LR(0) Parsers
CSCI 3130 Formal Languages and Automata Theory

Siu On CHAN

Chinese University of Hong Kong

Fall 2016
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

\[ \Sigma = \{ \text{if, return, (,), {, }, ;, ==, ID, INT_LIT, ...} \} \]
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 (years)</th>
</tr>
</thead>
<tbody>
<tr>
<td>try all parse trees</td>
<td>$\geq 10^{80}$</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

```
S → SA | A
A → (S) | ()
```

input: ()()
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 $B$

\[
\ldots \rightarrow 2 \ (\cdot)() \rightarrow 3 \ ()\cdot() \rightarrow 4 \ A\cdot() \rightarrow 5 \ S\cdot() \rightarrow \ldots
\]

✓: NFA $N$ accepts
NFA acceptance condition

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

A rule \( B \rightarrow \beta \) has just been completed if

Case 1  input/buffer so far is exactly \( \beta \)
Examples: \( 3\) \((\cdot)\)(\cdot) and \( 4\) \( A\)(\cdot)

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

This case can be chained
Designing NFA for Case 1

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

$S' \rightarrow SA \mid A$
$A \rightarrow (S') \mid ()$

All blue $\rightarrow$ are $\varepsilon$-transitions
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 $C \rightarrow \alpha \cdot B \gamma$

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

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

$$
\begin{array}{c}
\xrightarrow{\varepsilon} q_0 \\
\rightarrow B \rightarrow \bullet \beta
\end{array}
$$

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

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

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

$$
\overline{B \rightarrow \beta \bullet}
$$

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

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

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 contains only one rule: a completed rule $B \rightarrow \beta \bullet$, and such rules appear only in accepting states
LR(0) grammars

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 $B \rightarrow \beta \bullet$
  - and completed rules appear only in accepting states

### Shift state:
- no completed rule

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

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

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

\[
((\bullet)) \quad \Rightarrow \quad (A\bullet)
\]

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 $B$
### Example

<table>
<thead>
<tr>
<th>State</th>
<th>Stack</th>
<th>State</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>•(())()</td>
<td>q₁</td>
<td>$</td>
</tr>
<tr>
<td>2</td>
<td>(•)()</td>
<td>q₅</td>
<td>$1</td>
</tr>
<tr>
<td>3</td>
<td>()•()</td>
<td>q₈</td>
<td>$15</td>
</tr>
<tr>
<td></td>
<td>•A()</td>
<td>q₁</td>
<td>$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>A•()</td>
<td>q₄</td>
<td>$1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S•()</td>
<td>q₂</td>
<td>$1</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>•S()</td>
<td>q₁</td>
<td>$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>S(•)</td>
<td>q₅</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>
<tr>
<td></td>
<td>(</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>State</th>
<th>Stack</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>$S ()\bullet$</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>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>
</tbody>
</table>

Parser's output is the parse tree

\[ \bullet S \]
\[ \quad S \quad A \]
\[ \quad A \quad ( \quad ) \]
\[ ( \quad ) \]

\[ \bullet S \]
\[ \quad S \quad A \]
\[ \quad A \quad ( \quad ) \]
\[ ( \quad ) \]

\[ \bullet S \]
\[ \quad S \quad A \]
\[ \quad A \quad ( \quad ) \]
\[ ( \quad ) \]

\[ \bullet S \]
\[ \quad S \quad A \]
\[ \quad A \quad ( \quad ) \]
\[ ( \quad ) \]
Another LR(0) grammar

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

\[ C \rightarrow a C a \mid b C b \mid \# \]

NFA \( N \):

- \( q_0 \)
- \( C \rightarrow \bullet a C a \)
- \( C \rightarrow a \bullet C a \)
- \( C \rightarrow a C \bullet a \)
- \( C \rightarrow a C a \bullet \)
- \( C \rightarrow \bullet b C b \)
- \( C \rightarrow b \bullet C b \)
- \( C \rightarrow b C \bullet b \)
- \( C \rightarrow b C b \bullet \)
Another LR(0) grammar

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

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>$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>
Deterministic PDAs

PDA for LR(0) parsing is deterministic

Some CFLs require non-deterministic PDAs, such as

$$L = \{ w w^R \mid w \in \{a, b\}^* \}$$

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

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

\[ C \rightarrow a C a \mid b C b \mid \varepsilon \]

NFA \( N \):

\[ C \rightarrow \bullet a C a \]
\[ C \rightarrow a \bullet C a \]
\[ C \rightarrow a C \bullet a \]
\[ C \rightarrow a C a \bullet \]

\[ C \rightarrow \bullet b C b \]
\[ C \rightarrow b \bullet C b \]
\[ C \rightarrow b C \bullet b \]
\[ C \rightarrow b C b \bullet \]
Example 2

\[
C \rightarrow \bullet a C_a
\]
\[
C \rightarrow \bullet b C_b
\]
\[
C \rightarrow \bullet
\]

\[
C \rightarrow a \bullet C_a
\]
\[
C \rightarrow \bullet a C_a
\]
\[
C \rightarrow \bullet b C_b
\]
\[
C \rightarrow \bullet
\]

\[
C \rightarrow b \bullet C_b
\]
\[
C \rightarrow \bullet a C_a
\]
\[
C \rightarrow \bullet b C_b
\]
\[
C \rightarrow \bullet
\]

\[
C \rightarrow a C \bullet a
\]
\[
C \rightarrow a C \bullet a
\]
\[
C \rightarrow a C_a \bullet
\]
\[
C \rightarrow b C_b \bullet
\]

\[
C \rightarrow a C_a \mid b C_b \mid \varepsilon
\]

shift-reduce conflicts
Parser generator

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

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

**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; }

PARSING COMPUTER PROGRAMS

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) grammar

LR(1) grammars resolve such conflicts by one symbol lookahead

States in NFA $N$

LR(0):

$A \rightarrow \alpha \bullet \beta$

LR(1):

$[A \rightarrow \alpha \bullet \beta, a]$

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