# Lecture 7: Bottom-up parsing

David Hovemeyer

September 22, 2020

601.428/628 Compilers and Interpreters

# How do yacc and bison work?

- yacc and bison work by magic
- Any questions?

# Seriously, though

- yacc and bison generate *shift/reduce*, *bottom-up* parsers
  - using the LALR(1) algorithm
- Today we'll investigate the basic principles
  - using a simpler algorithm called SLR

# Bottom-up parsing

# Bottom-up parsing

An approach to parsing that is more powerful (handles a larger set of possible grammars) than predictive (top-down) parsing.

The basic idea is to start with the string of terminal symbols, and use grammar productions to reduce the string to the grammar's start symbol. In other words, we're constructing a derivation in reverse.

# A very simple expression grammar

Just $+$ and * operators, a and b as primary expressions

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow a$
$F \rightarrow b$

# Leftmost vs. rightmost derivations

In a leftmost derivation, each step replaces the leftmost nonterminal

In a rightmost derivation, each step replaces the rightmost nonterminal

# Example bottom-up parse

A "backwards" derivation (each production "generates" the left-hand nonterminal by replacing the right-hand symbols)

| Working string | Production |
|---|---|
| $\underline{a} + b * a$ | $F \to a$ |
| $\underline{F} + b * a$ | $T \to F$ |
| $\underline{T} + b * a$ | $E \to T$ |
| $E + \underline{b} * a$ | $F \to b$ |
| $E + \underline{F} * a$ | $T \to F$ |
| $E + T * \underline{a}$ | $F \to a$ |
| $E + \underline{T * F}$ | $T \to T * F$ |
| $\underline{E + T}$ | $E \to E + T$ |
| $E$ | |

Notice that when we read the series of steps in the parse from bottom to top, it forms a rightmost derivation of the input string.

# Sentential forms

A sentential form is a string of symbols (terminal and/or nonterminal) which can be derived from a grammar's start symbol.

- a.k.a. "working string"

A right-sentential form is a sentential form that can be derived using a rightmost derivation from a grammar's start symbol.

For example, $\boxed{\text{E} + \text{T} * \text{a}}$ is a right-sentential form of the grammar above, because it can be derived from the start symbol E using a rightmost derivation.

# Handles

As in top-down parsing, the main issue in bottom-up parsing is knowing when to apply a production. The notion of *handles* is helpful in making this determination.

Given a right sentential form $\alpha\beta$w, then $\beta$ is a handle of the sentential form if the production

$$\mathsf{A} \rightarrow \beta$$

is used in the rightmost derivation of the sentential form to expand

$$\alpha\mathsf{Aw}$$

into

$$\alpha\beta\mathsf{w}$$

# Handle pruning

If we can reliably find the handle $\beta$ of any right-sentential form, then we know how to reduce $\beta$ to make progress in the reverse construction of the rightmost derivation of the input string. This idea is called *handle pruning*.

Note that the string w (possibly empty) can only consist of terminal symbols. (If there were any nonterminal symbols in w, then reducing $\beta$ would not be part of a reversed rightmost derivation.)

# Shift/reduce parsing

Given the notion of handles and handle pruning, we can describe a simple approach, *shift/reduce parsing*, for bottom-up parsing.

Shift-reduce parsing uses two data structures:

- the input string, followed by the special $ terminal symbol to mark the end of the string
- a stack of symbols, initially containing the special $ symbol

At each step, a shift/reduce parser has two choices for what to do next:

1. Shift the next terminal symbol from the input string onto the stack
2. Reduce the handle which appears on the top of the stack

# Example shift/reduce parse

Using input string a + b * a

| Stack | Input string | Action |
| --- | --- | --- |
| $ | a + b * a $ | |

# Example shift/reduce parse

Using input string `a + b * a`

| Stack | Input string | Action |
|---|---|---|
| $ | a + b * a $ | shift |
| $ a | + b * a $ | reduce F → a |
| $ F | + b * a $ | reduce T → F |
| $ T | + b * a $ | reduce E → T |
| $ E | + b * a $ | shift |
| $ E + | b * a $ | shift |
| $ E + b | * a $ | reduce F → b |
| $ E + F | * a $ | reduce T → F |
| $ E + T | * a $ | shift |
| $ E + T * | a $ | shift |
| $ E + T * a | $ | reduce F → a |
| $ E + T * F | $ | reduce T → T * F |
| $ E + T | $ | reduce E → E + T |
| $ E | $ | accept |

# Parser construction

# Items, item sets

So, how can we automatically construct a bottom-up parser?

From the grammar, derive a set of *items*. An item represents, for a particular production, where the parser might be with respect to (eventually) reducing the production.

For example, in the production

$$E \rightarrow E + T$$

there are 4 items:

$$E \rightarrow \bullet\, E + T$$
$$E \rightarrow E \bullet + T$$
$$E \rightarrow E + \bullet\, T$$
$$E \rightarrow E + T \bullet$$

# What items mean

The dot ($\bullet$) indicates how close the parser is to reducing the production. In the case of the first item,

$$E \rightarrow \bullet\ E + T$$

the parser "wants" to apply the production, and is about to start working parsing the initial E nonterminal on the right-hand side of the production.

In the last item

$$E \rightarrow E + T\ \bullet$$

the parser has successfully parsed the entire right hand side of the production, and can consider reducing the production. (In other words, it is possible that the handle $E + T$ is now on the top of the stack.)

# The LR(0) automaton

A challenge of bottom-up parsing is that at any given point in parsing, many items may be "active" simultaneously. So, the parser uses the notion of *item sets* to represent this possibility.

The LR(0) automaton is an automaton we can construct to guide a bottom-up parser in making parsing decisions.

# LR(0) automaton construction

We start by constructing an augmented grammar. The augmented grammar adds a single production,

$S' \rightarrow S$

where S is the original grammar's start symbol. The parser will know that it has completed parsing when the original start symbol S is reduced to the augmented grammar's start symbol S'.

Next, we construct the canonical LR(0) item sets. To do so, we need to define two functions, CLOSURE and GOTO.

# CLOSURE function

The CLOSURE function builds a "complete" item set starting from an initial item set. CLOSURE(I), where I is an item set, is defined as follows:

> Repeatedly: If $\boxed{\text{A} \rightarrow \alpha \bullet \text{B} \beta}$ is an item in I, then add all items $\boxed{\text{B} \rightarrow \bullet \gamma}$ to CLOSURE(I).

The item sets constructed by the CLOSURE function form the states of the LR(0) automaton.

# GOTO function

The GOTO function is used the construct the transitions of the LR(0) automaton. GOTO(I, X). where I is an item set and X is a grammar symbol (terminal or nonterminal), is defined as follows

If $\boxed{A \rightarrow \alpha \bullet B \beta}$ is an item in I, then the closure of all items $\boxed{A \rightarrow \alpha B \bullet \beta}$ is in GOTO(I, B)

Basically, the GOTO function describes how the dot ($\bullet$) is moved when a nonterminal is reduced.

# LR(0) automaton construction algorithm

C is the set of all LR(0) item sets. To start, put CLOSURE($\{S' \rightarrow \bullet S\}$) in C.

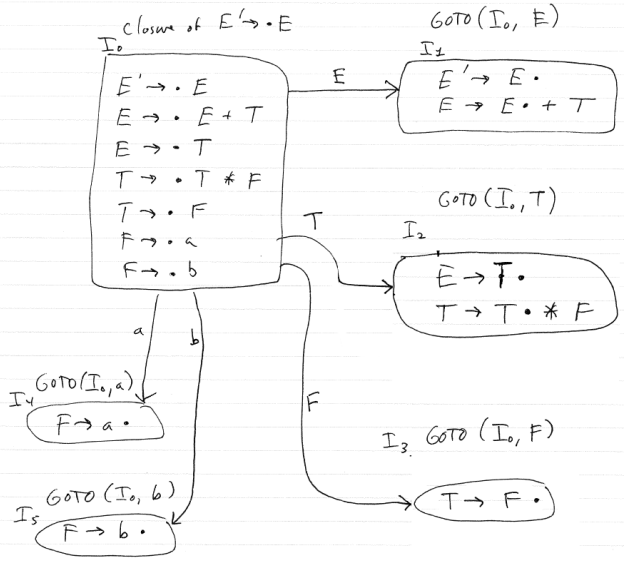
Repeat:

```
For each set I in C
    For each grammar symbol X
        if GOTO(I,X) is nonempty and is not already in C
            add GOTO(I, X) to C
            add a transition from I to GOTO(I, X) on symbol X
```

until no more item sets are added to C

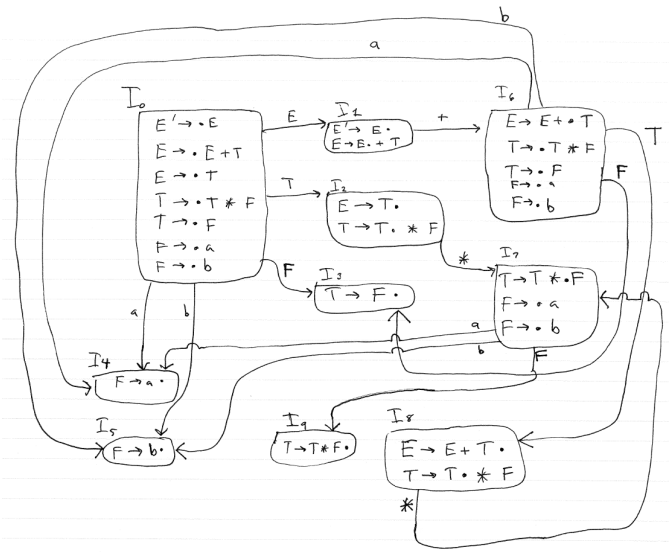
# LR(0) construction example

Part of the LR(0) automaton for our expression grammar, showing the initial item set $I_0$ and the item sets added by computing the GOTO sets from the initial item set



# Complete LR(0) automaton

To complete the automaton, we continue the algorithm, adding additional states and transitions as needed.



# SLR parsing

# SLR parsing

SLR ("Simple LR") is a simple bottom-up parsing algorithm that makes use of the LR(0) automaton.

Basic idea: keep a stack of states in addition to the input string and symbol stack. Each symbol on the symbol stack has a matching state on the state stack.

The top state tells us where we (currently) are in the LR(0) automaton.

The parser must decide, at each step, whether to shift a terminal symbol onto the stack or reduce a production. It uses the LR(0) automaton to guide this decision (explained on next slide.)

# SLR: shift or reduce?

If the current (top) state has a transition on the next terminal symbol in the input string,

- shift it onto the symbol stack
- whatever state is reached by following the transition, push the state on the state stack

Otherwise, reduce. One of the items in the current state will be of the form

$$A \rightarrow \alpha\bullet$$

indicating that the symbols matching $\alpha$ are cleared from the symbol stack and A pushed on. The states corresponding to $\alpha$ are cleared from the state stack. After clearing the states, we return to whatever state is currently on the top of the state stack and look for a transition on A. Whatever state the transition leads to is pushed on the state stack.

# Example SLR parse

| State Stack | Symbol Stack | Input | Action |
|---|---|---|---|
| $ 0 | $ | a + b * a $ | |

# Example SLR parse

| State Stack | Symbol Stack | Input | Action |
|---|---|---|---|
| $ 0 | $ | a + b * a $ | shift |
| $ 0 4 | $ a | + b * a $ | reduce F → a |
| $ 0 3 | $ F | + b * a $ | reduce T → F |
| $ 0 2 | $ T | + b * a $ | reduce E → T |
| $ 0 1 | $ E | + b * a $ | shift |
| $ 0 1 6 | $ E + | b * a $ | shift |
| $ 0 1 6 5 | $ E + b | * a $ | reduce F → b |
| $ 0 1 6 3 | $ E + F | * a $ | reduce T → F |
| $ 0 1 6 8 | $ E + T | * a $ | shift |
| $ 0 1 6 8 7 | $ E + T * | a $ | shift |
| $ 0 1 6 8 7 5 | $ E + T * b | $ | reduce F → b |
| $ 0 1 6 8 7 9 | $ E + T * F | $ | reduce T → T * F |
| $ 0 1 6 8 | $ E + T | $ | reduce E → E + T |
| $ 0 1 | $ E | $ | accept |

# Table-driven bottom-up parsing

# LR parse tables

There is a general way to construct a shift-reduce parser. The information about how to decide which action (shift/reduce/accept/error) the parser should take at any point is encoded in an *LR parse table*.

An LR parse table consists of columns for each terminal and nonterminal symbol. The terminal columns are the ACTION columns, while the nonterminal columns are the GOTO columns.

ACTION columns allow the parser to make a decision, based on the next terminal symbol in the input string, what action to take. GOTO columns determine how to determine which state to go to next when a production is reduced.

ACTION[i, a] is the ACTION entry for state i when terminal a is seen.
GOTO[i, A] is the state to go to when nonterminal A is reduced in state i.

# SLR parse table construction

Construct the LR(0) automaton for the grammar. Number the productions of the grammar.

For each transition from state i to k in the LR(0) automaton labeled with a terminal symbol a, set ACTION[i, a] to **s**k, meaning "shift state k onto the stack".

For each transition from state i to k in the LR(0) automaton labeled with a nonterminal symbol A, set GOTO[i, A] to k. This means that if we reduce a handle on the stack to an A nonterminal, clearing the handle from the stack and leaving state i on top of the stack, we should push state k.

# SLR parse table construction (continued)

For each state i with an item of the form A $\rightarrow \alpha \bullet$, where A is any nonterminal except S' (the start symbol of the augmented grammar), then for each terminal symbol a in FOLLOW(A), set ACTION[i, a] to **r**k, meaning "reduce production k", where k is the number of the production A $\rightarrow \alpha$.

If state i contains the item S' $\rightarrow$ S $\bullet$, then set ACTION[i, \$] to **acc**, meaning "accept".

The resulting parse table is called the SLR(1) parse table.

# Example SLR(1) table construction

Grammar productions (numbered):

(1) $E' \rightarrow E$
(2) $E \rightarrow E + T$
(3) $E \rightarrow T$
(4) $T \rightarrow T * F$
(5) $T \rightarrow F$
(6) $F \rightarrow a$
(7) $F \rightarrow b$

# FIRST and FOLLOW sets

| Nonterminal | FIRST set | FOLLOW set |
|---|---|---|
| E' | {a, b} | { $ } |
| E | {a, b} | { +, $ } |
| T | {a, b} | { *, +, $ } |
| F | {a, b} | { *, +, $ } |

# Constructed SLR(1) parse table

Based on LR(0) automaton shown previously; blank entries are *error* actions

| | ACTION | | | | | GOTO | | |
|---|---|---|---|---|---|---|---|---|
| **State** | **a** | **b** | **+** | ***** | **$** | **E** | **T** | **F** |
| 0 | s4 | s5 | | | | 1 | 2 | 3 |
| 1 | | | s6 | | acc | | | |
| 2 | | | r3 | s7 | r3 | | | |
| 3 | | | r5 | r5 | r5 | | | |
| 4 | | | r6 | r6 | r6 | | | |
| 5 | | | r7 | r7 | r7 | | | |
| 6 | s4 | s5 | | | | | 8 | 3 |
| 7 | s4 | s5 | | | | | | 9 |
| 8 | | | r2 | s7 | r2 | | | |
| 9 | | | r4 | r4 | r4 | | | |

# General LR table-driven parsing

Given an LR parse table, we can automate parsing of an input string.

Note that the general LR parser uses a stack of states rather than a stack of symbols; however, each state (except for state 0) is uniquely associated with a particular symbol, because all incoming edges to a particular state must be on the same symbol. (This is a trivial consequence of the way in which the LR(0) automaton is constructed; the transitions are generated by the computation of GOTO sets on specific symbols.)

# Example table-driven parse

| Step | Stack | Input | Action |
|---|---|---|---|
| 1 | $ 0 | a + b * a $ | |

# Example SLR(1) table-driven parse

| Step | Stack | Input | Action |
|---|---|---|---|
| 1 | $ 0 | a + b * a $ | shift 4 |
| 2 | $ 0 4 | + b * a $ | reduce 6 (F → a), goto 3 |
| 3 | $ 0 3 | + b * a $ | reduce 5 (T → F), goto 2 |
| 4 | $ 0 2 | + b * a $ | reduce 3 (E → T), goto 1 |
| 5 | $ 0 1 | + b * a $ | shift 6 |
| 6 | $ 0 1 6 | b * a $ | shift 5 |
| 7 | $ 0 1 6 5 | * a $ | reduce 7 (F → b), goto 3 |
| 8 | $ 0 1 6 3 | * a $ | reduce 5 (T → F), goto 8 |
| 9 | $ 0 1 6 8 | * a $ | shift 7 |
| 10 | $ 0 1 6 8 7 | a $ | shift 4 |
| 11 | $ 0 1 6 8 7 4 | $ | reduce 6 (F → a), goto 9 |
| 12 | $ 0 1 6 8 7 9 | $ | reduce 4 (T → T * F), goto 8 |
| 13 | $ 0 1 6 8 | $ | reduce 2 (E → E + T), goto 1 |
| 14 | $ 0 1 | $ | accept |

# Conflicts

# Conflicts in bottom-up parsing

Sometimes when we construct the parse tables we'll encounter a situation where we have conflicting actions in an entry. This may indicate that the grammar is ambiguous, or it could indicate that the parsing algorithm simply isn't powerful enough to handle the grammar.

Shift/reduce conflict: we don't know whether to shift a token or reduce a production.

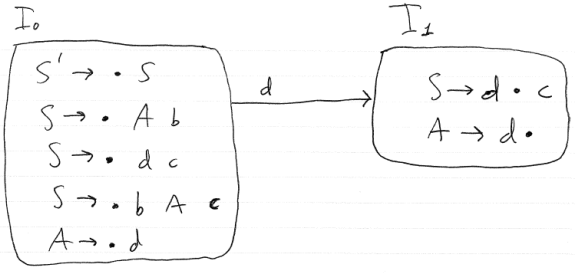
Reduce/reduce conflict: there are two possible productions which we could reduce.

# Example shift/reduce conflict

Grammar:

S' $\rightarrow$ S
S $\rightarrow$ A b
S $\rightarrow$ d c
S $\rightarrow$ b A c
A $\rightarrow$ d

Partial LR(0) automaton:



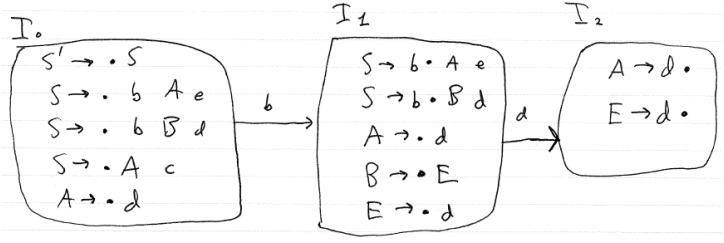

The problem is that the terminal symbol c is in the FOLLOW set of A (because of the S $\rightarrow$ b A c production), so when the parser sees a "c" in state 1, it doesn't know whether to shift or reduce.

# Example reduce/reduce conflict

Grammar:

S' → S
S → b A e
S → b B d
S → A c
A → d
B → E c
E → d

Partial LR(0) automaton:




In state 2, we won't know which production to reduce.

# LR(1) items

Idea: with each item, keep track of which terminal symbols could follow immediately after a reduction of the production. This is called an LR(1) item.

Example LR(1) item:

$A \to \alpha \bullet, a$

means that a state containing the item, the production $A \to \alpha$ is reduced if we see an a as the next input symbol.

Keeping track of this additional information gives the parser more context to decide when a production should be reduced.

# More powerful bottom-up parsing algorithms

There are more powerful ways to construct LR parse tables, based on LR(1) items:

- LR: powerful, but generates very large parse tables
- LALR: nearly as powerful as LR (at least for grammars that typically arise in the implementation of a programming language), but the parse tables are generally as small as SLR(1) tables

Most bottom-up parser generators (e.g., yacc, bison) use the LALR algorithm to construct the parse table.