LANGUAGE PROCESSORS

UNIT 6: BOTTOM-UP PARSING

David Griol Barres
dgriol@inf.uc3m.es
Computer Science Department
Carlos III University of Madrid
Leganés (Spain)

OUTLINE

- Bottom-up parsing
- LR(k) methods
  - Shift-reduce Parsing
  - LR Parsing Engine
  - Model of an LR parser
  - The LR Parsing Table
  - Constructing the canonical LR(0) collection
  - Limitations of LR(0) parsing
  - SLR(1)
  - LALR parser

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Bottom-up parsing

- A bottom-up parser starts with the string of terminals and builds the parse tree from the leaves upward, working backwards to the start symbol.
- The parsers searches for substrings of the working string that match the right side of some production. When such a substring is found, it substitutes it for the nonterminal on the left hand side of the production.

Bottom-up parsers

- Shift-reduce parsing:
  1. **Operator-precedence parsing:**
     - It chooses a specific action based on the precedence of the operators:
       - Not two consecutive nonterminals.
       - Not productions to ε.
       - Disjoint precedence relationships.
     - Specific analysis table.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Bottom-up parsers

- Example:

S → aAb

 | bAc

 | aAd

A → e

- We only take into account the symbol that is at the top of the stack → we may not come to a valid symbol sequence to reduce:
  - aA ⇨ {b c d} but taking the previous history {b d}

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Bottom-up parsers

- Shift-reduce parsing:

2. **LR: LR(0), SLR(1),LR(1),LALR(1)**

L : Read from left-to-right.

R : Rightmost derivation.

(k) : k look-ahead symbols (how many of them are needed to take the right decisions when parsing).

S : simple

LA : look-ahead

SLR(k) < LALR(k) < LR(k)

expressivity →

← complexity

LR(k) methods

- **Simple LR (SLR):**
  - The easiest to implement.
  - The least powerful.
- **Canonical LR:**
  - The most powerful.
  - The most expensive to implement.
- **LALR (lookahead LR):**
  - Intermediate in power and cost between the other two.

Bottom-up parsing: Shift-reduce parsers

- The largest class of grammars for which shift-reduce parsers can be built successfully are the LR grammars.
- For a small but important class of grammars (operator grammar) we can easily construct efficient shift-reduce parsers by hand.
- Automatic parser generators (e. g., yacc, CUP) generate an LALR(1) parser.

Bottom-up parsing: LR

- LR(k): left-to-right scanning, right-most derivation, k look-ahead characters.
- Advantages:
  - LR parsers can be constructed for virtually all programming language constructs for which a G2 grammar can be written.
  - The LR parsing method is the most general nonbacktracking shift-reduce parsing method known, yet it can be implemented efficiently.
  - An LR parser detects syntactic errors early.
- Drawback:
  - Too much work to construct an LR parser by hand.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Shift-reduce Parsing

- Parser state: a stack of terminals and non-terminals.
- Parsing actions: a sequence of *shift* and *reduce* operations.
  - shift: move lookahead token to stack.
  - reduce: replace symbols β from top of the stack with non terminal symbol A corresponding to production A ::= β

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Problem

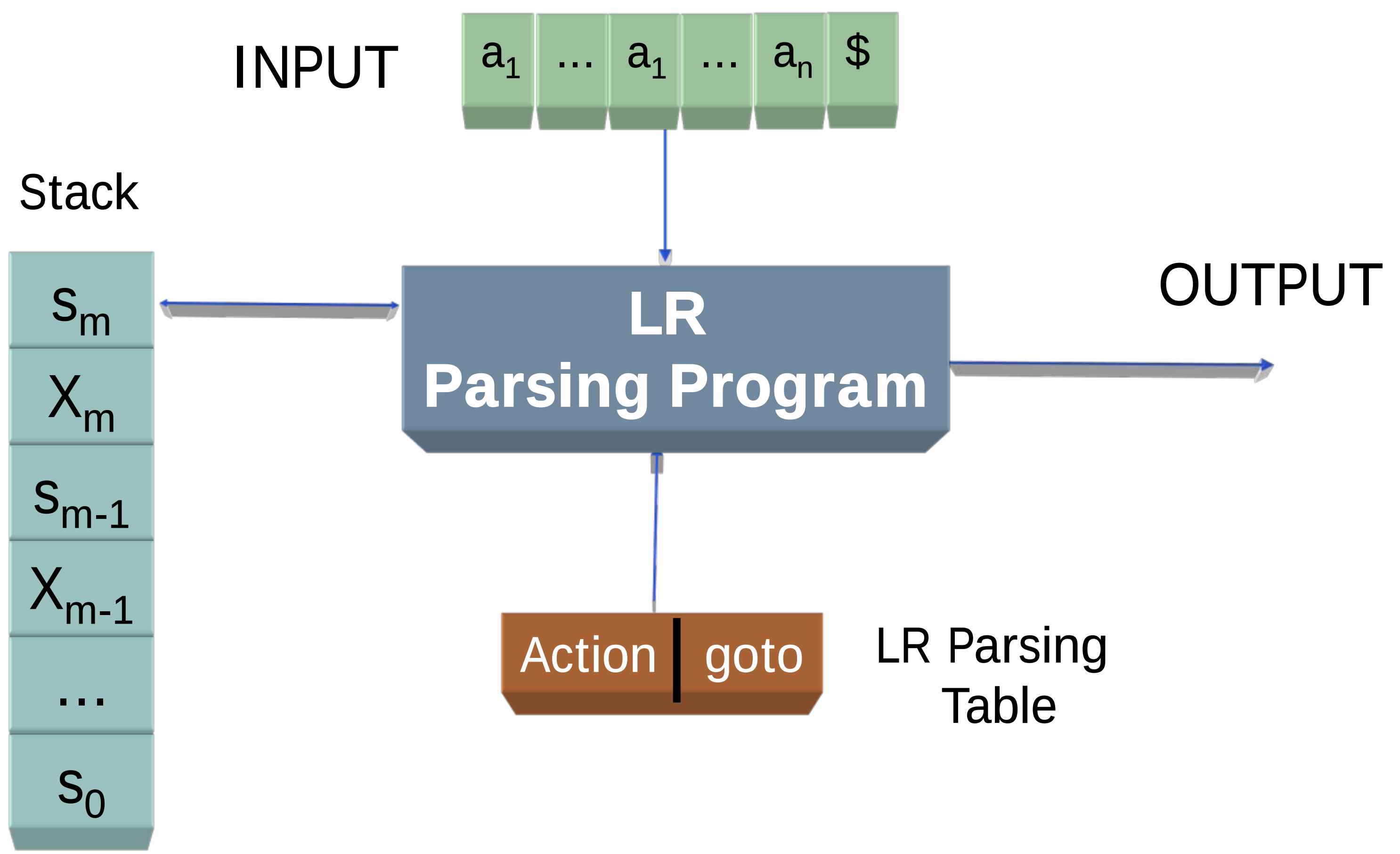
- How do we know which action to take: whether to shift or reduce, and which production?
- Issues:
  - Sometimes can reduce but should not.
  - Sometimes can reduce in different ways.

LR Parsing Engine

- Basic Mechanism:
  - Use a set of parser states.
  - Use a stack.
  - Use a parsing table to:
    - Determine what action to apply (shift/reduce).
    - Determine the next state.
- The parser actions can be precisely determined from the table.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Model of an LR parser



David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

The LR Parsing Table

	Terminals ∪ {λ}	Non terminals
State	Next action and next state	Next State
	Action table	**Goto table**

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

LR Parsing

- Let X_i be a grammar symbol and s_i a state symbol
- Parsing table
  - Action[*sm*, *ai*]=
    - Error: syntactic error
    - Accept: the input is accepted, end of parsing
    - Shift: push *ai* and the state *sm* onto the stack
    - Reduce: pops symbols from the stack
  - Goto[*sm*, X_i]= *sk*

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Constructing LR parsing tables

An LR(0) item of a grammar G is:

- A production of G with a dot at some position of the right side.
- The dot indicates how much of a production the parser has seen at a given point:

Example: production $A \rightarrow XYZ$ yields the following four items:

$A \rightarrow \bullet XYZ$

$A \rightarrow X \bullet YZ$

$A \rightarrow XY \bullet Z$

$A \rightarrow XYZ \bullet$

Question:
Which items are generated by the production $A \rightarrow \varepsilon$?

Constructing LR parsing tables

- Definitions
  - **Valid LR(0) Item**

    $A \rightarrow \beta_1 \bullet \beta_2$ is a valid item of $\alpha\beta_1$ iff:

    $S \rightarrow^* \alpha A w \rightarrow^* \alpha\beta_1\beta_2 w \quad (A \in \Sigma_N, \alpha, \beta_1, \beta_2 \in \Sigma^*, w \in \Sigma^*_T)$
  - **State**
    - Set of items.
    - States of the parser.
    - The set of states: ***canonical LR(0) collection***

- The items are the states of a FA which recognizes viable prefixes.
- The states are groups of the FA states (FA minimization).

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Constructing LR parsing tables

- Input:
  1. **Augmented grammar** G'
  2. ***closure***(I), $I \equiv$ *set of items*
  3. ***goto***(I, X), $X \in (\Sigma_T \cup \Sigma_N)$
- Output
  - canonical LR(0) collection
- **Augmented grammar G' of G**
  - Add S', $\Sigma_N = (\Sigma_N \cup S')$ | S' axiom
  - Add $S' \rightarrow S$, $P = (P \cup S' \rightarrow S)$

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Constructing the canonical LR(0) collection

G

1. S → A B end
2. A → type
3. A → id A
4. B → begin C
5. C → code

G'

1. S' → S
2. S → A B end
3. A → tipo
4. A → id A
5. B → begin C
6. C → code

LR(0) items:

I_0: S' → •S
S → •A B end
A → •type
A → •id A

I_1: S' → S•

I_2: S → A•B end
B → •begin C

I_3: A → type•

I_4: A → id•A
A → •type
A → •id A

I_5: S → A B•end

I_6: B → begin•C
C → •code

I_7: A → id A•

I_8: S → A B end•

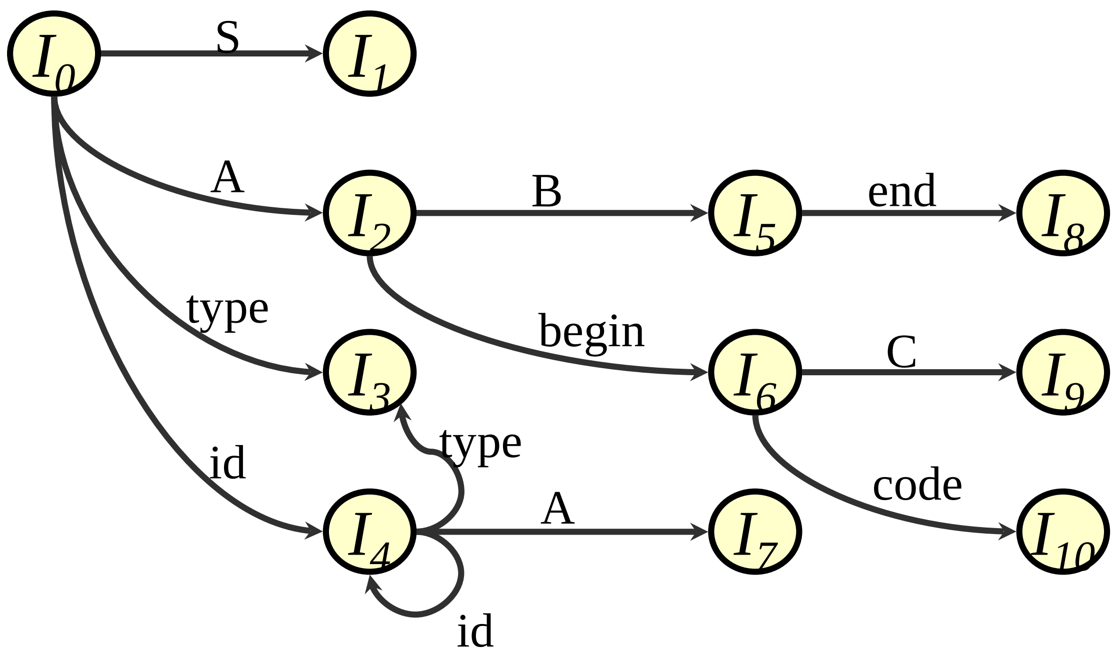
I_9: B → begin C•

I_{10}: C → code•

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Constructing the canonical LR(0) collection

- The canonical collection defines a DFA which recognizes the viable prefixes of G, where I_0 is the initial state and $I_j \forall j \neq 0$ the final states



David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Constructing the canonical LR(0) collection

- ***closure***(*I*)

function ***closure***(***I***);
begin
  $J := I$;
  repeat
    for $\forall J_i\ (A \rightarrow \alpha \bullet B\beta) \in J,\ \forall p\ (B \rightarrow \gamma) \in P \mid (B \rightarrow \bullet\, \gamma) \notin J$
      do $J := J \cup (B \rightarrow \bullet\, \gamma)$;
  until no more items can be added to *J* ;
  return *J*
end

- Example:

$A \rightarrow B$
$B \rightarrow id \mid C\ num \mid (\ D\ )$
$C \rightarrow +\ D$
$D \rightarrow id \mid num$

¿ $closure(A \rightarrow \bullet\, B)$?

$A \rightarrow \bullet\, B$
$B \rightarrow \bullet\, id \mid \bullet\, C\ num \mid \bullet(\ D\ )$
$C \rightarrow \bullet\, +\ D$

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Constructing the canonical LR(0) collection

- ***goto*(I, X)**
  - If I is the set of items that are valid for some viable prefix γ, then goto(I, X) is the set of items that are valid for the viable prefix γX

function ***goto*(*I*, *X*)**;
begin
  $J := \emptyset$;
  $\forall\ I_i \mid (B \rightarrow \alpha \bullet X\ \beta) \in I,\ J := J \cup$ ***closure***$(B \rightarrow \alpha\ X \bullet \beta)$;
  return J
end

$B \rightarrow (\bullet D\ )$
$D \rightarrow \bullet\ id$
$D \rightarrow \bullet\ num$

- Example:
  $A \rightarrow B$
  $B \rightarrow id \mid C\ num \mid (\ D\ )$
  $C \rightarrow +\ D$
  $D \rightarrow id \mid num$

$I = \{B \rightarrow \bullet\ id,\ B \rightarrow \bullet\ (\ D\ )\}$
¿ $goto\ \{I, (\ \}$?

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Constructing the canonical LR(0) collection

- The algorithm to construct the canonical collection of sets of LR(0) is as follows:
  1. I_o is defined as closure([S'→ ·S])
  2. I_n = goto(I_{n-1},N) $\forall N \in (\Sigma_T \cup \Sigma_N)$ for which $\exists$ [$A \rightarrow \alpha \cdot N\beta$] $\in I_{n-1}$
     $A \in \Sigma_N$, $\alpha\ \beta \in (\Sigma_T \cup \Sigma_N \cup \varepsilon)$
  3. Apply step 2 until no new states are generated.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Constructing the analysis table

	Action				Goto		
Sets	Non terminal 1	…	Non terminal m	$	Terminal 1	…	Terminal m'
I_0							
…							
I_n							

1. Construct the canonical collection of sets (previous slide).
2. Determine Actions for each Set
   1. If [A → α ·aβ] ∈ Ii, a ∈ Σ_T and goto(Ii, α)=Ij then Action(i, α) = (Shift, j)
   2. If [S' → S ·] ∈ Ii, then Action(i, $) = Accept
   3. If [A → α ·] ∈ Ii, and A is not S', then for every a ∈ FOLLOW(A), Action(i,a) = (Reduce, A → α)
3. Determine Gotos for each Non terminal
   1. If goto(Ii,A) = Ij, then goto(i,A) = j

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

LR Parsing

- A configuration of an LR parser
  - $(s_0\ X_1\ s_1\ X_2\ s_2\ \ldots X_m\ s_m,\ a_i\ a_{i+1}\ \ldots\ a_n\$)$
- $Action[s_m, a_i]$ = shift s
  - $(s_0\ X_1\ s_1\ X_2\ s_2\ \ldots X_m\ s_m\ \boldsymbol{a_i}\ \boldsymbol{s},\ a_{i+1}\ \ldots\ a_n\$)$
- $Action[s_m, a_i]$ = reduce $A \rightarrow \beta$
  - $(s_0\ X_1\ s_1\ X_2\ s_2\ \ldots X_{m-r}\ s_{m-r}\ \boldsymbol{A}\ \boldsymbol{s},\ a_i\ a_{i+1}\ \ldots\ a_n\$)$

    where $s=Goto[s_{m-r}, A]$ and $r=|\beta|$ (r non-terminal symbols and r terminal symbols are extracted from the stack)

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

LR parsing algorithm

Set *ip* to point the first symbol of *w*\$ (*s* is on top of the stack and ip points to the a symbol)

```
Repeat forever begin
    case Action[s, a]
        Shift s'
            push a
            push s'
            advance ip to the next input symbol
        Reduce A → β
            pop 2*|β| symbols from the stack
            let s' be the state now on top of the stack
            s= Goto[s', A]
            push A
            push s
        Acept return
        Error error()
end
```


David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

LR(0) summary

- LR(0) parsing recipe:
  - Start with an LR(0) grammar.
  - Compute LR(0) states and build DFA.
  - Build the LR(0) parsing table form the DFA.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Limitations of LR(0) parsing

- Very few grammars are LR(0).
- For other grammars: shift/reduce and reduce/reduce conflicts.
- The limitations are caused by trying to decide what action to take only by considering what has been seen so far.

SLR(1)

- Take into account the symbol that follows the current input.
- The concepts of *item*, *closure*, and *goto* are extended by adding the look-ahead symbol.
- Uses the set of elements defined for LR(0).
- Specific algorithm to construct the analysis table:
  - Input: augmented grammar.
  - Output: action, goto.

LALR parser

- Motivation
  - Often used in practice because has less states than the canonical LR (LALR and SLR have the same number of states)
- Merge sets of LR(1) states with the same core
  - If the I_i state contains [A->$\alpha \bullet \beta$, a] and state I_j contains [A->$\alpha \bullet \beta$, b] we can form a union state I_{ij} where [A->$\alpha \bullet \beta$, a/b]
- LALR(1) grammars are a subset of LR(1) grammars.
  - Merging may produce reduce/reduce conflicts, but no shift-reduce conflicts
  - Some errors may appear later