Shift-Reduce Parsing

CS 331 Programming Languages
Lecture Slides
Monday, February 16, 2026

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

© 2017–2026 Glenn G. Chappell

Unit Overview
Lexing & Parsing

Topics

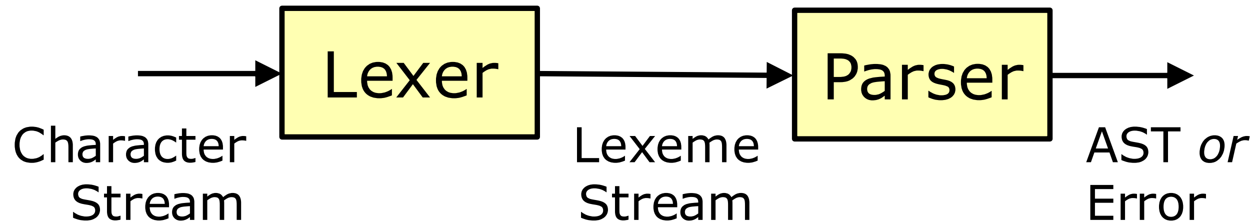
- ✓ Introduction to lexing & parsing
- ✓ The basics of lexical analysis
- ✓ State-machine lexing I
- ✓ State-machine lexing II
- ✓ State-machine lexing III
- ✓ The basics of syntax analysis
- ✓ Recursive-descent parsing I
- ✓ Recursive-descent parsing II
- ✓ Recursive-descent parsing III
- Shift-reduce parsing
- Parsing wrap-up

Lexical Analysis (Lexing): The basics of lexical analysis; State-machine lexing I–III

Syntax Analysis (Parsing): The basics of syntax analysis through Parsing wrap-up

Review

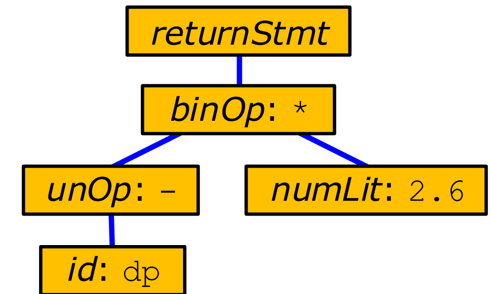
Review
Introduction to Lexing & Parsing



`return (-dp * 2.6) // x`

`return (-dp * 2.6) // x`

key | punct | op | id | op | num lit | punct



Two steps:

- **Lexical analysis** (**lexing**)
- **Syntax analysis** (**parsing**)

The output of a parser is typically an *abstract syntax tree* (*AST*). Specifications of these vary.

Review
The Basics of Syntax Analysis

Parsing methods can be divided into two broad categories: *top-down* and *bottom-up*.

Top-Down Parsers

- Go through derivation top to bottom, **expanding** nonterminals.
- Sometimes hand-coded and sometimes automatically generated.
- Method we look at: **Predictive Recursive Descent**.

Bottom-Up Parsers

- Go through the derivation bottom to top, **reducing** substrings to nonterminals.
- Code is almost always automatically generated.
- Method we look at: **Shift-Reduce**.

Review
Recursive-Descent Parsing I/II/III

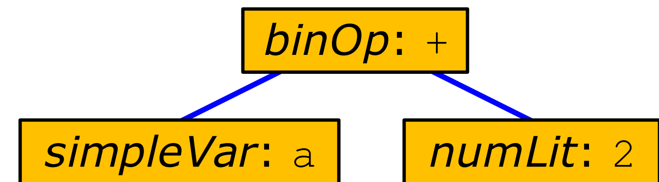
Recursive Descent is a top-down parsing method. **Predictive** = no backtracking. Predictive Recursive-Descent parsers that base decisions on *k* lexemes can use LL(*k*) grammars.

On success, a parser typically returns an **abstract syntax tree** (**AST**). Each internal node represents an operation. Its subtrees are the ASTs for the entities the operation is applied to.

Expression

```
a + 2
```

AST (diagram)



AST (Lua Representation)

```
{{ BIN_OP, "+" },
 { SIMPLE_VAR, "a" },
 { NUMLIT_VAL, "2" }}
```

See `rdparser3.lua`.

Shift-Reduce Parsing

Shift-Reduce Parsing
Introduction [1/2]

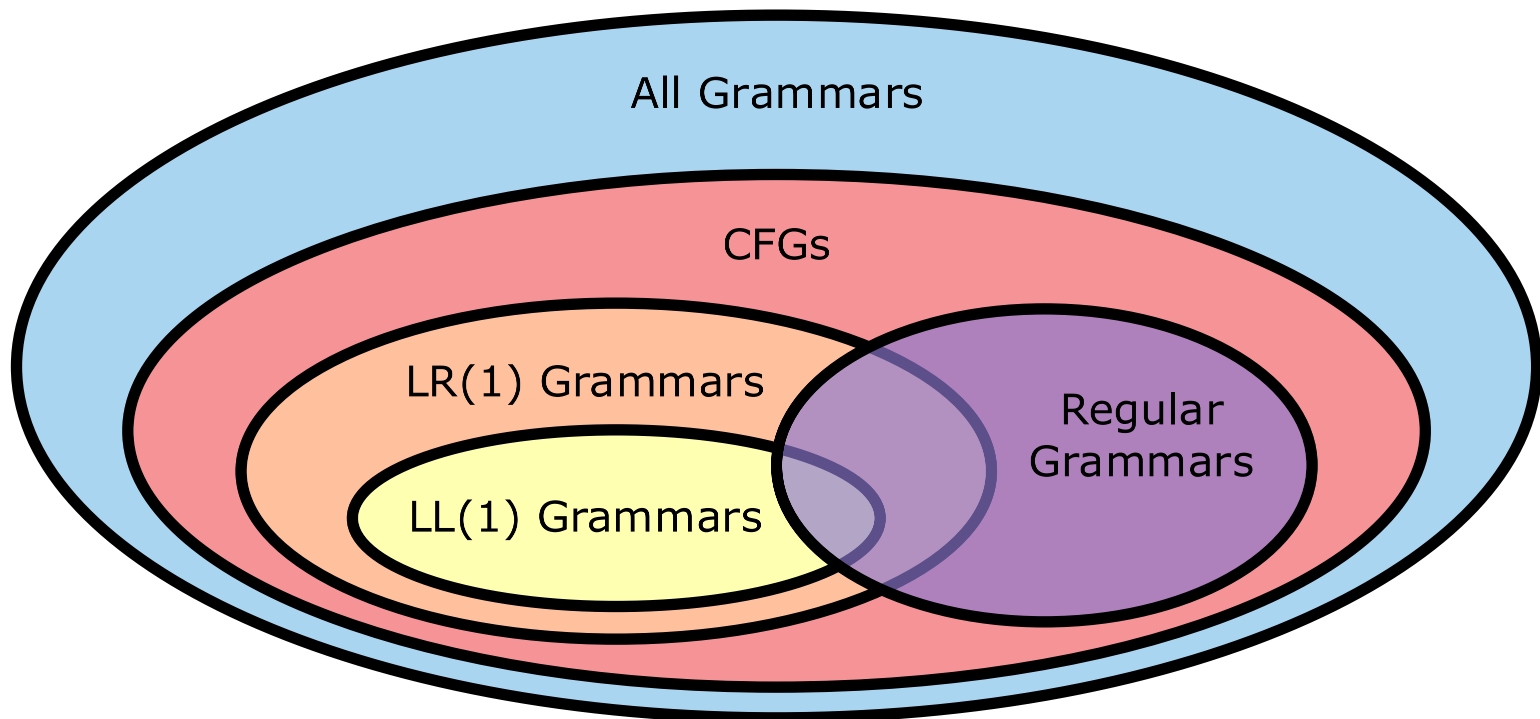
Now we look at a parsing method called **Shift-Reduce**.

- A bottom-up parsing method.
- Code is almost always automatically generated.
- Not practical when first introduced [D. Knuth 1965], but after additional work, became heavily used. Still used today.

The CFGs that a Shift-Reduce parser can use if it makes certain decisions based on *k* upcoming lexemes are called **LR(*k*) grammars**. Unlike top-down parsers, multi-symbol look-ahead is uncommon; *k* is rarely more than 1. So: **LR(1) grammars**.

Shift-Reduce Parsing
Introduction [2/2]

The LR(1) grammars form a strictly larger category than the LL(1) grammars. So Shift-Reduce parsing is a more general technique than Recursive-Descent parsing.

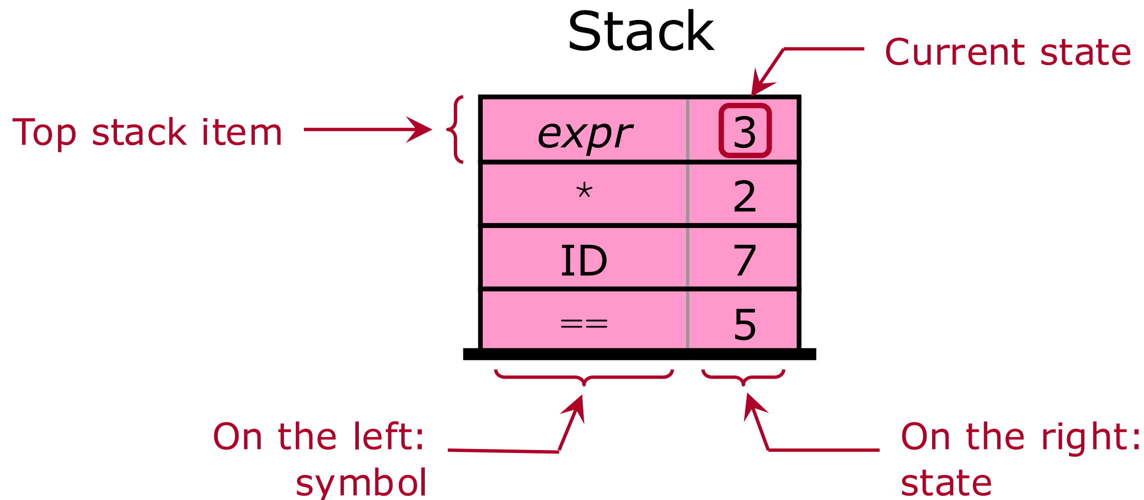


A Shift-Reduce parser does its work by running an automaton called a *Shift-Reduce automaton*. We look at these next.

Shift-Reduce Automata — Overview [1/3]

A **Shift-Reduce automaton** is a state machine with an associated stack. It is always based on a grammar. We will number both productions in the grammar and states (1, 2, 3, …).

Each stack item holds both a *symbol* from the grammar—either terminal or nonterminal—and a *state* (number). The *current state* is the state in the top stack item.



Shift-Reduce Parsing
Shift-Reduce Automata — Overview [2/3]

A Shift-Reduce automaton runs in a series of steps. At each step, one of four actions is performed.

- **SHIFT.** Shift the next input symbol onto the stack (push).
- **REDUCE.** Apply a production in reverse, reducing symbols on the stack to a single nonterminal, which replaces them on the stack.
- **ACCEPT.** Done, successful (input is syntactically correct).
- **ERROR.** Done, unsuccessful (input contains a syntax error).

Operation of the automaton uses a **parsing table** in two parts.

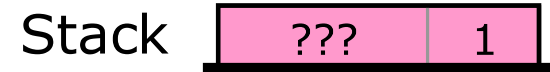
- The **action table** has a row for each state and a column for each terminal. Each entry specifies one of the above four actions.
- The **goto table** has a row for each state and a column for each nonterminal. An entry in this table may specify a state.

Shift-Reduce Automata — Overview [3/3]

Start the automaton by pushing the start state on the stack, along with any symbol we want; this symbol is ignored.

Symbol in this item will be ignored.

Stack



Run the automaton as a series of steps, each doing a lookup in the action table. Look up using the *current state* (from the top stack item) and the *current symbol* (from the input). Perform the action specified in the action table; if necessary, refer to the goto table.

Stop the automaton when the action table indicates either **ACCEPT** (successful) or **ERROR** (unsuccessful).

Shift-Reduce Parsing
Shift-Reduce Automata — Action Table [1/2]

At each step, we do a lookup in the **action table** using the *current state* (state in the top-of-stack item) and the *current symbol* in the input. We perform the action that the table entry indicates.

An action table entry can be S#, R#, ACCEPT, or ERROR (a blank cell in my tables), where "#" represents a number.

Here is the action indicated by each kind of entry.

S# (*# is the number of a state*)

SHIFT to the given state number. Push an item holding the the current input symbol and the given state number on the stack. Advance to the following input symbol.

Continued on next slide …

Shift-Reduce Automata — Action Table [2/2]

Continuing possible action-table entries:

R# (*# is the number of a production*)

REDUCE by the grammar production with the given number. Pop items forming the right-hand side of the production, then push an item holding the nonterminal on the left-hand side and a state determined using the **goto table**—discussed shortly. Note that the number of items to pop depends on the length of the right-hand side of the production.

ACCEPT

Stop. Indicate success; that is, input is syntactically correct.

ERROR—I represent this by a blank table cell.

Stop. Indicate failure; that is, input contains a syntax error.

Shift-Reduce Automata — Goto Table

The **goto table** is used only at the end of a **REDUCE** operation, when determining the state to push. A goto table entry can be either G# or blank. The G# entries are those that are used. Here "#" is the number of a state.

We look up in the goto table using the state *before the push* and the symbol we have decided to push (the left-hand side of the production we just reduced by). The number after the "G" is the state that goes in the new top-of-stack item; it becomes the new current state.

Shift-Reduce Parsing
Shift-Reduce Automata — Example [1/14]

The *Shift-Reduce Parsing Table* handout, linked on the class webpage, gives the information necessary to run a Shift-Reduce automaton for the given grammar.

In these slides, I describe the process of parsing "`x + 9`".

Our automaton requires an end-of-input lexeme, represented by $. Also, from the point of view of the parser, "`x`" is a lexeme in the ID category, and "`9`" is a lexeme in the NUMLIT category.

Input: `x + 9` $

I will mark the current position in the input with an arrow (↑).

Shift-Reduce Parsing
Shift-Reduce Automata — Example [2/14]

Start the automaton at the beginning of the input. Push an item containing any symbol (I leave the symbol portion blank, since we do not care what this symbol is) and the start state: 1.

Input: `x + 9 $`
↑

Stack

Shift-Reduce Automata — Example [3/14]

Input: `x + 9` $
↑

Stack

Do a lookup in the action table, using the current state (in the top-of-stack item) and the current symbol in the input.

State: 1. Symbol: ID (`x`).

Action table says S2: **SHIFT** to state 2.

Push an item containing ID (`x`) / 2, and advance the input.

Shift-Reduce Automata — Example [4/14]

Input: `x + 9 $`
↑

Stack

ID (`x`)	2
	1

State: 2. Symbol: `+`.

Action table says R6: **REDUCE** by production 6.

Pop the right-hand side of the production (ID, 1 item).

Push an item containing the left-hand side (*factor*) and the state from a goto table lookup.

State from stack *before push*: 1. Nonterminal to push: *factor*.

The goto table says G9.

So push an item containing *factor* / 9.

The input will not be advanced.

Shift-Reduce Automata — Example [5/14]

Input: `x + 9` $
↑

Stack

factor	9
	1

State: 9. Symbol: `+`.

Action table says R4: **REDUCE** by production 4.

Pop the right-hand side of the production (*factor*, 1 item).

Push an item containing the left-hand side (*term*) and the state from a goto table lookup.

State from stack *before push*: 1. Nonterminal to push: *term*.

The goto table says G10.

So push an item containing *term* / 10.

The input will not be advanced.

Shift-Reduce Parsing
Shift-Reduce Automata — Example [6/14]

Input: `x + 9 $`
↑ (under +)

Stack

term	10
	1

State: 10. Symbol: `+`.

Action table says R2: **REDUCE** by production 2.

Pop the right-hand side of the production (*term*, 1 item).

Push an item containing the left-hand side (*expr*) and the state from a goto table lookup.

State from stack *before push*: 1. Nonterminal to push: *expr*.

The goto table says G13.

So push an item containing *expr* / 13.

The input will not be advanced.

Shift-Reduce Automata — Example [7/14]

Input: `x + 9` $
↑

Stack

expr	13
	1

State: 13. Symbol: `+`.

Action table says S14: **SHIFT** to state 14.

Push an item containing `+` / 14, and advance the input.

Shift-Reduce Automata — Example [8/14]

Input: `x + 9 $`
↑ (pointing between `+` and `9`)

Stack

+	14
expr	13
	1

State: 14. Symbol: NUMLIT (`9`).

Action table says S3: **SHIFT** to state 3.

Push an item containing NUMLIT (`9`) / 3, and advance the input.

Input: `x + 9 $`
↑ (under `$`)

Stack

NUMLIT (`9`)	3
+	14
expr	13
	1

State: 3. Symbol: $.

Action table says R7: **REDUCE** by production 7.

Pop the right-hand side of the production (NUMLIT, 1 item).

Push an item containing the left-hand side (*factor*) and the state from a goto table lookup.

State from stack *before push*: 14. Nonterminal to push: *factor*.

The goto table says G15.

So push an item containing *factor* / 15.

The input will not be advanced.

Shift-Reduce Automata — Example [10/14]

Input: `x + 9 $`
↑ (pointing at `$`)

Stack

factor	15
+	14
expr	13
	1

State: 15. Symbol: $.

Action table says R4: **REDUCE** by production 4.

Pop the right-hand side of the production (*factor*, 1 item).

Push an item containing the left-hand side (*term*) and the state from a goto table lookup.

State from stack *before push*: 14. Nonterminal to push: *term*.

The goto table says G16.

So push an item containing *term* / 16.

The input will not be advanced.

Shift-Reduce Parsing
Shift-Reduce Automata — Example [11/14]

Input: `x + 9` $
↑ (pointing at $)

Stack

term	16
+	14
expr	13
	1

State: 16. Symbol: $.

Action table says R3: **REDUCE** by production 3.

Pop the right-hand side of the production (*expr* + *term*, 3 items).

Push an item containing the left-hand side (*expr*) and the state from a goto table lookup.

State from stack *before push*: 1. Nonterminal to push: *expr*.

The goto table says G13.

So push an item containing *expr* / 13.

The input will not be advanced.

Shift-Reduce Automata — Example [12/14]

Input: `x + 9` $
↑

Stack

expr	13
	1

State: 13. Symbol: $.

Action table says S17: **SHIFT** to state 17.

Push an item containing $ / 17, and advance the input.

Shift-Reduce Parsing
Shift-Reduce Automata — Example [13/14]

Input: `x + 9 $`
↑

We had better not do any more **SHIFT** operations!

Stack

$	17
expr	13
	1

State: 17. Symbol: *must not matter*.

Action table says R1: **REDUCE** by production 1.

Pop the right-hand side of the production (*expr* $, 2 items).

Push an item containing the left-hand side (*all*) and the state from a goto table lookup.

State from stack *before push*: 1. Nonterminal to push: *all*.

The goto table says G18.

So push an item containing *all* / 18.

The input will not be advanced.

Shift-Reduce Automata — Example [14/14]

Input: `x + 9 $`
↑

Stack

all	18
	1

State: 18. Symbol: *must not matter*.

Action table says ACCEPT: **ACCEPT** = terminate, successful.

Stop. Indicate success.

Look at what is on the stack:
the start symbol. A bottom-up parser must end a successful run in something like this manner (right?).

If we wish to make a parser using a Shift-Reduce automaton, then we still have two questions to answer:

(1) How do we construct an AST?

(2) How do we generate the parsing table (action + goto)?

Shift-Reduce Parsing
Constructing an AST

Q. How can a Shift-Reduce parser construct an AST?

A. The Shift-Reduce automaton can hold *three* things in each stack item: symbol, state, AST.

Process

- When doing **SHIFT**, push the AST for the lexeme being shifted.
- When doing **REDUCE**, construct and push the AST for the new nonterminal, based on the ASTs in the popped stack items.
- When doing **ACCEPT**, the top of the stack holds the start symbol and the associated AST. Return this AST to the caller.

If an AST is stored as a pointer-based tree, and a stack item holds a pointer to the root node of the AST, then this process can be very efficient. In particular AST nodes never need to be copied.

Shift-Reduce Parsing
Parsing Table Generation [1/4]

Q. How is the parsing table (action + goto) generated?

A. We will not cover details. We look at an overview of the issue.

First, note that generation of the parsing table is not part of the running of the parser. It happens when the parser is written.

Given a CFG, there is a relatively straightforward algorithm to generate a Shift-Reduce parsing table.

- We consider a state to correspond to a description of a position the parsing process: which productions we are currently handling, where we are in each, and what we expect in upcoming input.
- We repeatedly consider what happens when we move forward in the input: whether this gives us a new state and a transition to it.
- When we stop getting new states/transitions, we are done.

A CFG is an LR(1) grammar if this algorithm never specifies more than one action for a state + input combination.

Shift-Reduce Parsing
Parsing Table Generation [2/4]

Unfortunately, when given a typical CFG specifying the syntax of a programming language, this algorithm tends to generate very large parsing tables. Thus, when Shift-Reduce parsing was first introduced [D. Knuth 1965], it was considered to be impractical.

As often happens, this kicked off investigations into other algorithms for generating parsing tables.

One effort in this direction was that of researcher Frank DeRemer, who, in 1969, described methods for generating smaller parsing tables for restricted classes of LR(1) grammars.

Shift-Reduce Parsing
Parsing Table Generation [3/4]

DeRemer’s most successful idea was to merge two states into one if analysis shows they involve the same position in the same production. This can result in a much smaller parsing table.

But this merging might not work. It can result in states that specify multiple actions for the same input. The LR(1) grammars for which it works are called **Look-Ahead LR(1) grammars**, or **LALR(1) grammars**. Many PL syntax grammars are LALR(1).

Parsers produced in this way are often called **LALR parsers**. Note, however, that the parser itself is identical to what we described earlier: a shift-reduce automaton that produces an AST. The only difference is the way the parsing table is generated.

LALR parsers are common. Source code for them is produced by parser generator **Yacc** and its descendants, such as GNU **Bison**.

Shift-Reduce Parsing
Parsing Table Generation [4/4]

Sometimes we want to use a grammar that is not LALR(1).

If we do DeRemer's merging with such a grammar, then we get a parsing table that specifies more than one action for a single input. Such a table is said to have a **conflict** (a **reduce-reduce conflict** or a **shift-reduce conflict**).

An idea that can be helpful in these cases is to run DeRemer's merging anyway. Then we manually edit the parser source code that the automatic parser generator outputs, writing our own code to resolve conflicts.