

Bottom-Up Parsing

Goal: Trace a **rightmost derivation in reverse** by starting with the input string and working back towards the start symbol.

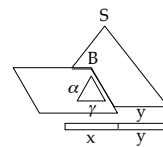
Observation: in each step of a rightmost derivation sequence, the string to the right of the handle must contain only terminals.

LR parsing: Reads input from **left to right** and constructs **rightmost derivation in reverse**

Overall approach

- Find the next right-hand side of a production (**handle**) such that its replacement by left-hand side non-terminal will yield previous right-sentential form
- As input is consumed, change state to encode possibilities (recognize **valid prefixes**); if handle is found, **REDUCE**, otherwise **SHIFT** (or **ERROR**)

$$S \Rightarrow_{rm}^* \alpha B \gamma \Rightarrow_{rm} \alpha \gamma \Rightarrow_{rm}^* xy$$



Example

Consider the grammar

- 1 **<goal>** ::= a **<A>** **** e
- 2 **<A>** ::= **<A>** b c
- 3 | b
- 4 **** ::= d

and the input string *abbcd*.

Why is (3,3) not a handle for **a<A>bcd**?

| Prod'n | Sentential Form | Handle* |
|--------|-----------------|---------|
| -- | abbcd | 3,2 |
| 3 | a<A>bcd | 2,4 |
| 2 | a<A>de | 4,3 |
| 4 | a<A>e | 1,4 |
| 1 | <goal> | |

- The trick appears to be scanning the input and finding valid right-sentential forms.

* (rule, position of right end of handle in input string).

Handles

We are trying to find a substring of the current right-sentential form where:

- α matches some production $A ::= \alpha$
- reducing to A is one step in the reverse of a rightmost derivation.

Such a string is called a **handle**.

Formally,

- a **handle** of a right-sentential form γ is a production $A ::= \beta$ and a position in γ where β may be found.

Convention: position specifies the right end of the handle.

- If $(A ::= \beta, k)$ is a handle, then replacing the β in γ at position k with A produces the previous right-sentential form in a rightmost derivation of γ .

Handles

Provable fact:

The substring to the right of a handle contains only terminal symbols.

Proof:

Follows from the fact that all γ_i are right-sentential forms.

Corollary

The right end of a handle is to the right of the previously reduced variable.

Shift-reduce parsing

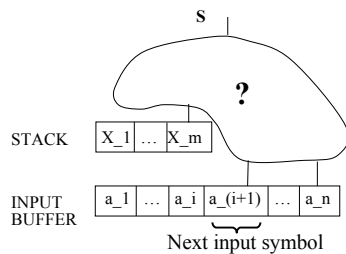
One scheme to implement a handle-pruning, bottom-up parser is called a **shift-reduce parser**.

Shift-reduce parsers use a **stack** and an **input buffer**

1. Initialize stack with $\$$
2. Repeat until the top of the stack is the goal symbol and the input token is **eof**
 - a) **find the handle**
if we don't have a handle on top of the stack, **shift** an input symbol onto the stack
 - b) **prune the handle**
if we have a handle ($A ::= \beta, k$) on top of the stack, **reduce**
 - i) pop $|\beta|$ symbols off the stack
 - ii) push A onto the stack

Shift-reduce parsing

Conceptual view of bottom-up parsing algorithms (assumes a restricted class of unambiguous grammars):



Example

Left-recursive expression grammar

– Example LL(1) grammar

(original form, before left factoring)

```
1 <goal> ::= <expr>
2 <expr> ::= <expr> + <term>
3         | <expr> - <term>
4         | <term>
5 <term> ::= <term> * <factor>
6         | <term> / <factor>
7         | <factor>
8 <factor> ::= num
9          | id
```

$x - 2 * y$

| Stack | Input | Handle | Action |
|-----------|-----------|--------|--------|
| | x - 2 * y | | shift |
| x | - 2 * y | | shift |
| x - | 2 * y | | shift |
| x - 2 | * y | | shift |
| x - 2 * | y | | shift |
| x - 2 * y | | | reduce |
| x - 2 | | | reduce |
| x - | | | reduce |
| x | | | reduce |
| | | | accept |

1. Shift until top of stack is the right end of a handle
2. Find the left end of the handle and reduce

5 shifts + 9 reduces + 1 accept

Viabale prefix

A **viabale prefix** is

1. a prefix of a right-sentential form that does not continue past the right end of the rightmost handle of that sentential form, or
 2. a prefix of a right-sentential form that can appear on the stack of a shift-reduce parser.
- It is always possible to add terminals onto the end of a viable prefix to obtain a right-sentential form.
 - As long as the prefix represented by the stack is viable, the parser has not seen a detectable error.

If the grammar is unambiguous, there is a unique rightmost handle.

LR(k) grammars are unambiguous.

Shift-reduce parsing

- Grammars that are often used to construct shift-reduce parsers:
 - operator grammars (will not discuss here--in book)
 - LR(1) grammars
 - canonical LR(1) grammars
 - simple LR(1) grammars (SLR(1))
 - lookahead LR(1) grammars (LALR(1))
- Grammars use different methods or levels of "context" information to detect handle.
- LR(1), SLR(1) and LALR(1) grammars use **finite automata** (NFAs or DFAs) to recognize viable prefixes and store "context" information.

LR(k) grammars

Informally, we say that a grammar G is $LR(k)$ if, given a rightmost derivation

$$S = \gamma_0 \Rightarrow_{rm} \gamma_1 \Rightarrow_{rm} \gamma_2 \Rightarrow_{rm} \dots \Rightarrow_{rm} \gamma_n = w$$

we can, for each right-sentential form in the derivation,

1. isolate the handle of each right-sentential form
2. determine the production by which to reduce by scanning γ_i from left to right, going at most k symbols beyond the right end of the handle of γ_i .

LR parsing

There are three commonly used algorithms to build tables for an "LR" parser:

1. SLR(1) = LR(0) + FOLLOW

- smallest class of grammars
- smallest tables (number of states)
- simple, fast construction

2. LR(1)

- full set of LR(1) grammars
- largest tables (number of states)
- slow, large construction

3. LALR(1)

- intermediate sized set of grammars
- same number of states as SLR(1)
- canonical construction is slow and large
- better construction techniques exist

An LR(1) parser for either ALGOL or PASCAL has several thousand states, while an SLR(1) or LALR(1) parser for the same language may have several hundred states

SLR(1) parsing

- Viable prefix of a right-sentential form:
 - contains both terminals and nonterminals
 - can be recognized with a DFA
- Building a SLR parser
 - construct DFA for recognizing viable prefixes
 - augment with FOLLOW to disambiguate actions

States in the NFA are LR(0) items

States in the DFA are sets of LR(0) items (subset construction)

Note: An "augmented grammar" is one where the start symbol appears only on the *lhs* of productions. For the rest of LR parsing, we will assume the grammar is augmented with a production $S' ::= S$

LR(0) items

An **LR(0) item** is a string $[\alpha]$, where α is a production from G with a \bullet at some position in the rhs

- The \bullet indicates how much of an item we have seen at a given state in the parsing process.
- $[A ::= \bullet XY \zeta]$ indicates that the parser is looking for a string that can be derived from $XY\zeta$
- $[A ::= XY \bullet \zeta]$ indicates that the parser has seen a string derived from XY and is looking for one derivable from ζ

LR(0) Items

(no lookahead)

$A ::= XY\zeta$ generates 4 LR(0) items.

1. $[A ::= \bullet XY \zeta]$
2. $[A ::= X \bullet Y \zeta]$
3. $[A ::= XY \bullet \zeta]$
4. $[A ::= XY \zeta \bullet]$

Canonical LR(0) items

- The SLR(1) table construction algorithm uses a specific set of sets of LR(0) items.
- These sets are called the **canonical collection of sets** of LR(0) items for a grammar G .
- The canonical collection corresponds to the set of states of the DFA that recognizes viable prefixes. Each state is the set of valid LR(0) items at a particular point in the parse.
- The LR(0) item $[A ::= \beta_1 \bullet \beta_2]$ is valid for a viable prefix $\alpha\beta_1$ if there is a derivation

$$S' \Rightarrow_m^* \alpha A w \Rightarrow_m \alpha \beta_1 \beta_2 w$$

In general, an item will be valid for many viable prefixes.

Canonical Collection of LR(0) items

To construct the canonical collection we need two functions:

– closure(I)

- if $[A ::= \alpha \bullet B\beta] \in I$, then, in state j , the parser might next see a string derivable from $B\beta$
- to form its closure, add all items of the form $[B ::= \bullet \gamma] \in G$

– GOTO(I, X)

- If I is the set of items that are valid for some viable prefix γ , then $\text{GOTO}(I, X)$ is the set of items that are valid for the viable prefix γX .

Closure(I)

- Given an item $[A ::= \alpha \bullet B\beta]$, its closure contains the item and any other items that can generate legal substrings to follow α
- Thus, if the parser has viable prefix α on its stack, the input should reduce to $B\beta$ (or γ for some other item $[B ::= \bullet \gamma]$ in the closure).

To compute closure(I):

```
function closure(I)
  repeat
    new_item ← false
    for each item  $[A ::= \alpha \bullet B\beta] \in I$ , each production  $B ::= \gamma \in G$ 
      if  $[B ::= \bullet \gamma] \notin I$  then
        add  $[B ::= \bullet \gamma]$  to  $I$ 
        new_item ← true
    endif
  until (new_item = false)
  return I
```

Goto(I, X)

- Let I be a set of LR(0) items and X be a grammar symbol.
- Then, $\text{GOTO}(I, X)$ is the closure of the set of all items $[A ::= \alpha X \bullet \beta]$ such that $[A ::= \alpha \bullet X\beta] \in I$
- If I is the set of valid items for some viable prefix γ , then $\text{goto}(I, X)$ is the set of valid items for the viable prefix γX .
- $\text{goto}(I, X)$ represents state after recognizing X in state I .

To compute $\text{goto}(I, X)$:

```
function goto(I, X)
  J ← set of items  $[A ::= \alpha X \bullet \beta]$  such that  $[A ::= \alpha \bullet X\beta] \in I$ 
  J' ← closure(J)
  return J'
```

Collection of sets of LR(0) items

We start the construction of the collection of sets of LR(0) items with the item $[S' ::= \bullet S]$, where

S' is the start symbol of the augmented grammar G'
 S is the start symbol of G

To compute the collection of sets of LR(0) items

```
procedure items(G')
  S0 ← closure([S' ::= • S])
  items ← {S0}
  ToDo ← {S0}
  while ToDo not empty do
    remove Si from ToDo
    for each grammar symbol X do
      Snew ← goto(Si, X)
      if Snew is a new state then
        items ← items U {Snew}
        ToDo ← ToDo U {Snew}
      endif
    endfor
  endwhile
  return items
```

LR(0) machines

LR(0) DFA

- states - canonical sets of LR(0) items
- edges - goto transitions
- recognizes all viable prefixes
- no lookahead

Reducing a handle (*rhs* of production) to a nonterminal can be viewed as:

1. returning to the state at beginning of the handle
2. making a transition on a nonterminal from this state

To return to the state at beginning of the handle, we must use the stack to store the state!

SLR(1) tables

SLR(1) parser

- augment LR(0) machine
- add FOLLOW information using one token of lookahead
- encoded as ACTION, GOTO tables

ACTION table

- for each [state, lookahead] pair
 - have we reached end of handle?
 - if not, shift
 - if at end of handle, reduce
 - may also accept or error
 - use lookahead to guide decision

GOTO table

- for each [state, nonterminal] pair
 - pick state to go to after reduction

The Algorithm

1. Construct the collection of sets of LR(0) items for G .
2. State i of the parser is constructed from I_i .
 - a) if $[A ::= \alpha \bullet a\beta] \in I_i$ and $\text{goto}(I_i, a) = I_j$, then set ACTION[i, a] to "shift j ". (a must be a terminal)
 - b) if $[A ::= \alpha \bullet] \in I_i$, then set ACTION[i, a] to "reduce $A ::= \alpha$ " for all a in FOLLOW(A).
 - c) if $[S' ::= S \bullet] \in I_i$, then set ACTION[i, eof] to "accept".
3. If $\text{goto}(I_i, A) = I_j$, then set GOTO[i, A] to j .
4. All other entries in ACTION and GOTO are set to "error".
5. The initial state of the parser is the state constructed from the set containing the item $[S' ::= \bullet S]$

SLR(1) parser example

| The Grammar | | | |
|-------------|-------|-------|--|
| 1 | E ::= | T + E | |
| 2 | | T | |
| 3 | T ::= | id | |

| The Augmented Grammar | | | |
|-----------------------|----|-----|-------|
| 0 | S' | ::= | E |
| 1 | E | ::= | T + E |
| 2 | | | T |
| 3 | T | ::= | id |

| Symbol | FIRST | FOLLOW |
|--------|--------|------------|
| S' | { id } | { eof } |
| E | { id } | { eof } |
| T | { id } | { +, eof } |

Example LR(0) states

S_0 : [$S ::= \bullet E$],
 [$E ::= \bullet T + E$],
 [$E ::= \bullet T$],
 [$T ::= \bullet id$]

S_1 : [$S ::= E \bullet$]

S_2 : [$E ::= T \bullet + E$],
 [$E ::= T \bullet$]

S_3 : [$T ::= id \bullet$]

S_4 : [$E ::= T + \bullet E$],
 [$E ::= \bullet T + E$],
 [$E ::= \bullet T$],
 [$T ::= \bullet id$]

S_5 : [$E ::= T + E \bullet$]

Example GOTO function

Start

$S_0 \leftarrow \text{closure}(\{[S ::= \bullet E]\})$

Iteration 1

$\text{goto}(S_0, E) = S_1$

$\text{goto}(S_0, T) = S_2$

$\text{goto}(S_0, id) = S_3$

Iteration 2

$\text{goto}(S_2, +) = S_4$

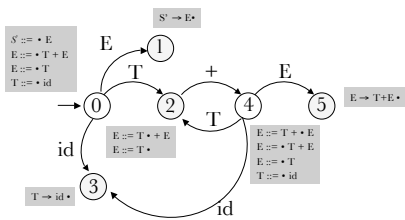
Iteration 3

$\text{goto}(S_4, id) = S_3$

$\text{goto}(S_4, E) = S_5$

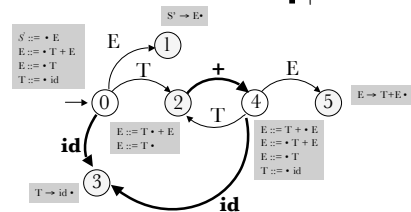
$\text{goto}(S_4, T) = S_2$

The DFA



Building the SLR(1) Table: Shift Entries

Enter a shift n (where n is the state to go to) for each transition on a terminal symbol



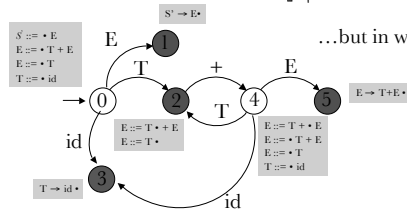
| | E | T | $+$ | id |
|-------|-----|-----|-----|------|
| S_0 | | | | |
| S_1 | | | | |
| S_2 | | | | |
| S_3 | | | | |
| S_4 | | | | |
| S_5 | | | | |

Building the SLR(1) Table: Reduce Entries

A reduce should occur in any state containing an item with a \bullet at the end of a production...

| State | id | + | E |
|-------|--------|---|--------|
| 0 | shift | | |
| 1 | | | reduce |
| 2 | shift | | |
| 3 | reduce | | |
| 4 | shift | | |
| 5 | | | reduce |

...but in which columns?



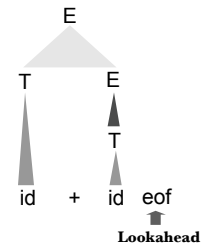
The SLR(1) Solution

Use **FOLLOW sets!**

If (for example) $T+E$ is on the stack, the next symbol in the input should be a terminal that can come after an E in a sentential form

$S' ::= E$
 $E ::= T + E$
 $E ::= T$
 $T ::= id$

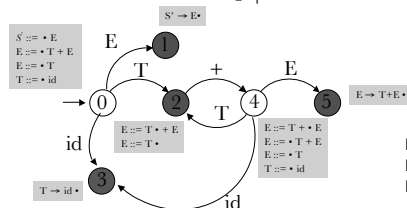
$FOLLOW(S') = \{eof\}$
 $FOLLOW(E) = \{eof\}$
 $FOLLOW(T) = \{+, eof\}$



Reduce Entries

A reduce is entered in the column for every terminal in $FOLLOW(X)$, where X is the non-terminal on the left side of the production

| State | id | + | E |
|-------|--------|---|--------|
| 0 | shift | | |
| 1 | | | reduce |
| 2 | shift | | |
| 3 | reduce | | |
| 4 | shift | | |
| 5 | | | reduce |



$FOLLOW(S') = \{eof\}$
 $FOLLOW(E) = \{eof\}$
 $FOLLOW(T) = \{+, eof\}$

GOTO

Last problem:

What state is the DFA in after the reduction?

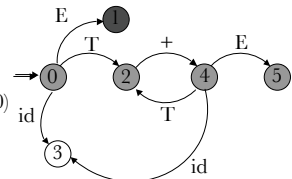
Solution:

The automaton "rewinds" as symbols are popped off the stack, and from there takes the transition for the pushed non-terminal (left hand side)

Example

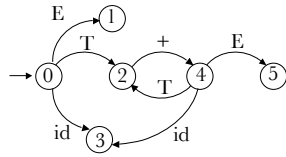
In state 5, reduce by $E ::= T + E$:

1. Pop $T + E$ (return to state 0)
2. Push E , go to state 1



GOTO Table

$\text{goto}(S_0, E) = S_1$
 $\text{goto}(S_0, T) = S_2$
 $\text{goto}(S_0, \text{id}) = S_3$
 $\text{goto}(S_2, +) = S_4$
 $\text{goto}(S_4, \text{id}) = S_3$
 $\text{goto}(S_4, E) = S_5$
 $\text{goto}(S_4, T) = S_2$



| | E | T | id | + | \$ |
|----|----|----|----|----|----|
| S0 | S1 | S2 | S3 | | |
| S2 | | S2 | S3 | S4 | |
| S4 | | S2 | S3 | S5 | |

Final Step

- Notice that to reduce by $S' ::= E$ amounts to finishing building the tree for the input string
- So, this entry is changed to “accept” in the table

| | E | T | id | + | \$ |
|----|--------|----|----|----|----|
| S0 | accept | S2 | S3 | | |
| S2 | | S2 | S3 | S4 | |
| S4 | | S2 | S3 | S5 | |

Final ACTION and GOTO tables

| | E | T | id | + | \$ |
|----|--------|----|----|----|----|
| S0 | accept | S2 | S3 | | |
| S2 | | S2 | S3 | S4 | |
| S4 | | S2 | S3 | S5 | |

| State | id | + | \$ |
|-------|-------|-------|-------|
| S0 | shift | shift | shift |
| S2 | shift | shift | shift |
| S4 | shift | shift | shift |