LR(0) Parsers

CSCI 3130 Formal Languages and Automata Theory

Siu On CHAN
Fall 2019

Chinese University of Hong Kong
First phase of `javac` compiler: lexical analysis

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
CFG of the java programming language

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(\infty) 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: ( )()
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

input: ( ) ( )

\[
S \rightarrow SA \mid A \\
A \rightarrow (S') \mid ( )
\]
LR(0) parsing using a PDA

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 (●) ( ) $\Rightarrow$ 3 ( )●( ) $\Rightarrow$ 4 $A$●( ) $\Rightarrow$ 5 $S$●( ) $\Rightarrow$ ...

✓: 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 $4A\bullet(\ )$

Case 2  Or buffer so far is $\alpha\beta$ and there is another rule $C \rightarrow \alpha B\gamma$
   Example: $7S(\ )\bullet$

This case can be chained
Designing NFA for Case 1

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

\[
S \rightarrow SA \mid A \\
A \rightarrow (S) \mid ()
\]
Designing NFA for Case 1

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

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

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

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

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

![Transition Diagram]

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

![Transition Diagram]

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

![Graph]

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

![Transition Diagram]

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

Dead state (empty set) not shown for clarity

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

- $S \rightarrow S \cdot A$
- $A \rightarrow \cdot(S')$
- $A \rightarrow \cdot()$

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

- $A \rightarrow (S)\cdot$
Simulating DFA $D$

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

\[
\text{( ( )•) } \quad \Rightarrow \quad \text{( ( ) ) )}
\]

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 \( B \to X_1 \ldots X_k \)

1. \( P \) pops \( k \) labels \( q_k, \ldots, q_1 \) from its stack
2. constructs part of the parse tree
   \[
   \begin{array}{c}
   B \\
   \leftarrow / \\
   X_1 \quad X_2 \quad \ldots \quad X_k
   \end{array}
   \]
3. \( P \) goes to state \( q_1 \) (last label popped earlier), pretend next input symbol is \( B \)
## Example

<table>
<thead>
<tr>
<th>State</th>
<th>Stack</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$</td>
<td>()()()</td>
</tr>
<tr>
<td>2</td>
<td>$1</td>
<td>(())</td>
</tr>
<tr>
<td>3</td>
<td>$15</td>
<td>()•()</td>
</tr>
<tr>
<td>4</td>
<td>$1</td>
<td>A(())</td>
</tr>
</tbody>
</table>

### Stack Diagram

```
S
A
( )

state stack

5  S •( ) q2 $1

6  S (•) q5 $12
```
Example

<table>
<thead>
<tr>
<th>State</th>
<th>Stack</th>
<th>State</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>$S(\ )\bullet$</td>
<td>$q_8$</td>
<td>$$125$</td>
</tr>
<tr>
<td></td>
<td>$A$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$(\ )$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>$S \bullet A$</td>
<td>$q_2$</td>
<td>$$1$</td>
</tr>
<tr>
<td></td>
<td>$A$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$(\ )$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$(\ )$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$S \bullet A$</td>
<td>$q_3$</td>
<td>$$12$</td>
</tr>
<tr>
<td></td>
<td>$A$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$(\ )$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

parser’s output is the parse tree
Another LR(0) grammar

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

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

NFA \(N\):

\(C \rightarrow \bullet aCa\)
\(C \rightarrow a \bullet Ca\)
\(C \rightarrow a C \bullet a\)
\(C \rightarrow aC\bullet a\)
\(C \rightarrow \bullet bCb\)
\(C \rightarrow b \bullet Cb\)
\(C \rightarrow bC \bullet b\)
\(C \rightarrow bCb\bullet\)

\(q_0\)
Another LR(0) grammar

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

Input: \( ba\#ab \)

<table>
<thead>
<tr>
<th>Stack</th>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>1</td>
<td>S</td>
</tr>
<tr>
<td>$1</td>
<td>4</td>
<td>S</td>
</tr>
<tr>
<td>$14</td>
<td>3</td>
<td>S</td>
</tr>
<tr>
<td>$143</td>
<td>2</td>
<td>R</td>
</tr>
<tr>
<td>$1435</td>
<td>5</td>
<td>S</td>
</tr>
<tr>
<td>$14</td>
<td>6</td>
<td>S</td>
</tr>
<tr>
<td>$146</td>
<td>8</td>
<td>R</td>
</tr>
</tbody>
</table>
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 \( N \):
Example 2

$C \rightarrow \bullet a Ca$
$C \rightarrow \bullet b Cb$
$C \rightarrow \bullet$

$C \rightarrow a \bullet Ca$
$C \rightarrow \bullet a Ca$
$C \rightarrow \bullet b Cb$
$C \rightarrow \bullet$

$C \rightarrow b \bullet Cb$
$C \rightarrow \bullet a Ca$
$C \rightarrow \bullet b Cb$
$C \rightarrow \bullet$

$C \rightarrow a Ca | b Cb | \varepsilon$

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

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

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

**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 (n == 0) {
    return x;
}
```

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

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

CFGs of most programming languages are not LR(0)

LR(0) parser cannot tell apart

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

States in NFA $N$

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

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