Bottom-up parsing

Goal
Given a grammar $G$, construct a parse tree for string $w$ by starting at the leaves and working to the root.

Strategy
- construct a rightmost derivation, in reverse:
$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$$
- For each right-sentential form $\gamma_0 \cdots \gamma_n$:
  - pick a production $A \rightarrow \alpha$
  - replace $\alpha$ with $A$

Table-driven, bottom-up parsing techniques
- general strategy: shift-reduce parsing (AS&U, §4.5)
- operator precedence parsers (we will not cover these) (AS&U, §4.6)
- LR parsers (AS&U, §4.7)

Finding reductions: An example

Consider the grammar:
\begin{align*}
1 & \quad \langle \text{goal} \rangle ::= a \langle A \rangle \langle B \rangle e \\
2 & \quad \langle A \rangle ::= \langle A \rangle b c \\
3 & \quad \mid b \\
4 & \quad \langle B \rangle ::= d
\end{align*}

Construct a rightmost derivation for input string $abbcde$:
$$\langle \text{goal} \rangle \Rightarrow a \langle A \rangle \langle B \rangle e \Rightarrow a \langle A \rangle de \Rightarrow a \langle A \rangle bcde \Rightarrow abbcde$$

Next Reduction | Production | Position
--- | --- | ---
$abbcde$ | 2 | 2
$a(A)bde$ | 4 | 4
$a(A)b$ | 1 | 4
$(\text{goal})$ | — | —

Handle

Definition
A handle of a right-sentential form $\gamma$ is a pair $(\alpha \rightarrow \beta, k)$ where:
- $\alpha \rightarrow \beta \in P$
- $k$ is the position in $\gamma$ of $\beta$'s rightmost symbol
- replacing $\beta$ with $\alpha$ at position $k$ produces the right-sentential form that preceded $\gamma$ in the rightmost derivation.

Properties
- Because $\gamma$ is a right-sentential form, the substring to the right of a handle contains only terminal symbols.
- $\Rightarrow$ we don't need to scan past the handle (very far)
- If $G$ is unambiguous, then every right-sentential form has a unique handle.

Uniqueness of handles

Theorem
If $G$ is unambiguous, then every right-sentential form has a unique handle.

Sketch of proof
Proof just follows from definitions:
- If $G$ is unambiguous
  - $\Rightarrow$ rightmost derivation is unique.
  - $\Rightarrow$ a unique production $\alpha \rightarrow \beta$ applied to $\gamma_{n-1}$ to $\gamma_n$ and a unique position $k$ at which $\alpha \rightarrow \beta$ is applied
  - $\Rightarrow$ a unique handle $(\alpha \rightarrow \beta, k)$.
A Running Example Grammar

This is a left-recursive expression grammar:

1. \( \text{goal} \rightarrow \text{expr} \)
2. \( \text{expr} \rightarrow \text{expr} + \text{term} \)
3. \( \text{term} \rightarrow \text{term} - \text{factor} \)
4. \( \text{factor} \rightarrow \text{factor} * \text{factor} \)
5. \( \text{factor} \rightarrow \text{factor} / \text{factor} \)
6. \( \text{factor} \rightarrow \text{num} \)
7. \( \text{factor} \rightarrow \text{id} \)

An Example Parse

Example Expression

\[ x - 2 \times y \]

Sentential Form

\[ (\text{id}, x) - (\text{num}, 2) \times (\text{id}, y) \]

Handle pruning

Handle Pruning

The process of finding a handle and reducing it to the appropriate left-hand side.

Informal overview

To construct a rightmost derivation

\[ S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = \omega, \]

apply the following algorithm:

do \( i = n \) to \( 1 \) by \(-1\)

1. find the handle \( \alpha_i \rightarrow \beta_i \) in \( \gamma_i \)
2. replace \( \beta_i \) with \( \alpha_i \) to generate \( \gamma_{i-1} \)

Key Challenge

Key is to find a handle efficiently. This has two parts:

- Find substring to be reduced: \( \beta_i \)
- Decide which production to use: \( \alpha_i \rightarrow \beta_i \)

Shift-Reduce Parsing

One implementation of a handle-pruning, bottom-up parser is the shift-reduce parser. Shift-reduce parsers require a stack and an input buffer.

The algorithm

1. push `$` onto the stack
2. \( \text{token} \leftarrow \text{next_token}() \)
3. repeat until (top of stack = \text{goal} & \text{token} = \text{eof})
4. if we have a handle \( \alpha \rightarrow \beta \) on top of the stack then reduce \( \beta \) to \( \alpha \)
5. pop \( \beta \) symbols off the stack
6. push \( \alpha \) onto the stack
7. else shift
8. shift \( \text{token} \) onto the stack
9. \( \text{token} \leftarrow \text{next_token}() \)

The parser must also recognize syntax errors.
Why is a stack sufficient?

Claim:
Handle will always appear at the top of the stack.

Why?
Because we construct a rightmost derivation (in reverse).

Sketch of proof:

- **Base case:** first handle to be reduced
  - shift tokens until handle appears at top of stack; reduce

- **Inductive step:** Assume that handle for \( k \)th reduction is at top of stack.
  - After reduce, new non-terminal (say \( A \)) is on top of stack
    - “Rightmost” derivation → next handle cannot end to the left of \( A \) (i.e. below top of stack)
  - Shift zero or more input symbols to obtain next handle at top-of-stack

See AS&U, § 4.5 for more formal version of this argument

Actions of Shift-Reduce Parsing

- **Shift-reduce parsers are easily built and easily understood**
- **We make it a little more complicated to handle errors**

4 Actions of a S-R Parser

1. **shift** — next input symbol is shifted onto the top of the stack
2. **reduce** — right end of handle is on top of stack; locate left end of handle within the stack; pop handle off stack and push appropriate non-terminal \( \text{lhs} \)
3. **accept** — terminate parsing and signal success
4. **error** — call an error recovery routine

Cost
Actions 3 & 4 are simple
Action 1 is a push and a call to the scanner
Action 2 takes \( \text{rhs} \) pops and 1 push

What can go wrong?

**Conflicts**

- Failure during parser construction. 2 possible reasons:
  1. 2.

**Shift/Reduce Conflicts**

- Usually due to ambiguous grammar
- **Option 1:** modify the grammar to eliminate the conflict
- **Option 2:** resolve in favor of shifting
- classic examples: “dangling else” ambiguity, insufficient associativity or precedence rules
Conflicts (continued)

Reduce/Reduce Conflicts
- Often, no simple resolution
- Option 1: try to redesign grammar, perhaps with changes to language
- Option 2: use context information during parse (e.g., symbol table)
- Classic real example: PL/1 call and subscript: id(id, id)

When Stack = ...id ( id, input = id)...
  ▶ Reduce by expr → id, or
  ▶ Reduce by param → id

Shift/reduce conflict

Example
The dangling-else ambiguity:

Abbreviate as:

The conflict
Consider the input: i i a e a.
After shifting i, i, a and reducing by $S \rightarrow a$, we get:

stack = [iS], next token = e.

Q. On token e, what action should we take?
  ▶ Shift e
  ▶ Reduce by $S \rightarrow iS$

Solution for the Example
Assume: Prefer to associate else with innermost if
  ▶ disambiguating rule: prefer shift over reduce
  ▶ if (E) { if (E) a else a }

The role of precedence and associativity

Conflict-resolution rules
- Precedence and associativity rules can be used to resolve shift/reduce conflicts in ambiguous grammars:
  ▶ lookahead with higher precedence → shift
  ▶ same precedence, left associative → reduce
  ▶ alternative to encoding them in the grammar

Advantages
- more concise, albeit ambiguous, grammars
- shallower parse trees ⇒ fewer reductions

A simpler expression grammar


LR(1) grammars

Informal definition

A grammar $G$ is LR(1) if, given a rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_n = w,$$

we can, for each right-sentential form in the derivation,
- isolate the handle of each right-sentential form,
- determine the production by which to reduce
by scanning $\gamma_i$ from left to right, going at most 1 symbol beyond the right end of the handle of $\gamma_i$.

Complexity
- one reduction per step in derivation
- one handle discovery per reduction

Key goal: Recognizing handles efficiently.

Why study LR(1) grammars?

LR(1) grammars are widely used to construct parsers
These parsers are flexible & efficient

Tools to build LR(1) parsers are widely available

- Virtually all context-free programming language constructs can be expressed in an LR(1) form
- LR grammars are the most general grammars parsable by a non-backtracking, shift-reduce parser — deterministic CFGs
- Efficient parsers can be implemented for LR(1) grammars — time proportional to tokens + reductions
- LR parsers detect an error as soon as possible in left-to-right scan of input
- LR grammars describe a proper superset of the languages recognized by predictive parsers

LR(1) is a beautiful example of applying sophisticated theory to develop easy-to-use tools for a complex problem

Table-driven LR(1) parsing

A table-driven LR(1) parser looks like:

```
source code
```

```
grammar
```

```
parser
```

```
table-driven parser
```

```
actions & goto tables
```

```
stack
```

```
scanner
```

```
 parser generator
```

The LR parser stack

**Differences from Shift-Reduce stack**

Stack two items per symbol: symbol and state. If shift-reduce stack contains:

$$X_1 X_2 \cdots X_{n-1} X_n$$

then LR parser stack contains:

$$X_1 S_1 X_2 S_2 \cdots X_{n-1} S_{n-1} X_n S_n$$

**Stack operations**

Let: $S_{\text{stack}}$ = $X_1 S_1 X_2 S_2 \cdots X_{n-1} S_{n-1} X_n S_n$

**Shift $S_{\text{stack}}$**:

The stack becomes

$$X_1 S_1 X_2 S_2 \cdots X_{n-1} S_{n-1} X_n S_n a_1 \text{ $S_{\text{new}}$}$$

**Reduce $A \rightarrow \beta$**:

The stack becomes

$$X_1 S_1 X_2 S_2 \cdots X_{n-1} S_{n-1} S_n a_1 S_{\text{new}}$$

where $S_{\text{new}} = \text{goto}[S_{n-1}, A]$. 


A fundamental theorem of LR parsing

**Theorem:** If a handle can be recognized by reading the symbols on stack, then a finite-state machine is sufficient to do so!

**Why?**

→ each handle contains the rhs of some production  
→ set of handles is finite  
→ handle position is made stack-relative  

State $S_i$ on LR parser stack is the state the FSM would be in if it read symbols $X_0 \ldots X_i$.

$\texttt{goto}[S_i,X_i]$ is the state transition function for the FSM.
The Goal

We want to use a state machine to handle the LR parse for us.

Consider the simple grammar $A \rightarrow x \ y$. The resulting state machine is:

These states correspond to different stages of the production.

0. $A \rightarrow \bullet x \ y$
1. $A \rightarrow x \ \bullet y$

The “•” is called “dot” or “the cursor”.

Each entry $A \rightarrow \alpha$ with a • somewhere in $\alpha$ is called an LR(0) item.

A state is a set of LR(0) items.

Multiple Productions

Consider the grammar

$A \rightarrow x \ y$
$A \rightarrow z$

To start, place the cursor at the beginning of the $A$ productions. This represents the beginning when we’ve received no input. You need to include both productions here, since we don’t know which of the two productions we will use.

Consider the grammar

$A \rightarrow x \ y$
$A \rightarrow z$

0. $A \rightarrow \bullet x \ y$
1. $A \rightarrow x \ \bullet y$
2. $A \rightarrow z \ \bullet$

If you are in state 0 and input an $x$, you will advance the cursor past the $x$ to get state 1.

If you are in state 0 and input a $z$, you will advance the cursor past the $z$ to get state 2.
Consider the grammar

\[
A \rightarrow x y \\
A \rightarrow z
\]

0. \(A \rightarrow \cdot x y\)  
\(A \rightarrow \cdot z\)
1. \(A \rightarrow x \cdot y \rightharpoonup\)
2. \(A \rightarrow z \cdot\)
3. \(A \rightarrow x y \cdot\)

If you are in state 1 and input a \(y\), you will advance the cursor past the \(y\) to get state 3.

These last two states are already complete, so no new states are formed.

Consider the grammar

\[
A \rightarrow x y \\
A \rightarrow q \\
A \rightarrow x z
\]

0. \(A \rightarrow \cdot x y\)  
\(A \rightarrow \cdot q\)  
\(A \rightarrow \cdot x z\)
1. \(A \rightarrow x \cdot y\)
2. \(A \rightarrow z \cdot\)
3. \(A \rightarrow x y \cdot\)

To start, copy all the \(A\) productions and place the cursor in front.

The transition from state 0 to 1 causes two productions to move: when we read \(x\), we could be parsing either \(xy\) or \(xz\).
Consider the grammar

\[
A \rightarrow xy \\
A \rightarrow q \\
A \rightarrow xz
\]

0. \( A \rightarrow \bullet xy \)  
1. \( A \rightarrow x \bullet y \)  
2. \( A \rightarrow q \bullet \)  
3. \( A \rightarrow \bullet xz \)

If we are in state 0, reading a \( q \) brings us to state 2.

Consider the grammar

\[
A \rightarrow xy \\
A \rightarrow q \\
A \rightarrow xz
\]

0. \( A \rightarrow \bullet xy \)  
1. \( A \rightarrow x \bullet y \)  
2. \( A \rightarrow q \bullet \)  
3. \( A \rightarrow \bullet xz \)

If we are in state 1, reading a \( y \) brings us to state 3.

Consider the grammar

\[
A \rightarrow xy \\
A \rightarrow q \\
A \rightarrow xz
\]

0. \( A \rightarrow \bullet xy \)  
1. \( A \rightarrow x \bullet y \)  
2. \( A \rightarrow q \bullet \)  
3. \( A \rightarrow \bullet xz \)

If we are in state 1, reading a \( z \) brings us to state 4.

None of the remaining states are expecting input.
Consider the grammar
\[ A \rightarrow x \ y \ A \]
\[ A \rightarrow z \]

0. \[ A \rightarrow \bullet \ x \ y \ A \leftarrow \]
\[ A \rightarrow \bullet \ z \]
1. \[ A \rightarrow x \ \bullet \ y \ A \]

State 0: Shift the \( x \) to make state 1.

Consider the grammar
\[ A \rightarrow x \ y \ A \]
\[ A \rightarrow z \]

0. \[ A \rightarrow \bullet \ x \ y \ A \leftarrow \]
\[ A \rightarrow \bullet \ z \]
1. \[ A \rightarrow x \ \bullet \ y \ A \]
2. \[ A \rightarrow z \ \bullet \]

State 0: Shift the \( z \) to make state 2.

Because the cursor is in front of an \( \lambda \) in state 3, we have to add the initial items for \( \lambda \) again. This operation is known as taking the closure of the state.
Consider the grammar

\[
A \rightarrow x \ y \ A \\
A \rightarrow z
\]

0. \( A \rightarrow \bullet \ x \ y \ A \)  \\
A \rightarrow \bullet \ z \\
1. \( A \rightarrow \bullet \ x \ y \ A \)  \\
2. \( A \rightarrow \bullet \ z \)  \\

State 2: No input expected.

Consider the grammar

\[
A \rightarrow x \ y \ A \\
A \rightarrow z
\]

0. \( A \rightarrow \bullet \ x \ y \ A \)  \\
A \rightarrow \bullet \ z \\
1. \( A \rightarrow \bullet \ x \ y \ A \)  \\
2. \( A \rightarrow \bullet \ z \)  \\
3. \( A \rightarrow \bullet \ x \ y \ A \)  \\
A \rightarrow \bullet \ z \\
4. \( A \rightarrow \bullet \ x \ y \ A \)  \\
A \rightarrow \bullet \ z \\

State 3: Shift the \( x \) to make state 4.

Consider the grammar

\[
A \rightarrow x \ y \ A \\
A \rightarrow z
\]

0. \( A \rightarrow \bullet \ x \ y \ A \)  \\
A \rightarrow \bullet \ z \\
1. \( A \rightarrow \bullet \ x \ y \ A \)  \\
2. \( A \rightarrow \bullet \ z \)  \\
3. \( A \rightarrow x \ y \ A \)  \\
A \rightarrow \bullet \ z \\
4. \( A \rightarrow x \ y \ A \)  \\
A \rightarrow \bullet \ z \\

State 3: Same situation with shifting \( z \), only we recycle state 2.
Being Several Places at Once

Consider the grammar:

\[ A \rightarrow x \ y \ A \]

\[ A \rightarrow z \]

1. \[ A \rightarrow x \ z \]
2. \[ A \rightarrow y \]
3. \[ A \rightarrow z \]
4. \[ A \rightarrow x \ y \]

State 0: no input expected. The automaton is complete.

Transitive Closures

\[ S \rightarrow x \ A \restriction q \]
\[ A \rightarrow B \ c \]
\[ B \rightarrow d \ A \restriction d \]

0. \[ S \rightarrow \bullet \ x \ A \]
1. \[ S \rightarrow \bullet \ c \]
2. \[ S \rightarrow \bullet \ d \]

State 0: shift \( q \) and take transitive closure to make state 1.

Transitive Closures

\[ S \rightarrow x \ A \restriction q \]
\[ A \rightarrow B \ c \]
\[ B \rightarrow d \ A \restriction d \]

0. \[ S \rightarrow \bullet \ x \ A \]
1. \[ S \rightarrow \bullet \ c \]
2. \[ S \rightarrow \bullet \ d \]

State 0: shift \( q \) and take transitive closure to make state 2.
Transitive Closures

\[
\begin{align*}
S &\rightarrow x A | q \\
A &\rightarrow B c \\
B &\rightarrow d A | d
\end{align*}
\]

State 1: shift \( A \) (actually, match \( A \)) to make state 3.

State 4: shift \( c \) to make state 6.

\[
\begin{align*}
0. &\quad S \rightarrow x A \\
1. &\quad S \rightarrow q \\
2. &\quad S \rightarrow x \bullet A \\
3. &\quad S \rightarrow x A \\
4. &\quad S \rightarrow q \bullet A
\end{align*}
\]

State 2: shift \( d \) to make state 5. A lot happens here!
Transitive Closures

\[ S \rightarrow x \mid q \]
\[ A \rightarrow B \mid c \]
\[ B \rightarrow d \mid d \]

1. \( S \rightarrow x \cdot A \)
2. \( S \rightarrow \cdot q \)
3. \( A \rightarrow B \cdot c \)
4. \( A \rightarrow B \cdot d \cdot A \)
5. \( B \rightarrow d \cdot A \cdot A \)

State 5: shift \( A \) to make state 7. The other shifts recycle. We are done.

Example

LR(1) items

The production \( \alpha \rightarrow \beta \gamma \delta \), with lookahead \( \alpha \) generates 4 LR(1) items

1. \( [\alpha \rightarrow \beta \gamma \delta, \alpha] \)
2. \( [\alpha \rightarrow \beta \gamma \delta, \epsilon] \)
3. \( [\alpha \rightarrow \beta \gamma \delta, \gamma] \)
4. \( [\alpha \rightarrow \beta \gamma \delta, \delta] \)

The \( \cdot \) indicates the position of the top of stack.

\( \alpha \rightarrow \beta \gamma \delta, \epsilon \) means that the input seen so far is consistent with the use of \( \alpha \rightarrow \beta \gamma \delta \) at this point in the parse.

\( \alpha \rightarrow \beta \gamma \delta, \gamma \) means that the input seen so far is consistent with the use of \( \alpha \rightarrow \beta \gamma \delta \), and the parser has already recognized \( \beta \gamma \).

\( \alpha \rightarrow \beta \gamma \delta, \delta \) means that the parser has seen \( \beta \gamma \delta \), and if next input token matches lookahead symbol \( \alpha \), then parser can reduce to \( \alpha \).

Lookahead component of LR(1) state

What does the lookahead component of state mean?

- Lookahead string is used to choose action when item has \( \cdot \) at right end
- Let stack \( \delta \gamma \) and let next token be \( a \neq EOF \)

\( \Rightarrow A \rightarrow \gamma \) is a handle only if there is a right-sentential form containing \( \delta \gamma \).

\( \Rightarrow \) State item \( [A \rightarrow \gamma \cdot, \alpha] \) indicates that \( a \) is acceptable when stack contains \( \delta \gamma \).

How is the lookahead component used?

1. For \( [\alpha \rightarrow \gamma \cdot, \alpha] \) and \( [\beta \rightarrow \cdot, \beta] \)
   - on \( \alpha \), reduce to \( \alpha \)
   - on \( \beta \), reduce to \( \beta \)

2. For \( [\alpha \rightarrow \gamma \cdot, \beta] \) and \( [\beta \rightarrow \cdot, \alpha] \)
   - on \( \alpha \), reduce to \( \alpha \)
   - on \( \beta \), reduce to \( \beta \)
   - else, for any \( b \in \text{FIRST}(\delta) \), shift

\( \Rightarrow \) Next symbol from input is enough to pick actions more precisely.
FIRST Sets for a Grammar

**Definition**

For a string of grammar symbols $\alpha$, define FIRST($\alpha$) as

- the set of terminal symbols that begin strings derived from $\alpha$

If $\alpha \Rightarrow^* \epsilon$, then $\epsilon \in$ FIRST($\alpha$)

FIRST($\alpha$) contains the set of tokens valid in the first position of $\alpha$

**Algorithm**

To build FIRST($X$):

1. If $X$ is a terminal, FIRST($X$) is \{X\}
2. If $X \rightarrow \epsilon$, then $\epsilon \in$ FIRST($X$)
3. If $X \rightarrow Y_1 \cdot Y_2 \cdot \ldots \cdot Y_k$ then put FIRST($Y_1$) in FIRST($X$) for all $1 \leq j < i$
   - (If $\epsilon \notin$ FIRST($Y_1$), then FIRST($Y_i$) is irrelevant, for $1 < i$)

Example: Grammar & FIRST sets

Grammar

1. goal $\rightarrow$ expr
2. expr $\rightarrow$ term $-$ expr
3. expr $\rightarrow$ term
4. term $\rightarrow$ factor $\cdot$ term
5. term $\rightarrow$ factor
6. factor $\rightarrow$ id

FIRST sets

<table>
<thead>
<tr>
<th>Symbol</th>
<th>FIRST</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal</td>
<td>{id}</td>
</tr>
<tr>
<td>expr</td>
<td>expr</td>
</tr>
<tr>
<td>term</td>
<td>term</td>
</tr>
<tr>
<td>factor</td>
<td>factor</td>
</tr>
<tr>
<td>id</td>
<td>+</td>
</tr>
</tbody>
</table>

Possible State Transitions

Consider a state containing an item $[A \rightarrow \alpha \cdot X \beta, a]$

- Push $X$ (token or NT) on the stack
  
  $[A \rightarrow \alpha \cdot X \beta, a] \xrightarrow{X} [A \rightarrow \alpha X \beta, a]$

  But if $X$ is a non-terminal, we can push $X$ on the stack only via some production $X \rightarrow \gamma$. So we need to look for strings that can be derived from $\gamma$

  $[A \rightarrow \alpha \cdot X \beta, a] \xrightarrow{X \rightarrow \gamma, b} [X \rightarrow \gamma, b] \forall b \in $ FIRST($\beta$),

  This says: $X$ generates $\gamma$ and then $\beta\gamma$ generates a string starting with $b$.

  Group above items into a single state, i.e., if a state contains item $[A \rightarrow \alpha \cdot X \beta, a]$, add items $[X \rightarrow \gamma, b]$ for all $X$ productions, and $\forall b \in $ FIRST($\beta$)

Computing the Closure

Algorithm to find “equivalent” item for a given set of items

The Closure Algorithm

```
Closure(s_i; set of items)
do
    changing ← false
    $\forall$ item $[A \rightarrow \alpha \cdot X \beta, a] \in I$
    $\forall$ production $X \rightarrow \gamma \in P$
    $\forall$ b $\in$ FIRST($\beta$)
    if $[X \rightarrow \gamma, b] \notin s_i$ then
        add $[X \rightarrow \gamma, b]$ to $s_i$
    changing ← true
while (changing)
```

Example

Q. What is $s_0 = \text{Closure}(\{[g \rightarrow \epsilon, \text{eof}]\})$ in the example grammar?
Computing the GOTO Function

**Algorithm for GOTO**

\[ \text{Goto}(s_n, x) \]

new \( \rightarrow \) \$

\forall \text{ items } i \in s_n \quad \text{ /* move the } * /$

\begin{aligned}
\text{if } i \in \{A \rightarrow \alpha \quad \epsilon \beta, A \rightarrow \alpha \}
\text{ then new } \leftarrow \text{ new } \cup \{A \rightarrow x \beta, A \rightarrow \alpha \}
\end{aligned}

\text{new } \leftarrow \text{ closure(new) } \quad \text{ /* make it a DFSM state */}

\text{return new}

**Complete LR(1) Table Construction Algorithm**

1. Build \( C \), the canonical collection of sets of \( \text{LR}(1) \) items
2. Iterate through \( C \), filling in \text{ACTION} and \text{GOTO} tables
   (Coming up)

---

**Example: Building the collection**

**Initial State**

\[ I_0 \leftarrow \text{ closure}(\{g \rightarrow e.e.eof\}) \]

\[ = \{ g \rightarrow e.e.eof, e \rightarrow e \rightarrow t.e.eof, e \rightarrow t.e.eof, \}
\]

\[ t \rightarrow e.t.eof, t \rightarrow e.t.eof, t \rightarrow t.e.eof, \}

\[ t \rightarrow t.e.eof, t \rightarrow t.e.eof \}

**Iteration 1**

\[ I_1 \leftarrow \text{ goto}(I_0, e) = \{ g \rightarrow e.e.eof \} \]

\[ I_2 \leftarrow \text{ goto}(I_0, t) = \{ e \rightarrow t.e.eof, \}
\]

\[ e \rightarrow t.e.eof \}

**Iteration 2**

\[ I_1 \leftarrow \text{ goto}(I_1, e) \]

**Iteration 3**

\[ I_1 \leftarrow \text{ goto}(I_1, t) \]

**Example: Summary**

\[ I_0: \{ g \rightarrow e.e.eof, \}
\]

\[ e \rightarrow t.e.eof, \}

\[ e \rightarrow t.e.eof \}

\[ t \rightarrow e.t.eof, t \rightarrow e.t.eof, t \rightarrow t.e.eof \}

\[ t \rightarrow t.e.eof, t \rightarrow t.e.eof, t \rightarrow t.e.eof \}

\[ I_1: \{ g \rightarrow e.e.eof, \}
\]

\[ e \rightarrow t.e.eof \}

\[ t \rightarrow e.t.eof, t \rightarrow e.t.eof \}

\[ t \rightarrow t.e.eof, t \rightarrow t.e.eof \}

\[ I_2: \{ e \rightarrow t.e.eof, \}
\]

\[ t \rightarrow e.t.eof, t \rightarrow e.t.eof \}

\[ t \rightarrow t.e.eof, t \rightarrow t.e.eof \}

\[ t \rightarrow t.e.eof, t \rightarrow t.e.eof \}

\[ t \rightarrow t.e.eof, t \rightarrow t.e.eof \}

---

**LR(1) Table Construction**

To build the table, we simply interpret the sets

1. \( G' \) = Augment grammar \( G \) by adding production \( S' \rightarrow S \)
   (e.g., \( \text{goal} \rightarrow \text{expr} \) in our example)
2. Construct the canonical collection of sets of \( \text{LR}(1) \) items for \( G' \)
3. State \( j \) of the parser is constructed from set \( I_j \)
   (a) if \( [A \rightarrow \alpha \epsilon \beta, b] \in I_j \) and \( \text{goto}(j, a) = I_j \), then set \( \text{action}(j, a) \) to "shift \( F \) (\( b \) must be a terminal)
   (b) if \( [A \rightarrow \alpha, a] \in I_j \), then set \( \text{action}(j, a) \) to "reduce \( A \rightarrow \alpha \)"
   (c) if \( [S' \rightarrow S, \epsilon] \in I_j \), then set \( \text{action}(j, \epsilon) \) to "accept"
4. If \( \text{goto}(j, A) = I_j \), then set \( \text{goto}(j, A) \) to \( j \)
5. All other entries in \text{action} and \text{goto} are set to "error"
6. The initial state of the parser is the state constructed from the set containing the item \( [S' \rightarrow \epsilon, \epsilon] \)
Conflicts During LR(1) Construction

Rules 3a, 3b, & 3c can construct two different actions for an entry in ACTION. If this happens, the grammar is not LR(1). Usually indicates an ambiguous construct in the grammar.

Example: dangling-else, again:

\[ S' \rightarrow S \]
\[ S \rightarrow \text{if } E \text{ then } S \text{ else } S \]
\[ S \rightarrow \text{other} \]

Abbreviate as:

\[ S' \rightarrow S \]
\[ S \rightarrow \text{iS} \]
\[ S \rightarrow \text{a} \]

The conflict

State \( I_4 \) of LR(1) parser is:

\[ I_4 : [S \rightarrow \text{iS} \cdot eS, e], [S \rightarrow \text{iS} \cdot cS, e], [S \rightarrow \text{iS} \cdot eS, e] \]

Q. What action do we take on \( e \)?

Item \( [S \rightarrow \text{iS} \cdot eS, e] \) says:

Item \( [S \rightarrow \text{iS} \cdot cS, e] \) says:

Solution

???

Comparing LALR with LR

What new conflicts are possible?

- \( \text{goto}(I, X) \) depends only on core \( I \), not on \( X \), so just merge the \( \text{goto} \) functions for merged states.
- \( \text{shift} \) action also depends only on core (e.g., \( [A \rightarrow \alpha \cdot cS, e] \))
- \( \text{reduce} \) action depends on both (e.g., \( [A \rightarrow \alpha \cdot a] \))
- \( \Rightarrow \) merging states as above does not introduce shift-reduce conflicts unless there was one before
- \( \Rightarrow \) new reduce-reduce conflicts are possible

Example: Final Tables

Fill in rows \( S_3 \) and \( S_5 \):

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1d - * eot</td>
<td>expr</td>
</tr>
<tr>
<td>( S_0 )</td>
<td>s4</td>
</tr>
<tr>
<td>( S_1 )</td>
<td></td>
</tr>
<tr>
<td>( S_2 )</td>
<td></td>
</tr>
<tr>
<td>( S_3 )</td>
<td></td>
</tr>
<tr>
<td>( S_4 )</td>
<td>s4</td>
</tr>
<tr>
<td>( S_5 )</td>
<td></td>
</tr>
<tr>
<td>( S_6 )</td>
<td></td>
</tr>
<tr>
<td>( S_7 )</td>
<td></td>
</tr>
</tbody>
</table>

The Grammar

1. goal \( \rightarrow \) expr
2. expr \( \rightarrow \) term - expr
3. | term
4. term \( \rightarrow \) factor * term
5. | factor
6. factor \( \rightarrow \) id

LALR(1) parsing

Definition (Core):

The core of a set of LR(1) items is the set of LR(0) items derived by ignoring the lookahead symbols.

Example: the two sets

\[ \{[A \rightarrow \alpha \cdot \beta, a], [A \rightarrow \alpha \cdot \beta, b] \} \]

have the same core.

Key Idea of LALR:

If two states, \( I_i \) and \( I_j \), have the same core, we can merge those states in the action and goto tables.
**LALR(1) table construction**

The simple algorithm

To construct LALR(1) parsing tables, we insert one step into the LR(1) table construction algorithm.

(1.5) For each core present among the set of LR(1) items, find all sets having that core and replace these sets by their union.

Update the goto function to reflect the replacement sets.

The resulting algorithm has large space requirements.

A better algorithm

A more space efficient algorithm can be derived by observing that:

- we can represent $i$, by its kernel, those items that are either the initial item $[S \rightarrow \bullet S, \text{eof}]$ or do not have the $\bullet$ at the left end of the rhs.
- we can compute shift, reduce, and goto actions for the state derived from $i$, directly from kernel($I_i$).

This avoids building the complete canonical collection.

---

**LR(1) versus LL(1) grammars**

Finding reductions in LR($k$) and LL($k$)

- LR($k$) ⇒ Parser must select a reduction based on
  1. everything to the left of the reducible phrase
  2. everything derived from the reducible phrase itself
  3. the next $k$ terminal symbols

- LL($k$) ⇒ Parser must select the reduction based on
  1. everything to the left of the reducible phrase
  2. the first $k$ terminals derived from the reducible phrase

Thus, LR($k$) has more information to choose reductions ⇒ LR($k$) parsers can parse more grammars than LL($k$).

"...in practice, programming languages do not actually seem to fall in the gap between LL(1) languages and deterministic (aka LR) languages."


---

**The hierarchy of context-free grammars**

Inclusion hierarchy for context-free grammars:

- Context-free grammars
  - Floyd-Evans parsable
  - Unambiguous CFG's
  - Operator precedence
  - LR
    - LR(1)
    - LALR(1)
    - SLR(1)
    - LR(0)
    - LL(1)