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

Siu On CHAN

Chinese University of Hong Kong

Fall 2015
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

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

if
  ParExpression
  (Expression
    Expression
      Primary
      Identifier
        ID
      Infixop
        ==
      Expression
        Primary
        Literal
          INT_LIT
    )
  Statement
    Block
      BlockStatements
        }
      BlockStatement
        Statement
          return
            Expression
              Primary
          Identifier
            ID
      }

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 based on what has been read so far

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

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 $B$.

\[
\begin{array}{cccccc}
\cdots \Rightarrow & 2 & (\bullet)(\ ) & \Rightarrow & 3 & (\ )\bullet(\ ) & \Rightarrow & 4 & A\bullet(\ ) & \Rightarrow & 5 & S\bullet(\ ) & \Rightarrow & \cdots \\
\quad & (\ ) & & & & (\ ) & & (\ ) & & (\ ) & & (\ ) & & (\ ) & \\
\end{array}
\]

\[\checkmark: \text{ NFA } N \text{ accepts}\]
NFA acceptance condition

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

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

**Case 1** input/buffer so far is exactly \( \beta \)

Examples: \( 3 \) \( ()\cdot() \) 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

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

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 $B \rightarrow \beta$
Designing NFA for Cases 1 & 2

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

\[
\begin{align*}
S & \to SA \mid A \\
A & \to (S) \mid ()
\end{align*}
\]
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 $C \rightarrow \alpha \cdot B\gamma$, $B \rightarrow \beta$

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

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

$$
q_0 \xrightarrow{\varepsilon} B \rightarrow \bullet \beta
$$

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

$$
B \rightarrow \alpha \bullet X \beta \xrightarrow{X} B \rightarrow \alpha X \bullet \beta
$$

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

$$
B \rightarrow \beta \bullet
$$

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

$$
C \rightarrow \alpha \bullet B \gamma \xrightarrow{\varepsilon} B \rightarrow \bullet \beta
$$

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
  
  \[
  A \rightarrow (S) \bullet
  \]
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 stack</th>
<th>state</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>()()()</td>
<td>q₁</td>
<td>$</td>
</tr>
<tr>
<td>(())()</td>
<td>q₅</td>
<td>$1</td>
</tr>
<tr>
<td>()()()</td>
<td>q₈</td>
<td>$15</td>
</tr>
<tr>
<td>A()</td>
<td>q₁</td>
<td>$</td>
</tr>
<tr>
<td>A()</td>
<td>q₄</td>
<td>$1</td>
</tr>
<tr>
<td>S()</td>
<td>q₁</td>
<td>$</td>
</tr>
<tr>
<td>S()</td>
<td>q₅</td>
<td>$12</td>
</tr>
</tbody>
</table>

5  $S$  ()  q₂  $1

6  $S$  ()  q₅  $12
Example

<table>
<thead>
<tr>
<th>state</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>$125</td>
</tr>
<tr>
<td>8</td>
<td>$12</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 \( N \):
Another LR(0) grammar

\[ C \rightarrow aC_a | bC_b | \# \]

**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

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

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

NFA \( N \):
Example 2

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
Parsing computer programs

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

```
if ParExpression else Statement
| Statement
| Expression
| |
```
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 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