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

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

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).
- Note that strings like \(acb\), which are not a prefix of any right-sentential form, are not accepted.
LR(0) Parsing (3)

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**
  - 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>I0</td>
<td>$\epsilon$</td>
<td>abcde</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)

<table>
<thead>
<tr>
<th>State</th>
<th>Stack (γ)</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>Reduce by $A ::= c$</td>
</tr>
</tbody>
</table>

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>14</td>
<td>abcA</td>
<td>de</td>
<td>Reduce by $A ::= bcA$</td>
</tr>
<tr>
<td>16</td>
<td>aA</td>
<td>de</td>
<td>Shift</td>
</tr>
<tr>
<td>17</td>
<td>aAd</td>
<td>e</td>
<td>Reduce by $B ::= d$</td>
</tr>
</tbody>
</table>

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>
<tr>
<td></td>
<td>S</td>
<td>ε</td>
<td>Done</td>
</tr>
</tbody>
</table>

LR(0) Parsing (10)

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

<table>
<thead>
<tr>
<th>State</th>
<th>Stack (γ)</th>
<th>Input (w)</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>c</td>
<td>abcde</td>
<td>Shift</td>
</tr>
<tr>
<td>11</td>
<td>a</td>
<td>bcde</td>
<td>Shift</td>
</tr>
<tr>
<td>12</td>
<td>ab</td>
<td>cde</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>Reduce by $A ::= c$</td>
</tr>
<tr>
<td>14</td>
<td>abcA</td>
<td>de</td>
<td>Reduce by $A ::= bcA$</td>
</tr>
<tr>
<td>16</td>
<td>aA</td>
<td>de</td>
<td>Shift</td>
</tr>
<tr>
<td>17</td>
<td>aAd</td>
<td>e</td>
<td>Reduce by $B ::= d$</td>
</tr>
<tr>
<td>18</td>
<td>aAB</td>
<td>ε</td>
<td>Shift</td>
</tr>
<tr>
<td>19</td>
<td>aABe</td>
<td>ε</td>
<td>Reduce by $S ::= aABe$</td>
</tr>
<tr>
<td></td>
<td>S</td>
<td>ε</td>
<td>Done</td>
</tr>
</tbody>
</table>

Cf: $S \Rightarrow aABe \Rightarrow aAde \Rightarrow abcAde \Rightarrow abcde$
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

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

gives rise to a 10 state LR(0) DFA!

Parser Generators (2)

An LR(0) DFA recognizing viable prefixes for

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

gives rise to the following DFA:

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.

Parser Generators (4)

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.

- Cup: LALR parser generator for Java.
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/)

Happy Parser for TXL (1)

We are going to develop a TXL (the Trivial eXpression Language) using Happy. The TXL CFG:

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

Happy Parser for TXL (2)

The TXL CFG continued:

\[
\begin{align*}
\text{mul-exp} & ::= \text{prim-exp} \\
& \quad | \text{mul-exp} \ast \text{prim-exp} \\
& \quad | \text{mul-exp} / \text{prim-exp} \\
\text{prim-exp} & ::= \text{INTEGER} \\
& \quad | \text{IDENTIFIER} \\
& \quad | (\text{exp}) \\
& \quad | \text{let IDENTIFIER} = \text{exp} \text{ in } \text{exp}
\end{align*}
\]

Happy Parser for TXL (3)

Haskell datatype for tokens:

\[
data \text{Token} = \begin{cases} 
\text{T_Int} \cdot \text{Int} & | \\
\text{T_Id} \cdot \text{Id} & | \\
\text{T_Plus} & | \\
\text{T_Minus} & | \\
\text{T_Times} & | \\
\text{T_Divide} & | \\
\text{T_LeftPar} & | \\
\text{T_RightPar} & | \\
\text{T_Equal} & | \\
\text{T_Let} & | \\
\text{T_In}
\end{cases}
\]
Haskell datatypes for AST:

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

data Exp = LitInt Int
        | Var Id
        | BinOpApp BinOp Exp Exp
        | Let Id Exp Exp
```

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

```haskell
%token
  int    { T_Int $1 }
  ident  { T_Id $1 }
  '+'    { T_Plus }
  '-'    { T_Minus }
  '='    { T_Equal }
  let    { T_Let }
  in     { T_In }
```

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

```haskell
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 }
    ...
```