G51MAL: Lecture 17

LR(0) Parsing and Parser Generators

Henrik Nilsson

University of Nottingham, UK
This Lecture

- Briefly explaining the basics of LR(0) parsing to show practical application of Deterministic PDA.
- Quick outline of the Happy parser generator.
LR(0) Parsing (1)

A DFA recognising **viable prefixes** for the CFG

\[
S ::= aABe \\
A ::= bcA \mid c \\
B ::= d
\]
LR(0) Parsing (2)

How to construct such a DFA is beyond the scope of this course. See e.g. Aho, Sethi, Ullman (1986) for details. However, some observations:

Consider a right-sentential form, e.g. abcAde. Note that prefixes a, ab, abc, abcA are recognised by the DFA (all states are considered final).

Note that strings like acb, which are not a prefix of any right-sentential form, are not accepted.
LR(0) Parsing (2)

How to construct such a DFA is beyond the scope of this course. See e.g. Aho, Sethi, Ullman (1986) for details. However, some observations:

- Consider a right-sentential form, e.g. $abcAde$. Note that prefixes $\epsilon, a, ab, abc, abcA$ are recognised by the DFA (all states are considered final).
How to construct such a DFA is beyond the scope of this course. See e.g. Aho, Sethi, Ullman (1986) for details. However, some observations:

- Consider a right-sentential form, e.g. $abcAde$. Note that prefixes $\epsilon$, $a$, $ab$, $abc$, $abcA$ are recognised by the DFA (all states are considered final).

- Note that strings like $acb$, which are not a prefix of any right-sentential form, are not accepted.
Given a DFA recognising viable prefixes, a LR(0) parser can be constructed as follows:
LR(0) Parsing (3)

Given a DFA recognising viable prefixes, a LR(0) parser can be constructed as follows:

- In a state \textit{without complete items}: \textit{Shift}
Given a DFA recognising viable prefixes, a LR(0) parser can be constructed as follows:

- In a state *without complete items*: *Shift*
  - Read next terminal symbol and push it onto internal *parse stack*. 
Given a DFA recognising viable prefixes, a LR(0) parser can be constructed as follows:

- In a state *without complete items*: **Shift**
  - Read next terminal symbol and push it onto internal *parse stack*.
  - Move to new state by following edge labelled by the read terminal.
LR(0) Parsing (4)

- In a state with a *single complete item*: Reduce
In a state with a **single complete item**: Reduce

- The top of the parse stack contains the **handle** of the current right-sentential form (since we have recognised a viable prefix for which a single **complete** item is valid).
In a state with a single complete item: Reduce

- The top of the parse stack contains the handle of the current right-sentential form (since we have recognised a viable prefix for which a single complete item is valid).
- The handle is just the RHS of the valid item.
LR(0) Parsing (4)

- In a state with a **single complete item**: *Reduce*
  - The top of the parse stack contains the *handle* of the current right-sentential form (since we have recognised a viable prefix for which a single *complete* item is valid).
  - The handle is just the RHS of the valid item.
  - Reduce to the previous right-sentential form by replacing the handle on the parse stack with the LHS of the valid item.
In a state with a *single complete item*: *Reduce*

- The top of the parse stack contains the *handle* of the current right-sentential form (since we have recognised a viable prefix for which a single *complete* item is valid).
- The handle is just the RHS of the valid item.
- Reduce to the previous right-sentential form by replacing the handle on the parse stack with the LHS of the valid item.
- Move to the state indicated by the new viable prefix on the parse stack.
LR(0) Parsing (5)

- If a state contains both complete and incomplete items, or if a state contains more than one complete item, then the grammar was not LR(0).
LR(0) Parsing (6)

Note: $\gamma w$ is the current right-sentential form.

<table>
<thead>
<tr>
<th>State</th>
<th>Stack ($\gamma$)</th>
<th>Input ($w$)</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>$\epsilon$</td>
<td>abccde</td>
<td></td>
</tr>
</tbody>
</table>
Note: $\gamma w$ is the current right-sentential form.

<table>
<thead>
<tr>
<th>State</th>
<th>Stack ($\gamma$)</th>
<th>Input ($w$)</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>$\epsilon$</td>
<td>$abc cde$</td>
<td>Shift</td>
</tr>
</tbody>
</table>
LR(0) Parsing (6)

Note: \( \gamma w \) is the current right-sentential form.

<table>
<thead>
<tr>
<th>State</th>
<th>Stack (( \gamma ))</th>
<th>Input (( w ))</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>I0</td>
<td>( \epsilon )</td>
<td>( abccde )</td>
<td>Shift</td>
</tr>
<tr>
<td>I1</td>
<td>( a )</td>
<td>( bccde )</td>
<td></td>
</tr>
</tbody>
</table>
Note: $\gamma w$ is the current right-sentential form.

<table>
<thead>
<tr>
<th>State</th>
<th>Stack ($\gamma$)</th>
<th>Input ($w$)</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>I0</td>
<td>$\epsilon$</td>
<td>$abccde$</td>
<td>Shift</td>
</tr>
<tr>
<td>I1</td>
<td>$a$</td>
<td>$bccde$</td>
<td>Shift</td>
</tr>
</tbody>
</table>
LR(0) Parsing (7)
LR(0) Parsing (7)

State | Stack (γ) | Input (w) | Move
--- | --- | --- | ---
12 | ab | ccde | Shift
LR(0) Parsing (7)

<table>
<thead>
<tr>
<th>State</th>
<th>Stack (γ)</th>
<th>Input (w)</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td>ab</td>
<td>ccde</td>
<td>Shift</td>
</tr>
<tr>
<td>I3</td>
<td>abc</td>
<td>cde</td>
<td></td>
</tr>
</tbody>
</table>
LR(0) Parsing (7)

State | Stack ($\gamma$) | Input ($w$) | Move
---|---|---|---
I2 | $ab$ | $ccde$ | Shift
I3 | $abc$ | $cde$ | Shift
LR(0) Parsing (7)

<table>
<thead>
<tr>
<th>State</th>
<th>Stack ($\gamma$)</th>
<th>Input ($w$)</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>$ab$</td>
<td>$ccde$</td>
<td>Shift</td>
</tr>
<tr>
<td>13</td>
<td>$abc$</td>
<td>$cde$</td>
<td>Shift</td>
</tr>
<tr>
<td>15</td>
<td>$abcc$</td>
<td>$de$</td>
<td></td>
</tr>
</tbody>
</table>
LR(0) Parsing (7)

<table>
<thead>
<tr>
<th>State</th>
<th>Stack (γ)</th>
<th>Input (w)</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td>ab</td>
<td>ccde</td>
<td>Shift</td>
</tr>
<tr>
<td>I3</td>
<td>abc</td>
<td>cde</td>
<td>Shift</td>
</tr>
<tr>
<td>I5</td>
<td>abcc</td>
<td>de</td>
<td>Reduce by $A ::= c$</td>
</tr>
</tbody>
</table>
LR(0) Parsing (8)

State | Stack ($\gamma$) | Input ($w$) | Move
--- | --- | --- | ---
14 | $abcA$ | $de$ |
LR(0) Parsing (8)

<table>
<thead>
<tr>
<th>State</th>
<th>Stack (γ)</th>
<th>Input (w)</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>I4</td>
<td>abcA</td>
<td>de</td>
<td>Reduce by A ::= bcA</td>
</tr>
</tbody>
</table>
### LR(0) Parsing (8)

<table>
<thead>
<tr>
<th>State</th>
<th>Stack ((\gamma))</th>
<th>Input ((w))</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>I4</td>
<td>(abcA)</td>
<td>(de)</td>
<td>Reduce by (A ::= bcA)</td>
</tr>
<tr>
<td>I6</td>
<td>(aA)</td>
<td>(de)</td>
<td></td>
</tr>
</tbody>
</table>

**Diagram:**

1. **State 14:**
   - Stack: \(S ::= a\cdot ABe\)
   - Stack: \(A ::= b\cdot cA\)
   - Stack: \(A ::= c\cdot c\)
   - Move: Shift
   - Input: \(a\)

2. **State 15:**
   - Move: Reduce by \(A ::= bcA\)
   - Input: \(c\)

3. **State 16:**
   - Stack: \(S ::= aA\cdot Be\)
   - Stack: \(B ::= d\cdot d\)
   - Move: Reduce by \(B ::= d\cdot d\)
   - Input: \(d\)

4. **State 18:**
   - Move: Reduce by \(S ::= aAB\cdot e\)
   - Input: \(e\)
LR(0) Parsing (8)

State | Stack ($\gamma$) | Input ($w$) | Move |
--- | --- | --- | --- |
I4 | $abcA$ | $de$ | Reduce by $A ::= bcA$ |
I6 | $aA$ | $de$ | Shift |
LR(0) Parsing (8)

<table>
<thead>
<tr>
<th>State</th>
<th>Stack (γ)</th>
<th>Input (w)</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>I4</td>
<td>abcA</td>
<td>de</td>
<td>Reduce by A ::= bcA</td>
</tr>
<tr>
<td>I6</td>
<td>aA</td>
<td>de</td>
<td>Shift</td>
</tr>
<tr>
<td>I7</td>
<td>aAd</td>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>
LR(0) Parsing (8)

State | Stack (γ) | Input (w) | Move
---|---|---|---
14 | abcA | de | Reduce by $A ::= bcA$
16 | aA | de | Shift
17 | aAd | e | Reduce by $B ::= d$
LR(0) Parsing (9)

State | Stack (\( \gamma \)) | Input (\( w \)) | Move
--- | --- | --- | ---
18 | \( aAB \) | \( e \) |
LR(0) Parsing (9)

State | Stack ($\gamma$) | Input ($w$) | Move
---|---|---|---
18 | $aAB$ | $e$ | Shift
LR(0) Parsing (9)

State | Stack ($\gamma$) | Input ($w$) | Move
---|---|---|---
18 | $aAB$ | $e$ | Shift
19 | $aABe$ | $\epsilon$ | Shift
LR(0) Parsing (9)

```
<table>
<thead>
<tr>
<th>State</th>
<th>Stack (γ)</th>
<th>Input (w)</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>aAB</td>
<td>e</td>
<td>Shift</td>
</tr>
<tr>
<td>19</td>
<td>aABe</td>
<td>ε</td>
<td>Reduce by S ::= aABe</td>
</tr>
</tbody>
</table>
```
LR(0) Parsing (9)

State | Stack (γ) | Input (w) | Move
----- | ---------- |---------- |-----
18    | aAB      | e        | Shift
19    | aABe     | ε        | Reduce by $S ::= aABe$
        | $S$       | ε        | Done
### LR(0) Parsing (10)

Complete sequence ($\gamma w$ is right-sentential form):

<table>
<thead>
<tr>
<th>State</th>
<th>Stack ($\gamma$)</th>
<th>Input ($w$)</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>I0</td>
<td>$\epsilon$</td>
<td>abccde</td>
<td>Shift</td>
</tr>
<tr>
<td>I1</td>
<td>$a$</td>
<td>bccde</td>
<td>Shift</td>
</tr>
<tr>
<td>I2</td>
<td>$ab$</td>
<td>ccde</td>
<td>Shift</td>
</tr>
<tr>
<td>I3</td>
<td>$abc$</td>
<td>cde</td>
<td>Shift</td>
</tr>
<tr>
<td>I4</td>
<td>$abcc$</td>
<td>$de$</td>
<td>Reduce by $A ::= c$</td>
</tr>
<tr>
<td>I5</td>
<td>$abcA$</td>
<td>$de$</td>
<td>Reduce by $A ::= bcA$</td>
</tr>
<tr>
<td>I6</td>
<td>$aA$</td>
<td>$de$</td>
<td>Shift</td>
</tr>
<tr>
<td>I7</td>
<td>$aAd$</td>
<td>$e$</td>
<td>Reduce by $B ::= d$</td>
</tr>
<tr>
<td>I8</td>
<td>$aAB$</td>
<td>$e$</td>
<td>Shift</td>
</tr>
<tr>
<td>I9</td>
<td>$aABe$</td>
<td>$\epsilon$</td>
<td>Reduce by $S ::= aABe$</td>
</tr>
<tr>
<td>$S$</td>
<td>$\epsilon$</td>
<td></td>
<td>Done</td>
</tr>
</tbody>
</table>

Cf: $S \Rightarrow aABe \Rightarrow aAde \Rightarrow abcAde \Rightarrow abccde$
Parser Generators (1)

- Constructing parsers by hand can be very tedious and time consuming.

---

Example grammar:

- $S \rightarrow aABe$
- $A \rightarrow bcA$
- $B \rightarrow d$

This gives rise to a 10-state LR(0) DFA!
Constructing parsers by hand can be very tedious and time consuming.

This is true in particular for LR($k$) and LALR parsers: constructing the corresponding DFAs is extremely laborious.
Constructing parsers by hand can be very tedious and time consuming.

This is true in particular for LR($k$) and LALR parsers: constructing the corresponding DFAs is extremely laborious.

E.g., our simple grammar

\[
\begin{align*}
S & ::= aABe \\
A & ::= bcA \mid c \\
B & ::= d
\end{align*}
\]

gives rise to a 10 state LR(0) DFA!
An LR(0) DFA recognizing viable prefixes for

\[ S ::= aABe \quad A ::= bcA \mid c \quad B ::= d \]
Parser Generators (3)

- Parser construction is a mechanical process. Why not write a program to do the hard work for us?
Parser Generators (3)

- Parser construction is a mechanical process. Why not write a program to do the hard work for us?
- A **Parser Generator** (or “compiler compiler”) takes a grammar as input and outputs a parser (a program) for that grammar.
Parser Generators (3)

- Parser construction is a mechanical process. Why not write a program to do the hard work for us?
- A **Parser Generator** (or “compiler compiler”) takes a grammar as input and outputs a parser (a program) for that grammar.
- The input grammar is augmented with **“semantic actions”**: code fragments that get invoked when a derivation step is performed.
Parser Generators (3)

- Parser construction is a mechanical process. Why not write a program to do the hard work for us?

- A **Parser Generator** (or “compiler compiler”) takes a grammar as input and outputs a parser (a program) for that grammar.

- The input grammar is augmented with **“semantic actions”**: code fragments that get invoked when a derivation step is performed.

- The semantic actions typically construct an AST or interpret the program being parsed.
Some examples of parser generators:

- Yacc (“Yet Another Compiler Compiler”): A classic UNIX LALR parser generator for C. [http://dinosaur.compilertools.net/]
- Bison: GNU project parser generator, a free Yacc replacement, for C and C++.
- Happy: a parser generator for Haskell, similar to Yacc and Bison. [http://www.haskell.org/happy/]
- Cup: LALR parser generator for Java.
Some examples of parser generators:

- **Yacc ("Yet Another Compiler Compiler"):** A classic UNIX LALR parser generator for C.  
  [http://dinosaur.compilertools.net/](http://dinosaur.compilertools.net/)
Some examples of parser generators:


- Bison: GNU project parser generator, a free Yacc replacement, for C and C++. 


Some examples of parser generators:

- **Yacc (“Yet Another Compiler Compiler”)**: A classic UNIX LALR parser generator for C. 
  [http://dinosaur.compilertools.net/](http://dinosaur.compilertools.net/)

- **Bison**: GNU project parser generator, a free Yacc replacement, for C and C++.

- **Happy**: a parser generator for Haskell, similar to Yacc and Bison. 
Parser Generators (4)

Some examples of parser generators:

- Bison: GNU project parser generator, a free Yacc replacement, for C and C++.  
- Cup: LALR parser generator for Java.
Parser Generators (5)

Parser Generators (5)

- Many more compiler tools for Java here: [http://catalog.compilertools.net/java.html](http://catalog.compilertools.net/java.html)
Parser Generators (5)

- Many more compiler tools for Java here: [http://catalog.compilertools.net/java.html](http://catalog.compilertools.net/java.html)
- And a general catalogue of compiler tools: [http://catalog.compilertools.net/](http://catalog.compilertools.net/)
We are going to develop a TXL (the Trivial eXpression Language) using Happy. The TXL CFG:

\[
\begin{align*}
\text{txl-program} & ::=} \text{exp} \\
\text{exp} & ::=} \text{add-exp} \\
\text{add-exp} & ::=} \text{mul-exp} \\
& \quad | \quad \text{add-exp} + \text{mul-exp} \\
& \quad | \quad \text{add-exp} - \text{mul-exp}
\end{align*}
\]
The TXL CFG continued:

```
mul-exp ::= prim-exp
   | mul-exp * prim-exp
   | mul-exp / prim-exp

prim-exp ::= INTEGER
   | IDENTIFIER
   | ( exp )
   | let IDENTIFIER = exp in exp
```
Haskell datatype for tokens:

```haskell
data Token = T_Int Int
             | T_Id Id
             | T_Plus
             | T_Minus
             | T_Times
             | T_Divide
             | T_LeftPar
             | T_RightPar
             | T_Equal
             | T_Let
             | T_In
```
Haskell datatypes for AST:

```
data BinOp = Plus | Minus | Times | Divide

data Exp = LitInt Int
         | Var Id
         | BinOpApp BinOp Exp Exp
         | Let Id Exp Exp
```
A simple Happy input file looks like follows:

```haskell
{ Module Header }
%name ParserFunctionName
%tokentype { TokenTypeName }

%token

Specification of Terminal Symbols
%

Grammar productions with semantic actions

{ Further Haskell Code }
```
The terminal symbol specification specifies terminals to be used in productions and relates them to Haskell constructors for the tokens:

```
%token
int       { T_Int $$ }
ident     { T_Id $$ }
'+'       { T_Plus }
'-'       { T_Minus }
...
'='       { T_Equal }
let       { T_Let }
in        { T_In }
```
Happy Parser for TXL (5)

The grammar productions are written in BNF, with an additional semantic action defining the return value for each production:

```
add_exp
  :  mul_exp  {$1}
  |  add_exp '+' mul_exp  {BinOpApp Plus $1 $3}
  |  add_exp '-' mul_exp  {BinOpApp Minus $1 $3}

mul_exp
  :  prim_exp  {$1}
  |  mul_exp '*' prim_exp  {BinOpApp Times $1 $3}
  |  mul_exp '/' prim_exp  {BinOpApp Divide $1 $3}
```
Precedence and Associativity

Happy (like e.g. Yacc and Bison) allows operator precedence and associativity to be explicitly specified to disambiguate a grammar:

```plaintext
%left '+' '-'
%left '*' '/'

exp : exp '+' exp { BinOpApp Plus $1 $3 }
    | exp '-' exp { BinOpApp Minus $1 $3 }
    | exp '*' exp { BinOpApp Times $1 $3 }
    | exp '/' exp { BinOpApp Divide $1 $3 }
    ...
```