Bottom-up parsing

Review of exam questions
LR(1) parsing

The content is mostly copied from

- https://web.stanford.edu/class/cs143/lectures/lecture07.pdf
- https://web.stanford.edu/class/cs143/lectures/lecture08.pdf
- Engineering a Compiler by Cooper and Torczon, 2nd Ed. ch. 3
- https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

Review

We have seen **Top-down parser**

- Recursive descent parsing
  - A simple function for each non-terminal in the grammar
- Predictive parser LL(1)
- LL(1) table-driven parsing
  - Stack, input, action

➔ Needs left-factored grammars
➔ An entry in table is defined multiple times
  - ◆ If G is ambiguous
  - ◆ If G is left recursive
  - ◆ If G is not left factored
➔ Most programming languages are not LL(1)

Bottom-up parsers

- Shift-reduce technique
- more general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing

- Bottom-up parsers don't need left-factored grammars
- Bottom-up is the preferred method

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Bottom-Up Parsing

Read the input left to right

Whenever we’ve matched the right hand side of a production,

- **reduce** it to the appropriate **non-terminal**
- and add that non-terminal to the parse tree

The upper edge of this partial parse tree is known as the **frontier**

The idea of bottom-up parsing

Revert to the “natural” grammar for our example:

```
E → T + E | T
T → int * T | int | (E)
```

Consider the string: **`int * int + int`**

> ★ Read the productions in reverse (from bottom to top)
> ★ This is a reverse rightmost derivation!

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

reduce a string to the start symbol by inverting productions:

- ➔ `int * int + int`
    - ◆ `T → int`
- ➔ `int * T + int`
    - ◆ `T → int * T`
- ➔ `T + int`
    - ◆ `T → int`
- ➔ `T + T`
    - ◆ `E → T`
- ➔ `T + E`
    - ◆ `E → T + E`
- ➔ `E`

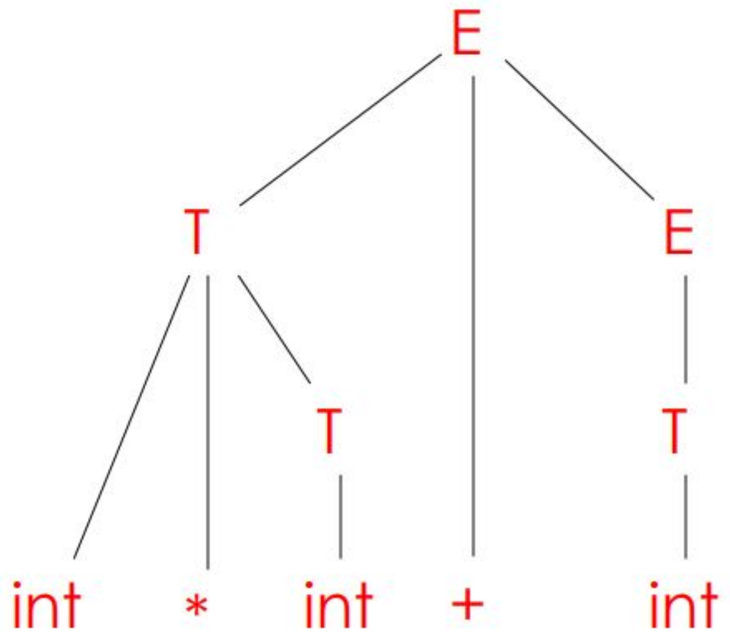
For derivation
Goal $= \gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \cdots \rightarrow \gamma_{n-1} \rightarrow \gamma_n =$ **sentence**,

The bottom-up parser discovers $\gamma_i \rightarrow \gamma_{i+1}$ before it discovers $\gamma_{i-1} \rightarrow \gamma_i$

Important Fact #1 about bottom-up parsing:

A bottom-up parser traces a rightmost derivation in reverse

```
➔   int * int + int
    ◆   T → int
➔   int * T + int
    ◆   T → int * T
➔   T + int
    ◆   T → int
➔   T + T
    ◆   E → T
➔   T + E
    ◆   E → T + E
➔   E
```



https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Example: draw bottom-up parse tree

$S \longrightarrow baTba$

$T \longrightarrow Ta \mid Tb \mid a \mid b \mid \epsilon$

input : b a a a b b a

Where Do Reductions Happen?

Important Fact #1 has an interesting consequence:

– Let $\alpha\beta\omega$ be a step of a bottom-up parse

– Assume the next reduction is by $X \rightarrow \beta$

That is $\boldsymbol{\alpha X \omega \rightarrow \alpha\beta\omega}$

– Then ω is a string of terminals

Why?

★ Because $\alpha X \omega \rightarrow \alpha\beta\omega$ is a step in a right-most derivation

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Summary of key ideas

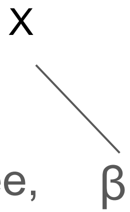
αβω Assume β is at position k, and we have a rule $X \to \beta$.

- Parser looks the current **frontier**
  - If it finds β in the frontier,
  - it can replace β with X to create a **new frontier**.
- **Handle: <X→β, k>** this pair is a handle
  - if replacing **β** with **X** at position k is the next step in a **valid derivation for the input string**
  - then the parser should replace **β** with **X**.

- **Reduction:**
  - This replacement is called a reduction because it reduces the number of symbols on the frontier, unless $|\beta| = 1$.

In the parse tree,

- build a node for X,
- add that node to the tree,
- and connect the nodes representing β as X's children.



Shift-Reduce Parsing

$x_1x_2 \ldots x_i \mid x_{i+1} \ldots x_n$

Notation:

Idea: Split string into two substrings

The dividing point is marked by a |

- The | is not part of the string

Left substring

- has terminals and non-terminals

Right substring

- is as yet unexamined by parsing (a string of terminals)

- Initially, all input is unexamined
  - $|x_1x_2 \ldots x_n$

Bottom-up parsing uses only two kinds of actions:

Shift

Reduce

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Shift and Reduce

Shift:

Move | one place to the right

Shifts a terminal to the left string

ABC|xyz ⇒ ABCx|yz

Reduce:

Apply an inverse production at the right end of the left string

If $A \rightarrow xy$ is a production, then

Cbxy|ijk ⇒ CbA|ijk

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Example with reductions

➔ `int * int | + int`
- ◆ `reduce T → int`

➔ `int * T | + int`
- ◆ `reduce T → int * T`

➔ `T + int |`
- ◆ `reduce T → int`

➔ `T + T |`
- ◆ `reduce E → T`

➔ `T + E |`
- ◆ `reduce E → T + E`

➔ `E |`

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

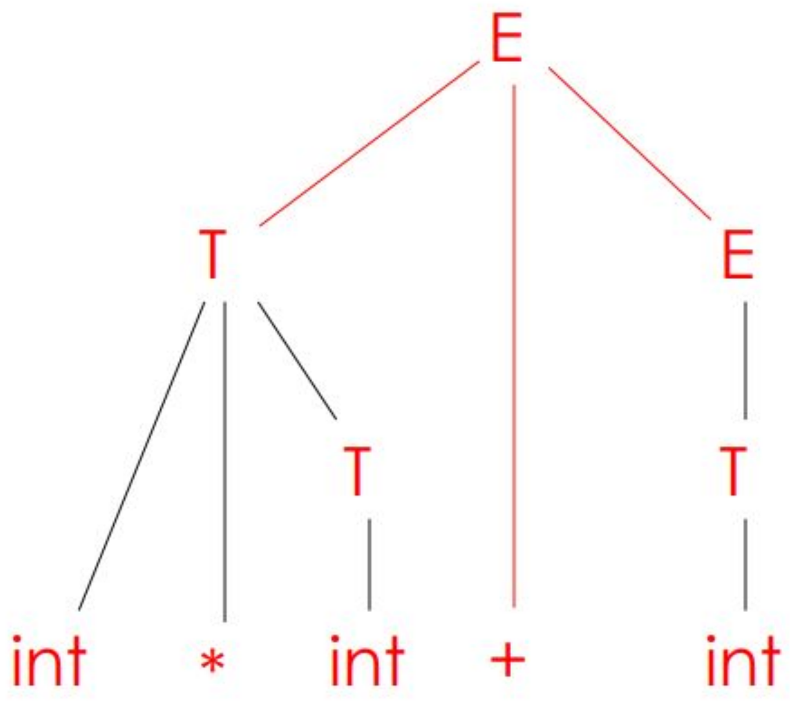
Example with shift-reduce parsing

- ➔ | int * int + int
  - ◆ shift
- ➔ int | * int + int
  - ◆ shift
- ➔ int * | int + int
  - ◆ shift
- ➔ int * int | + int
  - ◆ reduce T → int
- ➔ int * T | + int
  - ◆ reduce T → int * T
- ➔ T | + int
- ➔ T | + int
  - ◆ shift
- ➔ T + | int
  - ◆ shift
- ➔ T + int |
  - ◆ reduce T → int
- ➔ T + T |
  - ◆ reduce E → T
- ➔ T + E |
  - ◆ reduce E → T + E
- ➔ E |

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

The generation of parse tree

- ➔ | int * int + int
  - ◆ shift
- ➔ int | * int + int
  - ◆ shift
- ➔ int * | int + int
  - ◆ shift
- ➔ int * int | + int
  - ◆ reduce T → int
- ➔ int * T | + int
  - ◆ reduce T → int * T
- ➔ T | + int
  - ◆ shift
- ➔ T + | int
  - ◆ shift
- ➔ T + int |
  - ◆ reduce T → int
- ➔ T + T |
  - ◆ reduce E → T
- ➔ T + E |
  - ◆ reduce E → T + E
- ➔ E |



https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

The stack

Left string can be implemented by a stack

– Top of the stack is the |

- Shift
  - pushes a terminal on the stack
- Reduce
  - pops 0 or more symbols off of the stack (production rhs)
  - and pushes a non-terminal on the stack (production lhs)

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

1. $P \rightarrow E$
2. $E \rightarrow E + T$
3. $E \rightarrow T$
4. $T \rightarrow id (E)$
5. $T \rightarrow id$

An example Shift-Reduce Parsing with 1 lookahead

Stack	Input	Action
	id (id + id) $	shift
id	(id + id) $	shift
id (	id + id) $	shift
id (id	+ id) $	reduce T → id
id (T	+ id) $	reduce E → T
id (E	+ id) $	shift
id (E +	id) $	shift
id (E + id	) $	reduce T → id
id (E + T	) $	reduce E → E + T
id (E	) $	shift
id (E)	$	reduce T → id(E)
T	$	reduce E → T
E	$	reduce P → E
P	$	accept

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

Key issue

Example grammar:

E → T + E | T

T → int * T | int | (E)

Consider step

- **int | * int + int**
  - We could reduce by **T → int**
- **T | * int + int**
- A fatal mistake!
  - No way to reduce to the start symbol E

- **Issue1**: How do we decide when to shift or reduce?

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Another issue: Conflicts

- In a given state, more than one action (shift or reduce) may lead to a valid parse

- **A shift-reduce conflict**
  - If it is legal to shift or reduce
- A **reduce-reduce conflict**
  - if it is legal to reduce by two different productions

You will see such conflicts in your project!

★ **Issue2**: How do we solve conflicts?

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Handles

★ Intuition: Want to reduce only if the result can still be reduced to the start symbol

Assume a rightmost derivation

$$S \to^* \alpha X \omega \to \alpha \beta \omega$$

- Then $X \to \beta$ in the position after α is a handle of $\alpha\beta\omega$
- Can and must reduce at handles

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Handles formalize the intuition

– A handle is a string that can be reduced and also allows further reductions back to the start symbol (using a particular production at a specific spot)

We only want to reduce at handles

★ Note: We have said what a handle is, not how to find handles

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

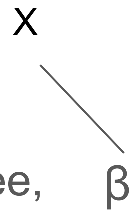
Summary of key ideas

αβω Assume β is at position k, and we have a rule $X \to \beta$.

- Parser looks the current **frontier**
  - If it finds β in the frontier,
  - it can replace β with X to create a **new frontier**.
- **Handle: <X → β, k>** this pair is a handle
  - if replacing **β** with **X** at position k is the next step in a **valid derivation for the input string**
  - then the parser should replace **β** with **X**.

- **Reduction:**
  - This replacement is called a reduction because it reduces the number of symbols on the frontier, unless |β| = 1.

In the parse tree,



- build a node for X,
- add that node to the tree,
- and connect the nodes representing β as X's children.

- ★ **Important Fact #2 about bottom-up parsing:**
  - In shift-reduce parsing, handles appear only at the top of the stack, never inside

Why?

Informal induction on # of reduce moves:

- initially, stack is empty
- Immediately after reducing a handle
  - right-most non-terminal on top of the stack
  - next handle must be to right of right-most non-terminal, because this is a right-most derivation
  - Sequence of shift moves reaches next handle

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Summary of Handles

In shift-reduce parsing, handles always appear at the top of the stack

Handles are never to the left of the rightmost nonterminal

- Therefore, shift-reduce moves are sufficient;
    - **No need for | to** move left

Bottom-up parsing algorithms are based on recognizing handles

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Recognizing handles

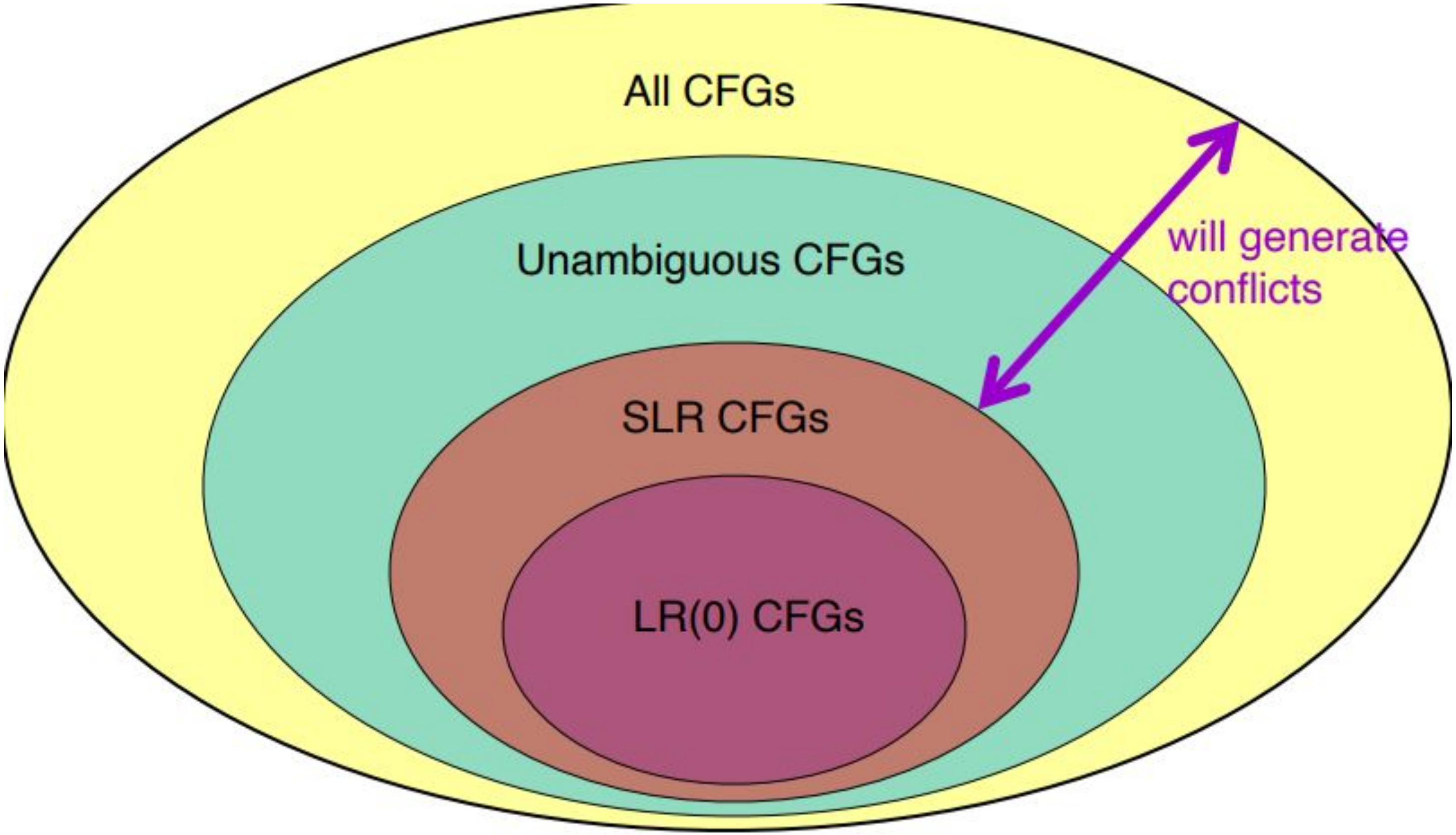
★ There are no known efficient algorithms to recognize handles

- Solution: use heuristics to guess which stacks are handles
- On some CFGs, the heuristics always guess correctly

– For the heuristics we use here, these are the SLR grammars

– Other heuristics work for other grammars

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf



https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

Viable Prefixes

It is not obvious how to detect handles

At each step the parser sees only the stack, not the entire input; start with that . . .

<u>α is a viable prefix</u>

- if there is an $\boldsymbol{\omega}$ such that $\boldsymbol{\alpha|\omega}$ is a state of a shift-reduce parser

What does this mean?

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

What does this mean?

- the right end of the handle
    - A viable prefix does not extend past this point

- **It's viable** prefix because it is a **prefix of the handle**

- As long as a parser has viable prefixes on the stack no parsing error has been detected

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

before shifting + to the stack
we have reduced (E) to B

we can only have **(, (E, (E)** on stack

- but we cannot have **(E)+** on stack because (E) is a handle and the items in the stack cannot exceed beyond the handle

A⟶B+id⟶(E)+id
Right most derivation

Operation performed	Stack	Comments
(.E)+id	**(**	shift (
(E.)+id	**(E**	shift E
(E).+id	**(E)**	shift)
B.+id	B	reduce (E) to B
B+.id	B +	shift +
B+id.	B + id	shift id
A	A	reduce B + id to A

★ (, (E, (E) are all viable prefixes for the handle (E)
★ and only these prefixes are present in stack of shift reduce parser.

https://www.geeksforgeeks.org/viable-prefix-in-bottom-up-parsing/

we keep on shifting the items until we reach the handle or an error occurs.

Once a handle is reached we reduce it with a non-terminal using the suitable production.

Thus viable prefixes help in taking appropriate shift-reduce decisions.

As long as stack contains these prefixes there cannot be any error.

$S \rightarrow AA$
$A \rightarrow bA \mid a$

Input string: **bbbaa**

S.No.	Reverse Rightmost Derivation with Handles	Viable Prefix	Comments
1.	S -> bbb**a**a	b, bb, bbb, bbba	Here, a is the handle so viable prefix cannot exceed beyond a.
2.	S -> bb**bA**a	b, bb, bbb, bbbA	Here, bA is the handle so viable prefix cannot exceed beyond bA.
3.	S -> b**bA**a	b, bb, bbA	Here also, bA is the handle so viable prefix cannot exceed beyond bA.
4.	S -> **bA**a	b, bA	Here also, bA is the handle so viable prefix cannot exceed beyond bA.
5.	S -> A**a**	A, Aa	Here, a is the handle so viable prefix cannot exceed beyond a.
6.	S -> **AA**	A, AA	Here, AA is the handle so viable prefix cannot exceed beyond AA.

https://www.geeksforgeeks.org/viable-prefix-in-bottom-up-parsing/

Important Fact #3 about bottom-up parsing:

For any grammar,

the set of viable prefixes is a **regular language**

Important Fact #3 is non-obvious

Next

we will show how to compute automata that accept viable prefixes

And how to build action, goto tables

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

A simple table driven **LR(1)** parser.

- LR(1): left-to-right scan, reverse rightmost derivation, and 1 symbol of lookahead

We will build these tables!

1	*Goal*	→	*List*
2	*List*	→	*List Pair*
3		|	*Pair*
4	*Pair*	→	(*Pair*)
5		|	( )

	Action Table			Goto Table	
State	eof	(	)	***List***	***Pair***
0		s 3		1	2
1	acc	s 3			4
2	r 3	r 3			
3		s 6	s 7		5
4	r 2	r 2			
5			s 8		
6		s 6	s 10		9
7	r 5	r 5			
8	r 4	r 4			
9			s 11		
10			r 5		
11			r 4		

Engineering a Compiler by Cooper and Torczon, 2nd Ed. ch. 3

Behaviour for the input string "**( )**"

Iteration	State	word	Stack	Handle	Action
initial	—	(	$ 0	— *none* —	—
1	0	(	$ 0	— *none* —	*shift 3*
2	3	)	$ 0 (3	— *none* —	*shift 7*
3	7	eof	$ 0 (3) 7	( )	*reduce 5*
4	2	eof	$ 0 *Pair* 2	*Pair*	*reduce 3*
5	1	eof	$ 0 *List* 1	*List*	*accept*

When it finds a handle <A →β, k>, it reduces β at k to A

Recognizing viable prefixes

Items:

An item is a production with a "." somewhere on the rhs, denoting a focus point

Items are often called "LR(0) items"

The item for $X \to \varepsilon$ is $X \to .$

The items for $T \to (E)$

Items

$T \to .(E)$

$T \to (.E)$

$T \to (E.)$

$T \to (E).$

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

Intuition

```
E → T + E | T
T → int * T | int | (E)
```

The problem in recognizing viable prefixes is that

- the stack has only bits and pieces of the rhs of productions
  - If it had a complete rhs, we could reduce
- These bits and pieces are always prefixes of rhs of productions

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

$$E \rightarrow T + E \mid T$$
$$T \rightarrow \texttt{int} * T \mid \texttt{int} \mid (E)$$

Intuition

Consider the input **(int)**

Then **(E |)** is a state of a shift-reduce parse

- **(E** is a prefix of the rhs of **T → (E)**
  - Will be reduced after the next shift

Item **T → (E.)**

- says that so far we have seen **(E** of this production
- and hope to see **)**

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

generalization

The stack may have many prefixes of rhs's

$$\mathbf{Prefix_1\ Prefix_2\ .\ .\ .\ Prefix_{n-1}\ Prefix_n}$$

Let $\mathbf{Prefix_i}$ be a prefix of rhs of $\mathbf{X_i} \rightarrow \boldsymbol{\alpha_i}$

- $\mathbf{Prefix_i}$ will eventually reduce to $\mathbf{X_i}$
- The missing part of $\mathbf{Prefix_{i-1}}$ of $\boldsymbol{\alpha_{i-1}}$ starts with $\mathbf{X_i}$
- i.e. there is a $\mathbf{X_{i-1}} \rightarrow \mathbf{Prefix_{i-1}\ X_i}\ \boldsymbol{\beta}$ for some $\boldsymbol{\beta}$

Recursively, $\mathbf{Prefix_{k+1} \ldots Prefix_n}$ eventually reduces to the missing part of $\boldsymbol{\alpha_k}$

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

example

$E \to T + E \mid T$

$T \to int * T \mid int \mid (E)$

Consider the string **(int * int)**:

(int * | int) is a state of a shift-reduce parser

From top of the stack:

> "**int ***" is a prefix of the rhs of **T → int * T**
>
> "**ε**" is a prefix of the rhs of **E → T**
>
> "**(**" is a prefix of the rhs of **T → (E**

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

example

$E \rightarrow T + E \mid T$

$T \rightarrow int * T \mid int \mid (E)$

The stack of items

T → int * .T

E → .T

T → (.E)

Says

We've seen **int *** of **T → int * T**

We've seen **ε** of **E → T**

We've seen **(** of **T → (E)**

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

Back to recognizing viable prefixes

Idea: To recognize viable prefixes, we must

– Recognize a sequence of partial rhs's of productions,

where

– Each sequence can eventually reduce to part of the missing suffix of its predecessor

NFA to recognize viable prefixes

1. Add **a new start** production

   $S' \to S$ to G

2. States are **the items of G**

3. Every state is an accepting state

4. Start state is $S' \to .S$

Transitions

For item $E \to \alpha.X\beta$

- add transition

  $E \to \alpha.X\beta \xrightarrow{X} E \to \alpha X.\beta$

For item $E \to \alpha.X\beta$ and production $X \to \gamma$

- Add transition

  $E \to \alpha.X\beta \xrightarrow{\varepsilon} X \to .\gamma$

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

Example

`E → T + E | T`

`T → int * T | int | (E)`

Start state:

S'→.E

- Transition with **E**

  S'→E.

- Transition with **ε**

  E→.T+E

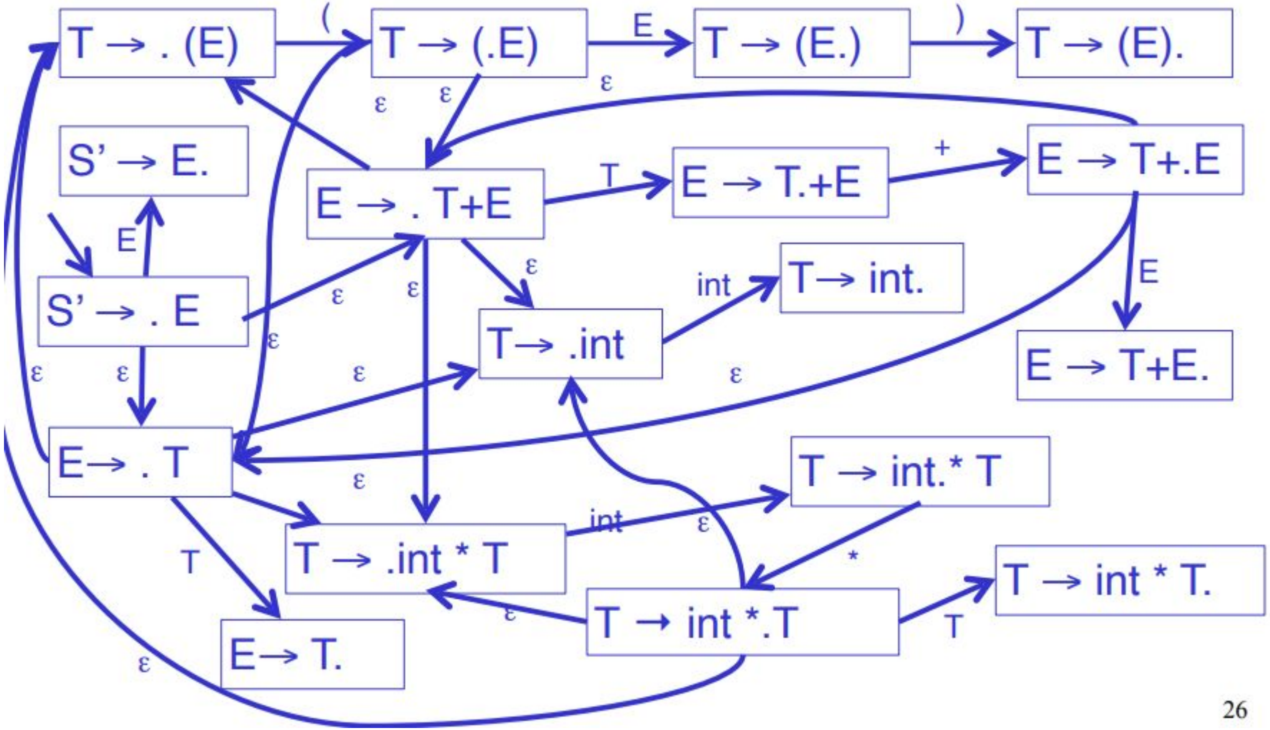
  E→.T

Let's draw the rest as NFA

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

E → T + E | T
T → int * T | int | (E)

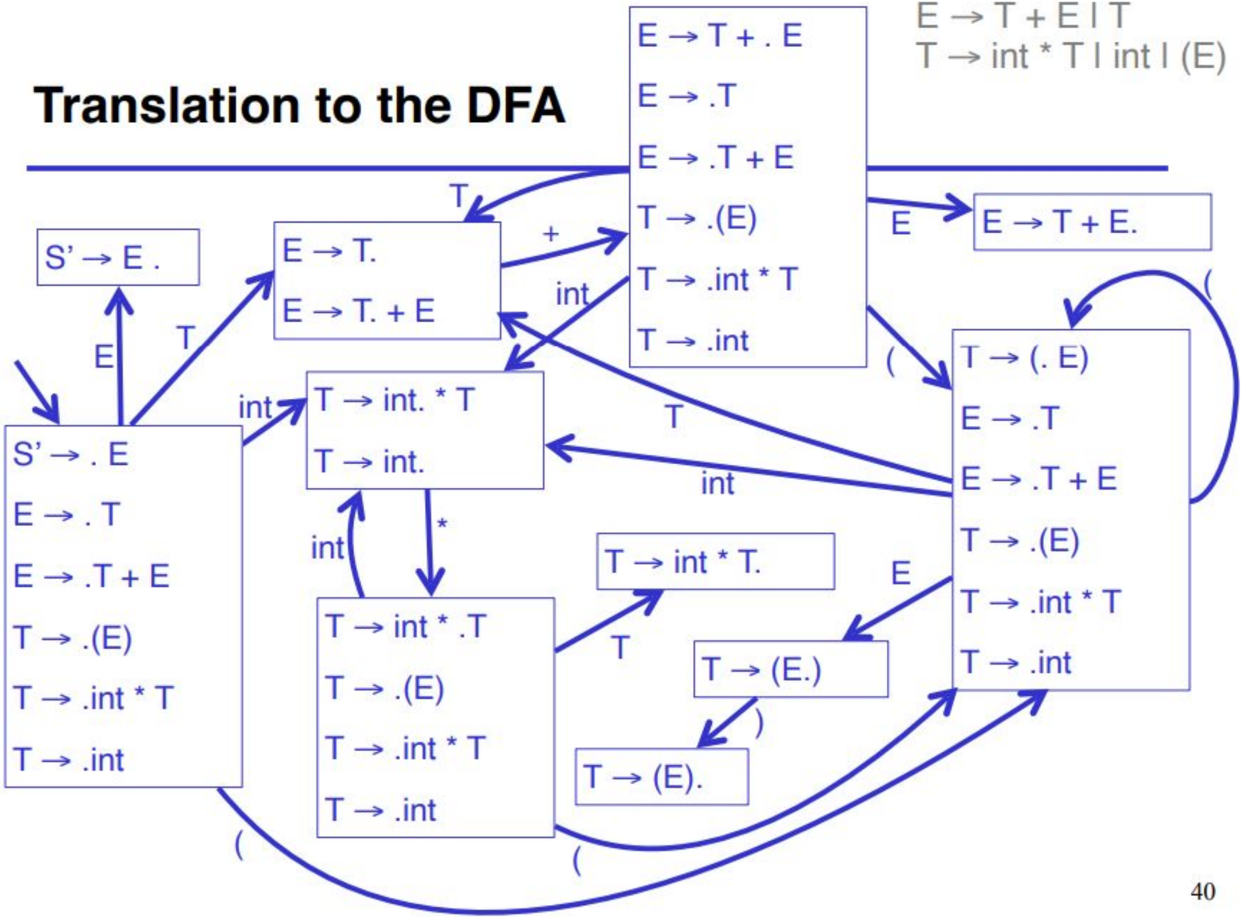
NFA for Viable Prefixes



https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

Translation to the DFA

$E \to T + E \mid T$
$T \to int * T \mid int \mid (E)$



https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

LR(0) Automaton

The states of the DFA are

“canonical collections of items”

or

“canonical collections of LR(0) items”

An LR(0) automaton represents

- all the possible rules that are currently under consideration by a shift-reduce parser.
- **compact finite state machine of the grammar.**

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

Another way of finding LR(0) automaton

1. $P \rightarrow E$
2. $E \rightarrow E + T$
3. $E \rightarrow T$
4. $T \rightarrow id\ (\ E\ )$
5. $T \rightarrow id$

State0 = Start(Kernel)

$P \rightarrow .\ E$

Compute the **ε-closure of State0**

$P \rightarrow .\ E$

$E \rightarrow .\ E + T$

$E \rightarrow .\ T$

keep repeating until no newly element added

Final **ε-closure of State0**

$P \rightarrow .\ E$

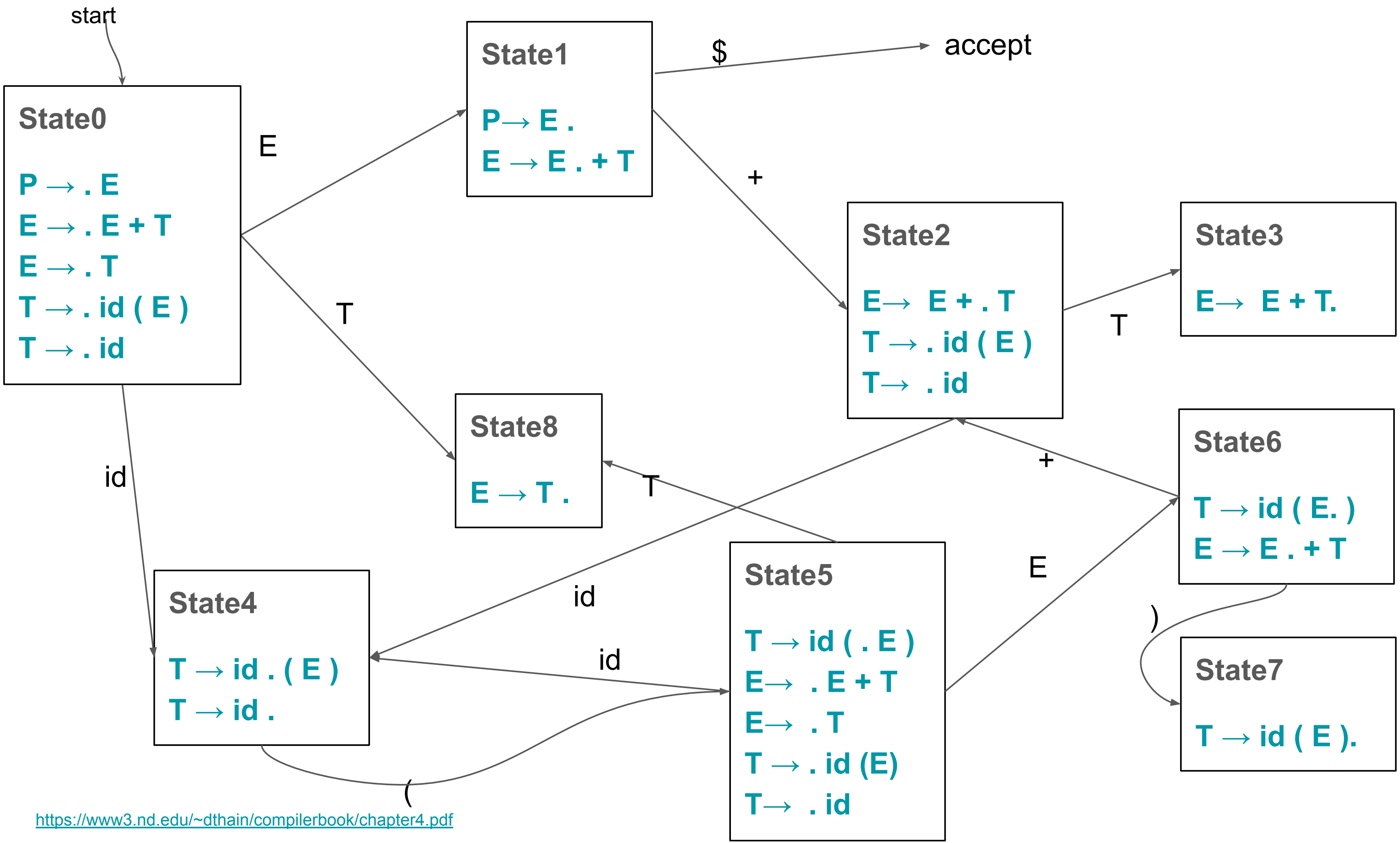
$E \rightarrow .\ E + T$

$E \rightarrow .\ T$

$T \rightarrow .\ id\ (\ E\ )$

$T \rightarrow .\ id$

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf



https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

LR(0) automaton

The LR(0) automaton tells us the choices available at any step of bottom up parsing

Reduction:

- A state containing an item with a "**.**" at the end of the rule, that indicates a possible reduction

Shift:

- A transition on a terminal that moves the "**.**" one position to the right indicates a possible shift.

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

Shift and reductions

Assume

– stack contains α

– next input is t

– DFA on input α terminates in state s

Reduce by $X \rightarrow \beta$ if

- s contains item $\mathbf{X \rightarrow \beta.}$

Shift if

- s contains item $X \rightarrow \beta.t\omega$
- equivalent to saying s has a transition labeled $\mathbf{t}$

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

Conflicts

shift-reduce conflict

A state

T → id . (E)
T → id .

S → ifthen S .
S → ifthen S . else S

★ Any state has a reduce item and a shift item:
X → β. and **Y → ω.tδ**

reduce-reduce conflict

S → id (E) .
E → id (E) .

★ Any state has two reduce items
  - **X → β.** and **Y → ω.**

LR(0) automaton

- forms the basis of LR parsing, by telling us which actions are available in each state.

- But, it does not tell us which action to take or how to resolve shift-reduce and reduce-reduce conflicts

SLR = "Simple Left-to-right scan"

SLR improves on LR(0) shift/reduce heuristics – Fewer states have conflicts

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

SLR Parsing

Idea:

use FOLLOW sets to resolve conflicts in the LR(0) automaton

- take the reduction $X \rightarrow \beta$ only when the next token on the input is in **FOLLOW(X)**

– stack contains α

– next input is t

– DFA on input α terminates in state s

Reduce by $X \rightarrow \beta$ if

- s contains item $X \rightarrow \beta.$
- **and** $t \in$ **Follow(X)**

Shift if

- s contains item $X \rightarrow \beta.t\omega$

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

If there are still conflicts under these rules;

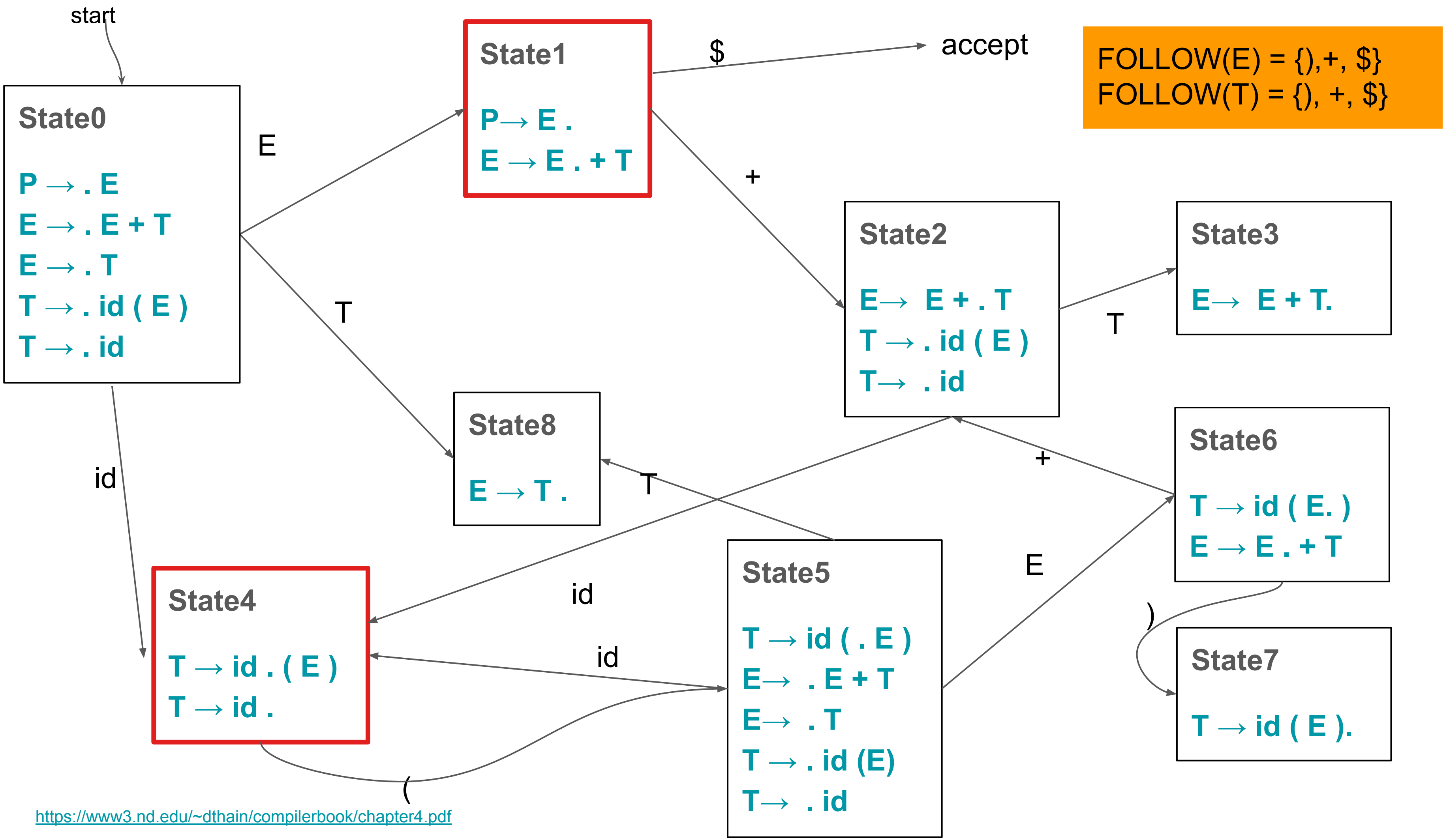
- the grammar is not SLR

The rules amount to a heuristic for detecting handles

- The SLR grammars are those where the heuristics detect exactly the handles

To solve further conflicts:

1. Fix grammar
2. choose to shift instead of reduce
3. Precedence Declarations
4. Declaring “*” has higher precedence than +”
   - **E → E * E .**
   - **E → E . + E**



https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

SLR parser table: goto table

Avoiding DFA Rescanning

Encode the DFA in a Table

- Goto table
  - the transitions to take when we back up into a state after a reduction
  - and then make a transition using the newly pushed (reduced) non-terminal
- Action table
  - what to do given the current state and the next input symbol

Goto table

goto[i,A] = j

- if $\textbf{state}_i \rightarrow^{A} \textbf{state}_j$

goto is just the transition function of the DFA

– One of two parsing tables

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

SLR parser table: actions

Action Table

For each state s_i and terminal t

action[i,t] = shift k

- If s_i has item $X \rightarrow \alpha.t\beta$ and **goto[i,t] = k**

action[i,t] = reduce $X \rightarrow \alpha$

- If s_i has item $X \rightarrow \alpha.$
- and $t \in$ **Follow(X)** and $X \neq S'$

action[i,t] = error

- If s_i has item $S' \rightarrow S.$

Otherwise,

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

SLR Parse Table for Grammar

1. P → E
2. E → E + T
3. E → T
4. T → id (E)
5. T → id

State	GOTO		ACTION				
	E	T	id	(	)	+	$
0	G1	G8	S4				
1						S2	R1
2		G3	S4				
3					R2	R2	R2
4				S5	R5	R5	R5
5	G6	G8	S4				
6					S7	S2	
7					R4	R4	R4
8					R3	R3	R3

R2 means reduced by 2: **E → E + T**
This pops 3 symbols from top of stack.

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

SLR Parsing Algorithm.

```
Let S be a stack of LR(0) automaton states.
Push <a, S0 onto S.
Let a be the first input token.
Loop:
     Let s be the top of the stack.
     If ACTION[s, a] is accept:
          Parse complete.
     Else if ACTION[s, a] is shift t:
          Push state t on the stack.
          Let a be the next input token.
     Else if ACTION[s, a] is reduce A → β:
          Pop states corresponding to β from the stack.
          Let t be the top of stack.
          Push GOTO[t, A] onto the stack.
     Otherwise:
          Halt with a parse error.
```

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

Stack =symbols	Symbols	Input	Action
0		id (id + id) $	shift 4
0 4	id	(id + id) $	shift 5
0 4 5	id (	id + id) $	shift 4
0 4 5 4	id (id	+ id) $	reduce T → id
0 4 5 8	id (T	+ id) $	reduce E → T
0 4 5 6	id (E	+ id) $	shift 2
0 4 5 6 2	id (E +	id)$	shift 4
0 4 5 6 2 4	id (E + id	) $	reduce T → id
0 4 5 6 2 3	id (E + T	) $	reduce E → E + T
0 4 5 6	id (E	) $	shift 7
0 4 5 6 7	id (E)	$	reduce T → id(E)
0 8	T	$	reduce E → T
0 1	E	$	accept

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

Step-by-Step Explanation for Parse of "id (id + id) $"

Step 1: `0` + `id` → shift 4

- Stack: `0`, Input: `id ( id + id ) $`
- ACTION[0,id] = S4 (shift to state 4)
- Stack becomes: `0 4`

Step 2: `0 4` + `(` → shift 5

- Stack: `0 4`, Input: `( id + id ) $`
- ACTION[4,(] = S5
- Stack becomes: `0 4 5`

Step 3: `0 4 5` + `id` → shift 4

- Stack: `0 4 5`, Input: `id + id ) $`
- ACTION[5,id] = S4
- Stack becomes: `0 4 5 4`

Step 4: `0 4 5 4` + `+` → reduce T → id

- Stack: `0 4 5 4`, Input: `+ id ) $`
- ACTION[4,+] = R5 (reduce by T → id)
- Pop 1 state (state 4), revealing state 5
- GOTO[5,T] = 8
- Stack becomes: `0 4 5 8`

Step 5: `0 4 5 8` + `+` → reduce E → T

- Stack: `0 4 5 8`, Input: `+ id ) $`
- ACTION[8,+] = R3 (reduce by E → T)
- Pop 1 state (state 8), revealing state 5
- GOTO[5,E] = 6
- Stack becomes: `0 4 5 6`

Step 6: `0 4 5 6` + + → shift 2

- Stack: `0 4 5 6`, Input: `+ id ) $`
- ACTION[6,+] = S2
- Stack becomes: `0 4 5 6 2`

Step 7: `0 4 5 6 2` + id → shift 4

- Stack: `0 4 5 6 2`, Input: `id ) $`
- ACTION[2,id] = S4
- Stack becomes: `0 4 5 6 2 4`

Step 8: `0 4 5 6 2 4` +) → reduce T → id

- Stack: `0 4 5 6 2 4`, Input: `) $`
- ACTION[4,)] = R5 (reduce by T → id)
- Pop 1 state (state 4), revealing state 2
- GOTO[2,T] = 3
- Stack becomes: `0 4 5 6 2 3`

Step 9: `0 4 5 6 2 3` +) → reduce E → E + T

- Stack: `0 4 5 6 2 3`, Input: `) $`
- ACTION[3,)] = R2 (reduce by E → E + T)
- Pop 3 states (states 3,2,6), revealing state 5
- GOTO[5,E] = 6
- Stack becomes: `0 4 5 6`

Step 10: `0 4 5 6` +) → shift 7

- Stack: `0 4 5 6`, Input: `) $`
- ACTION[6,)] = S7
- Stack becomes: `0 4 5 6 7`
-

Step 11: `0 4 5 6 7` + $ → reduce T → id(E)

- Stack: `0 4 5 6 7`, Input: $
- ACTION[7,$] = R4 (reduce by T → id(E))
- Pop 4 states (states 7,6,5,4), revealing state 0
- GOTO[0,T] = 8
- Stack becomes: `0 8`

Step 12: `0 8` + $ → reduce E → T

- Stack: `0 8`, Input: $
- ACTION[8,$] = R3 (reduce by E → T)
- Pop 1 state (state 8), revealing state 0
- GOTO[0,E] = 1
- Stack becomes: `0 1`

Step 13: `0 1` + $ → accept

- Stack: `0 1`, Input: $
- ACTION[1,$] = R1 (reduce by P → E) and accept

Notes on SLR parsing

SLR: Uses FOLLOW sets globally

➔ Reductions allowed if next token ∈ FOLLOW(left-hand-side)

Some common constructs are not SLR(1)

LR(1): Uses precise lookahead sets computed for each item in each state

➔ Reductions allowed only if next token ∈ specific lookahead set for that particular item

LR(1) is more powerful (**L**eft to right scan, **R**ightmost derivation, **1** symbol lookahead)

- An LR(1) item is a pair: (LR(0) item, x lookahead)

[T→ . int * T, $] means

- After seeing **T→ int * T** reduce if lookahead is **$**

More accurate than just using follow sets

LR(1)

The lookahead is always a subset of the FOLLOW of the relevant non-terminal.

For an item like **$A \rightarrow \alpha.B$** with a lookahead of **{L}**,

- add new rules like **$B \rightarrow .\gamma$** with a lookahead of **{L}**.

For an item like **$A \rightarrow \alpha.B\beta$**, with a lookahead of **{L}**,

- add new rules like **$B \rightarrow .\gamma$** with a lookahead
  - **FIRST(β)** If β cannot produce ε,
  - **FIRST(β) ∪ {L}** If β can produce ε,

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

Example LR(1)

1. S → V = E
2. S → id
3. V → id
4. V → id [E]
5. E → V

state0

S → . V = E	**{$}**
S → . id	**{$}**

Closure of state0

S → . V = E	**{$}**	
S → . id	**{$}**	
V → . id	**{=}**	**since “=” follows V**
V → . id [E]	**{=}**	

Closure of state1

S → id .	**{$}**
V → id .	**{=}**
V → id . [E]	**{=}**

- When the next token is $,
  - reduce by S → id.
- When the next token is =,
  - reduce by V → id.

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

Almost all practical programming languages have an LR(1) grammar

- LALR(1) is the merged states of LR(1):
  - lookahead is the union of the lookaheads of the LR(1) items
  - can parse most real languages,
  - tables are more compact,
  - used by YACC/Bison/CUP/etc.

Final relation

- LL(1) ⊂ SLR ⊂ LALR ⊂ LR(1) ⊂ CFG

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf