LALR(1) Parsing Study

by Kevin Dietz

Created August 2017

KevinKnowscs@gmail.com

http://github.com/KevinKnowscs/lalr1

FRONT MATTER

This notebook was created in its entirety using a 2017 generation, 10.5" iPad Pro, an Apple Pencil, and the GoodNotes 4 app.

I created this notebook because:

- I've always been fascinated by LALR(1) parsing and wanted to better understand the process.
- Some projects I'm interested in require an embedded LALR(1) parser generator (not statically-generated as bison does)
- I wanted to create a real-world GoodNotes notebook, to see how effective it is at creating purely digital content for complex subject matter involving a mixture of normal text, notation, sketches, and photographs of external content.

FOR QUESTIONS OR EMPLOYMENT INQUIRIES, PLEASE CONTACT ME AT:

KevinKnowsCS@gmail.com

Part 1

Introduction / Concepts

Why Study LALR(1) Parsing?

Why don't I "just use bison"

- It's interesting and challenging
- Bison is fine, but it is written in, and generates C code output
- So, if you are developing in something besides C, and can't find a good parser/compiler toolkit for your language, you may need to write one from scratch, or modify an existing one
- If you want to extend or tweak the basic LR parsing algorithm
  - Example: You are writing a text editor or IDE and want to write a high-performance syntax highlighter and refactorer that works on-the-fly as the user types
- If you have any interest in compilers, interpreters, debuggers, etc., it is necessary to understand this algorithm

An overview of why anyone would need to learn or know LALR(1) parsing

Greek Letters Review

Greek letters signify a string of zero or more symbols (terminals or nonterminals)

α - Alpha
β - Beta
γ - Gamma
δ - Delta
ε - Epsilon, special - used for "empty"
η - Eta
ω - Omega

It helps to know your Greek Letters

Overview of Notation

Understanding the notation used in the Dragon Book is critical to successfully understanding the content.

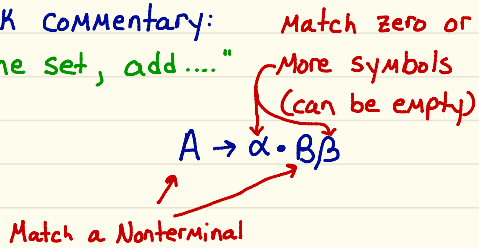
DO NOT TRY TO SKIP PAST LEARNING THE NOTATION

Description	Represents	Examples
• Lowercase letters early in the alphabet	A single terminal	a, b, c
• Uppercase letters early in the alphabet	A single nonterminal	A, B, C
• Uppercase letters late in the alphabet	Any single symbol (Terminal or Nonterminal)	X, Y, Z
• Lowercase letters late in the alphabet	A string of zero or more terminals	u, v, w
• Lower case Greek letters Note: In some cases, the Greek letter are subscripted	A string of zero or more symbols (terminals or nonterminals)	α, β, γ $\alpha_1, \alpha_2, \alpha_3$

Example: Here is some typical Dragon Book commentary:

"For every item $A \to \alpha \cdot B\beta$ in the set, add ...."

A sentence like this is instructing us to perform a pattern match as follows:



Learn the notation

Backus-Naur Form (BNF) Notation

I assume anyone reading this notebook is generally familiar with grammars and the use of BNF notation to describe them. I shall only provide a very brief overview here.

A grammar is described using a set of terminals, a set of nonterminals, and a set of productions.

Terminals are the root elements of the input string, such as identifiers, string and numeric literals, operators, etc. The input string is a series of terminals. Typically terminals are recognized by a lexer, whose job it is to break the input string up into terminals (sometimes also called 'tokens'). Thus, the input to the parser is a sequence of terminals, or tokens.

Nonterminals represent the higher level structure of the language, such as expressions, terms, factors, block statements, if statements, function blocks, etc.

Productions are shown in the form: $A \rightarrow \alpha$. This is telling us that we can substitute the nonterminal on the LHS with all the terminals and nonterminals on the RHS.

A very brief introduction to BNF.

Why Learning / Describing LALR(1) is Hard

Learning LALR(1) is a deeply conceptual endeavor.

We are using a domain-specific notation to describe the theory and the process for building the parsing table, which, when run through the LR parsing algorithm on a given input string, will result in a series of "shifts" and "reduces", while also traipsing through the internal parsing context of states and a stack.

So, there are a lot of "levels of indirection" in the process.

There are a lot more moving parts to this algorithm than simple algorithms like searching or sorting.

Why Learning LALR(1) is Hard

Furthermore, the dynamic movement and interplay across these concepts makes it difficult to digest through self-teaching merely by reading static text in a book.

Most people who study this have the advantage of it being taught to them by a professor in a classroom context, through lectures and recitations.

Static text is not a good medium by which to teach and learn such a complex subject.

With the advent of YouTube, today we can augment this material with online lectures freely available to anyone.

Tips for Learning LALR(1) Parsing

Employing both non-linear and deep learning will help learn LALR(1) faster and more thoroughly.

Tip #1

Learn the notation. This is critical.

Tip #2

It's okay to employ non-linear learning and scan ahead sometimes. This helps to understand "why" a piece of knowledge is necessary.

But, you also need to...

Tip #3

Dig in!! Lean into the pain. Work through each part of the process until you understand it. Ultimately, this will help you go faster.

Tips for learning LALR(1)

The LR Parsing Algorithm - Inputs & Outputs

Inputs: 1) A sequence of terminals (typically obtained from the output of a lexical analyzer)
2) The parser table (typically obtained as the output of an SLR, LR(1), or LALR(1) parser generator)

Outputs: 1) Accept or reject the input string
2) Optionally, a series of shift and reduce events that can be announced to the client application (for example, a compiler). Depending on the host language, the events may be implemented as inline-generated code, callback functions, lambda expressions, etc.

The Parser Table: A set of state information entries, indexed by a state specifier (a state number, etc.)

Each state entry contains an actions dictionary, indexed by terminals, and a gotos dictionary, indexed by nonterminals.

Describing the inputs and outputs to an LR parser.

The LR Parsing Algorithm - Actions & Gotos

Entries in the actions dictionary can be one of 4 actions:

shift: Shift the current terminal from the input string onto the stack and also shift the shift actions associated state onto the stack.

reduce: Find the production associated with reduce action. Pop 2 * the number of symbols on the RHS of the production (Example: If the reduction is for $W \rightarrow XYZ$ then pop 6 items off the stack).

The item now at the top of the stack is the "exposed state". Use this state to look up the entry in the gotos table to find the next state. Push the nonterminal on the LHS of the production, along with the next state onto the stack.

accept: Stop parsing and announce acceptance of the input string

error: Stop parsing and announce rejection of the input string.

Describing the possible actions that can be performed at each step of an LR parser.

The LR Parsing Algorithm - The Process

First, initialize a parsing stack containing the initial state.

Then, repeat the following steps until an accept or error action is reached:

- Find the state designated by the item at the top of the stack and lookup the state context for that state in the parsing table
- In the actions dictionary for the state context, lookup the action using the next terminal in the input string.
- Perform the designated action, as described in the previous page, pushing and popping items to and from the stack as-needed.

I will give several examples of this throughout the notebook. If the process is still unclear at this point, don't fret. It will become clear as the notebook progresses.

This notebook describes how to produce a parsing table for an arbitrary target grammar.

The basic LR parsing process.

Part 2

SLR Construction

SLR Construction - Overview

1) Create an augmented grammar by adding $S' \rightarrow S$ to the target grammar

2) Start the Sets-of-Items construction by computing closure($\{S' \rightarrow \bullet S\}$) to create I_0, the first set of items

3) Apply the goto operation to I_0, and all other subsequent sets of items to complete the Sets-Of-Items construction

4) Create the parsing table, consisting of actions and gotos by applying the FIRST and FOLLOW functions to the DFA

The terminology, notation, theory and examples of each of these steps will be covered in the next several pages

SLR Construction - The Closure Operation

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

Formal Definition

If $A \rightarrow \alpha \bullet B\beta$ is in closure(I), and $B \rightarrow \gamma$ is a production in G, then add $B \rightarrow \bullet\gamma$ to closure(I)

Informal Definition

Look for items in closure(I) that have a • preceding a nonterminal, then add new items to the set for every production for that nonterminal with a • at the beginning. Keep going until there are no more items to add

Example

A • precedes the E, so add all the E productions

I_0 starts with $\{ E' \rightarrow \bullet E \}$ so we add the E production items:

$E \rightarrow \bullet E + T$

$E \rightarrow \bullet T$

Put a • at the beginning of each item

to the closure set. Then we keep going with T, etc.

As I discuss in a later page, the closure operation allows us to compute all the possible ways that we could have gotten into that particular state

SLR Construction - Closure Operation on I_0

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

Compute I_0 = closure($\{E' \rightarrow \bullet E\}$)

I_0 = $E' \rightarrow \bullet E$ ← Seeded item

$E \rightarrow \bullet E + T$ } E productions

$E \rightarrow \bullet T$

$T \rightarrow \bullet T * F$ } T productions

$T \rightarrow \bullet F$

$F \rightarrow \bullet (E)$ } F productions

$F \rightarrow \bullet id$

There are no more dots preceding nonterminals, so we are done

It makes intuitive sense that we would start with $E' \rightarrow \bullet E$, as that item indicates that we have not yet processed any input

We begin the Sets-Of-Items construction by constructing I_0 = closure($E' \rightarrow \bullet E$) which is the first production in the augmented grammar

SLR Construction - Theory of Items, Dots, & Gotos

Theory

* The "dot" (•) represents a placeholder of how much of the input we've seen so far
* After • is what we might see next
* Each "item" is a possible way we could have gotten into this state
* We need to produce the "closure" so that we capture all the possible ways we could have entered the state

The set of items is a cognitive tool that allows us to keep track of where we are in the input string

Each item is a production plus a • acting as the placeholder

As we'll see later, creating the full set of items starting with I_0 and following all the gotos allows us to construct a DFA that recognizes "viable prefixes" of the grammar

SLR Construction - Even More Theory on Closures

What is a closure?

Why does it work?

As we've seen I_0 has the seed item:

$$E' \rightarrow \bullet E$$

This means we're expecting to see an E next in the input.

But how do we get an E? By the E productions, of course.

So, by virtue of being in state I_0, it also could mean that we are at the start of one of the E productions. Therefore, we add

$$E \rightarrow \bullet E + T \quad \text{and}$$
$$E \rightarrow \bullet T$$

to the context for this state.

Examining what the closure operation actually accomplishes, and why it works.

SLR Construction – Closure Theory

What is a closure?

Why does it work?

But now we see that

$E \rightarrow \bullet T$

is now part of I_0. So that means

"Oh, in addition to perhaps seeing an E next, I could also possibly expect to see a T next."

I_0 doesn't know what is going to be found next. It needs to identify the total list of nonterminals it <u>might</u> see next, and be prepared to branch off into the next state when the parser finally returns to I_0 and tells us what nonterminal was actually found.

I'm expecting to see an E next, but an E might start with a T, so that means I might see either an E <u>OR</u> a T next.

SLR Construction - More Closure Theory

What is a closure?

Why does it work?

So the nonterminal production items in I_0 are essentially saying

> "I need to see either an E, a T, or an F. So Mr. Parser, when you find one of those, please get back with me"

So the parser goes off merrily crunching away doing shifts and reduces, and at some point it finally does a reduction of $T \rightarrow F$ (for example) with State 0 exposed on the top of the stack, the parser is essentially saying

> "Here you go State 0. I found you a T that you said you were looking for"

SLR Construction - Identifying Gotos of I_0

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

I_0 =

	Gotos For
$E' \rightarrow \bullet E$	E
$E \rightarrow \bullet E + T$	
$E \rightarrow \bullet T$	T
$T \rightarrow \bullet T * F$	
$T \rightarrow \bullet F$	F
$F \rightarrow \bullet (E)$	$($
$F \rightarrow \bullet id$	id

We need a goto for any symbol (terminal or nonterminal) preceded by a dot

Therefore, we need gotos for:

E, T, F, $($, id

So we create new sets

I_1 = goto (I_0, E)
I_2 = goto (I_0, T)
I_3 = goto (I_0, F)
I_4 = goto (I_0, '$($')
I_5 = goto (I_0, id)

Gotos let us develop the actions table that guide us through the DFA to recognize the SLR grammar

SLR Construction - Theory of Gotos

This makes intuitive sense. If we're in I_0 ...

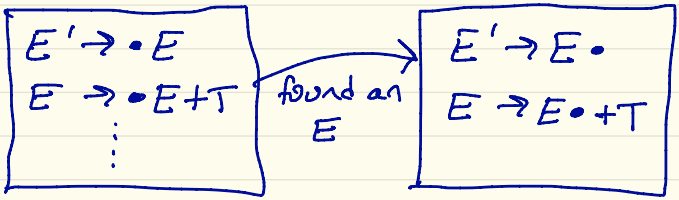
$I_0 =$

$E' \to \bullet E$

$E \to \bullet E + T$

$\vdots$

and the parser is now telling us "I found you an E", it makes logical sense that we would move to a new state that progresses past the E and is now expecting to see the rest of what I_0 was expecting after the E.



Why Gotos work

SLR Construction - Seeding the Goto Set items

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

To create the seed items of goto(I, X)

Formal Definition

Add $A \rightarrow \alpha X \cdot \beta$ to goto(I, X) for any production $A \rightarrow \alpha \cdot X \beta$ in I

Informal Definition

Look for productions in I that have a • preceding the symbol of interest, and move the • one space to the right

Example

If $E' \rightarrow \cdot E$ is in I, then add $E' \rightarrow E \cdot$ to goto(I, E)

To complete the goto set, we perform the closure operation

By creating and following the goto sets from the initial I_0 set, we can construct the full set of states to recognize the SLR grammar

SLR Construction - Constructing I_1

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

From Previous Page

I_0 =
- $E' \rightarrow \bullet E$
- $E \rightarrow \bullet E + T$
- $E \rightarrow \bullet T$
- $T \rightarrow \bullet T * F$
- $T \rightarrow \bullet F$
- $F \rightarrow \bullet (E)$
- $F \rightarrow \bullet id$

$I_1 = goto(I_0, E)$

Move the dots to the right of the E

Then we construct I_1 as follows:

$I_1 = Closure(\{ E' \rightarrow E \bullet ,\ E \rightarrow E \bullet + T \})$

I_1 =
- $E' \rightarrow E \bullet$
- $E \rightarrow E \bullet + T$

> Seeded items

There are no dots preceding any nonterminals, so we are done

We create $goto(I_0, E)$ by first seeding the result set by moving the dots to the right of the E, then performing the closure

SLR Construction - Constructing I_2

Grammar

0. $E' \to E$
1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

0 = Augmented Production

1-6 = Productions from target grammar

From Previous Page

$I_0 =$ $E' \to \bullet E$
$E \to \bullet E + T$
$E \to \bullet T$
$T \to \bullet T * F$
$T \to \bullet F$
$F \to \bullet (E)$
$F \to \bullet id$

$\}\ I_2 = goto(I_0, T)$

$I_2 = closure(\{ E \to T \bullet,\ T \to T \bullet * F \})$

$I_2 =$ $E \to T \bullet$
$T \to T \bullet * F$
$\rangle$ Seeded items

There are no more dots preceding any nonterminals, so we are done

We continue following the gotos of I_0 by constructing $I_2 = goto(I_0, T)$

SLR Construction - Constructing I_3

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

From Previous Page

$I_0 = E' \rightarrow \bullet E$
$E \rightarrow \bullet E + T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet T * F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet (E)$
$F \rightarrow \bullet id$

$I_3 = goto(I_0, F)$

$I_3 = Closure(\{T \rightarrow F \bullet\})$

$I_3 = T \rightarrow F \bullet$ — Seeded item

No other items to add

We continue following the gotos of I_0 by constructing $I_3 = goto(I_0, F)$

SLR Construction - Constructing I_4

Grammar

0. $E' \to E$
1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

0 = Augmented Production

1-6 = Productions from target grammar

From Previous Page

I_0 = $E' \to \bullet E$
$E \to \bullet E + T$
$E \to \bullet T$
$T \to \bullet T * F$
$T \to \bullet F$
$F \to \bullet (E)$ - $I_4 = goto(I_0, '(')$
$F \to \bullet id$

$I_4 = closure(\{ F \to (\bullet E) \})$

I_4 = $F \to (\bullet E)$ - Seeded item
$E \to \bullet E + T$
$E \to \bullet T$
$T \to \bullet T * F$
$T \to \bullet F$
$F \to \bullet (E)$
$F \to \bullet id$

This is the only item different from I_0

Items added similar to how I_0 was constructed

Note that I_4 is very similar to, but not identical to I_0

We continue following the gotos of I_0 by constructing $I_4 = goto(I_0, '(')$

SLR Construction - Constructing I_5

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

From Previous Page

$I_0 = E' \rightarrow \bullet E$
$E \rightarrow \bullet E + T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet T * F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet (E)$
$F \rightarrow \bullet id$ - $I_5 = goto(I_0, id)$

$I_5 = closure(\{ F \rightarrow id \bullet \})$

$I_5 = F \rightarrow id \bullet$ - Seeded item

No other items to add

We continue following the gotos of I_0 by constructing $I_5 = goto(I_0, id)$

This completes the gotos originating from I_0

SLR Construction - Gotos of I_1 / Constructing I_6

Grammar

0. $E' \to E$
1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

0 = Augmented Production

1-6 = Productions from target grammar

We've completed the gotos of I_0.

Now, we apply the same operations to the rest of the sets, starting with I_1

$I_1 = E' \to E \bullet$
$\quad\quad E \to E \bullet + T$

We see a • preceding the + in the second item, so we need

$I_6 = \text{goto}(I_1, +)$

$I_6 = E \to E + \bullet T$ — seeded item
$\quad\quad T \to \bullet T * F$
$\quad\quad T \to \bullet F$
$\quad\quad F \to \bullet (E)$
$\quad\quad F \to \bullet id$

Closure items added

We continue applying and following the gotos of the sets we've created

SLR Construction - Gotos of I_2 / Constructing I_7

Grammar

0. $E' \to E$
1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

0 = Augmented Production

1-6 = Productions from target grammar

$I_2 =$ $E \to T \bullet$
$T \to T \bullet * F$

$I_7 = goto(I_2, *)$

$I_7 =$ $T \to T * \bullet F$ - Seeded item
$F \to \bullet (E)$ } Closure items
$F \to \bullet id$ } added

We continue applying and following the gotos of the sets we've created

I_3 has no gotos, so we proceed to I_4 next.

SLR Construction - Gotos of I_4

Grammar

0. $E' \to E$
1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

0 = Augmented Production

1-6 = Productions from target grammar

I_4 = $F \to (\bullet E)$
$E \to \bullet E + T$
$E \to \bullet T$
$T \to \bullet T * F$
$T \to \bullet F$
$F \to \bullet (E)$
$F \to \bullet id$

Need gotos for
- $> E$
- $> T$
- $- F$
- $- ($
- $- id$

But some of these are duplicates, so we do not produce new states

goto(I_4, T) is the same as I_2

goto(I_4, F) is the same as I_3

goto(I_4, '(') is the same as I_4 (it goes to itself)

goto(I_4, id) is the same as I_5

The only new state is goto(I_4, E) which we will label I_8

We'll do I_8 on the next page. The rest of the states are duplicates

SLR Contruction - Constructing $I_8 = goto(I_4, E)$

Grammar

0. $E' \to E$
1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

0 = Augmented Production

1-6 = Productions from target grammar

$I_4 =$
$F \to (\bullet E)$
$E \to \bullet E + T$
$E \to \bullet T$
$T \to \bullet T * F$
$T \to \bullet F$
$F \to \bullet (E)$
$F \to \bullet id$

> $I_8 = goto(I_4, E)$

$I_8 = closure(\{ F \to (E \bullet),\ E \to E \bullet + T \})$

$I_8 =$
$F \to (E \bullet)$
$E \to E \bullet + T$

> Seeded items

No more items to add

This completes the gotos of I_4. I_5 has no gotos, so we proceed to I_6 on the next page.

SLR Construction - Gotos of I_6

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

$I_6 =$ $E \rightarrow E + \bullet T$
$T \rightarrow \bullet T * F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet (E)$
$F \rightarrow \bullet id$

Need gotos for
> T
- F
- $($
- id

Again, some are duplicates

goto (I_6, F) is the same as I_3

goto $(I_6, '(')$ is the same as I_4

goto (I_6, id) is the same as I_5

The only new state is goto(I_6, T) which we will label I_9

One new state, goto(I_6, T) will originate from I_6. We'll do that one next.

SLR Construction - Constructing $I_9 = goto(I_6, T)$

Grammar

0. $E' \to E$
1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

0 = Augmented Production

1-6 = Productions from target grammar

$I_6 = E \to E + \bullet T$
$T \to \bullet T * F$
$T \to \bullet F$
$F \to \bullet (E)$
$F \to \bullet id$

$I_9 = closure(\{ E \to E + T \bullet,\ T \to T \bullet * F \})$

$I_9 = E \to E + T \bullet$
$T \to T \bullet * F$

> Seeded items

No more items to add

This completes the gotos originating from I_6

SLR Construction – Gotos of I_7

Grammar

0. $E' \to E$
1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

0 = Augmented Production

1-6 = Productions from target grammar

$I_7 = T \to T * \bullet F$
$F \to \bullet (E)$
$F \to \bullet id$

Need gotos for
- F
- (
- id

goto $(I_7, '(')$ is the same as I_4

goto (I_7, id) is the same as I_5

goto (I_7, F) is new. We'll call it I_{10}

I_{10} = closure $(\{ T \to T * F \bullet \})$

$I_{10} = T \to T * F \bullet$ seeded item

No more items to add

We're almost done. Only one more new state to add.

SLR Construction - Gotos of I_8

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

Need gotos for

$I_8 = F \rightarrow (E \bullet)$ -)

$E \rightarrow E \bullet + T$ - +

goto $(I_8, +)$ is the same as I_6

goto $(I_8, ')')$ is new. We'll call it I_{11}

$I_{11} = closure(\{F \rightarrow (E) \bullet\})$

$I_{11} = F \rightarrow (E) \bullet$ - seeded item

No more items to add

This is the last set to add. To satisfy ourselves, we'll next look at I_9, I_{10}, and I_{11} to show that they have no new gotos.

SLR Construction - Gotos of I_9, I_{10}, I_{11}

Grammar

0. $E' \to E$
1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

0 = Augmented Production

1-6 = Productions from target grammar

$I_9 = E \to E + T \bullet$
$\quad\quad T \to T \bullet * F$

goto(I_9, *) is same as I_7

$I_{10} = T \to T * F \bullet$

No gotos

$I_{11} = F \to (E) \bullet$

No gotos

This completes the Sets-of-Items construction

We are done! Hooray. On the next 3 pages, we'll review and summarize the work.

SLR Construction - Summary of States

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

I_0 = closure($\{E' \rightarrow \bullet E\}$)
I_1 = goto(I_0, E)
I_2 = goto(I_0, T) = goto(I_4, T)
I_3 = goto(I_0, F) = goto(I_4, F) = goto(I_6, F)
I_4 = goto(I_0, '(') = goto(I_6, '(') = goto(I_7, '(')
I_5 = goto(I_0, id) = goto(I_6, id) = goto(I_7, id)
I_6 = goto(I_1, +) = goto(I_8, +)
I_7 = goto(I_2, *) = goto(I_9, *)
I_8 = goto(I_4, E)
I_9 = goto(I_6, E)
I_{10} = goto(I_7, F)
I_{11} = goto(I_8, ')')

↑ and goto(I_4, id)

Note that some of the gotos yield the same result set

When this happens we do not create a new set

Shows the final states produced by our hard work of following all the gotos of I_0, and the subsequently created sets

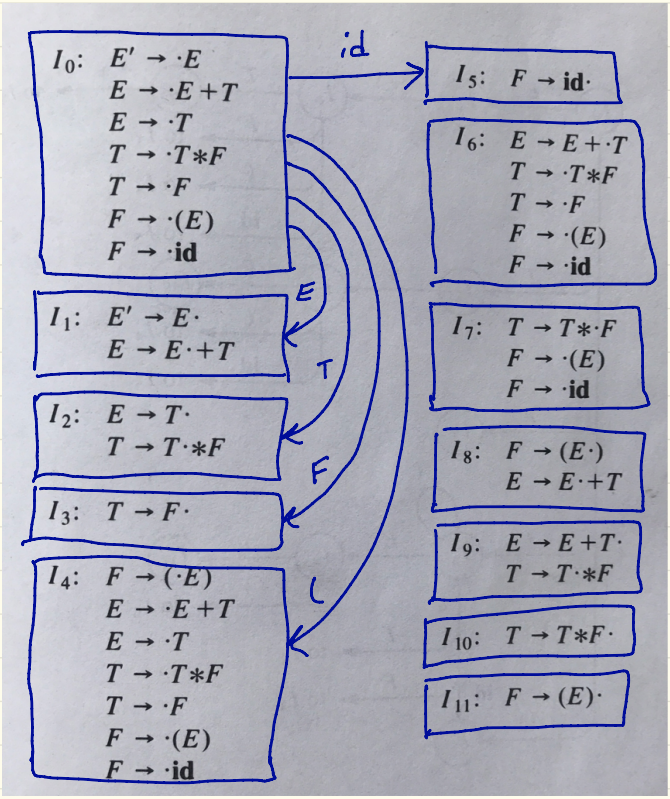
SLR Construction - Final List of States

Grammar

0. $E' \to E$
1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

0 = Augmented Production

1-6 = Productions from target grammar



DFA edges originating from I_0 are shown

Here is the final list of states taken directly from Dragon Book. See next page for the full DFA.

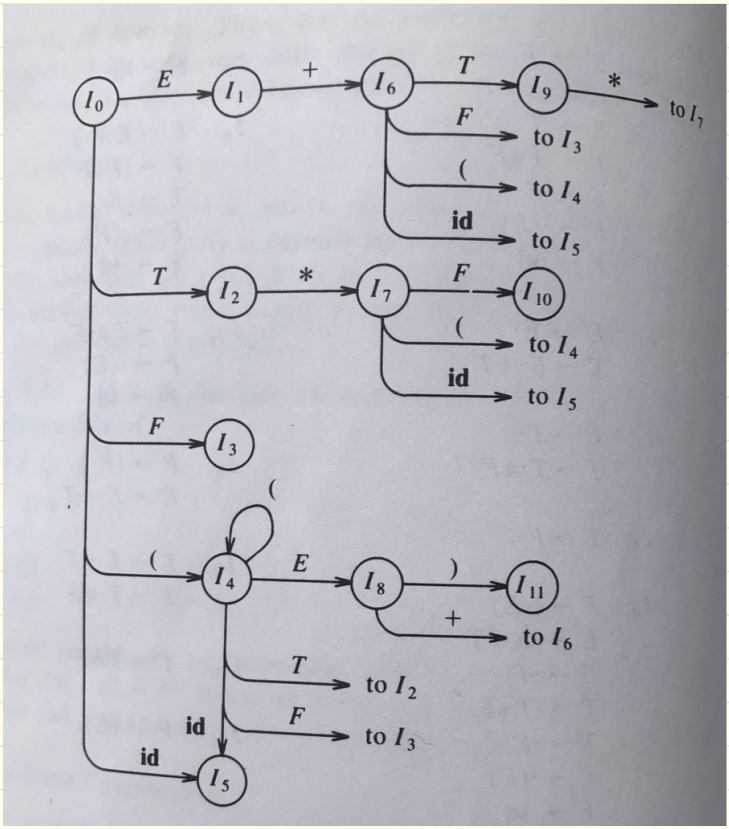
SLR Construction - Final DFA

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar



Final DFA taken directly from Dragon Book

SLR Construction - Using NFAs instead of DFAs

The next few pages show an alternate approach that will eventually arrive at the same DFA

If each state of D in Fig. 4.36 is a final state and I_0 is the initial state, then D recognizes exactly the viable prefixes of grammar (4.19). This is no accident. For every grammar G, the *goto* function of the canonical collection of sets of items defines a deterministic finite automaton that recognizes the viable prefixes of G. In fact, one can visualize a nondeterministic finite automaton N whose states are the items themselves. There is a transition from $A \to \alpha \cdot X\beta$ to $A \to \alpha X \cdot \beta$ labeled X, and there is a transition from $A \to \alpha \cdot B\beta$ to $B \to \cdot\gamma$ labeled ϵ. Then *closure*(I) for set of items (states of N) I is exactly the ϵ-*closure* of a set of NFA states defined in Section 3.6. Thus, *goto*(I, X) gives the transition from I on symbol X in the DFA constructed from N by the subset construction. Viewed in this way, the procedure *items*(G') in Fig. 4.34 is just the subset construction itself applied to the NFA N constructed from G' as we have described.

We could take a different approach. Create an NFA as described above, then use the subset construction to convert it to a DFA.

We would arrive at the same result. It would be the same result, because the closure/goto method essentially <u>is</u> the subset construction method.

Let's look at a few states to see how this would work

A brief tangent into an NFA technique allows us to deepen our understanding of the process

SLR Construction - Using NFAs instead of DFAs

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

The sets would be the items themselves

For this grammar, the enumeration of all items is:

0. $E' \rightarrow \bullet E$
1. $E' \rightarrow E \bullet$
2. $E \rightarrow \bullet E + T$
3. $E \rightarrow E \bullet + T$
4. $E \rightarrow E + \bullet T$
5. $E \rightarrow E + T \bullet$
6. $E \rightarrow \bullet T$
7. $E \rightarrow T \bullet$
8. $T \rightarrow \bullet T * F$
9. $T \rightarrow T \bullet * F$
10. $T \rightarrow T * \bullet F$
11. $T \rightarrow T * F \bullet$
12. $T \rightarrow \bullet F$
13. $T \rightarrow F \bullet$
14. $F \rightarrow \bullet (E)$
15. $F \rightarrow (\bullet E)$
16. $F \rightarrow (E \bullet)$
17. $F \rightarrow (E) \bullet$
18. $F \rightarrow \bullet id$
19. $F \rightarrow id \bullet$

In the NFA method, the initial states would be the items themselves. We would therefore start with an NFA with 20 states.

SLR Construction - Using NFAs Instead of DFAs

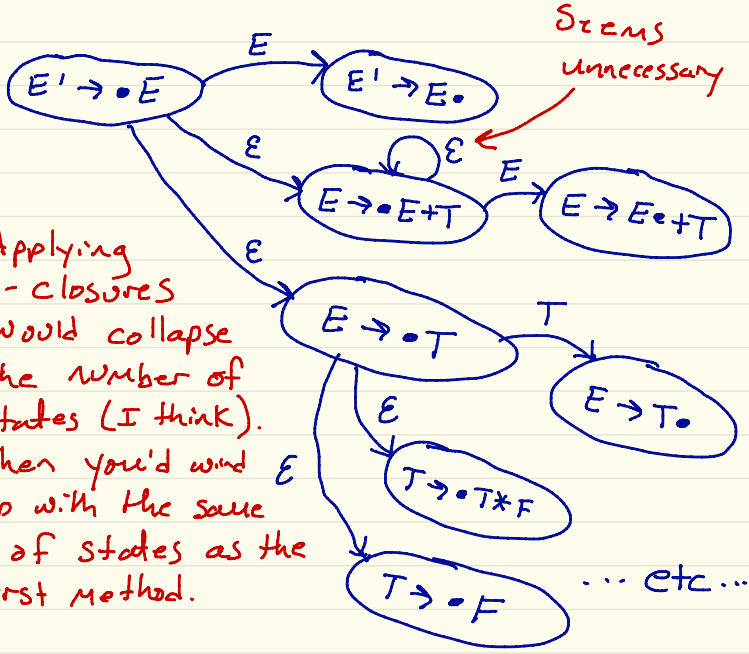
Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

The states and edges would start to look like:



Applying ε-closures would collapse the number of states (I think). Then you'd wind up with the same # of states as the first method.

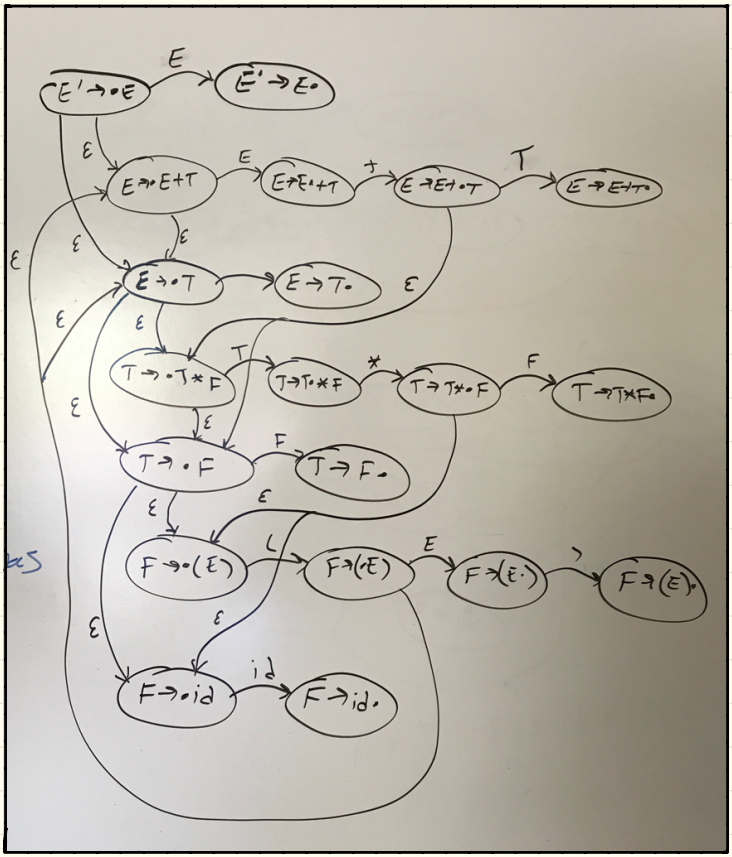
The rest of the NFA is not shown.

If we applied the subset construction technique to the above NFA, we would arrive at the same DFA as before

Shows the beginning of the potential NFA. Understanding this isn't strictly necessary, but it illustrates a deeper understanding of the process.

SLR Construction - Using NFAs

Resulting NFA



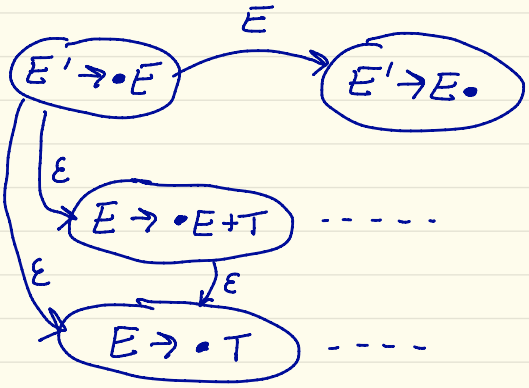
Full NFA before applying ε-closures

The full NFA

SLR Construction - The Theory of the NFA

Why does the NFA work?

What is the NFA really trying to tell us?



If we see an E, then we can move along the edge labeled E.

But if we don't yet have an E, and are in a state where we expect to see an E next, then we can follow the ε transitions into the E production states, that will give us all or part of the E that we are expecting to see

Trying to reason out the theory and purpose behind the NFA. The book doesn't explain this very well.

SLR Construction - NFA after ε-closures

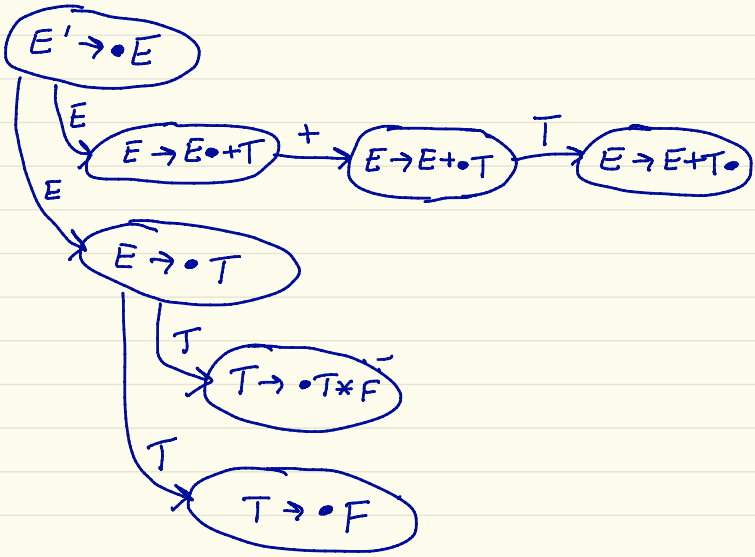
Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

After applying ε-closures, we'd be left with an NFA that exactly mirrors the grammar structure.



SLR Construction - Creating the Parsing Table

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

We've seen the DFA. But that's only an intermediate result of the process.

What we really need is the actual parsing table that will tell us when to perform shifts and reduces.

To start we need to backtrack to section 4.4 to compute the FIRST and FOLLOW functions.

SLR Parsing Tables

Now we shall show how to construct the SLR parsing action and goto functions from the deterministic finite automaton that recognizes viable prefixes. Our algorithm will not produce uniquely defined parsing action tables for all grammars, but it does succeed on many grammars for programming languages. Given a grammar, G, we augment G to produce G', and from G' we construct C, the canonical collection of sets of items for G'. We construct *action*, the parsing action function, and *goto*, the goto function, from C using the following algorithm. It requires us to know FOLLOW(A) for each nonterminal A of a grammar (see Section 4.4).

But to figure out the FOLLOW sets, we need to know the FIRST sets

We need to compute the FIRST and FOLLOW functions before we can complete the parsing table.

SLR Construction - Computing FIRSTS

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

For any grammar symbol X, FIRST(X) gives us the set of all possible terminals that X could begin with.

If X is a terminal, then FIRST(X) = {X}. That makes sense. A terminal symbol obviously begins with itself.

But for nonterminals, we need to figure out all the possible terminals that could be the first terminal of a substring of input that can reduce to X.

For example, consider F, which can either be an id or a "(E)". So we can easily intuit that

$$FIRST(F) = \{ id, (\}$$

Introduction to FIRST sets

SLR Construction - FIRST sets

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

FIRST(F) is an easy one though. Our human minds can scan the grammar and recognize that F is at the "bottom" of the grammar, and thus easily deduce that F must start with either an id or a (.

But even for this simple grammar, it is not immediately obvious what are FIRST(E) and FIRST(T).

So we need an algorithm that we can describe in programming terms, rather than relying on our human pattern-matching abilities.

Some FIRST sets are fairly obvious

SLR Construction - FIRST sets

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

The Dragon Book tells us how to compute FIRST(X)...

FOLLOW(...)

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is terminal, then FIRST(X) is $\{X\}$. Got it. ✓
2. If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X). Got it. ✓
3. If X is nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), ..., FIRST(Y_{i-1}); that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$) and so on. YIKES!!

Now, we can compute FIRST for any string $X_1 X_2 \cdots X_n$ as follows. Add to FIRST($X_1 X_2 \cdots X_n$) all the non-ϵ symbols of FIRST(X_1). Also add the non-ϵ symbols of FIRST(X_2) if ϵ is in FIRST(X_1), the non-ϵ symbols of FIRST(X_3) if ϵ is in both FIRST(X_1) and FIRST(X_2), and so on. Finally, add ϵ to FIRST($X_1 X_2 \cdots X_n$) if, for all i, FIRST(X_i) contains ϵ.

Oh brother!! Seriously? More of this notation to dissect. Let's dig in.

Rule #1 we've already covered.
Rule #2 is easy enough as well, but this sample grammar doesn't have and $X \rightarrow \epsilon$ productions.
But Rule #3 will require some attention

How to compute FIRST(X), presented in typical Dragon Book formalism. What are they really trying to say?

SLR Construction - FIRST sets

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

Let's take a discrete case before tackling it for an arbitrarily large production.

Imagine a production: $A \rightarrow WXYZ$

Since W is the first symbol, FIRST(A) must obviously contain everything in FIRST(W)

But what if FIRST(W) contains ε? Then the FIRST(A) could "flow through" FIRST(W) and pick up everything in FIRST(X).

But what if FIRST(X) also contains ε? Then FIRST(A) can flow through W and X and pick up everything in FIRST(Y).

Etc., for all symbols in the production

Deciphering Rule #3 for FIRST(X)

SLR Construction - FIRST sets

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

So if we generalize this idea to an arbitrarily large production

$$X \rightarrow Y_1 Y_2 Y_3 \cdots Y_n$$

then that's all the Dragon Book is trying to tell us. Start at the left edge and proceed along each symbol

$$X \rightarrow Y_1 Y_2 Y_3 \cdots Y_n$$

Move from left-to-right through the production

Add FIRST(Y_1) to FIRST(X)

If FIRST(Y_1) contains ε, then add FIRST(Y_2) to FIRST(X)

Etc. Repeat the process for each symbol in the production

Stop when you find the first FIRST(Y_i) that does not contain ε

Turning Dragon Book gibberish into something that actually makes sense

SLR Construction - FIRST sets

Grammar

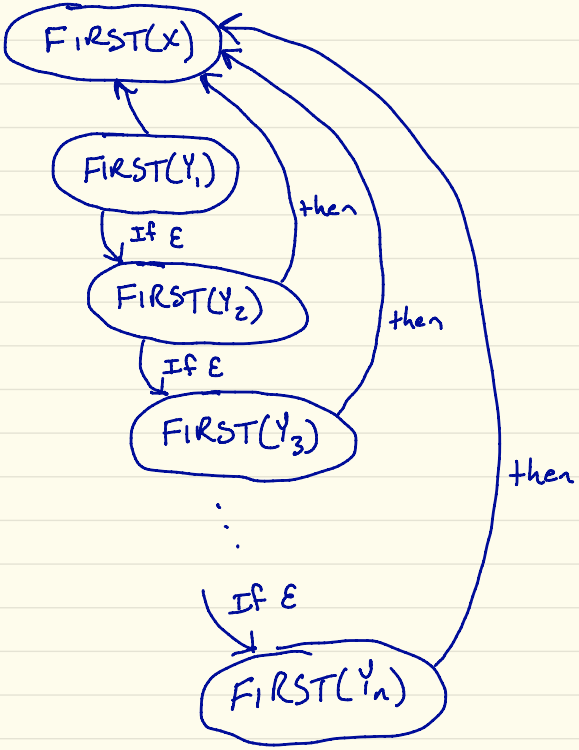
0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

We can visualize this idea as a dependency graph

$X \rightarrow Y_1 Y_2 Y_3 \cdots Y_n$



FIRST sets shown as a dependency tree for an arbitrarily large production

SLR Construction - FIRST sets

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

So let's work through it ...

$E' \rightarrow E$ tells us FIRST(E') contains all of FIRST(E)

$E \rightarrow E + T$ tells us nothing new. Obviously FIRST(E) contains all of FIRST(E)

$E \rightarrow T$ tells us FIRST(E) contains everything in FIRST(T)

$T \rightarrow T * F$ tells us nothing new.

$T \rightarrow F$ tells us FIRST(T) contains everything in FIRST(F)

$F \rightarrow (E)$ tells us FIRST(F) contains '('.

$F \rightarrow id$ tells us FIRST(F) contains id

Computing the FIRST sets for our sample grammar

SLR Construction – FIRST sets

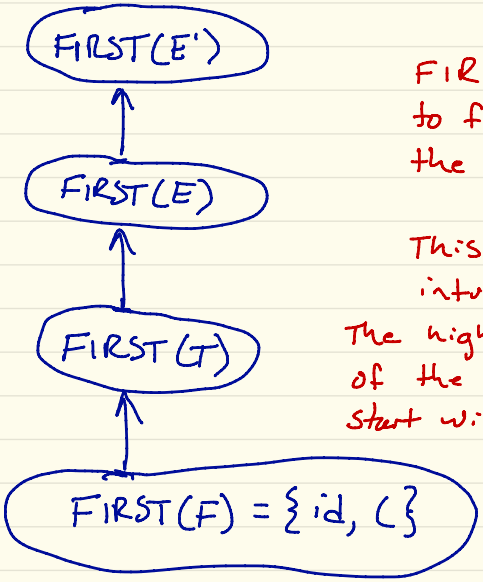
Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

We can represent these dependencies as a graph



FIRST sets tend to flow "up" in the grammar

This makes intuitive sense. The higher level constructs of the grammar could start with a lot of different symbols

So for this simple grammar, it turns out that

$FIRST(E') = FIRST(E) = FIRST(T) = FIRST(F) = \{ id, (\}$

Now that we've computed the FIRST sets, we can move on to the FOLLOW sets.

SLR Construction - Computing FOLLOW sets

Grammar

0. $E' \to E$
1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

0 = Augmented Production

1-6 = Productions from target grammar

The FOLLOW set is the set of all terminals that can appear immediately after a nonterminal

In other words,

FOLLOW(A) is the set of all terminals, a, such that a derivation $S \overset{*}{\Rightarrow} \alpha A a \beta$ exists.

By now we can anticipate what our beloved Dragon Book will tell us:

1. Place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker. Okay, that makes sense
2. If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except for ϵ is placed in FOLLOW(B).
3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$ where FIRST(β) contains ϵ (i.e., $\beta \overset{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B).

That's not too bad. We can figure this out. Let's dig in.

Introduction to FOLLOW sets

SLR Construction – FOLLOW Sets Rule #2

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

Consider a production

$$A \rightarrow \alpha B \beta$$

Whatever β can start with ...

Means it must be able to come after B

So FOLLOW(B) contains (almost) everything in FIRST(β), except we never put ε in the FOLLOW sets because they would have no purpose or meaning.

The third rule is a bit trickier, so we'll look at that one next.

Examining Rule #2 for constructing FOLLOW sets

SLR Construction - FOLLOW Sets Rule #3

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

For FOLLOW rule #3, consider a production $A \rightarrow \alpha B$, and also any productions where B could be followed by something that can contain ε ($A \rightarrow \alpha B\beta$ where FIRST(β) contains ε)

Since B is the last thing in the production, it means that the FOLLOW(A) can transfer to the FOLLOW(B)

$A \rightarrow \alpha B$

Anything in FOLLOW(A) — transfers to FOLLOW(B)

or when B is followed by something that can contain ε in FIRST

$A \rightarrow \alpha B\beta$ where FIRST(β) contains ε

Anything in FOLLOW(A) — transfers to FOLLOW(B)

Examining Rule #3 for constructing FOLLOW sets

SLR Construction - FOLLOW sets Rule #3

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

This seems a bit counter-intuitive.

When we constructed the FIRST sets it was the right-hand-side that transferred to the LHS

FIRST Construction ... when

$A \rightarrow WXYZ$

the FIRST(W) transfers to FIRST(A)

Right-To-Left Transfer of FIRST sets

But this phenomenon is the reverse in the FOLLOW construction

$A \rightarrow \alpha B$

FOLLOW(A) transfers to FOLLOW(B)

Why not the reverse direction?

Why does this work?

Left-to-Right Transfer of FOLLOW sets

Noticing how the "flow" of the FOLLOW set construction differs from construction of FIRST sets

SLR Construction - Follow Set Rule #3

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

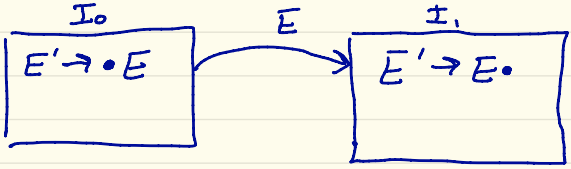
0 = Augmented Production

1-6 = Productions from target grammar

Let's look an example to see why this is the case

From Rule #1 we know FOLLOW(E') contains {$}

And we know State 0 & 1 from the DFA looks like



Now if we're in State I_1 and we see $ as the next input symbol, then we can accept the string.

Therefore it must mean that E can be followed by a $

So it is the FOLLOW set of the LHS that transfers to the RHS

More deeply examining why Rule #3 of FOLLOW set construction works

SLR Construction - FOLLOW Set Rule #3

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

The reverse would not work.

Suppose we thought that because

$$E' \rightarrow E$$

that FOLLOW(E') should contain FOLLOW(E)

But from the production

$$E \rightarrow E + T$$

We know that FOLLOW(E) contains {+}

But once we've reduced all the way back to $E' \rightarrow E$, we can only expect $ next.

Therefore FOLLOW(E') cannot contain {+}

So it is the LHS that transfers to the RHS, not the other way around

Examining why the "flow" of FOLLOW sets construction does not go in the other direction

SLR Construction - Building Our FOLLOW set

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

$E' \rightarrow E$ tells us
FOLLOW(E) contains FOLLOW(E')

$E \rightarrow E+T$ tells us
FOLLOW(T) contains FOLLOW(E)
-and- FOLLOW(E) contains {+}

$E \rightarrow T$ tells us nothing new

$T \rightarrow T*F$ tells us
FOLLOW(F) contains FOLLOW(T)
-and- FOLLOW(T) contains {*}

$T \rightarrow F$ tells us nothing new

$F \rightarrow (E)$ tells us
FOLLOW(E) contains ')'

$F \rightarrow id$ tells us nothing new

Also, from Rule #1 we know

FOLLOW(E') contains $

Creating the FOLLOW sets from our sample grammar by inspecting each production

SLR Construction - The FOLLOW set Shown Graphically

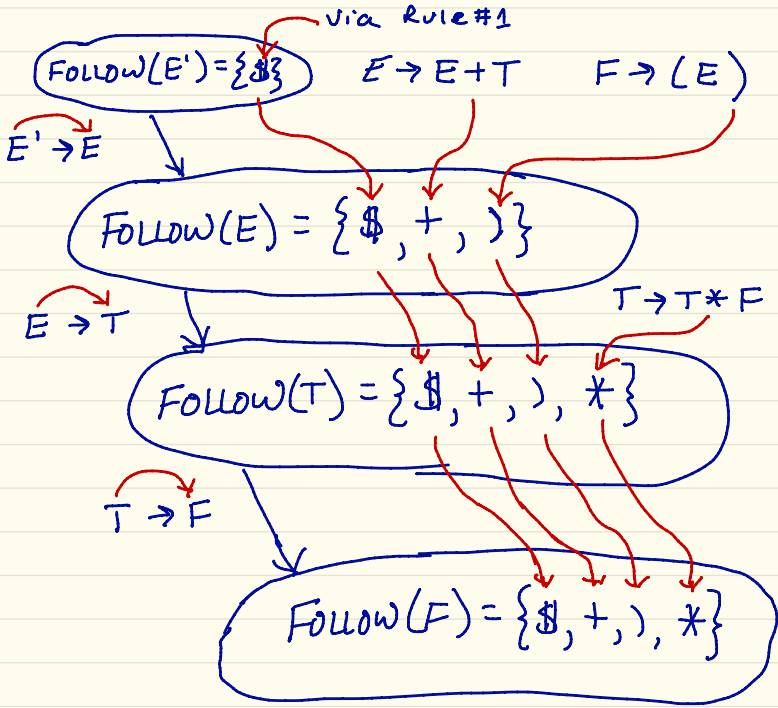
Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

We can represent the FOLLOW set construction as a dependency graph



FOLLOW sets tend to flow "down" the grammar, whereas FIRST sets flow "up"

Depicting the FOLLOW sets for our sample grammar in a graphical form. Very useful to visualize the "flow" of the FOLLOW set construction.

SLR Construction - Building the Parse Table

The action table consisting of the "shifts" comes directly from the DFA

Now we can finally produce the parsing table we are after. This part is actually quite straightforward.

A state's goto() function for terminals always means shift and proceed to the designated state. We write it as "sj" where j is the next state number

STATE	id	+	*	(	)	$
0	s5			s4		
1	s5					
2			s7			
3						
4	s5			s4		
5						
6	s5			s4		
7	s5			s4		
8		s6			s11	
9			s7			
10						
11						

Example: If the next input symbol is id, shift it onto the stack and proceed to State 5.

Filling in the actions table with shifts. Any goto for a terminal in a given state results in a shift.

SLR Construction - Reductions in the Parsing Table

Grammar

0. $E' \to E$
1. $E \to E + T$
2. $E \to T$ r2
3. $T \to T * F$
4. $T \to F$ r4
5. $F \to (E)$
6. $F \to id$

0 = Augmented Production

1-6 = Productions from target grammar

FOLLOW(E') = {$}
FOLLOW(E) = {$, +,)}
FOLLOW(T) = {$, +,), *}
FOLLOW(F) = {$, +,), *}

Now for the reductions. If we have an item in our set that looks like $A \to \alpha \bullet$, then that means we're ready to do a reduction if the next input symbol is in FOLLOW(A). Otherwise, we have an error. ← Except for E'.

I_0: No items with $A \to \alpha \bullet$, so no reductions to do

$I_1 = E' \to E \bullet$, $E \to E \bullet + T$ ⇒ {$} = accept

$I_2 = E \to T \bullet$, $T \to T \bullet * F$ ⇒ {$, +,)} all set to r2 Reduce by $E \to T$

$I_3 = T \to F \bullet$ ⇒ {$, +,), *} all set to r4 Reduce by $T \to F$

We label reductions with "rj" where j = the production number

Showing how we arrive at the reduction actions for states I_0, I_1, I_2, and I_3

SLR Construction - Reductions in the Parsing Table

Grammar

0. $E' \to E$
1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$ r6

0 = Augmented Production

1-6 = Productions from target grammar

$Follow(E') = \$$
$Follow(E) = \$, +,)$
$Follow(T) = \$, +,), *$
$Follow(F) = \$, +,), *$

$I_4 = F \to (\bullet E)$
$E \to \bullet E + T$
$E \to \bullet T$
$T \to \bullet T * F$
$T \to \bullet F$
$F \to \bullet (E)$
$F \to \bullet id$

No dots at the end of any items

No reductions to do

$I_5 = F \to id \bullet \Rightarrow \$, +,), *$ all set to r6

Reduce by $F \to id$

$I_6 = E \to E + \bullet T$
$T \to \bullet T * F$
$T \to \bullet F$
$F \to \bullet (E)$
$F \to \bullet id$

No dots at the end of any items

No reductions to do

Continue computing reduction actions for I_4, I_5, and I_6

SLR Construction - Reductions in the Parsing Table

Grammar

0. $E' \to E$
1. $E \to E + T$ r1
2. $E \to T$
3. $T \to T * F$ r3
4. $T \to F$
5. $F \to (E)$ r5
6. $F \to id$

0 = Augmented Production

1-6 = Productions from target grammar

FOLLOW(E') = $
FOLLOW(E) = $, +,)
FOLLOW(T) = $, +,), *
FOLLOW(F) = $, +,), *

$I_7 = T \to T * \bullet F$
$F \to \bullet (E)$
$F \to \bullet id$
No reductions to do

$I_8 = F \to (E \bullet)$
$E \to E \bullet + T$
No reductions to do

$I_9 = E \to E + T \bullet$ $\Rightarrow$ $, +,) all set to reduce by $E \to E + T$
$T \to T \bullet * F$

$I_{10} = T \to T * F \bullet$ $\Rightarrow$ $, +,), * all set to r3 Reduce by $T \to T * F$

$I_{11} = F \to (E) \bullet$ $\Rightarrow$ $, +,), * all set to r5 Reduce by $F \to (E)$

We finish computing the reduction actions by completing I_7, I_8, I_9, I_{10}, and I_{11}

SLR Construction - Gotos of the Parsing Table

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

FOLLOW(E') = $

FOLLOW(E) = $, +,)

FOLLOW(T) = $, +,), *

FOLLOW(F) = $, +,), *

To compute the gotos for the nonterminals of the parsing table, we take them directly from the DFA.

STATE	E	T	F
0	1	2	3
1			
2			
3			
4	8	2	3
5			
6		9	3
7			10
8			
9			
10			

The Gotos section of the parsing table

SLR Construction - Final Parsing Table

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

FOLLOW(E') = {$}
FOLLOW(E) = {$, +,)}
FOLLOW(T) = {$, +,), *}
FOLLOW(F) = {$, +,), *}

We've completed the parsing table! Let's collate the work we've done over the last few pages to see the full and final table (taken directly from the Dragon Book)

STATE	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

The final parsing table showing all shifts, reduces, and gotos

SLR Construction - Sample Walkthrough

Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

1-6 = Productions from target grammar

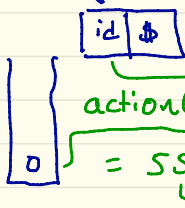
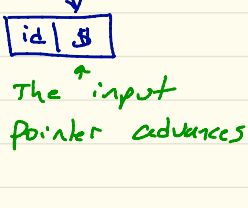
Input: id$ $ = EOF Marker

STEP 1

BEFORE	AFTER
	

Initialize the parser by pushing state 0 onto the stack

STEP 2

BEFORE	AFTER
	

Look up id in State 0. Shift id and State 5 onto the stack

Showing the parsing steps of the simplest possible valid input string: id$. State 0 shifts id onto the stack.

SLR Construction - Sample Walkthrough

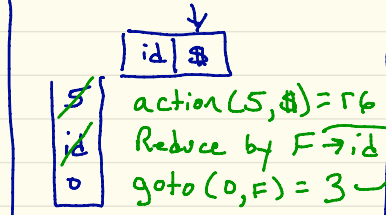
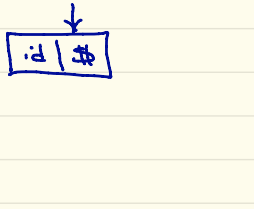
Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

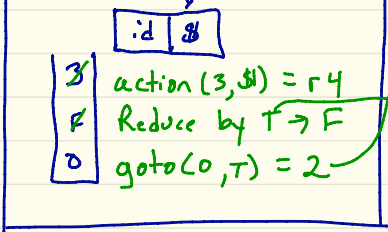
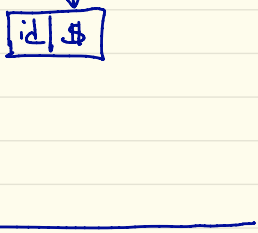
1-6 = Productions from target grammar

STEP 3

BEFORE	AFTER
	

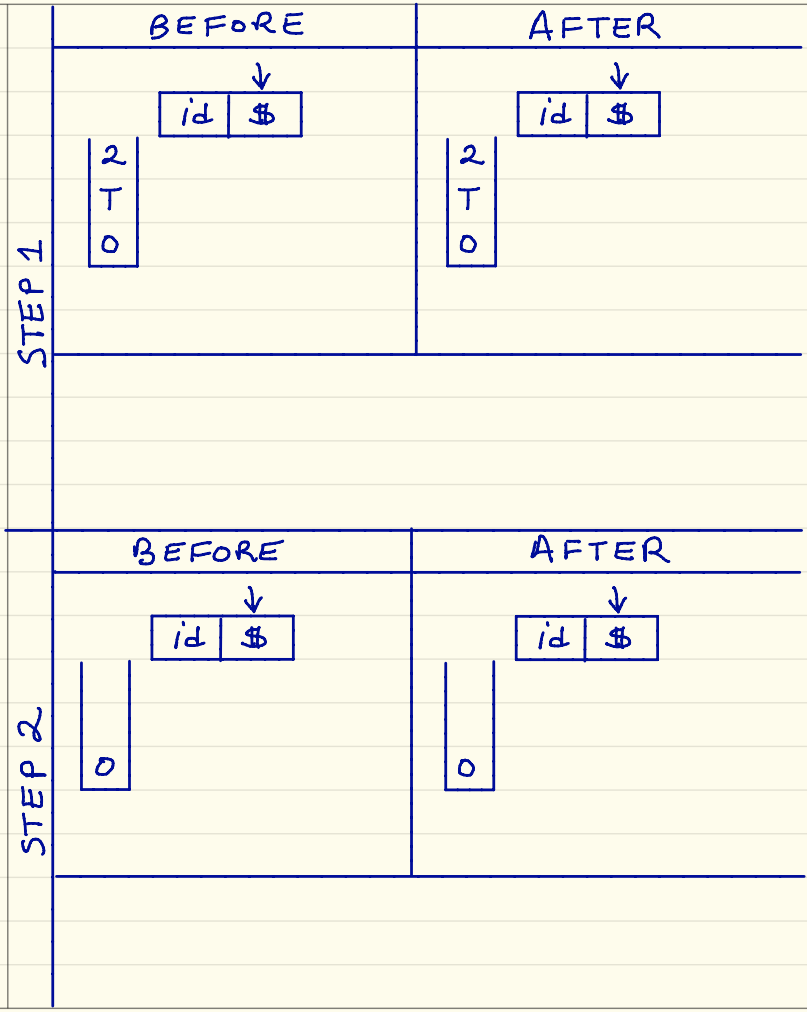
State 5 tells us to reduce by production 6 ($F \rightarrow id$). Since the production contains 1 symbol (id) we pop $2 * 1 = 2$ items off the stack, exposing state 0. We look up goto(0, F), giving us state 3. We push F and state 3 onto the stack

STEP 4

BEFORE	AFTER
	

Same basic sequence as Step 2. The reduction is $T \rightarrow F$. Next state is 2. 2 & T replace the stack.

State 5 reduces by $F \rightarrow id$. State 3 reduces by $T \rightarrow F$.


STEP 1
BEFORE
AFTER
id
$
2
T
0
STEP 2
BEFORE
AFTER
id
$
0

SLR Construction - Sample Walkthrough

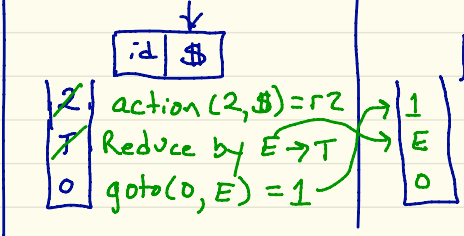
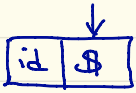
Grammar

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

0 = Augmented Production

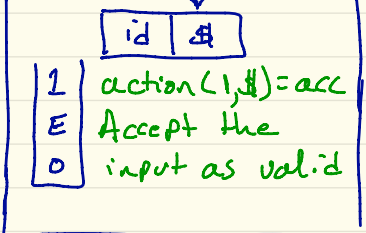
1-6 = Productions from target grammar

STEP 5

BEFORE	AFTER
	

Similar to previous step, the reduction is now $E \rightarrow T$. 2 and T are popped off the stack, replace by E and 1

STEP 6

BEFORE	AFTER
	

We reach State 1 with the next input symbol = $. The action table tells us to accept the string as valid

State 2 reduces by $E \rightarrow T$. State 1 accepts with next input symbol = $.

Part 3

LR(1) Construction

LR(1) Construction - Problems with SLR

SLR Problems	Sadly, SLR parsers are not powerful enough to recogize grammars as complex as we'd like (Modern programming languages, for example) They produce too many shift-reduce and reduce-reduce conflicts. They don't use enough contextual information about the grammar to make the best possible shift/reduce decisions. For example, the Dragon Books walks through a sample grammar for which the SLR table calls for reduction in a particular state, even though there is no derivation for that configuration based on the next input symbol. The solution is a more powerful parser construction mechanism - LR(1)

Why SLR parsers are not powerful enough and therefore we need something better

LR(1) Construction - LR(1) Issues

LR(1) Problems

LR(1) parsers, however, have their own problems. Namely, they produce parsing tables that have a huge number of states (on the order of Millions for a modern grammar).

But we explore LR(1) parsers anyway, because they lay down some foundational concepts that we need for LALR(1) parsers.

The key aspect of LR(1) parsers is that the states contain more contextual information about the grammar. Specifically, the "items" in the set contain information about the next input symbol.

We'll look at this on the next page.

Why we study LR(1) even though they produce too many states

LR(1) Construction - Introducing LR(1) items

LR(0) vs LR(1) items

SLR Parsers use "LR(0)" items in their sets, which as we've seen previously, have the form:

$$[A \rightarrow \alpha \bullet X\beta]$$

Each item gives us two pieces of information: 1) The production and 2) The current position in the production sequence.

LR(1) parsers use "LR(1)" items in their sets. They have the form:

$$[A \rightarrow \alpha \bullet X\beta, a]$$

Where a represents the next input symbol, called the lookahead token.

Thus, LR(1) items give us 3 pieces of information: 1) Production, 2) Position, 3) Lookahead token

Examing LR(0) vs. LR(1) items

LR(1) Construction - Introducing Lookahead Sets

Lookahead Sets

It's often the case with LR(1) item sets that we have multiple items where the only thing that differs is the lookahead token.

To save space and improve cognitive reasoning, we collapse the like items as follows

Instead of (where α, β, and β are all the same)

$$\begin{aligned} &A \to \alpha \cdot B\beta,\ a_1 \\ &A \to \alpha \cdot B\beta,\ a_2 \\ &\vdots \\ &A \to \alpha \cdot B\beta,\ a_n \end{aligned}$$

We write it as a single item

$$A \to \alpha \cdot B\beta,\ \boxed{a_1/a_2/\ldots/a_n}$$

lookahead set

When done in this manner, the collection of lookahead symbols is called the "lookahead set"

Using lookahead sets as a shorthand notation

LR(1) Construction - Sample Grammar

A new grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$
2. $H \rightarrow cH$
3. $H \rightarrow d$

LR(1) parsers work similar to SLR parsers but the closure and goto procedures are modified to incorporate the next input symbol.

It's easier to walk through a sample grammar rather than wade through the Dragon Book's notation.

But the previous grammar we used for SLR construction is too big to use for a manual LR(1) procedure, so we'll use a new, simpler grammar.

$S' \rightarrow S$
$S \rightarrow H$
$H \rightarrow cH$
$H \rightarrow d$

NonTerminals: S', S, H

Terminals: c, d

Introducing a new sample grammar to explore LR(1) construction

LR(1) Construction - Starting the Sets-of-Items

Grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$
2. $H \rightarrow cH$
3. $H \rightarrow d$

Similar to SLR construction, we start the Sets-of-Items construction by computing

$$I_0 = \text{closure}(\{S' \rightarrow \bullet S, \$\})$$

That's because we should expect to make the final $S' \rightarrow S$ reduction only when the next input symbol is $\$$ (i.e., EOF)

But the closure operation itself is a little different than SLR. I'll point out the differences as the example unfolds.

But before we can do anything, we need to compute the FIRST sets, so we'll do that next.

The Sets-of-Items Construction starts out similar to SLR, but incorporates the lookahead token

LR(1) Construction - A Little Bit of Theory

LR(1) Theory

Because LR(1) starts with the item $S' \rightarrow S, \$$, which includes lookahead information, we get a taste for what is going to happen.

With LR(1), the lookahead contextual information is interwoven into the very fabric of the DFA.

That's not the case with SLR. In SLR, we don't consider the next input symbol until we create the parsing table reduction actions at the very end of the process when we find an item

$$A \rightarrow \alpha \bullet$$

We only then consider the next input by adding reductions only for FOLLOW(A)

How LR(1) avoids conflicts by incorporating lookahead tokens into the fabric of the DFA

LR(1) Construction - A Little Bit of Theory

LR(1) Theory

But by then it is too late

It's not that it doesn't work, it's just that it creates too many conflicts.

So with LR(1), we are trying to avoid conflicts that happen at the end of the process by incorporating the lookahead context from the very beginning, and all throughout the process.

That is the basic difference between SLR and LR(1)

Examining the core difference between SLR and LR(1)

FIRST sets

FIRST(H) = {c, d}

LR(1) Construction - Closures

Grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$
2. $H \rightarrow cH$
3. $H \rightarrow d$

$I_0 = S' \rightarrow \bullet S, \$$ ← seeded item

SLR Construction would have us add $S \rightarrow \bullet HH$ to I_0.

In LR(1), we represent the current item of interest with the pattern

$$A \rightarrow \alpha \bullet B\beta, a$$

$$S' \rightarrow \varepsilon \bullet S \varepsilon, \$$$

So $A = S'$, $\alpha = \varepsilon$, $B = S$, $\beta = \varepsilon$, $a = \$$

and we add new items to the set as

$$B \rightarrow \bullet\gamma, b$$

for any b in $FIRST(\beta a)$

In this case $\beta a = \$$ and thus $FIRST(\beta a) = \$$

Introducing LR(1) closures and how they differ from LR(0) closures

LR(1) Construction - Closures

Grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$
2. $H \rightarrow cH$
3. $H \rightarrow d$

So we add

$$S \rightarrow \bullet HH, \$$$

to the closure set, giving us (so far)

$$I_0 = S' \rightarrow \bullet S, \$$$
$$S \rightarrow \bullet HH, \$$$

Now we continue the closure by operating on $S \rightarrow \bullet HH, \$$

Back to the pattern $A \rightarrow \alpha \bullet B\beta, a$

$A = S$ $\quad \beta a = H\$$
$\alpha = \varepsilon$
$B = H$ $\quad$ FIRST($H\$$) = FIRST(H)
$\beta = H$ $\quad$ since H does not derive ε
$a = \$$

FIRST(H) = $\{c, d\}$

Working through the closure (I_0). Notice how the lookahead sets of the 2 H items transitions from the original $\{\$\}$ to $\{c, d\}$.

LR(1) Construction - Completing Closure(I_0)

Grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$
2. $H \rightarrow cH$
3. $H \rightarrow d$

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d

Since we have 2 H productions and two items in FIRST(H$), we add 4 new items to I_0, leaving us with:

I_0 =		Need Gotos For	
	$S' \rightarrow \bullet S, \$$	- S	- I_1
	$S \rightarrow \bullet HH, \$$	- H	- I_2
	$H \rightarrow \bullet cH, c/d$	- c	- I_3
	$H \rightarrow \bullet d, c/d$	- d	- I_4

Using lookahead set shorthand

There are no more dots preceding any nonterminals, so we are done.

gotos are computed similar to SLR construction except we carry along the lookahead symbol.

I_1 = goto(I_0, S) = $S' \rightarrow S \bullet, \$$

No dots before nonterminals so we are done with I_1

Showing the full closure of I_0 and identifying the gotos. Items with more than one lookahead symbol is collapsed using shorthand notation.

LR(1) Construction - Understanding I_0

Grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$
2. $H \rightarrow cH$
3. $H \rightarrow d$

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d

$S' \rightarrow \cdot S, \$$

↳ "I might see an S followed by a $"

$S \rightarrow \cdot HH, \$$

↳ "Or else I might see an HH followed by a $"

$H \rightarrow \cdot cH, c/d$

↳ "Or else I might see a cH followed by a c or d"

$H \rightarrow \cdot d, c/d$

↳ "Or else I might see a d followed by a c or d"

What I_0 is really telling us

Examining the information contained in I_0 from the perspective of a human's intuitive cognition.

LR(1) Construction - Understanding Lookaheads Better

Grammar

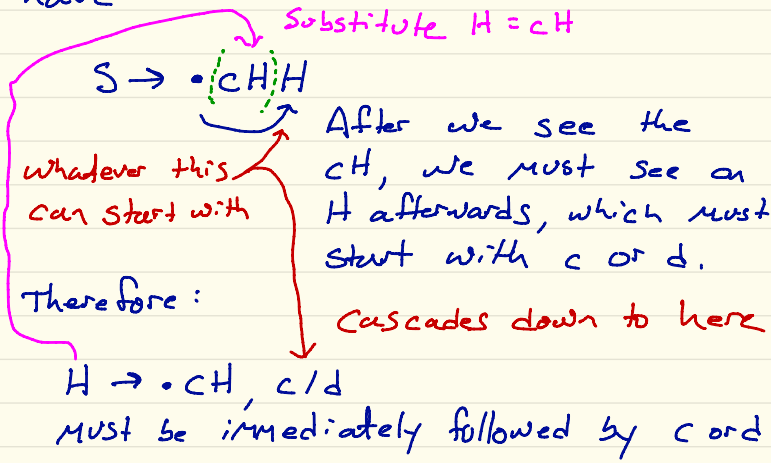
0. $S' \rightarrow S$
1. $S \rightarrow HH$
2. $H \rightarrow cH$
3. $H \rightarrow d$

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d

But from $H \rightarrow \cdot cH$, c/d how do we know that the cH must be followed by a c or d?

Because the prior item $S \rightarrow \cdot HH$, \$ is responsible for the presence of $H \rightarrow \cdot cH$.

If we were to substitute H into the first item, then we would have



We examine how the information in one item cascades down to the other items to get a better intuitive understanding of how the lookaheads are computed

LR(1) Construction - Constructing I_2

Grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$
2. $H \rightarrow cH$
3. $H \rightarrow d$

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d

I_2 = goto(I_0, H)

= closure({ $S \rightarrow H \cdot H, \$$ })

= $S \rightarrow H \cdot H, \$$ Seeded item

$\beta = \varepsilon$

$A \rightarrow \alpha \cdot B\beta, a$

$A = S$
$\alpha = H$
$B = H$
$\beta = \varepsilon$
$a = \$$

FIRST($\varepsilon\$$) = $\$$

So we add the H productions with $ as the lookahead

I_2 = $S \rightarrow H \cdot H, \$$
$H \rightarrow \cdot cH, \$$
$H \rightarrow \cdot d, \$$

No more dots before any nonterminals, so we are done

Showing how we use the item pattern match and FIRST function to complete the closure for I_2

LR(1) Construction - Constructing I_3

Grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$
2. $H \rightarrow cH$
3. $H \rightarrow d$

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d

$I_3 = goto(I_0, c)$

$= closure(\{ H \rightarrow c \cdot H, c/d \})$

$I_3 = H \rightarrow c \cdot H, \overset{\varepsilon}{c}\ d$ ← seeded item

$A \rightarrow \alpha \cdot B\beta, a$ ($\beta = \varepsilon$)

$A = H$ $\quad a = \{c, d\}$
$\alpha = c$
$B = H$ $\quad FIRST(\varepsilon c) = c$
$\beta = \varepsilon$ $\quad FIRST(\varepsilon d) = d$

Add H productions with c/d lookaheads

$H \rightarrow \cdot cH, c/d$ — No more dots before nonterminals
$H \rightarrow \cdot d, c/d$

$I_3 = H \rightarrow c \cdot H, c/d$
$H \rightarrow \cdot cH, c/d$
$H \rightarrow \cdot d, c/d$

Showing how the closure(I_3) works with the lookaheads.

LR(1) Construction – Constructing I_4

Grammar

0. $S' \to S$
1. $S \to HH$
2. $H \to cH$
3. $H \to d$

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d

$I_4 = goto(I_0, d)$

$= closore(\{H \to d\bullet, c/d\})$

$I_4 = H \to d\bullet, c/d$ ← seeded item

No more dots before nonterminals so we are done

This completes the gotos of I_0. Next, we'll move on to examining the newly created sets I_1, I_2, I_3, and I_4 for gotos.

Continuing with the gotos of I_0 by constructing $I_4 = goto(I_0, d)$

LR(1) Construction – Gotos of I_2: I_5, I_6, I_7

Grammar

0. $S' \to S$
1. $S \to HH$
2. $H \to cH$
3. $H \to d$

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d

We are done with the gotos of I_0.

I_1 has 1 item and needs no gotos.

Returning to I_2 to inspect it for gotos

I_2 = $S \to H \bullet H, \$$
$H \to \bullet cH, \$$
$H \to \bullet d, \$$

Need Gotos For
- H
- c
- d

Let's make

$I_5 = goto(I_2, H) = S \to HH \bullet, \$$

$I_6 = goto(I_2, c) = closure(H \to c \bullet H, \$)$

$I_7 = goto(I_2, d) = closure(H \to d \bullet, \$)$

$= H \to d \bullet \$$ (no items to add)

Continuing with the construction of LR(1) states by computing the gotos of I_2.

LR(1) Construction - Constructing I_6

Grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$
2. $H \rightarrow cH$
3. $H \rightarrow d$

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d

I_6 = closure({ $H \rightarrow c \cdot H, \$$ })

$I_6 = H \rightarrow c \cdot H, \$$ ← Seeded item

$A \rightarrow \alpha \cdot B\beta, a$

$A = H$ FIRST($\varepsilon\$$) = $\$$
$\alpha = c$
$B = H$ Add H productions
$\beta = \varepsilon$ with $\$$ lookahead
$a = \$$

So add ... $H \rightarrow \cdot cH, \$$
$H \rightarrow \cdot d, \$$

$I_6 = H \rightarrow c \cdot H, \$$
$H \rightarrow \cdot cH, \$$
$H \rightarrow \cdot d, \$$

This completes to gotos of I_2

Working on the gotos of I_2 by constructing
$I_6 = goto(I_2, c)$

LR(1) Construction - Gotos of I_3

Grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$
2. $H \rightarrow cH$
3. $H \rightarrow d$

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d

Now for the gotos of I_3

		Need Gotos For
I_3 =	$H \rightarrow c \cdot H,\ c/d$	H
	$H \rightarrow \cdot cH,\ c/d$	c
	$H \rightarrow \cdot d,\ c/d$	d

I_8 = goto(I_3, H)

goto(I_3, c) = closure($H \rightarrow c \cdot H, c/d$) = I_3
→ duplicate

goto(I_3, d) = closure($H \rightarrow d \cdot, c/d$) = I_4
→ duplicate

I_8 is the only new state created by gotos of I_3. The other two states are duplicates (I_3 & I_4).

LR(1) Construction - Constructing $I_8 = \text{goto}(I_3, H)$

Grammar

0. $S' \to S$
1. $S \to HH$
2. $H \to cH$
3. $H \to d$

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d

$I_8 = \text{goto}(I_3, H)$

$= \text{closure}(\{H \to cH\bullet, c/d\})$

$I_8 = H \to cH\bullet, c/d$ ← Seeded item

No more items to add.

This completes the gotos of I_3.

Next we examine the newly created states I_5, I_6, I_7, and I_8 for gotos. We see that 3 of the states - I_5, I_7, and I_8 have only one item each, all with dots at the end. Therefore, they need no gotos.

I_6 needs a goto for only 1 item - $H \to c\bullet H, \$$. Therefore we shall make $I_9 = \text{goto}(I_6, H)$.

Constructing I_8 to complete the gotos of I_3.

LR(1) Construction - Constructing I_9

Grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$
2. $H \rightarrow cH$
3. $H \rightarrow d$

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d

I_9 = goto(I_6, H)

= closure({ $H \rightarrow cH \bullet$, \$ })

I_9 = $H \rightarrow cH \bullet$, \$ ← Seeded item

No more dots before any nonterminals, so we are done

I_9 needs no gotos, which means we have now completed the full Sets-Of-Items construction for the grammar.

Completing the Sets-Of-Items construction by constructing I_9 = goto(I_6, H).

LR(1) Construction - Summary of States

Grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$
2. $H \rightarrow cH$
3. $H \rightarrow d$

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d

Summary of States

I_0 = closure($\{S' \rightarrow \bullet S, \#\}$)
I_1 = goto(I_0, S)
I_2 = goto(I_0, H)
I_3 = goto(I_0, c) = goto(I_3, c)
I_4 = goto(I_0, d) = goto(I_3, d)
I_5 = goto(I_2, H)
I_6 = goto(I_2, c) = goto(I_6, c)
I_7 = goto(I_2, d) = goto(I_6, d)
I_8 = goto(I_3, H)
I_9 = goto(I_6, H)

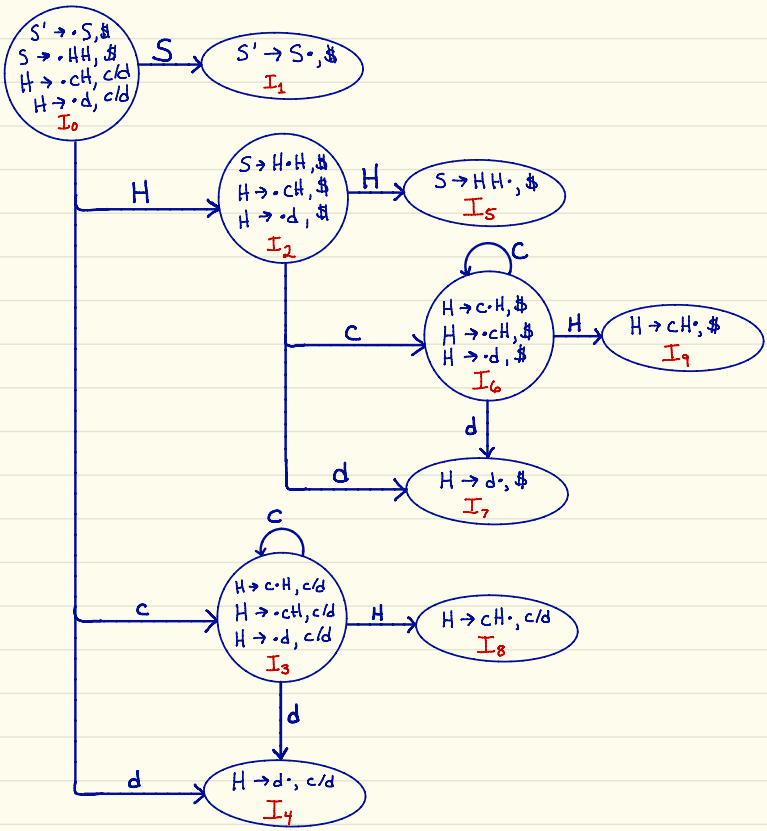
The final list of states for our sample grammar.

LR(1) Construction - Final DFA

Grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$
2. $H \rightarrow cH$
3. $H \rightarrow d$

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d



Same as DFA in Dragon Book except nonterminal labeled "H" instead of "C"

The final DFA of the sample grammar constructed from the Sets-Of-Items result.

LR(1) Construction - Building the Parsing Table

Grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$
2. $H \rightarrow cH$
3. $H \rightarrow d$

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d

Grammar	
0.	$S' \rightarrow S$
1.	$S \rightarrow HH$
2.	$H \rightarrow cH$
3.	$H \rightarrow d$
FIRST Sets	
S'	c, d
S	c, d
H	c, d

Building the table is almost the same as SLR construction. The only difference is in what terminals we assign the reductions to.

When dot is at the end, we add reductions

Whenever we find an item in the set $[A \rightarrow \alpha \bullet, \{lookaheads\}]$, then we add a reduction for that production to any of terminals found in the lookaheads set.

In SLR construction, we would add the reduction for all terminals in FOLLOW(A).

But in LR(1), since we have carefully tracked the possible next symbols all throughout the DFA, we can now take advantage of that information by being more judicious in which terminals we add the reductions to.

That is the essential difference between SLR and LR(1).

How we construct the LR(1) parsing table and how it differs from SLR.

LR(1) Construction - Parsing Table Reductions

Grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$ r1
2. $H \rightarrow cH$ r2
3. $H \rightarrow d$ r3

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d

State I_4:
We find $[H \rightarrow d\cdot, c/d]$ so we add reduction r3 to terminals c & d.

State I_5:
We find $[S \rightarrow HH\cdot, \$]$, so we add reduction r1 to the $ terminal.

State I_7
We find $[H \rightarrow d\cdot, \$]$, so we add reduction r3 to the $ terminal.

State I_8
We find $[H \rightarrow cH\cdot, c/d]$, so we add reduction r2 to the terminals c & d.

State I_9
We find $[H \rightarrow cH\cdot, \$]$, so we add reduction r2 to the $ terminal.

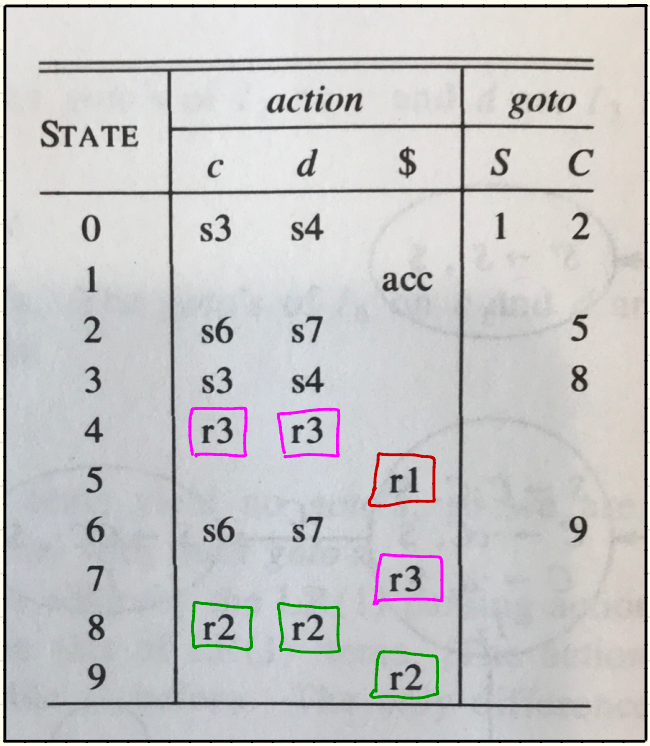
Examing each state for items matching $[A \rightarrow \alpha\cdot, \{a\}]$ and adding the associated reductions.

LR(1) Construction - Final Parsing Table

Grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$ r1
2. $H \rightarrow cH$ r2
3. $H \rightarrow d$ r3

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d



STATE	action			goto	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

The final parsing table taken directly from the Dragon Book.

The LR(1) construction mechanism culminates in the creation of the parsing table, giving us the shifts, gotos, and reductions for each state.

LR(1) Construction - Sample Walkthrough

Grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$
2. $H \rightarrow cH$
3. $H \rightarrow d$

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d

Input = "cccdd$"

Stack	Input	Action	# To Pop	Expose State	Next
0	cccdd$	Shift c			3
0 c 3	ccdd$	Shift c			3
0 c 3 c3	cdd$	Shift c			3
0 c3 c3 c3	dd$	Shift d			4
0 c3 c3 c3 d4	d$	H → d	2	3	8
0 c3 c3 c3 H8	d$	H → cH	4	3	8
0 c3 c3 H8	d$	H → cH	4	3	8
0 c3 H8	d$	H → cH	4	0	2
0 H2	d$	Shift d			7
0 H2 d7	$	H → d	2	2	5
0 H2 H5	$	S → HH	4	0	1
0 S1	$	Accept			

Intuitively

The first c shifts onto the stack and puts us in state 3.

We keep shifting c's onto the stack until we find the first d, which we also shift, landing us in state 4.

Now we reduce by H → d, which puts us into I_8, where we do successive H → cH reductions, one for each c that we originally shifted onto the stack.

Showing the states and actions the parser goes through for a sample input string.

LR(1) Construction - Sample Walkthrough

Grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$
2. $H \rightarrow cH$
3. $H \rightarrow d$

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d

The last $H \rightarrow cH$ reduction from state 8 exposes state 0, thereby taking us into state 2. We've now recognized the first H that state 0's $S \rightarrow HH$ needs to see.

The next d shifts onto the stack, causing the subsequent action to reduce by $H \rightarrow d$, which exposes state 3, taking us into state 5 as a result.

State 5 sees the next input as $, thereby causing the reduction $S \rightarrow HH$.

State 0, now exposed on the stack, is happy to see the parser has now recognized the second H that it is expecting via its $S \rightarrow \bullet HH, \$$ item and moves into state 1 as a result.

Since next input is $, State 1 accepts.

Examining an intuitive understanding of why the parser performs the actions that it does.

LR(1) Construction - What Would a Human Do?

Grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$
2. $H \rightarrow cH$
3. $H \rightarrow d$

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d

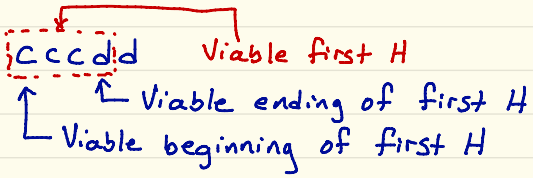
It's interesting to compare the process to a human thought process. How would a human attack the same input string?

A human understanding of the grammar:
- → "I need 2 H's"
- → "An H can start with 0 or more c's"
- → "An H always ends with a d"
- → "Therefore an H starts with a c or d"

Input: "cccdd$"

Step 1: First character is a c, so I know I have a viable first H

Step 2: Scan ahead to find the first d, a viable ending for the first H



Showing one possible way that a human might parse a short input string.

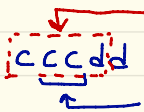
LR(1) Construction - What Would a Human Do?

Grammar

0. $S' \rightarrow S$
1. $S \rightarrow HH$
2. $H \rightarrow cH$
3. $H \rightarrow d$

FIRST(S') = c, d
FIRST(S) = c, d
FIRST(H) = c, d

Step 3: Confirm the middle characters of the viable first H contains only c's



Confirmed first H

Inner characters are c's so I've definitely found the first H

Step 4: Find second H. We see the d as the next character, so we match that as an H.

Step 5: Found two legal H's. Accept.

Human parsing relies on: 1) Intuitive understanding of grammar, 2) Ad hoc parsing mechanism, 3) Parallel processing of input symbols, 4) Backtracking

However these methods break down for large grammars and/or input strings.

For larger grammars, a human would need to resort to more methodical mechanisms, and would likely arrive at something similar to LR parsing.

Part 4

LALR(1) Construction

LALR(1) - Introduction and Motivation

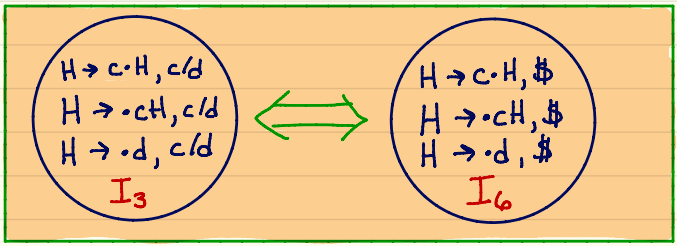
Why LALR(1) is so popular Almost as powerful as LR(1) Same number of states as SLR	LR(1) looks fantastic, but because we are continually looking at the lookahead sets and creating new states for them as-needed, the process creates, shall I be poetic, a "combinatorical explosion" in the number of states, creating far too many states (perhaps millions) to be manageable on 1972-vintage computers. Today's computers, having over a million times more memory than 1972, could probably handle LR(1) if it was really necessary. But why use extra memory if we don't need to? No one would ever do that. Fortunately we have LALR(1), a parser construction algorithm that gives us nearly all the power of LR(1), but only requiring the same number of states as SLR.

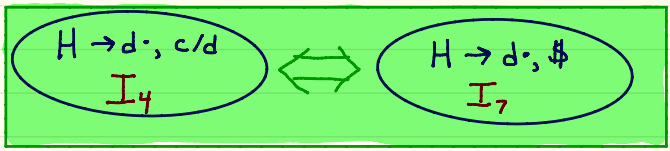
Discussing the memory concerns of LR(1), and introducing an alternative, LALR(1).

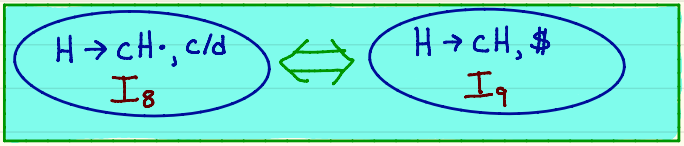
LALR(1) Construction - Identify States with Common Cores

Grammar	
0.	S' → S
1.	S → HH
2.	H → cH
3.	H → d
FIRST Sets	
S'	c, d
S	c, d
H	c, d

If we look closely at the states we created for the LR(1) sample grammar, we see several states that are nearly identical to eachother.







Showing states with common cores that can be merged.

LALR(1) Construction - Merging States

Reducing # of states by Merging

These states are said to have "common cores", because they only differ in their item's lookahead set. The dotted production portion of the items match between the sets.

Since the lookahead sets of an item only affect our reduction decisions, we can simply merge these similar states together. This greatly reduces the number of states needed and is the core essence of LALR(1) construction.

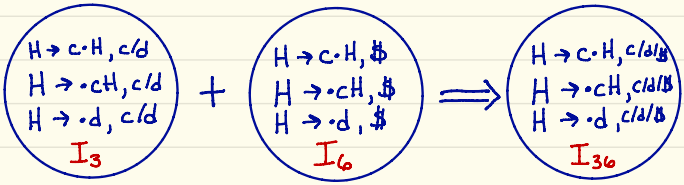
We then replace all of the goto references in the DFA by their merged equivalents.

States from LR(1) construction with common cores can be merged, making the number of states manageable again.

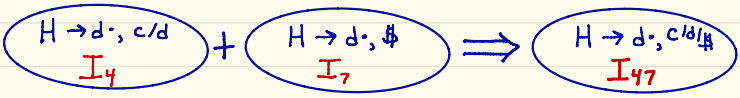
LALR(1) Construction - Merging States

Grammar	
0.	S' → S
1.	S → HH
2.	H → cH
3.	H → d
FIRST Sets	
S'	c, d
S	c, d
H	c, d

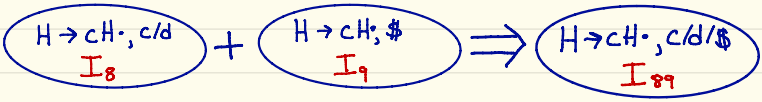
Merging States 3 & 6



Merging States 4 & 7



Merging States 8 & 9



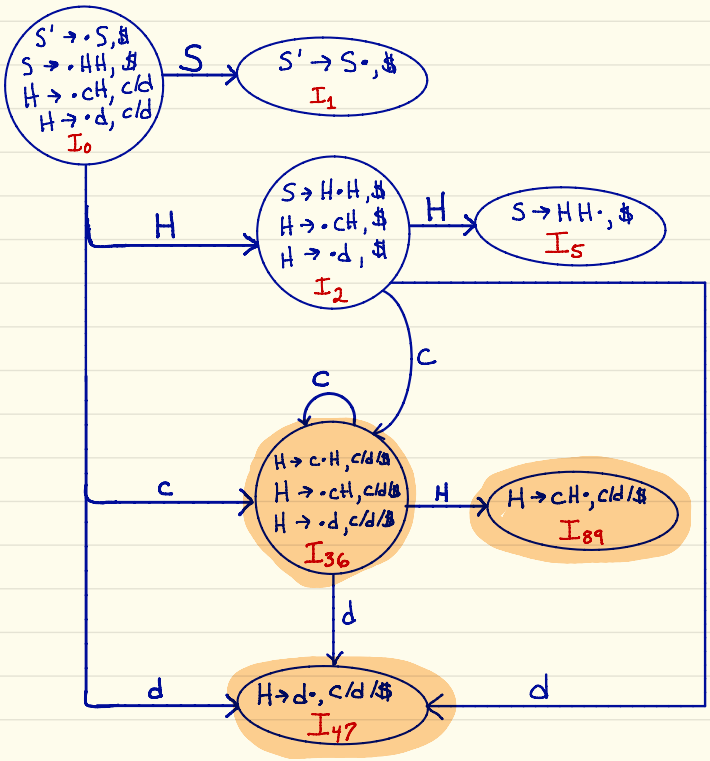
Showing how the lookaheads of the merged states take on the union of the lookaheads of the original states.

We merge states by unioning the lookahead sets.

LALR(1) Construction - Merging States

Grammar	
0.	S' → S
1.	S → HH
2.	H → cH
3.	H → d
FIRST Sets	
S'	c, d
S	c, d
H	c, d

New DFA with Merged States



Gotos from original DFA replaced to point to the merged states.

New DFA using merged states.

LALR(1) Construction - Parsing Table

Grammar	
0.	S' → S
1.	S → HH
2.	H → cH
3.	H → d
FIRST Sets	
S'	c, d
S	c, d
H	c, d

LALR(1) Parsing Table

STATE	action			goto	
	c	d	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

The parsing table is produced using the same algorithm, now using the DFA of the merged states. We see how the rows from the previous LR(1) parsing table simply collapse together.

Showing the parsing table from the new DFA.

LALR(1) Construction - Additional Observations

Grammar	
0.	S' → S
1.	S → HH
2.	H → cH
3.	H → d
FIRST Sets	
S'	c, d
S	c, d
H	c, d

Interestingly, the LALR(1) construction for this grammar has given us the exact same parsing table as we would have gotten with SLR.

We see states I_{47} and I_{89} calling for H reductions on next inputs of c, d, and $.

In SLR construction, we would have built the same DFA, and the corresponding states would call for the H reductions for any terminal in FOLLOW(H), which is {c, d, $}. Since this is the same set of symbols we get from unioning up the lookaheads when we did the LALR(1) state merging, we wind up with identical parsing tables.

That is not always the case. Next, we'll look at a grammar that fails with SLR, but succeeds with LALR(1).

Sometimes LALR(1) gives us the same result as SLR.

LALR(1) Construction - We're Not Done Yet

Grammar	
0.	S' → S
1.	S → HH
2.	H → cH
3.	H → d
FIRST Sets	
S'	c, d
S	c, d
H	c, d

The state merging method demonstrates what we are trying to accomplish, but it is impractical to implement.

What we need is a way to build the LALR(1) automaton without needing to use the much larger LR(1) automaton as an intermediate step.

And thus we come to what we are ultimately after, section "Efficient Construction of LALR Parsing Tables" of the Dragon Book.

This section is more confusing than it needs to be, sprinkling in some unnecessary tangents along the way. For that reason, it helped me to read Section 9.7.1.2, "The Channel Algorithm" taken from Grune & Jacobs work "Parsing Techniques: A Practical Guide".

The beginning of our quest for a usable LALR(1) construction mechanism

LALR(1) Construction - A New Sample Grammar

Grammar	
0.	S' → S
1.	S → L=R
2.	S → R
3.	L → *R
4.	L → id
5.	R → L

For this section, Dragon Book wants us to study a new grammar, one that is LALR(1) but is not SLR. Here it is:

0. S' → S
1. S → L = R
2. S → R
3. L → * R
4. L → id
5. R → L

To start, we need to construct the LR(0) automaton. We could do this the same way we've done previously, but for some reason Dragon Book wants to show us a new way that only uses "Kernel items". So we'll start there.

An LR(0) Kernel item is either the initial term S → · S, which is of course the item that seeds I_0 and thus kicks off the whole construction, or any item where the dot is not at the beginning.

A new sample grammar and introducing the concept of "Kernel items"

LALR(1) Construction - Examining the Kernel Items

Grammar	
0.	$S' \to S$
1.	$S \to L = R$
2.	$S \to R$
3.	$L \to * R$
4.	$L \to id$
5.	$R \to L$

So that means we have the following Kernel items:

$S' \to \bullet S$
$S \to L \bullet = R$
$S \to L = \bullet R$
$S \to L = R \bullet$
$S \to R \bullet$
$L \to * \bullet R$
$L \to * R \bullet$
$L \to id \bullet$
$R \to L \bullet$

Next, for a part of the procedure we're about to do, we need to pre-compute the nonterminal pairs C and A such that $C \overset{*}{\underset{rm}{\Rightarrow}} A\eta$.

Recall that $\overset{*}{\underset{rm}{\Rightarrow}}$ means a right-most derivation in zero or more steps, which means one of the possible derivations will always include C itself.

We'll start with S'.

Introducing how we can pre-compute some information rather than storing the non-Kernel items in the state.

LALR(1) Construction - Computing $C \overset{*}{\underset{rm}{\Rightarrow}} A\eta$

Grammar	
0.	$S' \to S$
1.	$S \to L = R$
2.	$S \to R$
3.	$L \to *R$
4.	$L \to id$
5.	$R \to L$

$S' \Rightarrow S$ therefore S is one of the possible A's in $S' \overset{*}{\underset{rm}{\Rightarrow}} A\eta$.

$S' \Rightarrow S$
$\Rightarrow L = R$
$\Rightarrow L = L$
$\Rightarrow L = id$

It's not really necessary to do these extra steps once we see L at the beginning

Therefore L is also in $S' \overset{*}{\underset{rm}{\Rightarrow}} A\eta$

$S' \Rightarrow S$
$\Rightarrow R$

Therefore R is also in $S' \overset{*}{\underset{rm}{\Rightarrow}} A\eta$

Repeating this process for each nonterminal gives us the table:

	A's such that $C \overset{*}{\underset{rm}{\Rightarrow}} A\eta$
S'	S', S, L, R
S	S, L, R
L	L
R	R, L

Don't forget that the original nonterminal always derives itself, and is therefore always in the result set

Showing how we pre-compute $C \overset{*}{\underset{rm}{\Rightarrow}} A\eta$ for each nonterminal in the grammar. This essentially gives us the same information as the closure operation, but we only need to do it once.

LALR(1) Construction - Gotos Using Only Kernel Items

Grammar	
0.	$S' \to S$
1.	$S \to L = R$
2.	$S \to R$
3.	$L \to *R$
4.	$L \to id$
5.	$R \to L$

Now we compute the LR(0) automaton using only kernel items. We don't compute the full closure as we've done previously.

Start with $I_0 = S' \to \bullet S$ That's all. Don't do the closure.

Now we still need to compute the full Set-Of-Items by following all of the gotos, but without the benefit of have the fully closed set of items in I_0.

To do that, we follow 2 rules

1) If $B \to \gamma \bullet X \delta$ is in I, then add $B \to \gamma X \bullet \delta$ to goto(I, X).
That just means "Move the dot one space to the right", same as we've done previously.

Introducing how we can compute the gotos of the states without the benefit of storing the non-kernel items in each state.

LALR(1) Construction - The Second Rule for Gotos

Grammar	
0.	S' → S
1.	S → L=R
2.	S → R
3.	L → *R
4.	L → id
5.	R → L

2) For every item $B \to \gamma \cdot C\delta$ in I, and every A such that $C \overset{*}{\underset{rm}{\Rightarrow}} A\eta$ where $A \to X\beta$ is production in the grammar, add $A \to X \cdot \beta$ to goto(I, X).

Note 1: I think this is a clearer description than the Dragon Book

Note 2: It's much easier to understand this rule by walking through an example.

Here's the description taken directly from Dragon Book.

> $[B \to \gamma \cdot X\delta, b]$ is in the kernel of I, then $[B \to \gamma X \cdot \delta, b]$ is in the kernel of $goto(I, X)$. Item $[A \to X \cdot \beta, a]$ is also in the kernel of $goto(I, X)$ if there is an item $[B \to \gamma \cdot C\delta, b]$ in the kernel of I, and $C \overset{*}{\underset{rm}{\Rightarrow}} A\eta$ for some η. If we precompute for each pair of nonterminals C and A whether $C \overset{*}{\underset{rm}{\Rightarrow}} A\eta$ for some η, then computing sets of items from kernels only is just slightly less efficient than doing so with closed sets of items.

This is for LR(1) items (with lookaheads). I'M focusing only on LR(0) items for now.

Concluding the description for how we compute the gotos using only Kernel-items.

LALR(1) Construction - Applying the Goto Rules to I_0

Grammar	
0.	$S' \to S$
1.	$S \to L=R$
2.	$S \to R$
3.	$L \to *R$
4.	$L \to id$
5.	$R \to L$

Back to $I_0 = S' \to \cdot S$

From rule #1, we create $I_1 = goto(I_0, S)$ and add $S' \to S\cdot$ to it.

$I_1 = S' \to S\cdot$

Now we keep looking at I_0 and move onto Rule #2.

We compare: $S' \to \cdot S$ (C=S)

To the pattern: $B \to \gamma \cdot C \delta$

We find a match with C=S. Now we lookup S in our pre-computed table and see S, L, and R.

Showing how we start the Sets-Of-Items construction by examining the gotos of I_0 using the 2 new rules that rely on the pre-computed table $C \overset{*}{\underset{rm}{\Rightarrow}} A\eta$.

LALR(1) Construction - Finding the Gotos of I_0

Grammar	
0.	$S' \to S$
1.	$S \to L=R$
2.	$S \to R$
3.	$L \to *R$
4.	$L \to id$
5.	$R \to L$

That means we have to look at all the S, L, and R productions:

$S \to L = R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

and match each of those productions to the pattern $A \to X\beta$, which is to say we are interested in the first symbol of each production.

We see we have a total of 4 symbols of interest:

L From $S \to L=R$ and $R \to L$
R From $S \to R$
* From $L \to *R$
id From $L \to id$

That tells us we need 4 more gotos originating from I_0

$I_2 = goto(I_0, L)$ $I_4 = goto(I_0, *)$
$I_3 = goto(I_0, R$ $I_5 = goto(I_0, id)$

Using the pre-computed table to identify the gotos of I_0.

LALR(1) Construction - How Goto Sets are Populated

Grammar	
0.	S' → S
1.	S → L=R
2.	S → R
3.	L → *R
4.	L → id
5.	R → L

To populate these new gotos, we only need to add items with a • after the first symbol (recalling that Rule #2 wants us to add items of the form $A \rightarrow X \bullet \beta$)

That gives us:

Dots placed after first symbol on RHS

$I_2 = S \rightarrow L \bullet = R$
$R \rightarrow L \bullet$

$I_4 = L \rightarrow * \bullet R$

$I_3 = S \rightarrow R \bullet$

$I_5 = L \rightarrow id \bullet$

Once again, since we are only building Kernel items, we do not compute the closure.

Now we see we need a goto(I_2, =)

$I_6 = goto(I_2, =) \Longrightarrow$ $I_6 = S \rightarrow L = \bullet R$

(From Rule #1)

Showing how the goto sets are populated using only the Kernel items from the original set, and the pre-computed table

LALR(1) Construction - Continue Building Goto Sets

Grammar	
0.	S' → S
1.	S → L=R
2.	S → R
3.	L → *R
4.	L → id
5.	R → L

I_4 needs a goto(I_4, R) from Rule #1 giving us:

$I_7 = goto(I_4, R) \Longrightarrow$ $I_7 = L \rightarrow *R\bullet$

But because I_4 has $L \rightarrow *\bullet R$, meaning we have a • before a nonterminal, R, so we have to apply Rule #2.

This time, C = R, so we look up R in our pre-computed table and we find R and L. So we look at all the R and L productions:

$L \rightarrow *R$
$L \rightarrow id$
$R \rightarrow L$

which tells us we need gotos for *, id, and L

But the first 2 are duplicates

goto(I_4, *) = $L \rightarrow *\bullet R$ = I_4
goto(I_4, id) = $L \rightarrow id\bullet$ = I_5

We continue with the Sets-Of-Items construction by following the gotos and populating them with Kernel items only (no closures)

LALR(1) Construction - Completing the Sets-Of-Items

Grammar	
0.	$S' \to S$
1.	$S \to L = R$
2.	$S \to R$
3.	$L \to *R$
4.	$L \to id$
5.	$R \to L$

But goto (I_7, L) is a new state giving us:

$$I_8 = \text{goto}(I_7, L) \Longrightarrow \boxed{I_8 = R \to L\bullet}$$

We continue scanning the newly created states and find that only state I_6

$$I_6 = S \to L = \bullet R$$

needs a goto for R, so we make

$$I_9 = \text{goto}(I_6, R) \Longrightarrow \boxed{I_9 = S \to L = R\bullet}$$

But we also need to apply Rule #2 to state I_6, just like we needed to for I_4 because there is a dot in front of the R.

That gives us:

$$\begin{cases} \text{goto}(I_6, *) = I_4 \\ \text{goto}(I_6, id) = I_5 \\ \text{goto}(I_6, L) = I_8 \end{cases}$$

Working our way through a few duplicate sets we complete the Sets-Of-Items construction using the Kernel-items-only technique.

LALR(1) Construction - Summary of States

Grammar	
0.	$S' \to S$
1.	$S \to L = R$
2.	$S \to R$
3.	$L \to * R$
4.	$L \to id$
5.	$R \to L$

$I_0 = goto(I_0, S)$
$I_1 = goto(I_0, L)$
$I_2 = goto(I_0, R)$
$I_3 = goto(I_0, *) = goto(I_6, *) = goto(I_4, *)$
$I_4 = goto(I_0, id) = goto(I_6, id) = goto(I_4, id)$
$I_5 = goto(I_2, =)$
$I_6 = goto(I_4, R)$
$I_7 = goto(I_4, L) = goto(I_6, L)$
$I_8 = goto(I_6, R)$
I_9

$I_0 = S' \to \cdot S$

$I_1 = S' \to S \cdot$

$I_2 = S \to L \cdot = R$
$R \to L \cdot$

$I_3 = S \to R \cdot$

$I_4 = L \to * \cdot R$

$I_5 = L \to id \cdot$

$I_6 = S \to L = \cdot R$

$I_7 = L \to * R \cdot$

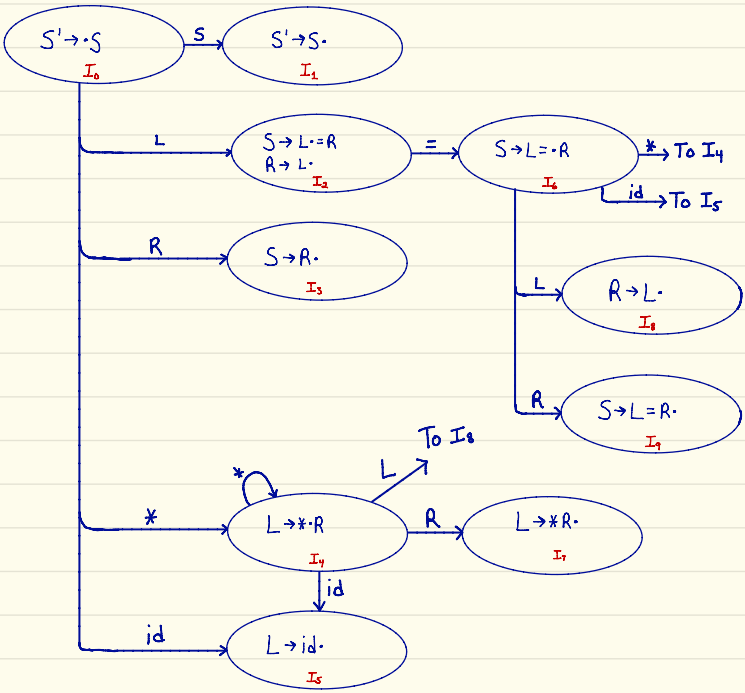
$I_8 = R \to L \cdot$

$I_9 = S \to L = R \cdot$

Showing the full Sets-Of-Items with their gotos and Kernel items.

LALR(1) Construction - The Resulting DFA

Grammar	
0.	S' → S
1.	S → L=R
2.	S → R
3.	L → *R
4.	L → id
5.	R → L



Showing the final LR(0) DFA of the sample grammar built from Kernel items only

LALR(1) Construction - Why Do We Need This?

Storing only Kernel items saves Memory

Why did the Dragon Book have us learn this new technique of constructing the LR(0) automaton with Kernel items?

(1) It seems unnecessary and distracts us from the core issue of how to compute the lookahead sets

(2) They wanted to show us a more space efficient storage for the state context. Let's remember that the book was written in the Mid-1980's when an average PC had only 640K of memory and every byte counted. In present-day 2017 My computer has 32Gb of RAM, and even that will likely seem small in the near future.

(3) The non-Kernel-items are unnecessary and we don't want to bother calculating lookahead sets for items that we don't need.

Speculating on why the Dragon Book authors wanted to teach us this technique even though it isn't strictly necessary.

LALR(1) Construction - Upgrading LR(0) to LALR(1)

How we add the lookahead sets to the LR(0) DFA

"Spontaneous Generation"

Now that we have the LR(0) automaton, we come to the heart of the matter, namely how to upgrade the automaton by injecting lookahead sets into the items. Doing this will turn it into an LALR(1) parser.

We can do this by inspecting the items and making some clever deductions using information gleaned from the grammar.

There are two basic sources of lookahead information:

(1) Spontaneous Generation
Sometimes we can deduce that an item must contain a certain lookahead symbol regardless of any other lookaheads present in any other items throughout the automaton. The item is said to "spontaneously generate" its own lookahead token.

Sometimes we can deduce that an item must contain a certain lookahead token

LALR(1) Construction - Lookahead Propagation

"Propagation"

"Channel Algorithm"

(2) Propagation

In other cases we can deduce that if a certain item contains a lookahead token, then that same token must also be present in other items in the automaton.

When this happens we say that a lookahead token "propagates" from one item to another.

Grune & Jacobs names this approach the "channel algorithm" because the propagation information can be thought of as channels that permeate through the automaton.

Thus, by starting with the spontaneously generated tokens, and then applying the propagation information in multiple passes, we can determine the complete set of lookaheads to make the automaton fully LALR(1).

Sometimes we can deduce that lookahead tokens must propagate from one item to another.

LALR(1) Construction - Other LALR(1) Algorithms

Channel algorithm

vs.

Relations algorithm

It should be noted that the channel algorithm is not the most efficient algorithm known for producing LALR(1) lookahead sets.

There is another algorithm, named the "relations algorithm" by Grune & Jacobs, originally published by Frank DeRemer and Thomas Pennello in 1982 that is significantly more efficient, and even that algorithm has had various optimizations added to it over the years.

I shall focus only on the channel algorithm here. Understanding the much more complicated relations algorithm requires some serious effort by a motivated student, and could be the sole subject of a separate notebook. Anyone wishing to study the relations algorithm would do well to first understand the more straight-forward channel algorithm presented here.

It makes sense to study the channel algorithm even though it is not the most efficient algorithm.

LALR(1) Construction - Using Dummy Lookahead Tokens

"Dummy Lookaheads"

As per usual, the theory and notation presented in the Dragon Book can be more confusing than it is illuminating.

I shall simply jump into the mechanics of the approach, letting the theory unfold as we go along.

The core idea is quite simple actually. We simply pretend that we already know the inbound lookahead token for each set. Then we compute the LR(1) closure for the set using the lookahead token that we are pretending to already know, and then just inspect the results we get.

We call this the "dummy" lookahead token, and we represent it with the symbol #.

Let's dive in and watch what happens.

Computing the LR(1) closures with a dummy lookahead token allows us to deduce the lookahead tokens of other items.

LALR(1) Construction - Closing I_0 with the Dummy LookAhead

Grammar	
0.	$S' \rightarrow S$
1.	$S \rightarrow L = R$
2.	$S \rightarrow R$
3.	$L \rightarrow * R$
4.	$L \rightarrow id$
5.	$R \rightarrow L$

We start by producing the LR(1) closure of $\{ S' \rightarrow \bullet S, \# \}$, where # is the previously discussed dummy lookahead token.

$S' \rightarrow \bullet S, \#$ $\overset{\varepsilon}{\Rightarrow}$ After the S we need to see an ε followed by #. Therefore, the input lookahead token for the S items is $FIRST(\varepsilon\#) = \#$

So therefore we add

$S \rightarrow \bullet L = R, \#$

$S \rightarrow \bullet R, \#$

Note: This is simply a review of the LR(1) closure technique discussed in the previous section.

Then we continue adding the closure items for L and R because we see they have items with preceding dots.

Reviewing how we compute LR(1) closures but this time doing it with the dummy lookahead.

LALR(1) Construction - Finding Spontaneously Generated Tokens

Grammar	
0.	S' → S
1.	S → L=R
2.	S → R
3.	L → *R
4.	L → id
5.	R → L

To close S → •L=R, # we compute FIRST(=R#) which is =.

So we add: L → •*R, =
L → •id, =

These items are telling us that one of the possible next symbols (not necessarily the only next symbol) of L → •*R and L → •id is =.

Therefore, the = lookahead token for these items is said to be spontaneously generated because it is not dependent on the inbound lookahead symbol #.

In essence, the parser is saying

"I might see *R followed by ="
"Or I might see id followed by ="

Showing L production items that have spontaneously generated lookahead tokens.

LALR(1) Construction - Items with Multiple LookAheads

Grammar	
0.	S' → S
1.	S → L=R
2.	S → R
3.	L → *R
4.	L → id
5.	R → L

To close $S \rightarrow \cdot R, \#$ we add the R production with a lookahead of $FIRST(\varepsilon\#) = \#$.

That gives us $R \rightarrow \cdot L, \#$

Now we once again need to close over the L items, but this time with a lookahead of $FIRST(\varepsilon\#) = \#$.

This gives us the items

$L \rightarrow \cdot *R, \#$
$L \rightarrow \cdot id, \#$

We can use shorthand notation to write these L items and the previous two as:

$L \rightarrow \cdot *R, =/\#$
$L \rightarrow \cdot id, =/\#$

The L production items have more than one lookahead token.

LALR(1) Construction - Summary of I_0 LR(1) Closure

Grammar	
0.	S' → S
1.	S → L=R
2.	S → R
3.	L → *R
4.	L → id
5.	R → L

Thus we end up with the LR(1) closure of I_0 looking like:

S' → ·S, #
S → ·L=R, #
S → ·R, #
L → ·*R, #/=
L → ·id, #/=
R → ·L, #

} Non-Kernel Items

Let's remember, however, that the LR(0) automaton we are building on is filled with kernel items only. This LR(1) closure operation is only an intermediate step to get us to our goal. We don't actually store these results in the I_0 state.

I think the propagation channels would be easier to reason about if we were working with the full set of closed LR(0) items, but that's not what Dragon Book chose to do.

The non-kernel items of the LR(1) closure help us establish the propagation channels, but they are only transient values and are ultimately discarded.

LALR(1) Construction - Establishing the Channels

Grammar	
0.	S' → S
1.	S → L=R
2.	S → R
3.	L → *R
4.	L → id
5.	R → L

Recall from our prior study of LR(1) parsers that lookahead tokens always propagate to the next dotted item in their goto sets.

Since we see # in the item S → •L=R, # that means the # lookahead will propagate to any states that contain:

Propagation Flow

I_0: S' → •S, □

I_0: S → •L=R, □ ← Transient Non-Kernel Item

I_2: S → L•=R, □

I_6: S → L=•R, □

I_9: S → L=R•, □

But since S' → •S is the only Kernel item actually stored in I_0, what kind of commentary can we make about S' → •S from this?

Answer: Any lookaheads for S' → •S must propagate to S → L•=R in goto(I_0, L).

Showing how propagation channels flow through the transient non-Kernel items.

LALR(1) Construction - Channels Flow from One State to Another

Grammar	
0.	$S' \rightarrow S$
1.	$S \rightarrow L = R$
2.	$S \rightarrow R$
3.	$L \rightarrow * R$
4.	$L \rightarrow id$
5.	$R \rightarrow L$

Since we are only storing kernel items in the states, we can remove the transient non-kernel item and directly express the channel as flowing from $I_0: S' \rightarrow \cdot S$ to $I_2: S \rightarrow L \cdot = R$.

$I_0: S' \rightarrow \cdot S$,
$I_2: S \rightarrow L \cdot = R$,

Direct channel after removing transient non-kernel items

I think it is a bit confusing in that by removing the transient non-kernel item we've now lost any obvious relationship between $I_0: S' \rightarrow \cdot S$ and $I_2: S \rightarrow L \cdot = R$, but again, that is how the Dragon Book chose to teach it to us.

And so we see using the Dragon Book technique that channels flow from an item in one state to another item in that state's goto list.

This is what Dragon Book's algorithm has us do, but it is less clear about why it works.

Examining how and why channels flow between states.

LALR(1) Construction - Back to Spontaneous Generation

Grammar	
0.	S' → S
1.	S → L=R
2.	S → R
3.	L → *R
4.	L → id
5.	R → L

The same logic and process applies to spontaneously generated lookaheads.

Returning to I_0, from item I_0: $L \rightarrow \cdot *R, =$ we can say that the next dotted item in the goto state - goto(I_0, *), or in other words, item I_4: $L \rightarrow *\cdot R, =$ has the lookahead token = spontaneously generated.

Again, I find the deduction confusing because a casual inspection of I_4: $L \rightarrow *\cdot R$ leaves one with no intuitive sense for how it would spontaneously generate the = lookahead.

To me it makes more sense to say that = has been spontaneously generated for the non-kernel item I_0: $L \rightarrow \cdot *R$, which has then propagated to I_4: $L \rightarrow *\cdot R$ via normal propagation rules.

Lookahead tokens are spontaneously generated in the next dotted item in the goto(I, X).

LALR(1) Construction - Rinse and Repeat

Grammar	
0.	$S' \to S$
1.	$S \to L=R$
2.	$S \to R$
3.	$L \to *R$
4.	$L \to id$
5.	$R \to L$

Continuing with the process, item $I_0: L \to \cdot id, =$ tells us that $I_5: L \to id \cdot, =$ is generated spontaneously.

After we use the non-kernel items as intermediates steps to establish the channels, we can just throw the non-kernel items away.

I have established the underlying mechanism and theory. The process must now be repeated for every kernel item in the LR(0) automaton.

This would consume several more pages of laborious and repetitive notes, so I shall simply jump to the final list of channels and proceed from there.

Having established the process, one need simply to apply it to all kernel items in the LR(0) automaton.

LALR(1) Construction - Summary of Analysis

Grammar	
0.	$S' \to S$
1.	$S \to L = R$
2.	$S \to R$
3.	$L \to *R$
4.	$L \to id$
5.	$R \to L$

Spontaneously Generated LookAheads

I_0: $S' \to \cdot S$, \$ ← Seeded lookahead token

I_4: $L \to *\cdot R$, =

I_5: $L \to id\cdot$, =

Propagation Channels

I_0: $S' \to \cdot S \implies I_1$: $S' \to S\cdot$

I_0: $S' \to \cdot S \implies I_2$: $S \to L\cdot = R$

I_0: $S' \to \cdot S \implies I_2$: $R \to L\cdot$

I_0: $S' \to \cdot S \implies I_3$: $S \to R\cdot$

I_0: $S' \to \cdot S \implies I_4$: $L \to *\cdot R$

I_0: $S' \to \cdot S \implies I_5$: $L \to id\cdot$

I_2: $S \to L\cdot = R \implies I_6$: $S \to L = \cdot R$

I_4: $L \to *\cdot R \implies I_4$: $L \to *\cdot R$

I_4: $L \to *\cdot R \implies I_5$: $L \to id\cdot$

I_4: $L \to *\cdot R \implies I_7$: $L \to *R\cdot$

I_4: $L \to *\cdot R \implies I_8$: $R \to L\cdot$

I_6: $S \to L = \cdot R \implies I_4$: $L \to *\cdot R$

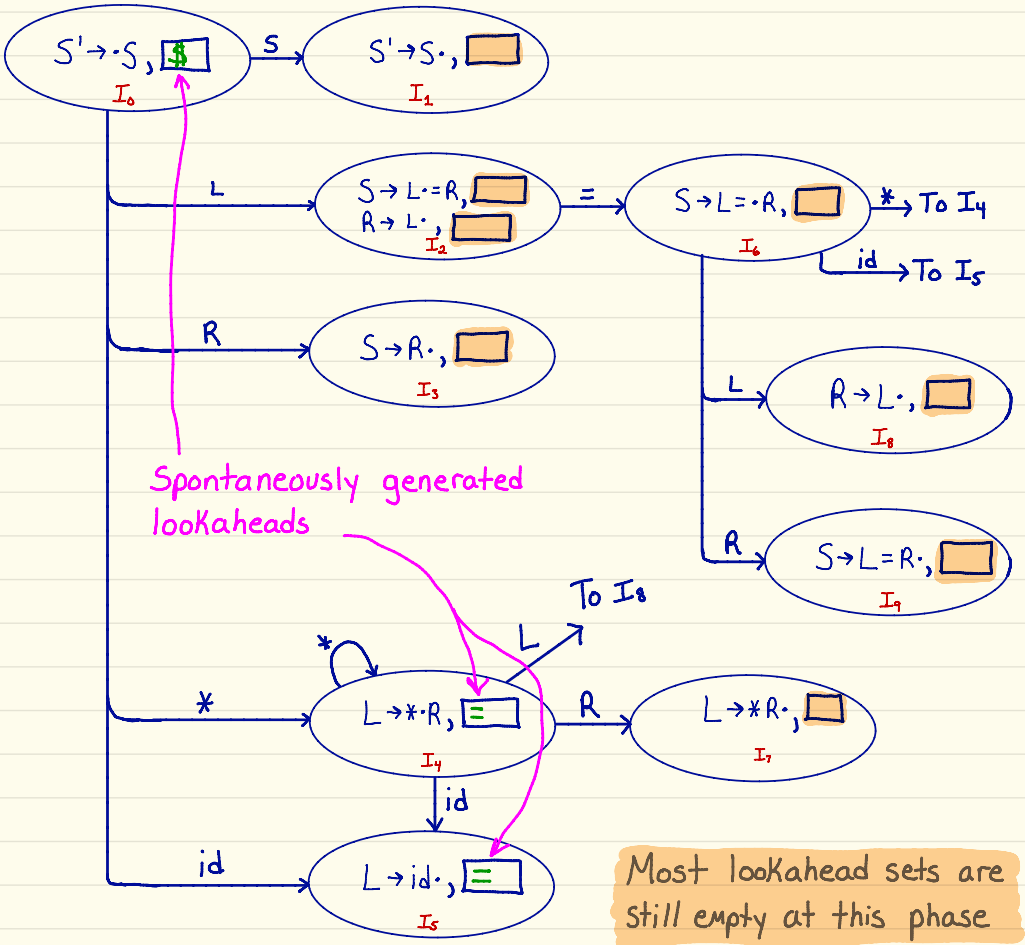
I_6: $S \to L = \cdot R \implies I_5$: $L \to id\cdot$

I_6: $S \to L = \cdot R \implies I_8$: $R \to L\cdot$

I_6: $S \to L = \cdot R \implies I_9$: $S \to L = R\cdot$

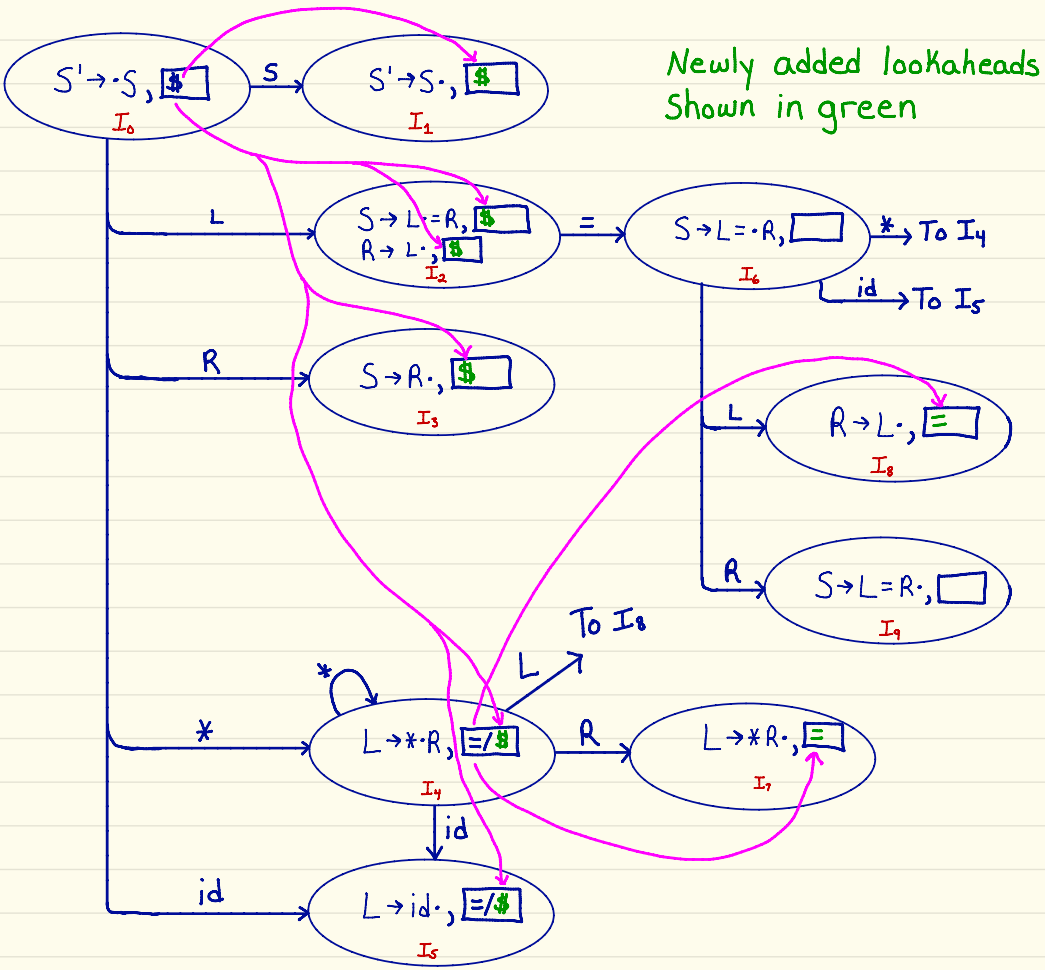
The final result of diligently searching for spontaneously generated lookaheads and propagation channels.

LALR(1) Construction - Initial Lookaheads



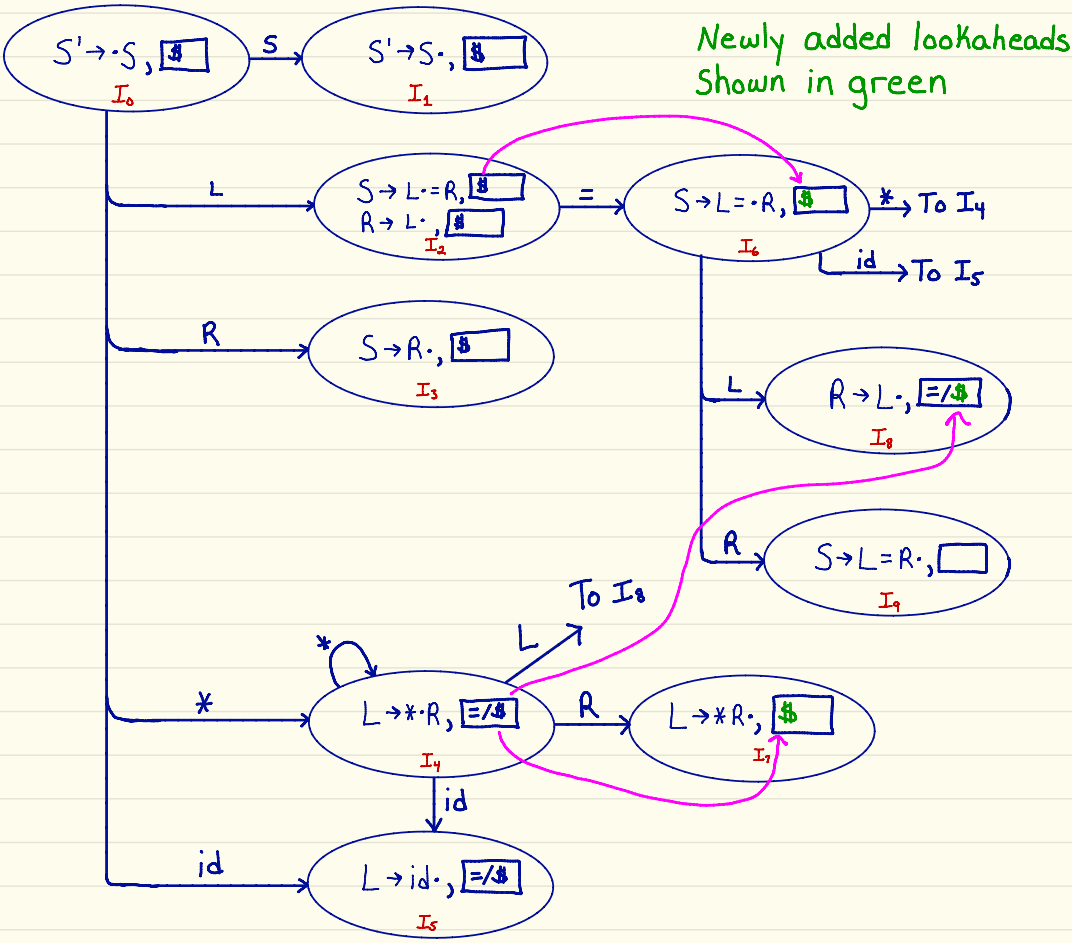
Beginning the process of upgrading the LR(0) DFA to LALR(1) by populating items with the spontaneously generated lookaheads

LALR(1) Construction - First Pass Propagation



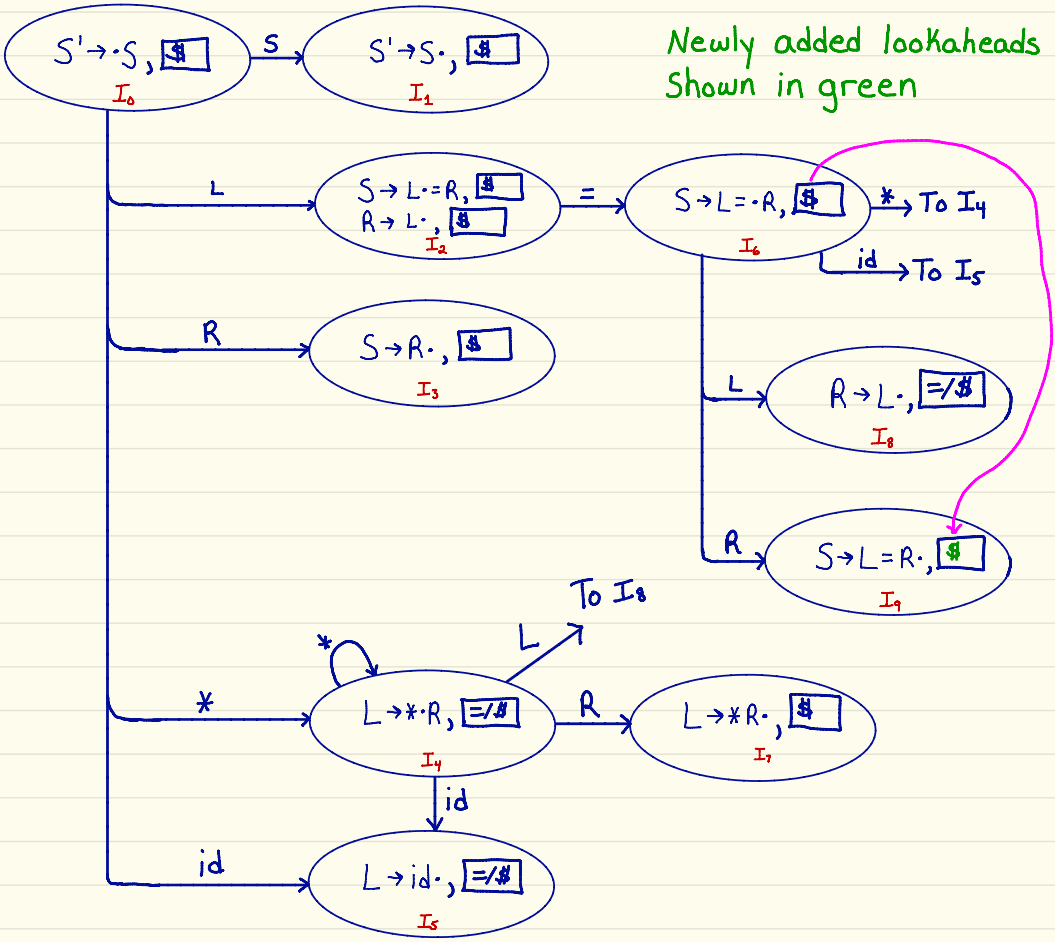
Showing the first pass of applying propagation rules to item lookaheads. The initial spontaneously generated lookaheads flow to items in I_1, I_2, I_3, I_4, I_5, I_7, and I_8.

LALR(1) Construction - Second Pass Propagation



Showing the second pass of applying the propagation rules. Lookaheads flow out of items in I_2 and I_4 into items in I_6, I_7, and I_8.

LALR(1) Construction - Third Pass Propagation



Showing the third and final pass of applying the propagation rules to the LR(0) automaton, thus completing the full LALR(1) DFA, with all lookahead sets fully populated.

LALR(1) Construction - Final States with Lookaheads

Grammar	
0.	$S' \to S$
1.	$S \to L = R$
2.	$S \to R$
3.	$L \to * R$
4.	$L \to id$
5.	$R \to L$

I_0: $S' \to \cdot S, \$$

I_1: $S' \to S \cdot, \$$

I_2: $S \to L \cdot = R, \$$
$R \to L \cdot, \$$

I_3: $S \to R \cdot, \$$

I_4: $L \to * \cdot R, = / \$$

I_5: $L \to id \cdot, = / \$$

I_6: $L = \cdot R, \$$

I_7: $L \to * R \cdot, \$$

I_8: $R \to L \cdot, = / \$$

I_9: $S \to L = R \cdot, \$$

Showing the final states of the LALR(1) automaton containing LR(1) kernel items only, with the lookahead sets achieved by following propagation channels to completion.

LALR(1) Construction - The Parsing Table

Grammar	
0.	S' → S
1.	S → L=R
2.	S → R
3.	L → *R
4.	L → id
5.	R → L

	Actions				Gotos		
	*	id	=	$	S	L	R
0	s4	s5			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s4	s5				8	7
5			r4	r4			
6	s4	s5				8	9
7				r3			
8			r5	r5			
9				r1			

Parsing table produced as per usual

- DFA edges for terminals result in shift actions
- DFA edges for non-terminals result in Gotos
- States with items ending with dots result in reduction actions for any terminal in that item's lookahead set
- Sets with item S'→S•,$ results in accept action for the $ terminal

We complete the parsing table in the same way as we did for the LR(1) construction mechanism shown in the prior section.

LALR(1) Construction - Sample Walkthrough: Step 1

Grammar	
0.	S' → S
1.	S → L = R
2.	S → R
3.	L → * R
4.	L → id
5.	R → L

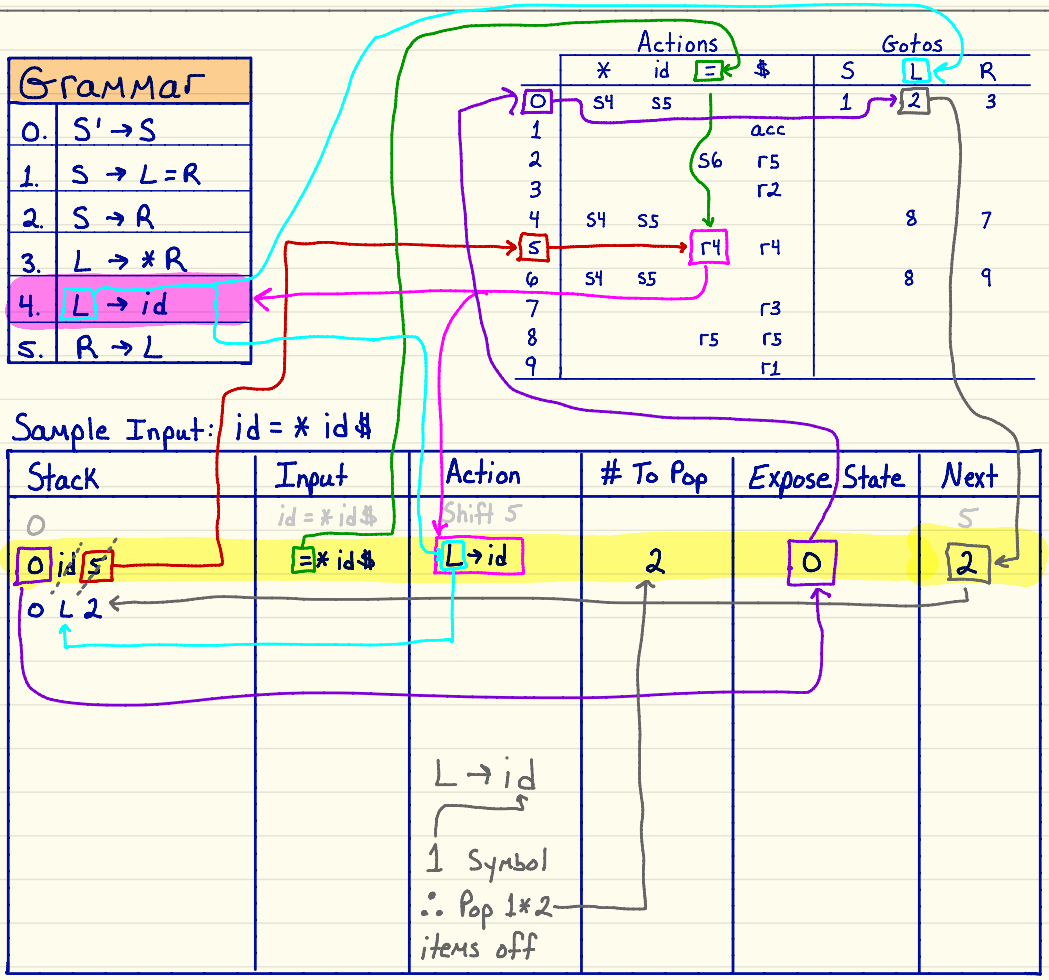
	Actions				Gotos		
	*	id	=	$	S	L	R
0	s4	s5			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s4	s5				8	7
5			r4	r4			
6	s4	s5				8	9
7				r3			
8			r5	r5			
9				r1			

Sample Input: id = * id $

Stack	Input	Action	# To Pop	Expose State	Next
0	id = * id $	Shift 5			5
0 id 5					

Reviewing the basic algorithm of an LR parser by tracing through the steps generated for the sample LALR(1) grammar for the input string "id * = id $".

LALR(1) Construction - Sample Walk through: Step 2



Reviewing the basic algorithm of an LR parser by tracing through the steps generated for the sample LALR(1) grammar for the input string "id * = id $".

LALR(1) Construction - Sample Walkthrough: All Steps

Grammar	
0.	S' → S
1.	S → L=R
2.	S → R
3.	L → *R
4.	L → id
5.	R → L

	Actions				Gotos		
	*	id	=	$	S	L	R
0	s4	s5			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s4	s5				8	7
5			r4	r4			
6	s4	s5				8	9
7				r3			
8			r5	r5			
9				r1			

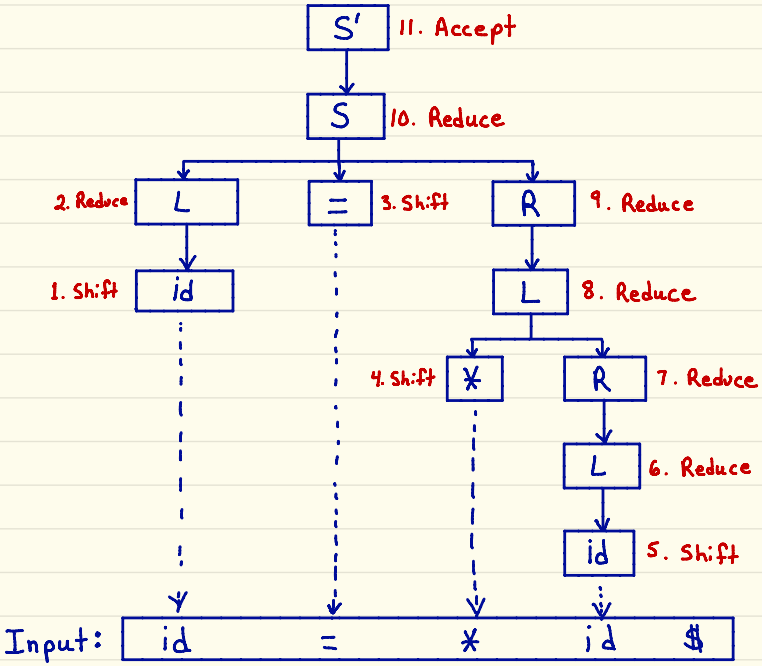
Sample Input: id = * id $

Stack	Input	Action	# To Pop	Expose State	Next
0	id = * id $	Shift 5			5
0 id 5	= * id $	L → id	2	0	2
0 L 2	= * id $	Shift 6			6
0 L 2 = 6	* id $	Shift 4			4
0 L 2 = 6 * 4	id $	Shift 5			5
0 L 2 = 6 * 4 id 5	$	L → id	2	4	8
0 L 2 = 6 * 4 L 8	$	R → L	2	4	7
0 L 2 = 6 * 4 R 7	$	L → *R	4	6	8
0 L 2 = 6 L 8	$	R → L	2	6	9
0 L 2 = 6 R 9	$	S → L=R	6	0	1
0 S 1	$	Accept			

Reviewing the basic algorithm of an LR parser by tracing through the steps generated for the sample LALR(1) grammar for the input string "id * = id $".

LALR(1) Construction - Sample Parsing Tree

Grammar	
0.	S' → S
1.	S → L=R
2.	S → R
3.	L → *R
4.	L → id
5.	R → L



Numbered sequence of parser steps shown in red

Showing how the reduction steps of the parser can be used to form a parsing tree