G52CMP: Lecture 8

Syntactic Analysis: Parser Generators

Henrik Nilsson

University of Nottingham, UK

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.)

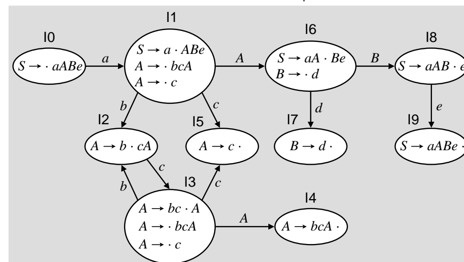
$$S \rightarrow aABe$$
$$A \rightarrow bcA \mid c$$
$$B \rightarrow d$$

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

Parser Generators (2)

An LR(0) DFA recognizing viable prefixes for

$S \rightarrow aABe \qquad A \rightarrow bcA \mid c \qquad B \rightarrow d$



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 $abccde$:

State	Stack (γ)	Input (w)	Move
...			
I6	aA	de	Shift
I7	aAd	e	Reduce by $B \rightarrow d$
I8	aAB	e	Shift
I9	$aABe$	ϵ	Reduce by $S \rightarrow aABe$
	S	ϵ	Done

- 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 are on the parse stack, associated with ***semantical information***, e.g. the corresponding AST. (Think of the RHS symbols as ***variables*** whose values are the corresponding semantical information.)
- If the goal is to construct an AST, the parser has access to the sub-ASTs and can construct an AST for the present derivation step.
- The AST gets constructed bottom-up.

Parser Generators (6)

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.

Parser Generators (7)

- ANTLR: LL(k) (recursive descent) parser generator and other translator-oriented tools for Java, C#, C++. `http://www.antlr.org/`
- Many more compiler tools for Java here: `http://catalog.compilertools.net/java.html`
- And a general catalogue of compiler tools: `http://catalog.compilertools.net/`

Happy Parser for TXL (1)

We are going to develop a TXL parser using Happy. The TXL CFG:

$$
\begin{array}{lcl}
\mathit{TXLProgram} & \rightarrow & \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{AddExp} \\
\mathit{AddExp} & \rightarrow & \mathit{MulExp} \\
 & | & \mathit{AddExp}\ \texttt{+}\ \mathit{MulExp} \\
 & | & \mathit{AddExp}\ \texttt{-}\ \mathit{MulExp}
\end{array}
$$

Note: ***Left-recursive!*** (To impart associativity.) LR parsers have no problems with left- or right-recursion.

Happy Parser for TXL (2)

The TXL CFG continued:

$$
\begin{array}{lcl}
\mathit{MulExp} & \rightarrow & \mathit{PrimExp} \\
 & | & \mathit{MulExp}\ \texttt{*}\ \mathit{PrimExp} \\
 & | & \mathit{MulExp}\ \texttt{/}\ \mathit{PrimExp} \\
\mathit{PrimExp} & \rightarrow & \underline{\mathit{IntegerLiteral}} \\
 & | & \underline{\mathit{Identifier}} \\
 & | & \texttt{(}\ \mathit{Exp}\ \texttt{)} \\
 & | & \texttt{let}\ \underline{\mathit{Identifier}}\ \texttt{=}\ \mathit{Exp}\ \texttt{in}\ \mathit{Exp}
\end{array}
$$

Happy Parser for TXL (3)

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
```


Happy Parser for TXL (4)

Haskell datatypes for AST:

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

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


Happy Parser for TXL (5)

A simple Happy parser specification:

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

%token
Specification of Terminal Symbols
%%
Grammar productions with semantic actions

{ Further Haskell Code }
```


Happy Parser for TXL (6)

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}
```


Happy Parser for TXL (6)

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.

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 }
    ...
```

See `HappyTXL2.y` for further details.

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:

```
exp :: { Int }
exp
  : exp '+' exp {$1 + $3}
  | exp '-' exp {$1 - $3}
  ...
```

But TXL has a `let`-construct . . .

What about VARIABLES???

A TXL Interpreter (2)

One way:

- Each semantic action returns a ***function*** of type
  ```
  Env -> Int
  ```
  where (for example)
  ```
  Type Env = Id -> Int
  ```
- The semantic action for evaluating a composite expression passes on the environment. E.g. semantic action for `+`:
  ```
  | exp '+' exp
  { \env -> $1 env + $3 env }
  ```


A TXL Interpreter (3)

- The semantic action for a variable looks up the variable value in the environment:
  ```
  | ident { \env -> env $1 }
  ```
- The semantic action for `let` extends the argument environment and evaluates the body in the extended environment:
  ```
  | let ident '=' exp in exp
  {\env -> let v = $4 env
           in $6 (\i -> if i == $2
                         then v
                         else env i)}
  ```


A TXL Interpreter (4)

- A program gets evaluated by applying the overall result function to the empty environment:
  ```
  (\_ -> error "undefined variable")
  ```

See `HappyTXLInterpreter.y` for further details.