FROWN
An LALR(k) Parser Generator

RALF HINZE

Institute of Information and Computing Sciences
Utrecht University
Email: `ralf@cs.uu.nl`
Homepage: `http://www.cs.uu.nl/~ralf/`

September, 2001

(Pick the slides at `.../~ralf/talks.html#T30`.)

Joint work with Ross Paterson and Doaitse Swierstra.

Outline

- Usage
- Recap: LR parsing
- Implementation
- Features
- Facts and figures

Running example: well-formed parentheses

```
data Tree          = Node [Tree]
empty              = Node []
join t (Node us)   = Node (t : us)
%{
Terminal           = '(' | ')';
Nonterminal        = Expr{Tree};
Expr{join t u}     :  '(', Expr{t}, ')', Expr{u};
    {empty}        |  ;
}%
frown ts           = fail "syntax error"
```


Usage

Frown `:-(` is invoked as follows:

```
frown Paren.g
```

This generates a Haskell source file (*Paren.hs*) that contains (among other things) the desired parser:

```
expr :: (Monad m) ⇒ [Char] → m Tree
```

Here, *Char* is the type of terminals and *Tree* is the type of semantic values associated with *Expr*.

The standard example: arithmetic expressions

```
type Op              =  Int → Int → Int
%{
Terminal             =  Nat{Int} | Add{Op} | Mul{Op} | L | R;
Nonterminal          =  Expr{Int} | Term{Int} | Factor{Int};
Expr{v1 `op` v2}     :  Expr{v1}, Add{op}, Term{v2};
    {e}              |  Term{e};
Term{v1 `op` v2}     :  Term{v1}, Mul{op}, Factor{v2};
    {e}              |  Factor{e};
Factor{e}            :  L, Expr{e}, R;
      {n}            |  Nat{n};
}%
frown ts             =  fail ("syntax error: " ++ show ts)
```



```
data Terminal        =  Nat Int | Add Op | Mul Op | L | R
lexer                :: String → [Terminal]
lexer []             =  []
lexer ('+' : cs)     =  Add (+) : lexer cs
lexer ('-' : cs)     =  Add (−) : lexer cs
lexer ('*' : cs)     =  Mul (∗) : lexer cs
lexer ('/' : cs)     =  Mul div : lexer cs
lexer ('(' : cs)     =  L : lexer cs
lexer (')' : cs)     =  R : lexer cs
lexer (c : cs)
    | isDigit c      =  let (n, cs') = span isDigit cs
                        in Nat (read (c : n)) : lexer cs'
    | otherwise      =  lexer cs
```


Things to note

- The terminal symbols are arbitrary Haskell patterns (of the same *Terminal* type).
- Both terminal and nonterminal symbols may carry multiple semantic values (or no value).
- The parser generated for start symbol $Start\{T_1\}\ldots\{T_n\}$ has type

```
start :: (Monad m) ⇒ [Terminal] → m (T1, ..., Tn)
```

A grammar may have several start symbols.

Outline

- ✔ Usage
- ✘ Recap: LR parsing
- ✘ Implementation
- ✘ Features
- ✘ Facts and figures

Shift-reduce parsing

The parsers that are generated by Frown :-(are so-called LALR(k) parsers ('LA' ≅ lookahead, 'L' ≅ left-to-right scanning of input, 'R' ≅ constructing a rightmost derivation in reverse).

LR parsing is a general method of shift-reduce parsing.

General idea: reduce the input string to the start symbol of the grammar. Either *shift* a terminal symbol onto the stack or *reduce* the RHS of a production on top of the stack to the LHS.

History: LR parsers were first introduced by Knuth in 1965. When DeRemer devised the LALR method in 1969, the LR technique became the method of choice for parser generators.

Running example

The grammar is first augmented by an EOF symbol (here '’$’') and a new start symbol (here '*Start*').

$$
\begin{array}{lll}
\%\{ & & \\
\mathit{Terminal} & = & \texttt{'('} \mid \texttt{')'} \mid \texttt{'\$'}; \\
\mathit{Nonterminal} & = & \mathit{Start}\{\,\mathit{Tree}\,\} \mid \mathit{Expr}\{\,\mathit{Tree}\,\}; \\
[0]\ \ \mathit{Start}\{\,t\,\} & : & \mathit{Expr}\{\,t\,\}, \texttt{'\$'}; \\
[1]\ \ \mathit{Expr}\{\,\mathit{join}\ t\ u\,\} & : & \texttt{'('}, \mathit{Expr}\{\,t\,\}, \texttt{')'}, \mathit{Expr}\{\,u\,\}; \\
[2]\ \ \mathit{Expr}\{\,\mathit{empty}\,\} & : & ; \\
\}\% & &
\end{array}
$$

A non-deterministic parser

$$\textbf{data}\ \mathit{Stack} \ = \ \mathit{Stack} \succ \mathit{Symbol}$$

$$
\begin{array}{lll}
\mathit{parse} & :: & (\mathit{Monad}\ m) \Rightarrow \mathit{Stack} \rightarrow [\,\mathit{Terminal}\,] \rightarrow m\ \mathit{Tree} \\
\mathit{parse} & = & \mathit{shift} \mid \mathit{reduce}_0 \mid \mathit{reduce}_1 \mid \mathit{reduce}_2
\end{array}
$$

$$
\begin{array}{lll}
\mathit{shift}\ \mathit{st}\ (t : \mathit{tr}) & = & \mathit{parse}\ (\mathit{st} \succ t)\ \mathit{tr} \\
\mathit{reduce}_0\ (\mathit{st} \succ \mathit{Expr}\{\,t\,\} \succ \texttt{'\$'}) & = & \mathit{return}\ t \\
\mathit{reduce}_1\ (\mathit{st} \succ \texttt{'('} \succ \mathit{Expr}\{\,t\,\} \succ \texttt{')'} \succ \mathit{Expr}\{\,u\,\}) & & \\
 & = & \mathit{parse}\ (\mathit{st} \succ \mathit{Expr}\{\,\mathit{join}\ t\ u\,\}) \\
\mathit{reduce}_2\ \mathit{st} & = & \mathit{parse}\ (\mathit{st} \succ \mathit{Expr}\{\,\mathit{empty}\,\})
\end{array}
$$

A sample parse

		()()$
shift	(	)()$
reduce_2	(E	)()$
shift	(E)	()$
shift	(E)(	)$
reduce_2	(E)(E	)$
shift	(E)(E)	$
reduce_2	(E)(E)E	$
reduce_1	(E)E	$
reduce_1	(E)E	$
reduce_1	E	$
shift	$E$$	
reduce_0	S	

Recognition of handles

Problem: How can we decide which action to take? In particular, how can we efficiently determine which RHS resides on top of the stack?

☞ This is another language recognition problem!

A *handle* is the RHS of a production preceeded by a left context.

$$S \Longrightarrow_r^* \alpha N \omega \Longrightarrow_r \alpha\beta\omega \Longrightarrow_r^* \omega'$$

Here, $\alpha\beta$ is a handle of the *right-sentential form* $\alpha\beta\omega$ (ω and ω' contain only terminals).

Grammar of handles and left contexts

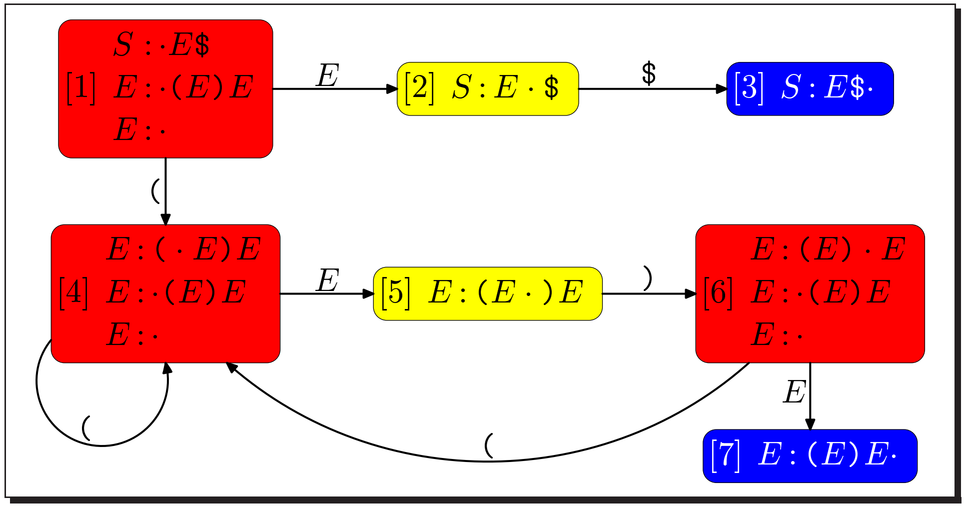
$\mathcal{H}(p)$ is the language of handles for production p; $\mathcal{L}(N)$ is the language of left contexts of N.

$$\begin{array}{lcl}
\mathcal{H}(0) & : & \mathcal{L}(Start), Expr, \texttt{'\$'}; \\
\mathcal{H}(1) & : & \mathcal{L}(Expr), \texttt{'('}, Expr, \texttt{')'}, Expr; \\
\mathcal{H}(2) & : & \mathcal{L}(Expr); \\
\mathcal{L}(Start) & : & ; \\
\mathcal{L}(Expr) & : & \mathcal{L}(Start); \\
 & | & \mathcal{L}(Expr), \texttt{'('}; \\
 & | & \mathcal{L}(Expr), \texttt{'('}, Expr, \texttt{')'};
\end{array}$$

☞ By construction, this grammar is left-recursive. Thus, $\mathcal{H}(p)$ and $\mathcal{L}(N)$ generate regular languages!

LR(0) automaton

Regular languages can be recognized by *deterministic finite state machines* (set of states construction; $N : \alpha \cdot \beta$ is called an *item*).



Outline

- ✔ Usage
- ✔ Recap: LR parsing
- ✘ Implementation
- ✘ Features
- ✘ Facts and figures

Towards Haskell

The LR(0) automaton can be directly encoded as a functional program.

The stack records the transitions of the LR(0) automaton.

$$\begin{array}{lll} \textbf{data}\ \mathit{GStack} & = & \mathit{VStack} \rightarrow \mathit{State} \\ \textbf{data}\ \mathit{VStack} & = & \mathit{GStack} \succ \mathit{Symbol} \end{array}$$

NB. $s \succ t \rightarrow s'$ is meant to resemble the transition $s \xrightarrow{t} s'$.

Shift and reduce states

State 2 is a *shift state*.

$$\mathit{parse}_2\ \mathit{st}\ (\texttt{'\$'} : \mathit{tr}) \quad = \quad \mathit{parse}_3\ (\mathit{st} \succ \texttt{'\$'} \rightarrow 3)\ \mathit{tr}$$

State 7 is a *reduce state*.

$$\begin{array}{l} \mathit{parse}_7\ (\mathit{st} \rightarrow 1 \succ \texttt{'('} \rightarrow 4 \succ E\{t\} \rightarrow 5 \succ \texttt{')'} \rightarrow 6 \succ E\{u\} \rightarrow 7) \\ \quad = \ \mathit{parse}_2\ (\mathit{st} \succ 1 \rightarrow E\{\mathit{join}\ t\ u\} \rightarrow 2) \\ \mathit{parse}_7\ (\mathit{st} \rightarrow 4 \succ \texttt{'('} \rightarrow 4 \succ E\{t\} \rightarrow 5 \succ \texttt{')'} \rightarrow 6 \succ E\{u\} \rightarrow 7) \\ \quad = \ \mathit{parse}_5\ (\mathit{st} \succ 4 \rightarrow E\{\mathit{join}\ t\ u\} \rightarrow 5) \\ \mathit{parse}_7\ (\mathit{st} \rightarrow 6 \succ \texttt{'('} \rightarrow 4 \succ E\{t\} \rightarrow 5 \succ \texttt{')'} \rightarrow 6 \succ E\{u\} \rightarrow 7) \\ \quad = \ \mathit{parse}_7\ (\mathit{st} \succ 6 \rightarrow E\{\mathit{join}\ t\ u\} \rightarrow 7) \end{array}$$

Conflicts

We have a *shift/reduce conflict* if a state contains both a shift and a reduce action (states 1, 4, and 6 in our running example).

☞ A shift/reduce conflict can possibly be resolved using *one* token of lookahead.

We have a *reduce/reduce conflict* if a state contains several reduce actions.

☞ A reduce/reduce conflict can possibly be resolved using k tokens of lookahead.

Computation of lookahead information

Idea: partially execute the LR(0) machine at compile time to determine the shifts that might follow a reduce action.

[1]	$E : 1\,\cdot$	$\{\$\}$
[3]	$S : 1\,E\,2\,\$\,3\,\cdot$	$\{\}$
[4]	$E : 4\,\cdot$	$\{)\}$
[6]	$E : 6\,\cdot$	$\{\$,)\}$
[7]	$E : 1\,(\,4\,E\,5\,)\,6\,E\,7\,\cdot$	$\{\$\}$
	$E : 4\,(\,4\,E\,5\,)\,6\,E\,7\,\cdot$	$\{)\}$
	$E : 6\,(\,4\,E\,5\,)\,6\,E\,7\,\cdot$	$\{\$,)\}$

NB. LALR(k) parsers merge the lookahead information for each production. In other words, the parsers generated by Frown :-(are slightly more general than LALR.

In Haskell: representation of the stack

For each transition we introduce a constructor.

```haskell
data Stack = Empty
           | St_1_4 Stack
           | St_1_2 Stack (Tree)
           | St_2_3 Stack
           | St_4_4 Stack
           | St_4_5 Stack (Tree)
           | St_5_6 Stack
           | St_6_4 Stack
           | St_6_7 Stack (Tree)
```


In Haskell: LR(0) machine

```haskell
expr tr                  = parse_1 tr Empty
parse_1 ts@[] st         = parse_2 ts (St_1_2 st (empty))
parse_1 ('(' : tr) st    = parse_4 tr (St_1_4 st)
parse_1 ts st            = frown ts
parse_2 tr@[] st         = parse_3 tr (St_2_3 st)
parse_2 ts st            = frown ts
parse_3 ts (St_2_3 (St_1_2 st (v0)))
                         = return (v0)
parse_4 ('(' : tr) st    = parse_4 tr (St_4_4 st)
parse_4 ts@(')' : tr) st = parse_5 ts (St_4_5 st (empty))
parse_4 ts st            = frown ts
```



```haskell
parse_5 (')' : tr) st = parse_6 tr (St_5_6 st)
parse_5 ts st         = frown ts
parse_6 ('(' : tr) st = parse_4 tr (St_6_4 st)
parse_6 ts st         = parse_7 ts (St_6_7 st (empty))
parse_7 ts (St_6_7 (St_5_6 (St_4_5 (St_1_4 st) (t))) (u))
                      = parse_2 ts (St_1_2 st (join t u))
parse_7 ts (St_6_7 (St_5_6 (St_4_5 (St_4_4 st) (t))) (u))
                      = parse_5 ts (St_4_5 st (join t u))
parse_7 ts (St_6_7 (St_5_6 (St_4_5 (St_6_4 st) (t))) (u))
                      = parse_7 ts (St_6_7 st (join t u))
```


Outline

- ✔ Usage
- ✔ Recap: LR parsing
- ✔ Implementation
- ✘ Features
- ✘ Facts and figures

Features

- Multiple entry points (start symbols).
- Multiple attribute values.
- Precedences to resolve conflicts.
- `--lookahead=`k: Use additional lookahead to resolve reduce/reduce conflicts (only used where needed).
- `--backtrack`: Produce a backtracking parser.

$$\mathit{start} \;::\; (\mathit{MonadPlus}\ m) \Rightarrow [\,\mathit{Terminal}\,] \rightarrow m\ (T_1, \ldots, T_n)$$

- `--monadic`: Monadic semantic actions.

- `--lexer`: Use monadic lexer instead of list of terminal symbols:

$$\begin{aligned} \mathit{get} &\;::\; (\mathit{Monad}\ m) \Rightarrow m\ \mathit{Terminal} \\ \mathit{start} &\;::\; (\mathit{Monad}\ m) \Rightarrow m\ (T_1, \ldots, T_n) \end{aligned}$$

- `--expected`: In case of error pass the set of expected tokens to the error routine.

$$\mathit{frown} \;::\; (\mathit{Monad}\ m) \Rightarrow [\,\mathit{Terminal}\,] \rightarrow [\,\mathit{Terminal}\,] \rightarrow m\ a$$

- Four different parser schemes (standard, LALR-like, stackless, combinator-based).

Outline

- ✔ Usage
- ✔ Recap: LR parsing
- ✔ Implementation
- ✔ Features
- ✘ Facts and figures

Facts and figures: expression parser

The expression grammar has 5 terminals, 4 nonterminals, and 7 productions. The LR automaton has 13 states and 23 transitions.

Running time (expr (`"a+b"` $+\!\!+$ concat ($\mathit{replicate}\ n$ `"*c+d"`))).

	1.000	10.000	100.000
`happy`	0.01	0.29	5.20
`happy -a`	0.02	0.40	5.92
`happy -c -g`	0.01	0.25	4.44
`happy -a -c -g`	0.01	0.23	3.85
`frown`	0.01	0.10	1.98
`frown -oc`	0.01	0.11	2.01
`frown -s`	0.01	0.10	2.03

Facts and figures: Haskell parser

The Haskell grammar has 61 terminals, 121 nonterminals, and 277 productions. The LR automaton has 490 states and 2961 transitions.

Compilation time.

	time	.hs	space	
happy		210K	150M	-fno-cpr
happy -c -g	4.5	246K	180M	-fno-cpr
happy -a -c -g	4.5	164K	100M	
frown -oc	4.8	300K	160M	
	.o	a.out	stripped	
happy	1.511K	2.536K	1.246K	
happy -c -g	1.093K	2.202K	1.134K	
happy -a -c -g	346K	1.711K	894K	
frown -oc	1.489K	2.547K	1.362K	

Running time.

	18K	300K	1653K
lex only	0.01	0.55	4.8
happy	0.23	2.68	12.1
happy -c -g	0.14	2.23	11.6
happy -a -c -g	0.07	1.83	10.6
frown -oc	0.14	1.95	8.5

Outline

- ✔ Usage
- ✔ Recap: LR parsing
- ✔ Implementation
- ✔ Features
- ✔ Facts and figures

Conclusion

- Encoding the LR(0) automaton as a set of mutually recursive functions rather than as a huge table gives more flexibility (use lookahead only where it is needed).
- Frown :-(generates good code (huge but fast). The output is fairly readable.
- With some additional work the generated parsers can produce good error messages (monadic lexer that keeps track of line numbers).