sets associated with each rule item. (The list of tokens is pendant to the LR(0) rule item; additional information we track in our parsing algorithm.)

> *Note: Our implementation of an LALR Rule Item is contained in OCYaccLALR.h, and consists of two integers, one representing the rule index, the second the position in the list of grammar symbols. Each Rule Item also has an associated set of follow tokens. Two LALR Rule Items are considered the same if the rule index and position are the same, regardless of the follow token set.*

When we start parsing, like with LR(1) we start with the 0th rule combined with the terminal symbol:

```
0. S -> .E$, {$}
```

LALR Closure Algorithm

Our LALR Closure algorithm is exactly the same as our LR(1) Closure algorithm, modified only to adjust for the altered LALR rule item format by unrolling our LALR-style item sets into an LR(1) style item set, performing the closure procedure, then rolling the item set back up. Note that this algorithm uses the LR(1) First algorithm unmodified.

> *Note: I suspect there is a better way to handle the closure process without resorting to Dragon Book style Kernels (discussed below); when I think of it, I'll update this section.*

Propagating Follow Sets

With our new definition of an LALR rule item, and our new definition of LALR Closure, we can construct our LALR state machine as before.

However, after we've construct the state machine using the same technique we've used for LR(0) and LR(1) state machines, we will need to do a post-processing step.

The reason for this is that as we construct our LALR item sets by following transitions from one item set to another, we may not capture all of the follow tokens in each item in all of our item sets.

To see why, consider the construction of the LALR state machine from the grammar above:

```
S -> G $
G -> C C
C -> c C
C -> d
```

During the construction process we will calculate the closure of Item Set 3 as:

```
IS 3
C -> .cC, {cd}
C -> c.C, {cd}
C -> .d, {cd}
```

Eventually we will attempt to calculate the transition through symbol c from state IS 1 (see the graph above), and we will calculate a new item set

```
C -> .cC, {$}
C -> c.C, {$}
C -> .d, {$}
```

We're now faced with a quandary. Thanks to our definition of a LALR Item Set, this will match the IS 3 item set, and our state machine construction algorithm will halt without updating the follow sets. We could modify IS 3, but we would then somehow need to modify all the transitions out of IS 3 as needed if IS 3 was already processed and added to our (growing) state machine.

Instead, we address this problem with a post-processing step, where we revisit our state machine, finding cases where follow tokens need to be propagated downwards, fixing up item sets like IS 3:

```
IS 3
C -> .cC, {cd$}
C -> c.C, {cd$}
C -> .d, {cd$}
```

We do this by defining a new algorithm that propagates the follow set of each item in an item set through a transition to an already constructed item set. Intuitively this is identical to the method we used to construct a new LR(1) item set while constructing the LR(1) state machine, adopted for our modified LALR item sets:

```
Propagate Follow

Given item set IS passing through grammar symbol g to a new item set IS'
Updates the follow list for item set IS'

1.    For each item I in item set IS, add to a new item set IS" each item I'
      which represents the transition I through g, if one exists.
```

For example, if I is 'A -> A.+M,{}' and g is '+', then I' is 'A -> A+.M,{*}'.*

```
2.    Calculate C = the LALR closure of IS".
```

Because our transition and closure process generates the same LALR rule items regardless of the follow tokens, as our LALR rule items are basically LR(0) rule items but with some additional information, the rule sets in IS" should match exactly the rule set in IS', except for the follow tokens.

```
3.    Merge all follow sets for each rule I" in IS" that matches a rule I' in
      IS'.
```

Using this algorithm, we can then do our post-processing step by propagating the follow lists for all our states in our LALR state machine, by basically repeating the step above on all our states and transitions until there are no further changes in our follow lists.

```
Propagate Follow Sets

Given an LALR state machine consisting of a set I of item sets, and T, a list of
transitions between item sets.
Updates the follow lists of each LALR item sets for our grammar.

1.    Construct S = the set of all item sets I. This is used to determine which
      item sets we must examine to determine if we need to propagate a specific
      follow set for a specific item set.
```

Implementation note: we handle the set by setting a flag in each item set; we add an item back to the item set by setting the flag.

```
2.    If S is not empty
2a.   Pop item set IS from S.
2b.   For every grammar symbol g that transitions IS through g to IS', where IS
      is not IS'
2b1.  Perform Propagate Follow on IS, g through to IS'.
2b2.  If IS' is updated by our propagate follow operation,
2b2a. Add IS' to our set S

3.    Finish.
```

This is guaranteed to halt if only because the number of grammar symbols are finite.

Our LALR Construction Algorithm

The LALR construction algorithm is exactly like our LR(1) construction algorithm, except we use the LALR rule items and the LALR closure algorithm. In addition, we then finish with the propagate algorithm outlined above.

```
Find LALR state machine.

Given G the augmented grammar, with rule 0 the inserted grammar (S -> R$, {$}), with R the
starting production named in the YACC file.

Return the LALR parsing table, consisting of a set GS of item sets, and a list of
transitions T between the item sets in GS.
```

```
1.    Construct the initial item set IS = closure(I), with I the item (S -> .R$,
      $)
2.    Construct Q = [ IS ] a queue of item sets to process
3.    Construct V = Q, the set of item sets already processed.
4.    Construct T as an empty list of transitions t(IS,symbol) -> IS'
5.    If Q is empty, go to 6
5a.   Remove IS from Q
5b.   For every item I in IS, find PS = the set of tokens and productions
      referred to by the current position in each item.
```

For example, if the item being examined is 'A -> A.+M,{}', the token for this item is '+'. If we have a item 'E -> A .', there is no token associated with this item, and the item is skipped.*

```
5c.   For each token or production P in PS,
5c1.  For each item I in IS, add to a new item set IS' each item I' which
      represents the transition I through P, if one exists.
```

For example, if I is 'A -> A.+M,{}', and P is '+', then I' is 'A -> A+.M,{*}'.*

Note that some items I cannot transition through P, so they are skipped.

Implementation note: in our code base, we collapse steps 5b and 5c into a single step by constructing a mapping M which maps a token or production rule P to the item set IS' being constructed. This allows us, in one shot, to figure out the P we're transitioning through for each I, and accumulate I' into a new IS' as we construct them.

```
5c2.  Find C = the closure of IS'.
5c3.  If C is not in I, add C to V and to Q.
5c4.  Add transition t(IS,P)->IS' to our list of transitions T.
5d.   Goto 5.
```

At this point we add our propagate final algorithm to properly update the follow lists for our LALR state machine.

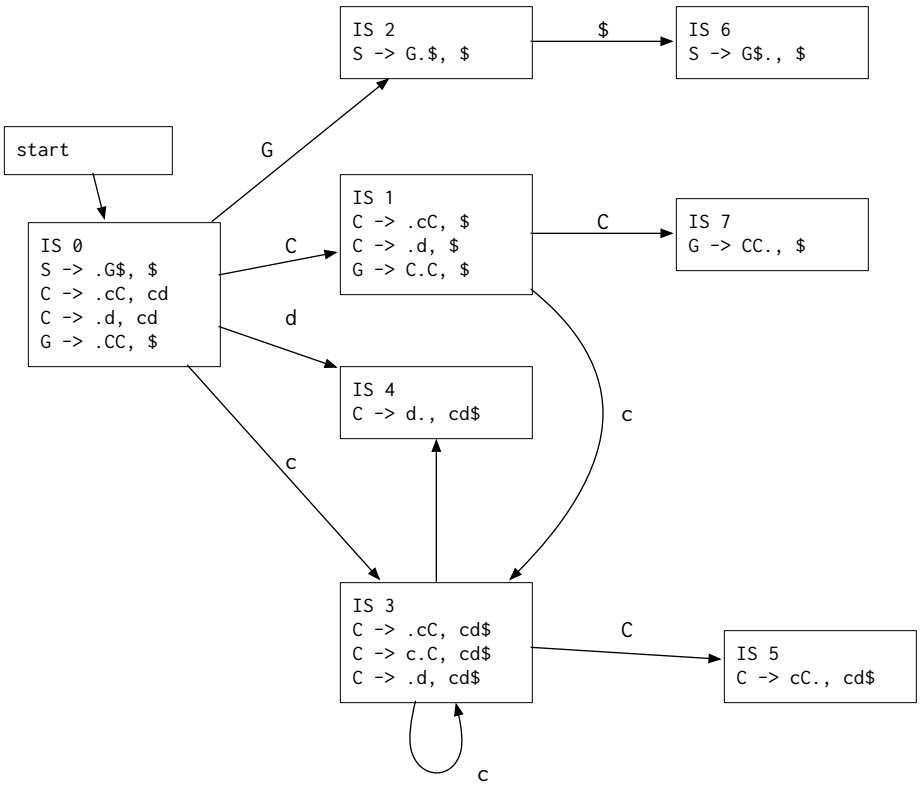
```
6.    Propagate Follow Sets, using our set of LALR Item Sets I and our list of
      transitions T.

7.    Return LALR as I, the set of item sets found, and T, the list of
      transitions between item sets.
```

Example Grammar

When applied against our example grammar:

```
S -> G $
G -> C C
C -> c C
C -> d
```

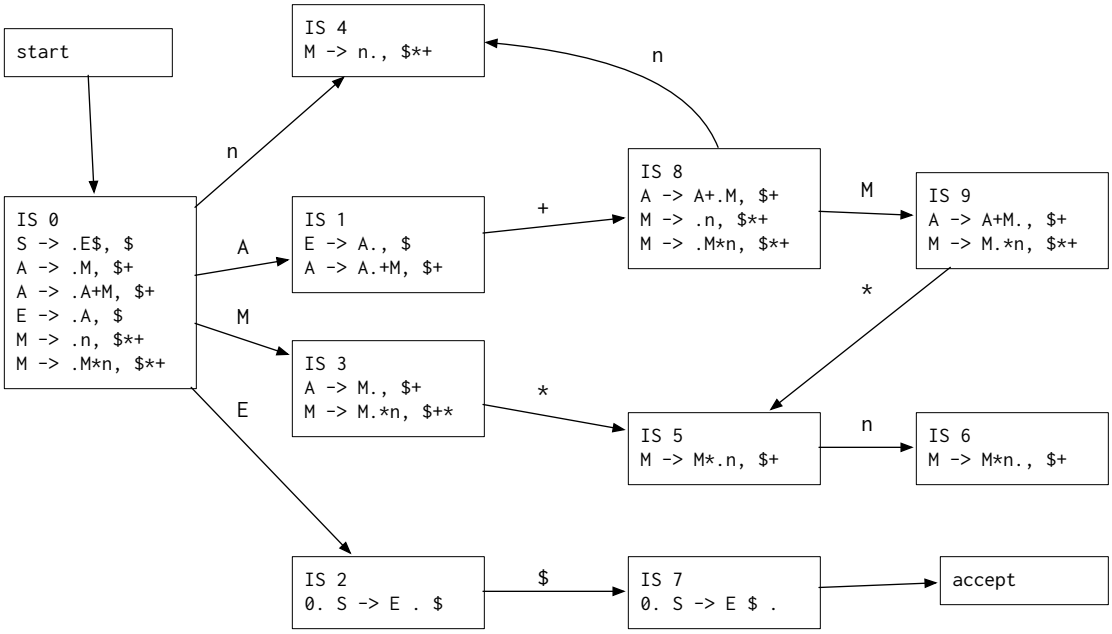
We get a state machine which is the same size as the state machine generated using the LR(0) algorithm, shaving 3 states off the overall state machine:



And when we apply this algorithm to our original grammar at the start of this document:

```
0. S -> E $
1. E -> A
2. A -> M
3. A -> A + M
4. M -> n
5. M -> M * n
```

We get the following state machine:



Notice that our state machine does not have the shift/reduce conflict in states 1, 3 or 9 that our original LR(0) grammar had. In fact, it looks very much like the LR(1) grammar above.

Constructing the LALR Action/Goto Tables

In order to use the results of our state machine in our LR(1) style parser (documented above), we need to construct the accept and goto tables. This can be done exactly as we did it with the SLR algorithm above.

```
LALR Table Construction

Given an LALR State Machine
Return the action table and goto table for our SLR state machine

1.    Build our action table A and goto table G:

1a.   Build an empty action table A = a(IS,P) -> action, where IS is an item set
      (or the index number of an item set), P is a token in our grammar, and
      action is one of "shift state", "reduce rule", "error" or "accept." By
      default fill this 2-dimensional table with error.
1b.   Build an empty goto table G = g(IS,P) -> state, where IS is an item set (or
      index), P is a production rule in our grammar, and state is the new item
      set (or index) we transition to. By default fill this 2-dimensional table
      with empty states.
1c.   Populate G with g(IS,P) with the transitions in our state machine that pass
      through a production rule P.
1d.   Populate A with a(IS,P) with the transitions in our state machine that pass
      through terminals in our grammar.
1e.   Now add the reduction rules. For our LR(0) state machine with our Follow
      set, if an item set IS contains an item where our parser position is at the
      end of the rule:
1e1.  Find the follow set of tokens T = F(P) for the production P of our rule in
      our item set IS.
```

```
1e2.   Add a reduction rule in our action table to reduce by our rule only for
       those tokens.

2.     Return our final tables A and G.
```

Some Optimizations

While the above code does correctly calculate our LALR state machine and action tables, it has a number of inefficiencies which can be optimized for.

The Kernel of an Item Set

The Dragon Book[31] defines the notion of a Kernel of an item set as the set of items in an item set that is either the initial item set [S -> .E$], or any item in the item set that does not have the position of the item set at the start. That is, for our example states IS 8 immediately above, omitting the tokens which follow, the Kernel of IS 8:

```
IS 8
A -> A+.M
M -> .n
M -> .M*n
```

Is:

```
Kernel IS 8
A -> A+.M
```

And the kernel of the start state IS 0 is:

```
Kernel IS 0
S -> .E$
```

We can always construct the full Item Set by performing the closure algorithm on the kernel of the item set. In a sense, the kernel of an item set is the smallest representation of an item set in our state machine.

This definition allows us to do a couple of things.

First, it saves a little space. Back in the days when a closet full of computers had a fraction of the power of a modern cell phone, this was important. Today, it's a little less important, but it still matters for very large grammars.

Second, we can use just the kernels to calculate our LR(0) state machines.

---

[31] Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D.; "Compilers: Principles, Techniques and Tools", Section 4.7: "Efficient Construction of LALR Parsing Tables"

Third, we can use the kernels of the items in an LR(0) state machine in order to compute our following lists in a more efficient manner than the above.

That discussion is beyond the scope of this paper.

Alternate Ways to Calculate an LALR grammar.

As we can see from the discussion about LR(0) kernels, one technique for calculating an LALR grammar is to first calculate the LR(0) grammar (either using the algorithm first outlined in this paper, or a modified algorithm adopted from The Dragon Book that uses kernels. Then using the kernel representation of the LALR grammar, follow sets are calculated for each of the items in the item sets that make up the LR(0) state machine. These follow sets are equal to the follow sets found in the algorithm above.

Another technique is given in Pager 1973[32]. In that paper, an LR(0) state machine is evaluated using two separate phases, the K-set Determination Phase, and the State-Splitting Phase. The result of the first K-set Determination Phase is an LALR state machine. After State-Splitting, an LR(1) state machine with fewer states than generated by Knuth's algorithm (above) is generated.

This technique is refined in Pager 2012[33]. In that paper, a Lane Table is constructed to remove ambiguities from an LR(0) state machine, eventually generating an LR(1) table.

A survey of these techniques, as well as others--including different parsing techniques (such as LL(k) parsing techniques and non-Chomsky Grammar parsing) can be found in Grune 2008.[34]

---

[32] Pager, David: "The lane tracing algorithm for constructing LR(k) parsers"

[33] Pager, David; Chen, Xin: "The Lane table Method of Constructing LR(1) Parsers"

[34] Grune, Dick; Jacobs, Ceriel J.H., "Parsing Techniques, A Practical Guide"

Error Handling

Up until now we've only considered what happens if we parse a correctly formatted file without errors. But the real world is full of errors, and it is worth considering the strategy for handling errors offered by OCYacc. The idea is based on the ad hoc error production handling technique outlined in Grune 2008.[35]

The idea basically involves using an error token; a special token that must be followed by another token which "swallows" erroneous input until either a next token is received, or until no more input can be found.

The Error Token

OCYacc introduces a special token, 'error', which conceptually represents an error at the specified point in the grammar. Consider, for example, a language where all statements are terminated with a semicolon:

```
statement : expression ';'      { /* Handle expression */ }
          | error ';'           { /* Handle Error */ }
          ;
```

Conceptually what we want the above declaration to say is "if we see an error, then parse up until we see a semicolon." If we see an error, we want the error statements to be skipped, then the semicolon read, the error handled, and we continue on. Of course the results cannot be relied upon, but at least we can continue scanning for more errors.

If we treat the special reserved word 'error' as a token, then as we parse our LR(1), LALR or SLR grammar, the error token would be treated as any other token, generating additional states in our state machine associated with error handling.

For example, consider the following augmented grammar:

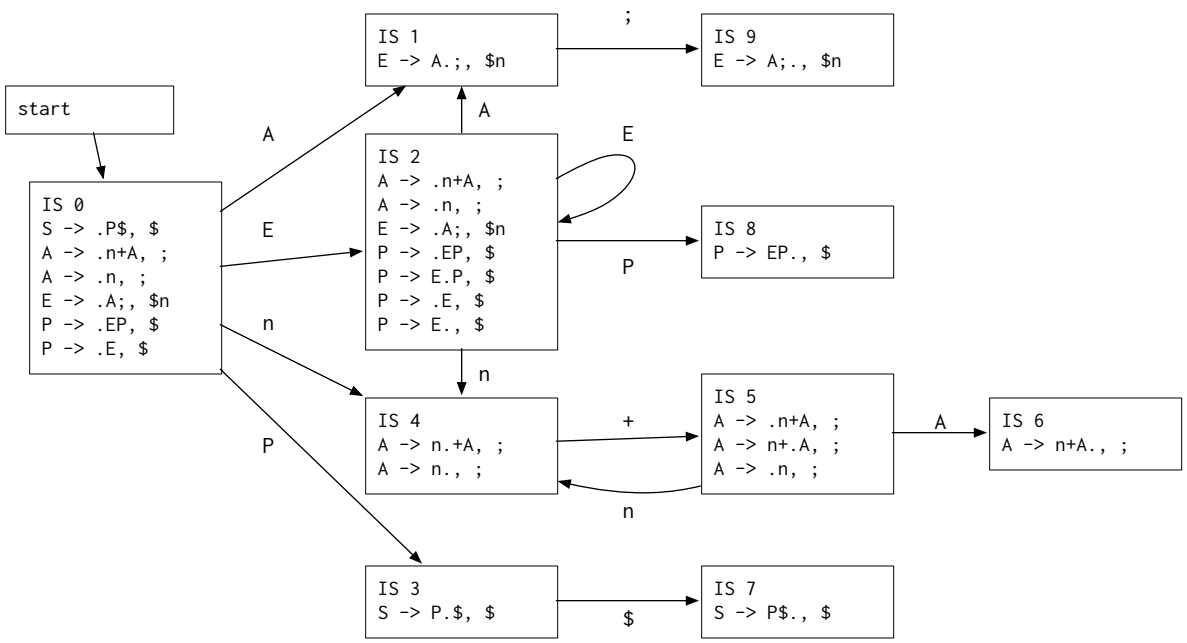
```
S -> P$
P -> EP
P -> E
E -> A;
A -> n+A
A -> n
```

This grammar parses statements of the form:

```
1 ; 1 + 2 ; 1 + 32 + 5 ;
```

The state machine generated by our LR(1) algorithm is:

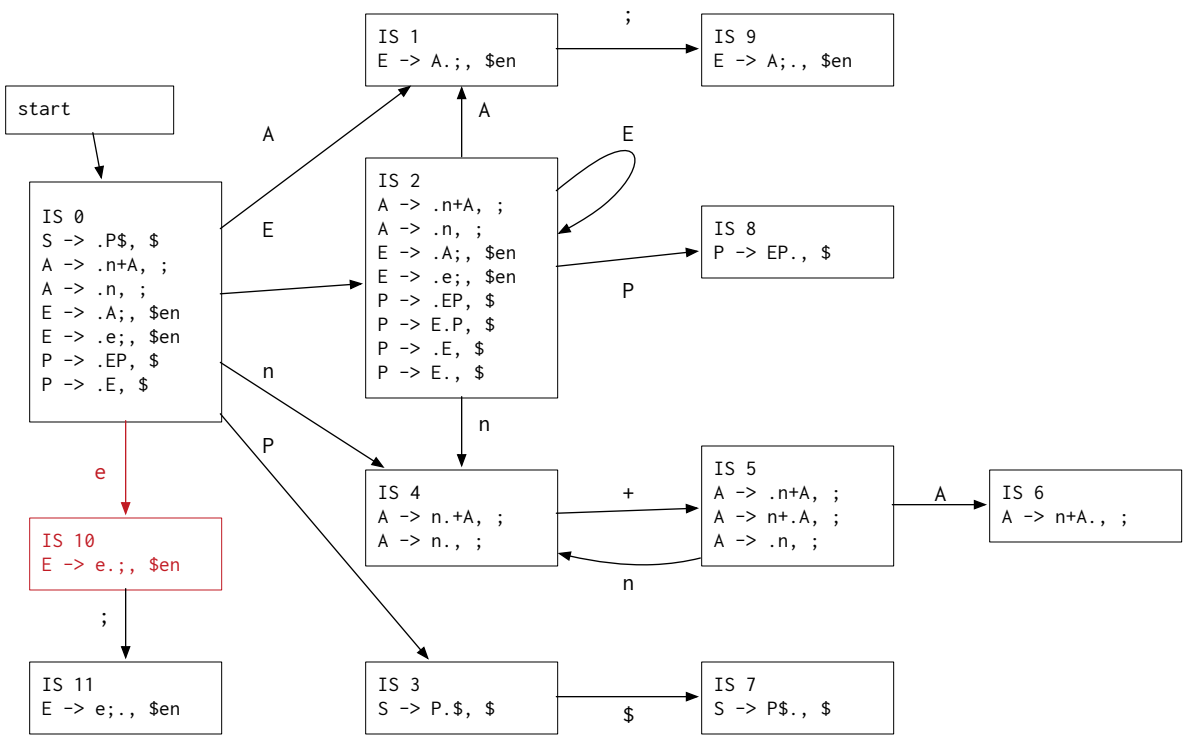
[35] Grune, Dick; Jacobs, Ceriel J.H., "Parsing Techniques, A Practical Guide", Section 16.8: Ad Hoc Methods



When we add our error grammar rule, which collapses any errors until we receive a semicolon token:

```
S -> P$
P -> EP
P -> E
E -> A;
E -> e;
A -> n+A
A -> n
```

Where 'e' is the error token, we generate the following state machine, with error states (that is, states reached through the 'error' token) highlighted in red:



Notice that the structure of our state machine remains very similar, but for the addition of two states, states IS 10 and IS 11. The transition from state IS 0 to IS 10 is triggered when we see an error token.

We can, if we wish, construct the accept and goto tables as before.

Processing Errors

The error token, however, is special; it indicates that the next token we were to shift or reduce on was not recognized by our state machine. Conceptually what we want to do is have the error token "swallow" all of the non-matching tokens until we find the next token which matches the stuff following our error token. That way, if we see the following input:

```
1 + 2 ; 1 3 ; 1 + 4 ;
```

On reading the '3' token, we would be in state 4. State 4 will shift on '+' and reduce on ';', but since we're looking at a number 'n', we are in error. The stack of our LR(1) parser would be:

```
State 4 <-- Top of stack
State 0
```

State 4 does not have an outbound transition for our error token. So we pop the stack, and we continue to pop the stack until we find a state which has an error token as an outbound item. In the process state information for partially constructed reductions are discarded as we discard items off the stack.

Once we pop the stack at state 0 (where we have an outbound error token), we then perform a shift operation to the error state.

> *If we never find an outbound error token, the only thing we can do is report a syntax error and halt the parser.*

This gives us the new stack:

```
State 10 <-- Top of stack
State 0
```

We now scan forward until we find a token which represents a transition out of our error state. In this case we scan a ';' token, and perform a shift on ';' to state 11.

```
State 11 <-- Top of stack
State 0
```

We then resume parsing as normal. In this case, we will perform a reduce operation, reducing on the error rule, which triggers execution of our error code.

If it turns out we never encounter a token which exits our error statement, we report the error (with an end of file warning along with diagnostic information), and halt the parser.

> *Note: This implies that the grammar symbol following an error token must be a token and not a production. Consider the case where it was a production. This would imply that we would somehow know when we can start parsing again to parse a production; the additional complexity would make error recovery conceptually difficult and add complexity to error handling.*

Reducing Spurious Error Messages

We borrow from YACC and Bison with the rule that error messages are disabled until we've successfully shifted 3 tokens past the token which brought us to state 11. This can be canceled by invoking a special method call, documented below. After three tokens have been successfully read, OCYacc assumes we have synchronized with the input language, and any further errors represent further problems.

Automatic Error Messages

In some cases before we start popping the stack we can determine based on the state we are in the expected tokens (by examining the current action table and seeing which columns have a shift or reduce action), and based on the rules associated with the state and the action table, we can produce an error which helps inform the user how to fix the error.

In our example above, when we are in state 4, the only outbound shift operation available is a shift on '+'. We can report to the user the error message "Syntax Error; expected '+'", in order to help the user know that he is missing an operator.

We can even extend on this idea by examining the state that follows the '+' operation and see if the current token is either shifted or reduced by the next state, in order to improve the fidelity of our error reporting. This can be used to distinguish amongst two or more outbound tokens from a given state.

For example, the current token is '3'. If we look at the state IS 5 (reached from our current state by the single outbound '+' transition), that state contains a shift operation on our current token. We can then assume with a modest degree of certainty the error is a missing '+' symbol, and even offer to insert it for the user by reporting "Syntax error; missing '+' was inserted for you."

These operations can even be done without invoking the error token.

Other Error Handling Techniques.

There are a number of other possible error handling techniques, outlined in Grune 2008.[36] However, the above techniques (the error token, automatic lookahead to determine missing symbols and automatic reporting of missing symbols if one cannot be unambiguously inserted) are implemented in OCYacc.

[36] Grune, Dick; Jacobs, Ceriel J.H., "Parsing Techniques, A Practical Guide", Chapter 16: "Error Handling."

GLR Parsers

TODO: Start working through GLR grammars.

Putting It All Together.

This section of the document *finally* discusses how the OCYacc source kit actually works.

Version 1.0 of the OCYacc tool produces LR(1) state machines from an LR(1) compliant grammar. Version 2.0 will incorporate the work in Tomita 1986[37] and optionally generate a GLR parser if requested.

The OCYacc Parser Generator

OCYacc does the following:

```
1.     Construct an internal representation of the .y input file. This is
       performed by OCYaccParser.

2.     Produce the LR1 state machine. This is performed using the
       OCYaccLR1::BuildStateMachine method call.

3.     Build the state machine goto table. This is done in
       OCYaccLR1::BuildGotoTable, and operates as below:
3a.    Build an empty goto table G = g(IS,P) -> state, where IS is an item set (or
       index), P is a production rule in our grammar, and state is the new item
       set (or index) we transition to. By default fill this 2-dimensional table
       with empty states.
3b.    Populate G with g(IS,P) with the transitions in our state machine that pass
       through a production rule P.

4.     Build the state machine action table. This is done in
       OCYaccLR1::BuildActionTable, and operates as below:

4a.    Build an empty action table A = a(IS,P) -> action, where IS is an item set
       (or the index number of an item set), P is a token in our grammar, and
       action is one of "shift state", "error" or "accept." By default fill this
       2-dimensional table with error.
4b.    Populate A with a(IS,P) with the transitions in our state machine that pass
       through terminals in our grammar.
4c.    Now add the reduction rules. For our LR(0) state machine, if an item set
       contains an item where our parser position is at the end of the rule, then
       we reduce by that rule.
4c1.   If, when adding a new reduction rule the entry in the table already has a
       reduction rule, then provide a warning and use the earlier declared rule to
       resolve the conflict.
4c2.   If adding a new rule the existing rule is a shift, then determine if both
       the shift symbol and the reduction rule have precedence associated with it.
4c2a.  If either does not have precedence, warn the user and resolve using a
       reduction.
4c2b.  If one rule has a higher precedence, then use that rule. (That is, if the
       symbol being shifted has higher precedence, keep the shift. If the rule has
       higher precedence, reduce.)
4c2c.  If both have the same precedence, determine if the precedence is left,
       right or non-assoc. If left, substitute the reduce. If right, keep the
       shift. If in error, then insert error into the table.
```

[37] Tomita, Masaru, "Efficient Parsing for Natural Language"

```
5.    Determine the accept state. This is determined by the
      OCYaccLR::FindAcceptState method. The accept state is the LR(1) Item Set
      which contains an Item that has the file position to the right of the
      terminal symbol. (That is, it has an item of the form S -> E$.)

6.    Using the constructed tables, generate the source kit. The code for this is
      contained in the class OCYaccGenerator.
```

The Generated Parser Program

OCYacc generates an LR(1) style parser, as outlined above.

Specifically the generated source kit will perform the following operations when parsing a file.

```
OCYacc LR Parsing Algorithm

Given A an action table of items A(state,token) -> action, where action is either "shift
state" or "reduce rule", G a goto table of items G(state,production), and an input A
consisting of symbols a1...an $, with $ representing the end of the file,

1.    Build a stack S and insert the initial state (an integer representing Index
      Set, and by construction assumed to be 0) onto the stack. Let ip point to
      the first symbol in input A.
2.    Let s be the state on top of the stack, let a be the symbol pointed to by
      ip.

3.    If s is the accept state, exit.

4.    If A(s,a) = shift s' then
4a.   Push s' (and associated context) onto stack S.
4b.   Advance ip to next symbol.
4c.   Goto 2.

5.    If A(s,a) = reduce rule, with rule of form P -> t1 t2 ... tl of length l.
5a.   Execute code associated with rule. (Performed in the method
      processReduction.)
5b.   Pop l symbols from the top of the stack.
5c.   Let s' = the new top state of the stack.
5d.   Push state G(s',P) (and associated context) onto stack.
5d.   Goto 2.

6.    If A(s,a) is error
6a.   Note the location of the error for error reporting.
6b.   Scan backwards in the stack S from the end to the front, looking for state
      s' in S where value A(s',error) which is not an error. (That is, scan
      backwards until we find a state with an error token.
6c.   If state s' can be found
6c1.  Unwind the stack until s' is at the top of the stack.
6b2.  Advance the token stream A until we find a token a' which can shift from
      state s'. If we hit the end of the file, report an error.
6b3.  Resume processing with state s' and a'. This should continue until we hit
      the reduction for our error state, and the appropriate error code can be
      executed.

6d.   If state s' cannot be found, perform ad-hoc error handling:
```

6d1. If there is only one a' for which A(s,a') is a shift production
6b1a. Print a report that the token is missing.
6b1b. Perform shift by a' and continue.

6d2. If there is only one possible reduction for A(s,a') for any a', perform the reduction. This is guaranteed to later trigger a shift error.

6d3. If there are five or fewer possible symbols a' for which A(s,a') is not an error, print a list of all five symbols, and pick the first a' found, and resume processing.

6d4. Otherwise, advance ip to next symbol, print error that we are skipping a symbol and resume processing.