Recursive-Descent Parsing continued
Shift-Reduce Parsing

CS F331 Programming Languages

CSCE A331 Programming Language Concepts

Lecture Slides

Monday, February 18, 2019

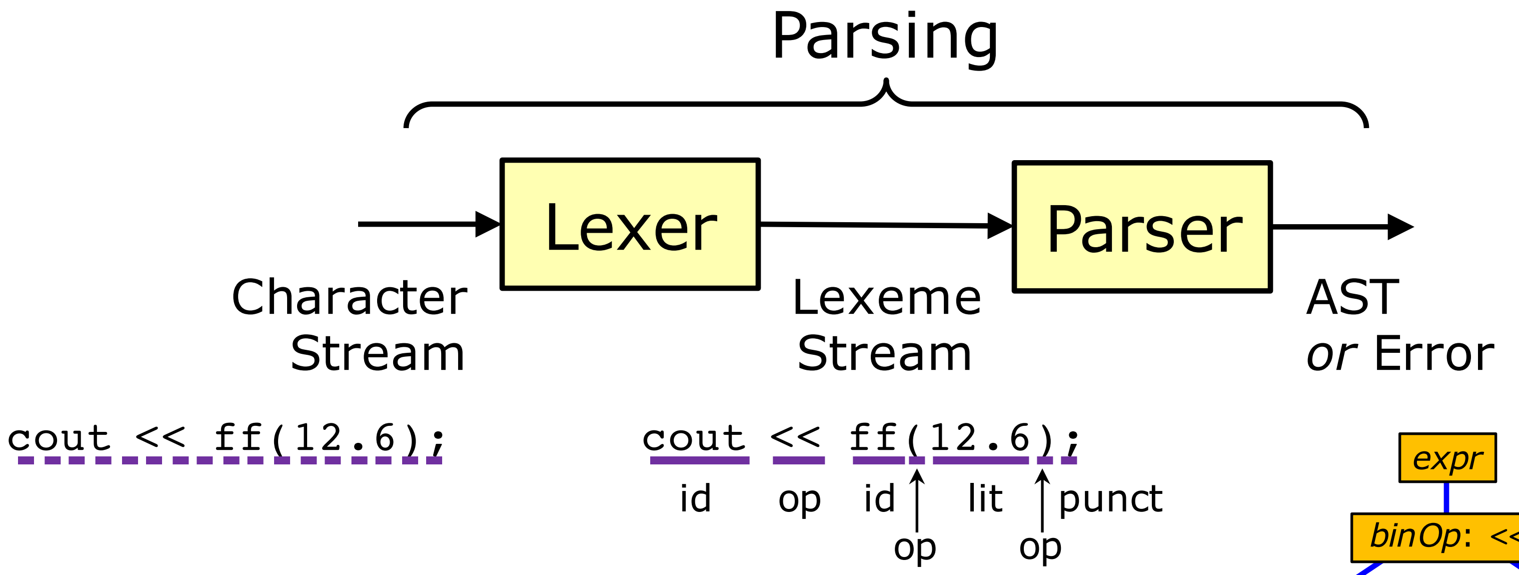
Glenn G. Chappell

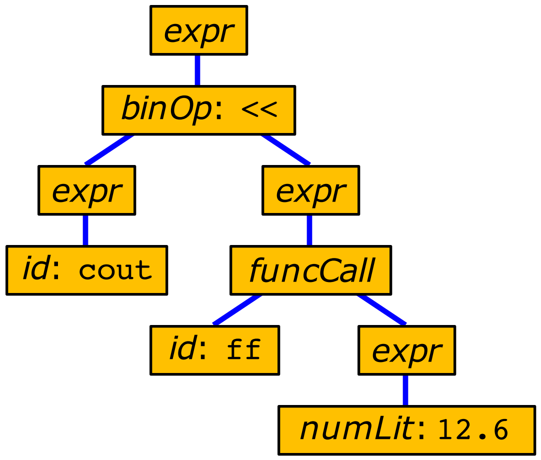
Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

Review
Overview of Lexing & Parsing





Two phases:

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

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

Review
The Basics of Syntax Analysis — Categories of Parsers

Parsing methods can be divided into two broad categories.

Top-Down

- Go through derivation from top to bottom, expanding nonterminals.
- Important subcategory: **LL parsers** (read input **L**eft-to-right, produce **L**eftmost derivation).
- Often hand-coded—but not always.
- Method we look at: **Predictive Recursive Descent**.

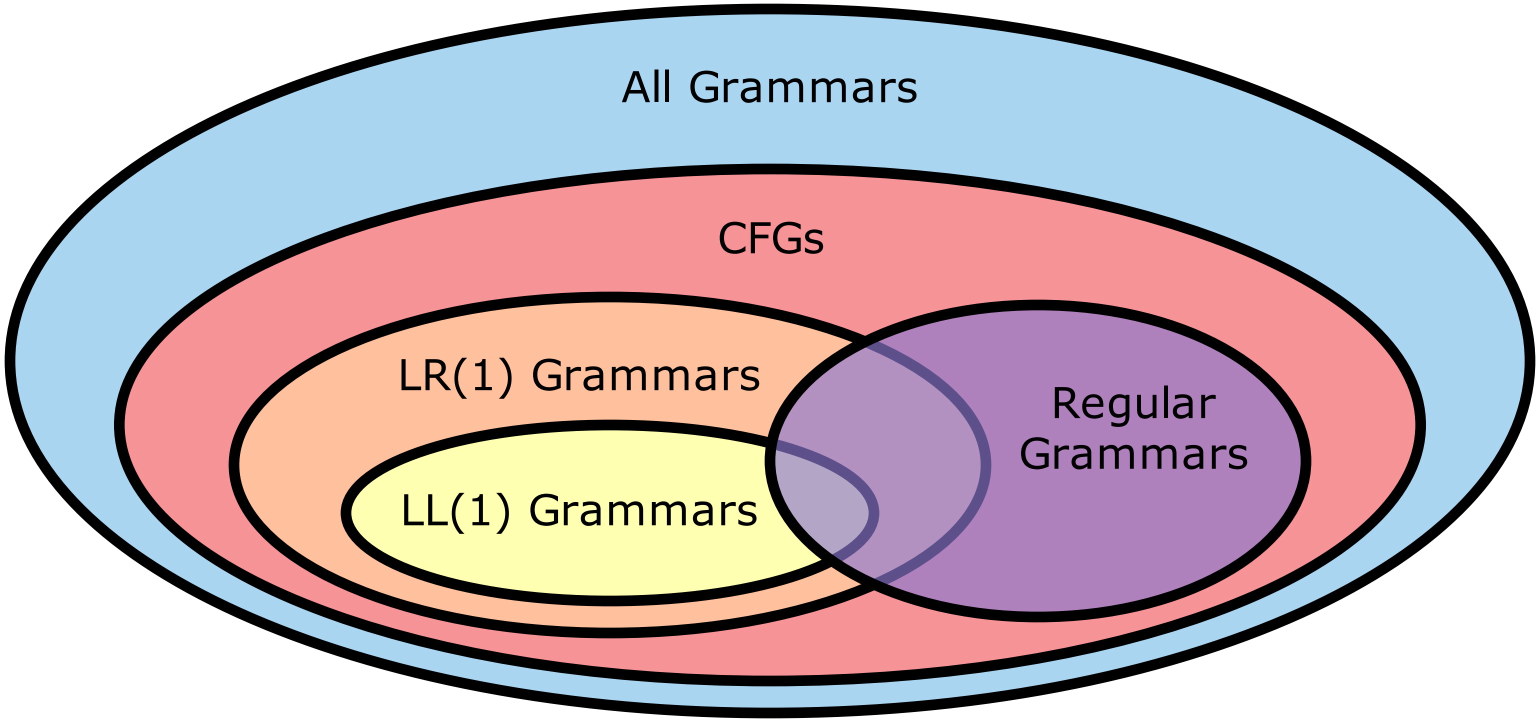
Bottom-Up

- Go through the derivation from bottom to top, reducing substrings to nonterminals.
- Important subcategory: **LR parsers** (read input **L**eft-to-right, produce **R**ightmost derivation).
- Almost always automatically generated.
- Method we look at: **Shift-Reduce**.

Review
The Basics of Syntax Analysis — Categories of Grammars

LL(1) grammars: those usable by LL parsers without lookahead.

LR(1) grammars: those usable by LR parsers without lookahead.



Review
Recursive-Descent Parsing [1/9]

Recursive Descent is a top-down parsing method.

- When we avoid backtracking: **predictive**. Predictive Recursive-Descent is an LL parsing method.
- There is one parsing function for each nonterminal. This parses all strings that the nonterminal can be expanded into.

A naively written parser may call some inputs *correct* when it cannot parse the entire input.

Example: `((x)))`

Two Solutions

- Introduce a new **end of input** lexeme (standard symbol: $). Revise the grammar to include it.
- After parsing, check to see the end of the input was reached.

Our parsers use the second solution.

Review
Recursive-Descent Parsing [2/9]

Parsing-function code is a translation of the right-hand side of the production for the nonterminal.

- A terminal in the right-hand side becomes a check that the input string contains the proper lexeme.
- A nonterminal in the right-hand side becomes a call to its parsing function.
- Brackets […] in the right-hand side become a conditional.
- Braces { … } in the right-hand side become a loop.

Review
Recursive-Descent Parsing [3/9]

We wish to parse arithmetic expressions in their usual form, with variables, numeric literals, binary `+`, `–`, `*`, and `/` operators, and parentheses. When given a syntactically correct expression, our parser should return an **abstract syntax tree** (**AST**).

All operators will be binary and left-associative: "`a + b + c`" means "`(a + b) + c`".

Precedence will be as usual: "`a + b * c`" means "`a + (b * c)`".

These may be overridden using parentheses: "`(a + b) * c`".

Due to the limitations of our in-class lexer, the expression "`k-4`" will need to be written as "`k - 4`".

Review
Recursive-Descent Parsing [4/9]

Grammar 3

expr → *term*
| *expr* ("+" | "–") *term*
term → *factor*
| *term* ("*" | "/") *factor*
factor → ID
| NUMLIT
| "(" *expr* ")"

Unfortunately, the natural grammar for expressions involving left-associative binary operators is not LL(1); it is, in fact, not LL(*k*) for any *k*.

When there is a choice to be made, an LL parser without lookahead must be able to make that choice based on the current lexeme. If this cannot be done, then the grammar is not LL(1).

Review
Recursive-Descent Parsing [5/9]

Sometimes, we can transform a non-LL(1) grammar into an LL(1) grammar that generates the same language.

Below is a revised form of Grammar 3.

Grammar 3b

expr → *term* { ("+" | "–") *term* }

term → *factor* { ("*" | "/") *factor* }

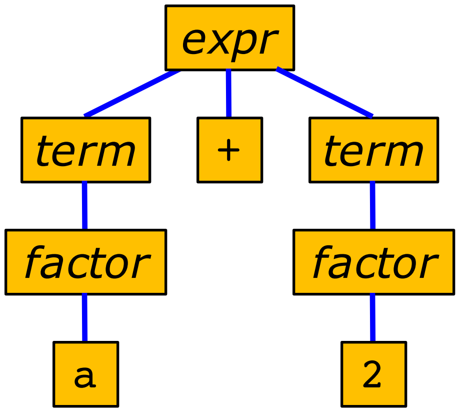
factor → ID
| NUMLIT
| "(" *expr* ")"

Review
Recursive-Descent Parsing [6/9]

We want write a parser that returns an **abstract syntax tree** (**AST**). First, we need to specify the format of an AST.

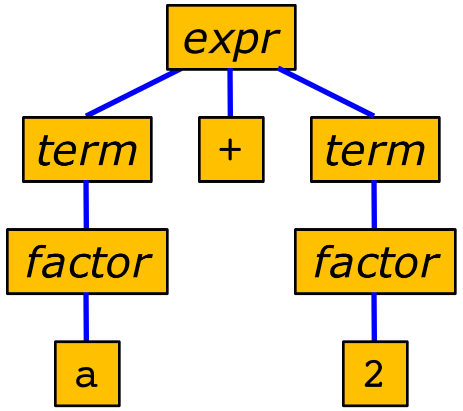
Recall that a **parse tree**, or **concrete syntax tree**, includes one leaf node for each lexeme in the input, and one other node for each nonterminal in the derivation.

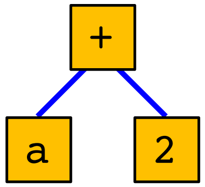
Below is the parse tree for `a + 2`, based on Grammar 3b.



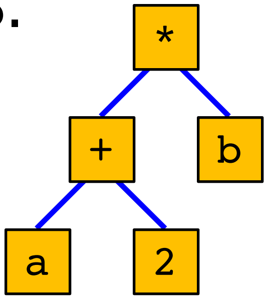
Review
Recursive-Descent Parsing [7/9]

An AST is similar, but it omits things that only serve to guide the parser: semicolons, commas, parentheses, extra nonterminals, etc. To the left is our parse tree. To the right is a reasonable AST for `a + 2`.

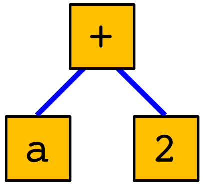


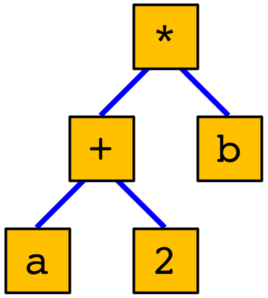


And here is the same style of AST for `(a + 2) * b`.



Review
Recursive-Descent Parsing [8/9]





We need to represent ASTs like these in Lua.

- Represent a single lexeme by its string form.
- If there is more than one node in an AST, then represent it as an array whose first item represents the root and whose remaining items each represent one of the subtrees rooted at the root's child nodes, in order.

The first AST, in Lua: `{ "+", "a", "2" }`

The second AST, in Lua: `{ "*", { "+", "a", "2" }, "b" }`

Review
Recursive-Descent Parsing [9/9]

It is better to describe our ASTs in a way that does not require tree drawings. So we specify the format of an AST for each line in our grammar. (Lines are numbered so we can refer to them.)

Grammar 3b

(1) *expr* $\rightarrow$ *term* { ("+" | "–") *term* }

(2) *term* $\rightarrow$ *factor* { ("*" | "/") *factor* }

(3) *factor* $\rightarrow$ ID

(4) | NUMLIT

(5) | "(" *expr* ")"

Recursive-Descent Parsing continued
Back to Example #3 — ASTs [8/8]

Grammar 3b

(1) *expr* → *term* { ("+" | "–") *term* }
(2) *term* → *factor* { ("*" | "/") *factor* }
(3) *factor* → ID
(4) | NUMLIT
(5) | "(" *expr* ")"

TO DO

- Based on Grammar 3b, write a Predictive Recursive-Descent parser that produces an AST, as described.

See `rdparser3.lua`, `userdparser3.lua`.

(1) If there is only a *term*, then the AST for the *expr* is the AST for the *term*. Otherwise, the AST is `{` *OO*`,` *AAA*`,` *BBB* `}`, where *OO* is the string form of the **last** operator, *AAA* is the AST for everything before it, and *BBB* is the AST for the last *term*.

(2) Similar to (1).

(3) AST for the *factor*: string form of the ID.

(4) AST for the *factor*: string form of the NUMLIT.

(5) AST for the *factor*: AST for the *expr*.

Recursive-Descent Parsing
Example #4: Better ASTs [1/4]

The ASTs we have specified are not quite what we want.

We need to know whether each node represents an operator, a variable identifier, etc. The lexer already figured this out, and it told the parser, but this information is not in the AST.

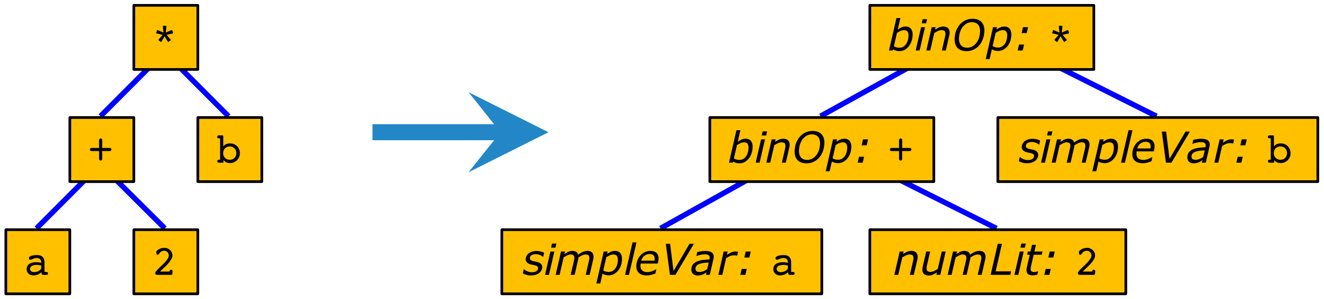
And there is other information we could store. For example, in many PLs*, "–" can be either a binary operator (`a - b`) or a unary operator (`-x`). The lexer does not know which it is. But the parser knows, and it can include this information in the AST.

*Including the one you will be writing a parser for.

Recursive-Descent Parsing
Example #4: Better ASTs [2/4]

Let's mark each node in the AST, indicating what kind of entity it represents.

We have three kinds of nodes: binary operators, simple variables, and numeric literals.



Recursive-Descent Parsing
Example #4: Better ASTs [3/4]

In the Lua form of our AST, we can replace each string with a two-item array. The first item in the array will be one of three constants: `BIN_OP`, `SIMPLE_VAR`, or `NUMLIT_VAL`. The second item will be the string form of the lexeme.

```
"/"                          { BIN_OP, "/" }
"abc"          →             { SIMPLE_VAR, "abc" }
"123"                        { NUMLIT_VAL, "123" }
```

So the AST for `a + 2` changes as shown below.

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

Recursive-Descent Parsing Example #4: Better ASTs [4/4]

```
"/"                      { BIN_OP, "/" }
"abc"        →           { SIMPLE_VAR, "abc" }
"123"                    { NUMLIT_VAL, "123" }
```

TO DO

- Rewrite the parser based on Grammar 3b so that it constructs and returns these improved ASTs.

See `rdparser4.lua`, `userdparser4.lua`.

Shift-Reduce Parsing
Introduction

Now we look at a class of bottom-up parsers that are table-based.

Producing the table happens before execution. This is a nontrivial process, which we briefly discuss later. For now:

- Parsing tables are almost always automatically generated.
- The idea first appeared in a 1965 paper of D. Knuth. A number of variations were published later.

Parser execution uses a state machine with a stack, called a **Shift-Reduce parser**. The name comes from two operations:

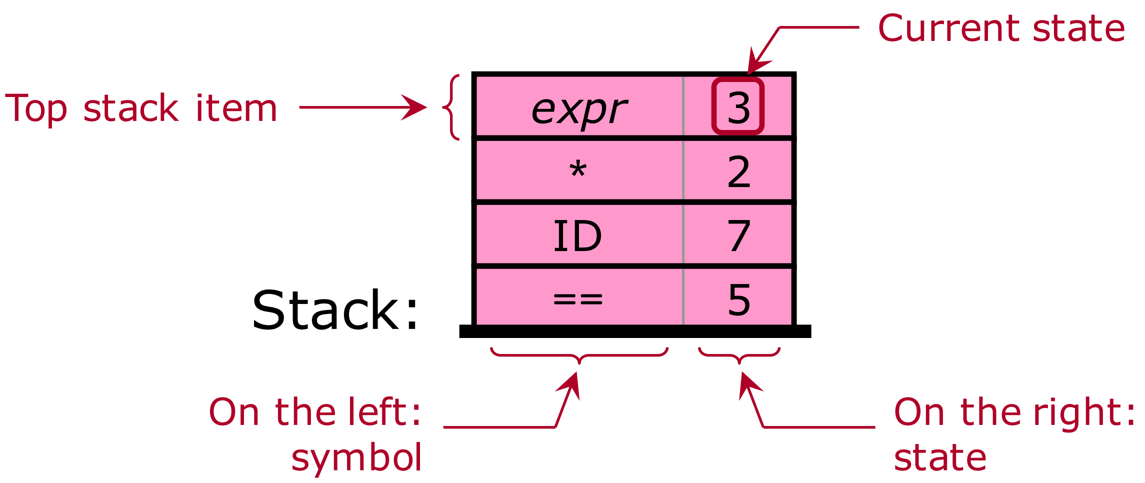
- **Shift.** Advance to the next input symbol.
- **Reduce.** Apply a production: replace substring with nonterminal.

In the form we present it, Shift-Reduce parsing is an LR parsing method that can handle all LR(1) grammars. As we will see, in practice, the class of grammars is usually further restricted.

Shift-Reduce Parsing Operation — Stack

Both productions in the grammar and states are numbered (1, 2, 3, …), for easy reference.

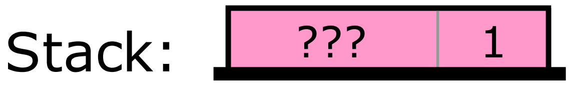
A Shift-Reduce parser runs as a state machine with an associated stack. Each stack item holds both a *symbol* from the grammar—either terminal or nonterminal—and a *state* (number). The **current state** is the state in the top stack item.



Shift-Reduce Parsing Operation — Table

Parser operation is guided by a table in two parts: **action table** and **goto table**. Each has one row for each state; we can place the two side by side. The action table has a column for each terminal; the goto table has a column for each nonterminal.

Start the state machine by pushing the start state (typically 1) on the stack, along with any symbol we want; this symbol is ignored.



Run the state machine as a series of lookups in the action table. Look up using the *current state* and the *current symbol* in the input. The action-table entry may require using the goto table.

Stop the state machine when the action table indicates either ACCEPT (successful parse) or ERROR (syntax error).

Shift-Reduce Parsing
Operation — 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 Parsing
Operation — 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

Input is syntactically correct. Stop.

ERROR (*I represent this by a blank table cell*)

A syntax error has been found. Stop.

Shift-Reduce Parsing Operation — 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
Running a Parser [1/10]

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

In these slides, I describe the process of parsing a very simple input: "`a = b`".

This parser requires an end-of-input lexeme, represented by $. Also, from the point of view of the parser, "`a`" and "`b`" are lexemes in the ID category.

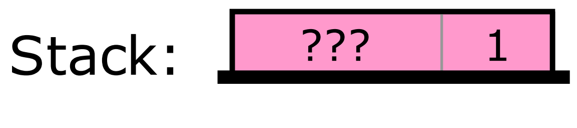
Input: `a = b $`

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

Shift-Reduce Parsing
Running a Parser [2/10]

Start the parser at the beginning of the input. Push an item containing any symbol (I will write “???”, since we do not care what this symbol is) and the start state: 1.

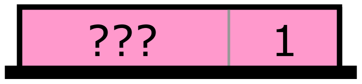
Input: `a = b $`



Shift-Reduce Parsing
Running a Parser [3/10]

Input: `a = b $`
↑

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: `a` (ID).

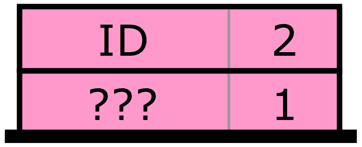
The action table says S2: shift to state 2.

Push an item containing ID / 2, and advance the input.

Shift-Reduce Parsing
Running a Parser [4/10]

Input: `a = b $`
↑

Stack



State: 2. Symbol: =.

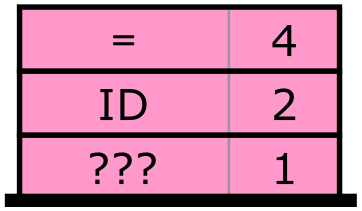
The action table says S4: shift to state 4.

Push an item containing = / 4, and advance the input.

Shift-Reduce Parsing
Running a Parser [5/10]

Input: `a = b $`
↑

Stack



=	4
ID	2
???	1

State: 4. Symbol: `b` (ID).

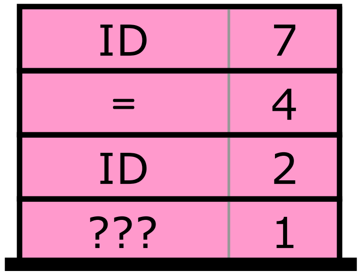
The action table says S7: shift to state 7.

Push an item containing ID / 7, and advance the input.

Shift-Reduce Parsing
Running a Parser [6/10]

Input: `a = b $`
↑

Stack



ID	7
=	4
ID	2
???	1

State: 7. Symbol: `$`.

The action table says R5: reduce by production 5.

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

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

State from stack *before push*: 4. Nonterminal to push: *expr*.

The goto table says G8.

So push an item containing *expr* / 8.

Note that the input will not be advanced.

Shift-Reduce Parsing
Running a Parser [7/10]

Input: `a = b $`
↑

State: 8. Symbol: `$`.

Stack

expr	8
=	4
ID	2
???	1

The action table says R2: reduce by production 2.

Pop the right-hand side of the production (ID = *expr*, three items).

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

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

The goto table says G3.

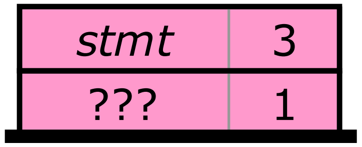
So push an item containing *stmt* / 3.

The input will not be advanced.

Shift-Reduce Parsing
Running a Parser [8/10]

Input: `a = b $`
↑

Stack



State: 3. Symbol: $.

The action table says S5: shift to state 5.

Push an item containing $ / 5, and advance the input.

Shift-Reduce Parsing
Running a Parser [9/10]

Input: `a = b $`
↑

We had better not do any more shifting!

Stack

$	5
stmt	3
???	1

State: 5. Symbol: *does not matter*.

The action table says R1: reduce by production 1.

Pop the right-hand side of the production (*stmt* $, two items).

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

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

The goto table says G6.

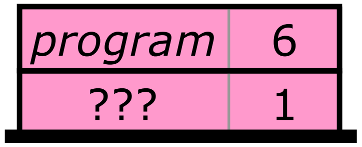
So push an item containing *program* / 6.

The input will not be advanced.

Shift-Reduce Parsing
Running a Parser [10/10]

Input: `a = b $`
```
              ↑
```

Stack



State: 6. Symbol: *does not matter*.

The action table says ACCEPT: terminate, successful parse.

Stop. Indicate success.

Note what is on the stack: the start symbol. A bottom-up parser will end a successful parse this way.

Shift-Reduce Parsing
TO BE CONTINUED …

Shift-Reduce Parsing will be continued next time.