This Lecture

- Parser generators ("compiler compilers")
- The parser generator Happy
- A TXL parser written using Happy
- A TXL interpreter written using Happy

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., this simple grammar (from the prev. lect.)
  \[
  S \rightarrow aABe \\
  A \rightarrow bcA | c \\
  B \rightarrow d
  \]
  gives rise to a 10 state LR(0) DFA!

Parser Generators (2)

An LR(0) DFA recognizing viable prefixes for

\[
S \rightarrow aABe \\
A \rightarrow bcA | c \\
B \rightarrow d
\]

Parser Generators (3)

- **Parser construction** is in many ways a very 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)

Consider an LR shift-reduce parser:

- Some of the actions when parsing \texttt{abcde}:

<table>
<thead>
<tr>
<th>State</th>
<th>Stack ((\gamma))</th>
<th>Input ((w))</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I_1)</td>
<td>(aA)</td>
<td>(de)</td>
<td>Shift</td>
</tr>
<tr>
<td>(I_2)</td>
<td>(aAB)</td>
<td>(\epsilon)</td>
<td>Reduce by (B \rightarrow d)</td>
</tr>
<tr>
<td>(I_3)</td>
<td>(a)</td>
<td>(\epsilon)</td>
<td>Shift</td>
</tr>
<tr>
<td>(I_4)</td>
<td>(aABe)</td>
<td>(\epsilon)</td>
<td>Reduce by (S \rightarrow aABe)</td>
</tr>
<tr>
<td>(I_5)</td>
<td>(S)</td>
<td>(\epsilon)</td>
<td>Done</td>
</tr>
</tbody>
</table>

- A reduction corresponds to a derivation step in the grammar (an LR parser performs a rightmost derivation in reverse).

Parser Generators (5)

- At a reduction, the terminals and non-terminals of the RHS of the production (the **handle**) are on the parse stack, associated with semantic information or semantic value; e.g., the corresponding AST fragments, expression values.
- Think of the RHS symbols as variables bound to the the semantic value resulting from a successful derivation for that symbol.
- Construction of AST, evaluation of expressions, etc. proceeds in **bottom-up** order.

Parser Generators (6)

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 (7)

- 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 parser using Happy. The TXL CFG:

```
TXLProgram → Exp
Exp      → AddExp
AddExp   → MulExp
         | AddExp + MulExp
         | AddExp - MulExp
```

Note: Left-recursive! (To impart associativity.)

LR parsers have no problems with left- or right-recursion (except right recursion uses more stack).

The TXL CFG continued:

```
MulExp → PrimExp
       | MulExp * PrimExp
       | MulExp / PrimExp
```

```
PrimExp → IntegerLiteral
         | 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 parser specification:
```
{ Module Header }
%name ParserFunctionName
%tokentype { TokenTypeName }

%token { int $1 }
%token { ident $1 } 
%token { '+' $1 } 
%token { '-' $1 } 
%token { '=' $1 } 
%token { 'let' $1 } 
%token { 'in' $1 }

%token { T_Plus } 
%token { T_Minus } 
%token { T_Times } 
%token { T_Divide } 
%token { T_Id } 
%token { T_In }

Grammar productions with semantic actions

{ Further Haskell Code }
```

The grammar productions are written in BNF, with an additional semantic action defining the semantic 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 }
```

It is also possible to add type annotations:
```
add_exp :: { Exp }
add_exp : mul_exp { $1 }
        | add_exp ' + ' mul_exp { BinOpApp Plus $1 $3 }
        | add_exp ' - ' mul_exp { BinOpApp Minus $1 $3 }
```

Most useful when semantic values are of different types.

See HappyTXL.y for the complete example.
Shift/Red. and Red./Red. Conflicts (1)

Context-free grammars are often initially ambiguous. Consider the grammar fragment:
\[ \text{Cmd} \rightarrow \ldots | \text{if Exp then Cmd} | \text{if Exp then Cmd else Cmd} \]

According to this grammar, a program fragment
\[ \text{if } e_1 \text{ then if } e_2 \text{ then } c_1 \text{ else } c_2 \]
can be parsed in two ways, with very different meanings (the "dangling else" problem):
\[ \text{if } e_1 \text{ then } (\text{if } e_2 \text{ then } c_1) \text{ else } c_2 \]
\[ \text{if } e_1 \text{ then } (\text{if } e_2 \text{ then } c_1 \text{ else } c_2) \]

Shift/Red. and Red./Red. Conflicts (2)

In LR-parsing, ambiguous grammars lead to shift/reduce and reduce/reduce conflicts:
- shift/reduce: some states have mixed complete and incomplete items:
  \[ A \rightarrow a \cdot B \rightarrow a \cdot b \]
  Should parser shift or reduce?

Precedence and Associativity

Happy (like e.g. Yacc and Bison) allows operator precedence and associativity to be explicitly specified to disambiguate a grammar:
\[
\%left '+' '-'
\%
expr : expr '+' expr { BinOpApp Plus $1 $3 }
| expr '-' expr { BinOpApp Minus $1 $3 }
| expr '*' expr { BinOpApp Times $1 $3 }
| expr '/' expr { BinOpApp Divide $1 $3 }
\]

A TXL Interpreter (1)

The semantic actions do not have to construct an AST. An alternative is to interpret the code being parsed. Basic idea:
\[
\text{exp} :: \{ \text{Int} \} \\
\text{exp} : \text{exp} ' + ' \text{exp} \{ \text{BinOpApp Plus }$1 \text{ }$3 \} \\
| \text{exp} ' - ' \text{exp} \{ \text{BinOpApp Minus }$1 \text{ }$3 \} \\
| \text{exp} ' * ' \text{exp} \{ \text{BinOpApp Times }$1 \text{ }$3 \} \\
| \text{exp} ' / ' \text{exp} \{ \text{BinOpApp Divide }$1 \text{ }$3 \} \\
\]

A TXL Interpreter (2)

One way:
- Each semantic action returns a function of type \( \text{Env} \rightarrow \text{Int} \)
  where (for example)
  \( \text{Type Env} \rightarrow \text{Id} \rightarrow \text{Int} \)
- The semantic action for evaluating a composite expression passes on the environment. E.g. semantic action for +:
  \[ \text{exp ' + ' exp} \{ \text{BinOpApp Plus }$1 \text{ }$3 \} \]

A TXL Interpreter (3)

- The semantic action for a variable looks up the variable value in the environment:
  \[ \text{id} \{ \text{\env} \rightarrow \text{env }$1 \} \]
- The semantic action for let extends the argument environment and evaluates the body in the extended environment:
  \[ \text{let ident ' = ' exp in exp} \{ \text{\env} \rightarrow \text{let v }= \text{s4 env in \$6 } \{ \text{\i }\rightarrow \text{if }\i \text{ }= \text{s2 then v else env }\i \}} \]

A TXL Interpreter (4)

- A program gets evaluated by applying the overall result function to the empty environment:
  \( \_ \rightarrow \text{error "undefined variable"} \)

See HappyTXLInterpreter.y for further details.