LALR(1) Parsing Study

This notebook was created in it's 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 that I'M interested in require an LALR(1) parser generater
- 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 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

Under Construction

BNF Notation

Under Construction

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

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 ...

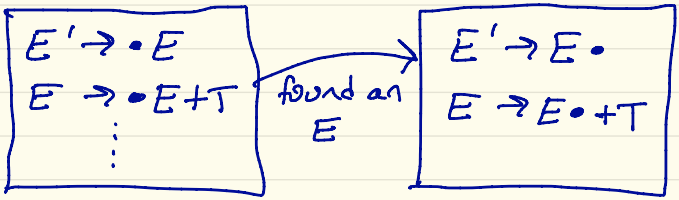
$I_0 =$

$E' \to \bullet E$

$E \to \bullet E + T$

$\vdots$

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

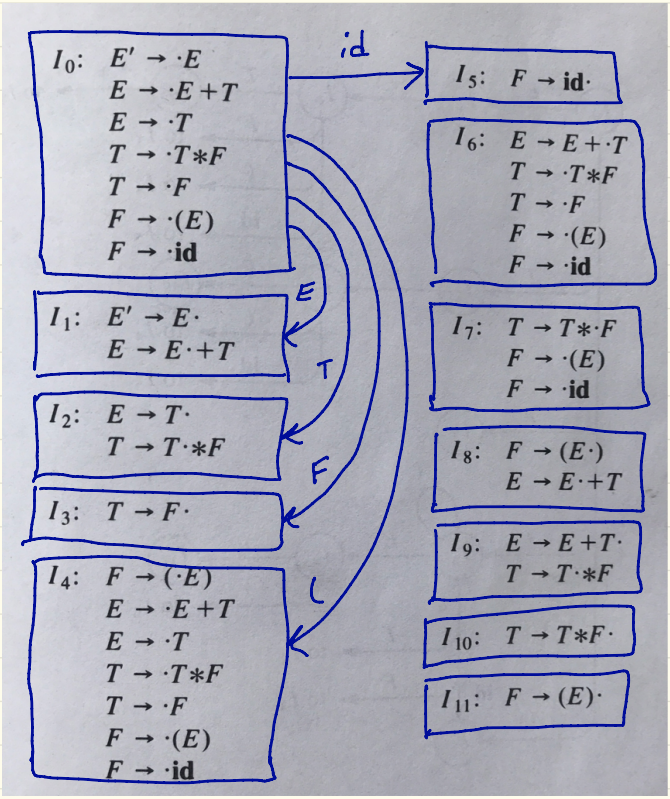
SLR Construction - Final List of States

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.

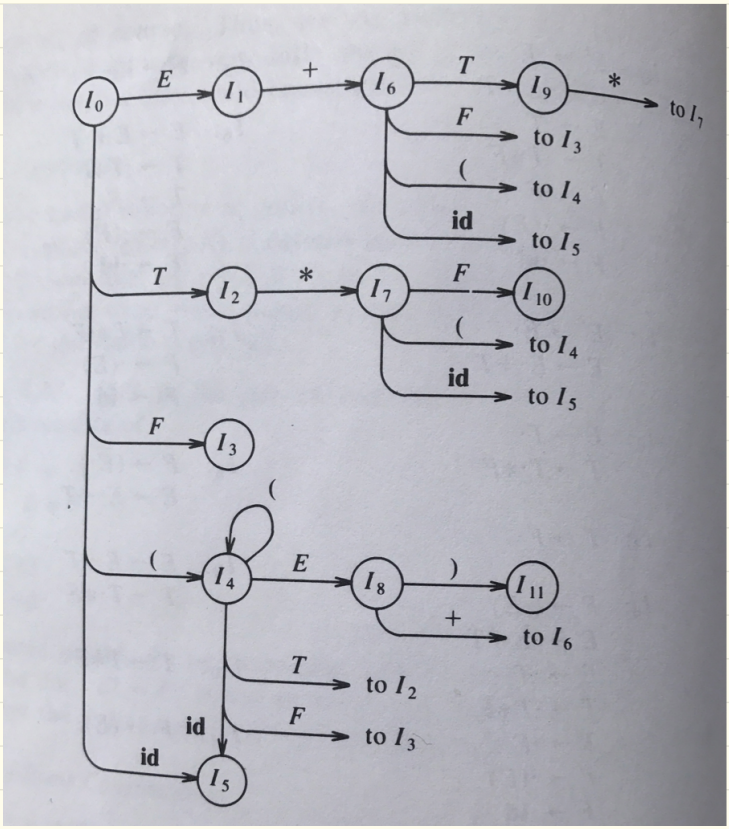
SLR Construction - Final DFA

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

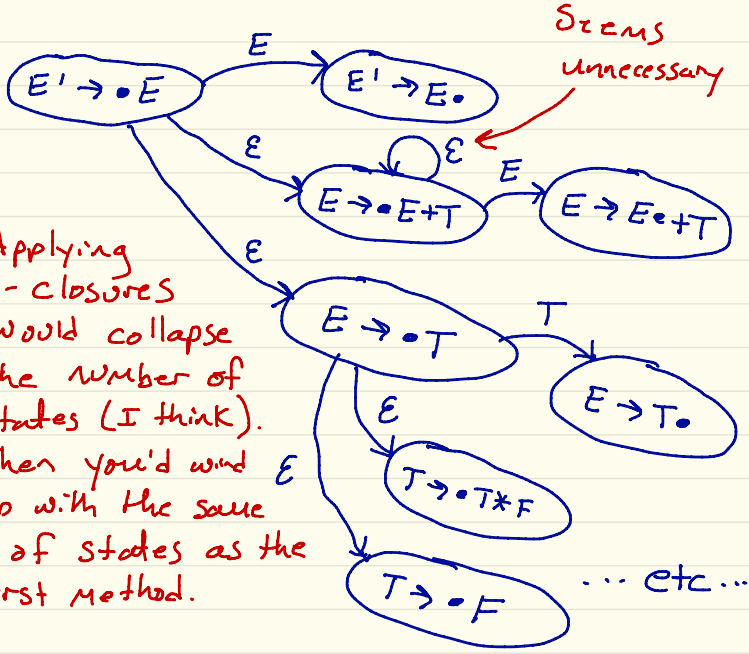
Grammar

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.

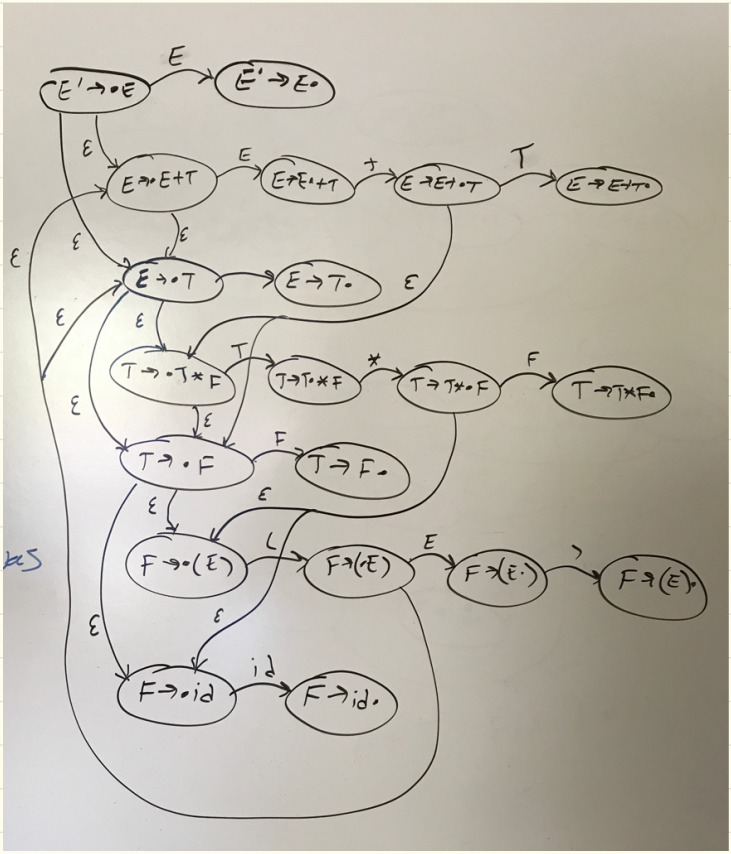
The rest of the NFA is not shown.

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



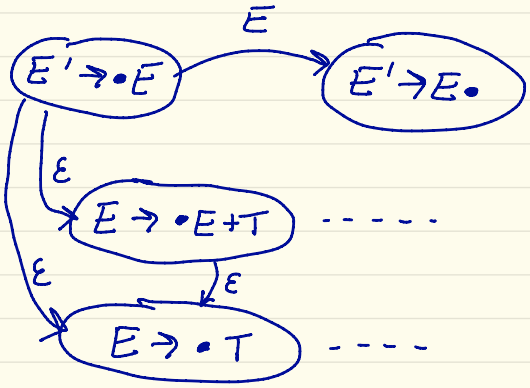
Full NFA before applying ε-closures

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

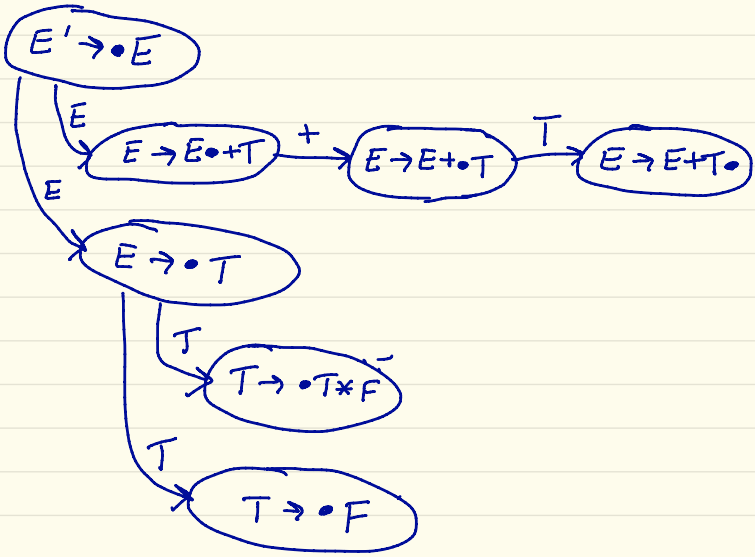
Grammar

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. $E' \to E$
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 Dragon Book tells us how to compute FIRST(X)...

FOLLOW(X)

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 $X \to \epsilon$ is a production, then add ϵ to FIRST(X). Got it. ✓
3. If X is nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ 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 $X \to \epsilon$ 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

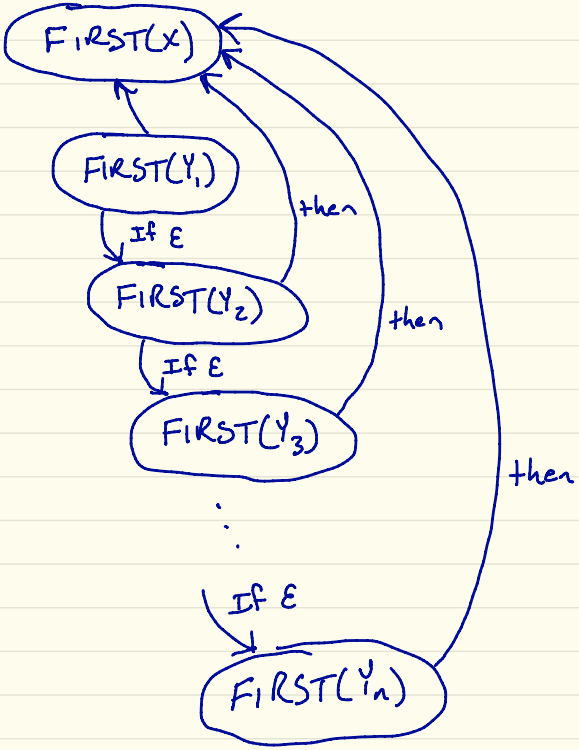
0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

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

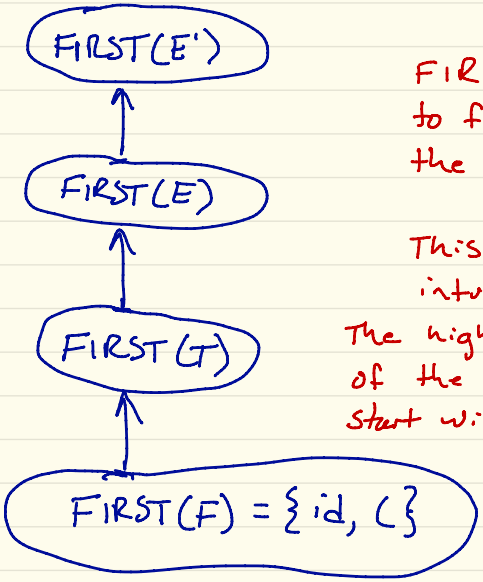
Grammar

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. $F \rightarrow id$

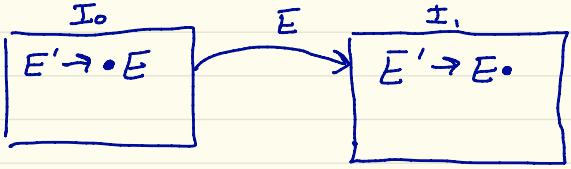
0 = Augmented Production

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

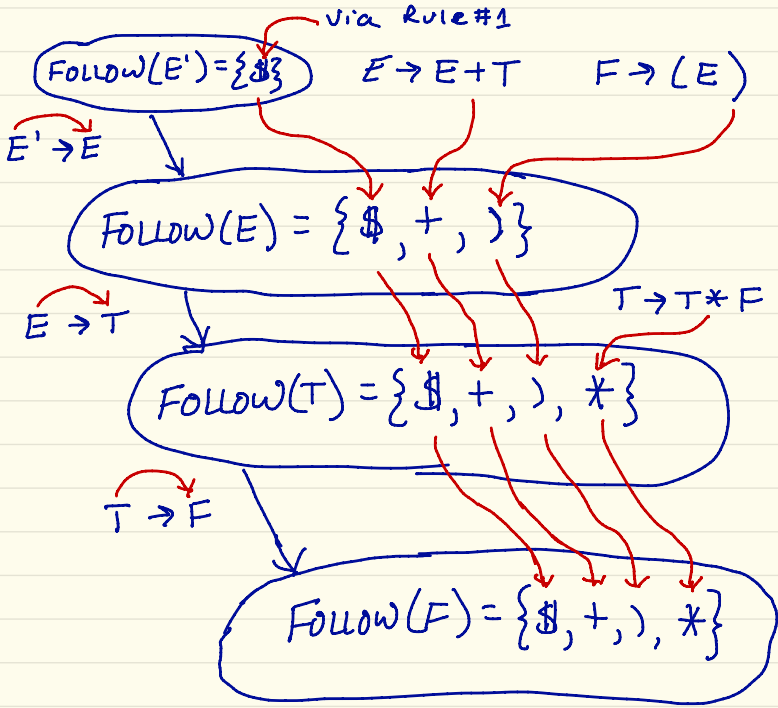
Grammar

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

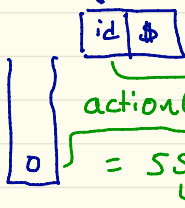
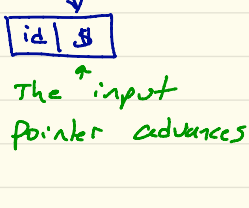
Input: id$ $ = EOF Marker

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

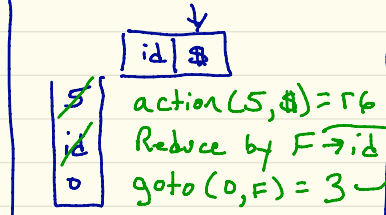
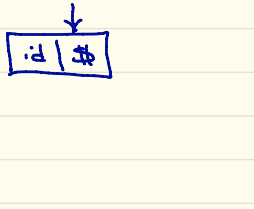
Grammar

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

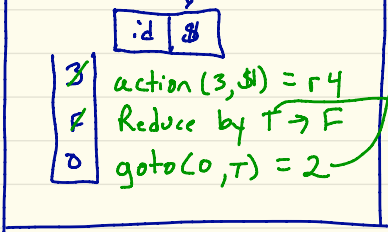
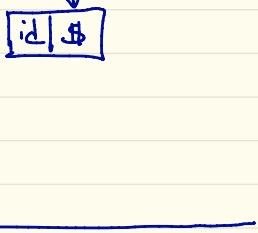
1-6 = Productions from target grammar

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 $T \rightarrow F$.

SLR Construction - Sample Walkthrough

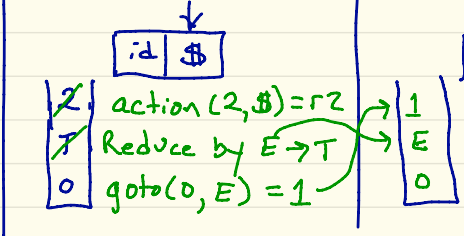
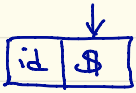
Grammar

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

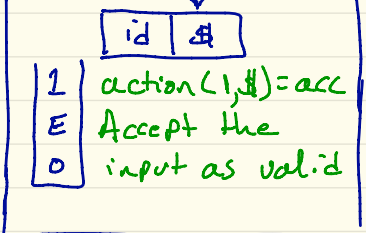
1-6 = Productions from target grammar

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

Under Construction

Part 4

LALR(1) Construction

Under Construction