

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 *reduction* or *handle-pruning*

Bottom-up parsing example

Consider the grammar

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

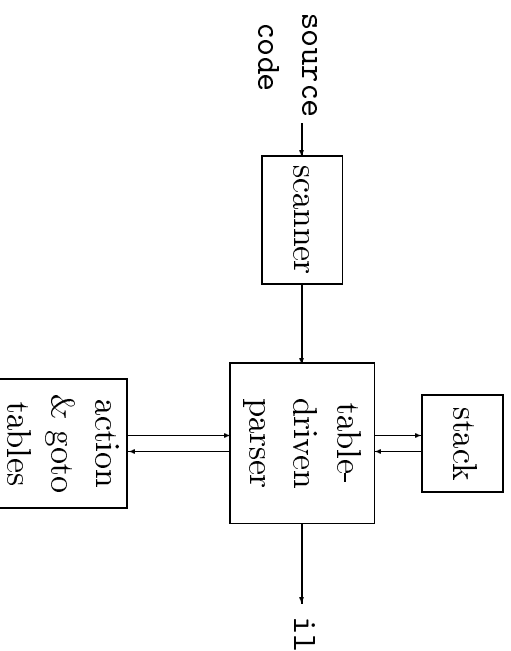
and the input string *abbcd*.

Production	Sentential Form	Handle
—	abbcd	3,2 (A \leftarrow b)
3	aAbcd	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.

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)

We'll learn how to build these tables later

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

Shift-reduce parsing

The 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.

Note: Equivalent to Figure 4.30, Aho, Sethi, and Ullman

Example parser

	Stack	Input	Action	GOTO
P0	S ::= E	id + id \$	shift 3	E T
P1	E ::= T + E	+ id \$	reduce P3 (T ::= id)	
P2	T	+ id \$	shift 4	
P3	T ::= id	id \$	shift 3	

	Stack	Input	Action	GOTO
S ₀	shift 3	id + id \$	shift 3	1 2
S ₁	—	+ id \$	reduce P3	—
S ₂	—	+ id \$	shift 4	—
S ₃	—	id \$	reduce P3	—
S ₄	shift 3	id \$	—	5 2
S ₅	—	id + id \$	—	—

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

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

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

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

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

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