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

\[
S ::= aAbe \quad A ::= bcA \mid c \quad B ::= d
\]

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 \(\epsilon, a, ab, abc, abcA\) 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>10</td>
<td>(\epsilon)</td>
<td>abbcde</td>
<td>Shift</td>
</tr>
<tr>
<td>11</td>
<td>(\alpha)</td>
<td>bbcd</td>
<td>Shift</td>
</tr>
</tbody>
</table>

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>code</td>
<td>Shift</td>
</tr>
<tr>
<td>13</td>
<td>(abc)</td>
<td>code</td>
<td>Shift</td>
</tr>
<tr>
<td>15</td>
<td>(abc)</td>
<td>de</td>
<td>Reduce by (A ::= c)</td>
</tr>
</tbody>
</table>
Parser Generators (1)

- 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

\[
S ::= aABe \\
A ::= bcA | c \\
B ::= d
\]

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

Parser Generators (2)

An LR(0) DFA recognizing viable prefixes for

\[
S ::= aABe \\
A ::= bcA \mid c \\
B ::= d
\]

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.

Happy Parser for TXL (1)

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

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

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:

%name ParserFunctionName
%tokentype TokenTypeName

{ Module Header }
{ Further Haskell Code }

Precedence and Associativity

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

%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 }
    ...