Bottom-Up Parsing and LR(1) Grammars

- A ***bottom-up parser*** is one that generates a parse tree by starting with the leaves (the annotated words output by the scanner) and building up towards the goal node for the language's grammar.

- Thus, unlike in top-down parsing, where the parser focused on the *lower* frontier of the partially-built parse tree, a bottom-up parser focuses on the *upper* frontier.

- In particular a bottom-up parser tries to extend the upper frontier of the parse tree upwards by finding a substring β in the current frontier that matches the right-hand side of some production $A \rightarrow \beta$.

- If it finds such a substring, the parser can add the node A as the parent of all of the nodes in β.

- Thus, finding productions $A \rightarrow \beta$ to expand the parse tree upwards (while maintaining a parse tree that represents a valid derivation of the input sentence) is the main task of a bottom-up parser. This task is called ***handle finding***.

- A ***handle*** is specifically a pair $\langle A \rightarrow \beta, k \rangle$, where β is a substring in the upper frontier of the parse tree whose right end is at position k, and the replacement of β with A is the next step in a valid derivation of the input program.

- Once a handle is found, performing the replacement of β with A is known as a ***reduction***.

- A bottom-up parser repeats the process of finding a handle and performing the corresponding reduction until one of two things happens:
  - The upper frontier of the parse tree is reduced to a single node corresponding to the grammar's goal symbol. If this happens and the parser has also consumed all of the words in the input stream, then the parse succeeds.
  - The parser can't find a handle. In this case, the parse fails.

- Importantly, a valid derivation must start with the grammar's goal symbol and work towards the final sentence by applying a series of productions:

  $$goal = \gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \ldots \rightarrow \gamma_{n-1} \rightarrow \gamma_n = sentence$$

- A bottom-up parser discovers the steps of this derivation in reverse order. In other words, a bottom-up parser discovers $\gamma_i \rightarrow \gamma_{i+1}$ before it discovers $\gamma_{i-1} \rightarrow \gamma_i$.

- Because the scanner returns classified words in left-to-right order, a bottom-up parser must look for a *rightmost* derivation. In particular, in a *reverse* rightmost derivation, the leftmost word is considered first, since in a forward rightmost derivation, the leftmost leaf is considered last.

- When looking at bottom-up parsers, we will be interested in a particular class of grammars, called LR(1) grammars, for which a reverse rightmost derivation can be found efficiently using 1 word of lookahead.
  - The name LR(1) derives from the following properties of these parsers:
    - **L**eft-to-right scan of the source program
    - **R**ightmost derivation (in reverse)
    - **1** word of lookahead

- LR(1) grammars are strictly more expressive than LL(1) grammars. In fact, the LL(1) grammars are a subset of the LR(1) grammars.

- Interestingly, while increasing the lookahead to some number *k* allows an LR(*k*) parser to recognize a larger set of *grammars* than an LR(1) parser, the set of *languages* that can be recognized by LR(*k*) parsers is the same as the set that can be recognized by LR(1) parsers.

- Formally determining whether a grammar is an LR(1) grammar is not straightforward. The easiest way to do this is to pass the grammar to an LR(1) parser generator (like Bison). If the generator fails, then the grammar is not LR(1).

- The most common bottom-up parsing technique is ***shift-reduce parsing***. We will explore this technique next.

Shift-reduce parsing

- A shift-reduce parser uses two main actions to move through the input and build a parse tree:
  - **Shift** – read one new input token from the scanner
  - **Reduce** – reduce an identified handle $\langle A \to \beta, k\rangle$ by replacing the substring β with the symbol *A*, as described above

- In addition to these two main actions, a shift-reduce parser can do the following two things:
  - **Accept** – accept the input string, signifying a successful parse
  - **Error** – reject the input string, signifying a failed parse

- The actions of a shift-reduce parser are centered around a stack, called the ***parse stack***, onto which grammar symbols are pushed.

- In particular, when a shift action is performed, the newly-read token (a terminal symbol in the grammar) is pushed onto the stack.

- When a reduce action is performed, the topmost symbols on the stack, which comprise the string β, are popped, and the symbol *A* is pushed onto the stack in their place.

- In this way, language constructs are recognized immediately, as soon as all of the symbols comprising a particular construct are encountered (i.e. when they are on at the top of the parse stack).

- To see more concretely how a shift-reduce parser works, let's consider shift-reduce parsing in the context of a particular grammar:

r0	*Goal*	→	*List*
r1	*List*	→	*List Pair*
r2		|	*Pair*
r3	Pair	→	(*Pair*)
r4		|	( )

- This grammar simply describes the language of properly-nested parentheses.

- Let's examine how a shift-reduce parser for this language would operate on the input string "`(())`". Below is the sequence of actions the parser would take, along with the current contents of the parse stack, the current lookahead symbol, and the remaining input string:

Iteration	**Parse stack**	**Lookahead**	**Remaining input**	**Action**
0		`(`	`())`	shift
1	`(`	`(`	`))`	shift
2	`((`	`)`	`)`	shift
3	`(()`	`)`	`)`	reduce (with r4)
4	`(` *Pair*	`)`	`)`	shift
5	`(` *Pair* `)`	`eof`		reduce (with r3)
6	*Pair*	`eof`		reduce (with r2)
7	*List*	`eof`		accept

- As you might suspect, a few details are left out here, so questions arise, such as:
    - How does the parser know, for a given lookahead and parse stack, whether to shift, reduce, accept, or error?
        - For example, at iteration 4, how did the parser know to shift instead of reducing *Pair* to *List* using rule r1?
    - Given that a stack only provides efficient access to its top element, how can a shift-reduce parser operate efficiently?
        - For languages with larger constructs than the ones in our parentheses grammar, it could be quite expensive to delve into the stack at every iteration to check what symbols are below the top.

- In practice, a shift-reduce parser is an automaton, similar to the finite automata we saw when exploring regular expressions and scanners.

- In particular, a shift-reduce parser always exists in a specific state, known as the ***parser state***, which encodes information about the current contents of the parse stack and the actions the parser has so far taken. The parser's various actions cause it to transition between states.

- Importantly, for an LR(1) grammar, the combination of the current parser state and the current lookahead symbol uniquely determine the next action to be taken by the parser. The result of this action, in turn, determines the next parser state.

- For instance, in the example parse above, the parser would enter iteration 4 in a particular parser state that might encode information such as the fact that an unmatched left parenthesis exists on the parse stack.

- Given this parser state and the lookahead character "`(`", the parser's correct action is to shift the "`(`" character (and by doing so enter a new parser state) instead of reducing *Pair* to *List*.

- Because the parser state encodes information about the contents of the parse stack beneath its top, it also prevents the parser from needing to explicitly examine those contents, thereby allowing the parser to operate efficiently.

- Most shift-reduce parsers use tables to encode information about the parsing automaton, for example mapping lookahead and parser state combinations to actions and specifying state transitions.

The shift-reduce parse tables

- The shift-reduce parsing algorithm is driven by two tables, `Action` and `Goto`, which encode the parsing automaton and are referred to as the ***parse tables***.

- The `Action` table encodes the action the parser should take (i.e. shift, reduce, accept, error) for each combination of lookahead symbol and parser state.

- When the `Action` table specifies a shift action for some combination of lookahead and parser state, it includes transition information specifying the next parser state. However, when reduce actions in the `Action` table do not include transition information.

- To understand the `Goto` table, it's important to understand that a working shift-reduce parser not only pushes grammar symbols onto the parse stack, it also pushes parser states.

- In particular, every time a symbol is pushed onto the parse stack, the current parser state is pushed onto the stack on top of the symbol.

- In other words, grammar symbols and parser states are interleaved on the parse stack, with the current parser state always living at the top of the stack.

- For example, entering iteration 4 of our example parse above, the parse stack might actually look something like this, where each s_i is a parser state:

  s_0 (s_t *Pair* s_u

- Here, s_u is the current parser state. When the parser performs a shift action to push the) lookahead token onto the parse stack, it will also push the new current parser state, say s_v, onto the stack:

  s_0 (s_t *Pair* s_u) s_v

- By pushing parser states onto the parse stack along with symbols, a shift-reduce parser keeps track of the ***left context***—the grammar symbols seen so far—under which each symbol was added to the stack.

- In the example above, s_0 represents the left context when the left parenthesis (was pushed onto the call stack, and s_t represents the left context when the *Pair* was pushed onto the call stack.

- When the parser performs a reduce action with grammar rule $A \rightarrow \beta$, it pops all of the symbols in β (and their corresponding parser states) from the parse stack and replaces them with *A* by pushing *A* onto the stack.

- Before pushing *A* onto the parse stack, the parser checks what state is at the top of the stack after popping the symbols from β. This state represents the left context under which β was encountered and pushed onto the parse stack.

- Continuing with the example above, when the parser performs a reduction using the grammar rule *Pair* → (*Pair*) in iteration 5, it pops all of the symbols and states corresponding to the right-hand side of that rule from the parse stack, leaving s_0 at the top of the stack, representing the left context under which the (*Pair*) was encountered.

- In order to decide what state to transition to after performing this reduction, the parser refers to the `Goto` table, which maps combinations of parser states and *nonterminals* (i.e. the left-hand sides of grammar rules) to new parser states.

- In other words, the `Goto` table encodes state transition information used to determine the next parser state after performing a reduce action.

- In particular, after performing a reduction using the rule $A \rightarrow \beta$ and encountering state s_i on the parse stack as the left context under which β was encountered, the next parser state is found in the entry for A and s_i in the `Goto` table.

- Once this next state is known, it is pushed onto the top of the parse stack after the symbol A.

- The construction of the parse tables is typically automated, since they are very challenging to construct by hand.
    - We will explore an algorithm for constructing these tables in a bit.

The shift-reduce parsing algorithm

- Given our understanding of the basic mechanics of a shift-reduce parser and the purpose of the parse tables, we can put together a complete listing of the shift-reduce parsing algorithm (assuming the existence of the `Action` and `Goto` tables and the availability of the scanner function, `nextWord()`):

```
push start state s_0 onto empty stack
lookahead = nextWord()
while true:
     state = top of stack
     if Action[state, lookahead] = "reduce $A \rightarrow \beta$":
          pop 2*|$\beta$| symbols from stack
          context = top of stack
          push $A$ onto stack
          push Goto[context, $A$] onto stack
     else if Action[state, lookahead] = "shift s_i":
          push lookahead onto stack
          push s_i onto stack
          lookahead = nextWord()
     else if Action[state, lookahead] = "accept":
          return SUCCESS
     else:
          return FAILURE
```

- Again, we'll look at how the parse tables are constructed in a bit, but for now, here are the parse tables for the parentheses grammar above:

State #	**`Action` table**			**`Goto` table**	
	`eof`	`(`	`)`	*List*	*Pair*
0	–	shift 3	–	1	2
1	accept	shift 3	–	–	4
2	reduce r2	reduce r2	–	–	–
3	–	shift 6	shift 7	–	5
4	reduce r1	reduce r1	–	–	–
5	–	–	shift 8	–	–
6	–	shift 6	shift 10	–	9
7	reduce r4	reduce r4	–	–	–
8	reduce r3	reduce r3	–	–	–
9	–	–	shift 11	–	–
10	–	–	reduce r4	–	–
11	–	–	reduce r3	–	–

- Here, the values for the shift actions are parser states representing state transitions, the values for reduce actions refer to grammar rules, and the values in the `Goto` table are also parser states representing state transitions. In both tables, the entry value "–" corresponds to an error: the corresponding symbol is invalid if the parser is in the designated state.

- Given the complete parsing algorithm and these parse tables, we can reexamine the parsing of the string "`(())`" from above:

Iteration	Parse stack	Parser state	Lookahead	Remaining input	Action
0	0	0	`(`	`())`	shift 3
1	0 `(` 3	3	`(`	`))`	shift 6
2	0 `(` 3 `(` 6	6	`)`	`)`	shift 10
3	0 `(` 3 `(` 6 `)` 10	10	`)`	`)`	reduce r4
4	0 `(` 3 *Pair* 5	5	`)`	`)`	shift 8
5	0 `(` 3 *Pair* 5 `)` 8	8	`eof`		reduce r3
6	0 *Pair* 2	2	`eof`		reduce r2
7	0 *List* 1	1	`eof`		accept

- By considering this sequence, we can start have an intuitive sense of what some of the states encode. For example, state 3 appears to represent a state in which one unmatched left parenthesis exists on the parse stack, while state 6 appears to represent a state in which more than one unmatched left parentheses exists.

- Our main remaining question, then, is how to construct the parse tables. To understand how to do this, we first need to understand how the table construction algorithm represents parser states and transitions.

LR(1) Items

- Remember that a shift-reduce parser is actually a handle-finding automaton. In other words, it consists of a set of states with transitions between those states.

- These states and transitions are based on handles and potential handles (i.e. handles in progress), along with their associated lookahead symbols. In particular, we can think of each of the parsing automaton's states as representing a set of potential (or completed) handles.

- Every time the parser takes a particular action and/or sees a new lookahead symbol, the set of handles it might eventually be able to complete using the symbols on the parse stack—and the parser's progress towards completing each one of those handles— changes. These changes correspond to the transitions between parser states.

- Thus, in order to construct the parse tables, which encapsulate all of the information about the parser automaton's states and transitions, we need to be able to represent potential handles in progress. We will do this using ***LR(1) items***.

- An LR(1) item is denoted as follows:

  [$A \rightarrow \beta \bullet \gamma$, `a`]

- The components of an LR(1) item are as follows:
  - $A \rightarrow \beta\ \gamma$ is a production in the grammar.
  - • is a placeholder that represents the parser's progress towards recognizing the right-hand side of that production.
    - In other words, the symbols to the left of the • represent the symbols currently at the top of the parse stack, and the symbols to the right of the • represent the remaining symbols that would need to be encountered to use this production as a handle.
  - `a` is a terminal symbol representing legal right context for an *A*.

- Based on the position of the placeholder, an LR(1) item can take three possible forms:
  - A ***possibility*** – [$A \rightarrow \bullet \beta\ \gamma$, `a`] – this indicates that an *A* would be valid in the parser's current state and that recognizing a β would be the first step towards recognizing an *A*.
  - A ***partially-complete item*** – [$A \rightarrow \beta \bullet \gamma$, `a`] – this indicates that the parser has progressed from [$A \rightarrow \bullet \beta\ \gamma$, `a`] by recognizing a β. The next step towards recognizing an *A* would be to recognize a γ.
  - A ***complete item*** – [$A \rightarrow \beta\ \gamma \bullet$, `a`] – this indicates that the parser has recognized $\beta\ \gamma$ in a context where an *A* followed by an `a` would be valid. If the lookahead symbol is an `a`, then the item represents a valid handle, and the parser can reduce $\beta\ \gamma$ to *A*.

- There are many LR(1) items for our parentheses grammar, including:

  [*Goal* → • *List,* `eof`]
  [*Goal* → *List* •*,* `eof`]
  [*List* → *List* • *Pair*, `(`]
  [*Pair* → `(` • `)` , `)`]

- The of these items, [*Goal* → • *List,* `eof`], represents the initial state of the parser, looking for a string that can reduce to *Goal* followed by `eof`. Every parse begins in this state.

- The second of the items above, [*Goal* → *List* •*,* `eof`], represents the desired final state of the parser, having recognized a string that reduces to *Goal* followed by `eof`. This represents a successful parse.

The canonical collection of sets of LR(1) items

- In order to fill in the parse tables, an LR(1) parser generator builds a model of the handle-finding automaton called the ***canonical collection of sets of LR(1) items*** (or just ***canonical collection*** for short).

- As its name implies, the canonical collection is a collection of sets of LR(1) items, where each set corresponds to a state in the handle-finding automaton. The entire collection represents all of the possible states of the automaton.

- In the course of constructing the canonical collection, the parser generator also records transitions between sets in the collection in order to model the handle-finding automaton's state transitions.

- The construction of the canonical collection, then, is the key to generating the LR(1) parse tables.

- The algorithm that constructs the canonical collection begins with the parser's initial state [*Goal* → • *S,* `eof`] (in the case of the parentheses grammar, [*Goal* → • *List,* `eof`]). It then uses two main operations, taking a closure and computing a transition, to iteratively generate all of the parser automaton's states and transitions.

- We'll examine each of these two operations separately.

The closure operation

- The purpose of the closure operation is to complete an automaton state—that is, some set of LR(1) items—by computing all LR(1) items implied by some core set.

- For example, in the parentheses grammar, anywhere the production *Goal* → *List* is legal, any productions that derive a *List* are legal as well. Thus, the LR(1) item [*Goal* → • *List,* `eof`] implies both [*List* → • *List Pair,* `eof`] and [*List* → • *Pair,* `eof`].

- More generally, the closure operation iterates over a set *s* of LR(1) items. For each item in the set, if the placeholder • precedes a nonterminal *C*, then the closure operation adds to *s* an LR(1) item for each grammar production that derives *C*.
    - In each of these items, the placeholder • is placed at the beginning of the right-hand side of the production.

- This process continues until *s* is no longer changing.

- Here is a pseudocode specification for the closure operation, which uses the FIRST sets from our examination of top-down parsing:

```
closure($s$):
    while $s$ is still changing:
        for each item [$A \to \beta \bullet C\, \delta$, a] in $s$:
            for each production $C \to \gamma$:
                for each b in FIRST($\delta$a):
                    $s$ = $s \cup \{[C \to \bullet\, \gamma$, b]$\}$
    return $s$
```

- The idea here is straightforward: if [$A \to \beta \bullet C\ \delta$, a] is in *s*, then recognizing a *C* followed by a δa should result in a reduction to *A*. Thus, the `closure` procedure simply inserts into *s* all possible LR(1) items that represent a way to recognize a *C* followed (potentially) by δa.
    - Specifically, every terminal in FIRST(δa) is one that could legally follow a *C* in this situation.

- Going back to the example of the parentheses grammar, we can compute the initial state of the parsing automaton by computing the closure of the set containing only the initial item [*Goal* → • *List,* `eof`]. This adds these 8 items:

    [*List* → • *List Pair,* `eof`] [*List* → • *Pair,* `eof`] [*List* → • *List Pair,* `(`]
    [*List* → • *Pair,* `(`] [*Pair* → • `(` *Pair* `)` , `eof`] [*Pair* → • `( )` , `eof`]
    [*Pair* → • `(` *Pair* `)` , `(`] [*Pair* → • `( )` , `(`]

- This total set of 9 items represents the initial state of the parsing automaton and forms the first set in the canonical collection. We will refer to it as CC_0.

The transition operation

- In other words, the purpose of this operation is to compute the state to which the parsing automaton would transition after seeing a recognizing a specific grammar symbol in a given state represented by a set CC_i from the currently-built canonical collection.

- More generally, the transition operation takes as input a set *s* of LR(1) items and a grammar symbol *x* (terminal or nonterminal), and iterates over the items in *s*. Each time it finds the placeholder • immediately preceding *x* in an item in *s*, it creates a new item by moving the • past *x*.

- Each new item produced in this way is placed into a new set, and after all of the new items are computed, the closure operation is run on the new set to complete the new parser state.

- Here is a pseudocode listing for the transition operation:

```
goto(s, x):
    moved = {}
    for each item i in s:
        if i has the form [$A \rightarrow \beta \bullet x\, \delta$, a]:
            moved = moved ∪ {[$A \rightarrow \beta\, x \bullet \delta$, a]}
    return closure(moved)
```

- Again, the idea here is straightforward. The procedure simply simulates recognizing an *x* by moving the placeholder past *x* each time it immediately precedes *x*.

- Again going back to the example of the parentheses grammar, we can compute the parser state that would result from recognizing an initial `(` symbol by computing `goto(`CC_0`, ()`. The `goto` procedure's loop would generate the following 4 items that result from moving the placeholder • past a `(`:

  [$Pair \rightarrow$ (• $Pair$) , eof] [$Pair \rightarrow$ (•) , eof]
  [$Pair \rightarrow$ (• $Pair$) , (] [$Pair \rightarrow$ (•) , (]

- Running the closure operation on this set would produce an additional two items:

  $[Pair \rightarrow \bullet\ (\ Pair\ )\ ,\ )]\ [Pair \rightarrow \bullet\ (\ )\ ,\ )]$

- The total set of these 6 items would form a new set in the canonical collection and would represent a state in the parser automaton.

Building the canonical collection

- Given the `closure` and `goto` procedures, the algorithm for building the canonical collection is straightforward. It starts with the initial parsing automaton state CC_0 and continually generates new states by using `goto` to compute transitions out of existing states. This process is repeated until no additional states can be generated.

- Here is a pseudocode listing for the canonical construction algorithm:

```
$CC_0$ = closure([$Goal \rightarrow \bullet S$, eof])
CC = { $CC_0$ }
while new sets are being added to CC:
    for each unmarked set $CC_i$ in CC:
        mark $CC_i$ as processed
        for each $x$ following the • in an item in $CC_i$:
            tmp = goto($CC_i$, $x$)
            if tmp is not in CC:
                CC = CC ∪ tmp
            record transition from $CC_i$ to tmp on $x$
```

Filling the parse tables

- Once the canonical collection is built (and transition between states recorded), we can fill in the entries of the `Action` and `Goto` tables.

- In particular, to construct the parse tables, each CC_i in the canonical collection constructed by the algorithm above becomes a parser state, the items in CC_i generate entries in the `Action` table, and the transitions recorded during the construction of the canonical collection specify entries in the `Goto` table.

- When generating entries of the `Action` table, three different cases can arise:
    - An item of the form $[A \rightarrow \beta \bullet \texttt{c}\ \delta, \texttt{a}]$, where `c` is a terminal and β or δ can be ϵ, generates a shift action on `c`, since the item indicates that `c` would be a valid next step towards recognizing an *A*. The next parser state after the shift action is the result of computing `goto` on the current state with the terminal `c`.
    - An item of the form $[A \rightarrow \beta \bullet, \texttt{a}]$ generates a reduce action for the production $A \rightarrow \beta$ on the lookahead `a`, since the item indicates that the parser has recognized a handle, provided that the lookahead symbol is `a`.
    - An item of the form $[Goal \rightarrow \text{S} \bullet, \texttt{eof}]$ generates an accept action for the lookahead `eof`, since the item indicates the accepting state for the parser.

- Here is a pseudocode listing of the table filling algorithm:

```
for each $CC_i$ in CC:
    for each item $I$ in $CC_i$:
        if $I$ is $[A \rightarrow \beta \bullet \texttt{c}\ \delta, \texttt{a}]$ and goto($CC_i$, c) == $CC_j$:
            Action[$i$,c] = "shift $j$"
        else if $I$ is $[A \rightarrow \beta \bullet, \texttt{a}]$:
            Action[$i$,a] = "reduce $A \rightarrow \beta$"
        else if $I$ is $[Goal \rightarrow \text{S} \bullet, \texttt{eof}]$:
            Action[$i$,eof] = "accept"
    for each nonterminal $n$:
        if goto($CC_i$, c) == $CC_j$:
            Goto[$i$,$n$] = $j$
```

- Note that the table filling algorithm does not generate entries for items in which the placeholder • is before a nonterminal. While such items do not generate table entries, they do influence the final tables, since they force the `closure` procedure to include additional items that *do* generate table entries.

- It is during the table-filling algorithm that it becomes clear if a grammar is not an LR(1) grammar. In particular, if two LR(1) items generate different entries for the same location in the table, then the grammar contains an ambiguity that prevents it from being LR(1).
    - In particular, an error in which one item produces a shift action and another produces a reduce action for the same table entry is known as a ***shift-reduce conflict***.

    ○ An error in which two items produce a different reduce action for the same table entry is known as a ***reduce-reduce conflict***.

Table construction for the parentheses grammar

- Let's trace through a few steps of the canonical collection and table construction algorithms for the parentheses grammar.

- We've already computed the initial parser state, CC_0:

  [*Goal* → • *List,* `eof`] [*List* → • *List Pair,* `eof`] [*List* → • *Pair,* `eof`]
  [*List* → • *List Pair,* `(`] [*List* → • *Pair,* `(`] [*Pair* → • `(` *Pair* `)` , `eof`]
  [*Pair* → • `( )` , `eof`] [*Pair* → • `(` *Pair* `)` , `(`] [*Pair* → • `( )` , `(`]

- From CC_0, the canonical collection construction algorithm produces three different sets of items. The first, CC_1, is computed by `goto(`CC_0`,` *List*`)`:

  [*Goal* → *List* •, `eof`] [*List* → *List* • *Pair,* `eof`] [*List* → *List* • *Pair,* `(`]
  [*Pair* → • `(` *Pair* `)` , `eof`] [*Pair* → • `( )` , `eof`] [*Pair* → • `(` *Pair* `)` , `(`]
  [*Pair* → • `( )` , `(`]

- Next, CC_2 is computed by `goto(`CC_0`,` *Pair*`)`:

  [*List* → *Pair* •, `eof`] [*List* → *Pair* •, `(`]

- We computed CC_3 above when we examined the behavior of the goto operation by computing `goto(`CC_0`, ()`:

  [*Pair* → `(` • *Pair* `)` , `eof`] [*Pair* → `(` • `)` , `eof`] [*Pair* → `(` • *Pair* `)` , `(`]
  [*Pair* → `(` • `)` , `(`] [*Pair* → • `(` *Pair* `)` , `)`] [*Pair* → • `( )` , `)`]

- We can proceed with CC_1, from which we compute CC_4 as `goto(`CC_1`,` *Pair*`)`:

  [*List* → *List Pair* •, `eof`] [*List* → *List Pair* •, `(`]

- Interestingly, `goto(`CC_1`, ()` yields a set identical to CC_3, so we do not generate a new set for this combination.

- Now that we've computed several sets in the canonical collection (noting the transitions between them), we can begin to look at the table filling algorithm.

- We'll start with CC_0. For this set, we have no items that generate reduce actions and none that generate accept actions. However, there are several items of the form [$A \rightarrow \bullet$ `(` δ, `a`], and we have `goto(`CC_0`, ()` = CC_3. These result in an action "shift 3" being generated in the `Action` table for state 0 and lookahead `(`.

- In addition, we have `goto(`CC_0`, `*List*`)` = CC_1, which results in state 1 being entered in the `Goto` table under state 0 and nonterminal *List*. We also have `goto(`CC_0`, `*Pair*`)` = CC_2, which results in state 2 being entered in the `Goto` table under state 0 and nonterminal *Pair*.

- These three entries for state 0 (one in the `Action` table and two in the `Goto` table) match the entries in the parse tables we saw for the parentheses grammar above.

- We can also compute the parse table entries for CC_1. Interestingly, for this set, we have an item [$Goal \rightarrow List \bullet$, `eof`] that generates an "accept" action in the `Action` table for state 1 and lookahead `eof`.

- No items in CC_1 generate reduce actions. Again, though we have several items of the form [$A \rightarrow \bullet$ `(` δ, `a`], and we have `goto(`CC_1`, ()` = CC_3. These result in an action "shift 3" being generated in the `Action` table for state 1 and lookahead `(`.

- Finally, we have `goto(`CC_1`, `*Pair*`)` = CC_4, which results in state 4 being entered in the `Goto` table under state 1 and nonterminal *Pair*.

- Again, all of these entries for state 1 match the entries in the parse tables we saw above.

- These algorithms would continue to run, producing all 11 states and their corresponding entries in the parse tables, as we saw them above.