

Bottom-up parsers

Properties

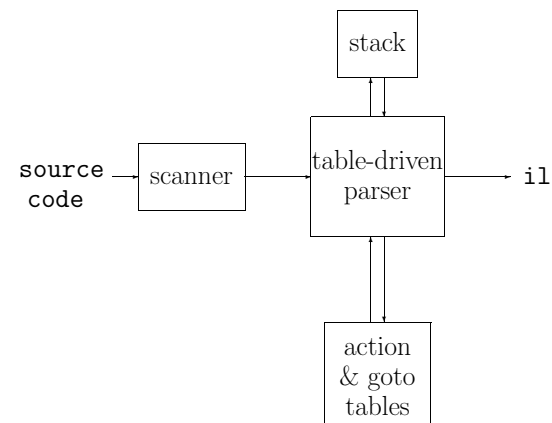
- start with input string, end with start symbol
- apply productions in reverse to input, replacing right-hand side (*rhs*) of production with *lhs* nonterminal
- final result is a rightmost derivation, in reverse.
- bottom-up parsers can use current stack and lookahead to choose production
- one type of bottom-up parsers called LR(k) are more powerful than LL(k) parsers because they can see the entire *rhs* before choosing a production

Definitions

- the *handle* is defined as the combination of
1) *rhs* to be replaced, and 2) its position
- replacement step is called a *reduction*

Bottom-up parsing using tables

A table-driven bottom-up parser looks like



Stack two items per state: *state* and *symbol*

Table building tools are readily available (**yacc**, **CUP**)

Bottom-up parsing example

Consider the grammar

```
1 | S ::= a A B e
2 | A ::= A b c
3 |   | b
4 | B ::= d
```

and the input string **abbcde**.

Production	Sentential Form	Handle
—	abbcde	3,2 ($A \leftarrow b$)
3	aAbcde	2,4 ($A \leftarrow A b c$)
2	aAde	4,3 ($B \leftarrow d$)
4	aABe	1,4 ($S \leftarrow a A B e$)
1	S	—

The problem is deciding when and which *rhs* to reduce.

Shift-reduce parsing

Shift-reduce parsers

- one approach to bottom-up parsing
- are simple to understand
- have a simple, table-driven, *shift-reduce* skeleton
- encode grammatical knowledge in tables

A shift-reduce parser has just four canonical actions:

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 *lhs*
3. *accept* — terminate parsing and signal success
4. *error* — call an error recovery routine

Skeleton parser

```

push  $s_0$ 
token = next_token()
repeat forever
  s = top of stack
  if action[s,token] = "shift  $s_i$ " then
    push token
    push  $s_i$ 
    token = next_token()
  else if action[s,token] =
    "reduce  $A ::= \beta$ " then
    pop  $2 * |\beta|$  symbols
    s = top of stack
    push  $A$ 
    push goto[s,A]
  else if action[s, token] = "accept" then
    return
  else error()

```

This takes k shifts, l reduces, and 1 accept, where k is the length of the input string and l is the length of the reverse rightmost derivation.

LR(1) grammars

Informally, we say that a grammar G is LR(1) if we can find the sequence of handles for a reverse rightmost derivation using at most 1 token of lookahead past the end of the handle.

Properties

- virtually all context-free programming language constructs can be expressed in an LR(1) form
- LR grammars are the most general grammars that can be parsed by a non-backtracking, shift-reduce parser
- efficient shift-reduce parsers can be implemented for LR(1) grammars
- LR parsers detect an error as soon as possible in a left-to-right scan of the input
- LR grammars describe a proper superset of the languages recognized by LL (predictive) parsers

Example parser

		ACTION			GOTO	
		id	+	\$	E	T
P0	S ::= E	S_0 shift 3	—	—	1	2
P1	E ::= T + E	S_1 —	—	accept	—	—
P2	T	S_2 —	shift 4	reduce P2	—	—
P3	T ::= id	S_3 —	reduce P3	reduce P3	—	—
		S_4 shift 3	—	—	5	2
		S_5 —	—	reduce P1	—	—

Stack	Input	Action
\$ 0	id + id \$	shift 3
\$ 0 id 3	+ id \$	reduce P3 (T ::= id)
\$ 0 T 2	+ id \$	shift 4
\$ 0 T 2 + 4	id \$	shift 3
\$ 0 T 2 + 4 id 3	\$	reduce P3 (T ::= id)
\$ 0 T 2 + 4 T 2	\$	reduce P2 (E ::= T)
\$ 0 T 2 + 4 E 5	\$	reduce P1 (E ::= T + E)
\$ 0 E 1	\$	accept

Shift-reduce parsing

Definitions

- a *right-sentential form* is any string that may occur in a legal rightmost derivation
- a *viable prefix* of a right-sentential form is any prefix that does not continue past the right end of its rightmost handle

Shift-reduce parsers

- operator precedence
define precedence between operands to guide reductions
- SLR(1) = LR(0) + FOLLOW
construct DFA for recognizing viable prefix, use FOLLOW to guide reductions
- LR(1) — construct DFA for recognizing viable prefix, storing lookahead information in DFA
- LALR(1) — construct DFA for recognizing viable prefix, propagating lookahead information in DFA

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 parse.

$[A ::= \bullet XYZ]$ indicates that the parser is looking for a string that can be derived from XYZ

$[A ::= XY \bullet Z]$ indicates that the parser has seen a string derived from XY and is looking for one derivable from Z

$LR(0)$ Items (no lookahead)

$A ::= XYZ$ generates 4 $LR(0)$ items.

1. $[A ::= \bullet XYZ]$
2. $[A ::= X \bullet YZ]$
3. $[A ::= XY \bullet Z]$
4. $[A ::= XYZ \bullet]$

LR(0) items

The Grammar

P1	E	::=	T + E
P2			T
P3	T	::=	id

The Augmented Grammar

P0	S'	::=	E
P1	E	::=	T + E
P2			T
P3	T	::=	id

LR(0) machine

Definitions

- closure of $[A ::= \alpha \bullet B\beta]$ contains itself and any items of form $[B ::= \bullet \gamma]$, repeat for new items.
- $goto(X)$ of $[A ::= \alpha \bullet X\beta]$ contains the closure of $[A ::= \alpha X \bullet \beta]$.

LR(0) DFA construction

1. begin with closure of start symbol $[S ::= \bullet \alpha]$
2. for each state, calculate $goto(X)$ for all grammar symbols X , generating states
3. repeat step 2 for all newly generated states

Properties

- states in the DFA are sets of $LR(0)$ items
- states represent viable prefixes of productions
- to recognize viable prefixes of language, save state of current production on stack when reducing new nonterminal

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_3 : $[T ::= id \bullet]$

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

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

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

LR(1) items

We can build SLR parsers using LR(0) items and FOLLOW information.

But, we can get more powerful parsers by keeping track of lookahead information in the states of the LR parser.

An $LR(k)$ item is a pair $[\alpha, \beta]$, where

α is a production from G with a \bullet at some position in the *rhs*

β is a lookahead string containing k symbols (terminals or **eof**)

LR(1) items

- example: $[A ::= X \bullet YZ, a]$
- several LR(1) items may have the same *core*

$[A ::= X \bullet YZ, a]$

$[A ::= X \bullet YZ, b]$

we represent this as

$[A ::= X \bullet YZ, \{a, b\}]$

LR(1) machine

Definitions

- *closure* of $[A ::= \alpha \bullet B\beta, a]$ contains itself and any items of form $[B ::= \bullet\gamma, \text{FIRST}(\beta a)]$, repeat for new items.
- *goto*(X) of $[A ::= \alpha \bullet X\beta, a]$ contains the closure of $[A ::= \alpha X \bullet \beta, a]$.

LR(1) DFA construction

1. begin w/ closure of start symbol $[S ::= \bullet\alpha, \text{eof}]$
2. for each state, calculate *goto*(X) for all grammar symbols X , generating states
3. repeat step 2 for all newly generated states

Properties

- $[A \rightarrow X \bullet YZ, \alpha] \Rightarrow$ have recognized X & YZ would be valid
- $[A \rightarrow X \bullet YZ, \alpha] \Rightarrow [Y \rightarrow \bullet\beta, \gamma] \& [Y \rightarrow \bullet\delta, \eta]$ are also valid, where $\gamma, \eta \in \text{FIRST}(Z\alpha)$
- recognizing Y takes parser to $[A \rightarrow XY \bullet Z, \alpha]$

LR(1) lookahead

What's the point of all these lookahead symbols?

- carry them along to allow us to choose correct reduction when there is any choice
- lookaheads are bookkeeping, unless item has \bullet at right end.
 - in $[A ::= X \bullet YZ, a]$, a has no direct use
 - in $[A ::= XYZ\bullet, a]$, a is useful
- allows use of (non-invertible) grammars where productions have the same *rhs*

The point

For $[A ::= \alpha\bullet, a]$ and $[B ::= \alpha\bullet, b]$, we can decide between reducing to A and to B by looking at limited right context!

Example LR(1) states

S_0 : $[S' ::= \bullet E, \$]$,
 $[E ::= \bullet T + E, \$]$, $\text{FIRST}(\epsilon \$) = \$$
 $[E ::= \bullet T, \$]$, $\text{FIRST}(\epsilon \$) = \$$
 $[T ::= \bullet \text{id}, +]$, $\text{FIRST}(+ E \$) = +$
 $[T ::= \bullet \text{id}, \$]$, $\text{FIRST}(\epsilon \$) = \$$

S_1 : $[S' ::= E \bullet, \$]$ S_3 : $[T ::= \text{id} \bullet, +]$
 $[T ::= \text{id} \bullet, \$]$

S_2 : $[E ::= T \bullet + E, \$]$,
 $[E ::= T \bullet, \$]$ S_5 : $[E ::= T + E \bullet, \$]$

S_4 : $[E ::= T + \bullet E, \$]$,
 $[E ::= \bullet T + E, \$]$, $\text{FIRST}(\epsilon \$) = \$$
 $[E ::= \bullet T, \$]$, $\text{FIRST}(\epsilon \$) = \$$
 $[T ::= \bullet \text{id}, +]$, $\text{FIRST}(+ E \$) = +$
 $[T ::= \bullet \text{id}, \$]$, $\text{FIRST}(\epsilon \$) = \$$

LR(1) table construction

The Algorithm

1. if S appears on *rhs* of production, create augmented grammar G' by adding $S' ::= S$
2. construct the collection of sets of $LR(1)$ items for G' .
3. State i of the parser is constructed from I_i .
 - (a) if $[A ::= \alpha \bullet a\beta, b] \in I_i$ and $\text{goto}(I_i, a) = I_j$, then set $\text{action}[i, a]$ to "shift j ". (a must be a terminal)
 - (b) if $[A ::= \alpha \bullet, a] \in I_i$, then set $\text{action}[i, a]$ to "reduce $A ::= \alpha$ ".
 - (c) if $[S' ::= S \bullet, \text{eof}] \in I_i$, then set $\text{action}[i, \text{eof}]$ to "accept".
4. If $\text{goto}(I_i, A) = I_j$, then set $\text{goto}[i, A]$ to j .
5. All other entries in action and goto are set to "error"
6. The initial state of the parser is the state constructed from the set containing the item $[S' ::= \bullet S, \text{eof}]$.

What can go wrong?

Multiply actions may exist in the ACTION table.

Two cases arise

shift/reduce

This is called a *shift/reduce* conflict. In general, it indicates an ambiguous construct in the grammar.

- can modify the grammar to eliminate it
- can resolve in favor of shifting

classic example: dangling else

reduce/reduce

This is called a *reduce/reduce* conflict. Again, it indicates an ambiguous construct in the grammar.

- often, no simple resolution
- parse a nearby language

classic example: PL/I call and subscript

Example ACTION and GOTO tables

The Augmented Grammar

P0		S'	::=	E
P1		E	::=	T + E
P2				T
P3		T	::=	id

	ACTION			GOTO	
	id	+	\$	E	T
S_0	shift 3	—	—	1	2
S_1	—	—	accept	—	—
S_2	—	shift 4	reduce P2	—	—
S_3	—	reduce P3	reduce P3	—	—
S_4	shift 3	—	—	5	2
S_5	—	—	reduce P1	—	—

The "reduce" actions are determined by the lookahead entries in the LR(1) items

Resolving conflicts

Precedence and *associativity* can be used to resolve shift/reduce conflicts in ambiguous grammars.

- same precedence & right associative, or higher precedence \Rightarrow *shift*
- same precedence & left associative, or lower precedence \Rightarrow *reduce*

Advantages:

- more concise, albeit ambiguous, grammars
- shallower parse trees \Rightarrow fewer reductions

\Rightarrow a simpler expression grammar

```

<expr> ::= <expr> * <expr>
        | <expr> / <expr>
        | <expr> + <expr>
        | <expr> - <expr>
        | ( <expr> )
        | -<expr>
        | id
        | num
  
```

Operator precedence parsers

Another approach to shift-reduce parsing is to use *operator precedence*.

Given $S \Rightarrow^* \alpha S_1 S_2 \beta$, there are three possible *precedence relations* between S_1 and S_2 .

1. S_1 in handle, S_2 not
(S_1 reduced before S_2) $S_1 > S_2$
2. both in handle
(reduced at same time) $S_1 = S_2$
3. S_2 in handle, S_1 not
(S_2 reduced before S_1) $S_1 < S_2$

A handle is thus composed of:

$\langle \rangle, \langle = \rangle, \langle = = \rangle, \dots$

To decide whether to shift or reduce, compare top of stack with lookahead (ignoring nonterminals):

- Shift if \langle or $=$
- Reduce if \rangle

Left end of handle is marked by first \langle found

Error recovery in shift-reduce parsers

The problem

- encounter an invalid token
- bad pieces of tree on stack

We want to *parse* the rest of the file

Restarting the parser

- find a restartable state on the stack
- move to a consistent place in the input
- print an informative message (*line number*)

Yacc's error mechanism

- designated token **error**
- valid in any production
- when an error is discovered, pops the stack until **error** is legal

Parsing example

The Grammar
 $E ::= E + E \mid E * E \mid id$

	+	*	id	\$
+	>	<	<	>
*	>	>	<	>
id	>	>	>	>
\$	<	<	<	>

Stack	Input	Precedence
\$	id + id * id \$	\$ < id
\$ < id	+ id * id \$	id > +
\$ < E	+ id * id \$	\$ < +
\$ < E +	id * id \$	+ < id
\$ < E + < id	* id \$	id > *
\$ < E + < E	* id \$	+ < *
\$ < E + < E *	id \$	* < id
\$ < E + < E * < id	\$	id > \$
\$ < E + < E * E	\$	* > \$
\$ < E + E	\$	+ > \$
\$ < E	\$	\$ > \$

Error recovery example

```
stmt_list : stmt
          | stmt_list ; stmt
```

can be augmented with **error**

```
stmt_list : stmt
          | error
          | stmt_list ; stmt
```

this should

- throw out the erroneous statement
- synchronize at ";" or "end"
- invoke `yyerror("syntax error")`

Other "natural" places for errors

- all the "lists"
- missing parentheses or brackets
- extra operator or missing operator

LR(1) grammars

Informally, we say that a grammar G is LR(1) if we can find the sequence of handles for a reverse rightmost derivation using at most 1 token of lookahead past the end of the handle.

Properties

- virtually all context-free programming language constructs can be expressed in an LR(1) form
- LR grammars are the most general grammars that can be parsed by a non-backtracking, shift-reduce parser
- efficient shift-reduce parsers can be implemented for LR(1) grammars
- LR parsers detect an error as soon as possible in a left-to-right scan of the input
- LR grammars describe a proper superset of the languages recognized by LL (predictive) parsers

LALR(1) parsers

There are two approaches to constructing LALR(1) parsing tables

1. build LR(1) sets of items, then merge states with same core
2. build LR(0) sets of items, then propagate lookahead information.

LALR(1) properties

- LALR(1) parsers have same number of states as LR(0) parsers (core LR(0) items are the same)
- may perform *reduce* rather than *error*, but will catch error before more input is processed
- LALR derived from LR with no shift-reduce conflict will also have no shift-reduce conflict
- LALR may create reduce-reduce conflict not in LR from which LALR is derived
- used by utilities such as **yacc**, **bison**, **cup**

LALR(1) parsers

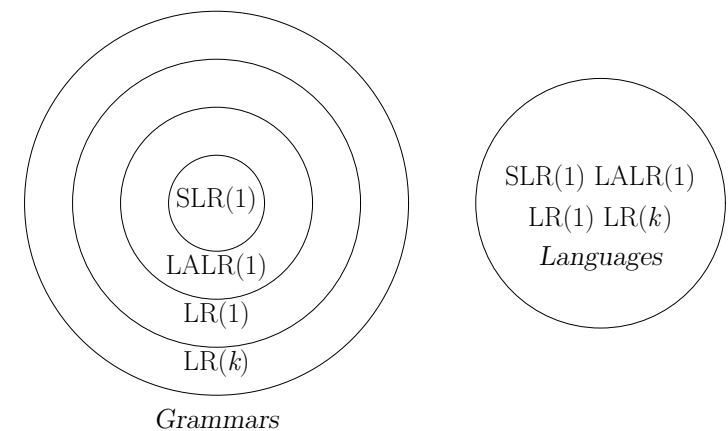
Problem

- LR(1) parsers are powerful, but have many more states than LR(0) (approximately $\times 10$ for Pascal)
- larger state tables longer to construct, run

LALR(1) parsers

- define the *core* of a set of LR(1) items to be the set of LR(0) items derived by ignoring the lookahead symbols.
- example of two sets of LR(1) items with same core:
 - $\{[A \Rightarrow \alpha \bullet \beta, \mathbf{a}], [A \Rightarrow \alpha \bullet \beta, \mathbf{b}]\}$, and
 - $\{[A \Rightarrow \alpha \bullet \beta, \mathbf{c}], [A \Rightarrow \alpha \bullet \beta, \mathbf{d}]\}$
- if two sets of LR(1) items, I_i and I_j , have the same core, we can merge the states that represent them in the **ACTION** and **GOTO** tables
- almost as powerful as LR(1), same size as LR(0)

LR(k) languages



Parsing review

Recursive Descent A hand coded recursive descent parser directly encodes a grammar (typically an LL(1) grammar) into a series of mutually recursive procedures. It has most of the linguistic limitations of LL(1).

LL(k) An LL(k) parser must be able to recognize the use of a production after seeing only the first k symbols of its right hand side.

Operator Precedence An ad hoc shift-reduce parser suitable for small expression grammars.

LR(k) An LR(k) parser must be able to recognize the occurrence of the right hand side of a production after having seen all that is derived from that right hand side with k symbols of lookahead.

Parsing review

	<i>Advantages</i>	<i>Disadvantages</i>
recursive descent LL(1)	fast simple automatable good error recovery	hand-coded maintenance hard no left recursion $LL(1) \subset LR(1)$
operator precedence	fast simple small table	poor error detection $L(G) \neq L(\text{parser})$ only small languages
LR(1)	fast early error detection automatable	larger table size poor error recovery