Lecture 14+15

Bottom-Up Parsing

CS 241: Foundations of Sequential Programs
Winter 2018

Troy Vasiga et al
University of Waterloo

Example CFG

1. S' $\rightarrow \vdash$ S $\dashv$
2. S $\rightarrow$ AyB
3. A $\rightarrow$ ab
4. A $\rightarrow$ cd
5. B $\rightarrow$ z
6. B $\rightarrow$ wz

Stacks in LR Parsing

- Recall that a stack in LL/top-down parsing is used in the following way:

  input processed + stack = current derivation

  (Note that the stack here is read from the top to bottom)

- For LR/bottom-up parsing, we have

  stack+input to be read = current derivation

  (stack is read from bottom to top here)

A trace

Derivation	Stack	Input read	Unread Input	Action
$\vdash$ abywz $\dashv$	ϵ	ϵ	$\vdash$ abywz $\dashv$	Shift $\vdash$
$\vdash$ abywz $\dashv$	$\vdash$	$\vdash$	abywz $\dashv$	Shift a
$\vdash$ abywz $\dashv$	$\vdash$ a	$\vdash$ a	bywz $\dashv$	Shift b
$\vdash$ abywz $\dashv$	$\vdash$ a b	$\vdash$ ab	ywz $\dashv$	Reduce A $\rightarrow$ ab
$\vdash$ Aywz $\dashv$	$\vdash$ A	$\vdash$ ab	ywz $\dashv$	Shift y
$\vdash$ Aywz $\dashv$	$\vdash$ A y	$\vdash$ aby	wz $\dashv$	Shift w
$\vdash$ Aywz $\dashv$	$\vdash$ A y w	$\vdash$ abyw	z $\dashv$	Shift z
$\vdash$ Aywz $\dashv$	$\vdash$ A y w z	$\vdash$ abywz	$\dashv$	Reduce B $\rightarrow$ w z
$\vdash$ AyB $\dashv$	$\vdash$ A y B	$\vdash$ abywz	$\dashv$	Reduce S $\rightarrow$ AyB
$\vdash$ S $\dashv$	$\vdash$ S	$\vdash$ abywz	$\dashv$	Shift $\dashv$
$\vdash$ S $\dashv$	$\vdash$ S $\dashv$	$\vdash$ abywz $\dashv$	ϵ	Accept

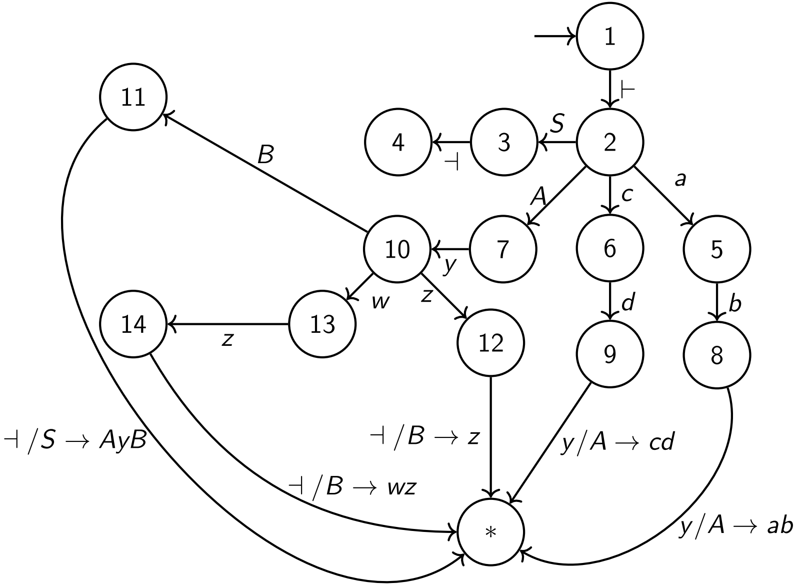
Shift: shifting a token from one place to another (push)

Reduce: size of the stack may be reduced (pop RHS, push LHS)

Shift/Reduce

- Somehow, we shifted at just the right time, and reduced just at the right time
- How did we know this?
  - Recall that for LL(1) parsing, we had a predictor table
  - For LR(1) parsing, we have an oracle, in the form of a DFA

An LR Oracle for the previous CFG



Constructing DFA oracle for LR(1) grammars

- This is difficult to do
  - Donald Knuth proved a theorem that we can construct a DFA (really, a transducer) for LR(1) grammars (1965)
  - This transducer tells us when to shift or reduce.
  - We will use the transducer (primarily) and build the DFA only for simple grammars which satisfy further restrictions of being LR(0) or SLR(1) (secondarily)

LR(0) ϵ-NFA formal definition

Given a CFG $G = (N, T, P, S)$, construct an ϵ-NFA $(Q, N \cup T, q_0, F, D)$ as follows:

- $Q = \{A \to \alpha \bullet \beta | A \to \alpha\beta \in P\}$
- $q_0 = \{S' \to \vdash \bullet S \dashv\}$
- $D[A \to \alpha \bullet X\beta, X] = \{A \to \alpha X \bullet \beta\}$
  - shift depending on whether X in T or N
- $D[A \to \alpha \bullet B\beta, \epsilon] = \{B \to \bullet\gamma | B \to \gamma \in P\}$
- $F = \{S' \to \vdash S \bullet \dashv\}$

The LR(0) DFA is the subset construction of this NFA, which recognizes a valid stack.
The LR parser has 2 actions:

- If you have stack K and input a, and Ka is recognized, you can **shift**.
- If you have stack K and input a, and the top of K is a state containing $A \to \alpha\bullet$ and a can follow A, **reduce** $A \to \alpha\bullet$
- If more than one of these is defined, you have a **conflict**.

Building an LR(0) automaton

- L –

- R –

- 0 –

Definition: An *item* is a production with a dot (•) somewhere on the RHS (which indicates a partially completed rule)

How to construct the automaton:

- make the start state the first rule, with the dot (•) in front of the left-most symbol of the RHS
- for each state, label an arc with the symbol that follows • and advance the • one position to the right in the next state.
- If the • precedes a non-terminal (e.g., A) add all productions with that non-terminal A on the LHS to the current state, with the • in the leftmost position

A sample construction of the DFA

Small example CFG:

1. S' $\rightarrow \vdash$ E $\dashv$
2. E $\rightarrow$ E $+$ T
3. E $\rightarrow$ T
4. T $\rightarrow$ id

Using the automaton

- For each input token
  - Start in the start state
  - Read the stack (from the bottom up) and read the current input, and do the action indicated for the current input
    - If there is a transition out of our current state on the current input, then *shift* (push) that input onto the stack
    - We know we can *reduce* if the current state has only one item and the • is the rightmost symbol
    - To *reduce*, pop the RHS off the stack, reread the stack (from the bottom-up), follow the transition for the LHS and push the LHS onto the stack
- Accept if S' on the stack when all input is read

Using the transducer

Example input: $\vdash$id+id+id$\dashv$

Stack	States visited	Input read	Unread Input	Action
ϵ	1	ϵ	$\vdash$id+id+id$\dashv$	shift
$\vdash$	1 2	$\vdash$	id+id+id$\dashv$	shift
$\vdash$ id	1 2 6	$\vdash$id	+id+id$\dashv$	reduce T$\rightarrow$id
$\vdash$ T	1 2 5	$\vdash$id	+id+id$\dashv$	reduce E$\rightarrow$T
$\vdash$ E	1 2 3	$\vdash$id	+id+id$\dashv$	shift
$\vdash$ E +	1 2 3 7	$\vdash$id+	id+id$\dashv$	shift
$\vdash$ E + id	1 2 3 7 6	$\vdash$id+id	+id$\dashv$	reduce T$\rightarrow$id
$\vdash$ E + T	1 2 3 7 8	$\vdash$id+id	+id$\dashv$	reduce E$\rightarrow$E+T
$\vdash$ E	1 2 3	$\vdash$id+id	+id$\dashv$	shift
$\vdash$ E +	1 2 3 7	$\vdash$id+id+	id$\dashv$	shift
$\vdash$ E + id	1 2 3 7 6	$\vdash$id+id+id	$\dashv$	reduce T$\rightarrow$id
$\vdash$ E + T	1 2 3 7 8	$\vdash$id+id+id	$\dashv$	reduce E$\rightarrow$E+T
$\vdash$ E	1 2 3	$\vdash$id+id+id	$\dashv$	shift
$\vdash$ E $\dashv$	1 2 3 4	$\vdash$id+id+id$\dashv$	ϵ	reduce S'$\rightarrow\vdash$E$\dashv$
S'	1	$\vdash$id+id+id$\dashv$	ϵ	accept

What can go wrong?

Two distinct problems:

Problem 1: What if the state looks like this?

$$\boxed{\begin{array}{c} \mathsf{A} \rightarrow \alpha\bullet\mathsf{c}\beta \\ \mathsf{B} \rightarrow\gamma\bullet \end{array}}$$

Do we try to shift the next character (as suggested by A $\rightarrow \alpha\bullet$c$\beta$) or do we reduce by B $\rightarrow\gamma$ (as suggested by B $\rightarrow\gamma\bullet$)?
This is known as a *shift-reduce conflict*.

Problem 2: What if the state looks like this?

$$\boxed{\begin{array}{c} \mathsf{A} \rightarrow \alpha\bullet \\ \mathsf{B} \rightarrow\beta\bullet \end{array}}$$

Do we reduce by A $\rightarrow \alpha$ or by B $\rightarrow \beta$?
This is known as a *reduce-reduce conflict*.

If any item A$\rightarrow\alpha\bullet$ occurs in a state in which it is not alone, then there is a shift-reduce or reduce-reduce conflict and the grammar is not LR(0).

Example with conflicts

Consider right-associative expressions. Modify our grammar slightly to allow (reverse RHS of second-rule).

1. S' $\rightarrow \vdash$ E $\dashv$
2. E $\rightarrow$ T $+$ E
3. E $\rightarrow$ T
4. T $\rightarrow$ id

DFA:

Parsing with conflicts

Suppose we are parsing a string that looks like ⊢id...

Picture of the stack:

Question: Should we reduce E→T?
Answer: It depends.

- If input is ⊢id⊣, then yes.
- If input is ⊢id+..., then no.

Looking ahead

If we add a lookahead token to the automaton, we can fix the conflict.

For each A$\rightarrow \alpha\bullet$, attach Follow(A).

For our grammar:

$$\text{Follow(E)} =$$

$$\text{Follow(T)} =$$

Consider our conflicting state: $\boxed{\begin{array}{c}\text{E}\rightarrow\text{T}\bullet \\ \text{E}\rightarrow\text{T}\bullet+\text{E}\end{array}}$

Interpretation: A reduce action $\boxed{\text{A}\rightarrow \alpha\bullet \quad\quad F}$ (where F is the Follow(A)) applies only if the next character is in F.

SLR(1) parser

When we add this one character of lookahead, we have an SLR(1) (Simple LR with 1 character of lookahead) parser
SLR(1) resolves many, but not all, conflicts.

- LR(1) parsing is more sophisticated than SLR(1) parsers
- LR(1) parses strictly more grammars
- LR(1) automaton is more complex
- LR(1) and SLR(1) are identical as parsing algorithms: the only difference is in the respective automaton they create

There is also a parser called LALR(1) (lookahead LR(1)), which falls between SLR(1) and LR(1).

- this is what Yacc and Bison use

Making this more efficient

Current running time of this algorithm:

Instead of scanning the stack each time...

Start the transducer in....

Running time:

Outputting a derivation

- Easy: each time we do a reduction, output the rule
- But, this isn't quite right. Derivations should start with the start symbol. Bottom-up parsing doesn't.

A simple observation

- Didn't we say that this was LR(1) parsing?
- Doesn't the "R" mean rightmost derivation?
- Aren't we always reducing the leftmost nonterminal?
- But notice the direction we are creating the derivation. Write the derivation in reverse.

Outputting the parse tree

Algorithm

- Create a "tree stack"
- Each time we reduce, pop the right hand side nodes from tree stack
- Push the left hand side node and make its children the nodes we just popped
- Example:

How the tree is actually built in LR parsing

How the tree is actually built in LL parsing

Assignment 8 hints

Note that the automaton in cfg-r format specifies:

state	symbol (terminal or non-terminal)	shift/reduce	next state / production number (for shifts) / (for reductions)

P1, P2: write a cfg-r derivation by hand

P3: Write a parser

- Read a CFG, the DFA and input
- Output cfg-r (derivation) if input is in the language, ERROR otherwise

P4: Write a parser for WLM

- Your parser will read tokens, output as it shifts.
- Find a way to embed the WLM grammar and DFA table in your program.

P5: Redo P4 to produce the output in `.wlmi` format.
Build a parse tree! Build a parse tree! Build a parse tree!

Going back

Looking at: $L = \{a^n b^m : n \geq m \geq 0\}$ (non-LL(1) language)

1. S' $\rightarrow \vdash$ S $\dashv$
2. S $\rightarrow$ a S
3. S $\rightarrow$ T
4. T $\rightarrow$ a T b
5. T $\rightarrow \epsilon$

What to do when you see the symbol:

- $\vdash$
- a
- b
- $\dashv$

Final fun facts

- Theorem: For any augmented LR(1) grammar, there is an equivalent LR(0) grammar.
- Theorem: The class of languages that can be parsed deterministically with a stack can be represented with an LR(1) grammar.
- Comparing LL(1) vs. LR(1)