LR(0) Parsers

CSCI 3130 Formal Languages and Automata Theory

Siu On CHAN
Fall 2020

Chinese University of Hong Kong
Parsing computer programs

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

First phase of `javac` compiler: lexical analysis

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

The alphabet of Java CFG consists of tokens like

\[
\Sigma = \{ \text{if, return, (, ), }, , ; , ==, \text{ID, INT_LIT, \ldots} \}\]
if (n == 0) { return x; }
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
How long would it take to parse this program?

<table>
<thead>
<tr>
<th>Try all parse trees</th>
<th>$\geq 10^{80}$ years</th>
</tr>
</thead>
<tbody>
<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. \( (\bullet)( ) \)
3. \( ( ) \bullet ( ) \)
4. \( A \bullet ( ) \)
5. \( S \bullet ( ) \)
6. \( S(\bullet) \)
7. \( S( ) \bullet \)
8. \( S \quad A \bullet \)

8/30
Features of LR(0) parser:

• Greedily reduce the recently completed rule into a variable
• Unique choice of reduction at any time
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 $(\bullet)(\ )$ $\Rightarrow$ 3 $(\ )\bullet(\ )$ $\Rightarrow$ 4 $A\bullet(\ )$ $\Rightarrow$ 5 $S\bullet(\ )$ $\Rightarrow$ ...

✓: NFA $N$ accepts
NFA acceptance condition

A rule $B \rightarrow \beta$ has just been completed if

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

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

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

Example: $7 \ S(\ ) \bullet$

This case can be chained
Designing NFA for Case 1

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

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

- $S \rightarrow SA | A$
- $A \rightarrow (S) | ()$

Diagram:
- Initial state $q_0$
- Transitions:
  - $S \rightarrow SA$
  - $S \rightarrow S \cdot A$
  - $A \rightarrow S \rightarrow SA$
  - $S \rightarrow \varepsilon$
  - $A \rightarrow \varepsilon$
  - $A \rightarrow (S)$
  - $A \rightarrow (S \cdot)$
  - $A \rightarrow (\cdot S)$
  - $A \rightarrow (\cdot)$
  - $A \rightarrow ()$
  - $A \rightarrow (S)\cdot$
  - $A \rightarrow (S \cdot)\cdot$
  - $A \rightarrow (\cdot)\cdot$
  - $A \rightarrow ()\cdot$
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.

- $S \rightarrow SA | A$
- $A \rightarrow (S) | ()$
Designing NFA for Cases 1 & 2

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

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:

\[
C \rightarrow \alpha \bullet B\gamma \quad \xrightarrow{\varepsilon} \quad B \rightarrow \bullet \beta
\]

All blue \(\rightarrow\) are \(\varepsilon\)-transitions.
Summary of the NFA

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

$$
\begin{array}{c}
q_0 \\
\epsilon
\end{array} \quad \rightarrow 
\begin{array}{c}
B \rightarrow \bullet \beta
\end{array}
$$

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

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

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

$$
\begin{array}{c}
C \rightarrow \alpha \bullet B \gamma \\
\epsilon
\end{array} \quad \rightarrow 
\begin{array}{c}
B \rightarrow \bullet \beta
\end{array}
$$

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

Dead state (empty set) not shown for clarity

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

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

**Shift** state:
- no completed rule

\[
S \rightarrow S \cdot A \\
A \rightarrow (S) \\
A \rightarrow ( )
\]

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

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

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

$\text{( ( ) )} \Rightarrow \text{ ( ( ) A )}$

After reducing $\text{( )}$ 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 $B$
Example

<table>
<thead>
<tr>
<th>State</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$q_1$</td>
</tr>
<tr>
<td>2</td>
<td>$q_5$</td>
</tr>
<tr>
<td>3</td>
<td>$q_8$</td>
</tr>
<tr>
<td>4</td>
<td>$q_4$</td>
</tr>
<tr>
<td>5</td>
<td>$q_2$</td>
</tr>
<tr>
<td>6</td>
<td>$q_5$</td>
</tr>
</tbody>
</table>

Diagram:

- State stack
  - State
  - Stack

- State stack
  - State
  - Stack

- State stack
  - State
  - Stack

- State stack
  - State
  - Stack

- State stack
  - State
  - Stack

- State stack
  - State
  - Stack

- State stack
  - State
  - Stack

- State stack
  - State
  - Stack
Example

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

The parser's output is the parse tree.

<table>
<thead>
<tr>
<th>State</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>$$$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$$$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$$$</td>
</tr>
</tbody>
</table>
Another LR(0) grammar

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

\[ C \rightarrow aCa \mid bCb \mid \# \]
Another LR(0) grammar

The grammar rules are:

\[ C \rightarrow \bullet a Ca \]
\[ C \rightarrow \bullet b Cb \]
\[ C \rightarrow \bullet # \]

The input string is:

\[ ba\#ab \]

The stack, state, and action table is:

<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>$143</td>
<td>5</td>
<td>S</td>
</tr>
<tr>
<td>$1435</td>
<td>7</td>
<td>R</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>

The diagram shows the transitions and actions for each input symbol.
Deterministic PDAs

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:**

- \( q_0 \) is the start state.
- Transitions include:
  - \( \varepsilon \) transitions leading to other states.
  - Transitions labeled with 'a' and 'b'.
  - finale state(s).
Example 2

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

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

Derivation

\[
S \rightarrow SA | A
\]

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

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; }
```

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

CFGs of most programming languages are not LR(0). LR(0) parser cannot tell apart
```
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 $N$

<table>
<thead>
<tr>
<th>LR(0):</th>
<th>LR(1):</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \rightarrow \alpha \cdot \beta$</td>
<td>$[A \rightarrow \alpha \cdot \beta, a]$</td>
</tr>
</tbody>
</table>

States in DFA $D$

<table>
<thead>
<tr>
<th></th>
<th>LR(0):</th>
<th>LR(1):</th>
</tr>
</thead>
<tbody>
<tr>
<td>shift-reduce</td>
<td>forbidden</td>
<td>some allowed</td>
</tr>
<tr>
<td>conflicts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reduce-reduce</td>
<td>forbidden</td>
<td>some allowed</td>
</tr>
<tr>
<td>conflicts</td>
<td></td>
<td>if resolvable with lookahead symbol $a$</td>
</tr>
</tbody>
</table>

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