LR(0) Parsers

CSCI 3130 Formal Languages and Automata Theory

Siu On CHAN
Fall 2018

Chinese University of Hong Kong
if (n == 0) { return x; }

First phase of javac compiler: lexical analysis

if (ID == INT_LIT) { return ID; }

The alphabet of Java CFG consists of tokens like

\[ \Sigma = \{ \text{if}, \text{return}, (, ), {, }, ;, ==, \text{ID}, \text{INT\_LIT}, \ldots \} \]
Parsing computer programs

if (n == 0) { return x; }

Parse tree of a Java statement
Identifier:
   IdentifierChars but not a Keyword or BooleanLiteral or NullLiteral
Literal:
   IntegerLiteral
   FloatingPointLiteral
   BooleanLiteral
   CharacterLiteral
   StringLiteral
   NullLiteral
Expression:
   LambdaExpression
   AssignmentExpression
AssignmentOperator:
   (one of) = *= /= %= += -= <<= >>= >>>= &= ^= |=

class Point2d {
    /* The X and Y coordinates of the point--instance variables */
    private double x;
    private double y;
    private boolean debug; // A trick to help with debugging

    public Point2d (double px, double py) { // Constructor
        x = px;
        y = py;
        debug = false; // turn off debugging
    }

    public Point2d () { // Default constructor
        this (0.0, 0.0); // Invokes 2 parameter Point2D constructor
    }
    // Note that a this() invocation must be the BEGINNING of
    // statement body of constructor

    public Point2d (Point2d pt) { // Another constructor
        x = pt.getX();
        y = pt.getY();
    }
    ...
}

Simple Java program: about 1000 tokens
Parsing algorithms

How long would it take to parse this program?

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>try all parse trees</td>
<td>$\geq 10^{80}$ years</td>
</tr>
<tr>
<td>CYK algorithm</td>
<td>hours</td>
</tr>
</tbody>
</table>

Can we parse faster?

CYK is the fastest known general-purpose parsing algorithm for CFGs.

Luckily, some CFGs can be rewritten to allow for a faster parsing algorithm!
Hierarchy of context-free grammars

context-free grammars

LR(∞) grammars

LR(1) grammars

LR(0) grammars

Java, Python, etc have LR(1) grammars

We will describe LR(0) parsing algorithm
A grammar is LR(0) if LR(0) parser works correctly for it
LR(0) parser: overview

\[ S \rightarrow SA \mid A \]
\[ A \rightarrow (S) \mid () \]

Input: \((())()\)

1. \(\bullet((())())\)
2. \((⊙)(())\)
3. \((())⊙()\)
4. \(A⊙()\)
5. \(S⊙()\)
6. \(S(⊙)\)
7. \(S(⊙)⊙()\)
8. \(S(⊙)A⊙()\)
9. \(S⊙()\)

8/31
LR(0) parser: overview

Features of LR(0) parser:

• Greedily reduce the recently completed rule into a variable
• Unique choice of reduction at any time

```
S → SA | A
A → (S) | ( )
```

input: ( ) ( )
To speed up parsing, keep track of partially completed rules in a PDA $P$.

In fact, the PDA will be a simple modification of an NFA $N$.

The NFA accepts if a rule $B \rightarrow \beta$ has just been completed and the PDA will reduce $\beta$ to $B$.

\[ ... \Rightarrow 2 \ (\bullet) \ (\ ) \Rightarrow 3 \ (\ ) \ (\bullet) \ (\ ) \Rightarrow 4 \ A \ (\bullet) \ (\ ) \Rightarrow 5 \ S \ (\bullet) \ (\ ) \Rightarrow ... \]

\[
\begin{array}{c}
( \ ) \\
A \\
( \ )
\end{array}
\]

\[\checkmark: \ NFA \ N \ accepts\]
A rule $B \rightarrow \beta$ has just been completed if

Case 1  input/buffer so far is exactly $\beta$

Examples: 3 $(\ )\bullet(\ )$ and 4 $A\bullet(\ )$

Case 2  Or buffer so far is $\alpha\beta$ and there is another rule $C \rightarrow \alpha B\gamma$

Example: 7 $S(\ )\bullet$

$\quad A$

$\quad \quad \quad (\ )$

This case can be chained
Designing NFA for Case 1

\[
\begin{align*}
S & \rightarrow SA \mid A \\
A & \rightarrow (S) \mid ()
\end{align*}
\]

Design an NFA \( N' \) to accept the right hand side of some rule \( B \rightarrow \beta \)
Designing NFA for Case 1

Design an NFA \( N' \) to accept the right hand side of some rule \( B \rightarrow \beta \)

\[
S \rightarrow SA | A \\
A \rightarrow (S) | ( )
\]
Designing NFA for Cases 1 & 2

Design an NFA $N$ to accept $\alpha\beta$ for some rules $C \rightarrow \alpha B \gamma$, $B \rightarrow \beta$ and for longer chains

$$S \rightarrow SA \mid A$$

$$A \rightarrow (S) \mid ()$$
Design an NFA $N$ to accept $\alpha\beta$ for some rules $C \to \alpha B \gamma, \quad B \to \beta$
and for longer chains

For every rule $C \to \alpha B \gamma, \quad B \to \beta$, add $C \to \alpha \bullet B \gamma$

All blue arrows are $\varepsilon$-transitions.
Summary of the NFA

For every rule $B \rightarrow \beta$, add

$$
\begin{array}{c}
\text{q0} \\
\leftarrow \varepsilon \\
\rightarrow B \rightarrow \bullet \beta
\end{array}
$$

For every rule $B \rightarrow \alpha X \beta$ ($X$ may be terminal or variable), add

$$
\begin{array}{c}
B \rightarrow \alpha \bullet X \beta \\
\rightarrow X \\
\rightarrow B \rightarrow \alpha X \bullet \beta
\end{array}
$$

Every completed rule $B \rightarrow \beta$ is accepting

$$
\begin{array}{c}
B \rightarrow \beta \bullet
\end{array}
$$

For every rule $C \rightarrow \alpha B \gamma$, $B \rightarrow \beta$, add

$$
\begin{array}{c}
C \rightarrow \alpha \bullet B \gamma \\
\leftarrow \varepsilon \\
\rightarrow B \rightarrow \bullet \beta
\end{array}
$$

The NFA $N$ will accept whenever a rule has just been completed
Equivalent DFA \( D \) for the NFA \( N \)

Observation: every accepting state contains only one rule: a completed rule \( B \rightarrow \beta \bullet \), and such rules appear only in accepting states.
A grammar $G$ is LR(0) if its corresponding $D_G$ satisfies:

Every accepting state contains only one rule:
- a completed rule of the form $B \rightarrow \beta \cdot$
- and completed rules appear only in accepting states

**Shift state:**
- no completed rule

$S \rightarrow S \cdot A$
$A \rightarrow \bullet (S)$
$A \rightarrow \bullet ()$

**Reduce state:**
- has (unique) completed rule

$A \rightarrow (S) \bullet$
Simulating DFA $D$

Our parser $P$ simulates state transitions in DFA $D$

$((())\bullet) \Rightarrow (A\bullet)\
(())$

After reducing $(())$ to $A$, what is the new state?

Solution: keep track of previous states in a stack
go back to the correct state by looking at the stack
Let's label $D$'s states
LR(0) parser: a “PDA” $P$ simulating DFA $D$

$P$’s stack contains labels of $D$’s states to remember progress of partially completed rules

At $D$’s non-accepting state $q_i$

1. $P$ simulates $D$’s transition upon reading terminal or variable $X$
2. $P$ pushes current state label $q_i$ onto its stack

At $D$’s accepting state with completed rule $B \rightarrow X_1 \ldots X_k$

1. $P$ pops $k$ labels $q_k, \ldots, q_1$ from its stack
2. constructs part of the parse tree

3. $P$ goes to state $q_1$ (last label popped earlier), pretend next input symbol is $B$
<table>
<thead>
<tr>
<th>State</th>
<th>Stack</th>
<th>Push/Pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>q1 $</td>
<td>$(()()$</td>
</tr>
<tr>
<td>2</td>
<td>q5 $1</td>
<td>$1$(()()$</td>
</tr>
<tr>
<td>3</td>
<td>q8 $15</td>
<td>$15$(()()$</td>
</tr>
<tr>
<td>4</td>
<td>q4 $1</td>
<td>$1$(()$</td>
</tr>
<tr>
<td>5</td>
<td>q2 $1</td>
<td>$1$S$(())$</td>
</tr>
<tr>
<td>6</td>
<td>q5 $12</td>
<td>$12$S$(())$</td>
</tr>
</tbody>
</table>

The diagram on the right shows a parse tree for the string $(()()$.
Example

<table>
<thead>
<tr>
<th>State</th>
<th>Stack</th>
<th>state</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>S ( )</td>
<td>q_8</td>
<td>$125</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>A</td>
<td>q_2</td>
<td>$1</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>S</td>
<td>q_3</td>
<td>$12</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parser’s output is the parse tree</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Another LR(0) grammar

\[ L = \{ w\# w^R \mid w \in \{a, b\}^* \} \]

\[ C \rightarrow a Ca \mid bCb \mid \# \]

NFA \( N \):

- \( q_0 \)
- \( C \rightarrow \bullet a Ca \)
- \( C \rightarrow a \bullet Ca \)
- \( C \rightarrow a C \bullet a \)
- \( C \rightarrow a Ca \bullet \)
- \( C \rightarrow \bullet b Cb \)
- \( C \rightarrow b \bullet Cb \)
- \( C \rightarrow b C \bullet b \)
- \( C \rightarrow b Cb \bullet \)

Transitions:

- \( \varepsilon \) transitions:
  - \( q_0 \) to \( C \rightarrow \bullet a Ca \)
  - \( C \rightarrow \bullet a Ca \) to \( C \rightarrow a \bullet Ca \)
  - \( C \rightarrow \bullet a Ca \) to \( C \rightarrow a C \bullet a \)
  - \( C \rightarrow \bullet a Ca \) to \( C \rightarrow a Ca \bullet \)
  - \( q_0 \) to \( C \rightarrow \bullet b Cb \)
  - \( C \rightarrow \bullet b Cb \) to \( C \rightarrow b \bullet Cb \)
  - \( C \rightarrow \bullet b Cb \) to \( C \rightarrow b C \bullet b \)
  - \( C \rightarrow \bullet b Cb \) to \( C \rightarrow b Cb \bullet \)

- \( a \) transitions:
  - \( C \rightarrow a \bullet Ca \) to \( C \rightarrow a C \bullet a \)
  - \( C \rightarrow a Ca \) to \( C \rightarrow a Ca \bullet \)
  - \( C \rightarrow a C \bullet a \) to \( C \rightarrow a Ca \bullet \)
  - \( C \rightarrow a Ca \bullet \) to \( C \rightarrow a Ca \bullet \)

- \( b \) transitions:
  - \( C \rightarrow b \bullet Cb \) to \( C \rightarrow b C \bullet b \)
  - \( C \rightarrow b C \bullet b \) to \( C \rightarrow b Cb \bullet \)

- \( \# \) transitions:
  - \( C \rightarrow \bullet \# \) to \( C \rightarrow \# \bullet \)
  - \( C \rightarrow \# \bullet \) to \( C \rightarrow \# \bullet \)

Start state: \( q_0 \)

Accept state: \( C \rightarrow a Ca \bullet \)

Transition: \( a \rightarrow A \)

Transition: \( b \rightarrow B \)
Another LR(0) grammar

\[ C \rightarrow aCa \mid bCb \mid \# \]

Input: \( ba\#ab \)

```
stack  state  action
$      1      S
$1     4      S
$14    3      S
$143   2      R
$143   5      S
$1435  7      R
$14    6      S
$146   8      R
```

Diagram showing the LR(0) parsing process.
PDA for LR(0) parsing is **deterministic**

Some CFLs require non-deterministic PDAs, such as

\[ L = \{ww^R \mid w \in \{a, b\}^*\} \]

What goes wrong when we do LR(0) parsing on \( L \)?
Example 2

\[ L = \{ww^R \mid w \in \{a, b\}^*\} \]

\[ C \rightarrow aCa \mid bCb \mid \varepsilon \]

NFA \( N \):

\[ C \rightarrow \bullet aCa \]

\[ C \rightarrow a\bullet Ca \]

\[ C \rightarrow aCa \]

\[ C \rightarrow b\bullet Cb \]

\[ C \rightarrow bCb \]

\[ C \rightarrow a \]

\[ C \rightarrow b \]
Example 2

\[
C \rightarrow \bullet \text{a} \text{C} \text{a}
\]

\[
C \rightarrow \bullet \text{b} \text{C} \text{b}
\]

\[
C \rightarrow \bullet
\]

\[
C \rightarrow \bullet \text{a} C \text{a}
\]

\[
C \rightarrow \bullet \text{a} C' \text{a}
\]

\[
C \rightarrow \bullet \text{b} C' \text{b}
\]

\[
C \rightarrow \bullet \text{b} C' \text{b}
\]

\[
C \rightarrow \bullet
\]

\[
C \rightarrow \bullet \text{a} \text{C} \text{a}
\]

\[
C \rightarrow \bullet \text{a} \text{C} \text{a}
\]

\[
C \rightarrow \bullet \text{b} \text{C} \text{b}
\]

\[
C \rightarrow \bullet \text{b} \text{C} \text{b}
\]

\[
C \rightarrow \bullet
\]

\[
C \rightarrow \bullet \text{a} \text{C} \text{a} \
C \rightarrow \bullet \text{b} \text{C} \text{b} \
C \rightarrow \varepsilon
\]

\[
C \rightarrow \text{a} \text{C} \text{a} | \text{b} \text{C'} \text{b} | \varepsilon
\]

shift-reduce conflicts
Motivation: Fast parsing for programming languages
LR(1) Grammar: A few words
LR(0) grammar revisited

LR(1) grammars
LR(0) grammars

LR(0) parser: Left-to-right read, Rightmost derivation, 0 lookahead symbol

Derivation
\[ S \rightarrow SA \rightarrow S() \rightarrow A() \rightarrow ()() \]

Reduction (derivation in reverse)
\[ ()() \rightarrow A() \rightarrow S() \rightarrow SA \rightarrow S \]

LR(0) parser looks for rightmost derivation

Rightmost derivation = Leftmost reduction
if (n == 0) { return x; }

if (ParExpression | Statement) | (Expression | Statement):

CFGs of most programming languages are not LR(0). LR(0) parser cannot tell apart if… then from if… then… else.
Parsing computer programs

if (n == 0) { return x; } else { return x + 1; }

CFGs of most programming languages are not LR(0)

LR(0) parser cannot tell apart

if ...then from if ...then ...else
LR(1) grammars resolve such conflicts by **one symbol lookahead**

States in NFA $N$

- LR(0):
  \[ A \rightarrow \alpha \cdot \beta \]
- LR(1):
  \[ [A \rightarrow \alpha \cdot \beta, a] \]

States in DFA $D$

- LR(0):
  no shift-reduce conflicts
  no reduce-reduce conflicts
- LR(1):
  some shift-reduce conflicts allowed
  some reduce-reduce conflicts allowed
  as long as can be resolved with lookahead symbol $a$

We won’t cover LR(1) parser in this class; take CSCI 3180 for details